Practical General Top-down Parsers

ALI AFROOZEH ANASTASIA IZMAYLOVA

Practical General Top-down Parsers

ALI AFROOZEH   ANASTASIA IZMAYLOVA

Practical General Top-down Parsers

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K.I.J. Maex

ten overstaan van een door
het College voor Promoties ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op dinsdag 11 juni 2019, te 10.00 uur

door

Anastasia Izmaylova

geboren te Moskou

Promotiecommissie

Promotores:	prof. dr. P. Klint	Universiteit van Amsterdam
	prof. dr. J.J. Vinju	Technische Universiteit Eindhoven
Overige leden:	prof. dr. E. Van Wyk	University of Minnesota
	prof. dr. R. Lämmel	University of Koblenz-Landau
	prof. dr. K. Sima'an	Universiteit van Amsterdam
	prof. dr. M. de Rijke	Universiteit van Amsterdam
	prof. dr. R.V. van Nieuwpoort	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) under the auspices of the research school Institute for Programming research and Algorithmics (IPA) and has been supported by the NWO TOPGO grant #612.001.011 “Domain-Specific Languages: A Big Future for Small Programs”.

Typeset with LaTeX

The Iguana photo on the cover is licensed from iStock, credits: Jandrie Lombard. The sans serif font on the cover is League Gothic, designed by Caroline Hadilaksono, Micah Rich and Tyler Finck. The serif font on the cover is Cinzel, designed by Natanael Gama.

Practical General Top-down Parsers

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. ir. K.I.J. Maex

ten overstaan van een door
het College voor Promoties ingestelde commissie,
in het openbaar te verdedigen in de Agnietenkapel
op dinsdag 11 juni 2019, te 12.00 uur

door

Ali Afroozeh

geboren te Rafsanjan

Promotiecommissie

Promotores:	prof. dr. P. Klint	Universiteit van Amsterdam
	prof. dr. J.J. Vinju	Technische Universiteit Eindhoven
Overige leden:	prof. dr. E. Van Wyk	University of Minnesota
	prof. dr. R. Lämmel	University of Koblenz-Landau
	prof. dr. K. Sima'an	Universiteit van Amsterdam
	prof. dr. M. de Rijke	Universiteit van Amsterdam
	prof. dr. R.V. van Nieuwpoort	Universiteit van Amsterdam

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) under the auspices of the research school Institute for Programming research and Algorithmics (IPA) and has been supported by the NWO TOPGO grant #612.001.011 “Domain-Specific Languages: A Big Future for Small Programs”.

Typeset with LaTeX

The Iguana photo on the cover is licensed from iStock, credits: Jandrie Lombard. The sans serif font on the cover is League Gothic, designed by Caroline Hadilaksono, Micah Rich and Tyler Finck. The serif font on the cover is Cinzel, designed by Natanael Gama.

Посвящается моей любимой маме

تقدیم به مادر و پدر و برادران عزیزم

Contents

Acknowledgments

We would like to thank Paul Klint, our first promoter for his guidance and support throughout these years. Paul's idea to write a dual thesis was brilliant, not only it made our lives much easier, but also it kept the integrity of our research as a whole. We are also thankful to Jurgen Vinju, our second promoter, for the freedom he gave us to explore our ideas. Jurgen has always been enthusiastic and full of new ideas.

We would also like to thank Eric Van Wyk for discussions and feedback on our papers and this thesis. The discussions on LR parsing, context-aware scanning, and language composition have been invaluable for this thesis. Finally, we thank Ralf Lämmel, and other committee members for assessing our thesis.

Prologue

This thesis is the result of our almost five year research at Centrum Wiskunde & Informatica (CWI) in Amsterdam. Our journey has been adventurous, and certainly not very conventional. Written with care and utmost attention to details, this thesis concludes this bumpy journey, and it is fair to say that it is truly unique, both in terms of content and also the way it has been formed. When we started our PhD, we could not envision that our paths would merge and conclude like this, but as Søren Kierkegaard, Danish philosopher and theologian, says *"life can only be understood backwards"*. Now, while writing the last sentences of this thesis, we can look back to better understand this journey. One thing is for sure, the collaboration between the two of us has been truly unique, enjoyable and productive!

(Almost) every PhD student starts with big, ambitious dreams about changing the state of the art, striving at creating something beautiful. However, as we went further in the journey, we started to realize that there were simply too many hard constraints on the nature of our research. Hit by the harsh limitations of the reality, we had to be happy with only incremental improvements over the existing work. Indeed, we could not solve the problem of parsing, and parsing will remain a difficult topic, but we hope that our work can help the reader to better understand the trade-offs of different parsing techniques. Like in every engineering practice, in the end, there are all these trade-offs that matter.

There are many technical discussions in this thesis, and at the end we reflect on many technical lessons that we have learned in the course of our research. However, we like to start the thesis differently: we like to tell some other, non-technical stories from these years. We have recently watched a movie by Lars von Trier, a Danish director known for his controversial style, who is a brilliant story teller. In his latest movie *"The House That Jack Built"*, Jack, who is also an engineer, on the way to Hell, divides his life story into five randomly chosen incidents. Although our stories are neither that dark nor that violent as Jack's, rather intended to be of humorous nature, we still liked the idea and decided to tell the informal story of our research journey as five randomly chosen incidents that happened between 2013 and 2018.

Incident 1: The Ceiling

CWI was undoubtedly a great place to work. The new wing of the building, where our group resided, was modern, with large windows that let a lot of light in. Moreover, the semi-private, spacious offices shared by two or three PhD students were great. After leaving CWI and joining the companies where people work in an open office plan, we can now truly appreciate how great small offices are for productivity.

However, CWI almost killed us, literally. The story dates back to December 15, 2014. Those days we were both living in a student complex in Science park in Amsterdam, at a walking distance to CWI. We were usually coming back to CWI to work in the evenings. Moreover, Nikhef, the physics research institute next to CWI, had an in-house Nespresso coffee machine. Nikhef and its coffee machine were easily reachable via the corridors of the old wing of the CWI building, thus offering a nice opportunity for a walk and a cup of good, yet affordable, coffee for just 50 cents. The smell of the bricks of the old wing and the taste of the coffee are still in our memory.

That evening, while working on the Meerkat parser combinators, we decided to get another cup of coffee from Nikhef. On the way, when almost entering the old wing, we heard some weird sounds. We stopped for a moment, however, as there was no light on the other floors, we concluded that nobody was there, and joked that only the ghosts of science at CWI could be awake at such a late hour. On the way back, as we approached our offices, a split second after we passed the corridor, the whole internal, wooden part of the ceiling collapsed. We were very lucky that the ceiling did not fall on us.

The next day, Jurgen described the scene as a disaster site! It turned out that there was a construction problem with the way wooden parts were installed. It was only a few days later, December 19, 2014, that the notifications for the CC 2015 papers arrived, and our first paper together had been accepted.

Incident 2: The Ring

While working on our data-dependent parsing framework, we had chosen Onward! as the venue for our initial publication. Having already found a cool idea, we were looking for a cool title. After all, the conference was radical and ambitious, looking for *"grand visions and new paradigms that could make a big difference in how we will one day build software"*, and the title should have shown that we are ambitious. While browsing the papers from previous years for inspiration, one title caught our attention: *"One VM to Rule Them All"* from Oracle Labs. We decided to choose the title *"One Parser to Rule Them All"* as not only it was very cool, but also it described our data-dependent parsing approach very accurately — one can use data-dependent grammars to virtually parse anything.

The paper got accepted at Onward! 2015, and we went to Pittsburgh to present our work. After the presentation, during one of the conference social events, some people asked us if we were fans of *The Lord of the Rings* film series. That was a very strange question, we thought. Why should we be fans of *The Lord of the Rings*?! How could data-dependent parsing possibly relate to this series?! Much to our surprise, after some discussion aimed at resolving the common confusion, it became clear that our title was believed to be an adoption of the phrase *"One Ring to rule them all"*, known for being part of the lines inscribed on the magic ring, the central element of the film series.

Not being educated enough on common nerdy topics, for example, not being able to distinguish between Star Wars and Star Trek, we had absolutely no clue about the existence of such a phrase. Even to this date we have not watched *The Lord of the Rings* film series and do not know for sure what this phrase means.

Incident 3: The Casino

One of the most fun parts of doing a PhD is the opportunity to visit various conferences and meet new people. And two things we did on every conference trip were visiting a zoo and a steak house. In April 2015 we went to London to attend ETAPS 2015 and to present our CC paper. While searching in TripAdvisor for the best steak house in London, an interesting choice popped up: the Hippodrome Casino in Leicester square. The choice was made, and together with some newly met Dutch attendees from Eindhoven and Twente we headed towards the casino. While the steak was quite decent, though not living up to the raving reviews, the rest of the experience was something to remember.

From the dining table we could see the crowd playing around with the roulette machines. The idea came to our mind to give it a try. Other accompanying dinner members declared the act as too adventurous and decided to leave. Agreed to invest/lose no more than £15 per person, we exchanged the money for chips and went to exercise our luck. Having never been around a roulette machine before that night, we can now attest that gambling will definitely alter the functioning of the brain. Trying to guess the next number based on the sequence of previous numbers was very fun. If there have been eight reds in a row, betting on the black should be safe, right? It turned out that human beings are quite good at seeing patterns in random events, but dealing with a sequence of independent, equally probable events is not easy. Nonetheless, the luck was obviously on our side — starting with the small, least risky bets, we could not resist taking higher and higher risk for larger and larger gain, winning almost £200 in just about an hour.

While we started the night very good, a break to have a drink ruined the whole night. We quickly lost everything we won before the break, and after our additional losses reached £100 we decided to leave the casino. We will never know what would have happened if we had just continued without taking a break. However, we blamed the whole thing on the beer and the bad luck it brought. We have never visited a casino again after that night.

Incident 4: The Steak

While the previous incident happened in a steak house, it was not really about steak, this one is! One of the biggest dilemmas faced by hungry academics after a full day of attending some interesting (and many more boring) talks is where to eat. You can identify hungry academics by seeing long lines of people with laptop backpacks going from one restaurant to another, desperately trying to find a place that can accommodate such a large group of people without a special reservation. Being yet very inexperienced in attending conferences, we were also following the crowd, most often led by a professor from Delft. It was really not fun to follow the crowd!

Having become more experienced, we learned that the ideal size of a group for dinner is at most eight, and it is better to keep it a secret, because if more hungry academics hear that you are going to dine, they will follow you and ruin your experience. It was always a pleasure to hang out with Anya Helene Bagge during academic events. Not only she knew the best (bird) zoos, but also she was into fine dining. In Pittsburgh, we teamed up with Anya to find a good steak house. We decided to go to Capital Grille together with a small group of Swedes. The steak house was classy, and the steaks were excellent especially when paired with Californian red wine. Remarkably, while most attendees went for a moderate, according to the American standard, 600-800 gram rib eye steak, one of the Swedish attendees went for a 1600 gram porterhouse steak, and to everyone's astonishment finished the whole thing.

After eating a large portion of perfectly cooked medium-rare steak, it was time for dessert. To adhere to the best practices of academic conferences, and to honor the best steak session ever organized at an academic conference, by the time of the the dessert, at our large, round dining table, we had appointed seven session chairs: seating chair, wine chair, dessert chair, comfy chair, professor chair, beer chair, and most importantly keynote eater chair.

Incident 5: The Parrot

Amsterdam is a beautiful city. Ignoring the fact that Amsterdam attracts perhaps the worst kind of tourists in the world, it is one of the best places to live. The city center with all canals and beautiful 17th century houses is a pleasure to walk around. Amsterdam is also very green, with many parks. One of the most unusual things one can notice in Amsterdam is the presence of green parrots in the parks. The origin is unknown, but the myth says that some owners just set their parrot pets free, and they started to breed.

One afternoon, we were walking along the Amstel river discussing some aspects of our research. Heading back towards CWI, we saw a group of parrots flying next to the OLVG hospital. One of them was flying very low. A car passed by, and we could not see that parrot anymore. After the car fully passed, we saw a green thing lying in the middle of the road. It became clear that the car had hit the parrot. We ran and picked up the parrot from the street. The parrot's eyes were swirling, and it felt like these were the parrot's last moments.

Having an academic approach to things, and not knowing how things are done in the real world, we ran towards the emergency room in the hospital. When the guard saw the parrot, he swiftly jumped out of his chair and told us that we cannot bring a bird to a hospital. If they can treat one animal at the emergency, why not the other, we asked ourselves. The guard was nice enough to call the animal ambulance, but it turned out that they cannot just send an ambulance for one parrot! There are simply very few animal ambulances in Amsterdam. We were left sitting outside, holding the parrot in hands, sadly waiting for its death. Other patients were gathering around us. *"You are very good people that tried to save the parrot"*, said one elderly woman. *"I think he is dying"*, sadly concluded the other woman looking at the swirly eyes of the parrot.

In about half an hour after the tragic accident, the parrot started moving. Although the parrot's neck became straight again, and its eyes were not swirling anymore, it started to make very loud sounds. It was unconceivable how such a small creature can make such loud, unpleasant sounds. *"He is in pain"* stated the first elderly woman. Presumably hearing the sounds of the parrot, the guard appeared again, attempting to convince us to accept the reality and let it go. *"There are 10,000 such parrots living in Amsterdam"* pointed out the guard, apparently alluding to the fact that the death of one parrot is no big deal. Finally, he suggested us to go to the Oosterpark, located next to the hospital, put the parrot on the grass and let him die. This was the moment to accept the parrot's fate.

Hit by the harshness of reality, we started walking to the park, firmly holding the poor parrot in hands. By the moment of reaching the park, the parrot was quietly sitting in hands, awake, and actively looking around. The state of the bird suggested that there was no internal bleeding, yet the possibility of a broken wing seemed feasible. Searching for a safe place to release the bird, we crossed a bridge, and all of a sudden, the parrot started to make the same loud, unpleasant sounds again. Seconds after, the parrot took a hard bite of Ali's finger, and freed himself, leaving the lifesavers to deal with the blood dropping from the finger. After some academic analysis of the situation, we came to the conclusion that most likely the car did not hit the parrot, it was the parrot who hit the car and became unconscious. The whole event was very strange and kept us in shock for some time, but it did not have any long lasting effect on our ability to work on the thesis.

We assume some things in life just happen for no reason.

Chapter 1

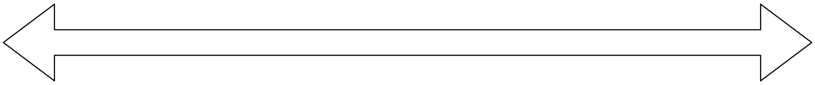
Introduction

Parsing is the process of analyzing a sequence of symbols and producing a hierarchical structure. Parsing is a common task in many applications, ranging from analyzing network data to processing natural languages. In this thesis we consider parsing in the context of programming languages, where a parser constructs a tree representation of source code according to the grammar of the language.

It is common to hear that parsing is a "solved" problem. Indeed, parsing is a well-understood subject in computer science. Knuth [54], Aho, Ullman [8] and many others laid out the theoretical foundation of parsing, and since the 70s we have efficient, linear-time parsing algorithms. The success of Yacc [44], and its ports to various programming languages, enabled language engineers to construct parsers from grammar definitions. However, grammars of most programming languages do not fit this elegant, classic parsing foundation, and still nowadays many parsers are written by hand as a set of mutually recursive functions.

When first parsing algorithms appeared, machine resources were scarce, and only linear-time parsing techniques were considered. The language engineer had to either use a tool such as Yacc, which accepts a restricted class of context-free grammars, or write the parser by hand. Since then, machines became more powerful, and the need for parsers in other areas such as source code analysis, language prototyping, and building Domain-Specific Languages (DSLs) arose. One of the main focuses of the research on parsing in the last decades has been on the development of more powerful parsing techniques that enable a more expressive, easy-to-use grammar specification, eliminating the need to manually write a parser.

Figure 1.1 shows the spectrum of parsing techniques for programming languages. On the left end of the spectrum, we consider the classic compiler construction field, where performance and good error messages are the most important factors. Most modern programming languages in use today have parsers that are generated using



Application:	Compiler Construction	DSLs, Language Prototyping
Focus:	Performance, Error Reporting	Expressiveness, Ease of Use
Runtime:	$O(n)$	$O(n^3)$
Examples:	LL (Recursive-Descent Parsing), LALR (Yacc)	GLR, GLL

Figure 1.1: The spectrum of parsing techniques for programming languages.

Bison (an open source variation of Yacc), or have handwritten recursive-descent parsers. Ruby[1], PHP[2], Perl[3], and OCaml[4] have Bison-based parsers, while Java[5], C#[6], Scala[7], and TypeScript[8] have recursive-descent parsers. GCC since version 3.4[9] has replaced its Yacc-derived C++ parser, and since version 4.1[10] its Bison-based C and Objective-C parsers, with handwritten recursive-descent parsers.

Grammars of programming languages in their natural form cannot be used out of the box for a recursive-descent implementation or by a tool such as Yacc. The language engineer has to massage the original grammar into a deterministic form, which can pose considerable difficulty, depending on the nature of the grammar and the parsing technology. In addition, often a carefully designed lexer, and various hacks in the interaction between the lexer and parser are required. This process is time-consuming and often requires deep knowledge of the inner workings of the parsing technology.

On the right end of the spectrum, we have applications of parsing that benefit from more freedom in syntax definition. Examples of such applications are domain-specific languages, language prototyping and reverse engineering. Using deterministic techniques for constructing parsers for these purposes can be difficult and time-consuming. In these applications, the language engineer's goal is to quickly construct a parser from the most natural version of the grammar that reflects the underlying language semantics, without the need to transform the grammar to conform to a more restricted class or to understand the underlying parsing algorithm. For these

[1] `https://github.com/ruby/ruby/blob/ruby_2_5/parse.y`

[2] `https://github.com/php/php-src/blob/PHP-7.2.3/Zend/zend_language_parser.y`

[3] `http://perldoc.perl.org/perl5100delta.html#New-parser`

[4] `https://github.com/ocaml/ocaml/blob/4.06/parsing/parser.mly`

[5] `http://hg.openjdk.java.net/jdk8/jdk8/langtools/file/1ff9d5118aae/src/share/classes/com/sun/tools/javac/parser/JavacParser.java`

[6] `https://github.com/dotnet/roslyn/blob/Visual-Studio-2017-Version-15.5/src/Compilers/CSharp/Portable/Parser/LanguageParser.cs`

[7] `https://github.com/scala/scala/blob/v2.12.4/src/compiler/scala/tools/nsc/ast/parser/Parsers.scala`

[8] `https://github.com/Microsoft/TypeScript/blob/v2.7.2/src/compiler/parser.ts`

[9] `https://gcc.gnu.org/gcc-3.4/changes.html#cplusplus`

[10] `https://gcc.gnu.org/gcc-4.1/changes.html`

applications, *general* parsing techniques that allow more flexibility and expressiveness in terms of syntax definition are considered.

General Parsing Techniques

To compare different parsing techniques along the spectrum of Figure 1.1, we need to discuss the concept of *non-determinism* in parsing. Parsing can be considered as a search problem. Given an input and a grammar, a parser has to find a sequence of grammar rules that generates the input. A deterministic parser has to make the right decision when there is more than one option, as it cannot backtrack to correct the mistake.

To write deterministic parsers, a language engineer needs to massage the grammar into the form accepted by the underlying parsing technique, for example, eliminate left recursion for recursive-descent parsing. In addition, deterministic parsers consider a fixed (often one) number of lookahead tokens when making a decision. As in most programming languages whitespace and comment are insignificant and can appear anywhere in the source program, the parser needs a way to bypass whitespace and comment to see the next significant token. To solve this problem, a separate lexical analysis (tokenization) phase is used before parsing. In particular, whitespace and comment are removed from the grammar during this phase, and the grammar is written as if no whitespace or comment exists.

Depending on the underlying parsing technology, rewriting a grammar to make it deterministic may be time-consuming. The separation of lexers and parsers is also not always straightforward. The main reason is that a tokenizer should decide about the type of a token without having access to the parsing context it appears in. Determining the type of a token during the tokenization phase is not always possible, and may require lexer hacks – feedback loop from the parser to the lexer. A famous example is the `>>` token which can represent a right shift operator or two closing generic type brackets, for example, `List<List<T>>` in Java.

Allowing more lookahead, or unlimited lookahead through backtracking, has been a common solution to increase the expressiveness of deterministic parsing techniques. Naive backtracking techniques, however, may lead to exponential runtime. General parsing algorithms [18,78,85] support all context-free grammars without any restriction, and have worst-case cubic runtime. In addition, as the full class of context-free grammars is closed under composition[11], using general parsing allows to write modular grammars out of the box [34]. This brings the best practices of software engineering such as reuse and modularity to developing grammars [50]. Moreover, as general parsing techniques effectively support unlimited lookahead, they can run on character-level grammars. This eliminates the need for a separate tokenizer and the limitations it entails.

[11] Other subclasses of context-free grammars, e.g., LR, are not closed under composition. There are, however, more restricted forms of grammar composition for subclasses of context-free grammars. For example, Schwerdfeger and Van Wyk showed how to statically verify composition of LR grammars. This technique is used to statically verify composition of grammars for language extensions [74].

The expressiveness and ease of use of general parsers comes at the cost of performance. General parsers have cubic worst case runtime, but such worst cases do not happen when used for real programming languages. General parsers, such as GLR [85] and GLL [78], are adaptive, i.e., they run linearly on the deterministic parts of the grammar [78]. Since real programming languages are deterministic in most parts, we can expect a near-linear performance on grammars of programming languages using general parsers. We note that general parsers tend to be slower than their deterministic counterparts on the deterministic parts of the grammar, mainly because the machinery of a general parser is more complicated and resource intensive. As we discuss in the rest of this chapter, the performance of general parsing is acceptable for the particular applications they are intended for. Another important consequence of using general parsing techniques is that the language engineer needs to explicitly deal with *ambiguity*. To deal with ambiguity, we follow a declarative view on syntax definition [34, 49].

Declarative Syntax Definition

Exploring all parsing paths of an input string may result in multiple parse trees. In a declarative syntax definition, the language engineer defines the grammar using its most natural form, free from any restriction imposed by the underlying parsing technique, and then specifies the desired parse trees using a set of disambiguation constructs [90].

In this thesis we use the grammar of OCaml [58] as a case study for discussing the operator precedence ambiguity, which is one of the most common and most difficult ambiguities to resolve. OCaml is a mixed-paradigm (functional and object-oriented), expression-based programming language that has a very large expression sublanguage. OCaml also features some unusual operator precedence rules that pose difficulties for non-LR parsing techniques[12]. A simplified excerpt of the OCaml grammar [58], written in its natural form, is given in Figure 1.2. As can be seen, the grammar is ambiguous, and has left and right recursive rules without any restriction. The OCaml language specification provides a precedence table, similar to the one shown in Figure 1.2 (right), that provides the rules for resolving precedence-related ambiguities. Figure 1.3 shows an example of declarative disambiguation constructs used to specify operator precedence rules for the grammar in Figure 1.2. In this example, `>` specifies the precedence relationship between operators, and `left` and `right` specify associativity.

Such declarative syntax definition schemes can be implemented in different ways. An obvious way would be to let the parser first produce all the parse trees, and then discard the ones that do not adhere to the disambiguation rules. Such an implementation, however, would be very slow and impractical. Another way is to change the underlying machinery of the parsing algorithm to apply the disambiguation constructs as early as possible. In this scheme, we apply the disambiguation constructs at parse time, and prune paths that will definitely lead to undesired parse trees.

Such natural grammars cannot be directly used in traditional deterministic parsing techniques. In particular, top-down parsers have difficulty in dealing with natural

[12] As we discuss in Chapter 5, the operator precedence ambiguities in OCaml map directly to shift-reduce conflicts in LR parsing.

```
expr ::= expr '.' field
       | expr expr
       | '-' expr
       | expr '*' expr
       | expr '+' expr
       | expr '-' expr
       | 'if' expr 'then' expr 'else' expr
       | expr ';' expr
       | '(' expr ')'
```

Operator	Associativity
.	–
function application	left
- (unary)	–
*	left
+ -	left
if	–
;	right

Figure 1.2: A simplified excerpt of the OCaml expression grammar (left), and its corresponding table of operator precedence (right).

```
expr ::= expr '.' field
       > expr expr                          left
       > '-' expr
       > expr '*' expr                      left
       > (expr '+' expr | expr '-' expr)    left
       > 'if' expr 'then' expr 'else' expr
       > expr ';' expr                      right
       | '(' expr ')'
```

Figure 1.3: A simplified excerpt of the OCaml grammar augmented with declarative operator precedence constructs. `left` and `right` specify left- and right-associativity, respectively, and `>` specifies the relative precedence of rules, according to the precedence table in Figure 1.2.

grammars because of lack of support for left recursion, and the automatic left-recursion removal technique leads to parse trees that are not left-associative. The parser for the OCaml compiler is written using `ocamlyacc`[13], a port of Yacc to OCaml. As Yacc supports declarative operator precedence disambiguation, constructing a parser for OCaml using Yacc is much easier than writing a recursive-descent parser. However, still, the Yacc grammar of OCaml is considerably larger than the natural reference one, and contains more nonterminals and rules.

In addition to natural parse trees that conform to the original grammar, using general parsing and explicit disambiguation rules provides the language engineer with more intuitive debugging and diagnosis options. The language engineer can always obtain all the parse trees, inspect ambiguities and fix the grammar by adding appropriate disambiguation rules. We believe such features make grammar debugging easier in comparison with deterministic and limited-backtracking parsing techniques. Deterministic LALR parsers report shift/reduce conflicts, which require knowledge of LALR machinery to understand and fix. Parsers with limited backtracking [9], better known as Parsing Expression Grammars (PEGs) [25], may report a parse error when

[13] `http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html`

the input can be accepted. This happens when the parser selects the wrong path when having multiple choices. In addition, PEGs may return a wrong parse tree when there is ambiguity, without reporting ambiguities to the user.

We note that by using a general parser and disambiguation rules we cannot guarantee that the underlying grammar is unambiguous. Applying a disambiguation rule wrongly may lead to a parse error, and not fully specifying disambiguation rules may leave some parts of the grammar ambiguous. To overcome this, the language engineer has to carefully test the parser with different inputs. We believe this lack of guarantee for being unambiguous is an acceptable trade-off when using general parsing algorithms. The language engineer starts with a fully ambiguous grammar and gradually applies disambiguation rules, and there is always the possibility to compare alternative parse trees to debug and fix the grammar.

The main question that we seek to answer in this thesis is how to build a correct and efficient parser directly from a natural grammar specification without resorting to manual grammar transformations that change the natural shape of the grammar rules, and as a result, influence the shape of the resulting parse trees.

The most widely used general parsing algorithm for programming languages is GLR. Notable tools based on GLR are ASF+SDF [88], Elkhound [61], and DMS® Software Re-engineering Toolkit [11]. GLR is a correct and mature parsing algorithm, however, as it operates on LR automata, it is hard to understand and modify. It is also more difficult to produce good error messages from GLR parsers. In 2010 Scott and Johnstone introduced the Generalized LL (GLL) parsing algorithm [78] that provides a viable alternative to GLR. GLL parsers are attractive as they are recursive-descent like and their runtime has a close relationship with the grammar. This makes GLL parsing easier to understand and modify.

A significant part of our thesis is dedicated to improvements and extension of the GLL parsing algorithm. In the rest of this chapter we first give a historical overview of the parsing techniques that led to the development of GLL, and then discuss generalization of recursive-descent parsing with focus on supporting left recursion. Then, we discuss our disambiguation framework that can deal with many challenges of parsing programming languages. Our disambiguation framework is based on data-dependent grammars [41] and is implemented on top of GLL. We also discuss embedding of context-free grammars as a set of combinators directly in a programming language. Our general parser combinators are based on Johnson's CPS recognizers [43], which, beside GLL, have been another main influencer of our work. Finally, we give an overview of our research questions and outline the rest of the chapters.

1.1 Evolution of the GLL Parsing Algorithm

In this section we give a historical overview of the parsing algorithms that influenced GLL, and discuss how GLL relates to other parsing techniques in a global picture. This discussion positions our work and provides necessary background for the rest of this thesis.

Research in parsing has a long history, and various parsing techniques were developed in parallel in different communities. Therefore, it is not surprising to see many similarities between parsing algorithms. In fact, it is sometimes difficult to distinguish one parsing algorithm from the other. Grune and Jacobs in their book "Parsing Techniques" [33] observe that existing parsing algorithms are very similar: *"Basically almost all parsing is done by top-down search with left-recursion protection; this is true even for traditional bottom-up techniques like LR(1), where the top-down search is built into the LR(1) parse tables."* GLL is particularly interesting in this respect. It is classified as a generalization of recursive-descent parsing, but from the terminology and the way it is formulated it appears to be very similar to Earley's algorithm [18] and even GLR [85].

Figure 1.4 shows the relationship between parsing algorithms that influenced GLL, and how they relate to our work. This comparison is inspired by a similar diagram in Scott and Johnstone's RNGLR paper [76], where they discuss the evolution of GLR parsers. At the bottom of Figure 1.4 we have the traditional LR [54] and recursive-descent parsing. As can be seen, most of the work are related to generalizing LR parsing, presumably because LR parsing supports left-recursive grammars, and compared to LL, accepts a larger subclass of context-free grammars. In the rest of this section, we first discuss the influence of LR parsing on GLL, and then discuss the relationship between recursive-descent parsing and GLL.

LR parsing was introduced by Knuth in his seminal paper in 1965. Two decades later, Tomita introduced Generalized LR (GLR) for parsing natural languages [85]. GLR splits the LR parsing stack at each conflict state, and uses a Graph-Structured Stack (GSS) to share the same segments of the stack. Tomita also introduced the Shared Packed Parse Forest (SPPF) format for representing all derivation trees of a sentence.

The original GLR algorithm by Tomita is correct when the grammar is ϵ-free[14], and fails on grammars with hidden left recursion [76]. A significant amount of research on LR parsing in the last decades is related to fixes and improvements to Tomita's algorithm. The two most important fixes are by Farshi [67], and Scott and Johnstone [76]. Another improvement to GLR is by Rekers [71], who shows how to construct more compact SPPFs from Farshi's GLR recognizers. Reker's SPPF structure has been used in the ASF+SDF Meta-Environment [89].

Although most of the development of GLR is attributed to Tomita, we should also mention Lang's theoretical framework for non-deterministic parsing [56], where he shows how to develop generalized versions of deterministic parsers. Billot and Lang later extended this theoretical framework to construct shared packed parse forests [12]. Tomita's GLR can be considered as a concrete realization of Lang's framework.

Construction of a Tomita-style SPPF leads to $O(n^{k+1})$ worst-case complexity, where k is the length of the longest grammar rule. This fact was first presented by Johnson in 1991 [42]. To guarantee the cubic worst-case runtime performance, which is common in general recognizers[15], the length of the grammar rules should be at most

[14] A grammar is ϵ-free if it does not contain rules that produce empty string (denoted by ϵ).

[15] A recognizer only says if an input string is accepted, without producing parse trees.

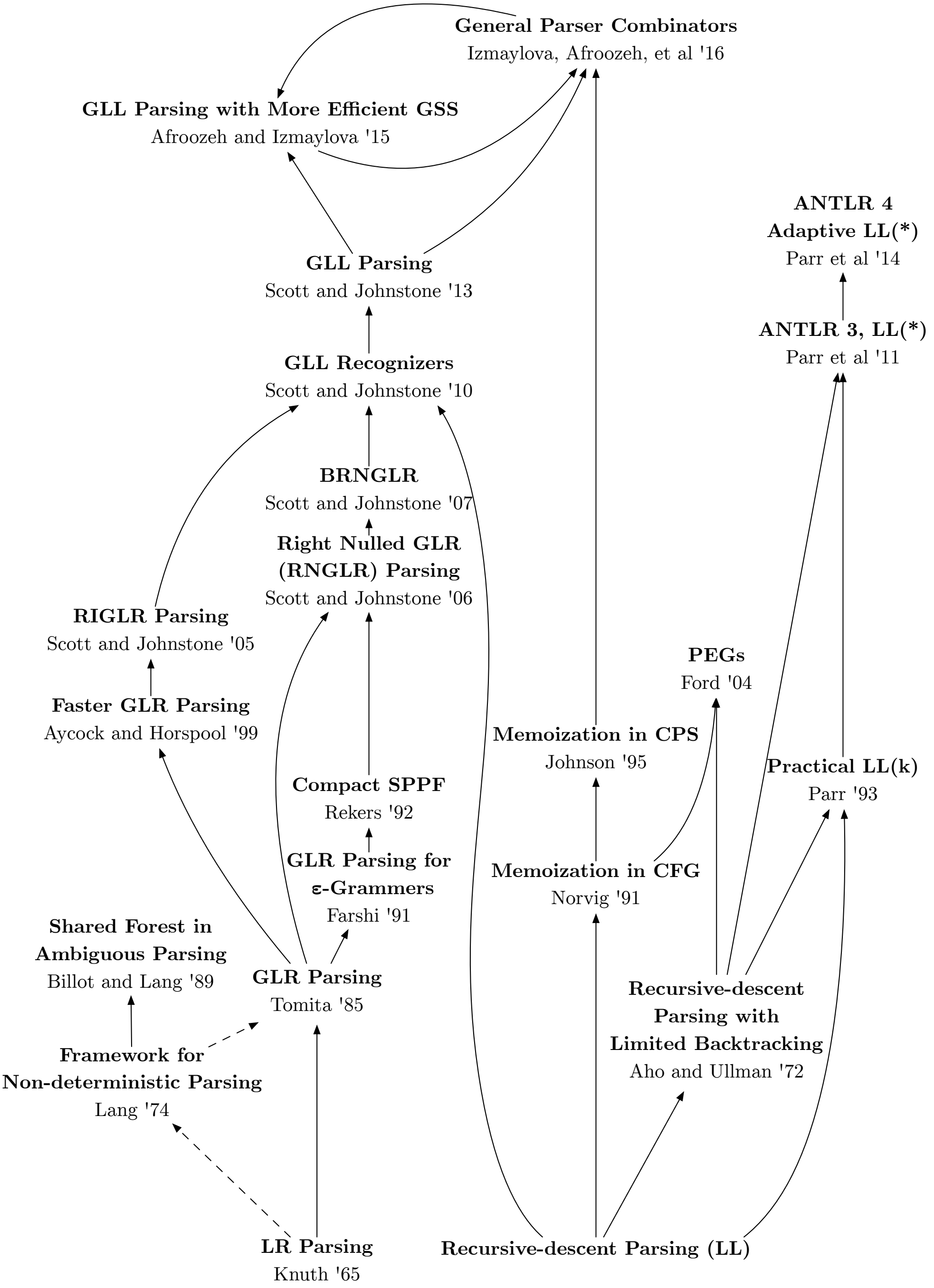


Figure 1.4: Evolution of the GLL parsing algorithm.

two. Scott and Johnstone introduced the notion of binarized SPPFs [80], which add additional nodes to the SPPF to effectively simulate grammars of length at most two. Binarized SPPFs enable general parsing in cubic time and space.

A parallel development in GLR parsing is the introduction of faster GLR parsers by Aycock and Horspool [10]. They replaced the traditional LR automata with larger automata that trades space for speed, leading to faster GLR parsing. Aycock and Horspool's algorithm does not support grammars with hidden left recursion. Scott and Johnstone gave a version of Aycock and Horspool's algorithm that accepts all context-free grammars [75]. They reported that while working on Aycock and Horspool's algorithm, they realized that this algorithm is closer to a generalization of LL than LR. This was the insight that led to the development of the GLL parsing algorithm [78].

A question that may come to one's mind is that if GLR is a correct and efficient algorithm, why there is a need for a new, generalized LL parser. The answer is that GLR parsers, due to their bottom-up nature, have a complex execution model, and therefore, are more difficult to understand. As a consequence, it is also more difficult to modify and extend GLR to implement disambiguation rules. Recursive-descent parsers, on the other hand, have a very intuitive execution model and have been extensively used to build parsers for programming languages. Unfortunately, recursive-descent parsers do not terminate on left-recursive grammars, and can have worst-case exponential runtime if implemented naively. Therefore, it is desirable to have a general parsing algorithm that is recursive-descent like, can natively support left recursion, and can provide an acceptable runtime bound for the worst case.

GLL is a generalization of recursive-descent parsing that supports all context-free grammars and provides a cubic worst-case runtime bound. At the core of GLL is the Graph Structured Stack (GSS), which effectively represents all calls as a graph. In GLL, GSS nodes can have cyclic edges, which allows to handle left recursion. As GLL parsers have a recursive-descent control flow and maintain a close relationship with the underlying grammar, they are easy to understand and modify. It is possible to write a GLL parser by hand, although the task is usually automated by a parser generator, and it is also possible to debug the parser with a programming language IDE. These characteristics make GLL attractive for both developing parsers and also for implementing disambiguation rules.

Although recursive-descent parsing is very intuitive and easy to understand, a direct generalization of recursive-descent parsing proved to be very difficult. Norvig in 1991 showed that by memoization we can expect polynomial worst-case runtime in recursive-descent parsing [66]. This result is not surprising, as memoization is usually used in dynamic programming techniques [15] to reduce the exponential runtime to polynomial.

Figure 1.4 shows a number of extensions of recursive-descent parsing that all use a combination of memoization and limited backtracking techniques to improve the expressiveness of LL parsing. These approaches, however, do not provide a solution to the problem of left recursion in top-down parsing. ANTLR 4 [70], the most recent work in this category, supports almost all context-free grammars, with the exception of indirect left-recursive ones. ANTLR 4 rewrites the grammar to eliminate left recursion before parsing, and thus does not natively support left recursion at runtime. In addition, ANTLR 4 does not construct a parse forest, and at most returns one parse tree.

The problem of left recursion needs a complete rethinking of control flow in recursive-descent parsing. To the best of our knowledge, the first elegant solution to the problem of left recursion is presented by Johnson in 1995 [43]. He showed that by formulating recursive-descent parsers in Continuation-Passing Style (CPS) and memoizing the results, it is possible to support left recursion in top-down parsing. Johnson's approach is formulated in a functional style, as a set of combinators. However, his approach did not gain much attention, especially in the compiler construction community. Our experience with both GLL and Johnson's CPS recognizers shows that the way they deal with left recursion is so similar that they can be considered the same algorithm. In fact, we consider Johnson's algorithm as an implementation of GLL in a functional style (see Chapter 7).

Our contributions to general top-down parsing in this thesis are influenced by both GLL and Johnson's CPS recognizers. Based on our work on Johnson's recognizers, we suggested a modification to the GSS structure[16] in the original GLL that led to considerable performance improvement. Moreover, our work on GLL parsing resulted in an extension to the memoization strategy in Johnson's recognizers that brings its worst-case complexity to cubic. We also show how to construct binarized SPPFs from cubic CPS recognizers in cubic time and space.

1.2 Context-free Grammars and Recursive-descent Parsing

We give now an overview of context-free grammars and parsing that is needed for the rest of this thesis. For a more extensive treatment of these topics, see [8,9,33]. A *language* is a set of *sentences*, where each sentence is a sequence of symbols. It is common to use a *grammar* to define the language. A grammar is a set of rules that dictate how sentences of a language can be generated. In *context-free* grammars, rules are of the form $A ::= \alpha$, where A (head) is a nonterminal, and α (body) is a possibly empty sequence of *terminal* and *nonterminal* symbols. The empty sequence is written as ϵ. Grammar rules with the same head are usually grouped as $A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, where each α_i is an *alternative* of A. As an example, consider the following grammar for basic arithmetic operations:

$$
\begin{aligned}
E ::=\ & E + E && (1)\\
\mid\ & E - E && (2)\\
\mid\ & E * E && (3)\\
\mid\ & E \,/\, E && (4)\\
\mid\ & \mathit{Digit} && (5)\\
\mathit{Digit} ::=\ & 0 \mid 1 \mid \ldots \mid 9 && (6)
\end{aligned}
$$

[16] The idea of changing the structure of GSS, by moving labels from GSS nodes to GSS edges, was first proposed by Alex ten Brink, while he was working on his Master's thesis at Eindhoven University of Technology.

$$\begin{aligned}
E &\overset{(1)}{\Rightarrow} E + E \\
&\overset{(5)}{\Rightarrow} \mathit{Digit} + E \\
&\overset{(6)}{\Rightarrow} 1 + E \\
&\overset{(3)}{\Rightarrow} 1 + E * E \\
&\overset{(5)}{\Rightarrow} 1 + \mathit{Digit} * E \\
&\overset{(6)}{\Rightarrow} 1 + 2 * E \\
&\overset{(5)}{\Rightarrow} 1 + 2 * \mathit{Digit} \\
&\overset{(6)}{\Rightarrow} 1 + 2 * 3
\end{aligned}$$

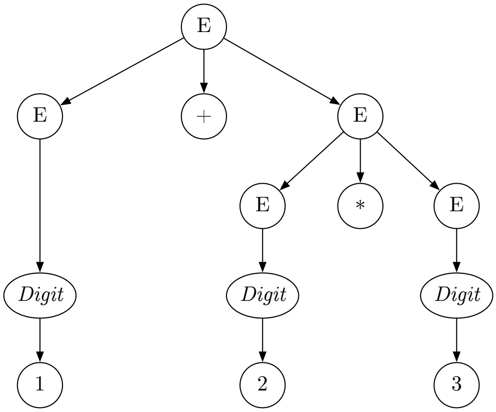


Figure 1.5: A left-most derivation of the sentence $1 + 2 * 3$ (left) and its corresponding derivation tree (right).

Using this grammar, we can generate the sentence $1 + 2 * 3$ as shown in Figure 1.5 (left). The $\Rightarrow$ sign denotes a *derivation step*, in which a nonterminal is replaced by one of its alternatives. In Figure 1.5 the number on each derivation step corresponds to the rule being applied. A *derivation* is a sequence of derivation steps. The derivation above is *left-most*, as at each step, the left-most nonterminal in the body of a rule is rewritten. We can visualize a derivation using a derivation tree. The derivation tree for the above derivation is shown in Figure 1.5 (right).

The problem of *parsing* is how to build a derivation tree for a given sentence, according to a grammar. Top-down parsing, which is the focus of this thesis, starts from the start nonterminal and tries to mimic a left-most derivation process that can generate the sentence. As an example, consider the following simple grammar that generates a list of a's, and the input string aaa.

$$\begin{aligned}
A &::= aA \\
&\mid a
\end{aligned}$$

A top-down parser starts from the start nonterminal, in this case A, and expands the alternatives until it matches the sentence. The recognition process of a top-down parser for this grammar is shown in Figure 1.6. Each step shows an expansion of A to one of its alternatives. The paths that are still expanding are shown with dashed arrows. Steps 1-3 expand the first alternative, $A ::= aA$, and there is a match for terminal a. At step 4, no more a can be matched, therefore, the parser goes back to the previous step and tries the other alternative of A, i.e., $A ::= a$, which also fails. At step 6, the parser again goes one step back and tries the other alternative. As there is a match, and there is no more symbols left in the input, the parser reports success.

Top-down parsers are also called *LL* parsers, where the first L stands for left to right scan of the input, and the second L for creating a left-most derivation. When

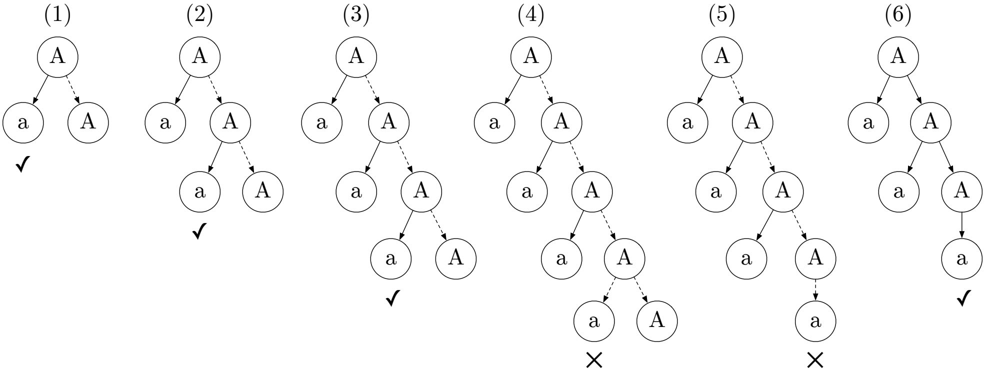


Figure 1.6: A top-down parser recognition steps for the grammar $A ::= aA \mid a$ and the input aaa.

selecting the alternatives of a nonterminal, a top-down parser has two choices. In *backtracking* parsers, the alternatives are tried in order, while in *predictive* parsers, the parser selects the right alternative. The grammars that admit predictive parsers are called $LL(k)$, where k is the number of symbols (lookahead) to look into the input when selecting an alternative. $LL(k)$ is a small class of context-free grammars, and many syntactic constructs in programming languages cannot be readily defined by $LL(k)$ grammars.

Top-down parsers can be implemented in either a table-driven or recursive-descent way. In table-driven top-down parsers, the grammar is represented as a tabular data structure, and the parser is a simple stack-based program that interprets the table against the input. A more common way to implement top-down parsers is to directly write the parser in a programming language as a set of (mutually recursive) functions, one for each nonterminal. A call to the nonterminal function first selects one of its alternatives, and then processes the symbols in the alternative in sequence. For a terminal symbol, the current input position is matched against the terminal, while for a nonterminal, the corresponding function for the nonterminal is called. Recursive-descent parsers are easy to write, understand and modify, and can be debugged using the programming language's development environment. Recursive-descent parsers also provide easy error reporting and recovery facilities.

The main shortcoming of recursive-descent parsing is lack of support for left recursion. Expressions in their natural form are often left-recursive. In fact, in order to obtain left-associative operators we need left-recursive rules. Left recursion can be automatically eliminated through the following grammar rewriting procedure [8]. Let A be a left-recursive nonterminal of the form:

$$A ::= A\alpha_1 \mid A\alpha_2 \mid ... \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid ... \mid \beta_m$$

Where β's do not start with A. We can eliminate left recursion, by introducing a new nonterminal A' and rewriting the grammar as follows:

$$\begin{aligned} A &::= \beta_1 A' \mid \beta_2 A' \mid ... \mid \beta_m A' \\ A' &::= \alpha_1 A' \mid \alpha_2 A' \mid ... \mid \alpha_n A' \mid \epsilon \end{aligned}$$

As can be seen, this rewriting strategy replaces left recursion with right recursion through the newly introduced nonterminal A'. Indirect left recursion [33], where $A \Rightarrow B\alpha \overset{*}{\Rightarrow} A\beta\alpha$, and hidden, direct or indirect, left recursion [33], where $A \overset{*}{\Rightarrow} \gamma A \mu$ and $\gamma \overset{*}{\Rightarrow} \epsilon$, can also be automatically rewritten, albeit through a more complicated process.

The main problem with automatic left-recursion elimination is that the resulting parse trees are loosely related to the ones of the original, natural grammar, i.e., the resulting trees no longer have the left-associative tree structure. In fact, many grammars with large expression grammars do not opt for automatic left-recursion removal. They are either parsed using bottom-up techniques that support left recursion, or have embedded operator precedence parsers [23] for their expression part. For example, the `javac` parser[17] of OpenJDK 8, uses an operator precedence scheme for parsing binary expressions as part of its recursive-descent parser.

1.3 Left Recursion in Recursive-descent Parsing

A straightforward implementation of a recursive-descent parser for a left-recursive grammar leads to non-termination, while left-recursion elimination techniques disfigure the grammar and lead to parse trees that are loosely related to the intended ones. Therefore, it is desirable to natively support left recursion in recursive-descent parsing.

In this section we discuss how left recursion can be supported in recursive-descent parsing, by explaining two working solutions, i.e., Generalized LL (GLL) and Johnson's CPS recognizers. For this discussion, we use a modified version of GLL parsing that uses a more efficient GSS [3]. Moreover, we only consider the recognizer version (without constructing parse trees) of these algorithms, and use the following left-recursive grammar as the running example.

$$\begin{aligned} E ::= &\ E + E \\ \mid &\ a \end{aligned}$$

1.3.1 GLL with Modified GSS

To overcome nontermination in face of left recursion, GLL replaces the programming language call stack with a GSS, which is a directed graph that represents all the calls during parsing. A GLL parser's execution is described in terms of actions for each *grammar slot*. A grammar slot is a position in the body of a rule, similar to an LR item [8], and is denoted by a dot, e.g., $A ::= \alpha \cdot \beta$.

[17] https://hg.openjdk.java.net/jdk8/jdk8/langtools/file/c8a87a58eb3e/src/share/classes/com/sun/tools/javac/parser/JavacParser.java

Instead of direct function calls for nonterminals, as it is in recursive-descent parsing, GLL performs actions on the GSS. A GSS node represents a call, recording the name of a nonterminal and the input index at which the call is made. GSS edges connect nodes that represent consecutive calls. A GSS edge is labeled with a grammar slot that corresponds to the return position from which parsing should be resumed, each time the call to the current GSS node produced a successful result.

The unit of work in GLL parsing is a *descriptor*. A descriptor captures a parsing state, i.e., the current grammar slot, the current input position, and the current GSS node. In GLL, the control flow is implemented in a trampoline-style loop [28], as opposed to direct function calls in recursive-descent parsing. In a trampoline-style control flow, execution is serialized into discrete units of work which are processed in a loop. In GLL, at each nondeterministic point during execution, a descriptor is added to the set of descriptors. This set can be implemented as a combination of a stack and a hash table. The stack implementation provides a recursive-descent like execution order of descriptors. The hash table is used to store all descriptors created during parsing, to avoid adding duplicate ones to the stack. At each iteration of the main loop of GLL, a descriptor is selected and processed. Parsing terminates when all descriptors are processed.

To explain how GLL deals with left recursion, we study the execution trace of a GLL parser for the running example grammar and the input string $a + a$. The execution trace is shown in Figures 1.7 and 1.8. Each step of the execution trace is separated into two parts. The left part shows the current grammar slot and the action that will be performed at this step. The result of this action is reflected on the GSS (the current GSS node is highlighted), the input index and the set of descriptors. We denote descriptors as tuples (L, u, i), where L is a grammar slot, u is a GSS node of the form (A, j), and i is the input index. Between each two steps, the parser returns to the main loop and removes a descriptor, which will be processed in the next step. The descriptors that are already processed are shown in gray in the set of descriptors.

Parsing starts by a call to the start nonterminal E at input position 0. As no GSS node corresponding to this call exists, a GSS node $(E, 0)$ is created. Then, the parser adds two descriptors $(E ::= \cdot E + E, (E, 0), 0)$ and $(E ::= \cdot a, (E, 0), 0)$, which correspond to the alternatives of E.

The first descriptor is processed at step 2. As the parser is before a nonterminal, i.e., $E ::= \cdot E + E$, a call to E at input position 0 is made. This step, which corresponds to calling a nonterminal, is called *create*. The parser first tries to find an existing GSS node with the given nonterminal name and input index, i.e., $(E, 0)$. As this GSS node exists, the parser does not add descriptors for its alternatives again. Instead, it just adds an edge labeled $E ::= E \cdot +E$ from the current node to itself. As a result, at this step, the left-recursive call is effectively terminated.

After terminating the left recursion, the parser processes the pending descriptor $(E ::= \cdot a, (E, 0), 0)$ at step 3. As the parser is before a terminal, it tries to *match* the terminal against the input string at the current input position. As there is a match, the parser moves to the grammar slot $E ::= a \cdot$, and increments the input index. Note that matching a terminal in GLL does not add any descriptors as the result of matching a terminal is always deterministic.

At step 4 the parser is at the end of the grammar rule $E ::= a\cdot$, and performs a *pop* action. Pop corresponds to returning from a function in recursive-descent parsing. Pop first examines if the current input index is already in the result set of the GSS node, shown as $\{\}$ next to GSS nodes. If this is the case, pop simply returns to the main loop. Otherwise, the input index is added to the result set, and for each outgoing edge from the GSS node, a new descriptor is added using the grammar slot of the edge. This effectively allows to continue parsing from the grammar slot of each outgoing edge with the new input position.

As the result of step 4, 1 is added to the result set of $(E, 0)$, and the descriptor $(E ::= E \cdot +E, (E, 0), 1)$ is added. This produces the first result for the left recursive call of E at input position 0. As can be seen, a left-recursive call is terminated on the second call to a left-recursive nonterminal at the same input position. Then, if other non-recursive alternatives of the nonterminal yield a result, parsing continues after the left-recursive call.

Another left-recursive call happens at step 6, where the parser is at grammar slot $E ::= E + \cdot E$ and input position 2. This call creates the GSS node $(E, 2)$ and adds two descriptors $(E ::= \cdot E + E, (E, 2), 2)$ and $(E, E ::= \cdot a, (E, 2), 2)$. Processing the first descriptor at step 7 adds a loop on the GSS node $(E, 2)$, which terminates this left-recursive call. Then, the pop action at step 9 adds 3 to the result set of $(E, 2)$ and adds two descriptors $(E ::= E \cdot +E, (E, 2), 3)$ and $(E ::= E + E\cdot, (E, 0), 3)$. Processing the first descriptor leads to a match action that fails. Processing the second descriptor adds 3 to the result set of $(E, 0)$. This is the first successful parse result, as the call to E at input position 0 has yielded a result which is equal to the length of the input. This pop action also adds the descriptor $(E ::= E \cdot +E, (E, 0), 3)$, which leads to a failed match action. Parsing terminates at this point as all the descriptors are processed.

Johnson's CPS Recognizers

Johnson's approach to deal with left recursion requires a formulation of recursive-descent parsing in Continuation-Passing Style (CPS), instead of the more common formulation in which a function returns a list of successes [96]. In CPS, a function does not directly return its result, rather it calls a continuation with its result. Continuations represent the rest of the computation and are passed as an extra argument to functions.

Similar to the execution trace of GLL, we use the grammar $E ::= E + E \mid a$ and the input $a + a$ to explain the execution of Johnson's CPS recognizers. The execution trace is illustrated in Figure 1.9. In this execution trace, we use function names that correspond to nonterminals, terminals and grammar slots. For example, f_E and f_a denote parsers for nonterminal E and terminal a, respectively.

The recognition process starts at step 1 by making the call $f_E(0, \kappa)$. The continuation κ, passed to the start call, prints success if the input index is equal to the length of the input. In Johnson's CPS recognizers, functions such as f_E are memoized, i.e., $f_E = \mathit{memo}(f_{E+E|a})$, where *memo* is a higher-order function that takes a function and returns its memoized version, and $f_{E+E|a}$ is the parser for the alternatives of E.

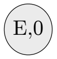
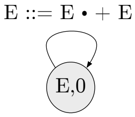
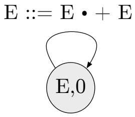
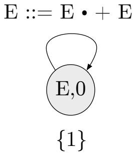
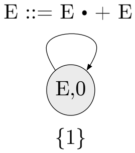
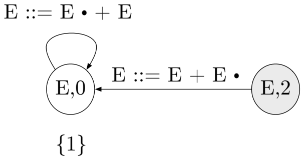
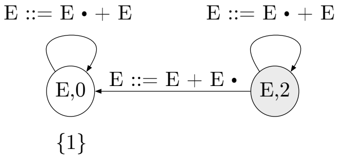
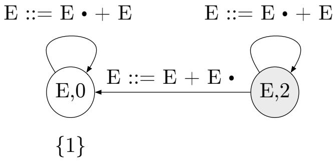
#	Action	Current Slot	Input	GSS	Set of Descriptors
1	start	E	a + a ↑		(E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
2	create	E ::= •E + E	a + a ↑		(E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
3	match (success)	E ::= • a	a + a ↑		(E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
4	pop	E ::= a •	a + a ↑		(E ::= E • + E, (E, 0), 1) (E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
5	match (success)	E ::= E • + E	a + a ↑		(E ::= E • + E, (E, 0), 1) (E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
6	create	E ::= E + •E	a + a ↑		(E ::= •E + E, (E, 2), 2) (E ::= •a, (E, 2), 2) (E ::= E • + E, (E, 0), 1) (E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
7	create	E ::= •E + E	a + a ↑	 E ::= E • + E E ::= E • + E E ::= E + E • E,0 E,2 {1}	(E ::= •E + E, (E, 2), 2) (E ::= •a, (E, 2), 2) (E ::= E • + E, (E, 0), 1) (E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
8	match (success)	E :: = •a	a + a ↑	 E ::= E • + E E ::= E • + E E ::= E + E • E,0 E,2 {1}	(E ::= •E + E, (E, 2), 2) (E ::= •a, (E, 2), 2) (E ::= E • + E, (E, 0), 1) (E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)

Figure 1.7: GLL execution trace for grammar $E ::= E + E \mid a$ and input `a+a` (part 1).

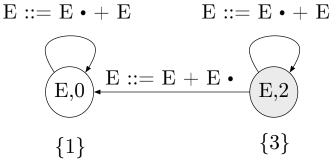
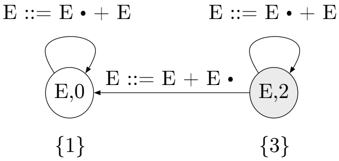
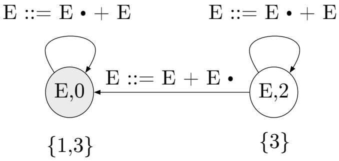
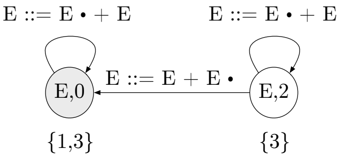
#	Action	Current Slot	Input	GSS	Set of Descriptors
9	pop	E :: = a •	a + a ↑	 E ::= E • + E E ::= E • + E E ::= E + E • E,0 E,2 {1} {3}	(E ::= E • + E, (E, 2), 3) (E ::= E + E •, (E, 0), 3) (E ::= •E + E, (E, 2), 2) (E ::= •a, (E, 2), 2) (E ::= E • + E, (E, 0), 1) (E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
10	match (fail)	E ::= E • + E	a + a ↑	 E ::= E • + E E ::= E • + E E ::= E + E • E,0 E,2 {1} {3}	(E ::= E • + E, (E, 2), 3) (E ::= •E + E, (E, 2), 2) (E ::= E + E •, (E, 0), 3) (E ::= •a, (E, 2), 2) (E ::= E • + E, (E, 0), 1) (E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
11	pop (parse success)	E ::= E + E •	a + a ↑	 E ::= E • + E E ::= E • + E E ::= E + E • E,0 E,2 {1,3} {3}	(E ::= E • + E, (E, 0), 3) (E ::= E + E •, (E, 0), 3) (E ::= E • + E, (E, 2), 3) (E ::= •E + E, (E, 2), 2) (E ::= •a, (E, 2), 2) (E ::= E • + E, (E, 0), 1) (E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)
12	match (fail)	E ::= E • + E	a + a ↑	 E ::= E • + E E ::= E • + E E ::= E + E • E,0 E,2 {1,3} {3}	(E ::= E • + E, (E, 0), 3) (E ::= E + E •, (E, 0), 3) (E ::= E • + E, (E, 2), 3) (E ::= •E + E, (E, 2), 2) (E ::= •a, (E, 2), 2) (E ::= E • + E, (E, 0), 1) (E ::= •E + E, (E, 0), 0) (E ::= •a, (E, 0), 0)

Figure 1.8: GLL execution trace for grammar $E ::= E + E \mid a$ and input `a+a` (part 2).

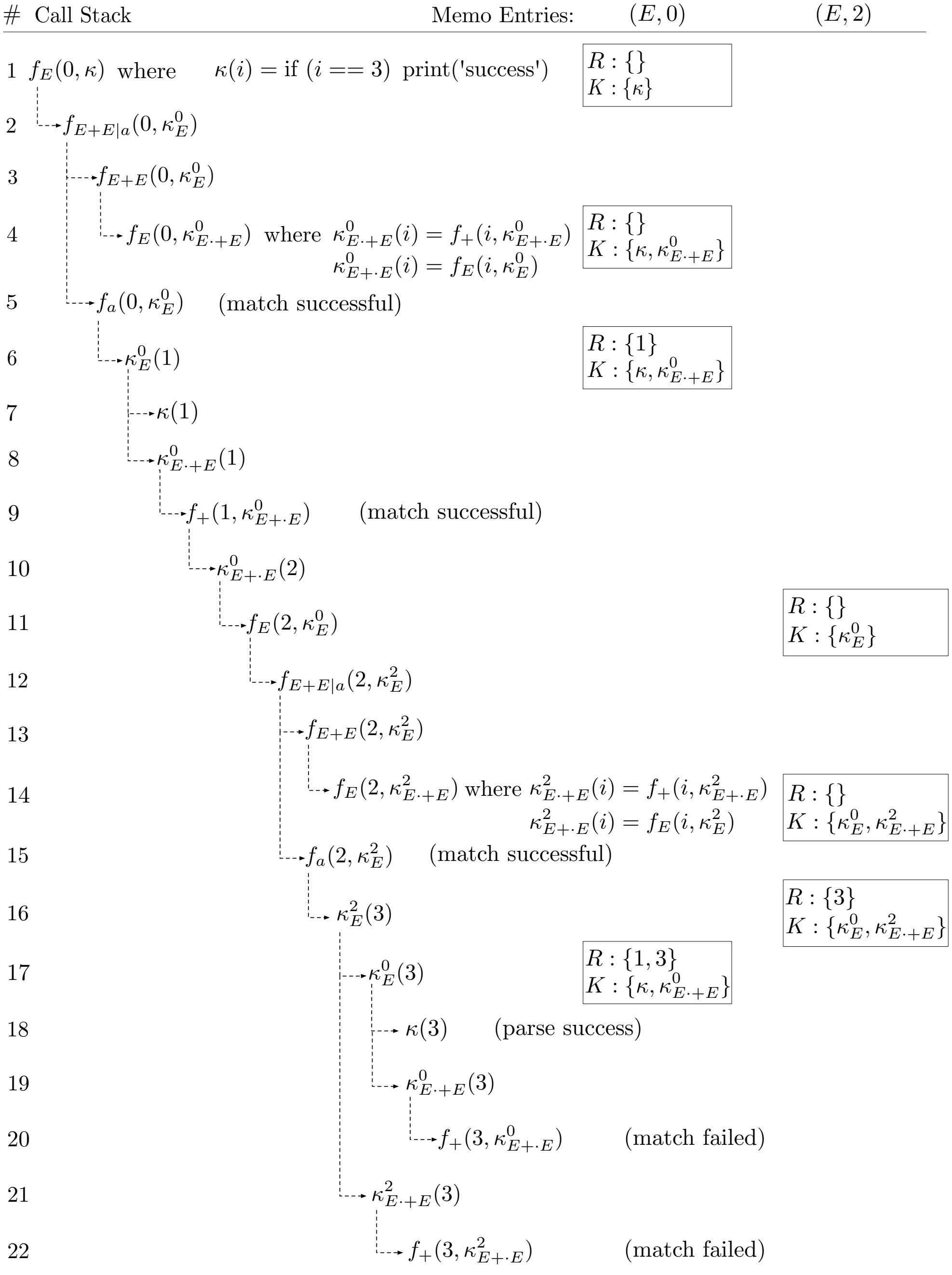
As $f_E(0, \kappa)$ is the first call to f_E at input position 0, a new memo entry, denoted by $(E, 0)$, is created, and then the function $f_{E+E|a}$ is called at input position 0 with the continuation κ_E^0 (step 2). Memo entries in Johnson's CPS recognizers consist of two sets: R for storing the results of the call, and K for storing the continuations that were passed to this call. In the execution trace, we show the state of the memo entries when they change. The continuation κ_E^0 is responsible for storing the successful parse results in the memo entry $(E, 0)$, and running all the pending continuations of the memo entry for the new results.

The call $f_{E+E|a}(0, \kappa_E^0)$ spawns two calls, each corresponding to an alternative of E. The first call (at step 3), which corresponds to the alternative $E ::= E + E$, first composes a chain of continuations, i.e., $\kappa_{E\cdot+E}^0$ and $\kappa_{E+\cdot E}^0$, for each symbol in the body of the alternative (see the execution trace for the definitions of these continuations), and then calls the parser for the first symbol, f_E, with $\kappa_{E\cdot+E}^0$. At step 4, as the memo entry $(E, 0)$ exists, the call $f_E(0, \kappa_{E\cdot+E}^0)$ just adds the continuation $\kappa_{E\cdot+E}^0$ to the memo entry's list of continuations and returns. This effectively terminates the left-recursive call.

At step 5, the second alternative of E is called at input position 0, i.e., $f_a(0, \kappa_E^0)$. As the next character in the input is a, the match succeeds, and the call $\kappa_E^0(1)$ is made (at step 6). This call stores the first result of parsing E into the memo entry $(E, 0)$, and calls all the pending continuations, namely κ and $\kappa_{E\cdot+E}^0$, at input position 1. As 1 is not the end of the input, the call $\kappa(1)$ returns without printing success. The other call, $\kappa_{E\cdot+E}^0(1)$ at step 8, continues parsing from the grammar slot $E ::= E \cdot +E$.

Another left-recursive call, $f_E(2, \kappa_E^0)$, happens at step 11. As this is the first call to E at input position 2, the memo entry $(E, 2)$ is created, the continuation κ_E^0 is stored, and then the call $f_{E+E|a}(2, \kappa_E^2)$ is made. Similarly to step 3, the left-recursive call is terminated (at step 14), and is continued when the first result from the non-left-recursive call $f_a(2, \kappa_E^2)$ is produced at step 15. At step 16, $\kappa_E^2(3)$ adds 3 to the memo entry $(E, 2)$, and then calls the continuations κ_E^0 and $\kappa_{E\cdot+E}^2$ at input position 3. The first call adds 3 to the results of memo entry $(E, 0)$, and calls $\kappa(3)$ which prints out success. The other call fails, as there is no more + left in the input to be matched.

As can be seen, the mechanism for dealing with left recursion in Johnson's CPS recognizers is basically the same as in GLL. Calls to a left-recursive nonterminal are terminated upon the second call at the same input position. Then, when any non-left-recursive alternative yields a result, the pending continuations that were passed to the left-recursive nonterminal are called with the new result. This effectively allows to continue parsing from the grammar position that is after the terminated left-recursive call. Although in Johnson's CPS recognizers the control flow is encoded in continuation-passing style, the basic mechanism to deal with left recursion is the same as in GLL.



# Call Stack
Memo Entries:
$(E, 0)$
$(E, 2)$
1 $f_E(0, \kappa)$ where $\kappa(i) =$ if $(i == 3)$ print('success')
$R: \{\}$ $K: \{\kappa\}$
2 $f_{E+E|a}(0, \kappa_E^0)$
3 $f_{E+E}(0, \kappa_E^0)$
4 $f_E(0, \kappa_{E\cdot+E}^0)$ where $\kappa_{E\cdot+E}^0(i) = f_+(i, \kappa_{E+\cdot E}^0)$
$\kappa_{E+\cdot E}^0(i) = f_E(i, \kappa_E^0)$
$R: \{\}$ $K: \{\kappa, \kappa_{E\cdot+E}^0\}$
5 $f_a(0, \kappa_E^0)$ (match successful)
6 $\kappa_E^0(1)$
$R: \{1\}$ $K: \{\kappa, \kappa_{E\cdot+E}^0\}$
7 $\kappa(1)$
8 $\kappa_{E\cdot+E}^0(1)$
9 $f_+(1, \kappa_{E+\cdot E}^0)$ (match successful)
10 $\kappa_{E+\cdot E}^0(2)$
11 $f_E(2, \kappa_E^0)$
$R: \{\}$ $K: \{\kappa_E^0\}$
12 $f_{E+E|a}(2, \kappa_E^2)$
13 $f_{E+E}(2, \kappa_E^2)$
14 $f_E(2, \kappa_{E\cdot+E}^2)$ where $\kappa_{E\cdot+E}^2(i) = f_+(i, \kappa_{E+\cdot E}^2)$
$\kappa_{E+\cdot E}^2(i) = f_E(i, \kappa_E^2)$
$R: \{\}$ $K: \{\kappa_E^0, \kappa_{E\cdot+E}^2\}$
15 $f_a(2, \kappa_E^2)$ (match successful)
16 $\kappa_E^2(3)$
$R: \{3\}$ $K: \{\kappa_E^0, \kappa_{E\cdot+E}^2\}$
17 $\kappa_E^0(3)$
$R: \{1, 3\}$ $K: \{\kappa, \kappa_{E\cdot+E}^0\}$
18 $\kappa(3)$ (parse success)
19 $\kappa_{E\cdot+E}^0(3)$
20 $f_+(3, \kappa_{E+\cdot E}^0)$ (match failed)
21 $\kappa_{E\cdot+E}^2(3)$
22 $f_+(3, \kappa_{E+\cdot E}^2)$ (match failed)

Figure 1.9: Execution trace of Johnson's CPS recognizers for the grammar $E ::= E + E \mid a$, and the input string a+a.

1.4 A Generic Framework for Disambiguation

General parsing algorithms, such as GLL, allow the language engineer to use the full class of context-free grammars to define the syntax. The freedom in syntax definition, however, comes at the price of ambiguity. A sentence that can be derived in multiple ways is called ambiguous. In programming languages, as opposed to natural language processing, almost always one parse tree is desired. Disambiguation is the process of selecting the desired parse tree among the set of parse trees.

In the context of general parsing, there are three common ways that can be used to deal with ambiguity:

- Manually rewrite the grammar to an unambiguous one that accepts the same language.
- Let the parser to produce all parse trees in the form of a parse forest, and then discard the undesired ones, for example, by matching against illegal parse tree patterns.
- Modify the underlying parsing technique to apply the disambiguation rules at runtime, by terminating parsing paths that lead to illegal parse trees.

Rewriting a grammar to remove ambiguity can be difficult, especially for grammars with complicated operator precedence rules. Post-parse filtering of a parse forest is a general, brute-force approach that in theory always works. However, producing all parse trees, especially when there are large number of ambiguities, can be very slow. Moreover, pattern matching on the parse forest to remove undesired parse trees can be complicated and costly.

In general parsing techniques it is common to use explicit, declarative runtime disambiguation, where the language engineer augments the context-free rules with a set of disambiguation constructs [34,49,90,95]. These constructs define which parse trees are illegal. At runtime, the parser applies the disambiguation rules and terminates the parsing paths that will lead to illegal parse trees as soon as possible. In declarative disambiguation, only the specified ambiguities are resolved. This is in contrast to implicit disambiguation techniques that are commonly used in many non-general (often deterministic) parsing techniques. Tools such as Yacc and ANTLR 4 [70], have explicit declarative disambiguation constructs for operator precedence, but implicitly resolve all the remaining ambiguities to be able to produce a single parse tree. It should be noted that Yacc generates warnings when automatically resolving the conflicts, which should be inspected by the user to avoid unexpected behavior.

Compared to grammar rewriting, declarative disambiguation makes the task of writing grammars faster and easier. The resulting grammars are also usually smaller and more readable. Compared to post-parse filtering, declarative disambiguation can lead to improvement in the performance, especially when there is a significant number of ambiguities. Compared to implicit runtime disambiguation, explicit and declarative disambiguation is more predictable.

Implementing disambiguation constructs at parser runtime requires extensive knowledge of the inner workings of the underlying parsing algorithm. As a result,

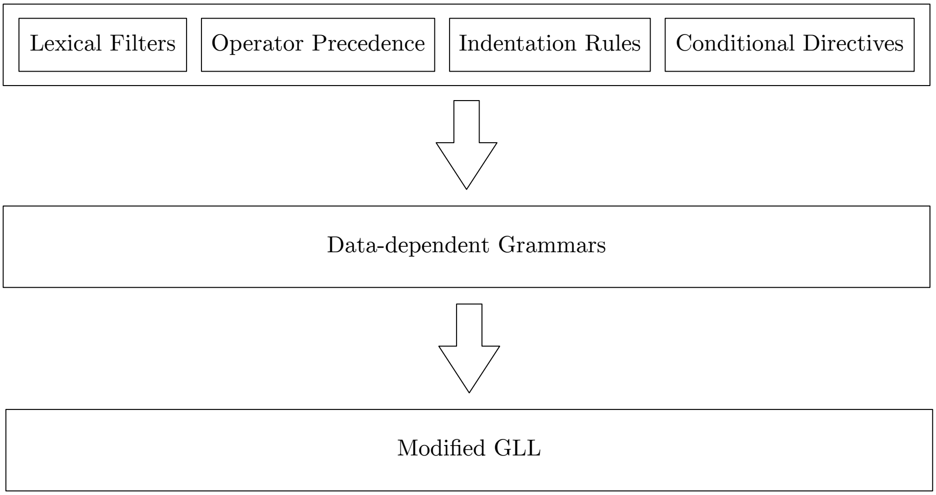


Figure 1.10: Our framework for implementing disambiguation constructs based on data-dependent grammars.

correct and efficient implementation of disambiguation constructs is not trivial. For each disambiguation construct and parsing algorithm there is a custom implementation based on the underlying parsing algorithm. For example, the same disambiguation construct can be implemented in GLR by modifying parse tables, and in GLL by modifying GSS nodes to carry additional information. This makes implementation of different disambiguation constructs difficult and time-consuming.

In this thesis we propose a framework for implementing different disambiguation constructs in a parser-independent way. To be able to deal with a wide range of ambiguities, including ambiguities in context-sensitive languages, e.g., Haskell and Python (indentation-sensitivity), and the `typedef` ambiguity in C, we base our framework on *data-dependent* grammars [41]. Data-dependent grammars extend context-free grammars with features such as arbitrary expressions, variable binding, parameters, and constraints. These features, which are well-known in general-purpose programming languages, bring more expressive power to context-free grammars, and make it possible to express many different disambiguation constructs without the need to modify the underlying parsing machinery.

Figure 1.10 shows a high-level view of our framework for implementing disambiguation constructs. At the top layer, we have common disambiguation constructs such as lexical filters, operator precedence, indentation rules and conditional directives. Users of our framework can modify the existing disambiguation constructs or define new ones. Instead of directly implementing these constructs in the context of a specific parsing algorithm, we desugar them to data-dependent grammars. In this view, data-dependent grammars act as an intermediate language for implementing various disambiguation constructs. Data-dependent grammars can be implemented on top of various parsing algorithms. In Chapter 3 we show an implementation of data-dependent grammars on top of GLL parsing.

A significant part of this thesis is dedicated to the semantics and implementation of disambiguation rules for operator precedence. Disambiguation rules for operator precedence are among the most important disambiguation constructs, and perhaps the most difficult to get right. For example, the declarative syntax definition tool SDF2 [95], which is based on the Scannerless GLR parsing algorithm [71,94], fails to disambiguate unconventional operator precedence rules in functional programming languages such as OCaml.

Ambiguities related to operator precedence have a special property that we call *safety*. A disambiguation mechanism is safe if it does not remove sentences from the language. In other words, if there is no ambiguity, a safe disambiguation mechanism does not apply. In Chapters 4 and 5, we introduce the notion of safe operator precedence disambiguation and provide a semantics for operator precedence that is safe and can deal with operator precedence cases that occur in programming languages such as OCaml. We provide an efficient implementation of the safe operator precedence semantics using data-dependent grammars in Chapter 5.

Our disambiguation framework can also deal with context-sensitive ambiguities, e.g., indentation rules in Python and Haskell. Parsers for indentation-sensitive languages often require custom modifications in the lexer (and the parser). Using data-dependent grammars we can provide a solution to parsing indentation-sensitive languages, without requiring a custom lexer implementation. As data-dependent grammars allow passing arbitrary values during parsing, indentation information can be carried and enforced in nested blocks. We provide high-level declarative constructs that define the indentation rules at the grammar level, which are then desugared to data-dependent grammars. This provides a concise and readable declarative syntax specification for indentation-sensitive languages. We discuss indentation-sensitivity and our solution in detail in Chapter 3.

1.5 Direct Embedding of Context-free Grammars

In traditional compiler construction tools, such as Yacc, grammars are defined using an external syntax, which is then transformed to an executable form, e.g., parse tables. Context-free grammars can also be directly embedded in a programming language as a set of combinators. This approach is popular in functional programming languages such as Haskell. In this approach, it is common to define parsers as functions, and then compose them using higher-order functions (combinators) such as sequence and alternation. Such embedding of context-free grammars is referred to as *shallow* embedding.

Combinator-style parsing is popular because parsers are first-class citizens of the programming language. This provides great flexibility, for example, it is a common practice in combinator-style parsing to pass values through the chain of functions to deal with context-sensitive constructs found in network protocols and indentation-sensitive languages.

```
trait Symbol {
  def ~(other: Symbol) = Seq(this, other)
  def |(other: Symbol) = Alt(this, other)
}

case class Seq(first: Symbol, second: Symbol) extends Symbol

case class Alt(first: Symbol, second: Symbol) extends Symbol

case class Nonterminal(name: String) extends Symbol {
  def ::=(alts: Alt) = Rule(this, alts)
}

case class Terminal(name: String) extends Symbol

case class Rule(head: Nonterminal, body: Alt)

implicit def s2t(s: String): Terminal = Terminal(s)
```

Figure 1.11: A deep embedding of context-free grammars in Scala.

It is also common to embed context-free grammars in a programming language as a data type. Such an embedding is called a *deep* embedding. Compared to a shallow embedding, a deep embedding can be more efficient as it is possible to preprocess the data type and apply various optimizations.

Scala [68] is a modern functional programming language that has a flexible syntax. This makes Scala particularly suitable for writing grammar embeddings. As an example, consider the grammar:

$$S ::= aSbS \mid aS \mid s$$

which can be directly encoded in Scala as:

```
S ::= "a" ~ S ~ "b" ~ S | "a" ~ S | "s"
```

As can be seen, this encoding very closely resembles the context-free grammar it represents. Scala provides various features that facilitate the development of embedded DSLs. For example, parentheses and dot can be omitted for method calls when there is only one argument, and special characters, such as ~ and `::=`, can be used as method names. In addition, Scala supports implicit compile-time type conversions.

Figure 1.11 shows a data type definition for deep embedding of context-free grammars. The trait[18] `Symbol` defines two methods ~ and `|` for sequence and alternation, respectively. The `Seq`, `Alt`, `Nonterminal` and `Terminal` case classes[19] define concrete symbols. The `Rule` case class defines a grammar rule, with a nonterminal head and its body. The method `::=` on the nonterminal type gets an alternative and creates a

[18] A trait in Scala is similar to a type constructor in Haskell or an interface in Java.

[19] A case class in Scala is similar to a value constructor in Haskell or a concrete, final class in Java.

```
type Result[T] = Option[T]

trait Parser extends ((String, Int) => Result[Int]) {

  def ~(p: => Parser): Parser =
    (input, i) => this(input, i).flatMap(j => p(input, j))

  def |(p: => Parser): Parser =
    (input, i) => this(input, i).orElse(p(input, i))
}

implicit def s2t(s: String): Parser =
    (input, i) => if (matches(input, i, s)) Some(i + s.length) else None

// Matches the input string from position j against the given string s
private def matches(input: String, j: Int, s: String): Boolean
```

Figure 1.12: A shallow embedding of context-free grammars in Scala.

rule with the current nonterminal as head. Finally, the implicit method `s2t` converts strings to terminals.

Given the definitions of Figure 1.11, the grammar above, with implicit conversions made explicit, and parentheses and dots for method calls put in place, is equivalent to:

```
val S = Nonterminal("S")
S.::=(s2t("a").~(S).~(s2t("b")).~(S)).|(s2t("a").~(S)).|(s2t("s"))
```

The result of executing this expression is the following instance of `Rule`:

```
Rule(Nonterminal("S"),
     Alt(Alt(Seq(Seq(Seq(Terminal("a"),
                         Nonterminal("S")),
                     Terminal("b")),
                 Nonterminal("S")),
             Seq(Terminal("a"),
                 Nonterminal("S"))),
         Terminal("s")
        )
    )
```

Figure 1.12 defines a shallow embedding of context-free grammars in Scala. Using these definitions, the grammar above can be written as:

```
val S : Parser = "a" ~ S ~ "b" ~ S | "a" ~ S | "s"
```

Although the result looks almost the same as the one for deep embedding, the underlying mechanism is different. In this version, `S` is a function, and thus is directly executable. For example, `S("aasbs", 0)` is a function call with the input string `"aasbs"` and input position `0`, which returns `Some(5)`.

In Figure 1.12, line 1 defines the type `Result[T]` which is an alias to Scala's `Option[T]` type. `Option` in Scala represents an optional value, similar to `Maybe` in Haskell or `Option` in ML. We use this type to represent the result of parsing, i.e., `Some(i)` when the parser succeeds with new input position `i` or `None` when the parser fails. The parser type is defined as a function from the input string and input position to parse result:

```
(String, Int) => Result[Int]
```

Scala, compared to other functional programming languages such as Haskell, does not support infix notation for function calls with two arguments. To simulate the infix notation, we define the trait `Parser` as a subtype of the `(String, Int) => Result[Int]` type, and define the sequence (~) and alternation (`|`) combinators as methods on the `Parser` trait (lines 5–9). The implementation of these methods are based on the `flatMap` and `orElse` methods available on Scala's `Option` type.

The sequence combinator returns a parser that, given an input string (`input`) and an input position (`i`), calls the first parser (`this`) at `i`, and if it succeeds, calls the second parser (`p`) with the result of the first parser. If the first parser fails, no call to the second parser is made, and the entire sequence fails. The alternation combinator returns a parser that, given an input string (`input`) and an input position (`i`), calls the first parser (`this`) at `i`, and only if it fails, calls the second parser (`p`) with the same input position. This effectively implements a simple backtracking scheme.

The implicit conversion at line 12 converts a string to a terminal parser that matches the given string against the input. It is easy to extend the terminal parser to use regular expressions. This flexibility of parser combinators allows the user to easily develop custom parsers using the features of the host programming language.

A straightforward shallow embedding of context-free grammars, as shown in Figure 1.12, fails on grammars with left recursion. Lack of support for left recursion has been a major limitation of traditional parser combinators [36,37,57]. An important requirement for building parser combinators as a shallow embedding is that the underlying parsing machinery has to be composable using the sequence and alternation operators. Unfortunately, the machinery of general parsing algorithms, such as GLR and Earley, is not composable using sequence and alternation, and hence, they cannot be used to build parser combinators as a shallow embedding. In Chapter 7 we show how to extend the definitions of Figure 1.12 to support left recursion and produce a binarized SPPF in cubic time and space. Our general parser combinators are based on Johnson's CPS recognizers [43] and are implemented in Scala.

1.6 Research Questions and Overview of Chapters

In this section we discuss the research questions, and give an overview of the remaining chapters.

1.6.1 Research Questions

Research Question 1. GLL is a relatively new general parsing algorithm that has not been yet widely used in practice. Can we make GLL faster, and build efficient general parsers based on GLL?

The answer to this question is presented in Chapters 2 and 6. In Chapter 2 we present a modification to the Graph Structured Stack (GSS), an important internal data structure of GLL, that leads to significant performance improvement. The results of our extensive performance evaluation in Chapter 6 show that GLL parsers can be practical for parsing real programming languages.

Research Question 2. Using general parsing algorithms for parsing programming languages goes hand in hand with (declarative) disambiguation. To build practical general parsers, it is essential to support disambiguation. Disambiguation constructs are typically implemented in the context of a specific parsing algorithm. Is it possible to implement various disambiguation constructs without the knowledge of the underlying parsing technique?

The answer to this question is presented in Chapter 3. We use data-dependent grammars as an intermediate language for implementing various disambiguation constructs. We extend GLL to support data-dependent grammars, and show how to define high-level disambiguation constructs that desugar to data-dependent grammars. We discuss the application of our technique to resolve various ambiguities such as ambiguities in indentation-sensitive languages, e.g., Haskell and Python, conditional directives in C#, `typedef` ambiguity in C, and operator precedence.

Research Question 3. How can we deal with intricate cases of operator precedence ambiguity that are present in functional programming languages such as OCaml?

The answer to this question is presented in Chapter 4. We introduce a derivation-based semantics for operator precedence disambiguation that is independent of the underlying parsing technique, and is safe, i.e., does not remove sentences from the language when there is no ambiguity, and can deal with operator precedence cases that occur in functional programming languages such as OCaml.

Our safe specification of operator precedence rules is implemented by an automatic grammar rewriting process that preserves the shape of the parse trees, conforming to the original ambiguous grammar. This rewriting, however, could lead to very large grammars, which affects the runtime of the parser.

Research Question 4. How can we implement our safe operator precedence technique in a way that does not require a grammar transformation that increases the size of the grammar, and is independent of the underlying parsing algorithm?

The answer to this question is presented in Chapter 5. We provide an implementation of the safe operator precedence semantics based on data-dependent grammars. This implementation has the advantage that it does not depend on a grammar transformation that increases the size of the grammar, and is efficient.

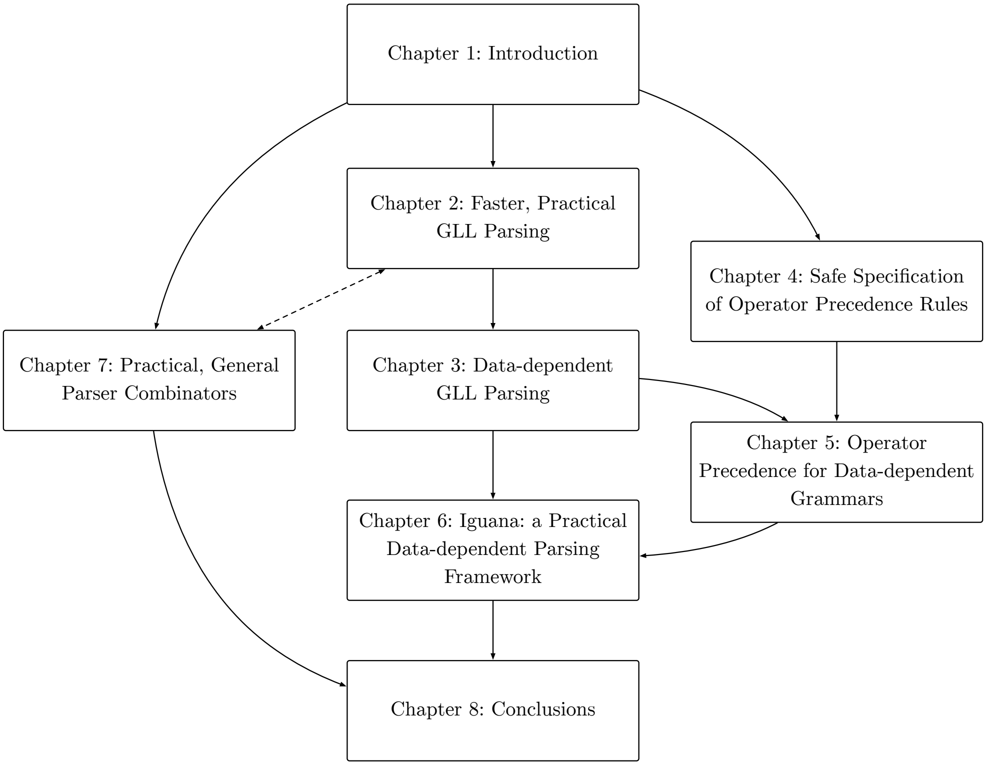

Chapter 1: Introduction
Chapter 2: Faster, Practical GLL Parsing
Chapter 3: Data-dependent GLL Parsing
Chapter 4: Safe Specification of Operator Precedence Rules
Chapter 5: Operator Precedence for Data-dependent Grammars
Chapter 6: Iguana: a Practical Data-dependent Parsing Framework
Chapter 7: Practical, General Parser Combinators
Chapter 8: Conclusions

Figure 1.13: The relationship between chapters.

Research Question 5. How can we implement general parser combinators that provide the expressiveness and worst-case cubic runtime of traditional general parsers, and the flexibility of parser combinators?

The answer to this question is presented in Chapter 7. We start with Johnson's Continuation-Passing Style (CPS) recognizers and apply a modification to the memoization strategy that achieves worst-case cubic runtime. Then, we show how to extend the cubic CPS recognizers to fully general parsers that produce binarized SPPFs in cubic time and space. We present a parser combinator library in Scala, called Meerkat, that is based on our cubic CPS parsers.

1.6.2 Overview and Origin of Chapters

Each chapter of this thesis is published separately in the proceedings of a peer-reviewed conference. There are some repetition in the introduction of the individual chapters, mostly related to the discussion of general parsing and declarative syntax definition, but we decided to keep the chapters as close as possible to the published version, so that they are self-contained and standalone.

The chapters follow three inter-related topics: (1) improvements to GLL parsing and extension to support data-dependent grammars, (2) operator precedence disambiguation, and (3) general parser combinators. Figure 1.13 shows the chapters and how they are related to each other. Chapters 2 and 7 are not directly dependent on each other, and the reader does not need to read one before the other, but ideas presented in these two chapters are complementary and give a full view of how a general top-down parsing algorithm can be formulated. In the following, we give a brief overview of the chapters including the venue where they were originally published.

We also explicitly describe the contributions of each of the two authors to each chapter. One unique aspect of this thesis is that it fuses the ideas from classical parsing with ideas from functional programming. This fusion is the result of the author's different background: Afroozeh has a background in classical parsing and gradually moved to functional programming, while Izmaylova has a background in functional programming and gradually moved to classical parsing. In the rest of this section, based on the alphabetical order, we refer to Afroozeh and Izmaylova as the first and second author, respectively.

Chapter 2: Faster, Practical GLL Parsing presents a number of improvements to GLL parsing. First, it presents a new GSS structure that leads to considerable performance gain on both highly ambiguous grammars and grammars of real programming languages. Second, it discusses a number of optimizations to GLL parsing, by relaxing some expensive and redundant checks. Third, it discusses an efficient implementation of lookup tables in an object-oriented implementation of GLL. Finally, it discusses the implementation of lexical disambiguation filters in GLL. The improved GLL parsers are evaluated against source code of real programming languages, such as Java, C# and OCaml, and report considerable performance improvement.

This chapter was originally published as:

A. Afroozeh and A. Izmaylova. Faster, Practical GLL Parsing. In *Proceedings of the 24th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, CC '15, pages 89–108. Springer, 2015.

Both authors contributed equally to the design and implementation of new GSS. For the rest of this work, the first author contributed more to the implementation of optimizations and disambiguation filters, while the second author contributed more to reasoning about correctness of the new GSS and other optimizations.

Chapter 3: Data-dependent GLL Parsing presents a parsing framework based on data-dependent grammars and an implementation based on GLL parsing. This chapter first presents data-dependent grammars as an intermediate parser-independent layer for implementing various disambiguation constructs. As data-dependent grammars are rather low-level, we give examples of high-level disambiguation constructs for lexical disambiguation filters, operator precedence, indentation-sensitive languages and conditional directives, and mappings from these high-level constructs to data-dependent grammars.

This chapter was originally published as:

A. Afroozeh and A. Izmaylova. One Parser to Rule Them All. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! '15, pages 151–170. ACM, 2015.

Both authors contributed equally to the design of data dependency in GLL and mappings from high-level disambiguation constructs. The first author was responsible for implementing the interpretive version of GLL and conducting evaluation experiments. The second author was responsible for the extension of GLL with data dependency and implementation of mappings from high-level constructs to data-dependent grammars.

Chapter 4: Safe Specification of Operator Precedence Rules presents a declarative approach to operator precedence that is safe, i.e., does not remove sentences from the language when there is no ambiguity, and can deal with intricate cases of operator precedence in functional programming languages such as OCaml. The safe operator precedence semantics is implemented as a grammar transformation. The parse trees resulting from the transformed grammar conform to the ones of the original, ambiguous grammar.

This chapter was originally published as:

A. Afroozeh, M. van den Brand, A. Johnstone, E. Scott, and J. J. Vinju. Safe Specification of Operator Precedence Rules. In *Proceedings of the 6th International Conference on Software Language Engineering*, SLE '13, pages 137–156. Springer, 2013.

The first author was the main author of this work, and carried out most of the design and implementation efforts. The other authors helped in formalization of ideas and presentation of the work.

Chapter 5: Operator Precedence for Data-dependent Grammars presents an implementation of the safe operator precedence semantics based on data-dependent grammars. This implementation does not have the problems of the rewriting approach presented in the previous chapter, and is efficient. This chapter also presents an extensive discussion of operator precedence techniques.

This chapter was originally published as:

A. Afroozeh and A. Izmaylova. Operator Precedence for Data-dependent Grammars. In *Proceedings of the ACM SIGPLAN Symposium/Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 13–24. ACM, 2016.

The first author worked on the related work study and conducted the evaluation experiments. The second author designed and implemented the operator precedence approach in data-dependent grammars.

Chapter 6: Iguana: A Practical Data-dependent Parsing Framework presents Iguana, our data-dependent parsing framework. This chapter discusses the high-level architecture of Iguana, its textual syntax, and the format of parse trees produced by Iguana. The main contribution of this chapter is an extensive performance evaluation, in which we compare the performance of Iguana with ANTLR [69, 70]. The results show that Iguana is practical for parsing real programming languages such as Java.

This chapter was originally published as:

A. Afroozeh and A. Izmaylova. Iguana: A Practical Data-Dependent Parsing Framework. In *Proceedings of the 25th International Conference on Compiler Construction*, CC '16, pages 267–268, Springer, 2016.

For this thesis, we extended the published paper to include the performance comparison between Iguana and ANTLR. Both authors contributed equally to the design and implementation of Iguana. The first author was more responsible for the implementation of the parts related to GLL, and the second author for the implementation of data-dependent grammars, mappings from high-level disambiguation constructs, and the IntelliJ IDEA plugin.

Chapter 7: Practical, General Parser Combinators presents general parser combinators that can deal with all context-free grammars and produce a binarized SPPF in cubic time and space. The presented parser combinators have the flexibility and expressiveness of traditional parser combinators, and the performance guarantee of general parsing algorithms. Our general parser combinators are based on Johnson's work on memoized Continuation-Passing Style (CPS) recognizers. First, we extend this work to achieve recognition in cubic time. Second, we extend the resulting cubic CPS recognizers to parsers that construct a binarized Shared Packed Parse Forest (SPPF). We used the approach presented in this work as the basis for Meerkat, a general parser combinator library in Scala.

This chapter was originally published as:

A. Izmaylova, A. Afroozeh, and T. v. d. Storm. Practical, General Parser Combinators. In *Proceedings of the ACM SIGPLAN Symposium/Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 1–12. ACM, 2016.

Izmaylova is the main author of this work. Izmaylova proposed the extension to the memoization strategy of the original CPS recognizers, worked on the formal proofs and carried out most of the implementation. Afroozeh implemented the binarized SPPFs and conducted the related work study. The other author participated in discussions and helped in presentation of the work.

1.7 Performance Evaluation

In most chapters of this thesis we provide a performance evaluation section, in which we mainly answer the following two questions:

1. How does the presented modification/extension in the chapter improve the parsing performance? For this type of evaluation, we compare the modified/extended version with the original one. For example, in Chapter 2 we propose a modification to the Graph Structured Stack (GSS) in GLL parsing that considerably improves the performance on both highly ambiguous and real programming languages. For evaluation, we compare the performance of GLL parsers with the new GSS with GLL parsers with the original GSS.

2. Is the presented modification/extension in the chapter practical? In many places we call a general parser practical if it can run nearly linearly on grammars of real programming languages. For example, in Chapter 3 we extend GLL parsing to support data-dependent grammars and show that the runtime is still near-linear on grammars of real programming languages such as Java and C#.

In these chapters, we did not provide performance comparison with other existing parsing tools. Comparing the performance of different parsing algorithms, implemented as part of existing tools, is very difficult, and, in fact, there are only very few research papers that do such comparison. There are a number of reasons why such fair performance comparison is difficult:

1. To have a fair performance comparison between different parsing techniques, we need to use the exact same grammar. For performance evaluation in the following chapters, we use the most natural version of the grammar from the language specification. Most parsers used in the actual compilers of the languages do not use such natural grammars. Instead, they use deterministic grammars that are tuned to perform well for a specific parsing technology. Based on our experience, making grammars more deterministic, or tuning the grammar, considerably improves the performance of parsing.

2. Almost all parsers used in front-ends of real programming languages use a separate tokenizer, which in combination with a deterministic parser makes the whole parsing process much faster. With context-aware scanning [91] we could get a considerable speedup compared to the character-level grammars (see Chapter 3). However, because of the non-deterministic nature of our parser, it explores many more paths than a deterministic parser with a separate tokenizer.

3. Compilers for many programming languages are not written in Java, and since Iguana is only currently implemented in Java, a direct comparison would be hard. We should also note that measuring performance on the JVM platform is in general difficult, mostly because of the Garbage Collection (GC) and Just-In-Time (JIT) compilation effects.

One of the most common questions we received during these years was about the performance of Iguana compared to ANTLR [69,70]. ANTLR is the most popular parsing tool out there, and since version 4 supports left recursion (with the exception of indirect left recursion) and declarative operator precedence disambiguation, allowing a natural grammar specification. ANTLR, however, is not a general parsing algorithm, as it cannot present all ambiguities in form of a parse forest, and returns at most one parse tree. In addition, ANTLR uses a separate tokenization phase before parsing. Nevertheless, from the end user's point of view we think it is important to provide a detailed performance comparison between Iguana and ANTLR.

In Chapter 6 we provide the results of our extensive performance comparison with ANTLR. We use similar natural grammars of Java and parse 22319 Java source files. The results show that Iguana is about 70% slower than ANTLR, considering the median relative running time of each file excluding the outliers. The median running time for Iguana and ANTLR for all files was 6.2 ms and 3.85 ms, respectively. These results show that Iguana is practical for parsing real programming languages such as Java.

1.8 Software Artifacts

In the context of this thesis we have developed two parsing frameworks, which are available under the open source BSD 2-clause[20] license.

Iguana Parsing Framework Iguana[21] is a parsing framework based on data-dependent grammars. Data-dependent grammars extend context-free grammars with arbitrary computation, variable binding, and constraints. These powerful features enable construction of parsers for context-sensitive languages. We also use data-dependent grammars as a layer to implement different disambiguation constructs such as operator precedence. The architecture of Iguana is described in Chapter 6.

Meerkat Parser Combinators Meerkat[22] is a general parser combinator library written in Scala, that combines the flexibility of traditional, monadic parser combinators and the expressivity and worst-case performance guarantees of state-of-the-art general parsing algorithms, such as GLL and GLR. Using the Meerkat library, it is possible to directly encode any context-free grammar, including the ones with direct and indirect left recursion, in Scala. Meerkat parsers support ambiguity, by producing a parse forest in cubic time and space, and behave nearly linearly on grammars of real programming languages.

[20] `https://opensource.org/licenses/BSD-2-Clause`

[21] `http://iguana-parser.github.io`

[22] `http://meerkat-parser.github.io`

Chapter 2

Faster, Practical GLL Parsing[1]

Summary Generalized LL (GLL) parsing is an extension of recursive-descent (RD) parsing that supports all context-free grammars in cubic time and space. GLL parsers have the direct relationship with the grammar that RD parsers have, and therefore, compared to GLR, are easier to understand, debug, and extend. This makes GLL parsing attractive for parsing programming languages.

In this chapter we propose a more efficient Graph-Structured Stack (GSS) for GLL parsing that leads to significant performance improvement. We also discuss a number of optimizations that further improve the performance of GLL. Finally, for practical scannerless parsing of programming languages, we show how common lexical disambiguation filters can be integrated in GLL parsing.

Our new formulation of GLL parsing is implemented as part of the Iguana parsing framework. We evaluate the effectiveness of our approach using a highly-ambiguous grammar and grammars of real programming languages. Our results, compared to the original GLL, show a speedup factor of 10 on the highly-ambiguous grammar, and a speedup factor of 1.5, 1.7, and 5.2 on the grammars of Java, C#, and OCaml, respectively.

[1] This chapter was originally published as: A. Afroozeh and A. Izmaylova. Faster, Practical GLL Parsing. In *Proceedings of the 24th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, CC '15, pages 89–108. Springer, 2015.

2.1 Introduction

Developing efficient parsers for programming languages is a difficult task that is usually automated by a parser generator. Since Knuth's seminal paper [54] on LR parsing, and DeRemer's work on practical LR parsing (LALR) [17], parsers of many major programming languages have been constructed using LALR parser generators such as Yacc [44].

Grammars of most real programming languages, when written in their most natural form, are often ambiguous and do not fit deterministic classes of context-free grammars such as LR(k). Therefore, such grammars need to be transformed to conform to these deterministic classes. This process can be time consuming and error prone for new users, and the resulting derivation trees may also be different from those of the original, natural grammar. This is especially noticeable when encoding operator precedence and associativity in deterministic parsing techniques that do not support left recursion, as without left recursion, it is not possible to directly encode left-associative operators.

In addition, writing a deterministic grammar for a programming language requires the grammar writer to think more in terms of the parsing technology, rather than the intended semantics. Finally, maintaining a deterministic grammar can be problematic. A real-world example is the grammar of Java. In the first version of the Java Language Specification [31] the grammar was represented in an LALR(1) form. In the second (and later) version of the Java Language Specification, the LALR(1) grammar is replaced by a grammar that is not LALR(1), most likely due to the difficulties of maintaining an LALR(1) grammar as the language evolved[2].

Generalized LR (GLR) [85] is an extension of LR parsing that effectively handles shift/reduce conflicts in separate stacks, merged as a Graph Structured Stack (GSS) to trim exponentiality. As GLR parsers can deal with any context-free grammar, there is no restriction on the grammar. Moreover, GLR can behave linearly on LR grammars, and therefore, it is possible to build practical GLR parsers for programming languages [61, 89].

Although GLR parsers accept any context-free grammar, they have a complicated execution model, inherited from LR parsing. LR parsers are hard to modify, and it is hard to produce good error messages from an LR parser. Some major programming languages have switched from LR-based parser generators, such as Yacc, to hand-written recursive-descent parsers. For example, GNU's GCC and Clang, two major C++ front-ends, have switched from LR(k) parser generators to hand-written recursive-descent parsers[3].

Recursive-descent (RD) parsers are a procedural interpretation of a grammar, directly encoded in a programming language. The straightforward execution model of RD parsers makes them easy to understand and modify. However, RD parsers do

[2] Even the first version of the Java Language Specification mentions that the presented LALR(1) grammar "cannot be parsed left-to-right with one token of lookahead because of certain syntactic peculiarities, some of them inherited from C and C++", and explains how these problems are resolved [31].

[3] http://clang.llvm.org/features.html#unifiedparser
http://gcc.gnu.org/wiki/New_C_Parser

not support left-recursive rules and have worst-case exponential runtime. Generalized LL (GLL) [78] is a generalization of RD parsing that can deal with any context-free grammar, including the ones with left recursive rules, in cubic time and space. GLL uses GSS to handle multiple function call stacks, which also solves the problem of left recursion by allowing cycles in the GSS. GLL parsers maintain the direct relationship with the grammar that RD parsers have, and therefore, provide an easy to understand execution model. Finally, GLL parsers can be written by hand and can be debugged in a programming language IDE. This makes GLL parsing attractive for parsing programming languages.

Contributions. We first identify a problem with the GSS in GLL parsing that leads to inefficient sharing of parsing results, and propose a new GSS that provides better sharing. We show that the new GSS results in significant performance improvement, while preserving the worst-case cubic complexity of GLL parsing. Second, we discuss a number of other optimizations that further improve the performance of GLL parsing. Third, we demonstrate how common lexical disambiguation filters, such as follow restrictions and keyword exclusion, can be implemented in a GLL parser. These filters are essential for scannerless parsing of real programming languages. The new GSS, the optimizations, and the lexical disambiguation filters are implemented as part of the Iguana parsing framework[4].

Organization of the chapter. The rest of this chapter is organized as follows. GLL parsing is introduced in Section 2.2. The problem with the original GSS in GLL parsing is explained in Section 2.2.3, and the new, more efficient GSS is introduced in Section 2.3. Section 2.4 gives a number of optimizations for implementing faster GLL parsers. Section 2.5 discusses the implementation of common lexical disambiguation mechanisms in GLL. Section 2.6 evaluates the performance of GLL parsers with the new GSS, compared to the original GSS, using a highly ambiguous grammar and grammars of real programming languages such as Java, C# and OCaml. Section 2.7 discusses related work on generalized parsing and disambiguation. Finally, Section 2.8 concludes this chapter and discusses future work.

2.2 GLL Parsing

2.2.1 Preliminaries

A context-free grammar is composed of a set of nonterminals N, a set of terminals T, a set of rules P, and a start symbol S which is a nonterminal. A rule is written as $A ::= \alpha$, where A (head) is a nonterminal and α (body) is a string in $(T \cup N)^*$. Rules with the same head can be grouped as $A ::= \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_p$, where each α_k is called an *alternative* of A. A *derivation step* is written as $\alpha A \beta \Rightarrow \alpha\gamma\beta$, where $A ::= \gamma$ is a rule, and α and β are strings in $(T \cup N)^*$. A *derivation* is a possibly empty sequence of derivation steps from α to β and is written as $\alpha \overset{*}{\Rightarrow} \beta$. A derivation is left-most if in

[4] https://github.com/iguana-parser/

each step the left most nonterminal is replaced by its body. A *sentential* form is a derivation from the start symbol. A *sentence* is a sentential form that only consists of terminal symbols. A sentence is called *ambiguous* if it has more than one left-most derivation.

2.2.2 The GLL parsing algorithm

The Generalized LL (GLL) parsing algorithm [78] is a fully general, worst-case cubic extension of recursive-descent (RD) parsing that supports all context-free grammars. In GLL parsing, the worst-case cubic runtime and space complexities are achieved by using a Graph-Structured Stack (GSS) and constructing a binarized Shared Packed Parse Forest (SPPF). GSS allows to efficiently handle multiple function call stacks, while a binarized SPPF solves the problem of unbounded polynomial complexity of Tomita-style SPPF construction [42]. GLL solves the problem of left recursion in RD parsing by allowing cycles in the GSS.

GLL parsing can be viewed as a grammar traversal process guided by the input string. At each point during execution, a GLL parser is at a grammar slot (grammar position) L, and maintains three variables: c_I for the current input position, c_U for the current GSS node, and c_N for the the current SPPF node. A grammar slot is of the form $X ::= \alpha \cdot \beta$ and corresponds to a grammar position before or after any symbol in the body of a grammar rule, similar to LR(0) items. A GSS node corresponds to a function call in an RD parser, and is of the form (L, i), where L is a grammar slot of the form $X ::= \alpha A \cdot \beta$, i.e., after a nonterminal, and i is the current input position when the node is created. Note that the grammar slot of a GSS node effectively records the return grammar position, needed to continue parsing after returning from a nonterminal. A GSS edge is of the form (v, w, u), where v and u are the source and target GSS nodes, respectively, and w is an SPPF node recorded on the edge.

GLL parsers produce a binarized SPPF. In an SPPF, nodes with the same subtrees are shared, and different derivations of a node are attached via packed nodes. A binarized SPPF introduces *intermediate* nodes, which effectively group the symbols of an alternative in a left-associative manner. An example of a binarized SPPF, resulting from parsing `"abc"` using the grammar $S ::= aBc \,|\, Ac$, $A ::= ab$, $B ::= b$ is shown in Figure 2.1.

A binarized SPPF has three types of nodes. *Symbol* nodes of the form (x, i, j), where x is a terminal or nonterminal, and i and j are the left and right extents, respectively, indicating the substring recognized by x. *Intermediate* nodes of the form $(A ::= \alpha \cdot \beta, i, j)$, where $|\alpha|, |\beta| > 0$, and i and j are the left and right extents, respectively. Terminal nodes are leaf nodes, while nonterminal and intermediate nodes have *packed nodes* as children. A packed node (shown as circles in the SPPF above) is of the form $(A ::= \alpha \cdot \beta, k)$, where k, the *pivot*, is the right extent of the left child. A packed node has at most two children, both non-packed nodes. A packed node represents a derivation, thus, a nonterminal or intermediate node having more than one packed node is ambiguous.

As mentioned before, a GLL parser holds a pointer to the current SPPF node, c_N, and at the beginning of each alternative, c_N is set to the dummy node, \$. As the parser

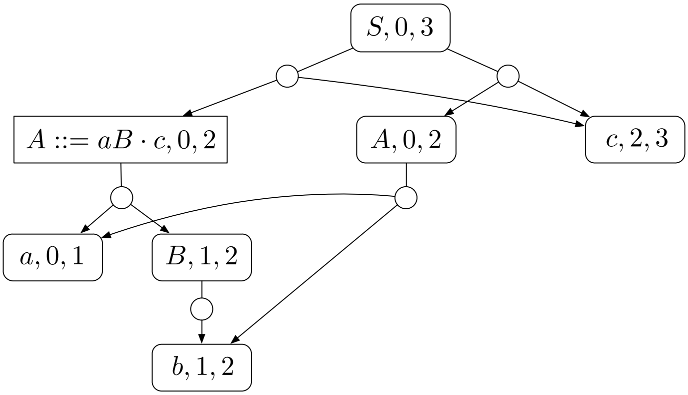


Figure 2.1: Binarized SPPF for the input `"abc"` using the grammar $S ::= aBc \,|\, Ac$, $A ::= ab$, $B ::= b$.

traverses an alternative, it creates terminal nodes by calls **getNodeT**(t, i, j), where t is a terminal, and i and j are the left and right extents, respectively. Nonterminal and intermediate nodes are created by calls **getNodeP**$(A ::= \alpha \cdot \beta, w, z)$, where w and z are the left and right children, respectively. This function first searches for an existing nonterminal node (A, i, j), if $|\beta| = 0$, or intermediate node $(A ::= \alpha \cdot \beta, i, j)$, where i and j are the left extent of w and the right extent of z, respectively. If such a node exists, it is retrieved, otherwise created. Then, w and z are attached to the node via a packed node, if such a packed node does not exist.

In GLL parsing, when the parser reaches a non-deterministic point, e.g., a nonterminal with multiple alternatives, it creates *descriptors*, which capture the parsing states corresponding to each choice, and adds them to a set, so that they can be processed later. A descriptor is of the form (L, u, i, w), where L is a grammar slot, u is a GSS node, i is an input position, and w is an SPPF node. A GLL parser maintains two sets of descriptors: $\mathcal{R}$ for pending descriptors, and $\mathcal{U}$ for storing all the descriptors created during the parsing, to eliminate the duplicate descriptors. A descriptor is added to $\mathcal{R}$, via a call to function **add**, only if it does not exist in $\mathcal{U}$. In addition, a set $\mathcal{P}$ is maintained to store and reuse the results of parsing associated with GSS nodes, i.e., the elements of the form (u, z), where z is an SPPF node. A GLL parser has a main loop that in each iteration, removes a descriptor from $\mathcal{R}$, sets c_U, c_I, and c_N to the respective values in the descriptor, and jumps to execute the code associated with the grammar slot of the descriptor. An example of a GLL parser, for the grammar Γ_0: $A ::= aAb \,|\, aAc \,|\, a$, is given in Figure 2.2.

We describe the execution of a GLL parser by explaining the steps of the parser at different grammar slots. Here, and in the rest of the chapter, we do not include the check for first/follow sets in the discussion. We also assume that the input string, of length n, is available as an array I. Parsing starts by calling the start symbol at input position 0. At this moment, c_U is initialized by the default GSS node $u_0 = (L_0, 0)$, where L_0 does not correspond to any actual grammar position.

```
        $\mathcal{R} := \emptyset; \mathcal{P} := \emptyset; \mathcal{U} := \emptyset$
        $c_U := (L_0, 0); c_I := 0; c_N := \$$

$L_0$ :**if**$(\mathcal{R} \neq \emptyset)$
          **remove**$(L, u, i, w)$ **from** $\mathcal{R}$
          $c_U := u;\ c_I := i;\ c_N := w;$ **goto** $L$
        **else if** (there exists a node $(A, 0, n)$)
          report success
        **else** report failure

$L_A$ :**add**$(A ::= .aAb, c_U, c_I, \$)$
        **add**$(A ::= .aAc, c_U, c_I, \$)$
        **add**$(A ::= .a, c_U, c_I, \$)$
        **goto** $L_0$

$L_{\cdot aAb}$ :**if**$(I[c_I] = a)$
          $c_N :=$ **getNodeT**$(a, c_I, c_I + 1)$
        **else goto** $L_0$
        $c_I := c_I + 1$
        $c_U :=$ **create**$(A ::= aA \cdot b, c_U, c_I, c_N)$
        **goto** $L_A$

$L_{\cdot aAc}$ :**if**$(I[c_I] = a)$
          $c_N :=$ **getNodeT**$(a, c_I, c_I + 1)$
        **else goto** $L_0$
        $c_I := c_I + 1$
        $c_U :=$ **create**$(A ::= aA \cdot c, c_U, c_I, c_N)$
        **goto** $L_A$

$L_{aA\cdot b}$ :**if**$(I[c_I] = b)$
          $c_R :=$ **getNodeT**$(b, c_I, c_I + 1)$
        **else goto** $L_0$
        $c_I := c_I + 1$
        $c_N :=$ **getNodeP**$(A ::= aAb\cdot, c_N, c_R)$
        **pop**$(c_U, c_I, c_N)$
        **goto** $L_0$

$L_{aA\cdot c}$ :**if**$(I[c_I] = c)$
          $c_R :=$ **getNodeT**$(c, c_I, c_I + 1)$
        **else goto** $L_0$
        $c_I := c_I + 1$
        $c_N :=$ **getNodeP**$(A ::= aAc\cdot, c_N, c_R)$
        **pop**$(c_U, c_I, c_N)$
         **goto** $L_0$
```

Figure 2.2: GLL parser for the grammar $A ::= aBb \mid aAc \mid a$

Let X be a nonterminal defined as $X ::= \alpha_1 \,|\, \alpha_2 \,|\, \ldots \,|\, \alpha_p$. A GLL parser starts by creating and adding descriptors, each corresponding to the beginning of an alternative: $(X ::= \cdot\alpha_k, c_U, c_I, \$)$. Then, the parser goes to L_0.

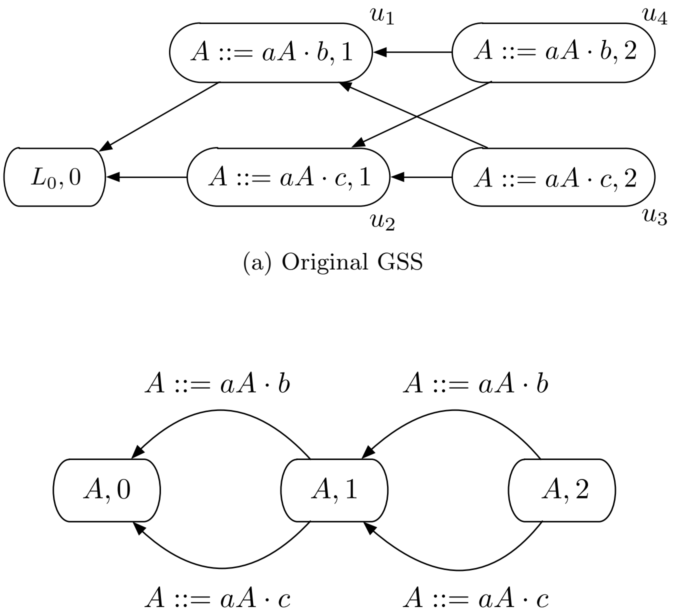
Based on the current grammar slot, a GLL parser continues as follows. If the grammar slot is of the form $X ::= \alpha \cdot t\beta$, the parser is before a terminal. If $I[c_I] \neq t$, the parser jumps to L_0, terminating this execution path, otherwise a terminal node is created by **getNodeT**$(t, c_I, c_I\!+\!1)$. If $|\alpha| \geq 1$, the terminal node is assigned to c_R, and an intermediate or nonterminal node is created by **getNodeP**$(X ::= \alpha t \cdot \beta, c_N, c_R)$, and assigned to c_N. The parser proceeds with the next grammar slot.

If the grammar slot is of the form $X ::= \alpha \cdot A\beta$, i.e., before a nonterminal, the **create** function is called with four arguments: the grammar slot $X ::= \alpha A \cdot \beta$, c_U, c_I, and c_N. First, **create** either retrieves a GSS node $(X ::= \alpha A \cdot \beta, c_I)$ if such a node exists, or creates one. Let v be $(X ::= \alpha A \cdot \beta, c_I)$. Then, a GSS edge (v, c_N, c_U) is added from v to c_U, if such an edge does not exists. If v was retrieved, the currently available results of parsing A at c_I are reused to continue parsing: for each element (v, z) in $\mathcal{P}$, a descriptor $(X ::= \alpha A \cdot \beta, c_U, h, y)$ is added, where y is the SPPF node returned by **getNodeP**$(X ::= \alpha A \cdot \beta, c_N, z)$, and h is the right extent of z. Finally, the call to **create** returns v, which is assigned to c_U. Then, the parser jumps to the definition of A and adds a descriptor for each of its alternatives.

If the grammar slot is of the form $A ::= \alpha\cdot$, the parser is at the end of an alternative, and therefore, should return from A to the calling rule and continue parsing. This corresponds to the return from a function call in an RD parser. The **pop** function is called with three arguments: c_U, c_I, c_N. Let (L, j) be the label of c_U. First, the element (c_U, c_N) is added to set $\mathcal{P}$. Then, for each outgoing edge (c_U, z, v) from c_U, a descriptor of the form (L, v, c_I, y) is created, where y is the SPPF node returned by **getNodeP**(L, z, c_N). Parsing terminates and reports success if all descriptors in $\mathcal{R}$ are processed and an SPPF node labeled $(S, 0, n)$, corresponding to the start symbol and the whole input string, is found, otherwise reports failure.

2.2.3 Problems with the original GSS in GLL parsing

To illustrate the problems with the original GSS in GLL parsing, we consider the grammar Γ_0 (Section 2.2.2) and the input string `"aac"`. Parsing this input string results in the GSS shown in Figure 2.3 (a). The resulting GSS has two separate GSS nodes for each input position, 1 and 2, and each GSS node corresponds to an instance of A in one of the two alternatives: aAb or aAc. This implies that, for example, the following two descriptors, corresponding to the beginning of the first alternative of A, are created and added to $\mathcal{R}$: $(A ::= \cdot aAb, u_1, 1, \$)$, which is added after creating u_1, and $(A ::= \cdot aAb, u_2, 1, \$)$, which is added after creating u_2. Although both descriptors correspond to the same grammar position and the same input position, they are distinct as their parent GSS nodes, u_1 and u_2, are different. The same holds for the following descriptors corresponding to the other alternatives of A: $(A ::= \cdot aAc, u_1, 1, \$)$, $(A ::= \cdot aAc, u_2, 1, \$)$ and $(A ::= \cdot a, u_1, 1, \$)$, $(A ::= \cdot a, u_2, 1, \$)$. This example demonstrates that, although the results of parsing A only depend on the alternatives of A and the current input position, GLL creates



(a) Original GSS

(b) New GSS

Figure 2.3: Original and new GSS for parsing "aac" using $A ::= aAb \mid aAc \mid a$.

separate descriptors for each instance of A, leading to multiple executions of the same parsing actions.

However, the calls corresponding to different instances of A at the same input position are not completely repeated. As can be seen, sharing happens one level deeper in GSS. For example, processing $(A ::= \cdot aAb, u_1, 1, \$)$ or $(A ::= \cdot aAb, u_2, 1, \$)$ matches a, increases input position to 2 and moves the grammar pointer before A, leading to the call to the same instance of A at input position 2, which is handled by the same GSS node u_4 connected to u_1 and u_2. This sharing, however, happens per nonterminal instance. For example, if we consider the input string "aaacc", a can be matched at input position 2, and therefore, the same result but associated with different instances of A will be stored in set $\mathcal{P}$ as $(u_3, (A, 2, 3))$ and $(u_4, (A, 2, 3))$. Both nodes u_3 and u_4 will pop with the same result $(A, 2, 3)$, and given that both u_3 and u_4 are shared by u_1 and u_2, descriptors that, again, encode the same parsing actions, but account for different parent GSS nodes, will be created: $(A ::= aA \cdot b, u_1, 3, w_1)$, $(A ::= aA \cdot b, u_2, 3, w_1)$ and $(A ::= aA \cdot c, u_1, 3, w_2)$, $(A ::= aA \cdot c, u_2, 3, w_2)$, where $w_1 = (A ::= aA \cdot b, 0, 3)$ and $w_2 = (A ::= aA \cdot c, 0, 3)$.

2.3 More Efficient GSS for GLL Parsing

In this section, we propose a new GSS that, compared to the original GSS, provides a more efficient sharing of parsing results in GLL parsing. We use the fact that all calls corresponding to the same nonterminal and the same input position should produce the same results, and therefore, can be shared, regardless of a specific grammar rule in which the nonterminal occurs. The basic idea is that, instead of recording return grammar positions in GSS nodes, i.e., grammar slots of the form $X ::= \alpha A \cdot \beta$, names of nonterminals are recorded in GSS nodes, and return grammar positions are carried on GSS edges. Figure 2.3 (b) illustrates the new GSS resulting from parsing `"aac"` using Γ_0.

First, we introduce new forms of GSS nodes and edges. Let $X ::= \alpha \cdot A\beta$ be the current grammar slot, i be the current input position, u be the current GSS node, and w be the current SPPF node. As in the original GLL, at this point, a GSS node is either retrieved, if such a node exists, or created. However, in our setting, such a GSS node is of the form (A, i), i.e., with the label that consists of the name of a nonterminal, in contrast to $X ::= \alpha A \cdot \beta$ in the original GSS, and the current input position. Let v be a GSS node labeled as (A, i). As in the original GLL, a new GSS edge is created from v to u. However, in our setting, a GSS edge is of the form (v, L, w, u), where, in addition to w as in the original GSS, the return grammar position L, i.e., $X ::= \alpha A \cdot \beta$, is recorded.

Second, we remove the default GSS node $u_0 = (L_0, 0)$, which requires a special label that does not correspond to any grammar position. In our setting, the initial GSS node is of the form $(S, 0)$ and corresponds to the call to the grammar start symbol S at input position 0, e.g., $(A, 0)$ in Figure 2.3 (b).

Finally, we re-define the **create** and **pop** functions of the original GLL to accommodate the changes to GSS. We keep the presentation of these functions similar to the ones of the original GLL algorithm [78], so that the difference between the definitions can be easily seen. The new definitions of the **create** and **pop** functions are given in Figure 2.4, where L is of the form $X ::= \alpha A \cdot \beta$, $|\alpha|, |\beta| \geq 0$, u and v are GSS nodes, and w, y, z are SPPF nodes.

The **create** function takes four arguments: a grammar slot L of the form $X ::= \alpha A \cdot \beta$, a GSS node u, an input position i, and an SPPF node w. If a GSS node (A, i) exists (if-branch), the alternatives of A are not predicted at i again. Instead, after a GSS edge (v, L, w, u) is added, if such an edge does not exist, the currently available results of parsing A at i, stored in $\mathcal{P}$, are reused. For each result (v, z) in $\mathcal{P}$, an SPPF node y is constructed, and a descriptor (L, u, h, y) is added to continue parsing with the grammar slot $X ::= \alpha A \cdot \beta$ and the next input position h, corresponding to the right extent of y. If a GSS node (A, i) does not exist (else-branch), such a node is first created, then, an edge (v, L, w, u) is added, and finally, a descriptor for each alternative of A with the input position i and parent node v is created and added.

The **pop** function takes three arguments: a GSS node u, an input position i, and an SPPF node z. If an entry (u, z) exists in $\mathcal{P}$, the parser returns from the function. Otherwise, (u, z) is added to $\mathcal{P}$, and, for each outgoing GSS edge of u, a descriptor is added to continue parsing with the grammar slot recorded on the edge, the current

```
create($L, u, i, w$) {
    if (there exists a GSS node labeled $(A, i)$) {
        let $v$ be the GSS node labeled $(A, i)$
        if (there is no GSS edge from $v$ to $u$ labeled $L, w$) {
            add a GSS edge from $v$ to $u$ labeled $L, w$
            for ($(v, z) \in \mathcal{P}$) {
                let $y$ be the SPPF node returned by getNodeP($L, w, z$)
                add($L, u, h, y$) where $h$ is the right extent of $y$
            }
        }
    } else {
        create a new GSS node labeled $(A, i)$
        let $v$ be the newly-created GSS node
        add a GSS edge from $v$ to $u$ labeled $L, w$
        for (each alternative $\alpha_k$ of $A$) { add($A ::= \cdot\alpha_k, v, i, \$$) }
    }
    return $v$
}

pop($u, i, z$) {
    if ($(u, z)$ is not in $\mathcal{P}$) {
        add $(u, z)$ to $\mathcal{P}$
        for (all GSS edges $(u, L, w, v)$) {
            let $y$ be the SPPF node returned by getNodeP($L, w, z$)
            add($L, v, i, y$)
        }
    }
}
```

Figure 2.4: The new definitions of the **create** and **pop** functions.

input position and the SPPF node constructed from w and z.

As the signatures of the **create** and **pop** functions stay the same as in the original GLL, replacing the original GSS with the new GSS does not require any modification to the code generated for each grammar slot in a GLL parser. Also note that the new GSS resembles the memoization of function calls used in functional programming, as a call to a nonterminal at an input position is represented only by the name of the nonterminal and the input position.

2.3.1 Equivalence

As illustrated in Sections 2.2 and 2.3, in the original GLL, sharing of parsing results for nonterminals is done at the level of nonterminal instances. On the other hand, in GLL with the new GSS, the sharing is done at the level of nonterminals themselves, which is more efficient as, in general, it results in less descriptors being created and processed. In Section 2.6 we present the performance results showing that significant performance speedup can be expected in practice. In this section we discuss the difference between GLL parsing with the original and new GSS for the general case, and show that the two GLL versions are semantically equivalent.

The use of the new GSS, compared to the original one, prevents descriptors of the form (L, u_1, i, w) and (L, u_2, i, w) to be created. These descriptors have the same grammar slot, the same input position, the same SPPF node, but different parent GSS nodes. In GLL with the original GSS, such descriptors may be added to $\mathcal{R}$ when, in the course of parsing, calls to different instances of a nonterminal, say A, at the same input position, say i, are made. Each such call corresponds to a parsing state where the current grammar slot is of the form $X ::= \tau \cdot A\mu$ (i.e., before A), and the current input position is i. To handle these calls, multiple GSS nodes of the form $(X ::= \tau A \cdot \mu, i)$, where the grammar slot corresponds to a grammar position after A, are created during parsing. We enumerate all such grammar slots with L_k, and denote GSS nodes (L_k, i) as u_k.

When a GSS node u_k is created, descriptors of the form $(A ::= \cdot\gamma, u_k, i, \$)$ are added. If $a_1a_2 \ldots a_n$ is the input string and $A \stackrel{*}{\Rightarrow} a_{i+1} \ldots a_j$, u_k will pop at j, and processing descriptors of the form $(A ::= \cdot\gamma, u_k, i, \$)$ will lead to creation of descriptors of the form $(A ::= \alpha B \cdot \beta, u_k, l, w)$, $i \leq l \leq j$, i.e., in an alternative of A, and of the form $(A ::= \gamma\cdot, u_k, j, (A, i, j))$, i.e., at the end of an alternative of A. All these descriptors encode the parsing actions that do not semantically depend on a specific u_k. Indeed, starting from the same grammar position in an alternative of A, say $A ::= \alpha \cdot \beta$, regardless of a specific u_k, the parsing continues with the next symbol in the alternative and the current input position, and either produces an (intermediate) SPPF node, which does not depend on u_k, moving to the next symbol in the alternative, or fails. Finally, when descriptors of the form $(A ::= \gamma\cdot, u_k, j, (A, i, j))$ are processed, the same SPPF node (A, i, j) will be recorded in set $\mathcal{P}$ for each u_k.

In the original GLL, when u_k is being popped, for each (u_k, z) in set $\mathcal{P}$, where z is of the form (A, i, j), and each outgoing edge (u_k, w, v), a descriptor (L_k, v, j, y), where y is the SPPF node returned by **getNodeP**(L_k, w, z), is added to continue parsing after A. Let v be a GSS node with index h, then h and j are the left and right extents of y, respectively. In the following we show how using the new GSS, descriptors equivalent to (L_k, v, j, y) are created, but at the same time, the problem of repeating the same parsing actions is avoided.

In GLL with the new GSS, when calls to different instances of a nonterminal, say A, at the same input position, say i, are made, a GSS node $u = (A, i)$ is retrieved or created. Similar to the original GLL, when u is created, descriptors of the form $(A ::= \cdot\gamma, u, i, \$)$ are added, and if $A \stackrel{*}{\Rightarrow} a_{i+1} \ldots a_j$, descriptors of the form $(A ::= \alpha B \cdot \beta, u, l, w)$, $i \leq l \leq j$, and of the form $(A ::= \gamma\cdot, u, j, (A, i, j))$ will also be added. The essential difference with the original GLL is that the label of u is A, and therefore, the descriptors corresponding to parsing A at i are independent of the context in which A is used. Upon the first call to A at i, regardless of its current context, such descriptors are created, and the results are reused for any such call in a different context. Finally, when descriptors of the form $(A ::= \gamma\cdot, u, j, (A, i, j))$ are processed, the SPPF node $z = (A, i, j)$ is recorded as a single element (u, z) in set $\mathcal{P}$.

In GLL parsing with the new GSS, whenever the parser reaches a state with a grammar slot of the form $X ::= \tau \cdot A\mu$, and the input position i, there will be an edge (u, L_k, w, v) added to u, where L_k is of the form $X ::= \tau A \cdot \mu$. Finally, for each (u, z) in set $\mathcal{P}$ and each edge (u, L_k, w, v), the descriptor (L_k, v, j, y) will be added, where y is the SPPF node returned by **getNodeP**(L_k, w, z).

2.3.2 Complexity

In this section we show that replacing the original GSS with the new GSS does not affect the worst-case cubic runtime and space complexities of GLL parsing. To introduce the new GSS into GLL parsing, we changed the forms of GSS nodes and edges. We also re-defined the **create** and **pop** functions to accommodate these changes. However, all these modifications had no effect on the SPPF construction, the **getNode** functions, and the code of GLL parsers that uses **create** and **pop** to interact with GSS. Specifically, this implies that when the main loop of a GLL parser executes, and the next descriptor is removed from $\mathcal{R}$, the execution proceeds in the same way as in the original GLL parsing until the call to either **create** or **pop** is made.

First, we show that the space required for the new GSS is also at most $O(n^3)$. In the new GSS, all GSS nodes have unique labels of the form (A, i), where $0 \leq i \leq n$. Therefore, the new GSS has at most $O(n)$ nodes. In the new GSS, all GSS edges have unique labels of the form (u, L, w, v), where L is of the form $X ::= \alpha A \cdot \beta$, the source GSS node u is of the form (A, i), and the target GSS node v is of the form (X, j). The label of an edge in the new GSS consists of L and w, where w has j and i as the left and right extents, which are also the indices of v and u, respectively. Given that $0 \leq j \leq i \leq n$, the number of outgoing edges for any source GSS node u is at most $O(n)$, and the new GSS has at most $O(n^2)$ edges. Thus the new GSS requires at most $O(n)$ nodes and at most $O(n^2)$ edges.

The worst-case $O(n^3)$ runtime complexity of the original GLL follows from the fact that there are at most $O(n^2)$ descriptors, and processing a descriptor may take at most $O(n)$ time, by calling **pop** or **create**. Now, we show that the worst-case complexity of both **create** and **pop** is still $O(n)$, and the total number of descriptors that can be added to $\mathcal{R}$ is still at most $O(n^2)$. All elements in set $\mathcal{P}$ are of the form (v, z), where v is of the form (A, i), and z has i and j as the left and right extents, respectively, where $0 \leq i \leq j \leq n$. Therefore, the number of elements in $\mathcal{P}$, corresponding to the same GSS node, is at most $O(n)$. Since a GSS node has at most $O(n)$ outgoing edges, $\mathcal{P}$ has at most $O(n)$ elements corresponding to a GSS node, and the new GSS and $\mathcal{P}$ can be implemented using arrays to allow constant time lookup, both **create** and **pop** have the worst-case complexity $O(n)$.

Finally, a descriptor is of the form (L, u, i, w), where w is either \$ or has j and i as the left and right extents, respectively, and j is also the index of u. Thus the total number of descriptors that can be added to $\mathcal{R}$ is at most $O(n^2)$.

2.4 Optimizations for GLL Implementation

The GLL parsing algorithm [78] is described using a set view, e.g., $\mathcal{U}$ and $\mathcal{P}$, which eases the reasoning about the worst-case complexity, but leaves open the challenges of an efficient implementation. The worst-case $O(n^3)$ complexity of GLL parsing requires constant time lookup, e.g., to check if a descriptor has already been added. Constant time lookup can be achieved using multi-dimensional arrays of size $O(n^2)$, however, such an implementation requires $O(n^2)$ initialization time, which makes it impractical for near-linear parsing of real programming languages, whose grammars are nearly deterministic.

For near-linear parsing of real programming languages we need data structures that provide amortized constant time lookup, without excessive overhead for initialization. One way to achieve this is to use a combination of arrays and linked lists as described in [45]. In this approach the user needs to specify, based on the properties of the grammar, which dimensions should be implemented as arrays or linked lists.

In this section we propose an efficient hash table-based implementation of GLL parsers. We show how the two most important lookup structures, $\mathcal{U}$ and $\mathcal{P}$, can be implemented using local hash tables in GSS nodes. The idea is based on the fact that the elements stored in these data structures have a GSS node as a property. Instead of having a global hash table, we factor out the GSS node and use hash tables that are local to a GSS node. In an object-oriented language, we can model a GSS node as an object that has pointers to its local hash tables. In the following, we discuss different implementations of $\mathcal{U}$ and $\mathcal{P}$. We consider GLL parsing with new GSS, and assume that n is the length of the input, and $|N|$ and $|L|$ are the number of nonterminals and grammar slots, respectively.

Descriptor elimination set ($\mathcal{U}$): set $\mathcal{U}$ is used to keep all the descriptors created during parsing for duplicate elimination. A descriptor is of the form (L, u, i, w), where L is of the form $A ::= \alpha \cdot \beta$, u is of the form (A, j), and w is either a dummy node, or a symbol node of the form (x, j, i), when $\alpha = x$, or an intermediate node of the form (L, j, i). As can be seen, in a descriptor, the input index of the GSS node is the same as the left extent of the SPPF node, and the input index of the descriptor is the same as the right extent of the SPPF node. Also note that the label of the GSS and SPPF node is already encoded in L. Thus we can effectively consider a descriptor as (L, i, j). We consider three implementations of $\mathcal{U}$:

- *Global Array*: $\mathcal{U}$ can be implemented as an array of size $|L| \times n \times n$, which requires $O(n^2)$ initialization time.

- *Global hash table*: $\mathcal{U}$ can be implemented as a single global hash table holding elements of the form (L, i, j).

- *Local hash table in a GSS node*: $\mathcal{U}$ can be implemented as a local hash table in a GSS node. This way, we only need to consider a descriptor as (L, i).

Popped elements ($\mathcal{P}$)**:** The set of popped elements, $\mathcal{P}$, is defined as a set of (u, w), where u is a GSS node of the form (A, i), and w is an SPPF node of the form (A, i, j). For eliminating duplicates, $\mathcal{P}$ can effectively be considered as a set of (A, i, j). We consider three implementations of $\mathcal{P}$:

- *Global Array*: $\mathcal{P}$ can be implemented as an array of size $|N| \times n \times n$, which requires $O(n^2)$ initialization time.
- *Global hash table*: $\mathcal{P}$ can be implemented as a global hash table holding elements of the form (A, i, j).
- *Local hash table in a GSS node*: $\mathcal{P}$ can be implemented as a local hash table in a GSS node. This way we can eliminate duplicate SPPF nodes using a single integer, the right extent of the SPPF node (j).

Hash tables do not have the problem of multi-dimensional arrays, as the initialization cost is constant. However, using a global hash table is problematic for parsing large input files as the number of elements is in order of millions, leading to many hash collisions and resizing. For example, for a C# source file of 2000 lines of code, about 1,500,000 descriptors are created and processed.

Using local hash tables in GSS nodes instead of a single global hash table provides considerable speedup when parsing large inputs with large grammars. First, by distributing hash tables over GSS nodes, we effectively reduce the number of properties needed for hash code calculation. Second, local hash tables will contain fewer entries, resulting in fewer hash collisions and requiring fewer resizing. In the Iguana parsing framework we use the standard `java.util.HashSet` as the implementation of hash tables. Our preliminary results show that, for example, by using a local hash table for implementing $\mathcal{U}$ instead of a global one, we can expect speedup of factor two. Detailed evaluation of the optimizations presented in this section, and their effect on memory usage, is future work.

There are two algorithmic optimizations possible that further improve the performance of GLL parsers. These optimizations remove certain runtime checks that can be shown to be redundant based on the following properties:

1) There is at most one call to the create function with the same arguments. Thus no check for duplicate GSS edges is needed.

The properties of a GSS edge (v, L, w, u) are uniquely identified by the arguments to **create**: L, u, i, w, where L is of the form $X ::= \alpha A \cdot \beta$, and $v = (A, i)$. Therefore, if it can be shown that there is at most one call to **create** with the same arguments, the check for duplicate GSS edges can be safely removed.

Let us consider a call **create**$(X ::= \alpha A \cdot \beta, u, i, w)$. This call can only happen if a descriptor of one of the following forms has been processed, where τ is a possibly empty sequence of terminals and $j \leq i$: (1) $(X ::= \cdot\alpha A\beta, u, j, \$)$ when $\alpha = \tau$; or (2) $(X ::= \gamma B \cdot \tau A\beta, u, j, z)$ when $\alpha = \gamma B\tau$, $|\gamma| \geq 0$. Therefore, for the call to happen more than once, the same descriptor has to be processed again. However, this can never happen as all the duplicate descriptors are eliminated.

2) There is at most one call to the getNodeP function with the same arguments. Thus no check for duplicate packed nodes is needed.

Let us consider a call **getNodeP**$(A ::= \alpha \cdot \beta, w, z)$, where w is either \$ or a non-packed node having i and k as the left and right extents, and z is a non-packed node having k and j as the left and right extents. This call may create and add a packed node $(A ::= \alpha \cdot \beta, k)$ under the parent node, which is either (A, i, j) when $|\beta \models 0$, or $(A ::= \alpha \cdot \beta, i, j)$ otherwise. Clearly, the same call to **getNodeP** will try to add the same packed node under the existing parent node.

Now suppose that the same call to **getNodeP** happens for the second time. Given that a GSS node is ensured to pop with the same result at most once (set $\mathcal{P}$ and **pop**), the second call can only happen if a descriptor of one of the following forms has been processed for the second time, where $u = (A, i)$ and τ is a possibly empty sequence of terminals: (1) $(A ::= \cdot\alpha\beta, u, i, \$)$ when either $\alpha = \tau$ or $\alpha = \tau X$; or (2) $(A ::= \gamma B \cdot \sigma\beta, u, l, y)$, $i \leq l \leq k$, when $\alpha = \gamma B\sigma$, $|\gamma| \geq 0$, and either $\sigma = \tau$ or $\sigma = \tau X$. This can never happen as all the duplicate descriptors are eliminated.

Note that the second optimization is only applicable for GLL parsers with the new GSS. In the original GLL, u can be of the form $(X ::= \mu A \cdot \nu, i)$, and therefore, multiple descriptors with the same grammar slot, the same input position, the same SPPF node, but different parent nodes, corresponding to multiple instances of A, can be added, resulting in multiple calls to **getNodeP** with the same arguments.

2.5 Disambiguation Filters for Scannerless GLL Parsing

Parsing programming languages is often done using a separate scanning phase before parsing, in which a scanner (lexer) first transforms a stream of characters to a stream of tokens. Besides performance gain, another important reason for a separate scanning phase is that deterministic character-level grammars are virtually nonexistent. The main drawback of performing scanning before parsing is that, in some cases, it is not possible to uniquely identify the type of tokens without the parsing context (grammar rule in which they appear). An example is nested generic types in Java, e.g., `List<List<T>>`. Without the parsing context, the scanner cannot unambiguously detect the type of `>>` as it can be either a right-shift operator or two closing angle brackets.

Scannerless parsing [73,94] eliminates the need for a separate scanning phase by treating the lexical and context-free definitions the same. A scannerless parser solves the problems of identifying the type of tokens by parsing each character in its parsing context, and provides the user with a unified formalism for both syntactical and lexical definitions. This facilitates modular grammar development at the lexical level, which is essential for language extension and embedding [13].

A separate scanning phase usually resolves the character-level ambiguities in favor of the longest matched token and excludes keywords from identifiers. In absence of a separate scanner, such ambiguities should be resolved during parsing. In the rest of this section we show how most common character-level disambiguation filters [90] can be implemented in a GLL parser.

To illustrate character-level ambiguities, we use the grammar below, which is adapted from [90]. This grammar defines a `Term` as either a sequence of two terms, an identifier, a number, or the keyword `"int"`. `Id` is defined as one or more repetition of a single character, and `WS` defines a possibly empty blank.

```
Term   ::= Term WS Term | Id | Num | "int"
Id     ::= Chars
Chars  ::= Chars Char | Char
Char   ::= 'a' | .. | 'z'
Num    ::= '1' | .. |'9'
WS     ::= ' ' | ϵ
```

This grammar is ambiguous. For example, the input string `"hi"` can be parsed as either `Term(Id("hi"))`, or `Term(Term(Id("h")),Term(Id("i")))`. Following the longest match rule, the first derivation is the intended one, as in the second one `"h"` is recognized as an identifier, while it is followed by `"i"`. We can use a *follow restriction* ($-\!\!/\!\!-$) to disallow an identifier to be followed by another character: `Id ::= Chars -/- Char`. Another ambiguity occurs in the input string `"intx"` which can be parsed as either `Term(Id("intx"))` or `Term(Term("int"), Term(Id("x")))`. We can solve this problem by adding a *precede restriction* ($-\!\!\backslash\!\!-$) as follows: `Id ::= Char -\- Chars`, specifying that `Id` cannot be preceded by a character. Finally, we should exclude the recognition of `"int"` as `Id`. For this, we use an exclusion rule: `Id ::= Chars \ "int"`.

Below we formally define each of these restrictions and show how they can be integrated in GLL parsing. For follow and precede restrictions we only consider the case where the restriction is a single character, denoted by c. This can be trivially extended to other restrictions such as character ranges or arbitrary regular expressions. We assume that I represents the input string as an array of characters and i holds the current input position.

Follow restriction. For a grammar rule $A ::= \alpha x \beta$, a follow restriction for the symbol x is written as $A ::= \alpha x \; -\!\!/\!\!- \; c\beta$, meaning that derivations of the form $\gamma A \sigma \Rightarrow \gamma \alpha x \beta \sigma \overset{*}{\Rightarrow} \gamma \alpha x c \tau$ are disallowed. For implementing follow restrictions, we consider the grammar position $A ::= \alpha x \cdot \beta$. If x is a terminal, the implementation is straightforward: if $i < |I|$ and $I[i] = c$, the control flow returns to the main loop, effectively terminating this parsing path. If x is a nonterminal, we consider the situation where a GLL parser is about to create a descriptor for $A ::= \alpha x \cdot \beta$. This happens when pop is executed for a GSS node (x, j) at i. While iterating over the GSS edges, if a GSS edge labeled $A ::= \alpha x \cdot \beta$ is reached, the condition of the follow restriction associated with this grammar position will be checked. If $I[i] = c$, no descriptor for this label will be added.

Precede Restriction. For a grammar rule $A ::= \alpha x \beta$, a precede restriction for the symbol x is written as $A ::= \alpha c \mathrel{-\!\!\!\!\!\setminus} x\beta$, meaning that derivations of the form $\gamma A \sigma \Rightarrow \gamma \alpha x \beta \sigma \overset{*}{\Rightarrow} \tau c x \beta \sigma$ are disallowed. The implementation of precede restrictions is as follows. When a GLL parser is at the grammar slot $A ::= \alpha \cdot x \beta$, if $i > 0$ and $I[i-1] = c$, the control flow returns to the main loop, effectively terminating this parsing path.

Exclusion. For a grammar rule $A ::= \alpha X \beta$, the exclusion of string s from the nonterminal X is written as $A ::= \alpha X \backslash s \beta$, meaning that the language accepted by the nonterminal X should not contain the string s, i.e., $L(X \backslash s) = L(X) - \{s\}$, where L defines the language accepted by a nonterminal. Similar to the implementation of follow restrictions for a nonterminal, when a GSS node (X, j) is popped at i, and the parser iterates over the outgoing GSS edges, if an edge $A ::= \alpha X \cdot \beta$ is found, the condition of the exclusion is checked. If the substring of the input from j to i matches s, no descriptor for the grammar position $A ::= \alpha X \cdot \beta$ is added, which effectively terminates this parsing path.

2.6 Performance Evaluation

To evaluate the efficiency of the new GSS for GLL parsing, we use a highly ambiguous grammar and grammars of three real programming languages: Java, C# and OCaml. We ran the Iguana GLL parsers in two different modes: *new* and *original*, corresponding to the new and original GSS, respectively. Iguana is our Java-based GLL parsing framework that can be configured to run with the new or original GSS, while keeping all other aspects of the algorithm, such as SPPF creation, the same. The optimizations given in Section 2.4, with the exception of removing checks for packed nodes, which is only applicable to GLL with the new GSS, are applied to both modes.

We ran the experiments on a machine with a quad-core Intel Core i7 2.6 GHz CPU and 16 GB of memory running Mac OS X 10.9.4. We executed the parsers on a 64-Bit Oracle HotSpot$^{\text{TM}}$ JVM version 1.7.0_55 with the `-server` flag. To allow for JIT optimizations, the JVM was first warmed up, by executing a large sample data, and then each test is executed 10 times. The median running time (CPU user time) is reported.

2.6.1 Highly Ambiguous Grammar

To measure the effect of the new GSS for GLL parsing on highly ambiguous grammars, we use the grammar $S ::= SSS \,|\, SS \,|\, b$. The results of running a GLL parser with the new and original GSS for this grammar on strings of b's is shown in Figure 2.5. As can be seen, the performance gain is significant. The median and maximum speedup factors for the highly ambiguous grammar, as shown in Figure 2.6, are 10 and 14, respectively. To explain the observed speedup, we summarize the results of parsing the strings of b's in Table 2.1. Note that the number of nodes and edges for the original GSS are slightly more than the numbers reported in [78], as we do not include the

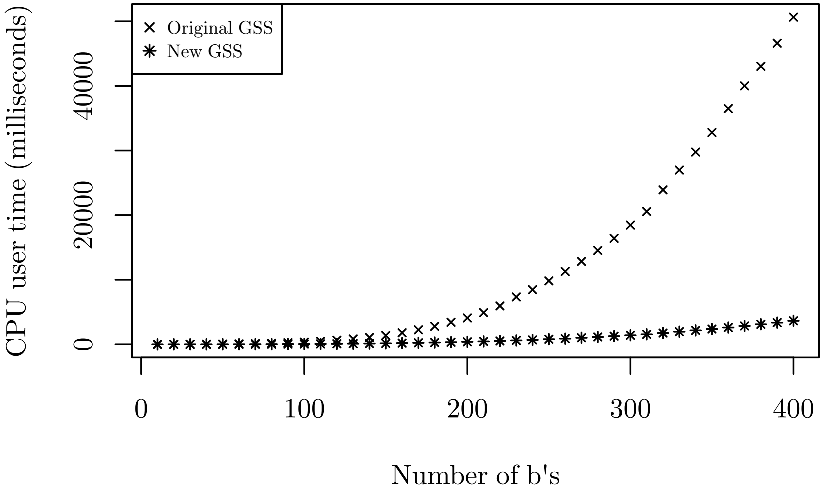


Figure 2.5: Running the GLL parsers for grammar $S ::= SSS \,|\, SS \,|\, b$

Table 2.1: The result of running highly ambiguous grammar on strings of b's.

size	time (ms)		# GSS nodes		# GSS edges	
	new	original	new	original	new	original
50	6	35	51	251	3877	18935
100	45	336	101	501	15252	75360
150	151	1361	151	751	34127	169285
200	386	4080	201	1001	60502	300710
250	791	9824	251	1251	94377	469635
300	1403	18457	301	1501	135752	676060
350	2367	32790	351	1751	184627	919985
400	3639	50648	401	2001	241002	1201410

check for first and follow sets. As can be seen, GLL with the new GSS has $n+1$ GSS nodes for inputs of length n, one for each call to S at input positions 0 to n. For GLL with the original GSS, there are 5 grammar slots that can be called: $S ::= S \cdot SS$, $S ::= SS \cdot S$, $S ::= SSS\cdot$, $S ::= S \cdot S$, and $S ::= SS\cdot$, which lead to $5n+1$ GSS nodes. In such a highly ambiguous grammar, most GSS nodes are connected, therefore, the iteration operations over edges in the create and pop functions will take much more time, as explained in Section 2.3.1.

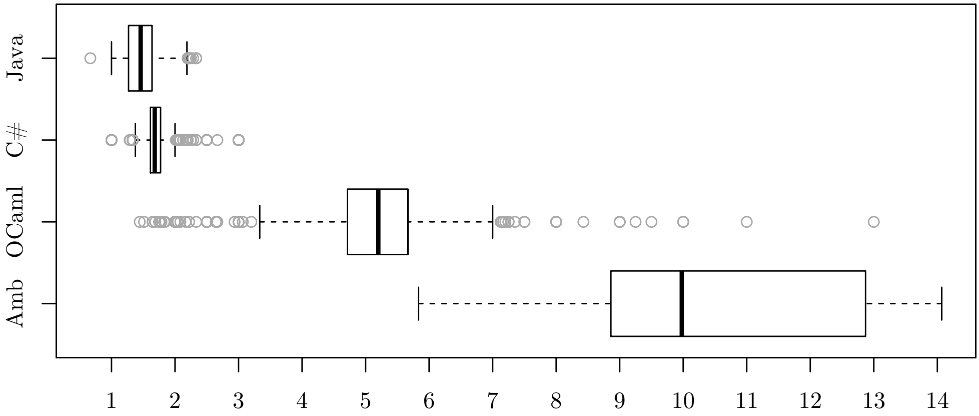


Figure 2.6: Comparing the speedup factor of the new and original GSS.

2.6.2 Grammars of Programming Languages

To measure the effect of the new GSS on the grammars of real programming languages, we have chosen the grammars of three programming languages from their language reference manual.

Java: We used the grammar of Java 7 from the Java Language Specification [32] (JLS). The grammar contains 329 nonterminals, 728 rules, and 2410 grammar slots. We have parsed 7449 Java files from the source code of JDK 1.7.0_60-b19. As shown in Figure 2.6, the median and maximum speedup factors for Java are 1.5 and 2.3, respectively.

C#: We used the grammar of C# 5 from the C# Language Specification [63]. The grammar contains 534 nonterminals, 1275 rules, and 4195 grammar slots. The main challenge in parsing C# files was dealing with C# directives, such as `#if` and `#region`. C# front ends, in contrast to C++, do not have a separate preprocessing phase for directives. Most C# directives can be ignored as comment, with the exception of the conditional ones, as ignoring them may lead to parse error. As the purpose of this evaluation was to measure the performance of GLL parsers on C# files, and not configuration-preserving parsing, we ran the GNU C preprocessor on the test files to preprocess the conditional directives. The rest of the directives were treated as comments. We have parsed 2764 C# files from the build-preview release of the Roslyn Compiler. As shown in Figure 2.6, the median and maximum speedup factors for C# are 1.7 and 3, respectively.

OCaml: We used the grammar of OCaml 4.0.1 from the OCaml reference manual [58]. The grammar of OCaml is different from Java and C# in two aspects. First, OCaml is an expression-based language, as opposed to Java and C#. This provides us with a grammar with different characteristics for testing the effectiveness of the new GSS. Second, the reference grammar of OCaml is highly ambiguous, having numerous operators with different associativity and priority levels. We used the grammar rewriting technique presented in [6] to obtain an unambiguous grammar. The rewritten grammar contains 685 nonterminals, 5728 rules, and 27294 grammar slots. We have parsed 871 files from the OCaml 4.0.1 source release. As shown in Figure 2.6, the median and maximum speedup factors for OCaml are 5.2 and 13, respectively. The rewriting technique [6] to encode operator precedence rules leads to more rules. This can be one reason for the more significant speedup for the OCaml case, compared to Java and C#. The other possible reason is the nature of OCaml programs that have many nested expressions, requiring high non-determinism. The case of OCaml shows that the new GSS is very effective for parsing languages with large and complex expression grammars, such as OCaml.

2.7 Related Work

For many years deterministic parsing techniques were the only viable option for parsing programming languages. As machines became more powerful, and the need for developing parsers in other areas such as reverse-engineering and source code analysis increased, generalized parsing techniques were considered for parsing programming languages. In this section we discuss several related work on applying generalized parsing to parsing programming languages.

Generalized parsing. Generalized parsing algorithms have the attractive property that they can behave linearly on deterministic grammars. Therefore, for the grammars that are nearly deterministic, which is the case for most programming languages, using generalized parsing is feasible [46]. For example, the ASF+SDF Meta-Environment [89] uses a variation of GLR parsing for source code analysis and reverse engineering.

The original GLR parsing algorithm by Tomita [85] fails to terminate for some grammars with ϵ rules. Farshi [67] provides a fix for ϵ rules, but his fix requires exhaustive GSS search after some reductions. Scott and Johnstone [76] provide an alternative to Farshi's fix, called Right Nulled GLR (RNGLR), which is more elegant and more efficient. GLR parsers have the worst-case $O(n^{k+1})$ complexity, where k is the length of the longest rule in the grammar [42]. BRNGLR is a variation of RNGLR that uses binarized SPPFs to enable GLR parsing in cubic time. Elkhound [61] is a GLR parser, based on Farshi's version, that switches to the machinery of an LR parser on deterministic parts of the grammar, leading to significant performance improvement. Another faster variant of GLR parsing is presented by Aycock and Horspool [10], which uses a larger LR automata, trading space for time.

Disambiguation. Disambiguation techniques that are used in different parsing technologies can be categorized in two groups: implicit or explicit disambiguation. Implicit disambiguation is mostly used in parsing techniques that return at most one derivation tree. Perhaps the name nondeterminism-reducer is a more correct term, as these techniques essentially reduce non-determinism during parsing, regardless if it leads to ambiguity or not. Yacc [44], PEGs [25] and ANTLR [70] are examples of parsing techniques that use implicit disambiguation rules. For example, in Yacc, shift/reduce conflicts are resolved in favor of shift, and PEGs and ANTLR use the order of the alternatives. Some parser generators that use implicit disambiguation give warning (at generation time) when the parser resolves the ambiguity. For example, Yacc generates warnings for resolved conflicts. Ignoring these warnings may lead to surprises at runtime, as the returned parse tree may not be the expected one. The generated warnings are usually highly tied to the inner workings of the parsing technique and often require expert knowledge for diagnosis.

Explicit disambiguation is usually done using declarative disambiguation rules. In this approach, the grammar formalism allows the user to explicitly define the disambiguation rules, which can be applied either during parsing, by pruning parsing paths that violate the rules, or be applied after the parsing is done, as a parse forest processing step. Post-parse filtering is only possible when using a generalized parser that can return all the derivations in form of a parse forest. Aho et. al show how to modify LR(1) parsing tables to resolve shift/reduce conflicts based on the the priority and associativity of operators [7]. In Scannerless GLR (SGLR) which is used in SDF2 [95], operator precedence and character-level restrictions such as keyword exclusion are implemented as parse table modifications, but some other disambiguation filters such as `prefer` and `avoid` as post-parse filters [90]. Economopoulos et al. [19] investigate the implementation of SDF disambiguation filters in the RNGLR parsing algorithm and report considerable performance improvement.

2.8 Conclusions

In this chapter we presented an essential optimization to GLL parsing, by proposing a new GSS, which provides a more efficient sharing of parsing results. We showed that GLL parsers with the new GSS are worst-case cubic in time and space, and are significantly faster on both highly ambiguous and near-deterministic grammars. As future work, we plan to measure the effect of the new GSS and the optimizations presented in Section 2.4 on memory, and to compare the performance of our GLL implementation with other parsing techniques.

Chapter 3

Data-dependent GLL Parsing[1]

Summary. Despite the long history of research in parsing, constructing parsers for real programming languages remains a difficult and painful task. In the last decades, different parser generators emerged to allow the construction of parsers from a BNF-like specification. However, still today, many parsers are handwritten, or are only partly generated, and include various hacks to deal with different peculiarities in programming languages. The main problem is that current declarative syntax definition techniques are based on pure context-free grammars, while many constructs found in programming languages require context information.

In this chapter we propose a parsing framework that embraces context information in its core. Our framework is based on data-dependent grammars, which extend context-free grammars with arbitrary computation, variable binding and constraints. We present an implementation of our framework on top of the Generalized LL (GLL) parsing algorithm, and show how common idioms in syntax of programming languages such as (1) lexical disambiguation filters, (2) operator precedence, (3) indentation-sensitive rules, and (4) conditional preprocessor directives can be mapped to data-dependent grammars. We demonstrate the initial experience with our framework, by parsing more than 20 000 Java, C#, Haskell, and OCaml source files.

[1] This chapter was originally published as: A. Afroozeh and A. Izmaylova. One Parser to Rule Them All. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! '15, pages 151–170. ACM, 2015.

3.1 Introduction

Parsing is a well-researched topic in computer science, and it is common to hear from fellow researchers in the field of programming languages that parsing is a *solved* problem. This statement mostly originates from the success of Yacc [44] and its underlying theory that has been developed in the 70s. Since Knuth's seminal paper on LR parsing [54], and DeRemer's work on practical LR parsing (LALR) [17], there is a linear parsing technique that covers most syntactic constructs in programming languages. Yacc, and its various ports to other languages, enabled the generation of efficient parsers from a BNF specification. Still, research papers and tools on parsing in the last four decades show an ongoing effort to develop new parsing techniques.

A central goal in research in parsing has been to enable language engineers (i.e., language designers and tool builders) to *declaratively* build a parser for a real programming language from an (E)BNF-like specification. Nevertheless, still today, many parsers are hand-written or are only partially generated and include many hacks to deal with peculiarities in programming languages. The reason is that grammars of programming languages in their simple and readable form are often not deterministic and also often ambiguous. Moreover, many constructs found in programming languages are not context-free, e.g., indentation rules in Haskell. Parser generators based on pure context-free grammars cannot natively deal with such constructs, and require ad-hoc extensions or hacks in the lexer. Therefore, additional means are necessary outside of the power of context-free grammars to address these issues.

General parsing algorithms [18,78,85] support all context-free grammars, therefore the language engineer is not limited by a specific deterministic class, and there are known declarative disambiguation constructs to address the problem of ambiguity in general parsing [34,90,95]. However, implementing disambiguation constructs is notoriously hard and requires thorough knowledge of the underlying parsing technology. This means that it is costly to declaratively build a parser for a given programming language in the wild if the required disambiguation constructs are not already supported. Perhaps surprisingly, examples of such languages are not only the legacy languages, but also modern languages such as Haskell, Python, OCaml and C#.

In this chapter we propose a parsing framework that is able to deal with many challenges in parsing existing and new programming languages. We embrace the need for context information at runtime, and base our framework on data-dependent grammars [41]. Data-dependent grammars are an extension of context-free grammars that allow arbitrary computation, variable binding and constraints. These features allow us to simulate hand-written parsers and to implement disambiguation constructs.

To demonstrate the concept of data-dependent grammars we use the IMAP protocol [62]. In network protocol messages it is common to send the length of data before the actual data. In IMAP, these messages are called literals, and are described by the following (simplified) context-free rule:

```
L8 ::= '~{' Number '}' Octets
```

Here `Octets` recognizes a list of octet (any 8-bit) values. An example of `L8` is `~{6}aaaaaa`. As can be seen, there is no data dependency in this context-free grammar, but the

IMAP specification says that the number of `Octets` is determined by the value parsed by `Number`. Using data-dependent grammars, we can specify such a data-dependency as:

```
L8 ::= '~{' nm:Number {n=toInt(nm.yield)} '}' Octets(n)

Octets(n) ::= [n > 0] Octets(n - 1) Octet
            | [n == 0] ϵ
```

In the data-dependent version, `nm` provides access to the value parsed by `Number`. We retrieve the substring of the input parsed by `Number` via `nm.yield` which is converted to integer using `toInt`. This integer value is bound to variable `n`, and is passed to `Octets`. `Octets` takes an argument that specifies the number of iterations. Conditions `[n > 0]` and `[n == 0]` specify which alternative is selected at each iteration.

It is possible to parse IMAP using a general parser, and then remove the derivations that violate data dependencies post parse. However, such an approach would be slow. Without enforcing the dependency on the length of `Octets` during parsing, given the nondeterministic nature of general parsing, all possible lengths of `Octets` will be tried.

There are many common grammar and disambiguation idioms that can be desugared into data-dependent grammars. Examples of these idioms are operator precedence, longest match, and the offside rule. Expecting the language engineer to write low-level data-dependent grammars for such cases would be wasteful. Instead, we describe a number of such idioms, provide high-level notation for them and their desugaring to data-dependent grammars. For example, using our high-level notation the indentation rules in Haskell can be expressed as follows:

```
Decls ::= align (offside Decl)*
        | ignore('{' Decl (';' Decl)* '}')
```

This definition clearly and concisely specifies that either all declarations in the list are aligned, and each `Decl` is offsided with regard to its first token (first alternative), or indentation is ignored inside curly braces (second alternative).

Our vision is a parsing framework that provides the right level of abstraction for both the language engineer, who designs new languages, and the tool builder, who needs a parsing technology as part of her toolset. From the language engineer's perspective, our parsing framework provides an out of the box set of high-level constructs for most common idioms in syntax of programming languages. The language engineer can also always express her needs directly using data-dependent grammars. From the tool builder's perspective, our framework provides open, extensible means to define higher-level syntactic notation, without requiring knowledge of the internal workings of a parsing technology.

The contributions of this chapter are:

- We provide a unified perspective on many important challenges in parsing real programming languages.
- We present several high-level syntactical constructs, and their mappings to data-dependent grammars.

- We provide an implementation of data-dependent grammars on top of the Generalized LL (GLL) parsing algorithm [78] that runs over ATN grammars [100]. The implementation is part of the Iguana parsing framework[2].
- We demonstrate the initial results of our parsing framework, by parsing 20363 real source files of Java, C#, Haskell (91% success rate), and excerpts from OCaml.

The rest of this chapter is organized as follows. In Section 3.2 we describe the landscape of parsing programming languages. In Section 3.3 we present data-dependent grammars, our high-level syntactic notation, and the mapping to data-dependent grammars. Section 3.4 discusses the extension of GLL parsing with data-dependency. In Section 3.5 we demonstrate the initial results of our parsing framework using grammars of real programming languages. We discuss related work in Section 3.6. A conclusion and discussion of future work is given in Section 3.7.

3.2 The Landscape of Parsing Programming Languages

In this section we discuss well-known features of programming languages that make them hard to parse. These features motivate our design decisions.

3.2.1 General Parsing for Programming Languages

Grammars of programming languages in their natural form are not deterministic, and are often ambiguous. A well-known example is the `if-then-else` construct found in many programming languages. This construct, when written in its natural form, is ambiguous (the dangling-else ambiguity), and therefore cannot be deterministic. Some nondeterministic (and ambiguous) syntactic constructs, such as `if-then-else`, can be rewritten to be deterministic and unambiguous. However, such grammar rewriting in general is not trivial, the resulting grammar is hard to read, maintain and evolve, and the resulting parse trees are different from the original ones the grammar writer had in mind.

Instead of rewriting a grammar, it is common to use an ambiguous grammar, and rely on some implicit behavior of a parsing technology for disambiguation. For example, the dangling-else ambiguity is often resolved using a longest match scheme provided by the underlying parsing technology. Relying on implicit behavior of a parsing technology to achieve determinism can make it quite difficult to reason about the accepted language. Seemingly correct sentences may be rejected by the parser because at a nondeterministic point, a wrong path was chosen. For example, Yacc is an LALR parser generator, but can accept any context-free grammar by automatically resolving *all* shift/reduce and reduce/reduce conflicts. Using Yacc, the language engineer should manually check the resolved conflicts in case of unexpected behavior.

A common theme in research in parsing has been to increase the recognition power of deterministic parsing techniques such as LL(k) or LR(k). One of the widely used

[2] `https://github.com/iguana-parser`

general parsing techniques for programming languages is the Generalized LR (GLR) algorithm [85]. GLR parsers support all context-free grammars and can produce a parse forest containing all derivation trees in form of a Shared Packed Parse Forest (SPPF) in cubic time and space [80]. Note that the cubic bound is for the worst-case, highly ambiguous grammars. As GLR is a generalization of LR, a GLR parser runs linearly on LR parts of the grammar, and as the grammars of real programming languages are in most parts near deterministic, one can expect near-linear performance using GLR for parsing programming languages. GLR parsing has successfully been used in source code analysis and developing domain-specific languages [34].

General parsing enables the language engineer to use the most natural version of a grammar, but leaves open the problem of ambiguity. In declarative syntax definition [34, 49], it is common to use declarative disambiguation constructs, e.g., for operator precedence or the longest match. As a general parser is able to return all ambiguities in form of a parse forest, it is possible to apply the disambiguation rules post-parse, removing the undesired derivations from the parse forest. However, such post-parse disambiguation is not practical in cases where the grammar is highly ambiguous. For example, parsing expression grammars without applying operator precedence during parsing is only limited to small inputs. Therefore, it is required to resolve ambiguity while parsing to achieve near-linear performance.

Implementing disambiguation mechanisms that are executed during parsing is difficult. This is because the implementation of such disambiguation mechanisms requires knowledge of the internal workings of a parsing technology. Therefore, the choice of the general parsing technology becomes very important when considering parsing programming languages. For example, GLR parsers operate on LR automata, and have a rather complicated execution model, as a parsing state corresponds to multiple grammar positions.

The Generalized LL (GLL) parsing algorithm [78] is a new generalization of recursive-descent parsing that supports all context-free grammars, including left recursive ones. GLL parsers produce a parse forest in cubic time and space in the worst case, and are linear on LL parts of the grammar. GLL parsers are attractive because they have the close relationship with the grammar that recursive-descent parsers have. From the end user's perspective, GLL parsers can produce better error messages, and can be debugged in a programming language IDE.

To deal with left recursive rules and to keep the cubic bound, a GLL parser uses a GSS to handle multiple call stacks. While the execution model of a GLL parser is close to recursive-descent parsing, the underlying machinery is much more complicated, and still an in-depth knowledge of GLL is required to implement disambiguation constructs. In this chapter, we propose a parser-independent framework for parsing programming languages based on data-dependent grammars. We use GLL parsing as the basis for our data-dependent parsing framework, as it allows an intuitive way to implement components of data-dependent grammars, such as environment threading, and enables an implementation that is very close to the stack-evaluation based semantics of data-dependent grammars [41].

3.2.2 On the Interaction between Lexer and Parser

Conventional parsing techniques use a separate lexing phase before parsing to transform a stream of characters to a stream of tokens. In particular, whitespace and comments are discarded by the lexer to reduce the number of lookahead in the parsing phase, and enable deterministic parsing.

The main problem with a separate lexing phase is that without having access to the parsing context, i.e., the applicable grammar rules, the lexer cannot unambiguously determine the type of some tokens. An example is `>>` that can either be parsed as a right shift operator, or two closing angle brackets of a generic type, e.g., `List<List<String>>` in Java. Some handwritten parsers deal with this issue by rewriting the token stream. For example, when the `javac` parser reads a `>>` token and is in a parsing state that expects only one `>`, e.g., when matching the closing angle bracket of a generic type, it only consumes the first `>` and puts the second one back to prevent a parse error when matching the next angle bracket.

To resolve the problems of a separate lexing phase, we need to expose the parsing context to the lexer. To achieve this, the separate lexing phase is abandoned, and the lexing phase is effectively integrated into the parsing phase. We call this model *single-phase* parsing. There are two options to achieve *single-phase parsing*. The first option is called *scannerless* parsing [73,95] where lexical definitions are treated as context-free rules. In scannerless parsing, grammars are defined to the level of characters. The second option is *context-aware* scanning [91], where the parser calls the lexer on demand. At each parsing state, the lexer is called with the expected set of terminals at that state.

In almost all modern programming languages longest match (maximal munch) is applied, and keywords are excluded from being recognized as identifiers. These disambiguation rules are conventionally embedded in the lexer. In *single-phase parsing*—scannerless or context-aware—longest match and keyword exclusion have to be applied during parsing, by using lexical disambiguation filters such as follow restrictions [73,90]. These disambiguation filters have parser-specific implementations [90,95]. In Chapter 2 we showed how lexical disambiguation filters can be implemented in the context of GLL parsing. In Section 3.3 we show how these filters can be mapped to data-dependent grammars. Also note that although a context-aware scanner employs longest match, for example by implementing the Kleene star ($*$) as a greedy operator, in some cases we still need to use explicit disambiguation filters, see Section 3.3.2.

3.2.3 Operator Precedence

Expressions are an integral part of virtually every programming language. In reference manuals of programming languages it is common to specify the semantics of expressions using the priority and associativity of operators. However, the implementation of expression grammars can considerably deviate from such precedence specification.

It is possible to encode operator precedence by rewriting the grammar: a new nonterminal is created for each precedence level. The rewriting is not trivial for real programming languages, and the resulting grammar becomes large. This rewriting

```
E ::= '-' E
    | E '*' E
    | E '+' E
    | 'a'
```

```
E ::= E '+' T
    | T
T ::= T '*' F
    | F
F ::= '-' F
    | 'a'
```

```
E  ::= T E1
E1 ::= '+' T E1 | ϵ
T  ::= F T1
T1 ::= '*' F | ϵ
F  ::= '-' F
     | 'a'
```

Figure 3.1: Three grammars that accept the same language: the natural, ambiguous grammar (left), the grammar with precedence encoding (middle), and the grammar after left-recursion removal (right).

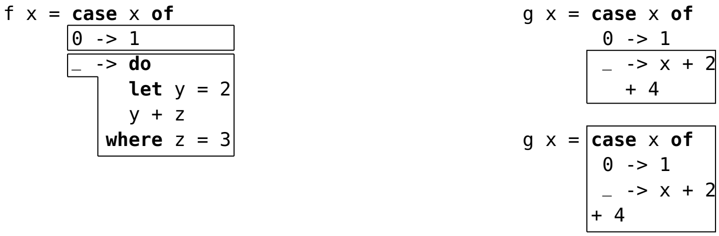
is particularly problematic in parsing techniques that do not support left recursion. The left-recursion removal transformation disfigures the grammar and adds extra complexity in transforming the trees to the intended ones. Figure 3.1 shows three versions of the same expression grammar.

In the 70s, Aho *et al.* [7] presented a technique in which a parser is constructed from an ambiguous expression grammar accompanied with a set of precedence rules. This work can be seen as the starting point for declarative disambiguation using operator precedence rules. Aho *et al.*'s approach is implemented by modifying LALR parse tables to resolve shift/reduce conflicts based on the operator precedence. However, the semantics of operator precedence in this approach is bound to the internal workings of LR parsing. There have been other solutions to build parsers from declarative operator precedence which we discuss in Section 3.6. In Section 3.3.4 we provide a mapping from operator precedence rules to data-dependent grammars.

3.2.4 Offside Rule

In most programming languages, indentation of code blocks does not play a role in the syntactic structure. Rather, explicit delimiters, such as `begin` and `end` or `{` and `}` are used to specify blocks of statements. Landin introduced the *offside* rule [55], which serves as a basis for indentation-sensitive languages. The offside rule says that all the tokens of an expression should be indented to the right of the first token. Haskell and Python are two examples of popular programming languages that use a variation of the offside rule.

Figure 3.2 shows two examples of the offside rule in Haskell. The keywords `do`, `let`, `of`, and `where` signal the start of a block where the starting tokens of the statements should be aligned, and each statement should be offsided with regard to its first token. In Figure 3.2 (left), `case` has two alternatives which are aligned, and the second alternative that spans several lines is offsided with regard to its first token, i.e., `_`. Figure 3.2 (right) shows two examples that look the same, but the indentation of the last part, `+ 4`, is different. In the top declaration `+ 4` belongs to the last alternative, but in the bottom declaration, `+ 4` belongs to the expression on the right hand side of `=`.



```
f x = case x of
        0 -> 1
        _ -> do
             let y = 2
             y + z
           where z = 3
```

```
g x = case x of
         0 -> 1
         _ -> x + 2
           + 4
```

```
g x = case x of
         0 -> 1
         _ -> x + 2
        + 4
```

Figure 3.2: Examples of indentation rules in Haskell.

Indentation sensitivity in programming languages cannot be expressed by pure context-free grammars, and has often been implemented by hacks in the lexer. For example, in Haskell and Python, indentation is dealt with in the lexing phase, and the context-free part is written as if no indentation-sensitivity exists. Both GHC and CPython, the popular implementations of Haskell and Python, use LALR parser generators. In Python, the lexer maintains a stack and emits `INDENT` and `DEDENT` tokens when indentation changes. In Haskell, the lexer translates indentation information into curly braces and semicolons based on the rules specified by the L function [60].

In Section 3.3.5 we show how data-dependent grammars can be used for single-phase parsing of indentation-sensitive programming languages in a declarative way. As data-dependent grammars are rather low-level for such solutions, we introduce three high-level constructs: **`align`**, **`offside`**, and **`ignore`** which are desugared to data-dependent grammars.

3.2.5 Conditional Directives

Many programming languages allow compiler *pragmas* that specify how the compiler (or the interpreter) processes parts of the input. The C family of programming languages, i.e., C, C++ and C#, allow preprocessor directives such as `#if` and `#define`. GHC also allows various compiler pragmas [60, §12.3]. For example, it is possible to enable C preprocessor directives in Haskell using `{-# LANGUAGE CPP #-}`.

Preprocessor directives pose considerable difficulty in parsing programming languages. The main reason is that they are not part of the grammar of a language, but can appear anywhere in the source code. In this regard, preprocessor directives are similar to whitespace and comment. However, conditional directives may affect the syntactic structure of a program, and cannot be simply ignored as a special kind of whitespace. This is especially important if we consider single-phase parsing where no lexing/preprocessing is available. We need a mechanism to allow the parser to switch between the preprocessor mode and main grammar, and to evaluate conditional directives to select the right branch.

Figure 3.3 (left) shows a C# example where ignoring the conditional directive will lead to a parse error, as the closing bracket of the `test` method is in the conditional directive, and one of the branches should be included in the input. The

```
void test()
{
#if Debug
  System.Console.WriteLine("Debug")
}
#else
}
#endif
```

```
#if X
/*
#else
/* */ class Q { }
#endif
```

Figure 3.3: Problematic cases of using C# directives [63].

```
   static void Main() {
      System.Console.WriteLine(@"hello,
#if Debug
      world
#else
      Nebraska
#endif
      ");
   }
```

Figure 3.4: C# multi-line string containing directives [63].

example in Figure 3.3 (right) shows another aspect of directives in C#. If `X` is true, `"/* #else /* */ class Q {}"` will be considered as part of the source code. If `X` is false, only the else-part will be considered: `"/* */ class Q {}"`. Note that when `X` is false, the if-part does not have to be syntactically correct, in this case an unclosed multi-line comment.

Among the family of C languages we selected C#, as parsing C# is more manageable. The problematic part of parsing C with directives is textual macros. Without a preprocessor to expand macros before parsing, we need to deal with macros at runtime. Parsing C without a preprocessor is future work. In C#, `#define` does not define a macro, rather it only sets a boolean variable. It should also be noted that C# supports multi-line strings, where directives should not be processed. Figure 3.4 shows a C# example that uses conditional directives in a multi-line string. In single-phase parsing, however, multi-line strings are not a problem, as we effectively parse each terminal in the context where it appears.

3.2.6 Miscellaneous Features

There are many other peculiarities in programming languages and data formats that cannot be expressed by context-free grammars. There has been considerable effort to build declarative parsers for data formats, e.g. PADS [22], and one of the main motivations for data-dependent grammars [41] is indeed to enable parsing data formats.

Examples of languages that require data-dependent parsing are data protocols, such as IMAP and HTTP, and tag-based languages such as XML. In programming languages, data-dependent grammars can be used to implement some language-specific disambiguation mechanisms. For example, to maintain a table of type definitions in C to allow resolving the infamous typedef ambiguity, e.g., in `x * y` which can be either interpreted as a variable of pointer type `x` or as a multiplication, depending on the type of `x`. We give an example of parsing XML and resolving the typedef ambiguity in C in Section 3.3.7.

3.3 Parsing Programming Languages with Data-dependent Grammars

In this section we describe data-dependent grammars [41], discuss our single-phase parsing strategy, and demonstrate how various high-level, declarative syntax definition constructs can be desugared into data-dependent grammars.

3.3.1 Data-dependent Grammars

Data-dependent grammars are defined as an extension of *context-free grammars* (CFGs), where a CFG is, as usual, a tuple (N, T, P, S) where

- N is a finite set of nonterminals;
- T is a finite set of terminals;
- P is a finite set of rules. A rule (production) is written as $A ::= \alpha$, where A (head) is a nonterminal, and α (body) is a string in $(N \cup T)^*$;
- $S \in N$ is the start symbol of the grammar.

We use A, B, C to range over nonterminals, and a, b, c to range over terminals. We use α, β, γ for a possibly empty sequence of terminals and nonterminals, and ϵ represents the empty sequence. It is common to group rules with the same head and write them as $A ::= \alpha_1 \,|\, \alpha_2 \,|\, \ldots \,|\, \alpha_n$. In this representation, each α_i is an alternative of A.

Data-dependent grammars introduce parametrized nonterminals, arbitrary computation via an expression language, constraints, and variable binding. Here, we assume that the expression language e is a simple functional programming language with immutable values and no side-effects. In a data-dependent grammar a rule is of the from $A(p) ::= \alpha$, where p is a formal parameter of A. Here, for simplicity of presentation and without loss of generality, we assume that a nonterminal can have at most one parameter. The body of a rule, α, can now contain the following additional symbols:

- $x = l : A(e)$ is a labeled call to A with argument e, label l, and variable x bound to the value returned by $A(e)$;
- $l : a$ is a labeled terminal a with label l;
- $[e]$ is a constraint;
- $\{x = e\}$ is a variable binding;

- $\{e\}$ is a return expression (only as the last symbol in α);
- $e \,?\, \alpha : \beta$ is a conditional selection.

The symbols above are presented in their general forms. For example, labels, variables to hold return values, and return expressions are optional.

Our data-dependent grammars are very similar to the ones introduced in [41] with four additions. First, terminals and nonterminals can be labeled, and labels refer to properties associated with the result of parsing a terminal or nonterminal. These properties are the start input index (or left extent), the end input index (or right extent), and the parsed substring. Properties can be accessed using dot-notation, e.g., for labeled nonterminal $b\!:\!B$, $b.l$ gives the left extent, $b.r$ the right extent, and $b.yield$ the substring.

Second, nonterminals can return arbitrary values (return expressions) which can be bound to variables. In several cases, we found this feature very practical as we could express data dependency without changing the shape of the original specification grammar. Specifically, cases where a global table needs to be maintained along a parse (C# conditional directives discussed in Section 3.3.6 and C typedef declarations in Section 3.3.7), or where semantic information needs to be propagated upwards from a complicated syntactic structure (`Declarator` of the C grammar in Section 3.3.7). In some cases a data-dependent grammar that uses return values can be rewritten to one without return values. However, in general, whether return values enlarge the class of languages expressible with the original data-dependent grammars is an open question for future work.

Third, we support regular expression operators (EBNF constructs): $*$, $+$, and $?$, by desugaring them to data dependent rules as follows: $A* ::= A+ \mid \epsilon$; $A+ ::= A+\, A \mid A$; and $A? ::= A \mid \epsilon$. In the data-dependent setting, this translation must also account for variable binding. For example, if symbol $([e]\ A)*$ appears in a rule, and x is a free variable in e, captured from the scope of the rule, our translation lifts this variable, introducing a parameter x to the new nonterminal. In addition, EBNF constructs introduce new scopes: variables declared inside an EBNF construct, e.g., $(l\!:\!A\ [e])*$, are not visible outside, e.g., in the rule that uses it.

Finally, we also introduce a conditional selection symbol $e\,?\,\alpha\!:\!\beta$, which selects α if e is evaluated to true, otherwise β, i.e., introduces deterministic choice. Similar to EBNF constructs, we implement conditional selection by desugaring it into a data-dependent grammar. For example, $A ::= \alpha\ e\,?\,X : Y\beta$ is translated to $A ::= \alpha\ C(e)\ \beta$, where $C(b) ::= [b]\ X \mid [!b]\ Y$. We illustrate use of the conditional selection when discussing C# directives in Section 3.3.6.

3.3.2 Single-phase Parsing Strategy

We implement our data-dependent grammars on top of the generalized LL (GLL) parsing algorithm [78]. As general parsers can deal with any context-free grammar, lexical definitions can be specified to the level of characters. For example, comment in the C# specification [63] is defined as shown in Figure 3.5. Such character-level grammars, however, lead to very large parse forests. These parse forests reflect the

```
Comment ::= SingleLineComment
          | DelimitedComment

SingleLineComment ::= "//" InputCharacter*

InputCharacter ::= ![\r \n]

DelimitedComment ::= "/*" DelimitedCommentSect* [*]+ "/"

DelimitedCommentSect ::= "/" [*]* NotSlashOrAsterisk

NotSlashOrAsterisk ::= ![/ *]
```

Figure 3.5: Comment definition in C#.

full structure of lexical definitions, which are not needed in most cases. We provide the option to use an on-demand context-aware scanner, where terminals are defined using regular expression. For example, `Comment` in C# can be compiled to a regular expression. In cases where the structure is needed, or it is not possible to use a regular expression, e.g., recursive definitions of nested comments, the user can use character-level grammars.

Our support for context-aware scanning borrows many ideas from the original work by Van Wyk and Schwerdfeger [91], but because of the top-down nature of GLL parsing, there are some differences. The original context-aware scanning approach [91] is based on LR parsing, and as each LR state corresponds to multiple grammar rules, there may be several terminals that are valid at a state. The set of valid terminals in a parsing state is called *valid lookahead* set [91]. In GLL parsing, in contrast, the parser is at a single grammar position at each time, i.e., either before a nonterminal or before a terminal in a single grammar rule. Therefore, in GLL parsing, the valid lookahead set of a terminal grammar position contains only one element, which allows us to directly call the matcher of the regular expression of that terminal.

We use our simple context-aware scanning model for better performance, see Section 3.5.1. The implementation of the context-aware scanner in [91] is more sophisticated. The scanner is composed of all terminal definitions, as a composite DFA. This enables a longest match scheme across terminals in the same context, for example in programming languages where one terminal is a prefix of another, e.g., `'fun'` and `'function'` in OCaml. To enforce longest match across terminals we use follow/precede restrictions, in this case a follow restriction on `'fun'` or a precede restriction on identifiers. Moreover, keyword reservation in [91] is done by giving priority to keywords at matching states of the composite DFA. In our model, keyword exclusion should be explicitly applied in the grammar rules using an exclude disambiguation filter. We explain follow/precede restrictions and keyword reservation in Section 3.3.3.

In single-phase parsing layout (whitespace and comments) are treated the same way as other lexical definitions. Because layout is almost always needed in parsing

```
A ::= α B !>> c β          |    A ::= α b:B [input.at(b.r) != c] β
A ::= α c !<< B β          |    A ::= α b:B [input.at(b.l-1) != c] β
A ::= α B \ s β            |    A ::= α b:B [input.sub(b.l,b.r)!= s] β
```

Figure 3.6: Mapping of lexical disambiguation filters.

programming languages, we support automatic layout insertion into the rules. There are two approaches to deal with layout insertion: a layout nonterminal can be inserted exactly before or after each terminal node [47, 91]. Another way is to insert layout between the symbols in a rule, like in SDF [34]. We use SDF-style layout insertion: if $X ::= x_1 x_2 \ldots x_n$ is a rule, and L is a nonterminal defining layout, after the layout insertion, the rule becomes $X ::= x_1 L\, x_2\, L \ldots L\, x_n$. A benefit of SDF-style layout insertion is that no symbol definition accidentally ends or starts with layout, provided that the layout is defined greedily (see Section 3.3.3). This is helpful when defining the offside rule (see Section 3.3.5).

3.3.3 Lexical Disambiguation Filters

Common lexical disambiguation filters [90], such as follow restrictions, precede restrictions and keyword exclusion, can be mapped to data-dependent grammars without further extensions to the parser generator or parsing algorithm. These disambiguation filters are common in scannerless parsing [73] and have been implemented for various generalized parsers [90, 95].

A follow restriction (`!>>`) specifies which characters cannot immediately appear after a symbol in a rule. This restriction is used to locally define longest match (as opposed to a global longest match in the lexer). For example, to enforce longest match on identifiers we write `Id ::= [A-Za-z]+ !>> [A-Za-z]`. A precede restriction (`!<<`) is similar to a follow restriction, but specifies the characters that cannot immediately precede a symbol in a rule. Precede restrictions can be used to implement longest match on keywords. For example, `[A-Za-z] !<< Id` disallows an identifier to start immediately after a character. This disallows, for example, to recognize `intx` as the keyword `'int'` followed by the identifier `x`. Finally, exclusion (\) is usually used to implement keyword reservation. For example, `Id \'int'` excludes the keyword `int` from beging recognized as `Id`.

Figure 3.6 shows the mapping from character-level disambiguation filters to data-dependent grammars. The mapping is straightforward: each restriction is translated into a condition that operates on the input. A note should be made regarding the condition implementing precede restrictions. This condition only depends on the left extent, `b.l`, that permits its application before parsing `B`. We consider this optimization in the implementation of our parsing framework, permitting application of such conditions before parsing labeled nonterminals or terminals.

The restrictions of Figure 3.6 are just examples and can be extended in many ways. For example, instead of defining the restriction using a single character, we can use

```
E ::= '-' E
    | E '*' E
    | E '+' E
    | 'if' E 'then' E 'else' E
    | a
```

```
E ::= '-' E
    > E '*' E      left
    > E '+' E      left
    > 'if' E 'then' E 'else' E
    | a
```

Figure 3.7: An ambiguous expression grammar (left), and the same grammar disambiguated with `>` and **`left`** (right).

regular expressions or character classes. One can also define similar restrictions for related disambiguation purposes. For example, consider the cast expression in C#:

```
cast-exp ::= '(' type ')' unary-exp
```

An expression such as `(x)-y` is ambiguous, and can be interpreted as either a type cast of `-y` to the type `x`, or a subtraction of `y` from `(x)`. In the C# language specification, it is stated that this ambiguity is resolved during parsing based on the character that comes after the closing parentheses: if the character following the closing parentheses is ∼, `'!'`, `'('`, an identifier, a literal or keywords, the expression should be interpreted as a cast. We can implement this rule as follows:

```
cast-exp ::= '(' type ')' >>> [∼!(A-Za-z0-9] unary-exp
```

The `>>>` notation specifies that the next character after the closing parentheses should be an element of the specified character class. The implementation of `>>>` is similar to that of `>>` with an additional aspect: it adds the condition on the automatically inserted layout nonterminal after `')'` instead.

These examples show how more syntactic sugar can be added to the existing framework for various common lexical disambiguation tasks in programming languages without changes to the underlying parsing technology.

3.3.4 Operator Precedence and Associativity

Expression grammars in their natural form are often ambiguous. Consider the expression grammar in Figure 3.7 (left). For this grammar, the input string `a+a*a` is ambiguous with two derivation trees that correspond to the following groupings: `(a+(a*a))` and `((a+a)*a)`. Given that `*` normally has higher precedence than `+`, the first derivation tree is desirable. We use `>`, **`left`**, and **`right`** to define priority and left- and right-associativity, respectively. Figure 3.7 (right) shows the disambiguated version of this grammar by specifying `>` and **`left`**, where `-` has the highest precedence, and `*` and `+` are left-associative.

Ambiguity in expression grammars is caused by derivations from the left- or right-recursive ends in a rule, i.e., $E ::= \alpha\underline{E}$ and $E ::= \underline{E}\beta$. We use `>`, **`left`**, and **`right`** to specify which derivations from the left- and right-recursive ends are not valid with respect to operator precedence. For example, `E ::= '-' E > E '*' E` specifies that `E`

```
E ::= E '*' E   left
    > E '+' E   left
    | '(' E ')'
    | a
```

```
E(p) ::= [2 >= p] E(2) * E(3)  //2
       | [1 >= p] E(0) + E(2)  //1
       | '(' E(0) ')'
       | a
```

Figure 3.8: An expression grammar with `>` and **`left`** (left), and its translation to data-dependent grammars (right).

in the `'-'`-rule (parent) should not derive the `'*'`-rule (child). The `>` construct only restricts the right-recursive end of a parent rule when the child rule is left-recursive, and vice versa. For example, in Figure 3.7 (right) the right `E` in the `'+'`-rule is not restricted because the `'if'`- rule is not left-recursive. This is to avoid parse error on inputs that are not ambiguous, e.g., `a + if a then a else a`. Note that `'if' E 'then' E 'else'` in the `'if'`-rule acts as a unary operator. In addition, the `>` operator is transitive for all the alternatives of an expression nonterminal. Finally, **`left`** and **`right`** only affect binary recursive rules and only at the left- and right-recursive ends.

Although `>` is defined as a relationship between a parent rule and a child rule, its application may need to be arbitrary deep in a derivation tree. For example, consider the input string `a * if a then a else a + a` for the grammar in Figure 3.7 (right). This sentence is ambiguous with two derivation trees that correspond to the following groupings:

```
(a * (if a then a else a)) + a
 a * (if a then a else (a + a))
```

The first grouping is not valid as `'if'` binds stronger than `'+'`, but we defined `'+'` to have higher priority than `'if'`. This example shows that restricting derivations only at one level cannot disambiguate such cases. A correct implementation of `>` thus also restricts the derivation of the `'if'`-rule from the right-recursive end of the `'*'`-rule if the `'*'`-rule is derived from the left-recursive end of the `'+'`-rule.

We now show how to implement an operator precedence disambiguation scheme using data-dependent grammars. We first demonstrate the basic translation scheme using binary operators only, and then discuss the translation of the example in Figure 3.7. Figure 3.8 (left) shows a simple example of an expression grammar that defines two left-associative binary operators `*` and `+`, where `*` is of higher precedence than `+`. Figure 3.8 (right) shows the result of the translation into the data-dependent counterpart. The basic idea behind the translation is to assign a number, a *precedence level*, to each left- and/ or right recursive rule of nonterminal `E`, to parameterize `E` with a precedence level, and based on the precedence level passed to `E`, to exclude alternatives that will lead to derivation trees that violate the operator precedence.

In Figure 3.8 (right) each left- and right-recursive rule in the grammar gets a precedence level (shown in comments), which is the reverse of the alternative number in the definition of `E`. The precedence counter starts from 1 and increments for each encountered `>` in the definition. The number 0 is reserved for the unrestricted use of `E`,

```
E(l,r) ::= [4 >= l] '-' E(l,4)                                //4
         | [3 >= r, 3 >= l] E(3,3) '*' E(l,4)                 //3
         | [2 >= r, 2 >= l] E(2,2) '+' E(l,3)                 //2
         | [1 >= l] 'if' E(0,0) 'then' E(0,0) 'else' E(0,0)   //1
         | a
```

Figure 3.9: Operator precedence with data-dependent grammars (binary and unary operators).

illustrated using the round bracket rule. Nonterminal `E` gets parameter `p` to pass the precedence level, and for each left- and right-recursive rule, a predicate is added at the beginning of the rule to exclude rules by comparing the precedence level of the rule with the precedence level passed to the parent `E`. Finally, for each use of `E` in a rule, an argument is passed.

In the `'*'`-rule, its precedence level (`2`) is passed to the left `E`, and its precedence level plus one (`3`) is passed to the right `E`. This allows to exclude the rules of lower precedence from the left `E`, and to exclude the rules of lower precedence and the `'*'`-rule itself from the right `E`. Excluding the `'*'`-rule itself allows only the left-associative derivations, e.g., `(a*a)*a`, as specified by **`left`**. In the `'+'`-rule, its precedence level plus one (`2`) is passed to the right `E`, excluding the `'+'`-rule. The value `0` is passed to the left `E`, permitting any rule. Note that passing `0` instead of `1` to the left `E` of the `'+'`-rule achieves the same effect but enables better sharing of calls to `E`, as the sharing of calls (using GSS) is done based on the name of the nonterminal and the list of arguments. In the round bracket rule, `0` is passed to `E` as the use of `E` is neither left- nor right-recursive, hence, the precedence does not apply.

Now we discuss the translation of the example shown in Figure 3.7 that contains both binary and unary operators. For this we need to distinguish between the rules that should be excluded from the left and from the right `E`. This is achieved as follows. First, `E` gets two parameters, `l` and `r` (Figure 3.9), to distinguish between the precedence level passed from left and right, respectively. Second, a separate condition on `l` is added to a rule when the rule can be excluded from the right `E` (i.e., rules for binary operators and unary postfix operators). A separate condition on `r` is added to a rule when the rule can be excluded from the left `E` (i.e., rules for binary operators and unary prefix operators). Third, `l`- and `r`-arguments are determined for the left and right `E`'s as follows. An `l`-argument to the left `E` and `r`-argument to the right `E` are determined as in the example of Figure 3.8. For example, `E(3,_) '*' E(_,4)`, where `3` is the precedence level of the `'*'`-rule, and `4` is the precedence level plus one. Note that `r=4` does not exclude the unary operators of `E`. Now, an `l`-argument to the right `E`'s is propagated from the parent `E`. This effectively excludes a unary prefix rule from the right `E`'s when the parent `E` is the left `E` of a rule of higher precedence than the unary operator. Finally, given that there are no unary postfix operators, an `r`-argument to the left `E`'s is not propagated from the parent `E` and can be the same as the respective `l`-argument.

```
Decls ::= align (offside Decl)*
        | ignore('{' Decl (';' Decl)* '}')

Decl  ::= FunLHS RHS

RHS   ::= '=' Exp 'where' Decls
```

Figure 3.10: Simplified version of Haskell's `Decls`.

We have also extended this approach for grammars that allow rules of the same precedence and/or associativity groups. For example, binary `+` and `-` operators have the same precedence, but are left-associative with respect to each other.

Our translation of operator precedence to data-dependent grammars resembles the precedence climbing technique [14,70]. In contrast to precedence climbing that requires a non-left recursive grammar, our approach works in presence of both left- and right-recursive rules.

3.3.5 Indentation-sensitive Constructs

In this section we show how the offside rule can be translated into data-dependent grammars. We use Haskell as the running example, but our approach is also applicable for other programming languages that implement the offside rule.

In Haskell, one can write a `where` clause consisting of a block of declarations, where the structure of the block is defined by using either explicit delimiters or indentation (column number). For example, the structure of the following blocks, one written with explicit delimiters, such as curly braces and semicolons (left), and the other written using indentation (right), is the same:

```
{ x = 1 * 2 + 3; y = x + 4 }          x = 1 * 2
                                            + 3
                                      y = x + 4
```

Figure 3.10 shows a simplified excerpt of the Haskell grammar, defined using our parsing framework. The first alternative explicitly enforces indentation constraints on a declaration block. First, it requires that all declarations of a block are aligned (**align**) with respect to each other, i.e., each declaration starts with the same indentation. Second, it requires that the offside rule applies to each declaration, i.e., all non-whitespace tokens of a declaration are strictly indented to the right of its first non-whitespace token. In contrast, the second alternative of `Decls` enforces the use of curly braces and semicolons, and explicitly ignores (**ignore**) indentation constraints even when imposed by an outer scope.

In our meta-notation, **align** only affects regular definitions (EBNF constructs) such as lists and sequences, **offside** affects nonterminals, and **ignore** applies to a sequence of symbols. The translation of these high-level constructs into data-dependent grammars is illustrated in Figures 3.11 and 3.12.

```
Decls ::= a0:Star1(a0.l)
        | ignore('{' Decl Star2 '}')

Decl  ::= FunLHS RHS

RHS   ::= '=' Exp 'where' Decls

Star1(v) ::= Plus1(v) | ϵ

Plus1(v) ::= offside a1:Decl [col(a1.l) == col(v)]
           | Plus1(v) offside a1:Decl [col(a1.l) == col(v)]

Star2 ::= Plus2 | ϵ

Plus2 ::= Plus2 Seq2 | Seq2

Seq2  ::= ';' Decl
```

Figure 3.11: Desugaring of **align**.

```
Decls(i,fst) ::= a0:Star1(a0.l,i,fst)
               | '{' Decl(-1,0) Star2 '}'

Decl(i,fst)  ::= FunLHS(i,fst) RHS(i,0)

RHS(i,fst)   ::= o0:'=' [f(i,fst,o0.l)] Exp(i,0) o1:'where' [f(i,0,o1.l)] Decls(i,0)

Star1(v,i,fst) ::= Plus1(v,i,fst) | ϵ

Plus1(v,i,fst) ::= Plus1(v,i,fst) a1:Decl(a1.l,1) [col(a1.l) == col(v), f(i,0,a1.l)]
                 | a1:Decl(a1.l,1) [col(a1.l) == col(v), f(i,fst,a1.l)]

Star2 ::= Plus2 | ϵ

Plus2 ::= Plus2 Seq2 | Seq2

Seq2  ::= ';' Decl(-1,0)

f(i,fst,l) = i == -1 || fst == 1 || col(l) > col(i);
```

Figure 3.12: Desugaring of **offside** and **ignore**.

The basic idea of translating **align** is to use the start index of a declaration list, and constrain the start index of each declaration in the list by an equality check on indentation at the respective indices. Figure 3.11 shows the result after first desugaring **align** and then translating EBNF constructs (Section 3.3.1). Desugaring **align** alone results in:

```
Decls ::= a0:(offside a1:Decl [col(a1.l) == col(a0.l)])*
```

Labels `a0` and `a1` are introduced to refer to the start index of a declaration list, `a0.l`, and to the start index of each declaration in the list, `a1.l`, respectively, and the constraint checks whether the respective column numbers (given tabs of 8 characters) are equal. As in case of precede restrictions in Section 3.3.3, this constraint only depends on the start indices and can be applied before parsing `Decl`. The EBNF translation introduces nonterminals for each EBNF construct, where `Star1` and `Plus1` also get parameter `v` as the use of `a0` has to be lifted during the translation.

Figure 3.12 shows the result of desugaring **`offside`** and **`ignore`** from Figure 3.11. The basic idea is to pass down `Decl`'s start index and constrain the indentation of any non-whitespace terminal that can appear under the `Decl`-node, except for the leftmost one, to be greater than the indentation of `Decl`'s start index. Two parameters, `i` and `fst`, are introduced to `Decl` and to all nonterminals reachable from it. The first parameter is used to pass `Decl`'s start index, calculated at the **`offside`** application site (`a1.l`), to any nonterminal reachable from `Decl`, and to constrain terminals reachable from `Decl`.

The second parameter, `fst`, which is either `0` or `1`, is used to identify and skip the leftmost terminal that should not be constrained. The value `1` is passed at the application site of **`offside`** and propagated down to the first nonterminal of each reachable rule if the rule starts with a nonterminal. The value `0` is passed to any other nonterminal of a reachable rule when the first symbol of the rule is not nullable. Our translation also accounts for nullable nonterminals (not shown here), and in such cases the value of `fst` also depends on a dynamic check whether the left and right extents of the node corresponding to a nullable nonterminal are equal.

Finally, each terminal reachable from `Decl` gets a label (labels starting with `o`), to refer to its start index, and a constraint, encoded as a call to boolean function `f`. Note that in the definition of `f`, condition `i == -1` corresponds to the case when `Decl` appears in the context where the offside rule does not apply or is ignored, and condition `fst == 1` to the case of the leftmost terminal.

The **`offside`**, **`align`** and **`ignore`** constructs are examples of reasonably complex desugarings to data-dependent grammars. Their existence and their aptness to describe the syntax of Haskell is a witness of the power of data-dependent grammars and the parsing architecture we propose.

3.3.6 Conditional Directives

In this section we present our solution for parsing conditional directives in C#. As discussed in Section 3.2.5, most directives can be regarded as comment, but conditional directives have to be evaluated during parsing, as they may affect the syntactic structure of a program.

Conditional directives can appear anywhere in a program. Therefore, it is natural to define them as part of the layout nonterminal. Figure 3.13 shows relevant parts of the layout definition (`Layout`)[3] we used to parse C# (follow restrictions enforce longest

[3] For readability reasons, we omit uses of `Whitespace?` (optional whitespace) before and after terminal `'#'`, and uses of `Whitespace` after terminals `'define'`, `'undef'`, `'if'`, `'elif'`.

```
global defs = {}

Layout ::= (Whitespace | Comment | Decl | If | Gbg)* !>> [\ \t\n\r\f] !>> '/*'
                                                      !>> '//' !>> '#'

Decl ::= '#' 'define' id:Id  {defs=put(defs,id.yield,true)} PpNL
       | '#' 'undef' id:Id   {defs=put(defs,id.yield,false)} PpNL

If   ::= '#' 'if' v=Exp(defs) [v] ? Layout : (Skipped (Elif|Else|PpEndif))
Elif ::= '#' 'elif' v=Exp(defs) [v] ? Layout : (Skipped (Elif|Else|PpEndif))
Else ::= '#' 'else' Layout

Gbg     ::= GbgElif* GbgElse? '#' 'endif'
GbgElif ::= '#' 'elif' Skipped
GbgElse ::= '#' 'else' Skipped

Skipped ::= Part+
Part    ::= PpCond | PpLine | ... // etc.
PpCond  ::= PpIf PpElif* PpElse? PpEndif
PpIf    ::= '#' 'if' PpExp PpNL Skipped?
PpElif  ::= '#' 'elif' PpExp PpNL Skipped?
PpElse  ::= '#' 'else' PpNL Skipped?
PpEndif ::= '#' 'endif' PpNL
```

Figure 3.13: The grammar of conditional directives in C#.

match). In addition to whitespace characters (`Whitespace`) and comments (starting with `'/*'` or `'//'`), the layout consists of declaration directives (`Decl`) and conditional directives (`If`).

According to the C# language specification, the scope of symbols introduced by declaration directives `#define` and `#undef` is the file they appear in. Therefore, we need to maintain a global symbol table `defs` to declare (see `Decl`) and access (see `If` and `Elif`) symbol definitions while parsing. In C# one can define/undefine a symbol, but a value cannot be assigned to a symbol. Thus, the symbol table needs to associate a boolean value with a symbol.

To enable global definitions, our parsing framework supports global variables that can be declared using the **`global`** keyword. e.g., `defs` in Figure 3.13. In our parsing framework, a global variable is implemented by using parameters and return values to thread a value through a parse. In this case, each nonterminal that directly or indirectly accesses a global variable should get an extra parameter, and each nonterminal that can directly or indirectly update a global variable should return the new value of the variable if the variable is used after an occurrence of the nonterminal in a rule. Note that, assuming immutable values, such implementation of global variables properly accounts for the nondeterministic nature of generalized parsing. This way updates to a variable made along one parse do not interfere with updates made along an alternative parse.

The basic idea of single-phase parsing of C# in presence of conditional directives is as follows. Recall that in our parsing strategy (Section 3.3.2) layout is inserted between symbols in a grammar rule. Conditional directives are evaluated as part of the layout nonterminal, and based on the result of the evaluation, the next lines of source code are either treated as the actual source code (true case), or as a sequence of valid C# tokens (false case), also consuming directives that should not be evaluated. To achieve this, the grammar of Figure 3.13 uses two different definitions for `#if`, `#elif` and `#else`. The bottom definition (`PpIf`, `PpElif` and `PpElse`), which is found in the C# specification, simply defines directives as part of valid C# tokens (`Skipped`), while the top definition (`If`, `Elif` and `Else`) uses data dependency. Note that conditional directives can be nested. This is expressed by using `Layout` in `If`, `Elif` and `Else`, and `Skipped` in `PpIf`, `PpElif` and `PpElse`.

Whenever an `#if`-directive and its expression are parsed as part of `If`, the expression is evaluated using the symbol table (`defs`). `Exp` (not shown in Figure 3.13) defines a simple boolean expression. To enable evaluation of expressions while parsing, `Exp` uses data dependency and extends the `PpExp` rules, found in the C# specification, with return values and boolean computation. If the expression evaluates to true (note the use of conditional selection), the parser first continues consuming layout, including the nested directives, and then, after no layout can be consumed, the parser returns to the next symbol in the alternative.

If the expression evaluates to false, the parser consumes part of the input as a list of valid C# tokens (`Skipped`) until it finds the corresponding `#elif`-, `#else`- or `#endif`-part. Note that `Skipped` also consumes nested `#if`-directives (`PpCond`), if any, but in this case, conditions are not evaluated. The definition of `Skipped` also allows to consume invalid C# structure (only valid token-wise) when the condition is false, see Figure 3.3 (right). Finally, when all `#if`, `#elif` and `#else` directives are present, there will be dangling `#elif`, `#else`, and `#endif` parts remaining if one of the conditions evaluates to true. These dangling parts should be also consumed by the layout. The `Gbg` (garbage) nonterminal, defined as part of layout, does exactly this.

3.3.7 Miscellaneous Features

In this section we discuss the use of data-dependent grammars for parsing XML and resolving the infamous `typedef` ambiguity in C. XML has a relatively straightforward syntax. Figure 3.14 (top) shows the context-free definition of `Element` in XML, where `Content` allows a list of nested elements. The problem with this definition is that it can recognize inputs with unbalanced start and end tags, for example:

```
<note>
  <to>Bob</from>
  <from>Alice</to>
</note>
```

Using data-dependent grammars, the solution to match start and end tags is very intuitive. Figure 3.14 (bottom) shows a data-dependent grammar for XML elements. As can be seen, inside a starting tag, `STag`, the result of parsing `Name` is bound to `n`, and the respective substring, `n.yield`, is returned from the rule. The returned value is

```
Element ::= STag Content ETag
STag    ::= '<' Name Attribute* '>'
ETag    ::= '</' Name '>'

Element ::= s=STag Content ETag(s)
STag    ::= '<' n:Name Attribute* '>' { n.yield }
ETag(s) ::= '</' n:Name [n.yield == s] '>'
```

Figure 3.14: Context-free grammar of XML elements (top) and the data-dependent version (bottom).

assigned to `s` in the `Element` rule, and is passed to the end tag, `ETag`. Finally, in the `ETag`, the name of the end tag is checked against the name of the starting tag. If the name of the starting tag is not equal to the name of the end tag, i.e., `n.yield == s` does not hold, the parsing pass dies.

Now, we consider the problem of typedef ambiguity in C. For example, expression `(T)+1` can have two meanings, depending on the meaning of `T` in the context: a cast to type `T` with `+1` being a subexpression, or addition with two operands `(T)` and `1`. If `T` is a type, declared using `typedef`, the first parse is valid, otherwise the second one.

To resolve the typedef ambiguity, type names should be distinguished from other identifiers, such as variables and function names, during parsing. In addition, the scoping rules of C should be taken into account. For example, consider the following C program:

```
typedef int T;
main() {
  int T = 0, n = (T)+1;
}
```

In this example, `T` is first declared as a type alias to `int` and then redeclared as a variable of type `int` in the inner scope introduced by the `main` function.

Figure 3.15 shows a simplified excerpt of our data-dependent C grammar. The excerpt shows the declaration and expression parts of the C grammar. As can be seen, a C declaration consists of a list of specifiers followed by a list of declarators. Each declarator declares one identifier. Keyword `typedef` can appear in the list of specifiers, for example, along with the declared type. A declarator can be either a simple identifier or a more complicated syntactic structure, e.g., array and function declarators, nesting the identifier. It is important to note that an identifier should enter the current scope when its declarator is complete. The expression part of Figure 3.15 shows the cast expression rule (the second rule from top), and the primary expression rule (the last one). Note that to resolve the typedef ambiguity, illustrated in our running example, an identifier should be accepted as an expression if it is not declared as a type name.

To distinguish between type names and other identifiers, we record names, encountered as part of declarators, and associate a boolean value with each name: `true` for type names and `false` otherwise. To maintain this information during parsing, we

```
global defs = [{}]

Declaration ::= x=Specifiers Declarators(x)

Specifiers ::= x=Specifier y=Specifiers {x || y}
             | x=Specifier {x}

Specifier  ::= "typedef" {true}
             | ...

Declarators(x) ::= s=Declarator {h=put(head(defs),s,x); defs=list(h,tail(defs))}
                     ("," Declarators(x))*

Declarator ::= id:Identifier {id.yield}
             | x=Declarator "(" ParameterTypeList ")" {x}
             | ...

Expr ::= Expr "-" Expr
       | "(" n:TypeName [isType(defs,n.yield)] ")" Expr
       | "(" Expr ")"
       | ...
       | Identifier [!isType(defs,n.yield)]
```

Figure 3.15: Resolving typedef ambiguity in C.

introduce global variable `defs`, holding a list of maps to properly account for scoping. At the beginning of parsing, `defs` is a list containing a single, empty map. At the beginning of a new scope, i.e., when `"{"` is encountered, an empty map is prepended to the current list resulting in a new list which is assigned to `defs` (not shown in the figure). At the end of the current scope, i.e., when `"}"` is encountered, the head of the current list is dropped by taking the tail of the list and assigning it to `defs`.

To communicate the presence of `typedef` in a list of specifiers, we extend each rule of `Specifier` to return a boolean value: `"typedef"`-rule returns `true`, and the other rules return `false`. `Specifiers` computes disjunction of the values associated with the specifiers in the resulting list. This information is passed via variable `x` to `Declarators`. We also extend the rules of `Declarator` to return the declared name, `id.yield`. After a declarator is parsed, the declared name can be stored in `defs`: pair `(s,x)` is added to the map taken from the head of the current list, and a new list, with the resulting map as its head, is created and assigned to `defs`.

Finally, `isType` function is used to check whether the current identifier is a type name in the current scope or not: `isType` iterates over elements in `defs`, starting from the first element, to look up the given name. If the name is not found in the current map, `isType` continues the search with the next element, representing the outer scope. If the name is found, `isType` returns the boolean value associated with the name. If none of the maps contains the name, `isType` returns `false`.

In our running example, after parsing the second declaration of `T`, appeared in the scope of the `main` function, pair `("T",false)` will be added to the map in the head of `defs`, effectively shadowing the previous typedef declaration of `T`, and causing the condition in the cast expression rule to fail.

3.4 Implementation

In this section we present our extension of the GLL parsing algorithm [78] to support data-dependent grammars. GLL parsing is a generalization of recursive-descent parsing that supports all context-free grammars, and produces a binarized Shared Packed Parse Forest (SPPF) in cubic time and space. GLL uses a Graph-Structured Stack (GSS) [85] to handle multiple function calls in recursive-descent parsing. The problem of left recursion is solved by allowing cycles in GSS. We discuss SPPF and GSS in GLL parsing in more detail in Sections 3.4.1 and 3.4.2, respectively. As GLL parsers are recursive-descent like, the handling of parameters and environment is intuitive, and the implementation remains very close to the stack-based semantics, which eases the reasoning about the runtime behavior of the parser.

We use a variation of GLL that uses a more efficient GSS [3]. GLL parsing can be seen as a grammar traversal process that is guided by the input. At each point during parsing, a GLL parser is at a grammar slot (grammar position before or after a symbol in a rule) and executes the code corresponding to this slot. Because of the nondeterministic nature of general parsing, a GLL parser needs to record all possible paths and process them later, and at the same time eliminate duplicate jobs. The unit of work in GLL parsing is a *descriptor* which captures a parsing state. Descriptors allow a serialized, discrete view of tasks performed during parsing. GLL parsing has a main loop, in a trampolined style, that executes the descriptors one at a time until no more descriptors left.

The standard way of implementing a GLL parser is to generate code for each grammar slot [78]. Such implementation relies on *dynamic goto*s to allow arbitrary jumps to the main loop or other grammar slots. In our GLL implementation, a grammar is modeled as a connected graph of grammar slots. This model of context-free grammars resembles Woods' *Recursive Augmented Transition Networks* (ATN) [100] grammars. As such, our implementation of GLL over ATN grammars provides an interpreter version of GLL parsing. We present our interpretive formulation of GLL parsing in Section 3.4.4.

3.4.1 SPPF

It is known that any parsing algorithm that constructs Tomita-style SPPF is of unbounded polynomial complexity [42]. To achieve parsing in cubic time and space, GLL uses a *binarized* SPPF [78] format, which has additional *intermediate* nodes. Intermediate nodes allow grouping of the symbols of a rule in a left-associative manner, thus allowing the parser to always carry a single node at each time, instead of a list of nodes. This is the key in preserving the cubic bound. The use of intermediate nodes effectively achieves the same as restricting a grammar to have rules of length at most two, but without requiring rewriting the original grammar, and transforming back the resulting derivation trees to the ones of the original grammar.

Definition 1 A binarized SPPF is a compact representation of a parse forest that has the following types of nodes.

- *nonterminal* nodes of the form (A, i, j) where A is a nonterminal, and i and j are the left and right extents;
- *terminal* nodes of the form (t, i, j) where t is a terminal, and i and j are the left and right extents;
- *packed* nodes of the form (L, k) where L is a grammar slot and k is the pivot of the node; and
- *intermediate* nodes of the form (L, i, j) where L is the grammar slot, and i and j are the left and right extends.

The left and right extents of a node represent the substring in the input, associated with the node. As GLL parsing is context-free, nodes with the same label, the same left and the same right extents can be shared. Nonterminal and intermediate nodes have packed nodes as their children. Packed nodes represent a derivation, and can have at most two children, which are non-packed nodes. If a non-packed node is ambiguous, it will have more than one packed node. The pivot of a packed node is the right extent of its left child, and is used to distinguish between packed nodes under a non-packed node.

The binarized SPPF resulting from parsing the input string `-a+a` with the grammar $E ::= -E \,|\, E + E \,|\, a$ is shown in Figure 3.16 (top), where packed nodes are depicted with small circles. For a better visualization, we have omitted the labels of packed nodes. The input is ambiguous and has the following two derivations: `(-(a+a))` or `((-a)+a)`. This can be observed by the presence of two packed nodes under the root node. The left and right packed nodes under the root node correspond to the first and second alternatives, respectively.

SPPF construction is delegated to two functions **nodeT** and **nodeP**. The **nodeT**(t, i, j) function takes terminal t, and two integer values i and j (left and right extents) and returns an existing terminal node with these properties, otherwise a new node. **nodeP**(L, w, z) takes a grammar slot L, and two non-packed nodes w and z. **nodeP** returns an existing non-packed node labeled L with two children w and z. If no such node exists, then a non-packed node labeled L will be created, and w and z are connected to the newly created non-packed node via a packed node. The details of GLL parse tree construction is discussed in [78], and implementation techniques for efficient sharing of nodes are presented in [3, 45].

3.4.2 GSS

At the core of GLL parsing is the Graph-Structured Stack (GSS) data structure. We use a variation of GLL that uses a more efficient GSS [3].

Definition 2 GSS in GLL parsing is a directed graph where

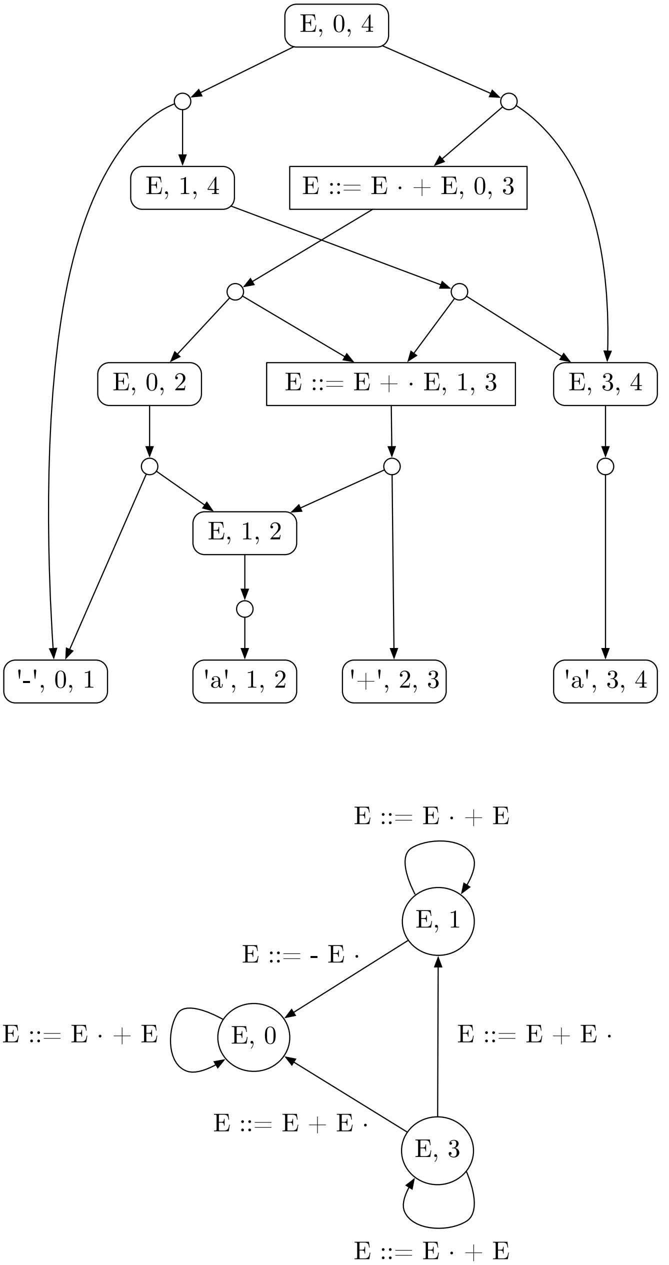


Figure 3.16: SPPF (top) and GSS (bottom) for the input `-a+a`.

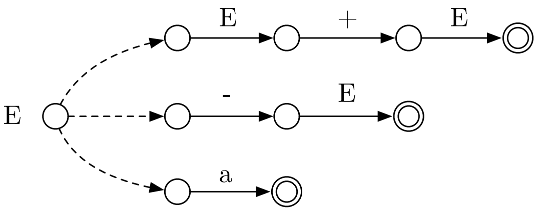


Figure 3.17: ATN Grammar for $E ::= E + E \,|\, -E \,|\, a$.

- nodes are of the form (A, i), where A is a nonterminal and i is an input position; and

- edges are of the form (u, L, w, v), where u and v are GSS nodes, L is a grammar slot, and w is an SPPF node recorded on the edge.

GSS was originally developed by Tomita [85] for GLR parsing to merge different LR stacks. Although GLL parsing uses the same term, there are two main differences between GSS in GLL parsing and GLR. First, in GLL parsing GSS represents function calls in recursive-descent parsing, similar to memoization of functions in functional programming, and therefore has the input position at which the nonterminal is called. Second, in GLL parsing GSS allows cycles in the graph that solve the problem of left-recursion in recursive-descent parsing.

The GSS resulting from parsing `-a+a` using the grammar $E ::= -E \,|\, E + E \,|\, a$ is shown in Figure 3.16 (bottom). As can be seen there is a cycle on all nodes, as they represent the left recursive calls to E at different input positions. In case of indirect left recursion, there will be a cycle in the GSS involving multiple nodes.

3.4.3 ATN Grammars

ATN grammars are an automaton formalism developed in the 70s to parse natural languages, and are similar to nondeterministic finite automata.

Definition 3 An ATN grammar is a tuple $(Q, F, \rightarrow)$ where

- Q is a finite set of states representing grammar slots;

- $F \subset Q$ is a finite set of states representing *final* grammar slots; and

- $\rightarrow$ is a transition relation of the form $\xrightarrow{A}$ (nonterminal), $\xrightarrow{t}$ (terminal), or $\xrightarrow{\epsilon}$ (epsilon).

For example, the ATN grammar for $E ::= E + E \,|\, -E \,|\, a$ is shown in Figure 3.17. In an ATN, there is a one-to-many relation, $S \subset \text{String} \times Q$, from a nonterminal name to a set of start states, each representing the initial state of an alternative.

$$\boxed{(\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P}) \Rightarrow (\mathcal{R}',\mathcal{U}',\mathcal{G}',\mathcal{P}')}$$

$$\text{Eps}\ \dfrac{\begin{array}{c} p \xrightarrow{\epsilon} q \\ n = \mathbf{nodeP}(q, w, \mathbf{nodeT}(\epsilon, i, i)) \end{array}}{(\{(p,i,u,w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{(q,i,u,n)\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Term-1}\ \dfrac{\begin{array}{c} p \xrightarrow{t} q \quad I[i] = t \\ n = \mathbf{nodeP}(q, w, \mathbf{nodeT}(t, i, i{+}1)) \end{array}}{(\{(p,i,u,w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{(q,i{+}1,u,n)\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Term-2}\ \dfrac{p \xrightarrow{t} q \quad I[i] \neq t}{(\{(p,i,u,w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Call-1}\ \dfrac{\begin{array}{c} p \xrightarrow{A} q \quad v = (A, i) \in \mathcal{N}(\mathcal{G}) \\ \mathcal{D} = \{d \mid (v, y) \in \mathcal{P}, d = (q, \text{rext}(y), u, \mathbf{nodeP}(q, w, y)), \\ d \notin \mathcal{U}\} \end{array}}{\begin{array}{c} (\{(p,i,u,w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \mathcal{D}, \mathcal{U} \cup \mathcal{D}, \\ \mathcal{G} \cup \{(v,q,w,u)\}, \mathcal{P}) \end{array}}$$

$$\text{Call-2}\ \dfrac{\begin{array}{c} p \xrightarrow{A} q \quad v = (A, i) \notin \mathcal{N}(\mathcal{G}) \\ \mathcal{D} = \{(s, i, v, \$) \mid s \in S(A)\} \end{array}}{(\{(p,i,u,w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \mathcal{D}, \mathcal{U}, \mathcal{G} \cup \{(v,q,w,u)\}, \mathcal{P})}$$

$$\text{Ret}\ \dfrac{\begin{array}{c} p \in F \\ \mathcal{D} = \{d \mid (u, q, y, v) \in \mathcal{G}, d = (q, i, v, \mathbf{nodeP}(q, y, w)), d \notin \mathcal{U}\} \end{array}}{(\{(p,i,u,w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \mathcal{D}, \mathcal{U} \cup \mathcal{D}, \mathcal{G}, \mathcal{P} \cup \{(u, n)\})}$$

Figure 3.18: GLL parsing over ATN grammars.

Constructing an ATN grammar from a CFG is straightforward. For each nonterminal in the grammar, and for each alternative of the nonterminal, a pair consisting of the nonterminal's name and a state representing the start state of the alternative is added to S. Finally, for each symbol in the alternative, a next state is created, and a transition, labeled with the symbol, from the previous state to this state is added. The last state of the alternative is marked as a final grammar slot.

3.4.4 Interpretive Formulation of GLL Parsing

In this section, we define GLL parsing over ATN grammars as a transition relation. In contrast to the imperative style used in [3, 78], we use the declarative rules of

Figure 3.18. Such GLL formulation is concise and easy to extend to support data-dependent grammars. The rules in Figure 3.18 use notation similar to one in [3, 78].

The unit of work of a GLL parser is a descriptor. A descriptor is of the form (p, i, u, w), where p is an ATN state representing a grammar slot, u is a GSS node, i is an input position, and w is an SPPF (non-packed) node. A GLL parser maintains a set $\mathcal{U}$ that holds descriptors created during parsing and is used to eliminate duplicate descriptors. In addition to $\mathcal{U}$, a set $\mathcal{R}$ is used to hold pending descriptors that are to be processed. Note that GLL parsing does not impose any order in which the descriptors in $\mathcal{R}$ are processed. Figure 3.18 defines the semantics of GLL parsing over ATN grammars as a transition relation on configuration $(\mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P})$, where $\mathcal{G}$ represents GSS (a set of GSS edges), such that $\mathcal{N}(\mathcal{G})$ gives a set of GSS nodes, and $\mathcal{P}$ is a set of parsing results that are associated with GSS nodes, i.e., a set of elements of the form (u, w).

During parsing a descriptor is selected and removed from $\mathcal{R}$, represented as $\{(p, i, u, w)\} \cup \mathcal{R}$, and given the rules, a deterministic choice is made based on the next transition in the ATN. The first three rules of Figure 3.18 are straightforward. An ϵ transition creates an ϵ-node (via call to **nodeT**) and intermediate node[4] (via call to **nodeP**), and adds a descriptor for the next grammar slot. The terminal rules (Term-1 and Term-2) try to match terminal t at the current input position, where I is an array representing the input string. If there is a match (Term-1), a terminal node (via **nodeT**) and intermediate node (via **nodeP**) are created, and a descriptor for the next grammar slot is added. If there is no match (Term-2), no descriptor is added.

Call-1 and Call-2 correspond to nonterminal transitions $\xrightarrow{A}$. Similar to calling a memoized function, a GLL parser first checks if a GSS node (A, i) exists. If such a node exists (Call-1), the parsing results associated with this GSS node are reused. These results are retrieved from $\mathcal{P}$, and for each result, nonterminal node y, a descriptor d is created (rext returns the right extent of y), and if the same descriptor has not been processed before $(d \notin \mathcal{U})$, it is added to $\mathcal{R}$. If the GSS node does not exist (Call-2), the call to the nonterminal is made, i.e., for each start state of the nonterminal $(s \in S(A))$, a descriptor is added. Both Call-1 and Call-2 add a new GSS edge to $\mathcal{G}$.

Finally, Ret corresponds to a final grammar slot (final states in ATNs) in which the parser returns from the current nonterminal call. First, the tuple with the current SPPF node and the current GSS node is added to $\mathcal{P}$. Second, for each outgoing GSS edge of the current GSS node, a descriptor is created and, if the same descriptor has not been processed before $(d \notin \mathcal{U})$, it is added to $\mathcal{R}$.

3.4.5 Data-dependent ATN Grammars

To support data-dependent grammars, we extend ATN grammars with the following forms of transitions:

- $\xrightarrow{x=l:A(e)}$ (parameterized, labeled nonterminals)

[4] In fact, when the next state is an end state, **nodeP** creates a nonterminal node, instead of an intermediate node. However, in the current discussion, this is not essential, therefore, we always refer to the result of **nodeP** as an intermediate node.

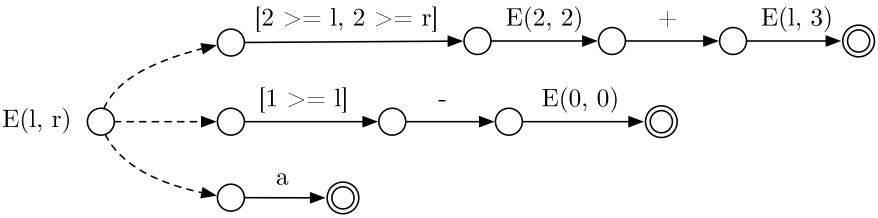


Figure 3.19: Data-dependent ATN grammar for $E ::= E{+}E \;>\; -E \,|\, a$ after desugaring operator precedence.

- $\xrightarrow{l:t}$ (labeled terminals)
- $\xrightarrow{x=e}$ (variable binding),
- $\xrightarrow{[e]}$ (constraint)
- $\xrightarrow{e}$ (return expression).

Two additional mappings are maintained: $L, X : Q \rightarrow \text{String}$ that map a state, representing a grammar slot after a labeled nonterminal, to the nonterminal's label (l) and to the nonterminal's variable (x), respectively. Here, as in Section 3.3.1, for simplicity of presentation and without loss of generality, we assume that nonterminals can have at most one parameter. We also only consider cases of labeled terminals and nonterminals, and when a return expression is present. Finally, we assume that expression language e is a simple functional programming language with immutable values and no side-effects, that labels and variables are scoped to the rules they are introduced in, and that labels and variables introduced by desugaring have unique names in their scopes.

An example of a data-dependent ATN is shown in Figure 3.19. This ATN grammar is the disambiguated version of the grammar shown in Figure 3.17 after desugaring operator precedence.

3.4.6 Data Dependency in GLL Parsing

In the following, p, q, s represent ATN states in Q, i is an input index, u, u' represent GSS nodes, and w, n, y represent SPPF nodes. To support data-dependent grammars, we introduce an environment, E, into GLL parsing. Here, we assume that E is an immutable map of variable names to values. In the data-dependent setting, a descriptor, the unit of work in GLL parsing, is of the form (p, i, E, u, w). Now, a descriptor contains an environment E that has to be stored and later used whenever the parser selects the descriptor to continue from this point. GSS is also extended to store additional data. A GSS node and a GSS edge are now of the forms (A, i, v) and (u, p, w, E, u'), respectively. That is, in addition to the current input index i, a GSS node stores an argument v, passed to a nonterminal A, to fully identify the call. A

GSS edge additionally stores an environment E, to capture the state of the parser before a call to a nonterminal is made.

Finally, a GLL parser constructs a binarized SPPF (Section 3.4.1), creating terminal nodes (**nodeT**), and nonterminal and intermediate nodes (**nodeP**). In GLL parsing intermediate nodes are essential. In particular, they allow the parser to carry a single node at each time by grouping the symbols of a rule in a left-associative manner. Nonterminal and intermediate nodes can be ambiguous. To properly handle ambiguities under nonterminal and intermediate nodes, we include environment and return values into the SPPF construction. Specifically, arguments to nonterminals and return values are part of nonterminal nodes, and environment is part of intermediate nodes.

Figure 3.20 presents the semantics of GLL parsing over ATN, defining it as a transition relation on configuration ($\mathcal{R}$,$\mathcal{U}$,$\mathcal{G}$,$\mathcal{P}$) where the elements are four main structures maintained by a GLL parser:

- $\mathcal{R}$ is a set of pending descriptors to be processed
- $\mathcal{U}$ is a set of descriptors created during parsing. This set is maintained to eliminate duplicate descriptors
- $\mathcal{G}$ is a GSS, represented by a set of GSS edges
- $\mathcal{P}$ is a set of parsing results (SPPF nodes created for nonterminals) associated with GSS nodes, i.e., a set of elements of the form (u, w)

During parsing, a descriptor is selected and removed from $\mathcal{R}$, represented as $\{(p, i, E, u, w)\} \cup \mathcal{R}$, and given the rules, a deterministic choice is made based on the next transition in the ATN. The simplest rules are Eps, Cond-1, Cond-2 and Bind. Eps creates the ϵ-node (via call to **nodeT**) and an intermediate node (via call to **nodeP**), and adds a descriptor for the next grammar slot. Cond-1 and Cond-2 depend on the evaluation of expression e in a constraint. If the expression evaluates to true, a new descriptor is added to continue with the next symbol in the rule (Cond-1), otherwise no descriptor is added (Cond-2). Bind evaluates the expression in an assignment and creates a new environment containing the respective binding. This environment is used to create the new descriptor added to $\mathcal{R}$.

Term-1 and Term-2 deal with labeled terminals. If terminal t matches (Term-1) the input string (represented by an array I) starting from input position i, a terminal node is created (assuming t is of length 1). Then, the properties, i.e., the left and right extents, and the respective substring, are computed from the resulting node (props(y)). Finally, a new environment, containing binding $[l = \text{props}(y)]$, is created and used to construct an intermediate node and a new descriptor. If the terminal does not match (Term-2), no descriptor is added.

Call-1 and Call-2 deal with labeled calls to nonterminals. First, argument e is evaluated, where E_1 allows the use of the left extent in e (lprop constructs properties with only left extent). If a GSS node, representing this call, already exists (Call-1), the parsing results associated with this GSS node are reused, and a possibly empty

$$\boxed{(\mathcal{R},\mathcal{U},\mathcal{G},\mathcal{P}) \Rightarrow (\mathcal{R}',\mathcal{U}',\mathcal{G}',\mathcal{P}')}$$

$$\text{Eps}\;\dfrac{\begin{array}{c} p \xrightarrow{\epsilon} q \\ n = \mathbf{nodeP}(q, w, \mathbf{nodeT}(\epsilon, i, i), E) \quad d = (q, i, E, u, n) \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{d\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Term-1}\;\dfrac{\begin{array}{c} p \xrightarrow{l:t} q \quad I[i] = t \\ y = \mathbf{nodeT}(t, i, i+1) \quad E_1 = E[l = \text{props}(y)] \\ n = \mathbf{nodeP}(q, w, y, E_1) \quad d = (q, i+1, E_1, u, n) \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{d\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Term-2}\;\dfrac{p \xrightarrow{l:t} q \quad I[i] \neq t}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Call-1}\;\dfrac{\begin{array}{c} p \xrightarrow{x=l:A(e)} q \\ E_1 = E[l = \text{lprop}(i)] \quad \llbracket e \rrbracket E_1 = v \quad u' = (A, i, v) \in \mathcal{N}(\mathcal{G}) \\ \mathcal{D} = \{d \mid (u', y) \in \mathcal{P}, E_2 = E[l = \text{props}(y), x = \text{val}(y)], \\ d = (q, \text{rext}(y), E_2, u, \mathbf{nodeP}(q, w, y, E_2)), d \notin \mathcal{U}\} \end{array}}{\begin{array}{c} (\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \mathcal{D}, \mathcal{U} \cup \mathcal{D}, \\ \mathcal{G} \cup \{(u', q, w, E, u)\}, \mathcal{P}) \end{array}}$$

$$\text{Call-2}\;\dfrac{\begin{array}{c} p \xrightarrow{x=l:A(e)} q \\ E_1 = E[l = \text{lprop}(i)] \quad \llbracket e \rrbracket E_1 = v \quad u' = (A, i, v) \notin \mathcal{N}(\mathcal{G}) \\ \mathcal{D} = \{(s, i, [p_0 = v], u', \$) \mid s \in S(A)\} \end{array}}{\begin{array}{c} (\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \mathcal{D}, \mathcal{U}, \\ \mathcal{G} \cup \{(u', q, w, E, u)\}, \mathcal{P}) \end{array}}$$

$$\text{Ret}\;\dfrac{\begin{array}{c} p \xrightarrow{e} q \quad q \in F \\ \llbracket e \rrbracket E = v \quad n = \mathbf{nodeP}(q, w, \arg(u), v) \\ \mathcal{D} = \{d \mid (u, s, y, E_1, u') \in \mathcal{G}, E_2 = E_1[L(s) = \text{props}(n), X(s) = v], \\ d = (s, i, E_2, u', \mathbf{nodeP}(s, y, n, E_2)), d \notin \mathcal{U}\} \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \mathcal{D}, \mathcal{U} \cup \mathcal{D}, \mathcal{G}, \mathcal{P} \cup \{(u, n)\})}$$

$$\text{Cond-1}\;\dfrac{\begin{array}{c} p \xrightarrow{[e]} q \quad \llbracket e \rrbracket E = \text{true} \\ d = (q, i, E, u, w) \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{d\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Cond-2}\;\dfrac{p \xrightarrow{[e]} q \quad \llbracket e \rrbracket E = \text{false}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

$$\text{Bind}\;\dfrac{\begin{array}{c} p \xrightarrow{x=e} q \quad \llbracket e \rrbracket E = v \\ d = (q, i, E[x = v], u, w) \end{array}}{(\{(p, i, E, u, w)\} \cup \mathcal{R}, \mathcal{U}, \mathcal{G}, \mathcal{P}) \Rightarrow (\mathcal{R} \cup \{d\}, \mathcal{U}, \mathcal{G}, \mathcal{P})}$$

Figure 3.20: GLL for data-dependent ATN grammars.

set of new descriptors ($\mathcal{D}$) is created. Each descriptor in the set corresponds to a result, nonterminal node y, retrieved from $\mathcal{P}$, so that the index of the descriptor is the right extent of y (rext), its environment contains bindings $[l = \mathrm{props}(y)]$ and $[x = \mathrm{val}(y)]$ (val retrieves the value from y), and its SPPF node is a new intermediate node. Note that $d \notin \mathcal{U}$ ensures that no duplicate descriptors are added at this point. If the corresponding GSS node does not exist, Call-2 creates one descriptor for each start state of the nonterminal ($s \in S(A)$). Each descriptor gets a new environment with binding $[p_0 = v]$, where p_0 is the nonterminal's parameter that we assume to have a unique name in the scope of a rule. Both Call-1 and Call-2 add a new GSS edge capturing the previous environment to $\mathcal{G}$.

Finally, in Ret-rule, the return expression is evaluated, and the nonterminal node is created which stores both the argument of the current GSS node ($\mathrm{arg}(u)$) and the return value. This node is recorded in $\mathcal{P}$ as a result associated with the GSS node. For each GSS edge directly reachable from the current GSS node, a new descriptor is created. Note that labels and variables at call sites, represented by the current GSS node, are retrieved via mappings L and X, respectively.

3.5 Evaluation

Our data-dependent parsing framework is implemented as an extension of our modified GLL parsing algorithm (see Chapter 2). The addition of data-dependency is at the moment a prototype and most of the effort was put into correctness, rather than performance optimization. As a frontend to write data-dependent grammars, we extended the syntax definition of Rascal [51], a programming language for meta-programming and source code analysis, and provided a mapping to Iguana's internal representation of data-dependent grammars.

In Section 3.2 we enumerated a number of challenges in parsing programming languages, and in Section 3.3, we provided solutions based on data-dependent grammars (directly or via desugaring) that address these challenges. For each challenge we selected a programming language that exhibits it, and wrote a data-dependent grammar[5], derived from the specification grammar of the language. For evaluation, we parsed real source files from the source distribution of the language and some popular open source libraries, see Table 3.1. Table 3.2 summarizes the evaluation results. In the following we discuss these results in detail, and provide an analysis of the expected performance in practice.

Java To evaluate the correctness of our declarative operator precedence solution using data-dependent grammars, we used the grammar of Java 7 from the main part of the Java language specification [32]. This grammar contains an unambiguous left-recursive expression grammar, in a similar style to the expression grammar in Figure 3.1 (middle).

We replaced the expression part (consisting of about 30 nonterminals) of the Java specification grammar with a single `Expression` nonterminal that declaratively expresses

[5] `https://github.com/iguana-parser/grammars`

Table 3.1: Summary of the projects used in the evaluation.

Language	Projects	Version	Description
Java	JDK	1.7.0_60-b19	Java Development Kit
	JUnit	4.12	Unit testing framework
	SLF4J	1.7.12	A Java logging framework
C#	Roslyn	build-preview	.NET Compiler Platform
	MVC	6.0.0-beta5	ASP.NET MVC Framework
	EntityFramew.	7.0.0-beta5	Data access for .NET
Haskell	GHC	7.8	Glasgow Haskell Compiler
	Cabal	1.22.4.0	Build System for Haskell
	Git-annex	5.20150710	File manager based on Git
	Fay	0.23.1.6	Haskell to JavaScript compiler

operator precedence using `>`, **`left`** and **`right`**. The resulting grammar, which we refer to as the *natural* grammar, is much more concise and readable, see Table 3.2. The resulting parser parsed all 8067 files successfully and without ambiguity.

The natural grammar of Java produces different parse trees compared to the original specification grammar, and therefore it is not possible to directly compare the parse trees. To test the correctness of parsers resulting from the desugaring of `>`, **`left`**, and **`right`** to data-dependent grammars, we tested their resulting parse trees against a GLL parser for the same natural grammar of Java, using our rewriting technique for operator precedence rules (see Chapter 4). Both parsers, using desugaring to data-dependent grammars and rewriting operator precedence rules, produced the same parse trees for all Java files, providing an evidence that our desugaring of operator precedence to data-dependent grammars implements the same semantics as the rewriting in Chapter 4.

Despite its prototype status, the data-dependent parser is at the moment on average only 25% slower than the rewritten one. The main reason for performance difference is that in the rewriting technique in Chapter 4, the precedence information is statically encoded in the grammar, and therefore there is no runtime overhead, while in the data-dependent version passing arguments and handling environment is done at runtime. The problem with the rewriting technique is that the rewriting process itself is rather slow and the resulting grammar is very large.

C# To evaluate our data-dependent framework on parsing conditional directives, we used the grammar of C# 5 from the C# language specification [63]. As mentioned in Section 3.2.5, existing C# compilers resolve preprocessor directives in the lexing phase, and the parser is not aware of directives. However, the C# language specification has context-free rules that describe the syntax of directives. Our solution to parsing conditional directives (Section 3.3.6) leverages layout that is automatically inserted

Table 3.2: Summary of the results of parsing with character-level data-dependent grammars of programming languages.

Lang.	Challenge	Solution	Spec. Gram.		Data-dep. Gram.		# Files	% Success
			# Nont.	# Rules	# Nont.	# Rules		
Java	Operator precedence	`>`, **`left`** and **`right`**	200	485	169	435	8067	100% (8067)
C#	Conditional directives	**`global`** variables and dynamic layout	387	1000	395	1013	5839	99% (5838)
Haskell	Indentation sensitivity	**`align`**, **`offside`** and **`ignore`**	143	431	152	452	6457	72% (4657)

between symbols in grammar rules. We used the context-free syntax of directives in C# as the starting point. We extended the layout definition to include directives. Then, the conditional directive rules were modified to allow parse-time evaluation of conditions and selection of the corresponding path.

The resulting data-dependent grammar is only different from the specification grammar in the layout definition, and the difference is minimal. As can be seen in Table 3.2 there are only 8 additional nonterminals and 13 additional rules (about 1.3% of the whole grammar). Using the character-level grammar of C# we could parse 5838 files out of 5839. The parser timed out after 30 seconds on a very large source file from the Roslyn framework. The file, which appears to be automatically generated, contains 156033 lines of code and is of size 4.8 MB.

Although the grammar of C# is near deterministic, the reason for time out is that character-level grammars generate very large parse trees, a node for each character. Nevertheless, this file could be parsed using a context-aware parser of C#. We discuss the performance gain of using a context-aware scanner in Section 3.5.1.

Haskell To evaluate our parsing framework for indentation sensitive programming languages, we used the grammar of Haskell [60]. The specification grammar of Haskell is written using explicit blocks, as if no indentation sensitivity exists, and the lexer translates indentation to physical block delimiters. We took the Haskell grammar as written in the specification as the starting point and added extra rules that specify layout sensitivity using **`align`**, **`offside`** and **`ignore`** constructs. As shown in Table 3.2, the data-dependent version has only 21 additional rules (about 4% of the whole grammar). From the total number of 6457 Haskell files, we could successfully parse 4657 files (72%). The reason for the parse error in other files is that they contained some syntactic constructs from GHC extensions that we do not support yet.

Besides numerous undocumented GHC extensions we found in the source files, many Haskell files contained CPP directives which were resolved by running the C preprocessor, `cpp`, before parsing. In the future, we plan to deal with C directives during parsing, the same way we did for C#. One last issue about parsing Haskell is that indentation rules alone are not sufficient to unambiguously parse Haskell, and there is a need for a syntactic longest match that uses indentation information. For example, the following input string is ambiguous, where both derivations are correct regarding the indentation rules:

```
f x = do print x
           + 1
```

In the first derivation, the right hand side is an infix plus-expression, consisting of a `do`-expression and `1`. The second derivation consists of only a `do`-expression that has `print x + 1` as its subexpression. According to the Haskell language specification the second interpretation is valid, as in `do` expressions longest match should be applied. We resolved this issue by defining a special kind of follow restriction, similar to Section 3.3.3, that bypasses the layout and checks for the indentation level of the next non-whitespace token when the token is not a keyword or a delimiter.

OCaml We used excerpts of OCaml source files to test our operator precedence translation against deep and problematic operator precedence cases. OCaml, in contrast to other three programming languages we used for the evaluation, uses a natural, ambiguous expression grammar in its language specification. The data-dependent grammar of OCaml is basically the same as the reference manual, where the alternative operator in the expression part is replaced with `>` and additional **`left`** and **`right`** operators added. We are not yet able to unambiguously parse full OCaml programs, as they contain operator precedence ambiguities across indirect nonterminals. An example is `pattern-matching` which can derive `expr` on its right-most end:

```
expr ::= expr '+' expr
       | 'function'  pattern-matching

pattern-matching ::= pattern '->'  expr
```

For example, the input string `function x -> x + 1` is ambiguous with the following derivations: `(function x -> x) + 1` or `function x -> (x + 1)`. As the `function` alternative has `pattern-matching` and not `expr` on its right-most end, the operator precedence rules do not apply in our current scheme. The translation of operator precedence in presence of indirect nonterminals to data-dependent grammars seems possible with an additional analysis of indirect nonterminals, but is left as future work.

3.5.1 Running Time and Performance

Data-dependent grammars [41] provide a *pay-as-you-go* model. If a pure context-free grammar is specified, the worst-case complexity of the underlying parsing technology is retained. However, in the general case no guarantees can be made. Our data-dependent parsers, implemented on top of GLL parsing, are worst-case $O(n^3)$ on pure context-free grammars [78]. The more practical question, however, is how data dependency affects the runtime performance of parsing real programming languages.

In this section, we provide empirical results showing that parsers for data-dependent grammars can behave nearly linearly on grammars of real programming languages. We ran the experiments on a machine with a quad-core Intel Core i7 2.6 GHz CPU and 16 GB of physical memory running Mac OS X 10.10.4. We used a 64-Bit Oracle HotSpot™ JVM version 1.8.0_25. Each file was executed 10 times and the mean running time (CPU user time) was reported. The three first runs of each file were skipped to allow for JIT optimizations.

Figure 3.21 shows the log-log plots (log base 10) of running time (ms) against file size (number of characters) for all the files we parsed (see Tables 3.1 and 3.2). For showing the linear behavior we used the character-level grammars, as they exhibit the relationship between the running time and the number of characters better than the context-aware version. The red line in the plots is the linear regression fit. The goodness of each fit is indicated by the adjusted R^2 value in each plot. The equation in each plot describes a power relation with original data, and as all the coefficients

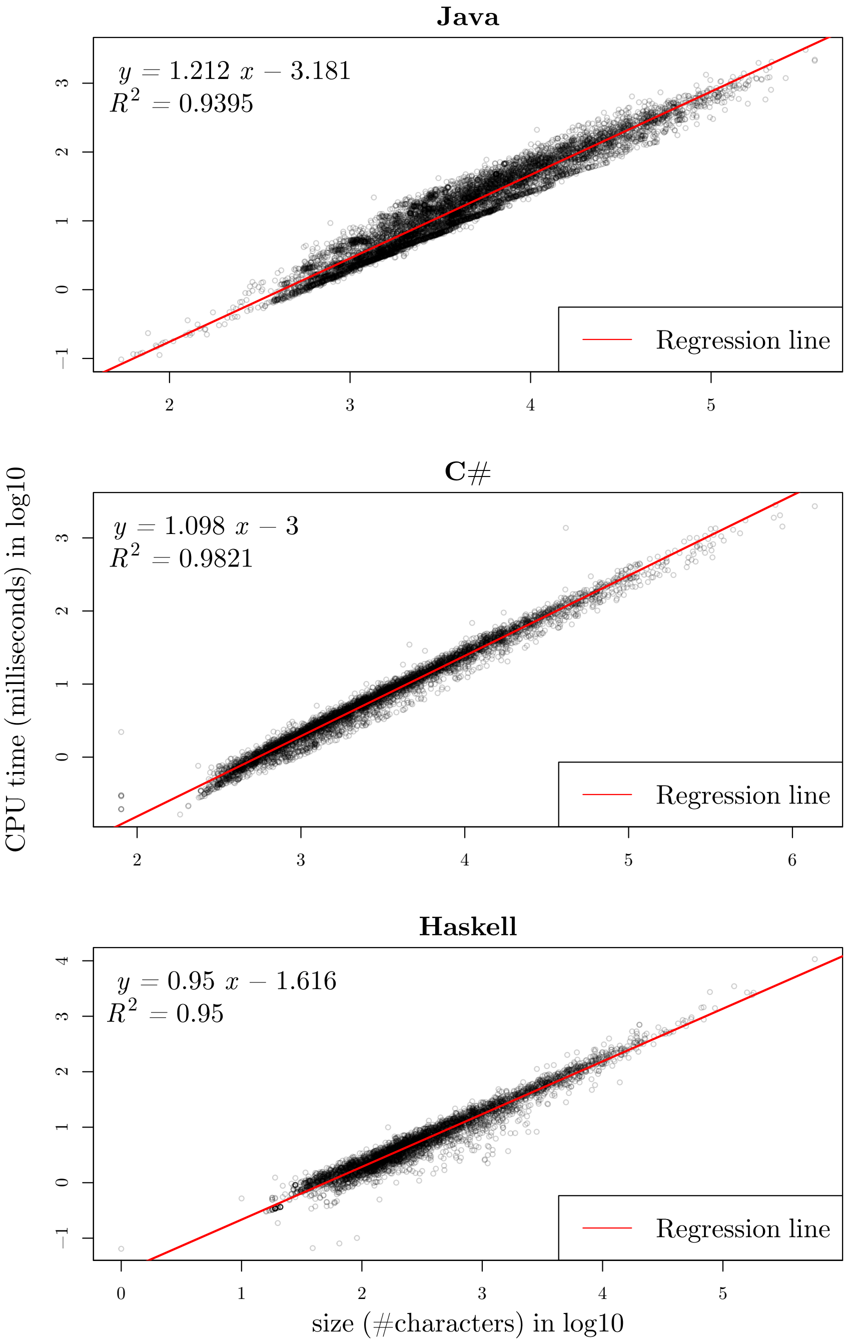


Figure 3.21: Running time of the character-level parsers for Java, C#, and Haskell.

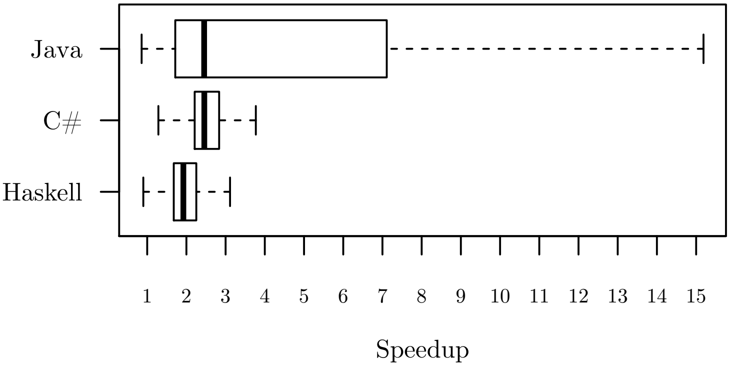


Figure 3.22: The relative speedup using context-aware scanning instead of character level grammars.

(1.212, 1.098, 0.950) are close to 1, we can conclude that the running time is near-linear on these grammars.

To compare the performance difference between character-level and context-aware parsing, we ran both context-aware and character-level parsers on all the source files. Figure 3.22 shows the relative performance gain (speedup) using a context-aware parser compared to a character-level parser for each file. For a better visualization we omitted the outliers from the box plots. The median and maximum speedup for Java is (2.45, 15.1), for C# (2.45, 4) and for Haskell (1.9, 3). The precise impact of context-aware scanning for general parsing and data-dependent grammars is future work, but our preliminary investigation revealed that using character-level grammars for parsing layout is very expensive, as it is a very common operation, see Section 3.3.2.

3.6 Related Work

Parsing is a well-researched topic, and many features of our parsing framework are related in one or another way to other existing systems. Throughout this chapter we have discussed some related work, which we do not repeat here. In this section we discuss direct related work and our inspirations.

Data dependency implementation Data-dependent grammars have many similarities with attribute grammars [59] and attribute directed parsing [99]. A detailed discussion of related systems is provided by Jim *et al.* [41]. From the implementation perspective, Jim *et al.* present the Yakker parser generator [41], which is based on Earley's algorithm [18], but we have a GLL-based interpretation of data-dependent grammars. We also extend the SPPF creation functionality of GLL parsing (taking environments into account), while SPPF creation is not discussed in Jim *et al.*'s approach. Another difference between our implementation and Yakker is that Yakker directly supports regular operators, by applying longest match. We, however, believe that all ambiguities should be returned by the parser, and avoid such implicit heuristics. Therefore, we desugar regular operators to data-dependent BNF rules.

We use an interpretative model of parsing based on Woods' ATN grammars [100]. Woods used an explicit stack to run ATN grammars, similar to a pushdown automata. However, as with any top-down parser, such execution of ATN grammars does not terminate in presence of left recursion. Jim *et al.* 's data-dependent framework operates on a data-dependent automata [39], which is a variation of ATN grammars interpreted with Earley's algorithm.

Indentation-sensitive parsing Besides modification to the lexer which has been used in GHC and CPython, there are a number of other systems that provide a solution for indentation-sensitive parsing. Parser combinators [36] are higher-order functions that are used to define grammars in terms of constructs such as alternation and sequence. This approach has been used in parsing indentation-sensitive languages [37]. Traditional parser combinators do not support left-recursion and can have exponential runtime. Another main difference between parser combinators and our approach is that we do not give the end user access to the internal workings of the parser. Since parser combinators are normal functions, the user can modify them. Our approach provides an external DSL for defining parsers, while parser combinators provide an internal DSL. Therefore, our approach compared to parser combinators provides more control over the syntax definition.

Erdweg *et al.* present an extension of SDF to define layout constraints on grammar rules [20]. These constraint-based approach is implemented by modifying the underlying SGLR [95] parser. Most constraints can be solved during parsing. Constraints that are not resolved will lead to ambiguity which can be removed by post-parse filtering. Adams presents the notion of indentation-sensitive grammars [2], where symbols in a rule are annotated by the relative position to the immediate parents. This technique is implemented for LR(k) parsing.

We do not offer a customized solution for indentation-sensitivity for a specific parsing technology, rather we use the general data-dependent grammars framework, and map indentation rules to them. In addition, we define high-level constructs such as **`align`**, **`offside`**, and **`ignore`** which are desugared to lower-level data-dependent grammars. This enables a syntax definition model that is closer to what the user has in mind. We think the use of high-level constructs leads to cleaner, more maintainable grammars.

Operator precedence SDF2 uses a parser-independent semantics of operator precedence which is based on parent-child relationship on derivation trees [95]. This semantics is implemented in SGLR parsing [95] by modifying parse tables. Although the SDF2 semantics for operator precedence works for most cases, in some cases it is too strong, i.e., rejecting valid sentences, and in some cases it cannot disambiguate the expression grammar.

In Chapter 4 we discussed the precedence ambiguity, and proposed a grammar rewriting that takes an ambiguous grammar with a set of precedence rules and produces a grammar that does not allow precedence-invalid derivations. Our current solution has the same semantics: it does not remove sentences when there is no precedence ambiguity, and can deal with corner cases found in programming languages such as OCaml. In addition, our operator precedence solution is desugared to data-dependent grammars, thus it is independent of the underlying parsing technology.

Conditional directives Recent work in parsing conditional directives target all variations [29,48]. Gazzillo and Grimm [29] give an extensive overview of related work in this area. However, to the best of our knowledge, none of the existing systems employ a single-phase parsing scheme, rather they use a separate scanner and annotate the tokens based on the conditional directives they appear in. Our approach in using data-dependent grammars to evaluate the conditional directives is new. The treatment of other features of preprocessors, such as macros, is future work.

3.7 Conclusions

We have presented our vision of a parsing framework that is able to address many challenges of declarative parsing of real programming languages. We have built an implementation of data-dependent grammars based on the GLL parsing algorithm. We also have shown how to map common idioms in syntax of programming languages, such as lexical disambiguation filters, operator precedence, indentation-sensitivity, and conditional directives to data-dependent grammars. These mappings provide the language engineer with a set of out of the box constructs, while at the same time, new high-level constructs can be added. The preliminary experiments with our parsing framework show that it can be efficient and practical. To fully realize our vision we will explore more syntactic features, and further optimize the implementation of our framework.

Chapter 4

Safe Specification of Operator Precedence Rules[1]

Summary. In this chapter we present an approach to specifying operator precedence based on declarative disambiguation constructs and an implementation mechanism based on grammar rewriting. We identify a problem with existing generalized context-free parsing and disambiguation technology: generating a correct parser for a language such as OCaml using declarative precedence specification is not possible without resorting to some manual grammar transformation. Our approach provides a fully declarative solution to operator precedence specification for context-free grammars, is independent of any parsing technology, and is safe in that it guarantees that the language of the resulting grammar will be the same as the language of the specification grammar. We evaluate our new approach by specifying the precedence rules from the OCaml reference manual against the highly ambiguous reference grammar and validate the output of our generated parser.

[1] This chapter was originally published as: A. Afroozeh, M. van den Brand, A. Johnstone, E. Scott, and J. J. Vinju. Safe Specification of Operator Precedence Rules. In *Proceedings of the 6th International Conference on Software Language Engineering*, SLE '13, pages 137–156. Springer, 2013.

4.1 Introduction

There is an increasing demand for front-ends for programming and domain-specific languages. We are interested in parser generation technology that can cover a wide range of programming languages, their dialects and embeddings. These front-ends are used for example to implement reverse engineering tools, to build quality assessment tools, to execute research in mining software repositories, or to build (embedded) domain specific languages. In these contexts the creation of the parser is a necessary and important step, but it is also an overhead cost that would preferably be mitigated. In such language engineering applications, as opposed to compiler construction, we may expect frequent updates and maintenance to deal with changes in the grammar.

Expression grammars are an important part of virtually every programming language. The natural specification of expressions is usually ambiguous. In programming languages books and reference manuals, the semantic definition of expressions usually includes a table of binary and unary operators accompanied with their priority and associativity relationships. This approach feels very natural, probably because this is the way we learn basic arithmetic expressions at school. Virtually all disambiguation techniques for expression grammars are driven by such precedence rules. However, the implementation of such rules varies considerably.

The implementation of operator precedence in grammars may considerably deviate from the initial design the language engineer had in mind. In manual rewriting approaches, grammars are *factored* to remove ambiguities. These approaches are not attractive for us because the resulting grammars are usually large, and hard to read and understand. For example, programming languages such as OCaml, C# and Java have many operators with a considerable number of priority levels and associativity relations. Manually transforming such expression grammars, to encode precedence rules, is time consuming, and can be error-prone for new users, especially when we consider the evolution of grammars [50]. Therefore, we consider declarative approaches in which the parser is generated from the set of precedence rules.

Generalized context-free parsing algorithms provide the opportunity to write any context-free grammar, and allow for language compositions, which helps in modeling embeddings and dialects. This makes generalized context-free parsing a good starting point for our purpose: satisfying the demand for powerful and maintainable front-ends. This is particularly important in the fields of domain-specific languages and reverse engineering, where grammars should be easy to understand, evolvable, and maintainable. Therefore, the focus of this chapter is mainly on providing a declarative framework for specification of precedence rules in generalized context-free parsing algorithms, such as Earley [18], GLR [11,61,71,85] and GLL [78].

4.1.1 From Yacc to SDF

In this section, we discuss two disambiguation techniques that influenced our work the most, and are related to generating parsers from ambiguous grammars using a set of precedence rules. Aho, Johnson, and Ullman [7] (AJU) present an approach in which the LR(1) [8] parsing tables are modified to eliminate shift/reduce conflicts based on the precedence of operator tokens, as specified by the user. The AJU method is not only a disambiguation mechanism, it is also a *nondeterminism reducer*, meaning that it has to resolve all shift/reduce and reduce/reduce conflicts, even when there is no ambiguity, to make the parser deterministic. This implies that the approach cannot predictably deal with expression grammars that are not inherently LR(1), unless the language engineer understands how additional shift/reduce and reduce/reduce actions, used for making the parser deterministic, affect the language. More importantly, the AJU precedence semantics is defined in terms of the deterministic LR parsers: to understand the semantics of the precedence rules, one must understand what an LR(1) conflict is and why it happens. Finally, this method is not directly applicable to non-LR parsers.

The AJU approach is implemented in Yacc[2] and is very popular. For example, the OCaml parser uses `ocamlyacc`[3], which is a variant of Yacc. However, the Yacc grammar of OCaml is considerably larger than the natural reference one, and contains more nonterminals and rules.

Although the AJU method is fast and effective when used in the context of arithmetic expressions, because it is bound to LR(1) parsing, it does not fit into our definition of declarative operator precedence techniques. We require that a mechanism for declarative specification of operator precedence rules (1) be *independent* of the underlying parsing technology, so that we can reason about the precedence semantics or use the mechanism in other parsing technologies, (2) be *safe*, meaning that the disambiguation mechanism derived from precedence rules should not change the underlying sentences of the language, and (3) be *complete*, i.e., be able to resolve all the ambiguities resulting from different precedence of operators.

There has been a number of efforts to formalize a parser-independent semantics for operator precedence, and to provide a declarative disambiguation mechanism. The most notable one is SDF[4] in which the semantics of operator precedence is defined as a filter on derivation trees. SDF precedence filters are implemented by removing transitions corresponding to filtered productions from adapted SLR(1) tables [94]. Although we believe that SDF was in the right direction in defining a declarative precedence mechanism, its filters lack the safety and completeness requirements. For example, precedence rules in SDF fail to disambiguate a left-associative binary operator having higher priority than a unary prefix operator. The limitations of SDF are discussed in detail in Section 4.2.1.

4.1.2 Contributions and Roadmap

In this chapter we present a new semantics for the declarative specification of operator precedence rules for context-free grammars. The key enablers of our technique are the safety and support for resolving deeply nested precedence conflicts. We also support indirect precedence conflicts when expression grammars are not expressed using a single recursive nonterminal but rather more. We show that our approach is

[2] `http://dinosaur.compilertools.net/yacc/`

[3] `http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html`

[4] `http://www.syntax-definition.org`

powerful enough to allow declarative specification of operator precedence in OCaml. More importantly, the semantics of our technique is implemented as a grammar transformation, making it independent of the underlying parsing technology. We also guarantee that the parsers we generate produce the exact same parse trees (as if the original grammar was used). The completeness proof —whether our technique resolves all precedence style ambiguities— and the soundness proof of the transformation —whether the transformation exactly implements the semantics— are future work.

The rest of this chapter is organized as follows. After this introduction, we give formal definitions which we need in the rest of this chapter. Then, we explain the problems with SDF in detail in Section 4.2.1. After that, the formal semantics of precedence rules and its implementation as a grammar transformation are presented in sections 4.3 and 4.4. We present the results of parsing the OCaml test suite in Section 4.5. Finally, a discussion of related work and a conclusion of this work are given in sections 4.6 and 4.7, respectively.

4.2 Motivation

A grammar is a 4-tuple (N, T, P, S) where N is a set of nonterminals, T a set of terminals, P a set of production rules of the form $A ::= \alpha$ where A, the *head* of the production rule, is a nonterminal and α, the *body* of the production rule, is a string in $(T \cup N)^*$. We shall assume that there are no repeated rules, so we can identify a production rule by writing its head and body. $S \in N$ is the start symbol of the grammar. By convention, in this chapter, nonterminals and terminals start with uppercase and lowercase letters, respectively. In addition, symbols, such as $+$ or $*$ are terminals. We use lowercase letters u, v, w to denote non-empty sequences of terminal symbols. A group of production rules that have the same head can be grouped as $A ::= \alpha_1|\alpha_2|...|\alpha_n$ where each $A ::= \alpha_i$ is a production. In this representation, each α_i is called an *alternative* of A.

A derivation step is of the form $\alpha A\beta \Rightarrow \alpha\gamma\beta$ where $\alpha, \beta \in (T \cup N)^*$ and $A ::= \gamma$ is a production rule. In a derivation step a nonterminal A is replaced with the body of its production rule. A derivation of σ from τ is a possibly empty sequence of derivation steps of the form $\tau \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \sigma$, which is also written as $\tau \overset{*}{\Rightarrow} \sigma$. A derivation is left-most if at each step its left most nonterminal is rewritten. A derivation from the start symbol is called a *sentential form* which is a sequence of terminals or nonterminals. A sentential form consisting only of terminal symbols is called a *sentence*.

A sentence is *ambiguous* if it has more than one left-most derivation. *Disambiguation* is a process which eliminates derivations. A disambiguation is said to be *safe* if it does not remove all derivations. Therefore, a safe disambiguation mechanism does not change the underlying language generated by a grammar.

4.2.1 Limitations of SDF

SDF features three meta notations $>$, *left*, and *right*, which specify the precedence, left and right associativity of operators, respectively [53]. Having $A ::= \gamma > B ::= \alpha$[5] disallows the derivation steps of $B ::= \alpha$ from all B's in γ. $A ::= A\alpha$ $\{left\}$ means that the A in $A\alpha$ should not derive $A ::= A\alpha$ itself. Right associativity is the same as the left, but applied on the right-most A. There are three problems with the semantics of SDF[6] disambiguation filters:

- It is *unsafe*: A filter is applied even when there is no ambiguity. For example, having $(E ::= E \mathbin{\char`^} E > E ::= -E)$ rejects the string `1 ^ - 1`, even though this string is not ambiguous. This is because, based on the semantics of SDF, $-E$ cannot appear under any of the E's in the body of the rule. SDF also allows the user to specify under which nonterminal the filtering should be carried out. For example, the user can specify that the filtering should only be carried out under the first E in the body of the rule, written as $(E ::= E \mathbin{\char`^} E <0> > E ::= -E)$ in SDF. This solves the problem for these two operators, but this explicit selection of the filtered nonterminal is transitively applied to all levels below, even where it should not be applied, which can produce wrong results.

- It is *incomplete:* The precedence relationship in SDF is defined as a one-level relationship. As a result, it cannot resolve ambiguities in some cases that require deeper than one level searching in the derivation trees. For example, a left-associative binary operator having higher priority than a prefix unary operator remains ambiguous. The problem with one-level filtering is explained in Section 4.2.2.

- It is *limited* to directly recursive rules. Although SDF has some extensions to filter priority modulo chain rules, general indirect recursion is not supported. Rules such as $E ::= E\,A$, where the right-most nonterminal, A, can eventually produce an E at the right-most position cannot be filtered using SDF priorities.

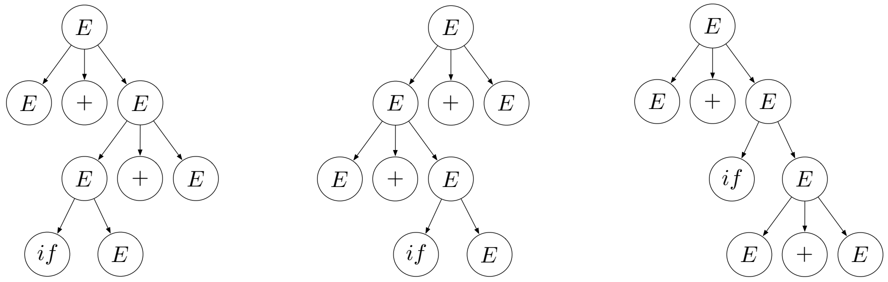
These limitations are encountered in practice. For example, the `if-then-else` operator in functional programming languages such as OCaml and Haskell acts as a unary operator with lower priority than left-associative binary operators. Indirect recursion also happens, for example, in the reference grammar of OCaml.

4.2.2 Problem with One-level Filtering

To illustrate the problem with one-level filtering, we consider the `if-then-else` construct in OCaml, which has lower priority than `+`. For example, the expression `1 + if x then 2 else 3 + 4` is interpreted as `1 + (if x then 2 else (3 + 4))` rather than

[5] SDF adheres to algebraic notations and writes $A ::= \gamma$ as $\gamma \rightarrow A$. In this chapter we use the more common $::=$ notation.

[6] We describe here SDF version 2 [53] but we simply call it SDF.



(a) $(E + ((\ \mathit{if}\ E) + E))$ (b) $(E(E + (\ \mathit{if}\ E)) + E)$ (c) $(E + (\ \mathit{if}\ (E + E)))$

Figure 4.1: Parse trees from parsing `1 + if 2 + 3`.

`(1 + (if x then 2 else 3)) + 4`. For notational simplicity, the `if...then..else` part is replaced with if .

$$\begin{aligned} E ::= & E\ +\ E \\ & |\ \mathit{if}\ \ E \\ & |\mathit{Num} \end{aligned}$$

Figure 4.1 shows the parse trees resulting from parsing the input `1 + if 2 + 3`. For a more compact presentation the terminals (1, 2 , 3) are removed.

In SDF, the precedence and associativity rules for disambiguating this case will be:

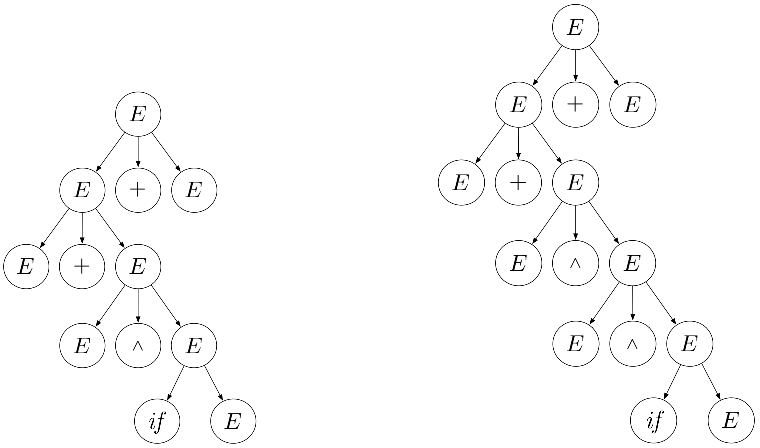
$$E ::= E\ +\ E\ \ \{\mathit{left}\} \tag{Rule 1}$$
$$E ::= E\ +\ E\ >\ E\ ::=\ \mathit{if}\ \ E \tag{Rule 2}$$

The disambiguation is not safe in this case: when Rule 2 is applied, $E ::=\ \mathit{if}\ \ E$ is removed under both E's, which rejects a sentence such as `1 + if 2 + 3`. We can make it safe by changing Rule 2 into ($E ::=\ E\ +\ E$ <0> $>\ E\ ::=\ \mathit{if}\ \ E$). Now if we examine the effect of the definitions on the shown parse trees in Figure 4.1, we can observe that the left-associativity removes the derivation in Figure 4.1a. However, none of the definitions affect the remaining two parse trees, and thus the disambiguation fails. The reason that SDF definitions fail to disambiguate this grammar is that patterns of depth greater than two are required. The first E in the body of $E ::= E + E$ can first derive $E ::= E + E$ and then the second E in the body of the newly derived rule derives $E ::= if\ \ E$. In other words, the following derivation

$$E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow E +\ \mathit{if}\ \ E + E$$

remains, which is not rejected by any of the defined patterns, but it is semantically incorrect. The derivation in Figure 4.1c is correct and is the only one that should remain after disambiguation.

For this grammar, a two level filtering can solve the problem, but in general, we may need filters of arbitrary depth. For example, consider the following grammar



(a) $(E(E + E(E\ {}^\wedge\ (\ \mathit{if}\ E))) + E)$ (b) $(E(E + E(E\ {}^\wedge\ (E\ {}^\wedge\ E(\ \mathit{if}\ E)))) + E)$

Figure 4.2: Some expression grammars require filters of arbitrary depth.

which has an additional expression rule $E ::=\ E\ {}^\wedge\ E$, where ${}^\wedge$ is right associative and has the highest priority.

$$\begin{aligned} E ::= {} & E\ {}^\wedge\ E \\ & |\, E\ +\ E \\ & |\, \mathit{if}\ E \\ & |\mathit{Num} \end{aligned}$$

To illustrate why filters of arbitrary depth may be needed, consider the following derivation:

$$E \Rightarrow E + E \Rightarrow E + E + E \Rightarrow E + E\ \ {}^\wedge E + E \overset{*}{\Rightarrow} E + E\ \ {}^\wedge\ \ E\ \ {}^\wedge\ \ ...\ \ {}^\wedge E + E$$

As can be seen, after deriving $E + E$, the second E may unboundedly produce $E\ {}^\wedge\ E$, leading to derivation trees with wrong precedence levels. Figure 4.2 shows two of such derivations. For disambiguating such cases, either an infinite number of filters or a mechanism to define filters with variable length is needed. It is not trivial to implement a variable length filter during parsing and it is very likely that the performance of such an implementation will suffer.

We have now established the gap in resolving ambiguities in expression grammars. In the following we propose a general solution that solves the aforementioned limitation, and at the same time improves other quality aspects.

4.3 Syntax and Semantics for Operator-style Disambiguation

Expression-style grammar rules display a specific kind of ambiguity, which we call *operator-style* ambiguity. We characterize and define two complementary and safe ambiguity removal schemes for exactly this kind of ambiguity: priority and associativity. Note that this does not imply that our mechanisms completely disambiguate any expression grammar. There may be other ambiguity hidden in the same rules with different causes. This other ambiguity should be left untouched for safety.

4.3.1 Definitions

Operator-style ambiguity

An operator-style-ambiguity exists if for some grammar nonterminal E there exist two leftmost derivations

$$xE\mu \Rightarrow x\beta E\mu \underset{lm}{\overset{*}{\Rightarrow}} xvE\mu \Rightarrow xvE\alpha\mu \tag{1}$$

$$xE\mu \Rightarrow xE\alpha\mu \Rightarrow x\beta E\alpha\mu \underset{lm}{\overset{*}{\Rightarrow}} xvE\alpha\mu \tag{2}$$

which contain identical sub-derivations $\beta \underset{lm}{\overset{*}{\Rightarrow}} v$.

The first derivation in the above definition effectively corresponds to the binding $x\beta(E\alpha)\mu$ and the second derivation corresponds to binding $x(\beta E)\alpha\mu$. Both derivations correspond to the same sentential form, but between them the order of applying $E\alpha$ and βE has been inverted. Note that it may happen that $\alpha = \beta$, but only for binary recursive rules, such as $E ::= E\gamma E$.

The benefit of the above characterization of operator-style ambiguity is that we use pairs of derivations that specifically allow an arbitrary distance ($\overset{*}{\Rightarrow}$) between application of $E\alpha$ and βE. This creates the potential for supporting deeper ambiguities, and indirectly recursive expression grammars. In addition, we now have defined clearly what it means for operator-style ambiguity removal to be safe: never both derivations (1) and (2) may be disallowed at the same time, and therefore, no sentence may be removed from the language.

Given a grammar which contains operator-style ambiguity, the language engineer has to specify which derivation should be disallowed. We first describe *priority-based* ambiguity removal.

Priority-based ambiguity removal via >

The user specifies a strict partial order > (irreflexive, antisymmetric and transitive) between the alternatives of E. If $\beta E > E\alpha$, derivations of the form (1) are disallowed. Vice versa, if $E\alpha > \beta E$, we choose to disallow derivations of the form (2). Note that we do not intend to apply the partial order on other cases of ambiguity, only for derivations of the form (1) and (2), it serves to choose one over the other.

This definition correlates with the common use of operator priority to specify disambiguation. For example choosing the first derivation gives the α "operator" priority over the β "operator". Since all derivations $\beta \overset{*}{\Rightarrow} v$ are available for both choices, priority disambiguation does not put constraints on other disambiguation choices.

The fact that $>$ is asserted to define a strict partial order is an important detail for satisfying the safety requirement. If $\alpha > \beta$ and $\beta > \alpha$ hold at the same time, both derivations above are disallowed, which removes sentences from the language and violates the safety property. Similarly $\alpha > \alpha$ is not allowed. The fact that $>$ is allowed to be partial implies that under-specified orderings may leave some operator precedence ambiguity intact. This means that it is up to the language engineer to fully declare what the relative precedence of operators is, and the priority relation can be developed incrementally.

There are, however, common situations in which we do not want to use or cannot enforce a strict partial order as required by $>$. In particular, if an expression-style rule has an alternative with both immediate left and right recursion, $E ::= E\gamma E$, then it is not possible to specify priority with itself, since $>$ must be irreflexive and antisymmetric. More generally, there may be two alternatives $E ::= E\gamma E \mid E\delta E$ where γ and δ are required to have a symmetric relation (such as $+$ and $-$ in arithmetic expressions), which also contradicts a strict partial order.

Symmetric Operator-style Ambiguity

Instantiating α and β from derivations (1) and (2) above as $\beta = E\delta$ and $\alpha = \gamma E$ both rules are now binary recursive. We can instantiate the derivations (1) and (2) above as:

$$xE\mu \Rightarrow xE\delta E\mu \overset{*}{\underset{lm}{\Rightarrow}} xvE\mu \Rightarrow xvE\gamma E\mu \tag{1'}$$

$$xE\mu \Rightarrow xE\gamma E\mu \Rightarrow xE\delta E\gamma E\mu \overset{*}{\underset{lm}{\Rightarrow}} xvE\gamma E\mu \tag{2'}$$

Also, taking $\beta = E\gamma$ and $\alpha = \delta E$ we can write derivations (1) and (2) above as

$$xE\mu \Rightarrow xE\gamma E\mu \overset{*}{\underset{lm}{\Rightarrow}} xvE\mu \Rightarrow xvE\delta E\mu \tag{1''}$$

$$xE\mu \Rightarrow xE\delta E\mu \Rightarrow xE\gamma E\delta E\mu \overset{*}{\underset{lm}{\Rightarrow}} xvE\delta E\mu \tag{2''}$$

Symmetric operator-style ambiguity is a special case of operator-style ambiguity in which both rules are binary. Often we have $\delta = \gamma$, although this is not necessary. To see why we call the ambiguity symmetric, consider the example where $\gamma = +$ and $\delta = -$, $(1')$ and $(2')$ both derive $y - y + y$ and, $(1'')$ and $(2'')$ both derive $y + y - y$. Then, $(1')$ and $(1'')$ represent $y - (y + y)$ and $y + (y - y)$, respectively.

Associativity-based ambiguity removal via *left* and *right*

We define two binary relations *left* and *right* between binary alternatives, for which holds that

$$(\mathit{left}\ \cap\ \mathit{right} = \emptyset)\ \wedge\ (\mathit{left}\ \cap\ '{>}' = \emptyset)\ \wedge\ (\mathit{right}\ \cap\ '{>}' = \emptyset)$$

In other words, $>$, *left* and *right* are mutually exclusive relations.

When $(\alpha, \beta) \in \mathit{left}$, associativity-based ambiguity removal disallows the derivations of the form $(1')$, which prefers the grouping $x(w\delta E)\gamma E\mu$ over $w\delta(E\gamma E)\mu$. Similarly, when $(\alpha, \beta) \in \mathit{right}$, associativity-based ambiguity removal disallows the derivations of the form $(2')$.

The restriction of $>$, *left* and *right* being mutually exclusive is a sufficient restriction for guaranteeing safety since now only one relation is allowed to be active at the same time and each of the relations is safe in itself.

Since $>$, *left* and *right* need to define an order between all alternatives of an expression grammar, with potentially many rules, we cannot expect the language engineer to specify each combination manually. This problem is dealt with in our formalism, which is described later, by providing automatic transitive closure for $>$ and a computation akin to Cartesian product for *left* and *right* groups of rules.

In summary, the three relations $>$, *left* and *right* allow a language engineer to remove all operator-style ambiguity of the form in Definition 1, either using an anti-symmetric, irreflexive, transitive relation $>$, or using one of the possibly reflexive, possibly symmetric and possibly non-transitive *left* and *right* relations as long as the three relations exclude each other. Note that in theory all operator-style ambiguity can be removed by simply asserting a full ordering among all recursive alternatives using $>$ or by putting all rules in a single *left* or *right* group, but this has no practical value. Instead, complete disambiguation of the operator-style ambiguity in a language definition needs to be considered language-by-language (see Section 4.5).

4.3.2 Pattern Notation for Illegal Derivations

As an intermediate step we now introduce a short notation for the derivations $(1), (2), (1')$ and $(2')$, called *patterns*. Each pattern is specific for a given grammar and combination of two alternative rules. In the next section, we demonstrate how to compute a unified set of patterns from a context-free grammar augmented with $(>, \mathit{left}, \mathit{right})$ relations, and how to use this set of patterns to compute a grammar transformation that implements the above semantics.

Operator ambiguity removal pattern

An operator ambiguity removal pattern (pattern for short) is a 4-tuple of the form $(\mathit{head}, \mathit{parent}, i, \mathit{child})$, where *head* is the nonterminal head of the expression grammar for which the precedence rules are defined, *parent* is an alternative of *head*, i is the index of a nonterminal in the body of *parent*, and *child* is the alternative that should be filtered from the nonterminal at position i of *parent*. The nonterminal at position i is called the *filtered nonterminal*.

In this chapter we write a pattern as $(E, \alpha \mathbin{\bullet} \beta \backslash \gamma)$ where E is the head, and $\alpha \mathbin{\bullet} \beta$ and γ are the parent and the child alternatives, respectively, and the filtered nonterminal is identified by a dot before it.

The semantics of patterns are the same as derivations discussed above. For example, the derivations (1) and (2) can be expressed as the patterns $(E, \beta \mathbin{\bullet} E \backslash E\alpha)$ and

Table 4.1: The semantics of the $>$ operator in terms of patterns.

$>$	$E ::= E\alpha_2 E$	$E ::= E\alpha_2$	$E ::= \alpha_2 E$
$E ::= E\alpha_1 E$	$(E, \bullet E\alpha_1 E \backslash E\alpha_2 E)$ $(E, E\alpha_1 \bullet E \backslash E\alpha_2 E)$	$(E, E\alpha_1 \bullet E \backslash E\alpha_2)$	$(E, \bullet E\alpha_1 E \backslash \alpha_2 E)$
$E ::= E\alpha_1$	$(E, \bullet E\alpha_1 \backslash E\alpha_2 E)$	—	$(E, \bullet E\alpha_1 \backslash E\alpha_2)$
$E ::= \alpha_1 E$	$(E, \alpha_1 \bullet E \backslash E\alpha_2 E)$	$(E, \alpha_1 \bullet E \backslash E\alpha_2)$	—

Table 4.2: The semantics of *left* associativity

left	$E ::= E\alpha_1 E$	$E ::= E\alpha_2 E$
$E ::= E\alpha_1 E$	$(E, E\alpha_1 \bullet E \backslash E\alpha_1 E)$	$(E, E\alpha_1 \bullet E \backslash E\alpha_2 E)$
$E ::= E\alpha_2 E$	$(E, E\alpha_2 \bullet E \backslash E\alpha_1 E)$	$(E, E\alpha_2 \bullet E \backslash E\alpha_2 E)$

$(E, \bullet E\alpha \backslash \beta E)$, respectively. Note that patterns are not implementation mechanisms. In Section 4.4 we show a grammar rewriting algorithm to implement patterns.

We now explain informally how to arrive at a set of patterns starting from a context-free grammar augmented with $(>, \mathit{left}, \mathit{right})$. Table 4.1 documents the semantics of priority in terms of patterns that are generated for each combination of left, right and binary recursive expression rules. Note that for binary rules sometimes two patterns are generated for the same combination of rules. The semantics of *left* in terms of the patterns is expressed similarly in Table 4.2. We leave the table for right associativity to the reader.

As can be seen, not all combinations of expression rules generate patterns. Exactly when the combination of rules would *not* be ambiguous and filtering would be unsafe no pattern is generated. This corresponds to the derivations $(1), (2), (1'), (2')$ using specific combinations of left and right recursive rules. In Section 4.4 we implement these tables.

4.3.3 Defining $>$, *left* and *right* in Practice

The following three features, which are taken from the design of SDF [94], are described here for the sake of completeness. They are essential for having concise expression grammars, as mentioned above.

Firstly, our formalism automatically transitively (but not reflexively) closes the $>$ relation precedence operator. As a result, when the language engineer defines $p_1 > p_2$ and $p_2 > p_3$ we automatically derive $p_1 > p_3$. Furthermore, when they accidentally define $p_1 > p_1$, or both $p_1 > p_2$ and $p_2 > p_1$, either directly, or indirectly

$$
\begin{aligned}
E ::= {} & E\ \mathit{Arg}+ \quad \text{// function application} \\
& |-E \quad \text{// unary minus} \\
& |\, E ** E \\
& |\, E + E \\
& |\, E - E \\
& |\, \mathit{if}\ E\ \mathit{then}\ E\ \mathit{else}\ E \\
& |\mathit{Id} \\
\mathit{Arg} ::= {} & E \\
& |\sim \mathit{label} : E
\end{aligned}
$$

Operator	Assoc
function application	–
unary minus	–
**	right
+, -	left
if-then-else	–

Figure 4.3: Excerpt from OCaml's expression grammar with "challenging" operator precedence.

via the closure, an error message must be produced. Now we can allow the short-hand $p_1 > p_2 > p_3$ to obtain elegant definitions. Note that the transitive closure step is carried out before generating the actual patterns. The actual patterns are generated from the calculated priority pairs only when is there is an operator-style ambiguity, as defined in Section 4.3.2 and documented in Table 4.1.

Secondly, many programming languages have groups of binary operators that have the same precedence level. For example, in $E ::= E + E \mid E - E$ both operators have the same precedence level but should be left associative with respect to each other. We define a left associative group containing a set of rules $(p_1|\ldots|p_n)(\mathit{left})$ to generate a set of associativity declarations:

$$\bigcup_{1 \leq i,j \leq n} p_i\ \mathit{left}\ p_j, \textbf{when}\ (p_i, p_j) \notin \mathit{right} \wedge (p_i, p_j)\ \notin\ '>'\ \wedge (p_j, p_i)\ \notin\ '>'$$

We do similarly for right associative groups. The groups simply compute the Cartesian product, but do not add tuples that would contradict a relation defined elsewhere. Finally, associativity groups may occur in the middle of a priority chain, as in $(p_1|\ldots|p_n)(A) > (q_1|\ldots|q_n)(B)$. In this case $>$ will be extended by combining each element of the two groups pairwise (and before closure). An additional safety feature (which is novel) is to simply statically check for $>$, *left* and *right* to be non-overlapping, as required.

Finally, some expression languages disallow certain direct nesting while indirect nesting is allowed. For example `1 == 2 == 3` should not be allowed while `true == (2 == 2)` is allowed. Normally we have to introduce a new expression nonterminal just to disallow this direct nesting. So, in order to be able to write concise grammars we add *non–assoc* declarations with the following semantics. If p_1 *non–assoc* p_2, then $(p_1\ \mathit{left}\ p_2) \wedge (p_1\ \mathit{right}\ p_2)$. Notice that *non–assoc* declarations are not safe: they intentionally and explicitly remove sentences from the language as generated by the grammar. We extend the associativity group semantics with *non–assoc* as well.

$$
\begin{array}{rlr}
E & ::= E \ \mathit{Arg}+ & (\mathit{non\text{–}assoc}) \\
 & > - \ E & \\
 & > E \ ** \ E & (\mathit{right}) \\
 & >(\ E \ + \ E \,|\, E \ - \ E\ ) & (\mathit{left}) \\
 & > \mathit{if} \ E \ \mathit{then} \ E \ \mathit{else} \ E & \\
 & \ |\mathit{Id} & \\
\mathit{Arg} & ::= E & \\
 & \ | \sim \mathit{label} : E &
\end{array}
$$

Figure 4.4: Example definition of challenging operator precedence rules.

Necessarily, any static safety checks on *left* and *right* need to be done before the tuples from *non–assoc* have been added.

To illustrate the syntax of our approach we use the example grammar in Figure 4.3 and its priority and associativity properties, which both are taken from the OCaml reference manual[7]. The grammar and the precedence rules can now be written as in Figure 4.4. We use ::=, >, *left*, *right* and *non–assoc* meta notation to encode both the syntax and the precedence table in one go.

4.4 Grammar Rewriting to Exclude Illegal Derivations

In this section we present an algorithm for transforming a grammar accompanied with a set of priority and associativity rules to a grammar that prevents the generation of illegal derivations (see figures 4.5 and 4.6).

1. We translate the definitions to a set of patterns (generatePatterns).

2. We apply these patterns to transform the grammar (rewriteGrammar)

The generation of patterns in Figure 4.5 follows exactly the semantics as defined earlier in tables 4.1 and 4.2. extractDefinitions produces a set of binary tuples which represent the associativity and priority declarations in a grammar. This set is an over-approximation of the patterns that will be generated later, since they are not specific for positions in the parents yet and may be ignored entirely if no ambiguity may arise. For a specific nonterminal, rightRecursive and leftRecursive compute which other nonterminals contribute to an eventual left/right recursion of that nonterminal. The generatePattern function then filters the extracted definitions making sure to introduce a pattern only where left recursion tangles with right recursion and vice versa, i.e., simulating exactly the priority and associativity semantics of Section 4.3.

[7] http://caml.inria.fr/pub/docs/manual-ocaml-4.00/expr.html

```
function EXTRACTDEFINITIONS(G)
    $'>' \leftarrow\ '>'\ \cup\{(p_i, q_j) | (p_1 \dots p_i)(A) > (q_1 \dots q_j)(B) \in G\}$   ▷ expand the groups
    $\text{P} \leftarrow \{(p_1, p_2) \mid p_1 > p_2 \in G\}+$   ▷ note the transitive closure
    $\text{L} \leftarrow \{(p, p) \mid p\ \mathit{left}\ p\ \in G\}, L' \leftarrow L$
    $\text{R} \leftarrow \{(p, p) \mid p\ \mathit{right}\ p\ \in G\}$
    $\text{L} \leftarrow L \cup \bigcup_{0 \le i,j \le n} \{(p_i, p_j) \mid (p_1| \dots |p_n)(\mathit{left}) \in G, (p_i, p_j) \notin R\}$
    $\text{R} \leftarrow R \cup \bigcup_{0 \le i,j \le n} \{(p_i, p_j) \mid (p_1| \dots |p_n)(\mathit{right}) \in G, (p_i, p_j) \notin L'\}$
    return $P \cup L \cup R$

function RIGHTRECURSIVE(G, N)   ▷ LEFTRECURSIVE is elided for brevity
    $R \leftarrow \{\text{N}\}$
    while R changes do $R \leftarrow R \cup \{X | X ::= \alpha Y \in G, Y \in R\}$
    return R

function PLAIN($x$) $=$ $x$ in which all $N_i$ are replaced by $N$.

function RULES($G, N$) $= \{\beta | N ::= \beta \in G\}$

function FRESH($N$) $= N_i$ where the integer index $i$ has not been used before.

function GENERATEPATTERNS(G)
    $D \leftarrow$ EXTRACTDEFINITIONS($G$)
    $R \leftarrow \{\}$
    for all $(A ::= X\alpha, A ::= \beta Y) \in D$ do
        if $X \in$ LEFTRECURSIVE$(G, A) \wedge Y \in$ RIGHTRECURSIVE$(G, A)$ then
            $R \leftarrow R \cup \{(A, \bullet X\alpha, \beta Y)\}$
    for all $(A ::= \alpha X, A ::= Y\beta) \in D$ do
        if $X \in$ RIGHTRECURSIVE$(G, A) \wedge Y \in$ LEFTRECURSIVE$(G, A)$ then
            $R \leftarrow R \cup \{(A, \alpha \bullet X, Y\beta)\}$
    return R
```

Figure 4.5: Translating priority and associativity definitions to safe patterns.

Given the set of patterns generated by GENERATEPATTERNS, we can now transform the grammar using the REWRITEGRAMMAR function as shown in Figure 4.6. It is important to note that we use indexed nonterminals names, such that when building parse trees, no new names for nonterminals are generated (indices can be removed easily). As each rewrite action can only remove some alternatives, no new shapes of rules are created by the algorithm (no additional chain rules). This preserves the shape of the parse forest as the language engineer specified in the original grammar.

The algorithm first deterministically generates a set of nonterminals to implement single-level filtering. Lines 2–10 reserve fresh nonterminal names. Lines 13–15 change existing rules to use the new nonterminals at the right positions. Lines 18–22 generate

```
function RewriteGrammar($(G, P)$)
    New $\leftarrow \emptyset$
    Slots[ ] $\leftarrow \emptyset$        $\triangleright$ an empty map from indexed nonterminal names to slots
    $\triangleright$ Stage 1, reserve nonterminal names
    for all patterns $(Y, \beta \cdot Y\gamma, \delta)$ in $P$ do
        $Y_i \leftarrow$ fresh($Y$)
        Slots[$Y_i$] $\leftarrow \beta \cdot Y\gamma$
        add $Y_i$ to New
        $G_1 \leftarrow G$
    $\triangleright$ Stage 2, update use sites
    for all patterns $(Y, \beta \cdot Y\gamma, \delta)$ in $P$ do
        if Slots[$Y_i$] $= \beta \cdot Y\gamma$ then
            replace $Y ::= \beta Y\gamma$ in $G_1$ with $Y ::= \beta Y_i\gamma$
    $\triangleright$ Stage 3, add definitions for new nonterminals
    for all $Y_i$ in New do
        if Slots[$Y_i$] $= \beta \cdot Y\gamma$ then
            for all $Y ::= \alpha$ in $G_1$ do
                if $\nexists$ a pattern $(Y, \beta \cdot Y\gamma, \delta) \in P$ with plain($\alpha$) $= \delta$ then
                    add $Y_i ::= \alpha$ to $G_1$
    $\triangleright$ Stage 4, look for nested ambiguity
    $(G'', G') \leftarrow (G_1, G)$
    while $G' \neq G''$ do
        $(G', \text{New}') \leftarrow (G'', \text{New})$
        for all $Y_i \in$ New$'$ do
            if Slots[$Y_i$] $= \cdot Y\gamma$ then
                for all grammar rules $Y_i ::= \mu W \in G'$ do
                    if plain($W$) $= Y \wedge \exists Z$ (plain($Z$) $= Y$
                        $\wedge\ W \in$ RightRecursive($G_1, Z$)) then
                        for all patterns $(Y, \cdot Y\gamma, \delta)$ do
                            $(G'', U) \leftarrow$ ApplyPattern($G''$, $W$, $\delta$, $Y_i ::= \mu W$)
                            (Slots[$U$], New) $\leftarrow$ (Slots[$W$], New $\cup \{U\}$)
            if Slots[$Y_i$] $= \beta \cdot Y$ then
                for all grammar rules $Y_i ::= W\mu$ in $G''$ do
                    if plain($W$) $= Y \wedge \exists Z$ (plain($Z$) $= Y$
                        $\wedge\ W \in$ leftRecursive($G_1, Z$)) then
                        for all patterns $(Y, \beta \cdot Y, \delta)$ do
                            $(G'', U) \leftarrow$ applyPattern($G''$, $W$, $\delta$, $Y_i ::= W\mu$)
                            (Slots[$U$], $NT$) $\leftarrow$ (Slots[$W$], New $\cup \{U\}$)
    return $G''$
```

Figure 4.6: Core algorithm that rewrites a grammar, applying patterns to remove alternatives from indexed nonterminals.

```
function ApplyPattern($G,\ W,\ \delta,\ V ::= \mu' W' \tau'$)
    $Y_{alts} = \emptyset$
    for all $\rho \in \textsc{rules}(G, W)$ do
        if $\textsc{plain}(\rho) \neq \textsc{plain}(\delta)$ then add $\rho$ to $Y_{alts}$
    if $\exists Z \in G :\ (\textsc{plain}(Z) = \textsc{plain}(W)) \vee (\textsc{rules}(G, Z) = Y_{alts})$ then
        $Y' \leftarrow Z$
    else
        $Y' \leftarrow \textsc{fresh}(W)$
        for all $\beta \in Y_{alts}$ do add $Y' ::= \beta$ to $G$
    remove $V ::= \mu' W' \tau'$ from $G$
    add $V ::= \mu' Y' \tau'$ to $G$
    return $(G, Y')$
```

Figure 4.7: The ApplyPattern helper function.

definitions for the new nonterminals by cloning the original while leaving out the filtered alternative. Then, in a fixed point computation (lines 24–42) we treat each level of newly generated nonterminals to a procedure for eliminating deeply nested cases. For left recursive positions (lines 29–35), we make sure that a nonterminal is generated which cannot derive a given postfix operator at arbitrary depth at the right-most position which has lower priority. For right recursive positions we do the opposite (lines 36–42). The applyPattern helper function in Figure 4.7 does the same as lines 13–42 for the first level, but it includes an explicit check for the existence of generated nonterminals to reuse. This check is necessary for termination as well as efficiency. The fixed point computation will terminate because a new nonterminal is only created in ApplyPattern if a nonterminal which defines the same subset of alternatives does not already exist. Since every step removes an alternative, eventually —in a worst case scenario— all singleton sets will have been generated and the algorithm terminates.

We illustrate the rewiring algorithm using the following example (grammar G):

$$E ::= E + E \ \ (\mathit{left}) \ > \ iE \mid a;$$

For this grammar, the $>$ and *left* relations generate the following patterns (see Figure 4.5):

$$(E,\ \cdot E + E,\ iE)$$
$$(E,\ E + \cdot E,\ E + E)$$

Now the RewriteGrammar function in Figure 4.6 can start. Lines 2–15 create the following grammar rule in G_1, applying the two patterns above and allocating two fresh nonterminals:

$$E ::= E_1 + E_2 \mid iE \mid a$$

Then, at lines 18–22 we define the two new nonterminals and extend G_1 with their definition:

$$\begin{aligned} E &::= E_1 + E_2 \mid iE \mid a \\ E_1 &::= E_1 + E_2 \mid a \\ E_2 &::= iE \mid a \end{aligned}$$

Finally we search for nested cases in lines 25–42. The outer loop executes twice. The first time, E_1 results in a new nonterminal E_3, and E_2 does nothing. The second time nothing changes and the algorithm terminates with the final grammar:

$$\begin{aligned} E &::= E_1 + E_2 \mid iE \mid a \\ E_1 &::= E_1 + E_3 \mid a \\ E_2 &::= iE \mid a \\ E_3 &::= a \end{aligned}$$

4.5 Validation Using the OCaml Case

We have conducted an extensive validating experiment. The goal is to show that our approach is indeed more powerful than SDF, and to provide evidence that the algorithm works for complicated, real-world examples.

4.5.1 Method

For this case study, we selected the OCaml (.ml) files in the test suite directory of the source release of OCaml 4.00.1. OCaml features the kind of ambiguity that SDF filtering semantics cannot solve and our method should be able to solve. The test suite contains numerous examples of different sizes and complexity, testing the language features. We believe the test suite is a good choice for testing our parser on safety and completeness, as the suite rigorously tests the language itself. The suite contains 387 files of which 158 (in the `tool-ocaml` folder) contain only source code comments that document expected output (assembler code) of the compiler. The other 229 files are examples of OCaml code that exercise all features of the language in different combinations to test the compiler.

For this case study, we used the Rascal meta-programming language [52] to define the OCaml grammar. We applied the operator precedence rewriting technique introduced in this chapter on the resulting grammar, and used the rewritten grammar to parse the input files. We used our implementation of the GLL parsing algorithm [78] for parsing the OCaml input files. The OCaml grammar written in Rascal is available at the GitHub repository[8] of this case study.

Our goal is to provide evidence of the equivalence between the original OCaml parser and the parser generated from our approach. This means that no parse error should be produced by our parser if no parse error was produced by the original OCaml parser, and the generated parser should produce single parse trees (no ambiguities), and that the structure of the abstract syntax trees should be exactly the same.

[8] https://github.com/cwi-swat/ocaml-operator-ambiguity-experiment/

```
Ptop_def                                                (
  [                                                       +
    structure_item ([1,0+0]..[1,0+5]) ghost               (
      Pstr_eval                                               1
      expression ([1,0+0]..[1,0+5])                           *
        Pexp_apply                                            (
        expression ([1,0+1]..[1,0+2])                             2
          Pexp_ident "+"                                          3
        [                                                     )
          <label> ""                                      )
            expression ([1,0+0]..[1,0+1])               )
              Pexp_constant Const_int 1
          <label> ""
            expression ([1,0+2]..[1,0+5])
              Pexp_apply
              expression ([1,0+3]..[1,0+4])
                Pexp_ident "*"
              [
                <label> ""
                  expression ([1,0+2]..[1,0+3])
                    Pexp_constant Const_int 2
                <label> ""
                  expression ([1,0+4]..[1,0+5])
                    Pexp_constant Const_int 3
              ] ] ]
```

Figure 4.8: The original AST print from the OCaml parser (left) and the stripped version containing only the structure and the labels (right).

To compare parse trees we adapted both the parser from the OCaml compiler and the output of our parser to produce exactly the same bracketed forms. The resulting files are then compared with `diff`, ignoring whitespace, to check for equivalence. It should be noted that the ASTs from the OCaml compiler were normalized, for example flat lists were converted to cons list. We performed the same transformation steps on our ASTs.

OCaml programs are basically composed of groups of expressions. The AST produced by the OCaml parser is complex and contains many features. However, because of the expression-like nature of the language, most of the unnecessary information can be removed, resulting in a bracketed form. We modified OCaml's default AST printer[9] to produce the bracketed form. For example, the original AST and its bracketed form, resulting from parsing the string `1+2*3` is shown in Fig. 4.8. The bracketed forms of all the examples we examined are available on GitHub[10].

For conducting the experiments we wrote a Rascal grammar definition using the notations defined in this chapter. The grammar is obtained from the OCaml reference

[9] The `parsing/printast.ml` file in the OCaml source release.

[10] `https://github.com/cwi-swat/ocaml-operator-ambiguity-experiment`

manual[11]. We tried to be as faithful as possible to the grammar in the reference manual, avoiding changes as much as possible.

4.5.2 Results

The priority and associativity properties, retrieved from the precedence tables in the language manual, resulted in a grammar that uses $>$ and *left*, *right* and *non–assoc* declarations. These declarations result in 830 ambiguity removal patterns. The rewriting was performed as explained in Section 4.4.

The rewritten grammar provided us with a very close over-approximation of what the OCaml language designers had in mind. Only a handful of ambiguities, such as the dangling-else ambiguity and identifier conflicts with keywords, remained, which were resolved using other ambiguity resolution features of Rascal.

We have performed the parsing and comparison process for the given 229 number of files in the case study. 215 files parse correctly and without ambiguity, of which, 182 files (84%) generate ASTs that are identical in both versions. This means that our parser produces the same grouping as the original OCaml parser, providing evidence for the correctness of our algorithms. For the rest (16%), our manual examination of the `diff` files shows that the differences are minor and are caused by AST de-sugaring and normalization steps in the OCaml compiler, and are not related to the operator precedence.

4.5.3 Discussion and Threats to Validity

One of the difficulties in this case study was how to compare ASTs. The ASTs from the OCaml parser, in some places, are significantly different from the grammar in the OCaml language specification. The reason is that the ASTs have been normalized for easier processing in later phases of the compiler. For example, flat argument lists are converted to cons lists, presumably to simplify the currying and partial function application features in OCaml. These changes are not documented in the reference manual. We examined the ASTs produced by the OCaml parser to deduce the normalization steps. We then mimicked these normalization steps as AST transformations before outputting the final bracketed form.

Moreover, OCaml has some language extension and syntactical varieties that are not documented in the language specification manual. The use of semicolon was particularly confusing. Semicolon is used in OCaml to separate expressions, defined by the rule $E ::= E \, ; E$ which is right associative. However, in the inputs we parsed, we observed several occasions in which semicolon ended an expression without being followed by another expression. We resolved this issue by allowing optional semicolons at the end of expressions.

[11] `http://caml.inria.fr/pub/docs/manual-ocaml-400/language.html`

4.6 Related Work

Besides the AJU and SDF methods which have been discussed in previous sections, there are a number of other work which present similar ideas. Aasa [1] proposes a framework for the specification of operator precedence for implementing programming languages. To the best of our knowledge, this is the only declarative approach to operator precedence that supports deeper patterns. In this work, a parse tree is considered precedence correct based on the weights given to operators in its sub-trees. This work correctly recognizes that, for example, a unary operator can be placed under the right most operand of a binary rule, regardless of their precedence. Our approach in defining precedence semantics is different in that instead of focusing on parse trees, we defined the semantics of precedence as derivations. The main shortcoming of Aasa's work is that operators must be unique, e.g., there cannot be a unary minus and a binary minus operator at the same time. In addition, there is no discussion of indirect recursion. Similar to our work, the disambiguation technique in Aasa's work is implemented as a grammar rewriting.

Thorup [83] presents an algorithm for transforming an ambiguous grammar with a set of partial illegal parse trees to a grammar excluding those derivations. The rewriting technique in this work expects a set of illegal parse trees, and in case the set is unbounded, as in Section 4.2.2, a set of parse forests with cycles. Then, the algorithm works bottom up, generating all production rules which do not result in any of those illegal parse trees. The resulting grammar of this step should go through another transformation to be simplified. The problem of how to find sufficient illegal parse trees is addressed in another work by the same author [82]. The rewriting technique presented by Thorup is not directly aiming at providing a declarative disambiguation mechanism, rather it is more an implementation mechanism. It also covers a wider range of ambiguities provided that enough illegal parse trees are given, but the overall procedure is complicated. We are not aware of any practical parser generator that uses this technique.

Visser [92] presents an approach for translating context-free grammars with priorities to character class grammars. This work describes a grammar transformation to give semantics to the SDF2 precedence relations. In this work, in a first step, a grammar's nonterminals are replaced by explicit sets of identities (integers) of its alternatives. Then, elements are removed from these sets based on the precedence relations. Since every rule is identified, the resulting parse trees do not show the signs of grammar transformation. In contrast to our work, character class grammars do not guarantee to preserve the language and do not support indirect recursion. Although character class grammars are formalized quite differently from our approach that directly manipulates grammars using indexed nonterminals, both methods use grammar transformation to implement the precedence relations.

4.7 Conclusions

Constructing a parser that correctly implements operator precedence rules, for a language such as OCaml, using its ambiguous reference manual and the set of operator precedence rules is not possible without resorting to some manual grammar transformation. In this chapter, we defined a parser-independent semantics for operator-style ambiguities that is safe and is able to deal with deeper level and indirect precedence ambiguities. We evaluated our approach using an experiment by comparing the output of the standard OCaml compiler front-end with the output of our own parser, generated from Rascal. The result is promising and shows that our approach is powerful enough to parse OCaml.

For other functional programming languages, such as Haskell and F#, which have similar expression grammars, our approach is expected to be equally beneficial. Although the focus of this chapter is mainly on generalized parsing algorithms, we should also emphasize that our approach can be used by any parser generator that supports left recursion.

Chapter 5

Operator Precedence for Data-dependent Grammars[1]

Summary. Constructing parsers based on declarative specification of operator precedence is a very old research topic, and there are various existing approaches. However, these approaches are either tied to a particular parsing technique, or cannot deal with all corner cases found in programming languages.

In this chapter we present an implementation of declarative specification of operator precedence for general parsing that (1) is independent of the underlying parsing algorithm, (2) does not require any grammar transformation that increases the size of the grammar, (3) preserves the shape of parse trees of the original, natural grammar, and (4) can deal with intricate cases of operator precedence found in functional programming languages such as OCaml.

Our new approach to operator precedence is formulated using data-dependent grammars, which extend context-free grammars with arbitrary computation, variable binding and constraints. We implemented our approach using Iguana, a data-dependent parsing framework, and evaluated it by parsing Java and OCaml source files. The results show that our approach is practical for parsing programming languages with complicated operator precedence rules.

[1] This chapter was originally published as: A. Afroozeh and A. Izmaylova. Operator Precedence for Data-dependent Grammars. In *Proceedings of the ACM SIGPLAN Symposium/Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 13–24. ACM, 2016.

```
expr ::= expr '.' field
       | expr expr
       | '-' expr
       | expr '*' expr
       | expr '+' expr
       | expr '-' expr
       | 'if' expr 'then' expr
       | expr ';' expr
       | '(' expr ')'
```

Operator	Associativity
.	–
function appl	left
- (unary)	–
*	left
+ -	left
if	–
;	right

Figure 5.1: A simplified excerpt of OCaml expression grammar (left), and its corresponding table of operator precedence (right).

5.1 Introduction

Expressions are basic blocks of programming languages, and perhaps, one of the most difficult parts when it comes to parsing. In reference manuals of programming languages it is common to define expressions using a natural, ambiguous grammar, and specify the precedence and associativity of operators in a table. For example, consider an excerpt of OCaml expression grammar [58] in Figure 5.1 (left) and its accompanying precedence table (right). Constructing parsers for such concise and natural grammar specifications is challenging.

A common approach to unambiguously parse expression grammars is to encode operator precedence rules by rewriting the grammar. This rewriting, which introduces a new nonterminal for each precedence level, is not trivial for grammars of real programming languages, and leads to large grammars. This rewriting is particularly problematic in parsing techniques that do not support left recursion, as the resulting parse trees are considerably different from the ones of the original grammar.

Instead of rewriting the grammar, it is more convenient to use an ambiguous grammar and a set of *declarative* constructs to specify operator precedence. Constructing parsers based on declarative specification of operator precedence is a very old research topic, dating back to the work of Floyd [23] on operator precedence grammars in 1963. Floyd's operator precedence grammars are a limited subset of deterministic grammars, and are mainly used in handwritten recursive-descent parsers for efficient parsing of expressions.

One of the most well-known operator precedence techniques is presented by Aho *et al.* [7]. This approach, which is based on LR parsing and implemented in Yacc [44], maps operator precedence to shift/reduce conflicts. The Yacc-style operator precedence is efficient and powerful. For example, the parser for OCaml, which has one of the most complicated expression grammars, is written using `ocamlyacc`, an OCaml port of Yacc.

When machine resources were scarce, only deterministic parsing techniques were considered. The success of Yacc [44], and its underlying LR parsing theory [17, 54], that has been developed in the 70s, enabled generation of linear parsers from a BNF

```
expr ::= expr '.' field
       > expr expr                         left
       > '-' expr
       > expr '*' expr                     left
       > (expr '+' expr | expr '-' expr)  left
       > 'if' expr 'then' expr
       > expr ';' expr                     right
       | '(' expr ')'
```

Figure 5.2: A simplified excerpt of OCaml expression grammar augmented with declarative operator precedence constructs.

grammar specification. Deterministic parsing techniques are efficient, and guarantee that there will be no ambiguity left in the grammar. However, deterministic parsing techniques are not expressive enough to support the syntax of programming languages out of the box. This means that the grammar writer needs to massage a grammar into a deterministic form.

As machines became more powerful and the need for front-ends in areas other than traditional compiler construction increased, more expressive parsing techniques were considered. For example, in areas such as source code analysis and development of domain-specific languages (DSLs), it is desirable to quickly construct (prototype) a parser. General parsing algorithms [18, 78, 85] can deal with any context-free grammar, and therefore, free the user from the restrictions of a particular deterministic parsing technique. Moreover, general parsers can run nearly linearly on grammars of real programming languages, while keeping the cubic bound on worst-case, highly ambiguous grammars [78, 80]. Of course, the machinery of a general parser imposes some performance overhead [46], but the performance of general parsing is acceptable for the particular applications they are intended for.

An important consequence of using general parsing techniques is that the language engineer needs to explicitly deal with ambiguity, and it is not always easy to pinpoint the cause of ambiguity and resolve it. It is possible to let the parser to produce all parse trees in the form of a parse forest, and then discard the undesired ones. However, in practice, it is desirable to apply disambiguation while parsing, to terminate parsing paths that lead to ambiguity as early as possible.

In a declarative syntax definition formalism, such as SDF [49], disambiguation constructs are declared by the user, rather than being implicitly imposed by the underlying parsing technique. For example, in a declarative approach, operator precedence information in Figure 5.1(right) can be expressed using `>`, `left` and `right`, as shown in Figure 5.2. As can be seen, the precedence information is specified using the `>` construct, where the first alternative has the highest precedence. Associativity is described using `left` and `right`. In case of `+` and `-`, which have the same precedence, but are left-associative with respect to each other, a left associativity group is used.

We distinguish between the notation, semantics, and implementation of an operator precedence approach. For example, in Yacc, the precedence of operators is globally specified based on tokens, as opposed to Figure 5.2, where precedence is locally defined for the alternatives of an expression nonterminal. A token-based notation for specifying operator precedence has two shortcomings. First, an extra mechanism is needed to distinguish between tokens that have different meanings in different rules, e.g., unary and binary minus. Second, there is no native way to specify the operator precedence of an *invisible* operator. For example, the function application operator, `expr ::= expr expr`, in Figure 5.1. The semantics of Yacc-style operator precedence is described in terms of shift/reduce conflicts in LR parsing, which is exactly how it is implemented. While Yacc is powerful and widely used, its operator precedence semantics is bound to the internal workings of LR parsing, and cannot be ported to non-LR parsing algorithms.

There have been a number of related work [1,14,53,84,95] to provide a parser-independent semantics of operator precedence. We discuss these work in detail in Section 7.7. Among them, SDF2 [95] is of particular importance as it provides an intuitive tree-based semantics of operator precedence, and is implemented in context of general parsing, Scannerless GLR (SGLR) [94]. The operator precedence semantics of SDF2 works for most cases, but in some cases it is too strong, removing sentences from the language when there is no ambiguity, and in some cases, it cannot deal with corner cases of operator precedence in programming languages such as OCaml.

In Chapter 4, we proposed a semantics for operator precedence that overcomes the aforementioned limitations of SDF2. Our semantics has the same effect as Yacc semantics, but does not depend on a specific aspect of a parsing algorithm, e.g., shift/reduce conflicts in LALR parsing, and can be implemented in the context of different parsing techniques. In Chapter 4, we proposed a grammar rewriting to implement this semantics. This rewriting preserves the shape of derivation trees. However, it leads to large grammars and introduces unnecessary nondeterminism to the grammar.

Data-dependent grammars [41] extend traditional context-free grammars with arbitrary computation, parametrized nonterminals, variable binding and constraints. Jim *et al.* [41] present the semantics of data-dependent grammars that does not depend on a particular general parsing algorithm. In Chapter 3, we showed that data-dependent grammars can be used as an intermediate layer for parser-independent implementation of various disambiguation strategies. We provided an implementation of the operator precedence semantics of Chapter 4, as desugaring from high-level operator precedence constructs (`>`, `left` ,and `right`) to data-dependent grammars. Compared to the grammar rewriting in Chapter 4, the desugaring preserves the size of the original grammar, while having comparable performance in practice.

In this chapter we extend and improve our translation of operator precedence disambiguation constructs to data-dependent grammars. We extend the translation scheme of Chapter 3 to support indirect operator precedence. We improve the performance by using a new strategy for left-recursive nonterminals. Our translation can deal with deep and indirect precedence cases in programming languages such as OCaml. We evaluate our approach by parsing OCaml and Java source files.

The rest of this chapter is organized as follows. Section 5.2 describes the problem of operator precedence in parsing. Section 5.3 introduces our solution to operator precedence using data-dependent grammars. Section 5.4 presents the evaluation of our technique using grammars of Java and OCaml. Section 5.5 discusses related work, and Section 5.6 concludes.

5.2 The Problem of Operator Precedence

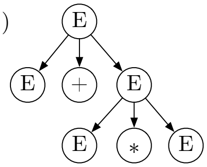
In this section we discuss expression grammars and operator precedence in detail. When explaining the examples, we use two different semantics of operator precedence, namely, Aho *et al.*'s approach [7] based on shift/reduce conflicts, which we refer to as Yacc-style semantics, and the SDF2 semantics [95] based on tree patterns. Understanding the difference between Yacc-style and SDF2-style semantics helps to understand the problem, and motivates our approach to operator precedence in Section 5.3.

5.2.1 Binary Operators

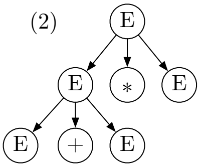
Consider a simple expression grammar with two binary operators `'*'` and `'+'`. Parsing `a+a*a` with this grammar results in two derivation trees, corresponding to the groupings `a+(a*a)` and `(a+a)*a`:

```
E ::= E '*' E
    | E '+' E
    | 'a'
```

(1)



(2)



Based on operator precedence in arithmetics, the first grouping is correct. Both Yacc-style and SDF2-style semantics of operator precedence can deal with this case. We first consider Yacc. This grammar leads to shift/reduce conflicts in the following LR states:

```
(1) E ::= E .'+' E          (2) E ::= E .'*' E
    E ::= E  '+' E.             E ::= E  '*' E.
    E ::= E .'*' E              E ::= E .'+' E
```

In the first state, the shift/reduce conflict between `E ::= E '+' E.` and `E ::= E .'*' E` corresponds to the precedence of `'+'` and `'*'`. If we give `'*'` higher precedence than `'+'`, Yacc resolves this conflict in favor of shifting `'*'`. The shift/reduce conflict between `E ::= E '+' E.` and `E ::= E .'+' E` corresponds to the associativity of `'+'`. If we define `'+'` left-associative, this conflict will be resolved in favor of reduce. The same holds for the second LR state.

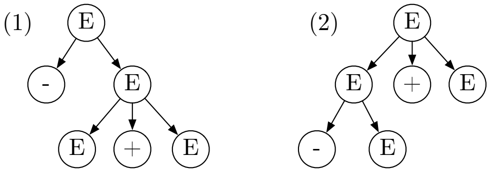
SDF2 [95] uses an operator precedence semantics based on patterns in derivation trees. This parser-independent semantics allows the language engineer to think in terms of tree patterns rather than shift/reduce conflicts. In SDF2, `>` defines a precedence relationship between two alternatives of a nonterminal. For example, `E ::= E '*' E > E ::= E '+' E`[2] means that all `E's` in the body of the `'*'`-rule cannot derive `E ::= E '+' E`. This effectively disallows derivation trees that correspond to the grouping of `'+'` under `'*'`. Associativity in SDF2 is specified using `left` and `right`. For example, `E ::= E '*' E left` means that the second `E` in the body of the `'*'`-rule cannot derive itself. SDF2 semantics can be applied during parsing, by modifying parse tables to remove violating derivation trees, or as a post-parse filtering step. In this chapter, we are concerned with the semantics of SDF2, and not a particular implementation.

5.2.2 Unary and Binary Operators

Combining both unary and binary operators makes the implementation of operator precedence more complicated. In this section we consider two common examples: one from the basic arithmetic and one from functional programming languages.

Arithmetics We consider a combination of unary `'-'` and binary `'+'` operators. Parsing `-a+a` with such a grammar results in two derivation trees, corresponding to the groupings `-(a+a)` or `(-a)+a`:

```
E ::= '-' E
    | E '+' E
    | 'a'
```



Based on operator precedence in arithmetics, where unary `'-'` has higher precedence than binary `'+'`, the second derivation tree is correct. Both Yacc-style and SDF2-style semantics can disambiguate this case. The explanation is similar to the one we gave for the binary-only example of the previous section.

Functional languages So far, we focused on expression grammars that have conventional precedence rules as in basic arithmetics. However, in functional programming languages, there are some combinations of precedence rules which are not found in basic arithmetics. In this chapter, we focus on OCaml [58], a popular dialect of ML, which allows imperative, object-oriented and functional styles of programming. The syntax of OCaml can be seen as a large expression grammar, as almost each construct is an expression. For example, consider the following conditional expression:

```
if b then x else x + 1
```

In OCaml, `'if-then-else'` acts as a unary prefix operator that has lower precedence than binary infix operators. Therefore, this expression should be grouped as `if b then x else (x + 1)` and not as `(if b then x else x) + 1`.

[2] SDF2 adheres to algebraic notation and writes $A ::= \alpha$ as $\alpha \to A$. In this chapter, we use an EBNF-like notation for writing grammar rules.

To observe Yacc's behavior on this grammar, we consider the following conflicting LR states:

```
(1) E ::= 'if' E 'then' E 'else' E.            (2) E ::= E .'+' E
    E ::= E .'+' E                                 E ::= E  '+' E.
```

State (2) corresponds to the associativity of the `'+'` operator, which we discussed in the previous section. State (1) corresponds to the precedence of `'if-then-else'` and `'+'`. As can be seen, in this state we can either reduce the `'if-then-else'` rule or shift `'+'`. Given that `'+'` in OCaml has higher precedence that `'if-then-else'`, the shift action is performed, producing the correct derivation tree.

For this input, the SDF2 semantics can also produce the correct derivation tree. Recall that SDF2 uses `>` to define precedence, and, in this case, `E ::= E '+' E > E ::= 'if' E 'then' E 'else' E` means that the `E`'s in the body of the `'+'`-rule cannot derive `'if-then-else'`, preventing the wrong derivation. We now consider a slightly different input:

```
1 + if b then x else x
```

where the `'if-then-else'` expression is the right operand of `'+'`. This expression is in fact unambiguous and can only have one grouping: `1 + (if b then x else x)`. Yacc can successfully parse this expression, producing the expected derivation tree. However, SDF2 gives a parse error for this example. As `'+'` has higher precedence than `'if-then-else'`, no `E` in the body of the `'+'` can derive `'if-then-else'`, and therefore, this input is rejected. To observe this behavior of the SDF2 semantics, we need a binary operator with higher precedence than a unary operator. In arithmetics, this case only happens for the power operator (`'^'`), for example, for the input `1 ^ - 1`.

The last example shows that in some cases SDF2 semantics is too strong and can remove sentences from the language even if there is no ambiguity. Therefore, the SDF2 semantics for operator precedence is not safe. We call a disambiguation mechanism safe *iff* it does not remove sentences from a language (see Chapter 4 for more details). In other words, when there is no ambiguity, a safe disambiguation mechanism must not apply. Although many disambiguation mechanisms are not safe, for example, longest match, for the operator precedence ambiguity we can ensure safety.

In SDF2, one can fine-tune the behavior of `>` by specifying the exact nonterminal under which filtering should happen. For example, to enforce that filtering should only happen under the left `E`, we can write `E ::= E '+' E <0> > 'if' E 'then' E 'else' E`. Here, `<0>` specifies that filtering should only happen under the nonterminal at position zero. This way of specifying precedence is tedious as the `<i>` operator is not transitive across alternatives, but even if we consider a transitive version, there is another problem with this tree-based semantics of operator precedence: in some cases, this semantic is too weak and cannot disambiguate an operator precedence ambiguity. Consider the following example:

```
1 + if b then x else x + 1
```

According to the precedence and associativity rules in OCaml, this example should be parsed as:

```
(a) 1 + (if b then x else (x + 1))
```

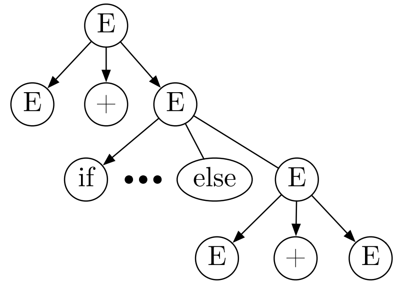
and not as two other alternative derivation trees:

```
(b) (1 + (if b then x else x)) + 1
(c) 1 + ((if b then x else x) + 1)
```

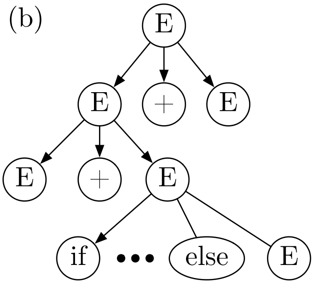
The last two interpretations are precedence-incorrect as they correspond to the cases where `'if-then-else'` binds stronger than `'+'`, and `'+'` is right-associative.

Yacc can deal with this input and produces the expected derivation tree (a). In the conflicting state (1), shown before, Yacc prefers to shift `'+'`, which effectively prevents `'if-then-else'` to bind stronger when it is followed by `'+'`, as in the derivations (b) and (c). In contrast, the SDF2 semantics cannot disambiguate this case, producing the derivation trees (a) and (b). The derivation tree (c) is rejected based on left associativity of `'+'`. To show the parent/child relationships in derivation trees, the two remaining derivation trees are shown below:

(a)



(b)



The SDF2 semantics for operator precedence is defined as a one-level relationship between a parent and a child rule. Based on this relationship, a derivation step from a nonterminal in the body of the parent rule is prohibited. If we examine the derivation trees above, the nodes in both trees are precedence correct with respect to their immediate children: `E ::= E + E` does not appear under the rightmost `E` in the `'+'`-rule, and `E ::= 'if' E 'then' E 'else' E` does not appear under the leftmost `E` in the `'+'`-rule.

The last example illustrates that in order to disambiguate this case, a semantics of operator precedence should only restrict derivation of `'if-then-else'` under the rightmost `E` of the `'+'` rule when this `E` is derived from the leftmost `E` of the `'+'`-rule. We call such cases of operator precedence *deep*. Deep cases commonly, but not only, happen when a left-associative binary operator has higher precedence than a unary prefix operator. This also holds for a right-associative binary operator with higher precedence than a unary postfix operator. Such cases do not happen in arithmetics, but are essential to allow natural writing (without parentheses) of expressions in languages such as OCaml.

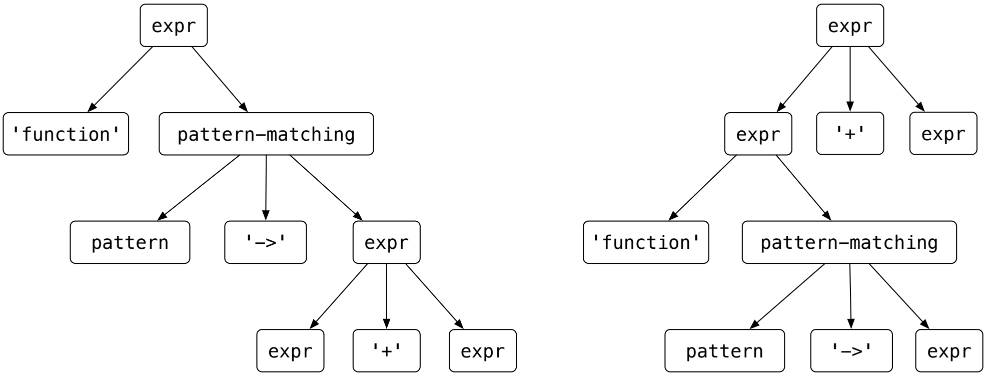


Figure 5.3: Two parse trees for the input `function x -> x + 1`

.

In Chapter 4, we provided a semantics of operator precedence that can deal with such deep cases, and also presented a grammar rewriting technique that implements this semantics. There are mainly two problems with this rewriting technique. First, the size of the generated grammar can be rather large. We have not done a formal analysis, but it appears that the size of the generated grammar is quadratic with respect to the original grammar. To preserve the shape of the original derivation trees, many intermediate nonterminals are introduced. These intermediate nonterminals may introduce nondeterminism into the grammar, and lead to inefficiency in parsing (see Section 5.3.6).

5.2.3 Indirect Recursive Nonterminals

Dealing with deep cases alone is not enough to resolve all precedence ambiguities in OCaml. For example, consider the following (simplified) rules from the OCaml language specification:

```
expr ::= expr '+' expr
       | 'match' expr 'with'  pattern-matching
       | 'function' pattern-matching
       | 'try' expr 'with' pattern-matching

pattern-matching ::= pattern '->' expr
```

As can be seen, the common pattern matching syntax is factored out into a separate `pattern-matching` nonterminal. As `pattern-matching` ends with `expr`, it can cause precedence ambiguity. All these unary prefix operators, `'match'`, `'function'`, and `try`, have lower precedence than binary operators in OCaml, thus, an input string such as `function x -> x + 1` should be parsed as `function (x -> (x + 1))` (Figure 5.3 left), and not as `(function x -> x) + 1` (Figure 5.3 right).

In a declarative syntax formalism with support for operator precedence, we would like the following definition to be able to return the correct derivation tree.

```
expr ::= expr '+' expr
       > 'function' pattern-matching
```

In Chapter 4, we only conjectured on the implementation of such indirect cases, by copying the full grammar part reachable from an indirect nonterminal and rewriting left or rightmost recursive ends. In this chapter, we show a dynamic, more systematic way of dealing with indirect cases of operator precedence. It should be noted that the Yacc-style semantics can deal with indirect cases, for example, in this case, when the token `'->'` is given higher precedence than `'+'` (Yacc considers the precedence of the last terminal of a rule as the precedence of the rule). The reason why Yacc can deal with indirect cases directly corresponds to how an LR automaton is constructed, more specifically closure on LR items. In our example, the closure on the item `pattern-matching ::= pattern '->' . expr` imports the item `expr ::= . expr '+' expr`, leading to the following LR state after a transition on `expr`:

```
pattern-matching ::= pattern '->' expr .
expr ::= expr .'+' expr
```

This state leads to a shift/reduce conflict that can be resolved based on the precedence relationship between `'+'` and `'->'`.

5.2.4 Discussion

So far, we discussed the problem of operator precedence in parsing using Yacc and SDF2 as two leading semantics. The Yacc-style semantics is safe, and can deal with deep and indirect cases. However, this semantics is bound to the inner workings of LR parsing, and cannot be ported to other non-LR parsing techniques. In fact, Yacc was designed to work with LALR grammars, not arbitrary context-free grammars. As a result, for example, the Yacc-style operator precedence cannot be used in a scannerless GLR parser that inserts layout (whitespace and comment) between symbols.

Our goal is to provide a declarative semantics of operator precedence, so that the grammar writer can think in terms of the grammar, rather than inner workings of a parsing algorithm. In this chapter we use a semantics of operator precedence that is defined in terms of derivation trees. Our semantics (Section 5.3.1) can be seen as an extension of SDF2 semantics, that makes it safe, and allows for deep and indirect operator precedence cases. The main contribution of this chapter is how to implement this semantics using data-dependent grammars. Our implementation preserves the size of the original grammar, does not depend on a particular parsing algorithm and is efficient.

5.3 Operator Precedence for Data-Dependent Grammars

5.3.1 Notation and Semantics for Operator Precedence

We use `>` as a high-level construct for declarative specification of operator precedence. In our semantics, `>` defines a partial order on alternatives of a nonterminal. For any two grammar rules r_1 and r_2 with the same head E, $r_1 > r_2$ applies if one of the

```
E ::= E '.' Id
    | E '.[' E ']'
    > E '+' E
    > 'if' E 'then' E
    | '(' E ')'
    | 'a'
```

```
E(p) ::= [3>=p] l=E(p) [l==0||l>=3] '.' Id       {0}                      // 3
       | [3>=p] l=E(p) [l==0||l>=3] '.[' E(0) ']' {0}                     // 3
       | [2>=p] l=E(p) [l==0||l>=2] '+' r=E(2)    {r==0 ? 2 : min(r,2)}    // 2
       |        'if' E(0) 'then' E(1)            {1}                      // 1
       |        '(' E(0) ')'                     {0}                      // -
       |        'a'                              {0}                      // -
```

Figure 5.4: Translation of precedence rules into data-dependent grammars.

rules is left-recursive ($E ::= E\beta$) and the other is right-recursive ($E ::= \alpha E$). This means that rules that are neither left- nor right-recursive, e.g., `E ::= '(' E ')'` are not affected by `>`.

We define operator precedence as a relationship between alternatives of a nonterminal, and not tokens. We consider three possible types of rules. Unary prefix rules ($E ::= \alpha E$) where α is nonempty and does not start with E, unary postfix rules ($E ::= E\beta$) where β is nonempty and does not end with E, and binary rules of the form $E ::= E\gamma E$, where γ is a possibly empty sequence of symbols. In this setting, α, β and γ act as operators.

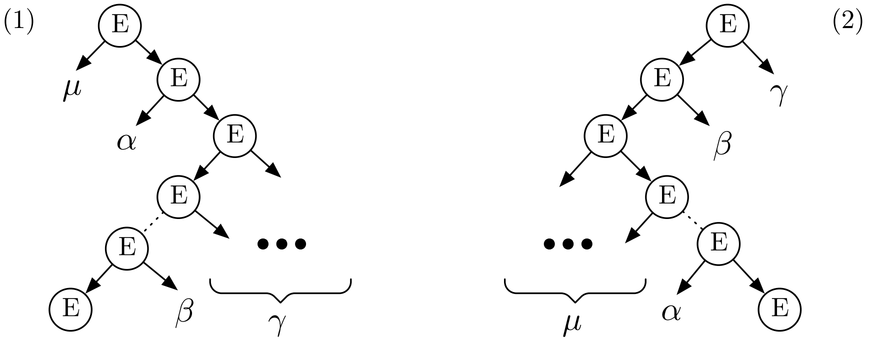
The reason why we only consider left- and right-recursive rules is that only left- and right-recursive ends can participate in an operator precedence ambiguity. In an operator precedence ambiguity involving two operators, the derivations differ in steps corresponding to the order of application of the respective operator rules. For example, for the binary operators in Section 5.2.1 we have the following leftmost derivations for the input `a+a*a`:

$$(1)\, E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E * E$$
$$(2)\, E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow a + E * E$$

The presence of two derivations for an operator ambiguity, and the fact that `>` only removes one of them, is the basic reasoning behind the safety of our operator precedence technique. More generally, for any two left- and right-recursive rules $E ::= E\beta$ and $E ::= \alpha E$ we have the following two leftmost derivations:

$$(1)\, \mu E \Rightarrow \mu\alpha E \underset{lm}{\overset{*}{\Rightarrow}} \mu\nu E\gamma \Rightarrow \mu\nu E\beta\gamma$$
$$(2)\, E\gamma \Rightarrow E\beta\gamma \underset{lm}{\overset{*}{\Rightarrow}} \mu E\beta\gamma \Rightarrow \mu\alpha E\beta\gamma \underset{lm}{\overset{*}{\Rightarrow}} \mu\nu E\beta\gamma$$

where both derivations have identical sub-derivations $\alpha \overset{*}{\Rightarrow} \nu$. The parse trees corresponding to these derivations have the shapes:



In the first parse tree, β binds stronger than α, and vice versa for the second parse tree. As can be seen, there can be an arbitrary distance ($\overset{*}{\Rightarrow}$) between the application of $E ::= \alpha E$ and $E ::= E\beta$. This captures the deep cases of operator precedence. If $E ::= \alpha E > E\beta$, the first derivation tree should be removed and vice versa. We discuss indirect cases in Section 5.3.5.

The semantics of associativity constructs, `left` and `right`, is similar to the one of the precedence. However, in contrast to precedence, we define associativity to affect only binary operators and apply only at one level and not arbitrary deep. The latter decision is based on the fact that we could not find a practical example, where deep application of associativity rules was useful, and where other precedence mechanism could not be used instead. We also support the `nonassoc` notation for defining non-associativity, e.g., for `E ::= E '>' E`, where no nesting of the same rule is allowed. In contrast to `left` and `right`, `nonassoc` is not safe as it removes sentences from the language.

5.3.2 Data-Dependent Grammars

Data-dependent grammars [41] are an extension of context-free grammars that support arbitrary computation, parameters, variable binding, and constraints. In a context-free grammar, a rule is defined as $A ::= \alpha$, where A (head) is a nonterminal, and α (body) is a possibly empty sequence of terminal and nonterminals. Data-dependent grammars allow definition of parametrized nonterminals, e.g., $A(p)$, similar to the way a function is defined. In addition to terminal and nonterminals, the body of rules in data-dependent grammars can have the following new types of symbols:

- Constraints of the form $[c]$. If c evaluates to false, the current parsing path terminates.
- Bindings of the form $x = A(a)$, where a is an argument to A, and x is a variable holding the value returned by the call $A(a)$.
- Arbitrary expressions of the form $\{e\}$.

Our data-dependent framework, presented in Chapter 3, also supports return values. An expression $\{e\}$ as the last symbol of a rule defines the return value.

Jim *et al.* [41] introduce the data-dependent automata to represent data-dependent grammars and use these automata to provide a stack-evaluation-based, nondeterministic operational semantics for data-dependent grammars. This semantics is very intuitive. For example, consider the following definition of a fixed-length iteration of a nonterminal `A`:

```
Iter(n) ::= [n > 0] Iter(n - 1) A
          | [n == 0] ϵ
```

Here, `Iter` gets an integer parameter `n` which is used to determine the choice of the alternative. If `n > 0`, the first alternative is selected, otherwise, the second. Using this definition, `Iter(5)` is much like a function call in a backtracking, recursive-descent parser. Direct implementation of this semantics of data-dependent grammars can result in exponential runtime and nontermination in presence of left-recursive rules. Therefore, to implement data-dependent grammars, Jim et al. use a modified Earley parsing algorithm, and we use our modified GLL parsing algorithm presented in Chapter 2.

5.3.3 Precedence

In this section, we show how grammars that specify precedence rules using `>` are translated into data-dependent counterparts. In our discussion we use the grammar of Figure 5.4 (left) as a running example. The translation scheme consists of the following steps:

Assign a precedence level to a rule A number pr_i, the *precedence level* of a rule, is assigned to each left- and/or right-recursive alternative of a nonterminal, where i indicates the i-th alternative. Numbering follows the reverse order of the alternatives and uses the current value of a counter. The counter starts from 1 and increments each time `>` is encountered.

In Figure 5.4 (left), nonterminal `E` has four left- and/or right-recursive alternatives. The first two (from the top) act as postfix operators, the third as a binary operator, and the fourth as a prefix operator. The number assigned to each alternative is shown in the comment next to the alternative (Figure 5.4, right). There are two observations. First, as precedence rules do not apply between two postfix or two prefix operators, `|` is used instead of `>` between the first two alternatives, resulting in the same number assigned to both of them. Second, the alternatives that are not left- or right-recursive do not get a number.

Pass a precedence level The nonterminal gets a parameter p, so that the precedence level of an alternative can be passed. The arguments to the left- and/or right-recursive ends of the i-th alternative are defined as follows. Each right-recursive alternative passes its precedence level to its right end: $E(p) ::= \alpha E(\text{pr}_i)$. Each left-recursive alternative passes the precedence level of a parent alternative to its left end: $E(p) ::= E(p)\alpha$. The argument 0 is passed to the nonterminal when it occurs at a position other than the left or right recursive end, i.e., where the precedence rules do not apply. In Figure 5.4 (right), `p` is passed to all the left ends, `2` to the right end of binary `'+'`, and `1` to the right end of prefix `'if'`.

Return a precedence level In addition to passing the precedence level to its right end, each right-recursive alternative also returns a value that depends on its precedence level. We distinguish the following two cases for a right-recursive alternative:

1. If there is a prefix operator of lower precedence than the alternative, the return expression is defined as follows:

   $E(p) ::= \alpha\ r{=}E(\mathrm{pr}_i)\ \{r{=}0\ ?\ \mathrm{pr}_i : \min(r, \mathrm{pr}_i)\}$,

   where variable r holds the value returned by the call to the right end. The construct _?_ : _ defines a conditional expression such that if the value of r is equal to 0, the alternative returns its precedence level, otherwise the value is defined as $\min(r, \mathrm{pr}_i)$. Intuitively, min will propagate the lowest precedence level upwards, in the chain of the recursive calls corresponding to the rightmost recursive ends.

2. If there is no such a prefix operator, the alternative simply returns its precedence level: $E(p) ::= \alpha\, E(\mathrm{pr}_i)\, \{\mathrm{pr}_i\}$.

Finally, we also need to provide a default return value for alternatives that are not right recursive, i.e., postfix operators or alternatives without recursive ends. We use 0 as the default value.

In Figure 5.4 (right), only the binary and prefix operators return a non-zero value: prefix `'if'` returns its precedence level, and binary `'+'` returns `r==0 ? 2 : min(r,2)` as there is prefix `'if'` which is of lower precedence than binary `'+'`.

Add constraints to a rule based on its precedence level Finally, each left-recursive alternative gets two constraints:

$$E(p) ::= [\mathrm{pr}_i \geq p]\ l{=}E(p)\ [l{=}0 \,\|\, l \geq \mathrm{pr}_i]\ \alpha,$$

where variable l holds the value returned by the call to the left end. The first constraint, $[\mathrm{pr}_i \geq p]$, is a precondition to the alternative. This constraint effectively excludes the current left-recursive alternative from the right end of a parent right-recursive alternative if the current alternative is of lower precedence than the parent one. Passing 0 to E makes any precondition true, therefore we refer to $E(0)$ as the unrestricted use of E. The second constraint, $[l{=}0 \,\|\, l \geq \mathrm{pr}_i]$, is a postcondition to the first symbol of the alternative. This constraint terminates the current left-recursive alternative if the value produced by the left end corresponds to a child alternative of lower precedence than the current alternative.

Discussion on Semantics and Implementation

We now discuss how our data-dependent encoding prevents undesired, precedence-violating derivation trees by restricting the left and right end of an alternative. Similar to Section 5.3.1, we consider a grammar with a left-recursive alternative $E ::= E\beta$ and a right-recursive alternative $E ::= \alpha E$. In addition, we assume that the grammar has other left- and right-recursive alternatives: $E ::= E\sigma_i$, $1 \leq i \leq m$, and $E ::= \gamma_j E$, $1 \leq j \leq k$. We use the following leftmost derivations:

(1) $E \Rightarrow E\beta \overset{*}{\underset{lm}{\Rightarrow}} \mu E\beta \Rightarrow \mu\alpha E\beta$

where $\overset{*}{\Rightarrow}$ indicates zero or more intermediate steps deriving the right-recursive alternatives $E ::= \gamma_j E$, such that $\gamma_1 \overset{*}{\underset{lm}{\Rightarrow}} \nu_1, \ldots, \gamma_k \overset{*}{\underset{lm}{\Rightarrow}} \nu_k$, and $\mu = \nu_1 \ldots \nu_k$ is a possibly empty sequence of terminals.

(2) $E \Rightarrow \alpha E \overset{*}{\underset{lm}{\Rightarrow}} \nu E \overset{*}{\underset{lm}{\Rightarrow}} \nu E\sigma \Rightarrow \nu E\beta\sigma$

where $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \nu$, and the second $\overset{*}{\Rightarrow}$ indicates zero or more intermediate steps deriving the left-recursive alternatives $E ::= E\sigma_i$, such that $\sigma = \sigma_m \ldots \sigma_1$ is a possibly empty sequence of terminals and nonterminals.

First, we consider derivation (1) and the case of direct nesting, i.e., zero intermediate steps. According to the precedence semantics of Section 5.3.1, the derivation step $E\beta \Rightarrow \alpha E\beta$ should only be valid if $E ::= \alpha E$ has the same or higher precedence than $E ::= E\beta$. We now look at how our encoding to data-dependent grammars achieves this. In our translation scheme, each left-recursive alternative $E ::= E\beta$ is translated into:

$$E(p) ::= [\mathrm{pr}_\beta \geq p]\ l{=}E(p)\ [l{=}0 \,\|\, l \geq \mathrm{pr}_\beta]\ \beta.$$

We consider an unrestricted call $E(0)$ (restricted calls $E(\mathrm{pr})$, $\mathrm{pr} > 0$, are discussed later in the context of derivation (2)). The precondition evaluates to true, and the alternative can only succeed if the postcondition to the recursive call is also true. As postfix operators and alternatives without recursive ends return 0, the postcondition permits these forms of alternatives at the left end. The postcondition, however, permits a right-recursive alternative $E(p) ::= \alpha E(\mathrm{pr}_\alpha)\ \{\mathrm{pr}_\alpha\}$ only if it is of the same or higher precedence than the current alternative, thus enforcing $\mathrm{pr}_\alpha \geq \mathrm{pr}_\beta$.

To enforce precedence rules at arbitrary depth (the case of multiple intermediate steps in the derivation), our translation scheme introduces min to the return expression of a right-recursive alternative. This propagates the lowest precedence level upwards, in the chain of the recursive calls corresponding to the rightmost recursive ends, to the left end of the current alternative. For example, consider the following chain of such recursive calls, where each call (→) is shown in the context of the respective right-recursive alternative and is made from its right end:

$$\gamma_1\ r_1{=}E(\mathrm{pr}_{\gamma_1}) \rightarrow \cdots \rightarrow \gamma_k\ r_k{=}E(\mathrm{pr}_{\gamma_k}) \rightarrow \alpha\ r{=}E(\mathrm{pr}_\alpha).$$

Here, r_j, $1 \leq j \leq k$, and r hold the value returned by the respective call. As our translation adds return expression $r = 0 \; ? \; \mathrm{pr}_{\gamma_j} : \min(r, \mathrm{pr}_{\gamma_j})$ to right-recursive rule $E ::= \gamma_j E$, these calls, when return, produce the following bindings: $r_k = \min(r, \mathrm{pr}_\alpha)$, $r_{j-1} = \min(r_j, \mathrm{pr}_{\gamma_j})$, $2 \leq j \leq k$, and finally, $l = \min(r_1, \mathrm{pr}_{\gamma_1})$. Thus, the postcondition above also requires that all $E ::= \gamma_j E$ and $E ::= \alpha E$ are of the same or higher precedence than $E ::= E\beta$.

The observant reader will note, however, that the return expression of a right-recursive alternative depends on whether there is a prefix operator of lower precedence. Below we discuss the role of the precondition to a left-recursive alternative. After that, it can be seen that it is sufficient to simply return the precedence level for a right-recursive alternative (no min is needed) if there is no prefix operator of lower precedence than the alternative.

Now, we consider derivation (2) and the case of zero intermediate steps in the second $\overset{*}{\Rightarrow}$. According to the precedence semantics of Section 5.3.1, the last derivation step should only be valid if $E ::= E\beta$ has the same or higher precedence than $E ::= \alpha E$. In our translation, each right-recursive alternative passes its precedence level to the right end: $E(p) ::= \alpha E(\mathrm{pr}_\alpha)$ (for brevity, we omitted the return expression). Given that only left-recursive alternatives are guarded with a precondition, call $E(\mathrm{pr}_\alpha)$ will try all prefix operators and alternatives without recursive ends. However, the call will only try the left-recursive alternative $E ::= E\beta$ if its precondition is true (see the translation above), thus enforcing $\mathrm{pr}_\beta \geq \mathrm{pr}_\alpha$.

To enforce precedence rules at arbitrary depth (the case of multiple intermediate steps in the second $\overset{*}{\Rightarrow}$), if the precondition to the left-recursive alternative is true, the precedence level of the parent alternative is passed to the left end of the current alternative. This way, the precedence level of the parent alternative, pr_α, also restricts the chain of recursive calls corresponding to the leftmost recursive ends: $E(\mathrm{pr}_\alpha)\sigma_1 \rightarrow \cdots \rightarrow E(\mathrm{pr}_\alpha)\sigma_m \rightarrow E(\mathrm{pr}_\alpha)\beta$, where each consecutive call to E is made from the left end of the respective left-recursive alternative. In our translation, each of these calls is guarded by the precondition: $\mathrm{pr}_{\sigma_i} \geq p$, $1 \leq i \leq m$, and $\mathrm{pr}_\beta \geq p$, thus enforcing all $E ::= E\sigma_i$ and $E ::= E\beta$ to be of the same or higher precedence than $E ::= \alpha E$.

5.3.4 Associativity

Using data dependency, associativity rules can be encoded in a similar way to precedence. We consider left- and right-associative rules, declared using `left` and `right`, and non-associative rules, declared using `nonassoc`. Our general scheme to handle both precedence and associativity consists of the following steps:

Assign a unique number to a rule specifying associativity We use the same counter as before. However, now, the counter also increments when an alternative specifying associativity is encountered, but only if this alternative shares the same precedence with the next or previous alternative. The current value of the counter is then assigned to the alternative, thus giving it a unique number within the same precedence group. All the alternatives within the same precedence group that do not specify associativity are assigned the same number. Consider $E ::= \alpha_4 > \alpha_3 \mid \alpha_2$ left $\mid \alpha_1 > \alpha_0$, where $\mid$ binds stronger than $>$, each $E ::= \alpha_i$ is left- and/or right-recursive, and $E ::= \alpha_2$ is binary. If the value of the counter when encountering the first $>$ (in the reverse order of the alternatives) is 1, the counter increments, and the number assigned to $E ::= \alpha_1$ is 2. The next alternative, $E ::= \alpha_2$, specifies associativity and has the same precedence as $E ::= \alpha_1$ and $E ::= \alpha_3$. Thus, the counter increments again, and the number assigned to $E ::= \alpha_2$ is 3. $E ::= \alpha_3$ is assigned 2, which is the same as for $E ::= \alpha_1$.

This way, alternatives of the same precedence are now described by a range $[\mathrm{pr}_i, \mathrm{pr}_j]$, $\mathrm{pr}_i \le \mathrm{pr}_j$, $i \le j$, where the i-th alternative is the first alternative after the last occurrence of >, and the j-th alternative is the alternative with the largest number before the next occurrence of >. We use $\mathrm{pr}_k \in [\mathrm{pr}_i, \mathrm{pr}_j]$ to refer to the number assigned to the alternatives that do not specify associativity within the group.

Pass the rule's unique number along with the precedence level The nonterminal gets two parameters, the first one to pass a precedence level, and the second one to pass its unique number. If a binary alternative is defined as left- or non-associative, its unique number is passed to its right end along with its precedence level: $E(p, p') ::= \alpha\, E(\mathrm{pr}_k, \mathrm{pr}_i)$, where pr_k is used as the alternative's precedence level. Otherwise, 0 is passed as the second argument to the right end: $E(p, p') ::= \alpha\, E(\mathrm{pr}_k, 0)$. All left-recursive rules of the nonterminal, i.e., binary and postfix operators, pass 0, along with the precedence level of a parent alternative, to its left end: $E(p, p') ::= E(p, 0)\, \alpha$. Passing 0 to the left end of an alternative prevents deep application of associativity rules, in contrast to precedence.

Return the rule's unique number along with the precedence level If a binary alternative is defined as right- or non-associative, its unique number is returned along with its precedence level. If there is no prefix operator of lower precedence than the alternative:

$E(p, p') ::= \alpha\ E(\mathrm{pr}_k, 0)\ \{(\mathrm{pr}_k, \mathrm{pr}_i)\}$ (right-associative)

$E(p, p') ::= \alpha\ E(\mathrm{pr}_k, \mathrm{pr}_i)\ \{(\mathrm{pr}_k, \mathrm{pr}_i)\}$ (non-associative)

where $(\mathrm{pr}_k, \mathrm{pr}_i)$ is a tuple expression. If there is such a prefix operator, the return expressions above are replaced with $(r.1\!=\!0\ ?\ \mathrm{pr}_k : \min(r.1, \mathrm{pr}_k), \mathrm{pr}_i)$, where variable r (see Section 5.3.3) holds the value returned by the call to the right end, and $r.1$ accesses the first element of the tuple. For all the other alternatives of the nonterminal, 0 is used as the second element of the tuple.

Add constraints to the rule based on its unique number If a binary alternative is defined as left- or non-associative, precondition $p' \ne \mathrm{pr}_i$ is also added to the alternative, resulting in $[\mathrm{pr}_k \ge p,\ p' \ne \mathrm{pr}_i]$, where the comma inside the brackets defines logical AND. If a binary alternative is defined as right- or non-associative, postcondition $l.2 \ne \mathrm{pr}_i$ is added to the alternative, resulting in $[l.1 = 0 \parallel l.1 \ge \mathrm{pr}_k,\ l.2 \ne \mathrm{pr}_i]$,

```
expr(p)
   ::= [7>=p] l=expr(p) [l==0||l>=7] '.' field          {0}
     | [6>=p] l=expr(p)[l==0||l>=6] r=expr(7)           {(r==0)? 6 : min(r,6)}
     |        '-' r=expr(5)                             {(r==0)? 5 : min(r,5)}
     | [4>=p] l=expr(p) [l==0||l>=4] '*' r=expr(5)      {(r==0)? 4 : min(r,4)}
     | [3>=p] l=expr(p) [l==0||l>=3] '-' r=expr(4)      {(r==0)? 3 : min(r,3)}
     | [3>=p] l=expr(p) [l==0||l>=3] '+' r=expr(4)      {(r==0)? 3 : min(r,3)}
     |        'if' expr(0) 'then' expr(2)               {2}
     | [1>=p] l=expr(p) [l==0||l>=2] ';' expr(1)        {1}
     |        '(' expr(0) ')'                           {0}
```

Figure 5.5: The translation of the OCaml excerpt from Figure 5.2 into a data-dependent grammar.

where variable l (see Section 5.3.3) holds the value returned by the call to the left end, and $l.1$ and $l.2$ access the first and second elements of the tuple, respectively.

Associativity groups The general scheme above is also applicable for binary alternatives forming an associativity group. For example, two binary operators, such as + and - (Figure 5.2), can be specified to be left-associative with respect to each other. In such cases, the left or/and right ends of a binary alternative in an associativity group must exclude the other binary alternatives of the group including the alternative itself. To encode this, all the binary alternatives of the associativity group are assigned the same unique number, say pr_m. This way, associativity related constraints, $p' \neq \text{pr}_m$ and $l.2 \neq \text{pr}_m$, effectively exclude all the alternatives of an associativity group from the left and/or right ends of the alternatives.

Optimization

Is it always necessary to operate with two arguments and tuples when both precedence and associativity rules are used? The answer is no. The translation can use one argument and return a single number when alternatives specifying associativity do not share the same precedence with other alternatives, and, in case of an associativity group, when alternatives inside the group do not share the same precedence with alternatives outside the group. This corresponds to cases where $\text{pr}_i = \text{pr}_j$, $i \leq j$, and when our general scheme produces, for example, the following translation for a left-associative alternative (here, we assume the case of no prefix operator of lower precedence):

$$E(p, p') ::= [\text{pr}_i \geq p, p' \neq \text{pr}_i]\ \alpha\ E(\text{pr}_i, \text{pr}_i)\ \{(\text{pr}_i, 0)\}$$

Instead, in such cases, we simplify the translation to:

$$E(p) ::= [\text{pr}_i \geq p]\ \alpha\ E(\text{pr}_i + 1)\ \{\text{pr}_i\}$$

This translation uses only one argument and passes the precedence level plus one to the right end, thus also excluding the alternative itself and disallowing right-associative derivation trees. The call $E(\mathrm{pr}_i\!+\!1)$ will propagate its argument to the left end of left-recursive alternatives. However, in this case, it cannot lead to deep application of the associativity rule, as only left-recursive alternatives of (strictly) higher precedence than $E ::= \alpha E$ can be tried, thus disallowing deep nesting of $E ::= \alpha E$. Similarly, for the left end of a right-associative alternative, the simplified translation is:

$$E(p) ::= [\mathrm{pr}_i \geq p]\ l{=}E(p)\ [l{=}0 \,\|\, l \geq \mathrm{pr}_i{+}1]\ \alpha,$$

where $\mathrm{pr}_i\!+\!1$, the lower bound on l, excludes from the left end alternatives of lower precedence and the alternative itself, thus disallowing left-associative derivation trees. The translation of the OCaml excerpt from Figure 5.2 into a data-dependent grammar is shown in Figure 5.5. This translation requires only one parameter.

5.3.5 Support for Indirect Cases

To extend the derivations of Section 5.3.1 to indirect cases, we consider more general forms of left- and right-recursive rules: $E ::= Y\beta$ and $E ::= \alpha X$, where $Y \overset{*}{\underset{lm}{\Rightarrow}} E\sigma$ and $X \overset{*}{\underset{lm}{\Rightarrow}} \tau E$. Then, we have:

$$(1)\ \mu E \Rightarrow \mu\alpha X \overset{*}{\underset{lm}{\Rightarrow}} \mu\nu\tau E \overset{*}{\underset{lm}{\Rightarrow}} \mu\nu\tau E\gamma \Rightarrow \mu\nu\tau Y\beta\gamma \overset{*}{\underset{lm}{\Rightarrow}} \mu\nu\tau E\sigma\beta\gamma$$

$$(2)\ E\gamma \Rightarrow Y\beta\gamma \overset{*}{\underset{lm}{\Rightarrow}} E\sigma\beta\gamma \overset{*}{\underset{lm}{\Rightarrow}} \mu E\sigma\beta\gamma \Rightarrow \mu\alpha X\sigma\beta\gamma \overset{*}{\underset{lm}{\Rightarrow}} \mu\nu\tau E\sigma\beta\gamma$$

In other words, nonterminals Y and X indirectly derive the left and right E-end, respectively. In addition, sub-derivations corresponding to the second and the last $\overset{*}{\Rightarrow}$ in (1) and (2) permit multiple intermediate nonterminals. Thus, in general, the rules containing the left and right E-end, such as $Z ::= E\theta_1$ and $W ::= \theta_2 E$, may have a different head (Z and W) than Y and X.

In our translation, we rely on reachability analysis that computes indirectly derivable left and right ends. The basic idea of our translation is to propagate the precedence level to/from the nonterminal's indirect left ($Z ::= \underline{E}\theta_1$) and right ($W ::= \theta_2\underline{E}$) ends via intermediate nonterminals by passing and returning values. Our translation adds parameter p_E to Y, X, Z and W. The argument passed to X or Y, and propagated via W or Z, depends on how X or Y is used in E. We distinguish two cases: (a) X and Y represent the same nonterminal that occurs in E as both the left and right end; and (b) X and Y represent distinct nonterminals, Y occurs in E only as the left end, and X only as the right end. In case (a), p_E is a tuple that encodes the left and right uses of X as follows:

$$E(p) ::= X((p,\$))\,\beta, \quad E(p) ::= \alpha\, X((\$,\mathrm{pr}_\alpha))$$

where the first element of the tuple is undefined (we use \$ for undefined values) when X is the right end, and the second element of the tuple is undefined when X is the left end. The other elements of the tuple are defined as if X was E. These arguments are propagated to Z and W, via X, and are used as follows:

$$Z(p_E) ::= E(p_E.1\!=\!\$\,?\,0\!:\!p_E.1)\,\theta_1$$

$$W(p_E) ::= \theta_2\, E(p_E.2\!=\!\$\,?\,0\!:\!p_E.2)$$

where $p_E.1$ and $p_E.2$ access the first and the second element of the tuple, respectively. In other words, the left end of $Z ::= E\theta_1$ is only restricted if X is called as the left end, and the right end of $W ::= \theta_2 E$ is only restricted if X is called as the right end.

In case (b), it is possible to directly use p and pr_α without the need to introduce a tuple. In the following, we only focus on case (a) as a more general case. Also, we only consider the case of one parameter to E as our discussion can be straightforwardly generalized to the case of two parameters.

In addition to getting parameters and arguments, X, Z and W return values. This way, the precedence level can be propagated upwards from the indirect left and right ends, via Z or W, to the uses of X. In case (a), the return values are also tuples:

$$Z(p_E) ::= l{=}E(p_E.1\!=\!\$\,?\,0\!:\!p_E.1)\ \theta_1\ \{(l,\$)\}$$

$$W(p_E) ::= \theta_2\ r{=}E(p_E.2\!=\!\$\,?\,0\!:\!p_E.2)\ \{(\$,r)\}$$

and are used in E as follows:

$$E(p) ::= x{=}X((p,\$))\ [x.1{=}\$\,\|\,(\mathrm{pr}_\beta\!\geq\! p,\ x.1{=}0\,\|\,x.1\!\geq\!\mathrm{pr}_\beta)]\ \beta$$

$$E(p) ::= \alpha\ x{=}X((\$,\mathrm{pr}_\alpha))\ \{x.2\!=\!\$\,?\,0\!:\!(x.2\!=\!0\,?\,\mathrm{pr}_\alpha\!:\!\min(x.2,\mathrm{pr}_\alpha))\}$$

where $x.1 = \$$ and $x.2 = \$$ check the presence of the indirect left and right end, respectively. In the second case, the check affects the return value: 0 if the value of $x.2$ corresponds to an alternative without the indirect right end for E, otherwise the value computed as if X was E (here, we only show the return expression for the case where there is a prefix operator of lower precedence than $E ::= \alpha X$). In the first case, the check affects pre- and postconditions. In the general case, the condition $\mathrm{pr}_\beta \geq p$ has to become a postcondition, except for the case when $x.1$ is never equal to \$, and none of pre- and postconditions is triggered if the value of $x.1$ corresponds to an alternative without the indirect left end for E.

5.3.6 Comparison with our Previous Translation Scheme

Finally, we discuss the design decision in our translation scheme that relates to the use of both parameters and return values. The use of return values is the main difference between the translation scheme we propose in this chapter and the grammar rewriting technique of Chapter 4. Specifically, to restrict the left and right ends of a nonterminal, we pass an argument to a right end, and propagate the argument passed by a parent alternative and use return values to restrict a left end. General parsing algorithms are efficient in dealing with left-recursive rules. For example, in GLL, left recursion is terminated after the first recursive call at the same input position, allowing non-left-recursive rules to produce results. Then, the left-recursive rules are re-tried, in a form of a loop, as long as new results can be produced.

Our experience with the grammar rewriting technique of Chapter 4 shows that the introduction of new, indexed nonterminals for the left ends, which also involves copying the rules to the new nonterminals, directly affects the efficiency in dealing with left recursion. In particular, it increases the stack of leftmost calls, corresponding to the new nonterminals, and does not allow sharing of parsing results corresponding to the copied rules. Our previous translation scheme introduces the same inefficiency problem as the rewriting technique. In that scheme, we do not use return values to restrict left ends, and the use of parameters and arguments for left ends essentially simulates introduction of indices to the left ends.

In contrast to the rewriting and our previous scheme, our current translation does not increase the stack of leftmost calls, as the argument of a parent alternative is passed to the left ends, thus allowing termination of left recursion as soon as possible. When the left-recursive alternatives are further re-tried in a loop, there is just an extra, precedence-related condition that needs to be checked. For right-recursive rules, however, the context of the current alternative can be used to restrict its right end, and therefore, our translation passes the precedence level to the right end of the alternative. In practice, we observed that the median and maximum speedup of parsers for Java using our new translation compared to the rewriting and previous translation are (1.5, 2.5) and (1.7, 3), respectively.

5.4 Evaluation

In this section we evaluate the performance of our approach to operator precedence disambiguation using our GLL-based implementation of data-dependent grammars [3]. For the evaluation we use the grammars of Java and OCaml[3]. The experiments were carried out on a machine with a quad-core Intel Core i7 2.6 GHz CPU and 16 GB of memory, running Mac OS X 10.10.5 and a 64-Bit Oracle HotSpot™ JVM version 1.8.0_51. Each file was parsed 10 times and the mean running time (CPU user time) was reported. The three first runs of each file were skipped to allow for JIT optimization.

5.4.1 Java

We have chosen the grammar of Java 7 from the main part of the Java language specification [32]. This grammar has an unambiguous, left-recursive expression part that encodes operator precedence by introducing new nonterminals for each precedence level. We have replaced the expression part of the Java specification grammar with a natural expression grammar, and specified operator precedence and associativity using `>`, `left` and `right`. We parsed 7449 files from the source distribution of JDK 1.7.0_60-b19. All files parsed successfully and without ambiguity.

[3] https://github.com/iguana-parser/grammars

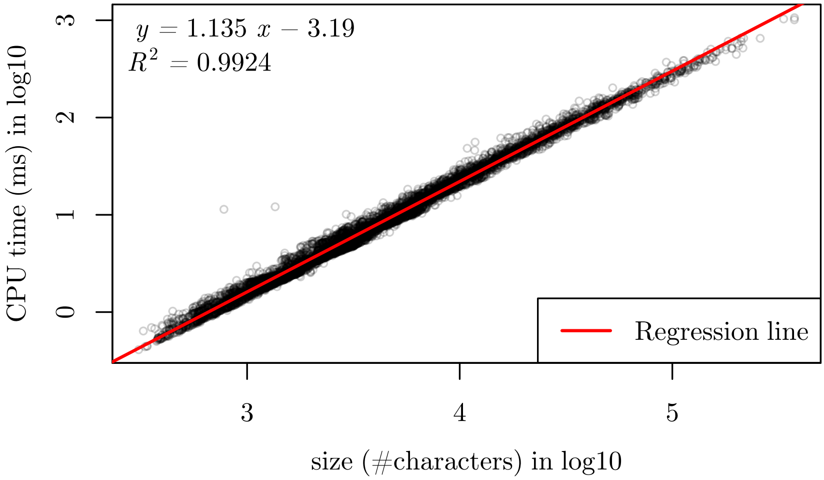


Figure 5.6: Running time of Iguana on the natural grammar of Java.

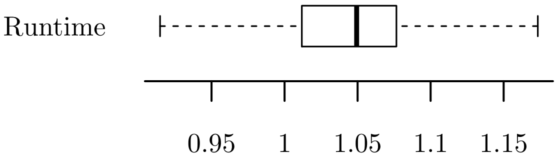


Figure 5.7: Runtime performance of Iguana using the natural grammar of Java vs. the specification grammar of Java.

Figure 5.6 shows the results of parsing Java files in a log-log (base 10) plot. The goodness of the fit is indicated by the R^2 value of 0.9924, and the equation of the regression line (log-log scale) is written in the plot. As the regression is calculated after a log transform of the original data, and the coefficient (1.135) is close to 1, we can conclude that the parser runs nearly linearly ($y \approx x^{1.135}$) on the natural grammar of Java.

To compare the speed of parsing with the natural grammar of Java that handles operator precedence at runtime, and the specification grammar of Java, we ran Iguana on the same set of 7449 Java source files. The relative runtime performance of the parser for the natural grammar of Java vs. the parser for the specification grammar of Java is shown in Figure 5.7. As can be seen, the median difference is 1.05, meaning that our approach to operator precedence is only on average 5% slower than for the specification grammar.

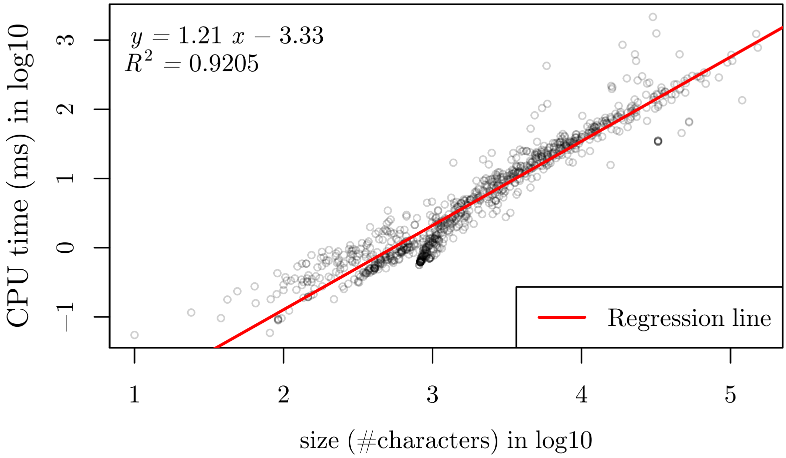


Figure 5.8: Running time of Iguana on the grammar of OCaml.

5.4.2 OCaml

Compared to Java, the OCaml language specification takes a very different approach to specifying its grammar. The expression grammar is ambiguous, and operator precedence and associativity rules are specified in a table, similar to Figure 5.1. We used the ambiguous expression grammar of OCaml and specified operator precedence and associativity of its operators using `>`, `left`, and `right`. We have parsed 945 files from the source distribution of the OCaml compiler version 4.02. From 945 files, 894 (94%) parse successfully and without operator precedence ambiguity. Figure 5.8 shows the running time of parsing these files. The goodness of the fit is indicated by the R^2 value of 0.9205, and the equation of the linear regression (log-log scale) is written in the plot. As can be seen, the running time shows a near-linear behavior ($y \approx x^{1.21}$), as the coefficient value (1.21) is close to 1.

OCaml, compared to Java, is a much more difficult language to parse. First, the syntax is ambiguously specified, and in many parts of the specification, the discussion of the desired parse tree is not precise enough. More importantly, there are syntactic extensions to OCaml which clash with the original syntax, and it is not clear if these extensions should only be enabled via a special flag to the compiler. To support a wider range of OCaml programs, we have incorporated some of the syntactic extensions into the grammar, and in many places we had to consult the LALR grammar of OCaml to determine how some parts should be disambiguated. For this evaluation, we also used the `ocamlyacc` grammar of OCaml, used by the OCaml compiler, and `camlp4`, an extensible syntax system for OCaml. However, even with these two parsers, we could not parse all the 945 `.ml` files. It is most likely that we are not aware of a configuration or flag while parsing those files.

Figure 5.9 (left) shows how many files from the source distribution of OCaml could

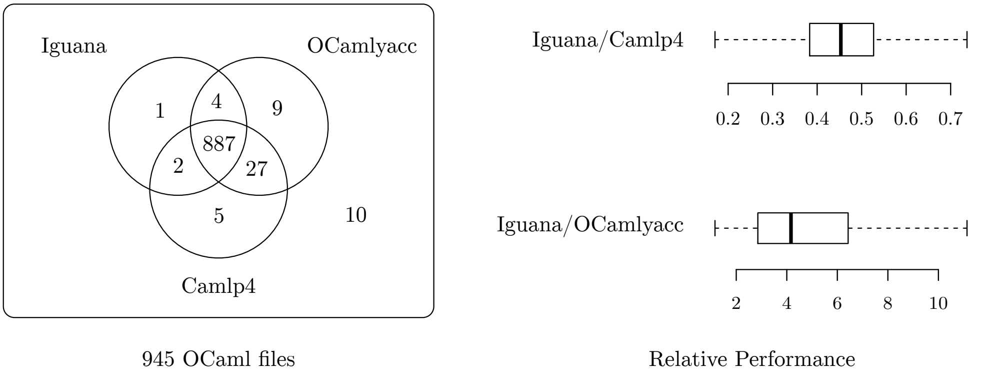


Figure 5.9: Distribution of the OCaml files parsed by each parser (left), and the relative performance of Iguana compared to `ocamlyacc` and `camlp4` (right).

be parsed by each parser. 10 files could not be parsed by any parser, but the majority of files, 887 (93%), could be parsed by all the parsers. Both `ocamlyacc` and `camlp4` use a separate lexing phase before parsing. Therefore, a performance comparison with a character-level grammar would not be fair. For performance comparison with `camlp4` and `ocamlyacc`, we used the context-aware [91] version of our OCaml grammar. See Chapter 3 for a discussion of context-aware scanning in Iguana.

The results of performance comparison are shown in Figure 5.9 (right). Each box plot shows the relative runtime of Iguana compared to the runtime of `camlp4` or `ocamlyacc`, for all 887 files that all parsers can successfully parse. As can be seen, the median running time of Iguana compared to `camlp4` is 0.45, meaning that Iguana is on average 2.2 times faster. The median running time of Iguana compared to `ocamlyacc` is 4.16, meaning that it is on average 4.16 times slower.

5.4.3 Other Ambiguities in OCaml

For parsing the expression part of OCaml unambiguously, only specifying operator precedence is not enough. There are some other kinds of ambiguity in OCaml which we discuss here.

Overlapping rules Consider the simplified excerpt of OCaml, augmented with operator precedence constructs, in Figure 5.2. For this grammar, the input string `a-a` is ambiguous with two derivation trees that correspond to the following groupings: `a(-a)` (the function application of `a` on `-a`) and `a-a` (binary minus). Although this ambiguity looks similar to operator precedence ambiguity, it cannot be disambiguated by using `left`, `right`, and `>`. To resolve this ambiguity, we use an except construct, which disallows the derivation of a certain rule at a certain grammar position. In this case, we can write `expr ::= expr expr !umins`, where `uminus` refers to the unary minus rule. This definition effectively disallows unary minus to be derived at the right-most `expr` of a function application.

Longest match ambiguities Another ambiguity that happens in the expression part of OCaml is related to nested patterns. For example consider the following OCaml pseudo-code:

```
let f = function
        | 0 -> match ... with
              | a -> ...
              | b -> ...
```

Even with specifying all operator precedence rules, and applying them in a deep and indirect setting, this sentence remains ambiguous. The reason is that the `b`-case can belong either to function `f` or to the `match` of the `0`-case. The OCaml language specification states that a pattern matching construct extends as long as possible (longest match). We resolved this issue by adding a custom follow restriction that bypasses layout: `... pattern-matching !>>> '|'`. These constructs are explained in more detail in Chapter 3.

Dangling else ambiguity Finally, we discuss the infamous dangling else ambiguity, which occurs between rules of the form:

```
Stmt ::= 'if' Expr 'then' Stmt
       | 'if' Expr 'then' Stmt 'else' Stmt
```

As can be seen, both rules have right-recursive ends, and the first rule is a prefix of the second rule. The dangling-else ambiguity, although looks very similar to operator precedence ambiguity, does not fit our semantics of operator precedence because both rules involved in the ambiguity have only right-recursive ends. Recall that in our operator precedence semantics, `>` triggers when one of the two rules is left- and the other one is right-recursive. In fact, the dangling-else ambiguity is an instance of longest match ambiguity, for which we use a follow restriction: `Stmt ::= 'if' Expr 'then' Stmt !>>> 'else'`.

5.5 Related Work

Throughout this chapter, we discussed the Yacc- and SDF-style operator precedence semantics, which we do not repeat here. In this section we discuss a number of related work that are directly related to our solution and inspired us the most.

5.5.1 Parsing OCaml

In Section 5.2 we motivated our new approach to operator precedence using examples of OCaml. The first question that comes to mind is how OCaml is actually parsed in practice.

OCamlyacc The parser for the OCaml compiler is written using ocamlyacc[4], a port of Yacc to OCaml. As we showed in Section 5.2, the Yacc-style resolution of operator precedence ambiguity can deal with all difficult cases in OCaml, provided that the grammar is LALR. This means that the grammar used for the OCaml compiler is not the natural, highly ambiguous specification grammar, rather it is an LALR version. Consider the following grammar rules, which are taken from the expression part of the OCaml specification grammar. The alternatives of expr are ordered based on precedence, with the highest precedence on top.

```
expr ::= expr '#' method-name
       | expr argument+
       | 'let' 'rec'? let-binding ('and' let-binding)∗ 'in' expr

argument ::= expr | '∼' label-name ':'  expr
```

The LALR counterpart of the rules above is as follows:

```
expr : simple_expr %prec below_SHARP
     | simple_expr simple_labeled_expr_list
     | let_bindings IN seq_expr;

let_bindings : let_binding
             | let_bindings and_let_binding;

let_binding: LET rec_flag let_binding_body;

rec_flag: REC
        | // empty;

simple_labeled_expr_list : labeled_simple_expr
                         | simple_labeled_expr_list labeled_simple_expr;

labeled_simple_expr : simple_expr %prec below_SHARP
                    | label_expr;

label_expr: LABEL simple_expr %prec below_SHARP;
```

As can be seen, the grammar is larger and contains many other nonterminals. Part of this verbosity is due to lack of support for EBNF in Yacc, e.g. simple_labeled_expr_list and rec_flag. Some of operator precedence information is encoded declaratively, e.g., expr: simple_expr is lower than #, but some others are encoded by using new nonterminals, e.g., simple_expr.

Moreover, the lexer used for the LALR grammar of OCaml is handwritten. This allows to hide some peculiarities in the syntax of OCaml from the LALR parser generator, e.g., how labeled arguments are parsed. OCaml also supports nested comments, which are dealt with in the lexer. It appears that the frontend for the OCaml compiler has been developed with considerable effort. The benefit of such a parser is that it is very fast. In contrast, our approach to parsing OCaml is fully

[4] http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html

declarative. The user encodes the specification grammar of OCaml using a single formalism for both lexical and context free parts, and resolves ambiguities using declarative disambiguation constructs.

Camlp4 Camlp4[5] (and Camlp5[6]), which stands for Caml Preprocessor and Pretty-Printer, is a system for writing extensible syntax for programming languages. Camlp4 is mostly used to allow syntactic extensions to OCaml programs.

Camlp4 uses a top-down recursive-descent parsing technique that interprets an in-memory representation of a grammar, and a handwritten lexer that conforms to the lexical syntax of OCaml. Camlp4 also allows the user to write a natural expression grammar, and to declaratively specify operator precedence and associativity. For example, an expression grammar for floating point arithmetic expressions, containing `'+'`, `'-'` and `'**'`, where `'**'` is right-associative and has higher precedence than left-associative `'+'` and `'-'`, can be encoded in Camlp4 as follows:

```
expr: [ "minus" LEFTA
        [ x = SELF; "+"; y = SELF -> x +. y
        | x = SELF; "-"; y = SELF -> x -. y]
      | "power" RIGHTA
        [ x = SELF; "**"; y = SELF -> x ** y]
      | "simple"
        [ x = INT -> float_of_int x ]];
```

`SELF` in this example refers to the `expr` nonterminal itself. The operator precedence scheme in Camlp4 works as follows. The parser tries alternatives in order, and each alternative that can parse at least one token is matched. The grammar is internally left-factorized to facilitate one token lookahead.

To deal with left-recursive calls, Camlp4 divides rules into two groups: `start` and `continue`. If a rule is left-recursive, i.e., starting with the nonterminal head or `SELF`, the rule is parsed using the `continue` parser, otherwise, using the `start` parser. In the example above, the `"simple"` rule belongs to the start group, while the other rules belong to the `continue` group. Parsing starts by calling the start function, and then a continue parser, based on the precedence and associativity level, is called with the value of the start function as argument. This approach to operator precedence can parse OCaml, but it is tied to the way left-recursion is implemented in a deterministic LL(1)-like parsing strategy.

5.5.2 General Parsers

In this section we discuss a number of operator precedence techniques that are implemented in context of general parsing.

Dypgen Dypgen[7] is a parser generator written in OCaml that allows definition of extensible grammars. Dypgen is based on GLR parsing and can handle all context-free

[5] https://github.com/ocaml/camlp4

[6] http://camlp5.gforge.inria.fr/

[7] http://dypgen.free.fr/

grammars. A distinguishing feature of Dypgen is that it natively allows passing values though parsing states. Dypgen allows definition of operator precedence via precedence *relations*. For example, consider an expression grammar consisting of `'*'` and `'+'` operators, where `'*'` has higher precedence, and both operators are left associative. This grammar, along with precedence relations, can be encoded in Dypgen as:

```
expr: INT                        p1
    | expr(<=p2) + expr(<p2)     p2
    | expr(<=p3) * expr(<p3)     p3
```

The precedence relation for this grammar is defined as `p1<p2<p3`, where `pi` is the precedence of the `i`th rule. The semantics of passing values in Dypgen is as follows. When a rule is reduced, its precedence is returned. Consequently, a shift action can only happen when the precedence returned from the previous reduce action matches the condition in the body of rules, e.g., `(<=p2)`.

Dypgen does not provide high-level notation for specifying operator precedence, and the user has to manually encode operator precedence using relations and conditions in the body of rules. In addition, the semantics of passing values in Dypgen is bound to the underlying LR automaton. In contrast to Dypgen, we provide high-level notation for specifying operator precedence, and desugar them into data-dependent grammars. Jim and Mandelbaum [40] report that they could implement data-dependent grammars on top of GLR parsing, by mapping to Dypgen's native features. Therefore, Dypgen can be used as a backend to realize our approach to operator precedence in GLR parsing.

Elkhound Elkhound [61] is a fast GLR parser generator that switches to the machinery of an LR parser on deterministic parts of the grammar. For dealing with operator precedence, Elkhound essentially uses the same approach as Yacc: shift/reduce conflicts are resolved by precedence and associativity of operators. However, because Elkhound is based on GLR parsing, it does not need to resolve all shift/reduce conflicts while parsing. Conflicts that do not correspond to precedence ambiguity are left intact and are effectively explored in parallel by GLR. Moreover, Elkhound uses a separate lexing phase, which discards layout. This allows correct resolution of precedence ambiguity in conflicting states.

Is it possible to use Elkhound's way of dealing with operator precedence in a scannerless setting where layout is part of the grammar? The answer is yes, but we need to put layout nonterminals after each terminal, as in [73] or [47]. The SDF-style layout insertion, i.e., between each two symbols in body of rules, does not work with shift/reduce way of resolving precedence ambiguity. In a conflicting state, the parser needs to decide to shift based on the next operator, but this operator is hidden behind a layout nonterminal.

ANTLR ANTLR [70] is a popular recursive-descent parser generator. Starting from version 4, ANTLR supports left-recursive rules and enables global backtracking using the ALL(*) strategy [70]. ANTLR 4 supports all context-free grammars except the ones with indirect or hidden left recursion. ANTLR 4 does not natively deal with left recursion, rather it uses a left-recursion transformation under-the-hood, and then transforms the trees back to the ones of the original, natural grammar. ANTLR 4 is not a general parser in the sense that it cannot deliver all the derivation trees in case of ambiguity, rather it uses an implicit ambiguity resolution scheme, in which the ambiguities are resolved based on the order of alternatives. Note that as ANTLR 4 uses a global backtracking scheme, it does not have the quirks of PEG-style [25] backtracking.

The support for left recursion and operator precedence in ANTLR are interwoven. When ANTLR rewrites left-recursive rules, it always adds precedence and associativity information to the rewritten rules: all rules are left-associative by default, and earlier alternatives have higher precedence. For example, an expression grammar containing `'+'` and `'%'` where both operators are left-associative and `'%'` has higher precedence than `'+'` is transformed to the following grammar [70]:

```
E[pr] ::= id ({3 >= pr}? '\%' E[4] | {2 >= pr}? '+' E[3])*
```

This transformation scheme is known as *precedence climbing* which mimics Clarke's technique [14], and requires a non-left-recursive grammar. Left-associative derivation trees, however, require left recursion. In ANTLR 4, a flat list, resulting from the expansion of Kleene star, is interpreted as left-associative. Moreover this technique does not allow associativity groups for operators that have the same precedence, but are left- or right-associative with respect to each other. One way to get left-associative groups in ANTLR 4 is to group operators: `E ::= E ('+'|'-') E`.

Our approach to translation of operator precedence resembles precedence climbing, in the sense that precedence level is passed, and illegal derivations are excluded using predicates. However, our approach works in present of left recursion, thus being able to natively construct parse trees that conform to the original grammar. In addition, we also support associativity groups. Finally, our approach is fully declarative, and no default precedence or associativity is applied: if the user writes a partially disambiguated grammar, by not specifying the precedence or associativity of some operators, the parser returns all the ambiguities.

Dynamic operator precedence In programming languages such as Prolog it is possible to redefine the precedence of operators at runtime. Such systems are fundamentally different from other related work we discussed, in the sense that the user does not directly work with the syntax of expressions. Prolog and similar dynamic operator precedence approaches use operator precedence grammars [23] to dynamically store precedence relationships in a table, and then interpret it. As the user does not have access to grammar of expressions in such dynamic operator precedence systems, the expressivity limitations of operator precedence grammars is not a problem. Favero [21] presents a detailed, step-by-step analysis of how dynamic operator precedence systems can be implemented.

Danielsson and Norell [16] present an approach for parsing mixfix operators for a user-defined operator precedence setting. Mixfix operators are a generalization of prefix and postfix operators. For example, `if-then-else` can be considered as a mixfix operator: `if [] then [] else []` which has three places for operands. This way of specifying operators is beneficial for systems in which the precedence and associativity of operators defined globally based on their token, not the rule in which they appear. The operator precedence and associativity information in this approach is encoded in a precedence graph. To deal with user-defined operator precedence, expressions are treated as flat lists of tokens, and then parsed again when the precedence graph is composed at runtime. The semantic of this approach is the same as in SDF2, by applying one level relationship between parents and children. This means that, for example, a unary prefix operator with lower precedence will be rejected on right of a binary operator.

Other approaches So far, we discussed operator precedence techniques that are used in parsing tools. However, there are many other approaches which have not found their way in practice, or the tools that implemented them are not available any more. Most notable approaches in this category are by Aasa [1], Thorup [84], and Visser [93]. All these approaches use a grammar rewriting technique to implement operator precedence.

Aasa [1] introduces an approach for declarative specification of operator precedence by assigning weights to operators in a parse tree. These weights define parse trees that are precedence correct. Aasa's approach is safe, and correctly identifies deep cases of operator precedence. A shortcoming of this approach is that operator precedence is defined token-based and globally. Therefore, operators in this approach have to be unique.

Thorup [84] presents a general grammar transformation technique that gets a grammar and a set of illegal sub-parse trees as input, and produces a grammar that does not yield derivation trees that are illegal. This approach can be used to implement operator precedence, if ambiguities in operator precedence are specified as tree patterns. As some precedence ambiguity patterns are arbitrary deep, it is not clear how they can be specified in this approach.

Visser [93] introduces a transformation from context-free grammars to character-class grammars, by applying the SDF2 semantics. Visser's approach is similar to the rewriting approach presented in Chapter 4, with the difference that instead of using indexed nonterminals and operating on the grammar, it replaces nonterminals with a set of integers and removes violating patterns. Because this approach has the underlying SDF2 semantics, it may remove sentences from the language if there is no ambiguity, and cannot deal with deep cases.

5.6 Conclusions

In this chapter we presented a technique for implementing a declarative specification of operator precedence, by desugaring to data-dependent grammars. Our approach is efficient and can deal with intricate cases of operator precedence found in functional programming languages such as OCaml. We evaluated our approach using the Iguana parsing framework, and the results show that our approach can be practical. Other general parsing algorithms such as GLR or Earley can also be used as a backend for data-dependent grammars, and adapt our operator precedence approach.

Chapter 6

Iguana: a Practical Data-dependent Parsing Framework[1]

Summary. Data-dependent grammars extend context-free grammars with arbitrary computation, variable binding, and constraints. These features provide the user with the freedom and power to express syntactic constructs outside the realm of context-free grammars, e.g., indentation rules in Haskell and type definitions in C. Data-dependent grammars have been recently presented by Jim *et al.* as a grammar formalism that enables construction of parsers from a rich format specification. Although some features of data-dependent grammars are available in current parsing tools, e.g., semantic predicates in ANTLR, data-dependent grammars have not yet fully found their way into practice.

In this chapter we present Iguana, a data-dependent parsing framework, implemented on top of the GLL parsing algorithm. In addition to basic features of data-dependent grammars, Iguana also provides high-level syntactic constructs, e.g., for operator precedence and indentation rules, which are implemented as desugaring to data-dependent grammars. These high-level constructs enable a concise and declarative way to define the syntax of programming languages. Moreover, Iguana's extensible data-dependent grammar API allows the user to easily add new high-level constructs or modify existing ones. We have used Iguana to parse various real programming languages, such as Java, C#, Haskell, and a significant subset of OCaml. In this chapter we describe the architecture and features of Iguana, and provide an extensive performance evaluation of Iguana, by comparing it to ANTLR 4, on a large number of Java source files.

[1] This chapter is an extension of our tool paper that was originally published as: A. Afroozeh and A. Izmaylova. Iguana: A Practical Data-Dependent Parsing Framework. In *Proceedings of the 25th International Conference on Compiler Construction*, CC '16, pages 267–268, Springer, 2016.

6.1 Introduction

Parsing is the first step in many tasks, such as compiler construction and static analysis, that deal with source code. When building a (domain-specific) programming language, it is desirable to quickly build a parser and spend most of the effort in other phases such as name resolution and type checking. Despite the long investment in theory and practice of parsing, constructing parsers remains a difficult task, often left to the experts. Syntax of most programming languages cannot be directly expressed using current parsing tools that are based on pure (deterministic) context-free grammars, and there is a need for (manual) grammar modification, and various hacks in the lexer and parser.

We advocate a declarative approach [34, 49] to syntax definition, where the user defines the syntax of a language using context-free grammars, and specifies the (un)desired parse trees using declarative disambiguation constructs. In Chapter 3 we described our parsing framework that can deal with many challenges of parsing programming languages. We based our parsing framework on data-dependent grammars [41], instead of pure context-free grammars. Data-dependent grammars extend context-free grammars with arbitrary computation, variable binding and constraints. In essence, these features allow the user to simulate handwritten parsers. Our data-dependent parsing framework is implemented on top of our modified version of the Generalized LL (GLL) parsing algorithm [3]. In particular, we extended GLL to support environment manipulation and return values.

In Chapter 3 we proposed to use data-dependent grammars as an intermediate layer for a parser-independent implementation of various disambiguation constructs. As data-dependent grammars are rather low-level for expressing many disambiguation constructs, we also demonstrated how several high-level syntactic constructs, e.g., operator precedence and indentation-sensitive constructs, can be desugared to data-dependent grammars. This provides a uniform view on disambiguation, and allows the user to add new syntactic constructs without the need to modify the machinery of a parsing algorithm.

In this chapter we present the architecture and features of Iguana, our data-dependent parsing framework. Iguana has been developed during the last years as our research playground and has been used in evaluating the results of the previous chapters. We present the architecture of Iguana, discuss its textual syntax, and also present the results of our performance comparison with ANTLR 4, one of the most popular parsing tools. The results show that Iguana is practical for parsing real programming languages.

6.2 Architecture

We designed Iguana to be extensible and flexible. Iguana can be used as a standalone library, where the language engineer uses its textual syntax to define a grammar. Iguana can also be used as the underlying parsing library for language workbenches and tools that have their own textual syntax for grammar definition. In this case,

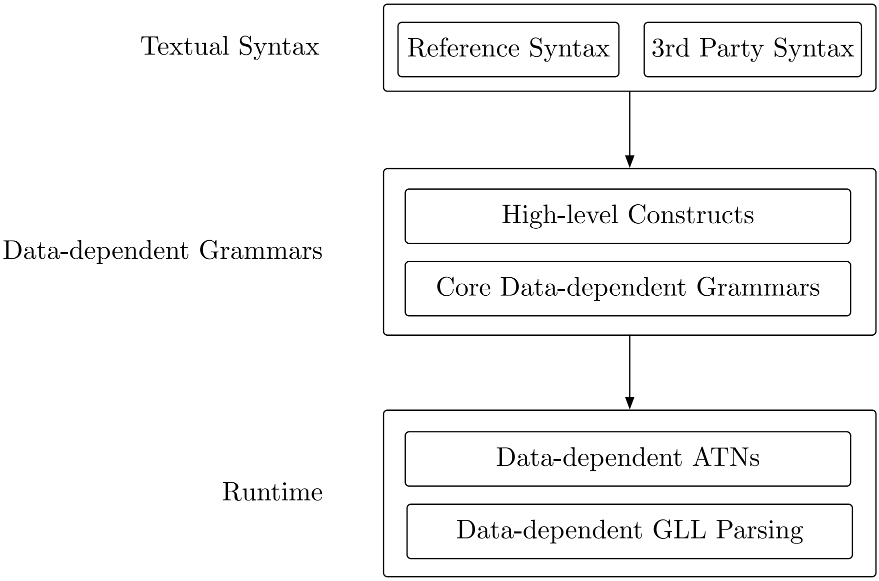


Figure 6.1: The architecture of Iguana.

Iguana's data-dependent API can be used to implement the textual syntax. In addition, custom disambiguation constructs can be added by desugaring to data-dependent grammars. Figure 6.1 shows the architecture of Iguana. In the following we discuss the architecture of Iguana and its components in more detail.

6.2.1 Runtime

Iguana is implemented on top of our version of GLL parsing. In Chapter 2 we proposed a modification to the Graph Structured Stack (GSS) of the original GLL algorithm, and showed that the new GSS makes GLL parsers significantly faster. To implement data-dependent features, we presented an extension of GLL that supports environment manipulation and return values (see Chapter 3).

Figure 6.2 shows a high-level overview of the Iguana runtime. As can be seen, the Iguana runtime has four high-level components: GSS, SPPF, Lookup Tables and Grammar Graph. We have discussed GSS and SPPF extensively in previous chapters. In the following we discuss Grammar Graph and Lookup Tables.

Iguana uses an interpretive version of GLL which operates on an in-memory representation of a grammar. We refer to this grammar representation as Grammar Graph, which is Iguana's implementation of data-dependent ATNs presented in Chapter 3. Interpreting the grammars at runtime eliminates the need for code generation and compilation cycles after every change to the grammar. This makes Iguana particularly suitable as a parsing framework for language workbenches, in which the user constantly changes the grammar.

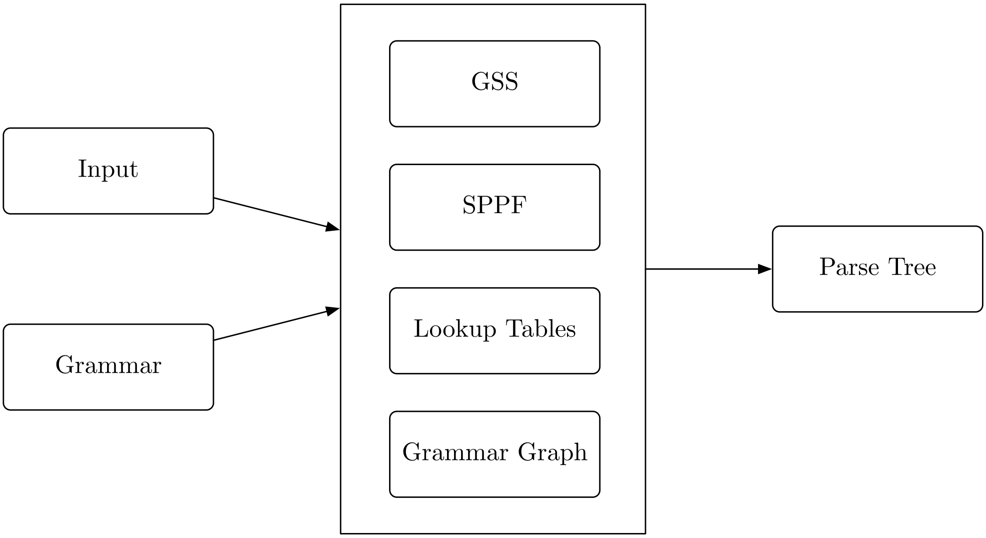


Figure 6.2: An overview of the Iguana Runtime.

For keeping the theoretical cubic complexity of GLL parsing, we need constant-time lookups, for example, for finding an existing GSS node with a particular label and input index. Implementing efficient lookup tables poses a significant engineering challenge, as it is not a trivial task to find the best trade-off between speed and memory usage. Our experience with various data structures showed that, for parsing real programming languages, hash tables distributed over the Grammar Graph (see Chapter 2) are the best implementation for Lookup Tables.

6.2.2 Data-dependent Grammars

This layer provides an API to construct an abstract representation of a data-dependent grammar. A data-dependent grammar can be directly defined using this API and later transformed to a data-dependent ATN to be interpreted. This layer also provides an API for transformation of data-dependent grammars. Our data-dependent grammars support all the features of the original data-dependent grammars [41], and some useful extensions such as return values. We call these features the *core* features. In addition, we provide a set of high-level syntactic constructs, e.g., for operator precedence and indentation rules, and transformations that desugar them to the core data-dependent grammars. Our parsing framework is extensible: the language engineer can add new high-level constructs, by using our API and providing the necessary desugaring to data-dependent grammars. Iguana's data-dependent features are discussed in detail in Chapter 3.

```
default layout terminal Layout
    : (WhiteSpace | Comment)*
    ;

terminal WhiteSpace
    : [ \t\r\n]
    ;

terminal Comment
    : TraditionalComment
    | EndOfLineComment
    ;

terminal CommentChar
    : [/]
    | [*]* ![/*]
    ;

terminal TraditionalComment
    : '/*' CommentChar* [*]+  '/'
    ;

terminal EndOfLineComment
    : '//' ![\r\n]* [\r\n]
    ;
```

Figure 6.3: Terminal and layout definitions in Iguana.

6.2.3 Textual Syntax

This layer provides a textual (concrete) syntax for defining grammars. Our textual syntax is faithful to the original syntax of data-dependent grammars and provides a grammar-centric, clean way of defining data-dependent grammars. Tool builders who wish to design their own syntax for grammar definition can integrate Iguana into their tool, as a parsing library for syntax definition, using our abstract representation of data-dependent grammars. In this section we discuss the textual syntax of Iguana using the grammar of Java 7, which is used for the performance evaluation in Section 6.3.

Iguana employs a single-phase parsing strategy [4], in which there is no separate lexing phase. Single-phase parsing effectively presents the parsing context to the lexing phase, and avoids the problems of a separate lexer in determining the type of tokens that have different meanings in different contexts. In the Iguana textual syntax, a terminal definition can be seen as a function that when called at a input index, returns the length of the matched substring. Terminals do not capture the structure and always return the longest possible match. We believe such semantics for terminals is the best option for parsing real programming languages because almost all lexical definitions in programming languages are greedy by default (longest match), and their internal structure is not needed.

In the Iguana textual syntax, terminals can be defined as single characters, corre-

```
IfStatement
    : 'if' '(' Expression ')' WSNoNL NL WSNoNL Statements 'fi'
    ;

layout terminal WSNoNL
    : [ \t]*
    ;

terminal NL
    : [\r\n]
    ;

default layout terminal WS
    : [ \t\n\r]*
    ;
```

Figure 6.4: An example of explicit layout insertion in Iguana.

sponding to the character-level scannerless parsing [73,94], or using regular expressions, corresponding to context-aware scanning [91]. Context-aware scanning brings the best of the both worlds: it avoids the problems of a separate scanning phase, while at the same time, brings considerable improvement in performance and memory usage compared to character-level grammars.

Currently, regular expressions in Iguana are implemented using Deterministic Finite Automata (DFA) [35], and therefore cannot be recursive. Moreover, advanced features such as lookahead and lookbehind, commonly found in backtracking regular expression engines, are not supported in Iguana.

Figure 6.3 shows some examples of terminal definitions in Iguana. The keyword `terminal` is used to specify a terminal definition. As can be seen, terminals can be defined in terms of other terminals, however, terminal definitions cannot be recursive. Another current limitation is that follow, precede restrictions and keyword exclusion [90] are only supported at a top-level terminal definition, i.e., a terminal definition which is not used by any other terminal definition.

Layout (whitespace and comment) can usually appear between any two tokens in the source code of programming languages. As Iguana uses a single-phase parsing strategy, it treats layout (whitespace and comment) like any other terminal or nonterminal. Since it is cumbersome to explicitly specify the layout in grammar rules, Iguana automatically rewrites the grammar to insert the layout definitions between symbols. In the Iguana textual syntax, a terminal or nonterminal can be marked as layout by using the `layout` keyword. If a layout definition is marked as `default`, it is used as the layout definition to be automatically inserted between symbols. In Figure 6.3 the terminal definition `Layout` is the default layout definition.

Other non-default layout definitions can be manually inserted between two symbols, overriding the default layout insertion. This is useful, for example, for cases where the user wants to use whitespace as a significant token in the grammar. As an example, consider an imaginary language in which the if-statement requires one and only one mandatory new line after its condition and before the statements. Figure 6.4 shows how the grammar of such a language can be defined in Iguana. In this grammar the layout definition `WSNoNL` defines whitespace characters excluding the new line characters. The presence of the `WSNoNL` in the `IfStatement` definition overrides the default layout, and allows the `NL` terminal to consume the only expected new line character.

Iguana also provides the special symbol _ which disallows the layout between two symbols. One use case for _ is defining nested comments. As nested comments are recursive, they cannot be defined as Iguana terminals, and should be defined as context-free rules, but at the same time, no layout should be inserted in such definitions.

Figure 6.5 (top) shows the ambiguous grammar of the expression part of the Java 7 grammar. In order to disambiguate this expression grammar, we use the following explicit disambiguation constructs:

- Lexical disambiguation constructs, such as follow and precede restrictions, and keyword exclusion.
- Operator precedence and associativity disambiguation constructs, such as `>`, `left`, `right`, and `nonassoc`. We discuss these constructs in more detail in Chapter 5.
- The exclusion disambiguation construct `!` for excluding a given alternative of a nonterminal. This disambiguation construct can be seen as an on-the-fly rewriting mechanism without explicitly introducing extra nonterminal into the grammar. The exclusion construct in Iguana is implemented using data-dependent grammars, by passing the excluded alternative as an argument and adding constraints for alternatives.

Figure 6.5 (bottom) shows the disambiguated version of the expression grammar of Java 7. The operator precedence ambiguities are resolved using the `>`, `left`, and `right` constructs. Besides the operator precedence disambiguation constructs, we also need the exclusion operator `!` and follow restrictions to fully disambiguate this grammar. An example of an ambiguous string that requires an exclusion operator is `(A) + 1`, which can be parsed as:

```
(A) (+ 1)  // Cast expression of a prefix expression
(A) + (1)  // Infix expression
```

According to the Java language specification, the second derivation is correct, as in the specification the cast rule is defined as:

```
'(' ReferenceType ')' UnaryExpressionNotPlusMinus
```

We can achieve the same behavior using an exclusion construct. To do this, we give a label to the `prefix` rule, using `%prefix`, and exclude it using the `!` operator as shown in Figure 6.5 (bottom). Note that the exclusion disambiguation construct is not safe [6] in general, i.e, it removes sentences from the language. In the case of the cast

```
Expression
    : Expression '.' Selector
    | MethodInvocation
    | Expression '[' Expression ']'
    | Expression ('++' | '--')
    | ('+' | '-' | '++' | '--' | '!' | '∼') Expression
    | "new" (ClassInstanceCreation | ArrayCreation)
    | '(' PrimitiveType ')' Expression
    | '(' ReferenceType ')' Expression
    | Expression ('*' | '/' | '%') Expression
    | Expression ('+' | '-') Expression
    | Expression ('<<'' | '>>' | '>>>') Expression
    | Expression ('<' | '>' | '<=' | '>=') Expression
    | Expression 'instanceof' Type
    | Expression ('==' | '!=') Expression
    | Expression '&' Expression
    | Expression '^' Expression
    | Expression '|' Expression
    | Expression '&&' Expression
    | Expression '||' Expression
    | Expression '?' Expression ':' Expression
    | Expression AssignmentOperator Expression
    | Primary
    ;
```

```
Expression
    :       Expression !instanceOf '.' Selector
    |       MethodInvocation
    |       Expression '[' Expression ']'
    |       Expression ('++' | '--')
    >       ('+' | '-' | '++' | '--' | '!' | '∼') Expression       %prefix
    |       "new" (ClassInstanceCreation | ArrayCreation)
    |       '(' PrimitiveType ')' Expression
    |       '(' ReferenceType ')' Expression !prefix
    > left  Expression ('*' | '/' | '%') Expression
    > left  Expression ('+' !>> '+' | '-' !>> '-') Expression
    > left  Expression ('<<' | '>>' | '>>>') Expression
    > left  Expression ('<' | '>' | '<=' | '>=') Expression        %comparison
    >       Expression 'instanceof' Type                           %instanceOf
    > left  Expression ('==' | '!=') Expression
    > left  Expression '&' Expression
    > left  Expression '^' Expression
    > left  Expression '|' Expression
    > left  Expression '&&' Expression
    > left  Expression '||' Expression
    > right Expression '?' Expression ':' Expression
    > right Expression !comparison AssignmentOperator Expression
    |       Primary
    ;
```

Figure 6.5: An excerpt of the Java 7 grammar written in the Iguana textual syntax, containing the ambiguous `Expression` definition (top), and the disambiguated version using the high-level disambiguation constructs in Iguana (bottom).

expression in Java, since we always have another derivation, we can safely use the `!` operator.

Another non-operator-precedence ambiguity in this grammar occurs in the sentence `1++ + 2`, which can be parsed as `((1++) + 2)` or `(1+(+ (+ 2)))`. The second parse is illegal and should be discarded, but since the cause of this ambiguity is the overlapping of the lexical tokens `+` and `++`, we need to explicitly specify a follow restriction for `'+'`, i.e., `'+' !>> '+'`, as shown in Figure 6.5 (bottom). This example shows an important property of our context-aware scanning: each terminal is considered in its context, and there is no global longest match rule for lexical definitions. Having a global longest match rule for lexical definitions would have the same problems as a separate lexer. Therefore, such ambiguities should be explicitly resolved using follow restrictions.

6.2.4 SPPF and Parse Trees

Iguana takes a grammar and a string as input and produces a parse tree as output. During the parsing GLL constructs a binarized SPPF [77], which contains all the derivations. Binarized SPPF is essential for keeping the cubic worst-case complexity of GLL parsing. After the parsing we convert the produced SPPF to a parse tree[2]. In this section we discuss Iguana parse trees in detail using some examples.

Figure 6.6 (left) shows a simplified excerpt of the Java 7 grammar, containing the method invocation syntax and a small subset of the expression grammar. Before parsing, this grammar goes through two transformations. First, EBNF constructs, such as $*$ and ?, are rewritten and replaced with (left-recursive) BNF definitions. Second, layout is inserted between symbols. We assume that the default layout definition `Layout` is defined for this grammar, and the terminal definitions `Identifier` and `Number` are available.

Figure 6.6 (right) shows the grammar after the EBNF to BNF and layout insertion transformations are performed[3]. The auxiliary nonterminals `Opt`, `Star` and `Plus` are introduced after the EBNF to BNF conversion. In addition, `Layout` is inserted between each two symbols in the body of the grammar rules.

The SPPF resulting from parsing the input `f(1, 2, 3)` with the grammar in Figure 6.6 (right) is shown in Figure 6.7 (top). In the visualizations in this section, nonterminal nodes have double border lines and are labeled with the name of the nonterminal. Terminal nodes have single border lines and are labeled with both the name of the terminal and the matched substring. Intermediate nodes have gray background, and are labeled with a grammar slot, which is a position (indicated by a dot) in the body of a grammar rule. Note that in Iguana, compared to the original GLL [78], we only add packed nodes when there is an ambiguity. Also, for a better visualization, the left and right extent indices of nodes are not shown.

[2] Our parse tree format is inspired by the parse tree formats used in the ATerm library [87] and Rascal [51].

[3] As can be seen, there is a `Plus` nonterminal in the rewritten grammar, but there is no + operator in the original grammar. The + nonterminal is introduced during the EBNF to BNF transformation, as we rewrite $A*$ to $A* ::= A+ \mid \epsilon$ to make the layout translation more uniform.

```
MethodInvocation
  : Identifier Arguments
  ;

Arguments
  : '(' (Exp (',' Exp)∗)? ')'
  ;

Exp
  : Exp '∗' Exp
  | Exp '+' Exp
  | Number
  ;
```

```
MethodInvocation
  : Identifier Layout Arguments
  ;

Arguments
  : '(' Layout Opt Layout ')'
  ;

// (Exp (',' Exp)∗)?
Opt
  : Exp Layout Star
  | // epsilon
  ;

// (',' Exp)∗
Star
  : Plus
  | // epsilon
  ;

// (',' Exp)+
Plus: Plus Layout ',' Layout Exp
  | ',' Layout Exp
  ;

Exp
  : Exp Layout '∗' Layout Exp
  | Exp Layout '+' Layout Exp
  | Number
  ;
```

Figure 6.6: The simplified `MethodInvocation` definition from the Java 7 grammar (left), and the same grammar after EBNF to BNF and layout insertion transformations (right).

After parsing an input, Iguana transforms the resulting SPPF to a parse tree. The parse tree corresponding to the SPPF in Figure 6.7 (top) is shown in Figure 6.7 (bottom). Iguana has an option to include or skip layout nodes in the produced parse tress. The parse tree shown in Figure 6.7 (bottom) has the layout nodes. As can be seen, in a parse tree, nonterminal and terminal nodes are the same as in the SPPF. However, there are no intermediate nodes in a parse tree. Moreover, for each EBNF construct, there is an explicit node in the parse tree, e.g., `(',' Exp)∗`. In Figure 6.7 (bottom), EBNF nodes are shown as ovals.

In an SPPF, ambiguities are represented using packed nodes [78]. An example of an ambiguous SPPF, corresponding to parsing the input `1 + 2 ∗ 3` with the `Exp` nonterminal of Figure 6.6, is shown in Figure 6.8 (top). Packed nodes are shown as circles in Figure 6.8 (top). The first and second packed nodes correspond to the grouping `((1 + 2) ∗ 3)` and `(1 + (2 ∗ 3))`, respectively.

When transform an SPPF to a parse tree, packed nodes are transformed into

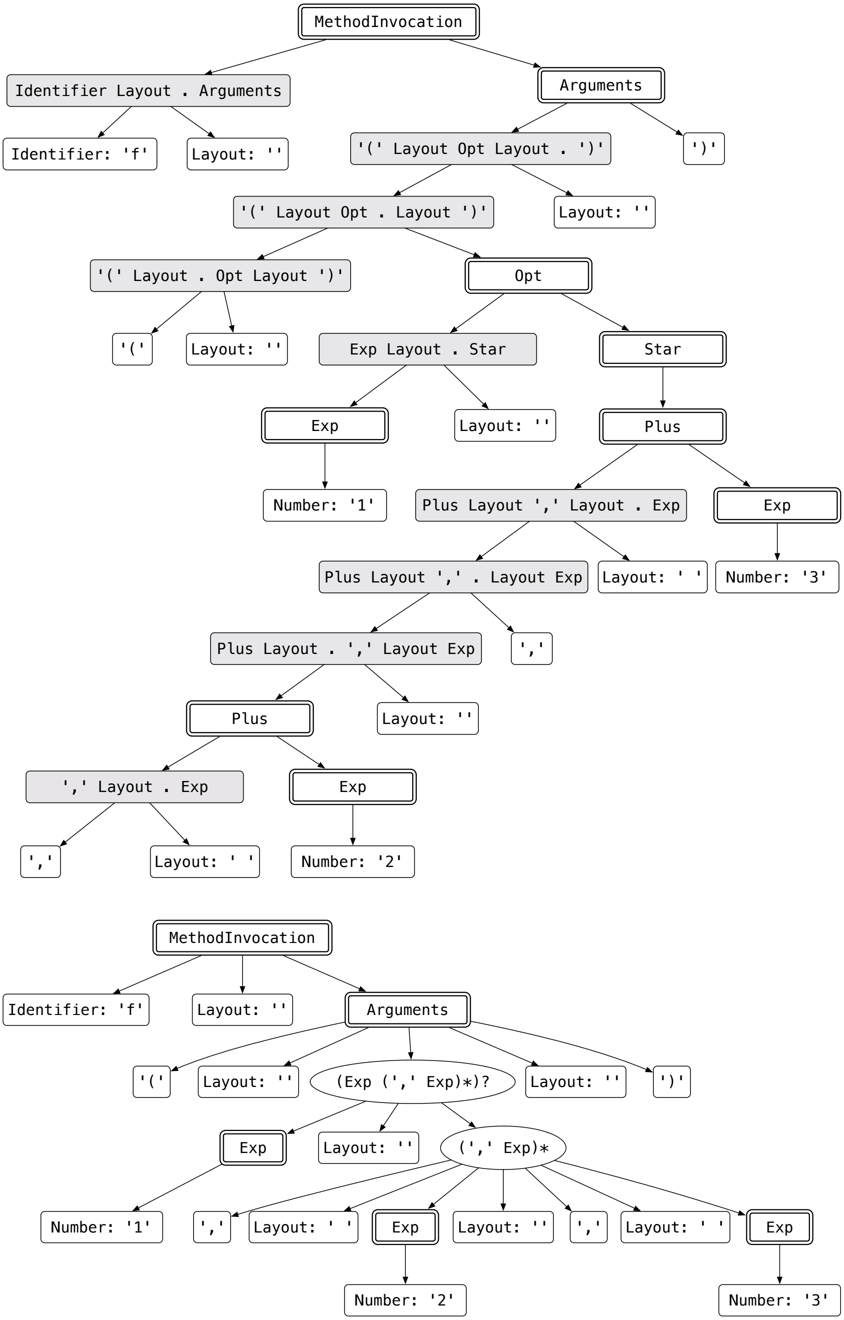


Figure 6.7: The SPPF (top) and parse tree (bottom) corresponding to parsing the input `f(1, 2, 3)` with the grammar of Figure 6.6.

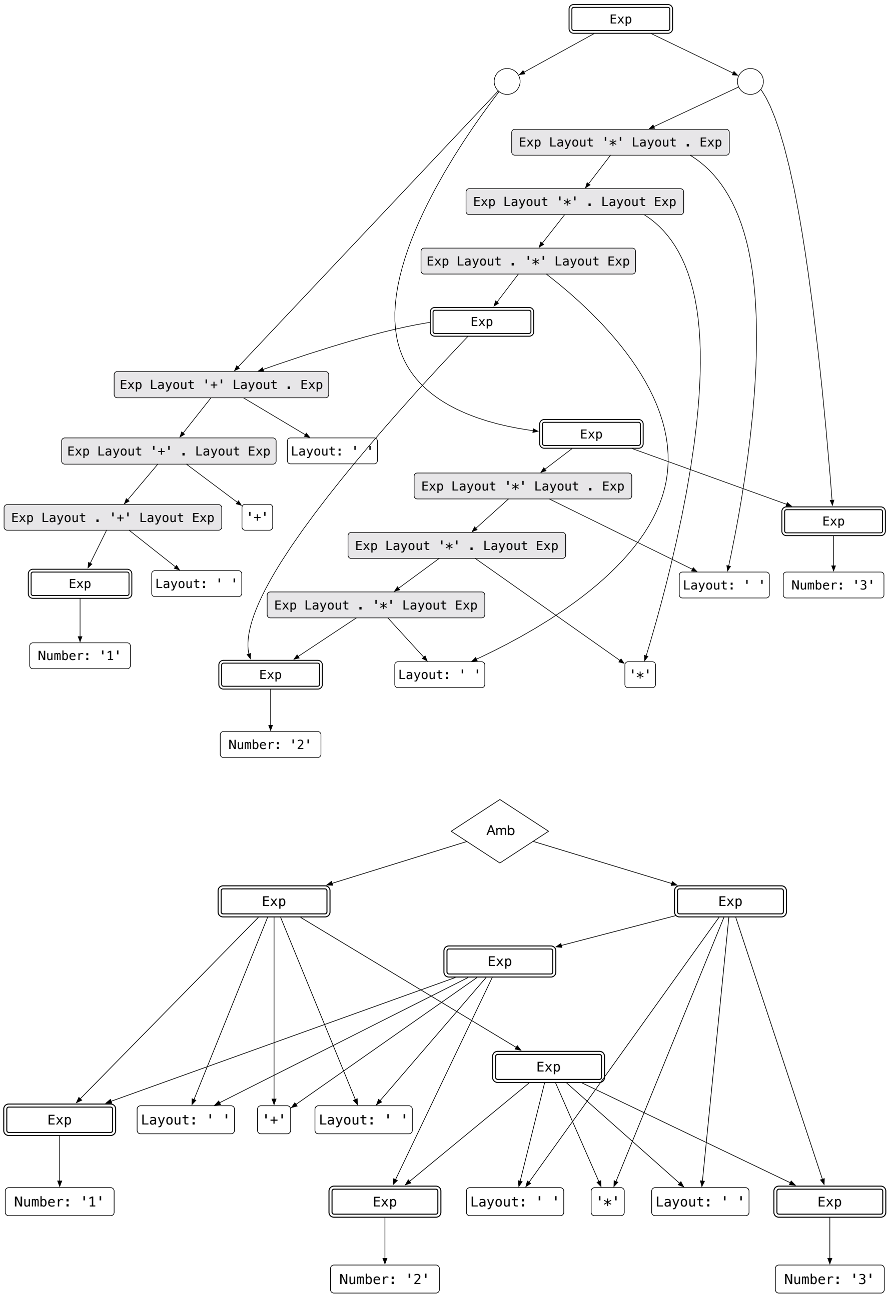

Exp
Exp Layout '∗' Layout . Exp
Exp Layout '∗' . Layout Exp
Exp Layout . '∗' Layout Exp
Exp
Exp Layout '+' Layout . Exp
Exp Layout '+' . Layout Exp
Layout: ' '
Exp
Exp Layout '∗' Layout . Exp
Exp Layout . '+' Layout Exp
'+'
Exp
Exp Layout '∗' . Layout Exp
Exp
Layout: ' '
Layout: ' '
Number: '3'
Exp Layout . '∗' Layout Exp
Number: '1'
Exp
Layout: ' '
'∗'
Number: '2'
Amb
Exp
Exp
Exp
Exp
Exp
Layout: ' '
'+'
Layout: ' '
Number: '1'
Exp
Layout: ' '
'∗'
Layout: ' '
Exp
Number: '2'
Number: '3'

Figure 6.8: The SPPF (top) and parse tree (bottom) corresponding to parsing the input `1 + 2 ∗ 3` with the `Exp` nonterminal of the grammar in Figure 6.6.

.

Table 6.1: Summary of the projects used for evaluating the performance of Iguana and ANTLR.

Project	Version	#Files	#Chars	SLOC	Size (on disk)
OpenJDK 7	jdk7u6-b08	14,161	837,651	3,628,651	168 MB
Elastic Search	2.4.6	4,438	377,147	752,995	40 MB
Guava	18.0	1,691	127,588	402,651	17 MB
RxJava	2.2.2	1,637	123,925	413,006	17 MB
JUnit	4.12	392	25,255	38,106	2 MB

ambiguity nodes. The parse tree corresponding to the SPPF of Figure 6.8 (top) is shown in Figure 6.8 (bottom). The ambiguity node in Figure 6.8 (bottom), which is shown as a diamond, has two children, each corresponding to a derivation.

6.3 Performance Evaluation

Iguana has been our research platform for the last years, and is under active development. Iguana is implemented in Java, and is available as an open source project[4]. In this section we evaluate the performance of Iguana by comparing it to ANTLR [69, 70], one of the most popular parsing tools currently available. ANTLR is widely used both as a library in various applications and also in language workbenches and tools such as Xtext[5]. In the following we refer to ANTLR version 4, but we simply call it ANTLR.

In Chapter 1 we enumerated challenges of comparing the performance of different parsing techniques. For this evaluation we tried to minimize the factors that make the comparison hard. First, both Iguana and ANTLR[6] are implemented in Java. Second, we used very similar grammars of Java, which have a natural expression part. However, we should note that ANTLR uses a separate lexing phase and also does not return all parse trees in form of a parse forest. This already gives ANTLR a big advantage in terms of performance. Nevertheless, we believe this section can provide the end user with enough insight about Iguana's performance when considering Iguana for parsing real programming languages.

6.3.1 Java Grammar and Source Files

For this evaluation we used the grammar of Java 7. Java is one of the most popular programming languages and is a good use case for evaluating parsing performance. Java has a C-style syntax which is very common, its grammar is large, and has a reasonably large and complex expression part.

For Iguana, we used a natural grammar of Java 7 from the Java Language Specification [32] as the basis, and replaced its term-factor-style expression grammar with a

[4] https://github.com/iguana-parser/

[5] https://www.eclipse.org/Xtext/

[6] ANTLR also has implementations in other languages such as C# and JavaScript.

natural, left-recursive version that is disambiguated using our declarative operator precedence constructs [5].

For ANTLR, we used a Java grammar from the ANTLR grammar repository on GitHub. Among several Java grammars available in the ANTLR grammar repository, we chose the one that was mentioned to have a better performance[7]. This grammar has a left-recursive, natural expression grammar and is close to the Java grammar we used for Iguana.

For the performance evaluation we used source code from several popular Java open source projects. Table 6.1 summarizes the projects we used in this evaluation.

6.3.2 Correctness

In order to compare the performance of Iguana against ANTLR, we first need to ascertain that both parsers can produce parse trees conforming to the syntax of Java. Note that establishing such correctness is not a straightforward task for the following two reasons. First, although the Iguana and ANTLR grammars of Java 7 that we use for the performance evaluation are very close to each other, they are not the same, and they are both different from the official Java specification grammar. Second, Iguana and ANTLR use different parse tree formats for the output parse trees. To circumvent these problems, we design the correctness analysis as follows:

- We select a well-known, widely-used Java library used for parsing Java source files and constructing Java ASTs. We use the AST format of the library as the canonical format, and the ASTs produced by the parser of the library as the reference ASTs.

- We convert the parse trees produced by both Iguana and ANTLR to the canonical AST format and compare them against the reference ASTs.

- We verify the correctness by running the comparison for a large corpus of Java source files.

For the correctness analysis, we have chosen the Java AST format of the Eclipse Java Development Tools (JDT)[8] as the canonical AST format. We have also used the Eclipse JDT parser, which is a hand-written recursive-descent parser, to construct the reference parse trees. The Eclipse JDT project provides Java language services for the Eclipse IDE, and it is also available as a standalone library (Eclipse JDT core) for analyzing Java source code.

Both ANTLR and Iguana provide visitors (the Visitor pattern [27]) for traversing the produced parse tree. We have written visitors to construct Eclipse JDT ASTs from both ANTLR and Iguana parse trees. Since both grammars are natural and close to the expected Java AST, the construction of Eclipse JDT ASTs was fairly straightforward, and did not require any significant structural transformation of the parse trees while traversing them. We note that constructing ASTs was slightly more

[7] https://github.com/antlr/grammars-v4/tree/master/java

[8] https://www.eclipse.org/jdt/overview.php

convenient when using ANTLR as it generates typed visitors when generating the parser for a grammar. Since Iguana is a grammar interpreter, the visitors are not typed, and we needed to match based on the string name of nonterminals. The source code for visitors can be found in our parser test suite project[9].

We compared the resulting Eclipse JDT ASTs from both ANTLR and Iguana against the ones produced by the Eclipse JDT parser. For comparing ASTs, we call the `ASTNode.subtreeMatch` method on the resulting AST objects. The `subtreeMatch` method recursively checks all the AST nodes. For comparing the ASTs we needed to consider the following issues:

- Java parsers cannot distinguish between a `FieldAccess` and `QualifiedName` without access to the type information. For example, `a.A` can either refer to the field `A` on an object referenced by `a`, or the qualified name `a.A` that represents type `A` from package `a`. Since this ambiguity cannot be resolved at parse time, most Java parsers accept one representation at parse time and postpone the actual check to a later phase when the necessary information is available. When comparing the ASTs from Iguana and ANTLR against the ones produced by the Eclipse JDT parser, we compare the string representation of `QualifiedName` and `FieldAccess` nodes if they are being compared against each other.

- The Eclipse JDT parser treats nested `InfixExpression` nodes with the same operator in a special way to prevent very deep expressions that may cause stack overflow while traversing the AST. For example, `1 + 2 + 3 + 4` is parsed as `{ left: 1, right: 2, op: +, extendedOperands: [3, 4])}`, instead of the expected `(((1 + 2) + 3) + 4)`. We convert "extended operands" of `InfixExpression` produced by the Eclipse JDT parser to the conventional nested expressions before comparing `InfixExpression` nodes.

- The Eclipse JDT parser parses the string `-2147483648` as a `NumberLiteral`, while according to the grammar of Java it should be parsed as a `PrefixExpression`. The reason why this number cannot be parsed as expected is that `2147483648` does not fit an integer in Java. We have a special check when comparing `PrefixExpression` and `NumberLiteral` nodes to cover this case.

The `subtreeMatch` method accepts an extra argument that allows customizing the matching. We created a custom matcher that resolves the above-mentioned issues while comparing ASTs. For all 22,319 Java files in the projects shown in Table 6.1, the ASTs produced by ANTLR and Iguana were the same, and were equal to the ones produced by the Eclipse JDT parser. Therefore, with high confidence we can say that both ANTLR and Iguana produce the same, correct parse trees.

[9] `https://github.com/iguana-parser/test-suite`

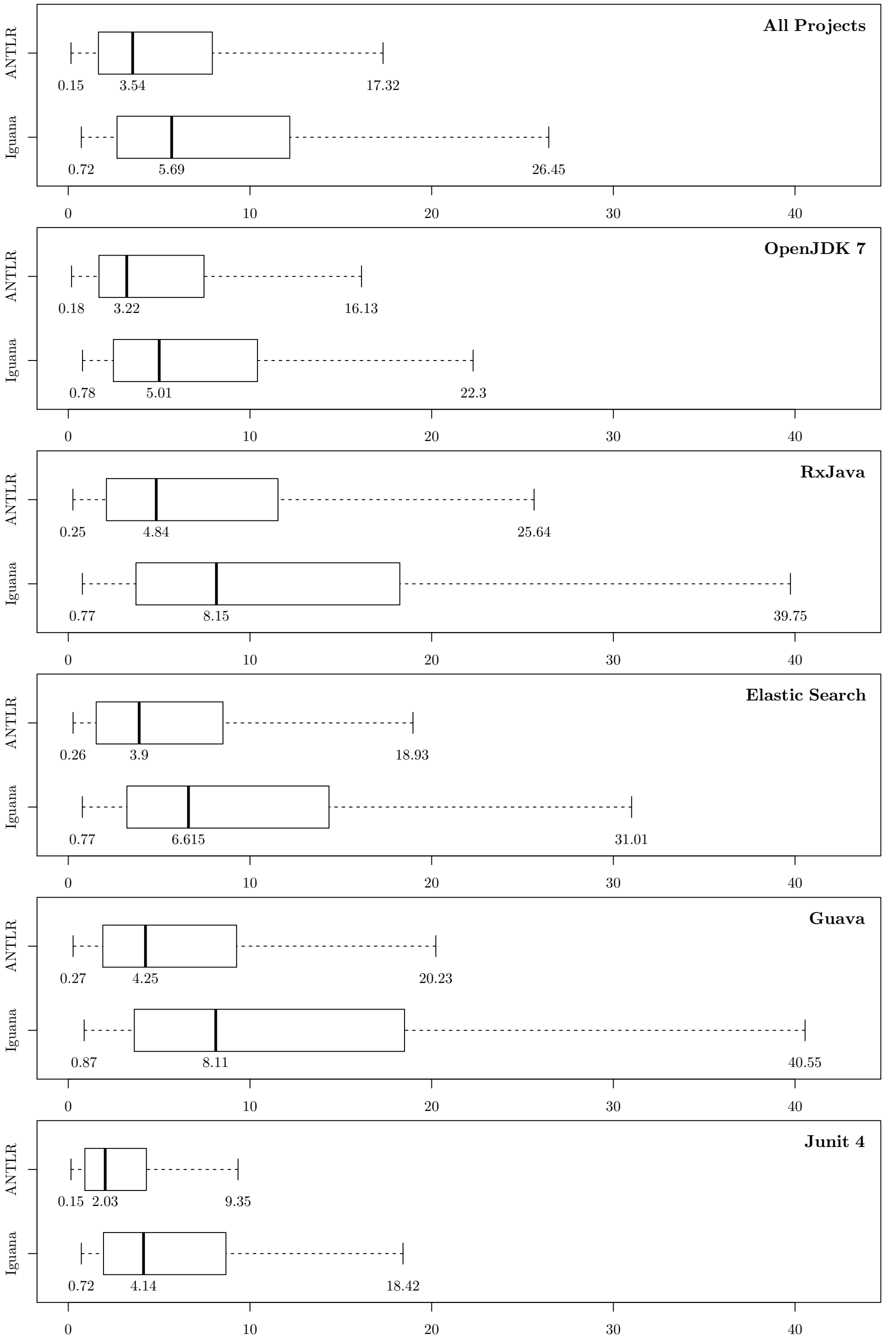


Figure 6.9: Running time of Iguana and ANTLR for the input files. The x axis shows the running time in milliseconds.

6.3.3 Performance

In this section we present the results of our performance comparison between ANTLR and Iguana. For this evaluation we used the latest master build[10] of Iguana, and the binary distribution of ANTLR version 4.7.1. For measuring the running time of parsers we use Java Measurement Harness (JMH)[11]. JMH is the state-of-the-art tool to perform benchmarking on the JVM platform, and prevents many pitfalls of running Java benchmarks.

The performance evaluation in this section was performed on a MacBook Pro with a quad-core Intel Core i7 2.6 GHz CPU and 16 GB of memory, running macOS X 10.13.6 and a 64-Bit Oracle HotSpot^{TM} JVM version 1.8.0_181. For running the benchmarks we did not explicitly specify a max heap size for the JVM, and therefore, the default 4GB for the max heap size was used. We discuss the memory usage in detail in the following section.

We used the single shot mode of JMH, which measures a single running time of the benchmark method. We ran the benchmarks for each input file 15 times, with 5 prior warmup iterations. The warmup iterations are essential for Just-In-Time (JIT) optimizations to take place. JMH reports the final running time for each file as the mean of 15 running iterations. We used two specific JVM arguments for running the benchmarks. First, we used a larger stack size, using the `-Xss4m` flag, to avoid stack overflow errors when producing deeply nested infix expressions that happen in few input files. Second, we used the new G1 garbage collector[12], using the `-XX:+UseG1GC` flag, which we found performing better in our case.

Figure 6.9 shows the performance results for all the files (All Projects in Figure 6.9) and each project separately. The reported time, for both Iguana and ANTLR, is the time from the moment we pass the input string to the parser until the moment the parse tree is constructed. We note that for ANTLR the reported time also includes the lexing time. The Eclipse JDT AST creation phase was only used to verify the correctness and is not included in the reported running time. Figure 6.9 excludes the outliers from the boxplots for better visualization. If we compare Iguana and ANTLR based on the median running time (5.69 ms vs 3.54 ms) and the maximum running time (26.45 ms vs 17.32 ms) for all the files, we can observe that these values are comparable.

Figure 6.10 shows the relative running time, the running time of Iguana divided by the running time of ANLTR, for each input file. As can be seen, if we consider the median running time for all the projects, Iguana is 69% slower than ANTLR. In some cases ANTLR is faster than Iguana by a factor of 4, but there are also cases where Iguana is faster than ANTLR.

We have inspected a number of the input files that showed extreme differences in performance between Iguana and ATNLR. Our observation is that Iguana is faster than ANTLR when a file contains many field access or method call chains. This

[10] From the commit hash `74a254c9` of the master branch of the Iguana repository, located at `https://github.com/iguana-parser/iguana`.

[11] `http://openjdk.java.net/projects/code-tools/jmh/`

[12] `https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/g1_gc.html`

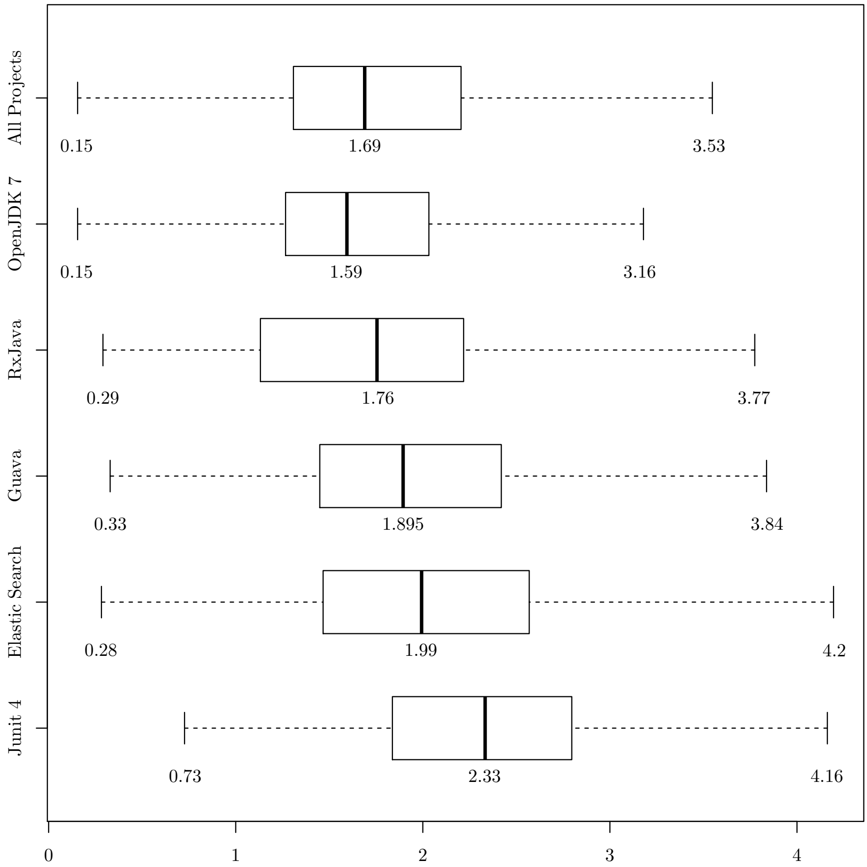


Figure 6.10: The relative running time of Iguana vs. ANTLR for each input file.

shows that the handling of expressions and the operator precedence disambiguation mechanism in Iguana are efficient. ANTLR is significantly faster than Iguana on files that are small (about 200 lines of code), and contains many string literals. We suspect that ANTLR's separate lexing phase and possibly a more efficient lexer implementation contributed to the big difference in these files.

The results of our performance evaluation in this section show that Iguana is practical for parsing real programming languages. Specifically, Iguana's performance is comparable to ANTLR, a mature parsing library that has been under development for many years. As expected, Iguana is slower than ANTLR because of using a single-phase parsing strategy and a more complicated machinery to construct a shared parse forest of possible ambiguities. However, if we consider the absolute running time values in Figure 6.9, we can observe that the difference is small for most cases. Moreover, we believe that the slight performance penalty for features such as single-phase parsing, explicit disambiguation and parse trees with ambiguity nodes is a reasonable trade-off.

6.3.4 Memory Usage

The Java Virtual Machine (JVM) provides automatic memory management via garbage collection. All new objects are allocated in the JVM Heap space, and are garbage collected when there are no more references pointing to them. The JVM Heap space can be controlled via the `-Xms` and `-Xmx` arguments, corresponding to the minimum and maximum heap size values[13], respectively. The JVM starts with the specified minimum heap size value and can grow dynamically to the maximum heap size value as needed. The JVM throws `OutOfMemoryError` if the maximum heap size is reached while allocating memory for a new object.

For comparing the memory usage of Iguana and ANTLR, we try to find the smallest value for the `-Xmx` parameter (max heap size) with which an input file can be successfully parsed. We parsed each input file, with both Iguana and ANTLR, starting with 1 MB value for the `-Xmx` parameter, and incrementing this value by 1 MB each time the parse failed due to `OutOfMemoryError` until we could successfully parse the input file.

Table 6.11 summarizes the memory usage (the smallest value for the `-Xmx` parameter) for all the input files for Iguana and ANTLR. For Iguana, 20,924 out of 22,319 files (93.75%) are successfully parsed with 9 MB of memory, and for ANTLR, 20,676 out of 22,319 files (92.64%) are successfully parsed with 5 MB of memory. Note that the reported value for the memory usage also includes the memory for the produced parse tree.

Our analysis shows that although larger files require more memory, there is no observable correlation between the size of a file and its memory consumption. The largest file in our input files, `EUC_TW_OLD.java`[14] (2.2 MB on disk), required 65 MB and 35 MB to parse with Iguana and ANTLR, respectively. This file contains very large expressions for concatenating strings. The maximum amount of memory needed to parse a file, however, was used for the file `BigClass.java`[15] which contains huge array initializers. The size of this file is only 647 KB on disk, but required 233 MB and 131 MB memory to parse with Iguana and ANTLR, respectively. We observed that in general Iguana requires more memory to parse a file than ANTLR. This is expected and can be explained as follows:

1. To deal with ambiguities Iguana needs to keep the SPPF in memory all the time. As, in the general case, it is not possible to impose any ordering on processing of descriptors in GLL, we need to keep all the SPPF nodes until all descriptors are processed. Constructing the SPPF and its associated lookup tables requires significant amount of memory. Moreover, GLL uses a binarized SPPF format

[13] The default minimum and maximum heap sizes are system dependent.

[14] `https://github.com/openjdk-mirror/jdk7u-jdk/blob/jdk7u6-b08/test/sun/nio/cs/EUC_TW_OLD.java`

[15] `https://github.com/openjdk-mirror/jdk7u-jdk/blob/jdk7u6-b08/test/java/lang/instrument/BigClass.java`

Size (MB)	Count	Percentage
9	20924	93.750
10	52	0.233
11	665	2.980
12	17	0.076
13	251	1.125
14	12	0.054
15	149	0.668
16	8	0.036
17	75	0.336
18	5	0.022
19	41	0.184
20	3	0.013
21	25	0.112
22	1	0.004
23	24	0.108
24	2	0.009
25	17	0.076
26	1	0.004
27	9	0.040
29	10	0.045
31	3	0.013
33	6	0.027
34	1	0.004
35	7	0.031
36	1	0.004
39	2	0.009
47	1	0.004
49	1	0.004
53	1	0.004
57	1	0.004
65	1	0.004
75	1	0.004
87	1	0.004
233	1	0.004

Size (MB)	Count	Percentage
5	20676	92.639
6	12	0.054
7	1337	5.990
8	8	0.036
9	176	0.789
10	2	0.009
11	72	0.323
13	16	0.072
15	8	0.036
16	1	0.004
17	5	0.022
21	3	0.013
23	1	0.004
35	1	0.004
131	1	0.004

Figure 6.11: Summary of memory usage (maximum heap size value) of all the input files for Iguana (left) and ANTLR (right).

with intermediate nodes, which also contributes to the extra memory usage of Iguana. In contrast, ANTLR returns at most one parse tree and does not maintain a shared parse forest while parsing, and therefore, has the memory advantage.

2. Iguana uses a single-phase parsing strategy. This means that the layout information (whitespace and comments) are also kept in the SPPF. Keeping the layout information in the SPPF contributes to the larger memory footprint of Iguana. In particular, in combination with our layout insertion strategy, these layout nodes significantly increase the number of the intermediate nodes in the resulting SPPF, see Figure 6.7 (top) as an example.

3. Iguana does not natively support EBNF, and converts EBNF definitions to BNF before parsing. The EBNF to BNF conversion introduces more intermediate nonterminals in the grammar, which leads to more SPPF nodes, see Figure 6.7 (top) as an example.

The results of this section show that the memory usage of Iguana is reasonable given that Iguana can produce all the ambiguities in the form of a parse forest and does not use a separate scanning phase before parsing. We expect to be able to reduce the runtime memory footprint by using a smarter way of handling layout in the SPPF. Scott and Johnstone have recently published an extended version of their GLL parsing algorithm that natively supports EBNF [79]. In the future, we plan to investigate how such native treatment of EBNF will interact with the data dependency support in Iguana, and how it will affect the memory consumption.

6.4 Conclusions

In this chapter we presented Iguana, our data-dependent parsing framework. Iguana has been our research platform in the last years, and has been used in evaluating our research results in the previous chapters. In this chapter we discussed Iguana's architecture and presented the performance results of Iguana compared to ANTLR. The results show that, compared to ANTLR, Iguana has slight performance and memory overhead. This is expected because of Iguana's more complicated machinery to support ambiguities. The results of this evaluation show that Iguana is practical for parsing real programming languages.

We discussed the textual syntax of Iguana using the Java 7 grammar as an example. In addition to the lexical and operator precedence disambiguation constructs, Iguana also supports other features that are based on data-dependent grammars. These features are useful in dealing with type definitions in C, and indentation-sensitive programming languages such as Haskell and Python [4].

We developed an IntelliJ plugin to facilitate the development of Iguana grammars. The plugin provides common features such as syntax highlighting, navigation, outline views, basic refactoring and some static grammar validation. As future work, we plan to integrate Iguana into the Rascal meta-programming language [51] as the default parsing library, and also work on the stability of features in Iguana.

Chapter 7

Practical, General Parser Combinators[1]

Summary. Parser combinators are a popular approach to parsing where context-free grammars are represented as executable code. However, conventional parser combinators do not support left recursion, and can have worst-case exponential runtime. These limitations hinder the expressivity and performance predictability of parser combinators when constructing parsers for programming languages.

In this chapter we present general parser combinators that support all context-free grammars and construct a parse forest in cubic time and space in the worst case, while behaving nearly linearly on grammars of real programming languages. Our general parser combinators are based on earlier work on memoized Continuation-Passing Style (CPS) recognizers. First, we extend this work to achieve recognition in cubic time. Second, we extend the resulting cubic CPS recognizers to parsers that construct a binarized Shared Packed Parse Forest (SPPF).

Our general parser combinators bring the best of both worlds: the flexibility and extensibility of conventional parser combinators, and the expressivity and performance guarantees of general parsing algorithms. We used the approach presented in this chapter as the basis for Meerkat, a general parser combinator library for Scala.

[1] This chapter was originally published as: A. Izmaylova, A. Afroozeh, and T. v. d. Storm. Practical, General Parser Combinators. In *Proceedings of the ACM SIGPLAN Symposium/Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 1–12. ACM, 2016

7.1 Introduction

Parsing is a well-researched topic in computer science. However, there is no “one size fits all” solution for all parsing problems. In particular, all solutions have to find a balance among trade-offs such as expressivity, performance, ease of use, and flexibility. Syntax of programming languages has traditionally been specified using context-free grammars. In parser generators, a grammar is written in a (E)BNF-like notation, which is transformed to parse tables or code. In parser combinators [36,37], on the other hand, a grammar is encoded using higher-order functions in a programming language, and thus is directly executable.

Parser combinators are higher-order functions used to define grammars in terms of constructs such as sequence and alternation. The seamless integration with the host programming language makes parser combinators flexible and extensible, compared to parser generators that use a fixed notation for syntax definition. The language developer can define custom combinators using the features of the host programming language. It is also possible to perform data-dependent tasks, such as parsing network protocols and indentation-sensitive languages, by allowing composition of functions that produce parsers based on the result of the previous parse. Monadic parser combinators [36,37] are often used for this.

Conventional parser combinators are recursive-descent like, and thus have an intuitive execution model, which makes them easy to debug. However, recursive-descent parsers fail to terminate in face of left recursion, and can have worst-case exponential runtime if implemented naively. The lack of support for left recursion is a major problem in expressing natural syntax of programming languages. Most notably, expression grammars, when written in their natural form, are left-recursive.

Grammars of most programming languages do not fit deterministic classes of context-free grammars, such as LR(k) or LL(k), and transforming a grammar to such forms is a tedious process. In addition, maintenance and evolution of deterministic grammars is difficult. There has been extensive research in general parsing algorithms [18,78,85], which accept the full class of context-free grammars and deliver all derivation trees in form of a parse forest. There exist worst-case cubic general parsers which are near linear on near-deterministic grammars. Therefore, it is practical to build parsers for programming languages using general parsing, especially in areas where more expressivity is needed, e.g., for developing domain-specific languages (DSLs) and source code analysis.

For decades, general parsing algorithms, more specifically Generalized LR (GLR), have been used in parser generators. Besides standalone GLR parser generators, such as SGLR (used in ASF+SDF Meta-Environment [89]) and Elkhound [61], the popular GNU Bison has also a GLR mode. However, general parsing has not become popular in the world of parser combinators. The main reason is the technical difficulty in realizing general parsers in a combinator style. The underlying machinery of traditional parsing algorithms such as GLR is not composable using the sequence and alternation operators: GLR parsers operate on LR automaton, and each LR state corresponds to multiple positions in grammar rules, and therefore, parsers cannot be directly defined using sequence and alternation.

Earley [18] and Generalized LL (GLL) parsing [78] are different from GLR in the sense that the parser directly operates on grammar rules, rather than an automaton. In particular, GLL parsing has a close relationship with the grammar, similar to recursive-descent parsing. Using Earley's algorithm or GLL, it is possible to define a grammar in a combinator style, and then interpret such a grammar. For example, parser combinators based on Earley parsing are presented in [72]. In Chapter 3, we provided an interpretive formulation of GLL parsing. Such formulations of context-free grammars provide a deep embedding, as the grammar is represented by an algebraic data type.

Deep embedding can benefit from under-the-hood transformations and optimizations. For example, it is possible to calculate first/follow sets and use this information for pruning the search space. However, such parser combinators still have the flavor of a parser generator. Moreover, an extension to such parser combinators may also require modification to the underlying parsing algorithm, for example, the modified Earley sets [41] and the modified GLL algorithm [4] to support data dependency. In contrast, parser combinators defined as shallow embedding enable directly executable parsers, as grammars are represented directly as functions. Although certain optimizations are difficult and require access to meta-syntax of the host language, shallow embedding is attractive, since it eases extension and modification through seamless integration with the host programming language.

In this chapter we present general parser combinators that combine the expressivity and performance guarantees of state-of-the-art general parsing algorithms with the flexibility and ease of use of conventional parser combinators. Our general parser combinators support the full class of context-free grammars and produce a parse forest in $O(n^3)$ time and space. One key distinction of our approach, compared to interpreter-based general parsing solutions, is that in our approach, like in conventional parser combinators, parsers are directly executable. We believe such a model is a more natural generalization of conventional parser combinators.

In this chapter we present general parser combinators defined as a direct embedding in a programming language. We base our general parser combinators on earlier work on memoized Continuation-Passing Style (CPS) recognizers by Johnson [43]. Johnson's approach is a functional formulation of recursive-descent parsing which also provides an elegant solution to the problem of left recursion.

More specifically, our contributions are:

- We modify Johnson's CPS recognizers [43] to obtain the worst-case cubic complexity (Section 7.2). We show that Johnson's original formulation may require unbounded polynomial time (Section 7.4).

- We extend cubic CPS recognizers to fully general parsers by constructing binarized SPPFs [78, 80] (Section 7.3). These parsers are cubic in time and space, which we prove in Section 7.5.

- We evaluate the performance of the resulting parsers using a highly ambiguous grammar and the grammar of Java 7 [32]. The results show worst-case cubic

```
val E = syn (  right (E ~ "^" ~ E)  & { case x~y => Pow(x,y) }
            |>       "-" ~ E       & { Neg(_) }
            |> left  (E ~ "*" ~ E   & { case x~y => Mul(x,y) }
            |         E ~ "/" ~ E   & { case x~y => Div(x,y) })
            |> left  (E ~ "+" ~ E   & { case x~y => Add(x,y) }
            |         E ~ "-" ~ E   & { case x~y => Sub(x,y) })
            |         "(" ~ E ~ ")"
            |         "[0-9]".r     ^ { s => Num(toInt(s))   }
            )
```

Figure 7.1: A natural expression grammar directly encoded in Scala using the Meerkat library.

runtime performance on the highly-ambiguous grammar and near linear runtime performance on the grammar of Java (Section 7.6).

Our general parser combinators are implemented as part of the Meerkat parser combinator library[2]. The Meerkat library provides combinators for lexical disambiguation, layout (whitespace and comment) insertion, EBNF, operator precedence, and execution of semantic actions. Figure 7.1 shows how an expression grammar can be written in a natural form using the Meerkat library. The grammar is disambiguated using declarative disambiguation combinators for operator precedence, such as `|>`, `left`, and `right`. In addition, semantic actions for AST generation are defined using the `&` and `^` combinators. More information on how to use Meerkat can be found in the GitHub repository. In this chapter, we only focus on the underlying parsing technique.

The rest of this chapter is organized as follows. Section 7.2 introduces the original Johnson's CPS recognizers, and our extension to the memoization strategy that makes them cubic. In Section 7.3 we extend the cubic CPS recognizers to parsers that construct binarized SPPFs in cubic time and space. We illustrate the unbounded polynomial behavior of the original Johnson's CPS recognizers in Section 7.4, and give the proof for the cubic bound for our CPS parsers in Section 7.5. In Section 7.6 we present the performance results of our CPS parsers using a highly ambiguous grammar and the grammar of Java. Section 7.7 discusses related work, and Section 7.8 concludes.

7.2 General Cubic CPS Recognizers

7.2.1 Basic Recursive-Descent Recognizers

In this section we introduce *basic* recursive-descent recognizers that use a simple backtracking strategy: the alternatives of a nonterminal are tried in order, and the next alternative is tried only if the current one fails. Figure 7.2 shows such a

[2] `https://github.com/meerkat-parser`

```
type Recognizer = Int => Result[Int]

def terminal(t: String): Recognizer =
  i => if (input.startsWith(t, i)) success(i + t.length) else failure

def epsilon: Recognizer = i => success(i)

def seq(rs: Recognizer*): Recognizer =
  rs.reduceLeft((cur, r) => (i => cur(i).flatMap(r)))

def rule(nt: String, alts: Recognizer*): Recognizer =
  alts.reduce((cur, alt) => (i => cur(i).orElse(alt(i)))
```

Figure 7.2: Combinator-style recognizers.

formulation[3]. We use basic Scala to explain the semantics of parser combinators as it is expressive enough to enable a concise, executable specification.

A basic recognizer is a function of type `Recognizer`, which is defined as a type alias to function type `Int => Result[Int]` (using the `type` keyword) with `Int` as a parameter type and `Result[Int]` as a return type. `Result[T]` is a generic type instantiated with `Int` to represent the result of a recognizer. In essence, a basic recognizer is a *partial* function: it takes an input position and either succeeds, returning the next input position, or fails. Partiality of the basic recognizers can be implemented using Scala's monadic `Option[T]`:

```
type Result[T] = Option[T]
def success[T](t: T): Result[T] = Some(t)
def failure[T](): Result[T] = None
```

`Result[T]` is defined as a type alias to `Option[T]`, and two functions, parameterized with type parameter `T`, can be used to compute success and failure. This way, success with the next position `i` is represented by the value constructor `Some`, which takes a value of type `T` and creates a value of type `Option[T]`, e.g., `Some(i)`, and failure by the value constructor `None`.

Basic recognizers can be composed with four combinators (higher-order functions): `terminal`, `epsilon`, `seq` and `rule` (Figure 7.2). The first two combinators construct basic recognizers for terminals and ϵ, respectively. For example, `terminal` returns a closure (defined using the `=>` notation) that takes an input position `i` and reports success with the next input position `i + t.length` if terminal `t` matches a substring of the input string starting from `i`, otherwise reports failure. For the sake of presentation, we assume that `input` is globally visible instead of being passed as an argument.

The combinator `seq` is used to encode sequential composition of multiple recognizers. The resulting recognizer chains the given recognizers as long as each of them produces a result, and if any of them fails, the entire sequence also fails. The asterisk ($*$) next

[3] The code snippets used in this chapter are available at: https://github.com/meerkat-parser/cps-parsers

to the Recognizer type permits a variable number of arguments, and therefore, rs refers to a sequence of recognizers. seq invokes the method reduceLeft on the sequence rs to reduce this sequence to a new recognizer. At each step, reduceLeft applies a binary operator, passed to reduceLeft as a closure, to the recognizers in the sequence. For example, reduceLeft called on a sequence of three elements a_1, a_2, a_3 with some binary operator f is semantically equivalent to $f(f(a_1, a_2), a_3)$. The binary operator in the definition of seq takes two recognizers, cur and r, and returns a new recognizer. This recognizer takes an input position i and first calls cur at i. Then, if cur succeeds at i, e.g., by returning a value Some(j) with the next input position j, the recognizer calls r at j returning the result, otherwise the recognizer fails returning None. The flatMap method defined in type Option[T] enables such composition of basic recognizers, systematically accounting for partiality.

The combinator rule is used to define a nonterminal with head nt and alternatives alts. To recognize a nonterminal at an input position using basic recognizers, the alternatives of the nonterminal are tried at the input position until one of them succeeds. rule invokes the method reduce on a sequence of recognizers, alts, to reduce alternatives to a new recognizer by applying an associative binary operator at each step. The binary operator takes two recognizers, cur and alt, and returns a recognizer that given an input position i, first calls cur at i, and if it succeeds, returns its result, otherwise, calls alt at i. This semantics of handling failure is provided by the orElse method defined in type Option[T]. The orElse method uses the call-by-name evaluation strategy in its argument so that if cur succeeds at i, the expression alt(i) is not evaluated.

Given the combinators of Figure 7.2, a recognizer for $A ::= a$ can be directly defined as follows:

```
val A = rule("A", terminal("a")) // A ::= a
```

where A is a variable to which the resulting recognizer is assigned. This formulation, however, can be problematic when recursive definitions are required. For example, consider the grammar $S ::= aSb \,|\, aS \,|\, s$ defined as follows:

```
val S = rule("S",
      seq(terminal("a"), S, terminal("b")), // S ::= a S b
      seq(terminal("a"), S),                //     | a S
      terminal("s"))                        //     | s
```

In a programming language with strict evaluation, this recursive definition is not well-defined: when the defining expression on the right-hand side of the assignment, which recursively uses variable S, is evaluated, S is unbound. One way to solve this problem is to use a fix-point combinator fix, defined as:

```
def fix[A,B](f: (A=>B)=>(A=>B)): A=>B = {
  lazy val p: A=>B = f(t => p(t))
  p
}
```

Here we use the definition of fix that can be used in languages with strict evaluation.

This is also reflected in the type signature. Using `fix`, the recognizer for S can be defined as:

```
val p = fix(S => rule("S",
      seq(terminal("a"), S, terminal("b")), // S ::= a S b
      seq(terminal("a"), S),                //     | a S
      terminal("s")))                       //     | s
```

The resulting recognizer is the fix point of the function passed to `fix` as a closure. This way, the recursive structure of S is encoded as an anonymous recursive function that is assigned to variable `p`. The use of `S` in the body of the closure replaces the recursive uses of the recognizer in the previous definition.

As illustrated, recognizers are directly constructed using combinators, resembling the grammar. In the next section, we generalize this framework to allow results other than `Option[Int]`. In particular, we reuse the definitions of Figure 7.2 *as is* to obtain continuation-passing style recognizers, and later parsers.

7.2.2 Full Backtracking Using Continuation-Passing Style

The first problem with basic recursive-descent recognizers is that backtracking is local to a nonterminal. As a result, the order of alternatives may influence the recognition of a sentence. For example, a recognizer for the grammar $A ::= a \,|\, ab$ reports failure when recognizing the input string `"ab"`, although it is apparent that the second alternative matches `"ab"`. The reason for failure is that the first alternative reports success, and the second alternative is never tried, while there is an unmatched `b` left. The second problem is that basic recognizers only return a single derivation of the input string, which depends on the order of alternatives. The grammar $S ::= aSb \,|\, aS \,|\, s$, for example, can derive `"aasb"` in two different ways, corresponding to the following leftmost derivations:

1. $S \Rightarrow aSb \Rightarrow aaSb \Rightarrow aasb$
2. $S \Rightarrow aS \Rightarrow aaSb \Rightarrow aasb$

However, basic recognizers can only deliver one. To support the full class of context-free grammars, basic recursive-descent recognizers require exhaustive search, and therefore, need full backtracking. In such a setting, a recognizer can potentially succeed multiple times at the same input position. To introduce full backtracking into basic recursive-descent recognizers, the recognition functions have to be adapted to produce multiple values.

One way to approach this is to use Continuation-Passing Style (CPS). In fact, to transform basic recognizers into CPS recognizers, it is sufficient to only redefine `Result[T]` and accompanying functions `success` and `failure`. Indeed, the combinators of Figure 7.2 can be used with any `Result[T]` that defines how to compose two functions via `flatMap` and how to combine two results via `orElse`, so that the details specific to `Result[T]`, such as partiality of basic recognizers, are systematically managed by `flatMap` and `orElse`.

```
type Result[T] <: MonadPlus[T, Result]
def success[T](t: T): Result[T]
def failure[T]: Result[T]

trait MonadPlus[T, M[_] <: MonadPlus[_, M]] {
    def map[U](f: T => U): M[U]
    def flatMap[U](f: T => M[U]): M[U]
    def orElse(r: => M[T]): M[T]
}
```

Figure 7.3: Monadic `Result[T]`.

The `MonadPlus` trait in Figure 7.3 specifies a monadic interface: `map`, `flatMap` and `orElse`. The type constraint, expressed using `<:`, requires `Result[T]` to define the methods of the interface. Note that `=>`, used before the parameter type in `orElse`, indicates call-by-name evaluation in its argument. This can also be achieved by explicitly constructing a closure. In addition, `success` defines how to lift a value to the one of type `Result[T]` (basic computation) while `failure` defines *zero* (computation with no value).

A CPS recognizer is a function of the same type as in Figure 7.2, but with `Result[T]` as in Figure 7.4, i.e., defined as the continuation monad [97]. In Figure 7.4, `K[T]` represents a continuation type defined as a type alias to `T => Unit`[4]. Now, `Result[T]` is a function type, a subtype of `K[T] => Unit`, which also defines the methods of the `MonadPlus` interface (we discuss them later).

The helper method `result` (on the bottom of Figure 7.4) is used to define values of type `Result[T]` using ordinary functions of type `K[T] => Unit`. In Scala, a type can extend a function type, and a function is an object that has an `apply` method, e.g., if `f` is of type `Int => Int`, `f(0)` is equivalent to `f.apply(0)`. Given a function `f` of type `K[T] => Unit`, the `result` method creates an instance of `Result[T]`, say `g`, such that the result of `g(k)` is equal to the result of `f(k)`.

A CPS recognizer takes an input position and returns a function of type `Result[Int]`. The returned function takes a continuation of type `K[Int]` and returns `Unit`. A continuation is a function, now additionally passed to recognizers, that represents the "rest" of the recognition process. Instead of directly returning a value, a recognizer "returns" success by calling its continuation with the next input position, as in `success`, and fails by not calling its continuation, as in `failure`. For example, given a CPS recognizer for a terminal `val f: Recognizer = terminal("a")` and an initial continuation `val k0: K[Int] = i => println("success: " + i)`, the result of evaluating `f(0)(k0)` is either `"success: 1"`, printed to the console if the input string starts with `"a"`, where `1` is the next input position, or nothing otherwise.

Similar to basic recognizers, CPS recognizers can be composed using the combinators of Figure 7.2, but now defined in terms of `flatMap` and `orElse` of Figure 7.4. In the definition of `seq`, given two CPS recognizers, `cur` and `r`, the result of their composition

[4] `Unit` is equivalent to type `void` in other programming languages, e.g., Java.

```
type K[T] = T => Unit // Continuation type

trait Result[T] extends (K[T] => Unit) with MonadPlus[T, Result] {

    def map[U](f: T => U): Result[U] =
      result(k => this(t => k(f(t))))

    def flatMap[U](f: T => Result[U]): Result[U] =
      result(k => this(t => f(t)(k)))

    def orElse(r: => Result[T]): Result[T] = {
      lazy val v = r
      return result(k => { this(k); v(k) })
    }
}

def success[T](t: T): Result[T] = result(k => k(t))

def failure[T](): Result[T] = result(k => {/*do nothing*/})

def result[T](f: K[T] => Unit): Result[T] =
  new Result[T] {
    def apply(k: K[T]) = f(k)
  }
```

Figure 7.4: `Result[T]` for CPS recognizers.

using `flatMap` is a CPS recognizer that given an input position `i`, first calls `cur` at `i`, as before, but now returns a function of type `Result[Int]`. This function, given a continuation `k`, first creates a new continuation `t => r(t)(k)` and then passes this continuation to the result of `cur(i)`, so that the second recognizer, `r`, is called via this continuation when the recognizer `cur` succeeds at `i` with a new input position. In other words, `seq` now constructs a continuation-passing chain of function calls, one for each given recognizer.

In the definition of `rule`, given two CPS recognizers, `cur` and `alt`, a new CPS recognizer, returned by the binary operator, sequentially calls `cur` and `alt` at an input position `i` and combines the results of `cur(i)` and `alt(i)` using `orElse`, so that when the new CPS recognizer is called at `i` with a continuation, say `k`, `k` is passed to both results, i.e., `cur(i)(k)` and `alt(i)(k)`. This way, `rule` tries all the alternatives. Note that in the definition of `orElse`, the use of variable `v` with keyword `lazy` ensures that the argument to `orElse` is (lazily) evaluated only once.

CPS recognizers support full backtracking as the `rule` combinator always tries all of its arguments. The runtime behavior of such recognizers is exponential in the worst case. Furthermore, these recognizers will fail to terminate in face of left-recursive rules. Since Norvig's work on memoization in top-down parsing [66], it is known that memoizing recognizers brings down the exponential runtime performance to polynomial. However, this type of memoization does not solve the problem of left

```
def memo[T](f: Int => Result[T]): Int => Result[T] = {
  val table: Map[Int, Result[T]] = HashMap.empty
  return i => table.getOrElseUpdate(i, memo_result(f(i)))
}

def memo_result[T](res: => Result[T]): Result[T] = {
  val Rs: MutableList[T]    = MutableList.empty
  val Ks: MutableList[K[T]] = MutableList.empty
  return result(k =>
     if (Ks.isEmpty) { // Called for the first time
       Ks += k
       val k_i: K[T] = t => if (!Rs.contains(t)) {
                              Rs += t
                              for (kt <- Ks) kt(t)
                            }
       res(k_i)
     } else {          // Has been called before
       Ks += k
       for (t <- Rs) k(t)
     })
}
```

Figure 7.5: Memo functions for CPS recognizers.

recursion. In the next section we introduce Johnson's memoized CPS recognizers [43] that solve the problem of left recursion.

7.2.3 Support for Left Recursion

Neither the basic recognizers of Section 7.2.1, nor the CPS recognizers of Section 7.2.2 support left-recursive rules. Consider the following left-recursive recognizer:

```
val A = fix(A => rule("A",
   seq(A, terminal("a")), terminal("a"))) // A ::= A a | a
```

The call to `A` at input position `0` leads to unbounded number of recursive calls at the same input position, as the recursive calls do not change the function's state and never reach a base case.

The memo functions of Figure 7.5 turn an arbitrary CPS recognizer into a memoized CPS recognizer. The `memo` function, when applied to a recognizer, returns a new recognizer that consults the memo table each time it is called at an input position `i`. If the memoized recognizer has not been yet called at `i`, the result of calling the original, unmemoized recognizer, `f`, at `i` is memoized, `memo_result(f(i))`, and returned after updating the memo table. Due to the call-by-name nature of `memo_result` (note `=>` before the parameter type), `f(i)` is not evaluated at this moment. This ensures that `f` is called at `i` at most once. If the memoized recognizer has been called at `i` before, its result is taken from the table. Note that the `getOrElseUpdate` method uses

```
def rule(nt: String, alts: Recognizer*): Recognizer =
  memo(alts.reduce((cur, alt) => i => cur(i).orElse(alt(i))))
```

Figure 7.6: Memoizing CPS recognizers.

the call-by-name evaluation strategy in its second argument, so that it is not evaluated if the key is found.

A memoized result, returned when `memo_result` is applied to the result of calling an unmemoized recognizer `f` at an input position `i`, has access to two lists: `Rs` and `Ks` (lines 7–8). The result list `Rs` stores all input positions produced by the unmemoized recognizer when it succeeds at `i`, and the continuation list `Ks` stores all continuations passed to the memoized recognizer when it is called at `i`. If the memoized result is called for the first time (`Ks.isEmpty` in line 10), the current continuation `k` is added to `Ks`, and the original, unmemoized result, `res`, is called with a new continuation `k_i` (defined in lines 12–15). Note that `f(i)` is evaluated at this moment (line 16), and therefore, `f` can be called at `i` at most once. Also, `k_i` is created only upon the first call to the memoized recognizer at `i`. Each time the unmemoized recognizer succeeds at `i` with an input position, `k_i` checks whether this input position has been seen before and if not (`!Rs.contains(t)`), first records it in `Rs`, and then, runs all the continuations recorded so far in `Ks` at this input position (for-loop, line 14). On the other hand, if the memoized result has been called before (else-branch, lines 17–20), the current continuation `k` is added to `Ks` and is called for each input position recorded in `Rs`.

To add memoization to CPS recognizers, the `rule` combinator needs to be re-defined as in Figure 7.6. Now, when a memoized left-recursive CPS recognizer is called at an input position `i`, its termination is guaranteed as the respective unmemoized recognizer (`f` in the body of `memo`) will never be called at `i` more than once. At the same time, the part of the execution path, which led to a left-recursive call and can produce new input positions for the left-recursive recognizer at `i`, is effectively recorded as continuations. A continuation is recorded for the left-recursive call, and, in case of indirect left recursion, for each call to a memoized recognizer at `i` that indirectly led to the left-recursive call. Each continuation captures the next step in the alternative after the current call returns, and a continuation defined in lines 12–15 is called at the end of each alternative. These continuations will be run (re-trying the terminated paths) for any input position produced by the left-recursive recognizer at `i`, and as long as new input positions are produced.

Intuitively, memoizing CPS recognizers does not reduce the number of execution paths as all the continuations passed to a memoized CPS recognizer at an input position will be recorded in the continuation list, and each of the recorded continuations will be run for each result produced by the recognizer at this input position. This follows from the definition of `memo_result`. In the if-branch, all recorded continuations are invoked on a newly produced result. In the else-branch, all existing results are input to a new continuation.

```
def memo_k[T](k: T => Unit): T => Unit = {
  val s: Set[T] = HashSet.empty
  return t => if(!s.contains(t)) { s += t; k(t) }
}
```

Figure 7.7: Memoization on continuations.

7.2.4 Memoization on Continuations

In Section 7.4 we show that the execution of memoized CPS recognizers of Figures 7.2, 7.4 and 7.6 can require $O(n^{m+1})$ operations, where m is the length of the longest rule in the grammar. The reason for such unbounded polynomial behavior is that the same continuation can be called multiple times at the same input position. As illustrated in Section 7.4.3, this happens when the same continuation, say κ, is added to continuation lists that are associated with calls made to the same recognizer at different input positions, say $i_1, \ldots, i_n$. If those calls produce the same input position, say j, κ will be called multiple times at j. These duplicate calls further result in adding the same continuations to the same continuation lists multiple times.

To reduce the worst-case complexity to cubic, the duplicate calls to continuations need to be eliminated. To achieve this, we extend the memoization strategy by adding memoization on continuations, as in Figure 7.7, and by re-defining `Result[T]` as in Figure 7.8. A memoized continuation consults the set of already passed arguments (Figure 7.7), input positions in case of recognizers, and runs the unmemoized continuation, `k`, only when it has not been called before with the current input position. The memoization on continuations prevents the same execution path to be explored more than once, where an execution path is identified by a grammar position, the input position of the parent nonterminal and the current input position. Note that continuations `k_i` in Figure 7.5, lines 12–15, have been already defined with the memoization semantics.

In Section 7.3 we explain how CPS recognizers can be extended to parsers that construct binarized SPPFs. In Section 7.5 we show that memoization on continuations is also sufficient to keep the cubic bound for such CPS parsers.

7.2.5 Trampoline

In our parser combinators, when one continuation calls another continuation, the number of calls in the call stack may exceed the default size of the stack. To avoid stack overflow, we can turn the calls into a trampoline-style loop. Using a trampoline, calls are handled in a loop over a custom stack data structure, ensuring that the stack does not grow too large. To implement a trampoline, we create and pass around an object of type `Trampoline`. This object maintains a custom stack of values that represent calls and runs a loop over this stack. In the definition of `orElse` and `memo_result`, instead of calling a continuation or a function of type `Result[T]`, a value representing the call is pushed on top of the stack in the trampoline object. When the main loop runs, a

```
trait Result[T] extends (K[T] => Unit) with MonadPlus[T, Result] {

    def map[U](f: T => U) =
      result(k => this(memo_k(t => k(f(t)))))

    def flatMap[U](f: T => Result[U]) =
      result(k => this(memo_k(t => f(t)(k))))

    def orElse(r: => Result[T]) = {
      lazy val v = r
      return result(k => { this(k); v(k) })
    }
}
```

Figure 7.8: `Result[T]` extended with memoization on continuations.

value is popped from the stack, and the actual call represented by this value is made. Parsing terminates when there are no elements left in the stack.

7.3 SPPF Construction for Cubic CPS Parsers

7.3.1 Binarized SPPF

General parsing algorithms explore all derivations of a sentence. To deal with potentially exponential number of parse trees, common subtrees are shared to form a Shared Packed Parse Forest (SPPF), introduced by Tomita [85]. For example, the two parse trees, resulting from parsing `"aasb"` using the grammar $S ::= aSb \mid aS \mid s$, and their corresponding SPPF are shown in Figure 7.9.

In an SPPF, there are two types of nodes: symbol nodes and packed nodes. Symbol nodes are of the form (x, i, j), where x is the name of a nonterminal, terminal or epsilon, and i and j are the left and right extents, indicating the start and end positions in the input string recognized by x. Packed nodes, shown by small circles in Figure 7.9, represent a derivation. When there is ambiguity, e.g., under the root node in Figure 7.9 (c), multiple packed nodes are present, each identifying a derivation[5].

Johnson [42] showed that any parsing algorithm that produces a Tomita-style SPPF has $O(n^{m+1})$ worst-case runtime, where m is the length of the longest rule in the grammar. Therefore, in order to guarantee $O(n^3)$ worst-case runtime for general parsing, which is common in general recognizers, rules need to be of length at most two. Although a grammar can be transformed to a grammar with rules of length at most two, this transformation leads to a large grammar, with many extra nonterminals. This affects the maintainability of the grammar and the parsing performance [80].

To enable general parsing in $O(n^3)$ without transforming the grammar, Scott and Johnstone [80] introduced binarized SPPFs, which have additional *intermediate* nodes.

[5] In Figure 7.9, we do not show packed nodes when there is no ambiguity.

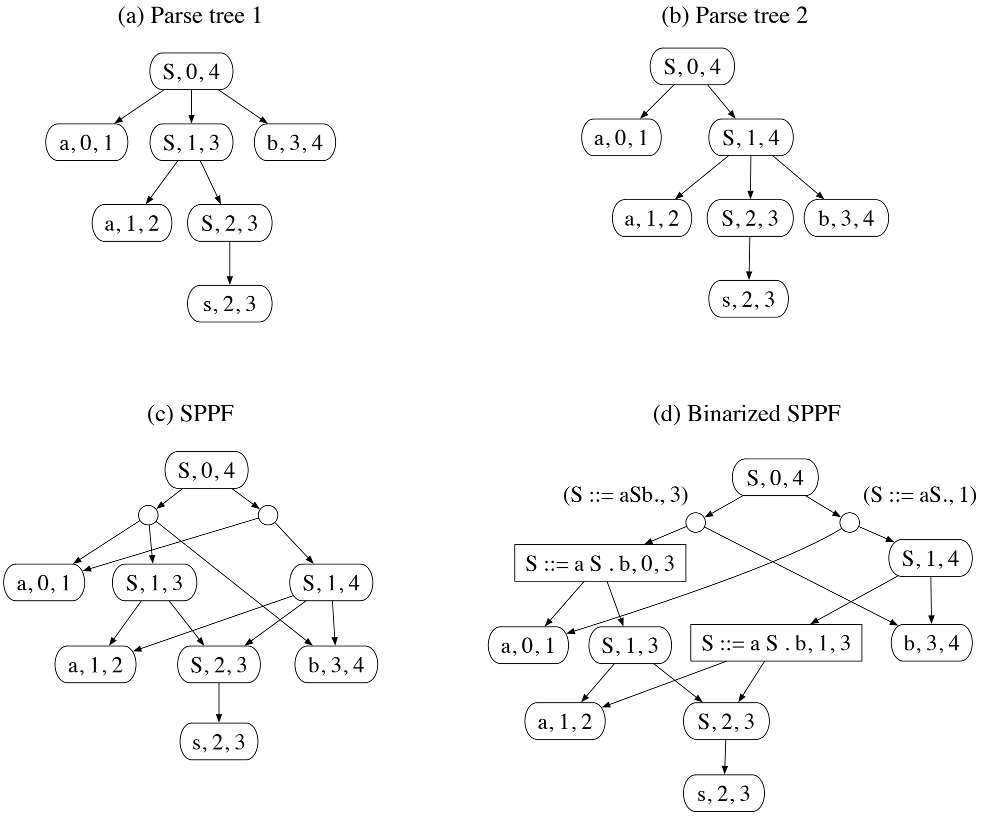


Figure 7.9: Two parse trees (a) and (b), the corresponding SPPF (c), and the binarized SPPF version (d).

Intermediate nodes are of the form (L, i, j), where L is a grammar position, and i and j are the left and right extents. Grammar positions for intermediate nodes are of the form $A ::= \alpha \cdot \beta$ where $|\alpha| \geq 2$. The binarized version of the SPPF is shown in Figure 7.9 (d). Similar to nonterminal nodes, intermediate nodes can be ambiguous. In this case, they have more than one packed node as children.

Packed nodes in a binarized SPPF are of the form (L, k), where L is a grammar position, and k, *pivot*, is an input position, which is equal to the left extent of the packed node's right child. For packed nodes under a nonterminal A, L is of the form $A ::= \alpha\cdot$, where α is an alternative of A. For example, in the binarized SPPF of Figure 7.9 (d), the left and right packed nodes under the root node have labels $(S ::= aSb\cdot, 3)$ and $(S ::= aS\cdot, 1)$, respectively. For packed nodes under an intermediate nodes, L is the same as the grammar position of the intermediate node. Packed nodes can have at most two children, which are non-packed nodes.

```
type Parser = Int => Result[NonPackedNode]

def terminal(t: String): Parser =
  i => if (input.startsWith(t, i))
         success(sppf.getTerminalNode(t, i, i + t.length))
       else
         failure

def epsilon: Parser = i => success(sppf.getEpsilonNode(i))

def seq(ps: Parser*): Parser = ps.reduceLeft(seq2)

def rule(nt: String, alts: Parser*): Parser =
  memo(alts.map(rule1(nt, _)).reduce((cur, p) => (i => cur(i).orElse(p(i)))))

private def seq2(p1: Parser, p2: Parser): Parser =
  fix(q => (i => p1(i).flatMap(
    t1 => p2(t1.rExtent).map(
      t2 => sppf.getIntermediateNode(q, t1, t2)))))

private def rule1(nt: String, p: Parser): Parser =
  fix(q => (i => p(i).map(t => sppf.getNonterminalNode(nt, q, t))))
```

Figure 7.10: Extension of CPS recognizers to parsers that construct binarized SPPF.

7.3.2 SPPF Construction

To extend CPS recognizers to parsers that produce binarized SPPFs, we redefine `terminal`, `epsilon`, `seq`, and `rule` as in Figure 7.10. Now, these combinators build parsers of type `Parser`, which is a function that takes an input position and returns a non-packed node, a symbol or intermediate node, as `Result[NonPackedNode]`.

The SPPF creation is delegated to an instance of `SPPFLookup` (`sppf`), which we assume is globally visible. The `SPPFLookup` interface provides methods that ensure sharing SPPF nodes. Each method of `SPPFLookup` first searches for an existing SPPF node with the given arguments and either returns an existing node, or creates a new one. The `getNonterminalNode` and `getIntermediateNode` methods take two non-packed nodes as arguments, which will be attached to the returned node via a packed node.

Each combinator returns a parser that is responsible for creation of a specific type of an SPPF node. A terminal node is created via `getTerminalNode` which takes the name of the terminal, the input position at which the function is called, and the next input position corresponding to the end of the matched terminal. Epsilon (`epsilon`) nodes have the same left and right extents, both being equal to the input position at which the parser is called.

The `seq2` combinator constructs a parser that creates an intermediate node based on the results of its two operands, `p1` and `p2`. An intermediate node is created via the `getIntermediateNode` method, which takes a label and two non-packed nodes, `t1` and `t2`, returned by the parsers `p1` and `p2`, respectively. Note that `seq2` uses its resulting parser, `q`, as the label of the intermediate node, and, as this definition is recursive, the fix point combinator is used.

Finally, the `rule` combinator first iterates over a sequence of parsers (by calling `map` on `alts`) to create parsers (by applying `rule1` to each parser of the sequence) that are responsible for creating nonterminal nodes. Then, `rule` reduces the resulting sequence of parsers. At the end of each alternative, a nonterminal node labeled `nt` is created. This is done by calling `getNonterminalNode` with the name of the nonterminal, the label of its packed node, and the non-packed node produced by `p` at `i`. Note that `rule1` uses its resulting parser, `q`, as the label of the packed node, and since this definition is recursive, the fix point combinator is used.

Sharing non-packed nodes relies on identifying non-packed nodes by their label and left and right extents. The label of a nonterminal or terminal node is a string, while the label of an intermediate node corresponds to a grammar position. In a parser generator setting, the labels of intermediate nodes can be determined by processing the grammar. In a parser combinator setting, on the other hand, no such preprocessing step exists, and labels corresponding to grammar positions should be dynamically determined. We use the identity of the parser object resulting from `seq2` as the label of the corresponding intermediate node (variable `q`) to effectively encode a grammar position. For example, for $S := aSb \,|\, aS \,|\, s$, which is defined as

```
val S = fix(S => rule("S", seq(terminal("a"), S, terminal("b")),
                           seq(terminal("a"), S),
                           terminal("s")))
```

the parsers resulting from applying `seq2` to the first two symbols, `terminal("a")` and `S`, in the first and second alternatives represent the unique grammar positions $S ::= aS \cdot b$ and $S ::= aS\cdot$, respectively.

7.3.3 Semantic Actions and Generation of ASTs

Binarized SPPFs are part of the internal machinery of a general parser, and are not intended for the end user. To provide a user-friendly format for processing parsing results, the Meerkat library supports conversion of binarized SPPFs to terms that reflect the underlying grammar. This approach is popular in tools that allow algebraic specification and term rewriting, for example in ASF+SDF [89]. Figure 7.11 shows a visualization of the terms corresponding to the binarized SPPF in Figure 7.9 (d).

The traversal of binarized SPPFs and generation of terms is straightforward. As shown in Section 7.3.1, packed nodes under nonterminal nodes have labels which correspond to grammar positions of the form $A ::= \alpha\cdot$. The type of a term is computed as the result of sequencing parsers (`seq` and `rule1`) and is stored in the packed nodes of a nonterminal. We traverse the binarized SPPF bottom-up, and for each SPPF node type, perform a specific action.

For terminal nodes, a terminal term is created that stores the name of the terminal and its associated matched string. For nonterminal or intermediate nodes that are ambiguous, i.e., have more than one packed node, an ambiguity term is created. An ambiguity term gets a list of terms as its children. For nonterminal nodes, a

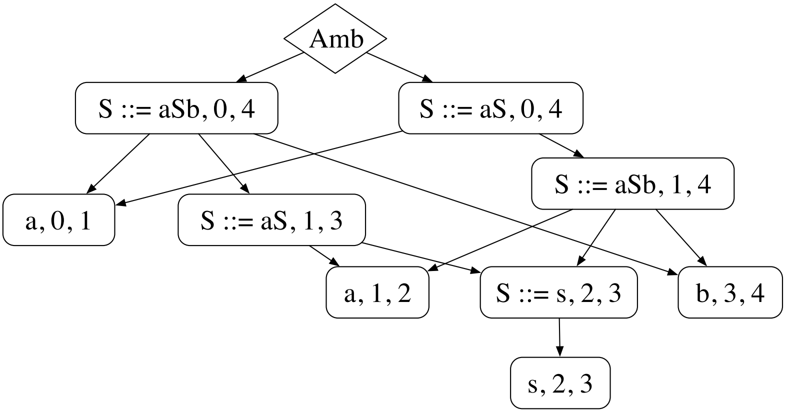


Figure 7.11: Terms for the binarized SPPF of Figure 7.9 (d).

nonterminal term is created. We bypass intermediate nodes, as they do not correspond to any constructs in the grammar and should not appear in final terms. This effectively flattens the intermediate nodes.

Besides creation of terms from a binarized SPPF, we support execution of semantic actions. Due to the inherent nondeterminism in general parsing, many parsing paths will eventually die. Therefore, it is desirable to postpone the execution of semantic actions until parsing is done. In the Meerkat library, semantic actions are stored in the packed nodes of an SPPF and executed post-parse by traversing the resulting binarized SPPF. An example of using semantic actions in the Meerkat library is shown in Figure 7.1.

The traversal mechanism for semantic actions is basically the same as for building terms, with the difference that we throw an exception when an ambiguous node is encountered. In parsing programming languages, almost always, a single parse tree should be returned. In case of an ambiguity, the user should first resolve the ambiguity, e.g., by investigating the terms corresponding to the ambiguous parse, and then run the semantic actions.

7.4 Complexity of Johnson's CPS Recognizers

In this section we start by introducing notation to support reasoning about the execution and complexity of the original Johnson's CPS recognizers. We show that the execution of such recognizers can require $O(n^{m+1})$ operations, where m is the length of the longest rule in the grammar.

7.4.1 Notation

We denote the resulting recognizers of the terminal, epsilon, seq and rule combinators of Figures 7.2, 7.4 and 7.6 as follows:

- $f_a = \text{terminal}(\text{“}a\text{”})$ denotes a recognizer for terminal a.
- $f_\epsilon = \text{epsilon}()$ denotes a recognizer for ϵ.
- $f_A = \text{rule}(f_{\alpha_1}, f_{\alpha_2}, \ldots, f_{\alpha_k})$ denotes a recognizer for nonterminal $A ::= \alpha_1 \,|\, \alpha_2 \,|\, \ldots \,|\, \alpha_k$, where $\alpha_i = x_1 x_2 \ldots x_m$ is an alternative consisting of a sequence of symbols.
- f_{α_i} is either a recognizer for a symbol, $f_{\alpha_i} = f_{x_1}$, $m = 1$, or a sequence of symbols, $f_{\alpha_i} = \text{seq}(f_{x_1}, f_{x_2}, \ldots, f_{x_m})$, $m \geq 2$.
- $f_{\alpha_1|\alpha_2|\ldots|\alpha_k}$ denotes the result of applying `reduce` in the body of the rule combinator, i.e., an unmemoized recognizer for A.

We also use the following notation:

- (f_A, i, κ) denotes a call to f_A at an input position i with a continuation κ passed to the result of calling f_A at i.
- $(R^{(A,i)}, K^{(A,i)})$ denotes the memo entry created upon the first call to f_A at an input position i, where $R^{(A,i)}$ is the result list, and $K^{(A,i)}$ is the continuation list.

For any call (f_A, i, κ), such that f_A is called at the input position i for the first time, continuations of the following forms are created:

- κ^i_A denotes the continuation (Figure 7.5, lines 12–15) that maintains the results produced by f_A at i: it records new input positions in $R^{(A,i)}$ and runs all the continuations pending for this call in $K^{(A,i)}$.
- $\kappa^i_{A::=\alpha\cdot x\beta}$ denotes the continuation that corresponds to a grammar position $A ::= \alpha \cdot x\beta$ in an alternative of A, $|\alpha| \geq 1$, and is uniquely identified by $A ::= \alpha \cdot x\beta$ and i. When f_A is called at the input position i for the first time, resulting in the unmemoized recognizer for A to be called at i with κ^i_A (Figure 7.5, line 16), continuations of this form are recursively created as follows, where function application binds stronger than $\rightarrow$:

  $$\kappa^i_{A::=\alpha\cdot x\beta} = j \rightarrow f_x(j)(\kappa^i_{A::=\alpha x\cdot\beta}),$$

  where f_x is the recognizer for the next symbol x in the alternative, and $\kappa^i_{A::=\alpha x\cdot\beta}$ corresponds to the next grammar position $A ::= \alpha x \cdot \beta$. This follows from the definition of seq, namely, the left reduce semantics, and composition via `flatMap`. Note that in case of recognizers, $\kappa^i_{A::=\alpha x\beta\cdot}$ refers to the same continuation as κ^i_A.

7.4.2 Execution

In Figure 7.5 we use data structures, such as hash maps and mutable lists, that are sufficient to explain the underlying semantics but require amortized constant or linear time for their operations. The complexity analysis of the next sections, however, assumes that certain operations execute in constant time during the execution of memoized CPS recognizers. Therefore, we need to discuss how to provide such constant-time operations.

We assume that the following operations execute in constant time: copying arguments into the stack when the function call is executed, assigning the value of a variable to another variable, and creating closures. Scala is a JVM-based language that handles primitive types by value and reference types by reference value. Reference values are fixed-size values representing addresses in memory. Therefore, passing arguments of reference types to a function, or assigning one variable to another, results in copying the reference values, which executes in constant time. To implement closures, Scala uses closure conversion, such that captured variables are turned into fields of anonymous classes, representing closures, and these fields are initialized by passing extra arguments to the constructors. Therefore, we assume that operations such as `success`, `failure`, `map`, `flatMap` and `orElse` in Figure 7.4 execute in constant time.

Now, we consider the execution of memoized CPS recognizers that performs one of the following calls at each step:

- (f_ϵ, i, κ), a call to the recognizer for ϵ. The execution continues with the call (κ, i), and the recognizer call returns after the continuation call returns.

- (f_a, i, κ), a call to the recognizer for a terminal a. If the terminal matches a substring in the input string starting from i (a constant-time operation), the execution continues with the call (κ, j), where j is the next input position after the match, otherwise no continuation is called, and the recognizer call returns.

- (f_A, i, κ), a call to the recognizer for a nonterminal A. This call requires memo-table lookup of the result of calling f_A at i. If the result is not found, i.e., this is the first call to f_A at input position i, a new function is created with two variables in its scope: the result list $R^{(A,i)}$ and continuation list $K^{(A,i)}$. As the memo table can be implemented as an array of length $n+1$, n is the length of the input, the lookup operation can execute in constant time. In addition, as the continuation list can be implemented as a linked list, and the result list as an array of size $n+1$, addition of a new element into the lists (`+=`) and element lookup into the result list (`contains`) can execute in constant time.

  The execution of (f_A, i, κ) continues with addition of κ to $K^{(A,i)}$ (if- and else-branch of `memo_result`). Depending on the check whether $K^{(A,i)}$ is empty (a constant-time operation), the execution continues with either the call to the unmemoized recognizer for A, $(f_{\alpha_1|\alpha_2|\ldots|\alpha_k}, i, \kappa^i_A)$, or iteration over $R^{(A,i)}$ (a linear operation) calling κ for each recorded input position. The call $(f_{\alpha_1|\alpha_2|\ldots|\alpha_k}, i, \kappa^i_A)$ will perform a constant number of steps to combine the results of calling f_{α_i} at i

(via `orElse`), and will eventually lead to a constant number of calls $(f_{\alpha_1}, i, \kappa^i_A), \ldots,$ $(f_{\alpha_k}, i, \kappa^i_A)$, corresponding to the alternatives of A.

- $(f_\alpha, i, \kappa^i_A)$, a call to the recognizer for an alternative of A. If $|\alpha| > 1$, this call results in a constant number of composition steps (via `flatMap`). Then, the continuations corresponding to the grammar positions in α are created, and the call $(f_x, i, \kappa^i_{A::=x\cdot\gamma})$, assuming $\alpha = x\gamma$, to the recognizer for the first symbol in α is made.

- $(\kappa^i_{A::=\alpha\cdot x\beta}, j)$ or (κ^i_A, j), a call to a continuation. The former call directly results in $(f_x, j, \kappa^i_{A::=\alpha x\cdot\beta})$, a call to the recognizer for x with the next continuation $\kappa^i_{A::=\alpha x\cdot\beta}$. The latter call leads to the check (constant-time) whether j exists in the result list, and if not, j is added (constant-time, as discussed above) to the result list. Finally, iteration over the continuation list $K^{(A,i)}$ (a linear operation) runs each continuation with the new input position j.

Consider calls of the forms (f_x, i, κ), where x is any symbol, and (κ^i_A, j). The execution of memoized CPS recognizers continues linearly until either a call (f_A, i, κ) or (κ^i_A, j) is executed: both calls may result in iteration over a list, the size of which depends on n, calling a continuation in each iteration step. When a call (f_A, i, κ) is the first call to f_A at the input position i, it does not lead to iteration but requires an $O(n)$ operation to create an array of size $n + 1$ for the result list.

7.4.3 Complexity

In this section we show that the execution of the original Johnson's CPS recognizers can require $O(n^{m+1})$ operations, where m is the length of the longest rule in the grammar. This unbounded polynomial behavior can be observed by the family of highly ambiguous grammars [42]:

$$S ::= S^m \mid SS \mid b \mid \epsilon,$$

where $m \geq 3$, and S^m denotes a sequence of S's of length m, e.g., SSS for $m = 3$. Because of the last three alternatives, any, possibly empty, sequence of b's can be recognized by S.

We consider parsing string b^n. The memoization technique of Section 7.2.3 ensures that for any input position i, there will be at most one call (f_{S^m}, i, κ^i_S), corresponding to the first alternative of S, made upon the first call to f_S at i. In the following we show that the total number of calls that will be made to all κ^i_S, $0 \leq i \leq n$, is $O(n^{m+1})$.

Let us consider calls (f_{S^m}, i, κ^i_S), $0 \leq i \leq n$. Each of these calls will first create continuations corresponding to the input position i and each of the grammar positions in the sequence S^m, and then, will make the call $(f_S, i, \kappa^i_{S::=S\cdot SS^{m-2}})$ to the recognizer for the first symbol in the alternative S^m. First, we consider the call:

$$(f_S, 0, \kappa^0_{S::=S\cdot SS^{m-2}}),$$

corresponding to input position 0. This call will result in addition of the continuation $\kappa^0_{S::=S\cdot SS^{m-2}}$ to $K^{(S,0)}$, and therefore, for each i_1 in $R^{(S,0)}$ a call $(\kappa^0_{S::=S\cdot SS^{m-2}}, i_1)$ will be made. In turn, each of these calls will result in the call to the recognizer for the second S in S^m:

$$(f_S, i_1, \kappa^0_{S::=SS\cdot S^{m-2}}),$$

and addition of the continuation $\kappa^0_{S::=SS\cdot S^{m-2}}$ to $K^{(S,i_1)}$. Again, for each i_2 in $R^{(S,i_1)}$, such that $i_1 \leq i_2$, a call $(\kappa^0_{S::=SS\cdot S^{m-2}}, i_2)$ will be made.

Given that $0 \leq i_1 \leq i_2 \leq n$, the total number of calls that will be made to $\kappa^0_{S::=SS\cdot S^{m-2}}$ is equal to $\binom{n+2}{2}$. Some of these calls, however, are duplicate. Indeed, for each i_2, there will be multiple i_1, $0 \leq i_1 \leq i_2$, such that $\kappa^0_{S::=SS\cdot S^{m-2}} \in K^{(S,i_1)}$ and $i_2 \in R^{(S,i_1)}$. Note that each duplicate call $(\kappa^0_{S::=SS\cdot S^{m-2}}, i_2)$ will result in addition of the continuation $\kappa^0_{S::=SSS\cdot S^{m-3}}$ to $K^{(S,i_2)}$, thus leading to the same continuation being added to the same continuation list multiple times. Finally, the total number of calls that will be made to $\kappa^0_{S::=SSS\cdot S^{m-3}}$ is equal to $\binom{n+3}{3}$.

Now, if we continue further, it can be seen that the total number of calls that will be made to κ^0_S, corresponding to the end of the alternative S^m, is equal to the number of all the combinations with repetition for the respective indices $0 \leq i_1 \leq i_2 \leq \ldots \leq i_{m-1} \leq i_m \leq n$, which is $\binom{n+m}{m}$.

Finally, if we consider all the calls $(f_S, i, \kappa^i_{S::=S\cdot SS^{m-2}})$, i.e., for all i where $0 \leq i \leq n$, the total number of calls made to the continuations κ^i_S, resulting from the first alternative, is $\binom{n+m+1}{m+1}$, which is a polynomial in n of order $m+1$.

7.5 Complexity of CPS Parsers

In Section 7.4 we showed that the execution of the original Johnson's CPS recognizers can require $O(n^{m+1})$ operations, where m is the length of the longest rule in the grammar. To obtain cubic CPS recognizers, we extended the memoization strategy of the original Johnson's CPS recognizers by adding memoization on continuations (Section 7.2.4). Finally, in Section 7.3 we extended the resulting cubic CPS recognizers to CPS parsers that produce binarized SPPFs in cubic time and space. In this section we give a formal proof that the complexity of our CPS parsers is indeed $O(n^3)$, where n is the length of the input. Note that we do not give a separate proof for the CPS recognizers as this proof is similar to the one for the CPS parsers.

We start by adapting the notation of Section 7.4.1 to the parser version. First, functions f_a, f_ϵ, f_A now denote the respective parsers instead of recognizers. Second, result list $R^{(A,i)}$ stores nonterminal nodes (A, i, j), instead of input positions. Finally, for any call (f_A, i, κ), such that the parser f_A is called at the input position i for the first time, continuations of the following forms can be created:

- κ^i_A denotes the continuation that maintains the results produced by f_A at i, now, recording new nonterminal nodes (A, i, j) in $R^{(A,i)}$.

  In addition, $\kappa^i_{A::=\alpha\cdot}$ now denotes the continuation that corresponds to a grammar position $A ::= \alpha\cdot$ and is created as:

$$\kappa^i_{A::=\alpha\cdot} = t \to \kappa^i_A(h^i_{A::=\alpha}(t)),$$

where $h^i_{A::=\alpha}$ is a function (passed to `map` in `rule1` of Figure 7.10) which is uniquely identified by i and $A ::= \alpha$. $h^i_{A::=\alpha}$ takes a non-packed node t and creates a nonterminal node (A, i, j). If $|\alpha| > 1$, t is an intermediate node of the form $(A ::= \alpha\cdot, i, j)$, otherwise a symbol node of the form (x, i, j) corresponding to x, the only symbol in α.

- $\kappa^i_{A::=\alpha\cdot x\beta}$ denotes the continuation that corresponds to a grammar position $A ::= \alpha \cdot x\beta$ in an alternative of A, $|\alpha| \geq 1$, and is uniquely identified by $A ::= \alpha \cdot x\beta$ and i. Upon the first call to f_A at i, continuations of this form are recursively created as:

  $$\kappa^i_{A::=\alpha\cdot x\beta} = t \to g^i_{A::=\alpha\cdot x\beta}(t)(\kappa^i_{A::=\alpha x\cdot\beta}),$$

  where t is either an intermediate node of the form $(A ::= \alpha \cdot x\beta, i, j)$ when $|\alpha| > 1$, or a symbol node of the form (y, i, j) when $|\alpha |= 1$. $g^i_{A::=\alpha\cdot x\beta}$ is a function (passed to `flatMap` in `seq2` of Figure 7.10), which is uniquely identified by i and $A ::= \alpha \cdot x\beta$. This function calls the parser for the next symbol f_x at the right extent of t, say j. Then, it creates the following continuation and passes this continuation to the result of $f_x(j)$:

  $$\kappa^{i,j}_{A::=\alpha x\cdot\beta} = t \to \kappa^i_{A::=\alpha x\cdot\beta}(h^{i,j}_{A::=\alpha x\cdot\beta}(t)),$$

  where t is a symbol node of the form (x, j, k). $h^{i,j}_{A::=\alpha x\cdot\beta}$ is the function (passed to `map` in `seq2` of Figure 7.10) created by $g^i_{A::=\alpha\cdot x\beta}$ to construct an intermediate node $(A ::= \alpha x \cdot \beta, i, k)$ given (x, j, k). Finally, $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$ calls $\kappa^i_{A::=\alpha x\cdot\beta}$ with the resulting intermediate node.

Lemma 1 For any parser f_B and any input position j, where $0 \leq j \leq n$, the number of continuations in the continuation list $K^{(B,j)}$ is $O(n)$.

Proof 1 For any nonterminal A such that $A ::= \alpha B\beta$ is an alternative of A, we have only the following calls that add a continuation to $K^{(B,j)}$:

1. $(f_B, j, \kappa^i_{A::=B\cdot\beta})$ when $|\alpha |= 0$ and $i = j$. This call can only result from $(f_{A::=B\beta}, j, \kappa^i_{A::=B\beta\cdot})$.

2. $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$ when $|\alpha| \geq 1$ and $0 \leq i \leq j$. This call can only result from $(\kappa^i_{A::=\alpha\cdot B\beta}, t)$, where t is an intermediate node $(A ::= \alpha\cdot B\beta, i, j)$, $|\alpha| > 1$, or a symbol node (y, i, j), $|\alpha |= 1$.

In the first case, given that CPS parsers for nonterminals are memoized, there will be at most one call to the parser for an alternative at each input position, such as $(f_{A::=B\beta}, j, \kappa^i_{A::=B\beta\cdot})$, and therefore, at most one call to the parser for the first symbol in the alternative, such as $(f_B, j, \kappa^i_{A::=B\cdot\beta})$. Thus the continuation $\kappa^i_{A::=B\cdot\beta}$ can be added to $K^{(B,j)}$ at most once. In the second case, given that continuations of the form $\kappa^i_{A::=\alpha\cdot B\beta}$ are memoized, for any i, $0 \leq i \leq j$, there will be at most one call $(\kappa^i_{A::=\alpha\cdot B\beta}, t)$ resulting in creation of $\kappa^{i,j}_{A::=\alpha B\cdot\beta}$ and the call $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$. Thus the continuation $\kappa^{i,j}_{A::=\alpha B\cdot\beta}$ is uniquely identified by i, j and $A ::= \alpha B \cdot \beta$ and can be added to $K^{(B,j)}$ at most once. Finally, given that $0 \leq i \leq n$, the total number of continuations in $K^{(B,j)}$ is at most $O(n)$. □

Lemma 2 For any parser f_A and any input position i, where $0 \leq i \leq n$, the number of elements in $R^{(A,i)}$ is $O(n)$.

Proof 2 Given that continuations of the form $\kappa^i_{A:=\alpha\cdot}$ and κ^i_A are memoized, for any non-packed node t with left extent i and right extent j, $i \leq j \leq n$, there will be at most one call $(\kappa^i_{A:=\alpha\cdot}, t)$ resulting in $(\kappa^i_A, (A, i, j))$, and there will be at most one call $(\kappa^i_A, (A, i, j))$ adding (A, i, j) to $R^{(A,i)}$. Thus the number of elements in $R^{(A,i)}$ is at most $O(n)$. □

Theorem 1 The complexity of CPS parsers that construct a binarized SPPF is $O(n^3)$.

Proof 3 We consider calls of the forms:

1. $(f_x, j, \kappa^i_{A::=x\cdot\beta})$ and $(f_x, j, \kappa^{i,j}_{A::=\alpha x\cdot\beta})$, $|\alpha| \geq 1$
2. $(\kappa^i_{A::=\alpha\cdot x\beta}, t)$
3. $(\kappa^i_A, (A, i, j))$

where $0 \leq i \leq j \leq n$, and t is a non-packed node with left extent i and right extent j. The execution of CPS parsers continues linearly until either a call to the parser for a nonterminal, of the form $(f_B, j, \kappa^i_{A::=B\cdot\beta})$ and $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$, or a call of form 2 or 3 is executed. In the parser version, a call of form 2 may require an $O(n)$ operation to create a continuation $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$[6]. We show that there will be at most $O(n^2)$ calls of form 1. Also, there will be at most $O(n^2)$ calls of form 2, each of which creates a continuation $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$, and at most $O(n^2)$ calls of form 3, each of which results in iteration over a continuation list.

Similar to the proof of Lemma 1, there are only two forms of calls that may lead to a call of form 1. Given that CPS parsers for nonterminals are memoized, there will

[6] Although in the complexity analysis we assume that all continuations are memoized, it can be shown that memoizing continuations of this form is not needed. We use this as an optimization in the Meerkat library.

be at most $O(n)$ calls of the form $(f_{A::=x\beta}, j, \kappa^i_{A::=x\beta\cdot})$ resulting in $(f_x, j, \kappa^i_{A::=x\cdot\beta})$, where $0 \leq i = j \leq n$. Also, given that continuations of the form $\kappa^i_{A::=\alpha\cdot x\beta}$ are memoized, there will be at most $O(n^2)$ calls of form 2 creating $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$ and resulting in $(f_x, j, \kappa^{i,j}_{A::=\alpha x\cdot\beta})$, where $0 \leq i \leq j \leq n$. Thus there will be at most $O(n^2)$ calls of form 1. Given that continuations of the form κ^i_A are memoized, there will be also at most $O(n^2)$ calls of form 3, $0 \leq i \leq j \leq n$, resulting in iteration.

Each call of the form $(f_B, j, \kappa^i_{A::=B\cdot\beta})$ or $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$, when it is the first call to f_B at the input position j, or each call of form 2 may result in a constant number of $O(n)$ operations to create arrays of size $n+1$ (to initialize a result list and/or to create memoized continuations) followed by a constant number of other calls of form 1 (already subsumed by the $O(n^2)$ calls above). Each call of the form $(f_B, j, \kappa^i_{A::=B\cdot\beta})$ or $(f_B, j, \kappa^{i,j}_{A::=\alpha B\cdot\beta})$, when it is not the first call to f_B at the input position j, or each call of form 3 may result in iteration over a list leading to at most $O(n)$ other continuation calls (by Lemma 1 and 2). Each of the $O(n)$ other continuation calls is either a duplicate call eliminated by the memoization, or a call to a continuation of the form $\kappa^i_{A::=\alpha\cdot x\beta}$ (already subsumed by the $O(n^2)$ continuation calls above), or of the forms $\kappa^{i,j}_{A::=\alpha x\cdot\beta}$ and $\kappa^i_{A=\alpha\cdot}$ directly resulting in a call already subsumed by the $O(n^2)$ calls above. Thus the complexity of CPS parsers that construct binarized SPPFs is at most $O(n^3)$. □

7.6 Evaluation

In this section we evaluate the performance of CPS parsers, as implemented in the Meerkat library. We use the highly ambiguous grammar Γ_3, $S ::= SSS \mid SS \mid b$, and the grammar of Java. The results show that Meerkat parsers are cubic on the highly ambiguous grammar, and behave nearly linearly on the Java grammar. The experiments were carried out on a machine running Mac OS X 10.9.4 on a quad-core Intel Core i7 2.6 GHz CPU with 16 GB of memory. The Meerkat library was compiled with Scala 2.11.2 and ran on a 64-Bit Oracle HotSpot$^{\text{TM}}$ JVM version 1.7.0_55. The reported time is the mean running time (CPU user time) of 10 runs for each parse. To allow for JIT optimizations, the first three runs for each file were skipped.

7.6.1 Parsing Γ_3

To evaluate the runtime performance of CPS parsers in the worst case, we ran the Meerkat parser for Γ_3 on sequences of b's, varying from 10 to 500. Γ_3 triggers the worst-case behavior for CPS parsers. Standard GLR parsers, except for BRNGLR that produces binarized SPPFs, are $O(n^4)$ on Γ_3. The results are shown in Figure 7.12. As can be seen, the resulting curve is cubic with high confidence, as indicated by the R^2 value of 0.9998.

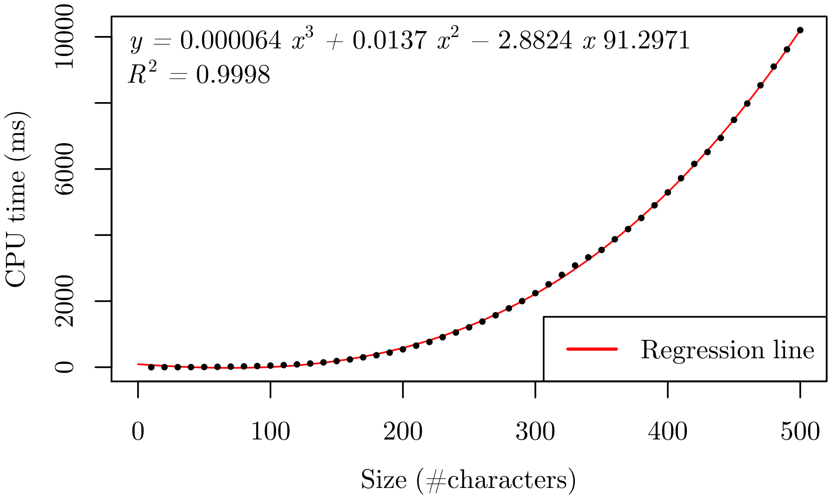


Figure 7.12: Runtime of parsing string of b's using Γ_3.

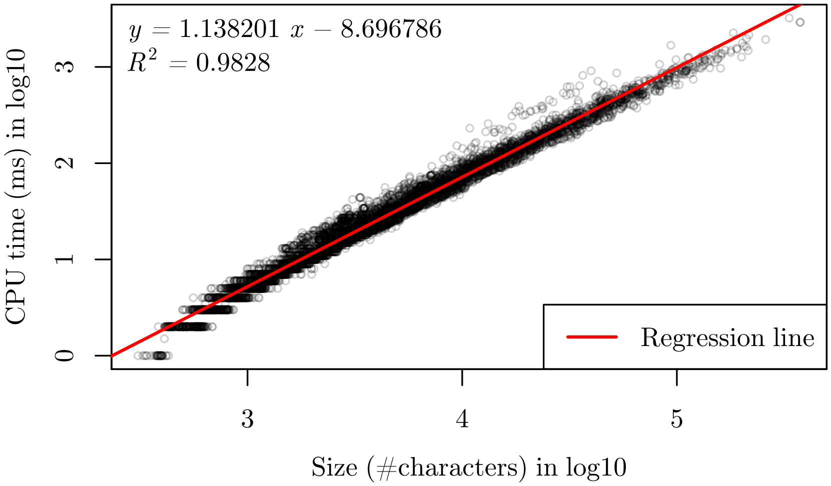


Figure 7.13: Runtime of parsing Java files.

7.6.2 Parsing Java

To evaluate the performance of CPS parsers on grammars of real programming languages, we have chosen the grammar of Java 7 from the main part of the Java Language Specification [32]. This grammar has a left-recursive unambiguous expression grammar that encodes operator precedence by introducing new nonterminals. We ran the parser for the character-level Java grammar[7] for 7449 Java files in the source release of JDK 1.7.0_60-b19. All files were parsed successfully and without ambiguity. Figure 7.13 shows the running time corresponding to the execution of the parser for the character-level Java grammar for increasing input sizes. We use a log-log (base 10) plot. The goodness of the fit is indicated by the R^2 value of 0.9828. The regression line equation (log-log scale) is written in the plot. As the regression is calculated after a log transformation of the original data, and the coefficient is close to one (1.138201), we can conclude that the running time for Java is near linear ($y \approx x^{1.138201}$).

7.7 Related Work

Conventional parser combinators are recursive-descent like. Therefore, a natural choice for generalizing parser combinators to support all context-free grammars is to generalize recursive-descent parsing. The main challenge is support for left recursion. In this section we discuss a number of parsing techniques that generalize recursive-descent parsing, with a focus on how support for left recursion is provided. We discuss each work based on the following aspects:

1. The mechanism used to support direct left recursion.
2. Support for indirect/hidden left recursion: not supported, requires extra mechanism, or *uniformly* works with the mechanism for direct left recursion.
3. The worst-case runtime complexity of the recognizer and parser.
4. The output of a parser (single parse tree or a parse forest).

Table 7.1 gives an overview of related work based on these aspects. Note that there are other parser combinator tools, e.g., Parsec [57] and the Scala parser combinator library [64], which are essentially basic recursive-descent recognizers (see Section 7.2), and we do not discuss them in this section.

7.7.1 Left-Recursion Curtailment Using the Input Length

Frost *et al.* [26] present an approach for supporting direct left recursion based on the length of the input. In this approach, the number of calls to recognizers at each input position is maintained. For non-left-recursive recognizers, the count will be at most one. For left-recursive ones, the count increases by each recursive call at the same input position. When the count exceeds the number of remaining tokens in the input string plus one, the call is curtailed, as no successful parse is possible at this point.

In this approach, each left-recursive recognizer can be called at each input position at most n times, where n is the length of the input. This brings the worst-case complexity of this approach to $O(n^4)$, compared to the expected $O(n^3)$ complexity. In addition, this approach requires extra machinery to accommodate indirect left

[7] `https://github.com/meerkat-parser/grammars`

Table 7.1: Overview of related work that extend recursive-descent parsing towards a general parsing solution.

Approach	Mechanism	Indirect/Hidden	Worst-case Complexity		Output
			Recognizer	Parser	
Left-recursion curtailment	Input-length heuristics	extra	$O(n^4)$	unbounded polynomial	Tomita-style SPPF
Left recursion in PEGs	Memoization and growing the seed	extra	$O(n^2)$	$O(n^2)$	Single parse tree
Cancellation parsing	Passing cancellation sets	extra	exponential	exponential	Single parse tree
ANTLR 4	Left-recursion elimination by rewriting	no	$O(n^4)$	$O(n^4)$	Single parse tree
GLL	Cycles in the GSS	uniform	$O(n^3)$	$O(n^3)$	Binarized SPPF
CPS parsers	Memoization in CPS	uniform	$O(n^3)$	$O(n^3)$	Binarized SPPF

recursion. Since the parser version of Frost *et al.*'s approach creates a Tomita-style SPPF, it is of unbounded polynomial complexity.

One essential difference between Frost *et al.*'s approach and CPS parsers is the moment when a chain of left-recursive calls is terminated. In CPS parsers, this happens when the second call to a recognizer at the same input position is made. Then, the results for left-recursive recognizers are effectively computed in a loop: as long as a new result is produced, the terminated parsing paths, recorded as continuations, are restarted at the new input position. As a result, handling left-recursive rules is more efficient in CPS parsers. Finally, it should be noted that Frost *et al.*'s approach cannot be used in cases where the length of the input is not known, for example, when reading from a network socket.

7.7.2 Left Recursion in PEGs

Parsing Expression Grammars (PEGs) [25] are an alternative to context-free grammars, where the unordered alternation operator is replaced with a *prioritized choice* operator. PEGs are commonly implemented as recursive-descent parsers with local backtracking: the alternatives of a nonterminal are tried in order, and the next alternative is tried only if the current one fails. As a result, PEGs produce at most one parse tree and cannot be ambiguous. PEGs suffer from the quirk that if an alternative is a prefix of another one, the second alternative is never tried. This, for example, leads to parse error on `"ab"` for the grammar $A ::= a|ab$, although it is apparent that the second alternative can correctly parse this input.

Packrat parsing [24] uses memoization to implement PEGs in linear time. However, Packrat parsers, like other recursive-descent parsers, do not support left recursion. Warth *et al.* [98] propose a mechanism to support left recursion by modifying the memoization in Packrat parsing. In this approach, a special fail value is put into the memo table before calling a parser at an input position for the first time, so that left-recursive calls fail. This ensures termination of the parser for a left-recursive nonterminal. Then, if any of the non-left-recursive alternatives can produce a result, the parser is restarted to re-try the left-recursive ones. As long as a new result is produced, the new result replaces the previous one in the memo table, and the parser is called again at this input position, reusing the last result from the memo table for the left-recursive calls. This process is called *growing the seed.*

For indirect left recursion, this approach uses an extra data structure, called *rule invocation stack*, to maintain the recursive calls between mutually left-recursive nonterminals. Warth *et al.*'s approach breaks the linear runtime guarantee of Packrat parsing, as for some left-recursive grammars, the runtime complexity of this approach is $O(n^2)$ [98]. Tratt [86] identifies a problem with Warth *et al.*'s approach for rules that are both left and right recursive, e.g., $E ::= E + E$. For such rules, this approach is biased towards producing a right-associative derivation, which does not conform to the semantics of PEGs.

7.7.3 Cancellation Parsing

Cancellation parsing [65] is a technique to support left recursion for Definite Clause Grammars (DCGs) in Prolog. The basic idea behind this technique is that each call to a nonterminal takes a set of already called nonterminals, the cancellation set. If the nonterminal is already in the set, the parser backtracks and tries the next alternative, otherwise the nonterminal is added to the cancellation set. This guarantees termination in presence of left recursion. To construct the parse trees corresponding to the terminated paths, for each left-recursive nonterminal A, a special token $\overline{A}$ is put before the rest of the token stream, using `untoken`, and a rule $A ::= \overline{A}$ is added to the grammar. After inserting $\overline{A}$, nonterminal A is called again with the current cancellation set.

```
A(c) ::= [A ∉ c] A(A ∪ c) 'a' untoken(Ā)  A(c)
       | 'a' untoken(Ā) A(c)
       | Ā        // Added rule for the inserted tokens
```

This approach works for direct and indirect left recursion, but does not work for hidden left recursion, i.e., when a left-recursive call is hidden behind a nullable nonterminal, e.g., $A ::= BAa$ and $B \overset{*}{\Rightarrow} \epsilon$. To support hidden left recursion, this approach requires grammar analysis to identify nullable nonterminals, and pass a boolean flag to each call. The need for grammar analysis for hidden left recursion and customization of left-recursive definitions (`untoken` and rules for $\overline{A}$) makes this approach fundamentally different from ours. Finally, this parsing technique is designed to have no side effects [65], and does not memoize previous results, thus is of worst-case exponential complexity.

7.7.4 ANTLR 4

ANTLR 4 [70] supports direct left-recursive rules by rewriting them to non-left-recursive ones. During this rewriting process, ANTLR inserts semantic predicates to resolve ambiguities based on the order of alternatives. The resulting parsers mimic the operator precedence technique by Clarke [14]: the rules that come earlier have higher precedence, and all rules are left-associative by default, unless explicitly marked as right-associative. ANTLR 4 does not support indirect left recursion because rewriting grammars to eliminate indirect left-recursion results in large grammars that have no obvious relationship with the original ones.

ANTLR 4 uses the Adaptive LL(*), ALL(*), strategy, in which a sub-parse is invoked for each alternative of a nonterminal and intermediate results are cached. ALL(*) effectively uses global backtracking to avoid the problems with PEG-style backtracking of previous versions of ANTLR. ALL(*) parsers have worst-case complexity of $O(n^4)$ and produce at most one derivation, since ambiguities are resolved during parsing. The ambiguity resolution mechanism is based on the order of alternatives, in which the sub-parse with the lowest alternative number (appearing earlier) is preferred. Because of complex grammar transformations performed by ANTLR 4 before parsing, and its lack of direct support for left recursion, this parsing strategy is not suitable for parser combinators.

7.7.5 GLL Parsing

Generalized LL (GLL) [78] is a fully general, worst-case cubic parsing algorithm. GLL uses a Graph Structured Stack (GSS) that handles multiple function call stacks, and produces binarized SPPFs. The problem of left recursion (direct/indirect/hidden) is uniformly solved by allowing cycles in the GSS. GLL parsers are recursive-descent like, and have a close relationship with the grammar.

Among all the related work we discussed so far, GLL parsing is the closest to our work, especially if we consider a version of GLL which uses a more efficient GSS (see Chapter 2). In fact, CPS parsers have the same performance characteristics as GLL parsers with the new GSS. Although CPS parsers and GLL use very different terms to describe their inner workings, presumably because of different communities they have been developed in, there are many similarities in these two approaches. Most notably, left recursion is handled in both approaches essentially in the same way.

In a GLL parser (with new GSS) when the parser is before a nonterminal, a GSS node corresponding to the nonterminal and the current input position is searched. If the GSS node exists, the GSS edge is added, recording the current grammar position, and the previous parsing results associated with this GSS node are reused. When a new result is added to the results associated with a GSS node, the parser will explore the paths recorded on outgoing GSS edges with the new result. Similarly, in CPS parsers, if a memoized parser has been already called at the current input position, a continuation is added, recording the current position in the sequence, and the parsing results associated with the parser are reused. When a new result is added to the results of a parser, the recorded continuations will be called with the new result.

The main difference between a GLL parser and CPS parser is how the control flow is designed. GLL uses a GSS, which is a global data structure that encodes all parsing paths, while in a CPS parser, the control flow is encoded in continuation-passing style. It is in principle possible to realize a direct embedding of context-free grammars based on GLL parsing, although such an implementation may not be trivial. The GLL parsing algorithm [78] was designed for a code generation setting, in which a grammar processing phase generates the required labels for grammar positions. In parser combinators based on GLL, these labels should be dynamically represented. For example, Spiewak [81] shows how to build parser combinators based on GLL by encoding GSS nodes and edges as closures. This encoding resembles a form of continuation-passing style. Moreover, parser combinators based on GLL may benefit from a different GSS structure such as the one we presented in Chapter 2, as it is more similar to function memoization. We believe that our general parser combinators, compared to a functional formulation of GLL, are a more natural and elegant choice for realizing general parser combinators, as they offer a more straightforward generalization of traditional parser combinators.

7.8 Conclusions

In this chapter we presented a foundation for general parser combinators based on an extension of Johnson's CPS recognizers. Johnson stated that for constructing a parse forest from his approach *"a straightforward implementation attempt would probably be very complicated"* [43]. To the best of our knowledge no parser version of Johnson's CPS recognizers existed before our work. One of our core contributions is the extension of CPS recognizers to parsers that construct binarized SPPFs [78,80]. We showed that binarized SPPFs are a perfect fit for CPS recognizers. In particular, intermediate nodes provide a natural way to build a node from a binary sequence combinator. The results of parsing Java show that CPS parsers are practical for large, real-world grammars, even in a dynamic parser combinator setting where static grammar analysis is not an option. As future work, we plan to experiment with more programming languages and explore optimization opportunities in the Meerkat library.

Chapter 8

Conclusions

In this thesis we have presented the design and implementation of practical general top-down parsers. Top-down parsing, usually in the form of recursive-descent parsing, is efficient, easy to understand and is widely used to manually write parsers for real programming languages. However, due to the difficulty of dealing with left recursion, top-down parsers always suffered from expressiveness problems compared to their bottom-up counterparts.

In many applications of parsing, such as designing domain-specific languages, where factors such as expressiveness and ease of use are more important than mere performance, parsing tools based on general parsing algorithms are considered. General parsing algorithms support all context-free grammars, including the ones with left recursion, and can present all the ambiguities in a compact parse forest format. Most of the research on developing general parsers has been devoted to bottom-up parsers, most notably on extending and improving the Generalized LR (GLR) parsing algorithm.

The Generalized LL (GLL) parsing algorithm, which has been developed by Scott and Johnstone in 2010, provided a viable alternative to GLR. GLL parsers are recursive-descent like and are relatively easy to understand. They can even be written by hand, and debugged using a programming language IDE. GLL parsers support all context-free grammars and have worst case cubic runtime complexity, but can run nearly linearly on grammars of real programming languages.

The other very important, and less widely known work, on generalizing recursive-descent parsing is Mark Johnson's work on Continuation-Passing Style (CPS) recognizers. While exploring an implementation of parser combinators based on GLL parsing, we stumbled upon Johnson's work by chance. Although Johnson's combinators are formulated very differently compared to GLL, they are so similar in dealing with left recursion that they can be considered the same algorithm. In fact, Johnson's recognizers can be considered as a functional implementation of GLL parsing.

Our work in this thesis has been inspired by both GLL and Johnson's CPS recognizers. Working with Johnson's recognizers led us to propose a more efficient GSS for GLL, which resembles traditional function memoization, and also shaped our work on data-dependent grammars. Our experience with GLL also gave us insight into how to make Johnson's CPS recognizers cubic and generate SPPF from them. In fact, part of our work can be seen as merging and unifying the GLL and Johnson's CPS recognizers.

Our main research focus in this thesis has been to realize general top-down parsers that are practical for parsing real programming languages. Since a general parser without disambiguation is not useful for parsing real programming languages, which almost always should be unambiguous, we have spent considerable time on disambiguation, especially operator precedence disambiguation. To achieve our goal we worked on the following topics:

- Algorithmic and implementation optimizations of the GLL parsing algorithm.
- Extension of GLL to support data-dependent grammars, to deal with languages that are not context-free and to provide a generic framework for implementing disambiguation constructs.
- Semantics and an efficient implementation for operator precedence disambiguation.
- General parser combinators, providing a viable alternative for users who would like to have a general parser directly embedded in a programming language such as Scala.

We have developed Iguana, a data-dependent parsing framework based on GLL, to evaluate our research results. The results of our performance evaluation show that Iguana is practical for parsing real programming languages, and in terms of performance, is comparable to a mature parsing tool such as ANTLR. We have also developed Meerkat [38], a general parser combinator library in Scala. We showed the practicality of Meerkat by encoding the full grammar of Java as a set of combinators and parsing large number of Java source files. In the remaining of this chapter we revisit the research questions, and discuss the future work that could be built on top of our work.

8.1 Revisiting the Research Questions

Research Question 1. GLL is a relatively new general parsing algorithm that has not been yet widely used in practice. Can we make GLL faster, and build efficient general parsers based on GLL?

The answer to this question is presented in Chapters 2 and 6. In Chapter 2 we presented a modification to the Graph Structured Stack (GSS), an important internal data structure of GLL, that leads to significant performance improvement. Our change was inspired by Johnson's CPS recognizers, and makes GSS more similar to the call stack of programming languages. This modification to GSS significantly reduces the number of descriptors, the unit of work in GLL parsing, and thus leads to considerable performance improvement. We believe this change not only makes GLL parsing faster, but also makes it easier to understand. This change also laid out the basis for our work on the extension of GLL to support data-dependent grammars. In Chapter 6 we presented the results of our performance comparison of Iguana and ANTLR. The results show that Iguana is practical for parsing real programming languages and has comparable performance to ANTLR.

Research Question 2. Using general parsing algorithms for parsing programming languages goes hand in hand with (declarative) disambiguation. To build practical general parsers, it is essential to support disambiguation. Disambiguation constructs are typically implemented in the context of a specific parsing algorithm. Is it possible to implement various disambiguation constructs without the knowledge of the underlying parsing technique?

The answer to this question is presented in Chapter 3. We found data-dependent grammars to be an excellent abstract intermediate language for defining various disambiguation constructs, and showed how to extend GLL to support data-dependent grammars. As data-dependent grammars are rather low-level, we have also provided high-level disambiguation constructs, e.g., for operator precedence and indentation-sensitivity, that desugar to data-dependent grammars. We discussed the application of our technique to resolve various ambiguities, for example, the ones found in indentation-sensitive languages such as Haskell and Python, conditional directives in C#, `typedef` ambiguity in C, and intricate cases of operator precedence in OCaml.

Research Question 3. How can we deal with intricate cases of operator precedence ambiguity that are present in functional programming languages such as OCaml?

The answer to this question is presented in Chapter 4. We introduced a derivation-based semantics for operator precedence disambiguation that is independent of the underlying parsing technique, and is safe, i.e., does not remove sentences from the language when there is no ambiguity. In addition, our operator precedence semantics can deal with so-called *deep* operator precedence cases that often occur in functional programming languages such as OCaml. In contrast to SDF-style operator precedence disambiguation, our operator precedence semantics considers arbitrary distance between subtrees that form an illegal combination and can resolve the ambiguity.

Our safe specification of operator precedence rules is implemented by an automatic grammar rewriting process that preserves the shape of the parse trees, conforming to the original ambiguous grammar. This rewriting, however, could lead to very large grammars, which affects the runtime of the parser.

Research Question 4. How can we implement our safe operator precedence technique in a way that does not require a grammar transformation that increases the size of the grammar, and is independent of the underlying parsing algorithm?

The answer to this question is presented in Chapter 5. We provide an implementation of the safe operator precedence semantics based on data-dependent grammars. This implementation has the advantage that it does not depend on a grammar transformation that increases the size of the grammar, and is efficient.

Research Question 5. How can we implement general parser combinators that provide both the expressiveness and worst-case cubic runtime of traditional general parsers, and the flexibility of parser combinators?

The answer to this question is presented in Chapter 7. We started with Johnson's Continuation-Passing Style (CPS) recognizers and applied a modification to the memoization strategy that guarantees worst-case cubic runtime. Then, we showed how to extend the cubic CPS recognizers to fully general parsers that produce binarized SPPFs in cubic time and space. We presented a parser combinator library in Scala, called Meerkat, that is based on our cubic CPS parsers.

We have demonstrated the practicality of Meerkat parsers by encoding the grammar of Java as a set of combinators using the Meerkat library, and parsed a large number of Java files. In comparison to Iguana, Meerkat parsers are more flexible: there is no need for a separate textual syntax and the language engineer can directly modify existing combinators or define new ones in Scala.

Since we have worked on both Iguana and Meerkat, one might ask when to choose one over the other. The answer to this question highly depends on the specific use case. Moreover, this discussion is not only restricted to Iguana vs. Meerkat, rather to the wider subject of deep vs. shallow embedding [30]. Iguana is a grammar interpreter (deep embedding), and can potentially have better performance. In Iguana, we have more fine-grained control over the grammar representation and the interpreter. In case of Meerkat (shallow embedding), the grammar representation is highly tied to the host programming language (in our case Scala), and we have no control over the interpreter, as it is the host programming language runtime itself. On the other hand, shallow embeddings like Meerkat are more flexible and extensible.

8.2 Future Work

In the following we outline some directions that could be explored in the future.

Error recovery

One of the important topics that we did not discuss in this thesis is error recovery. Top-down parsers have a good reputation for error reporting and error recovery, and we suspect it should be easier to provide good error recovery features for GLL, say compared to GLR. Currently, Iguana reports the parse error location, but does not provide any error recovery features. Providing error recovery should be the highest priority feature, especially for using Iguana in language workbenches and development tools.

Grammar debugger

Iguana can be configured to print extensive logging information. We found these execution traces very useful when trying to determine the cause of a parse error or an ambiguity. We can envision a tool that visualizes these traces in a user-friendly manner, to help the user debug the grammar. We can also build a grammar debugger for Iguana, to provide a familiar IDE-like debugging experience. Since the GLL runtime has a one-to-one relationship with the grammar, i.e., the parser is at one grammar position at each time during parsing, building such a debugger is much simpler for GLL than for bottom-up general parsers such as GLR.

Parsing the C family of programming languages

Parsing the C family of programming languages (C, C++, and Objective C) is considered a hard task. In Chapter 3 we discussed two difficult aspects of parsing the C family of programming languages, namely the `typedef` ambiguity and preprocessors (in the context of C#). In the future, we plan to investigate how Iguana can be used to parse the C family of programming languages, and provide an extensive evaluation by parsing a large corpus of input files, such as the source code of the Linux kernel.

Tooling for Iguana

We have already developed an IntelliJ IDEA plugin for defining grammars using the Iguana syntax. The plugin provides common IDE features such as syntax highlighting, auto completion, and outline view. We plan to improve this plugin by providing more features, and integrate Iguana into the Rascal meta-programming language as the default parsing library.

8.3 Concluding Remarks

In the course of our work that led to this thesis, we studied many different parsing algorithms. In this concluding remarks section we share some of the things we learned.

Similarity of parsing algorithms

Perhaps the most important observation, as best described by Grune and Jacobs, was that *"the more parsing algorithms one studies the more they seem similar, and there seems to be great opportunity for unification. Basically almost all parsing is done by top-down search with left-recursion protection; this is true even for traditional bottom-up techniques like LR(1), where the top-down search is built into the LR(1) parse tables."* [33]. Although theoretically, there is a great chance for unification, we believe that the differences on how the top-down search and the guard on left recursion are implemented will have a great impact on how a parsing algorithm can be used in practice.

The approaches to deal with left recursion, from a high-level point of view, are basically all the same. The left recursion call should be terminated as soon as possible, and only be continued if a non-left-recursive alternative yields a result. This is even true for the classic grammar rewriting technique to eliminate left-recursion where the left recursion is replaced with right recursion, guarded by non-left-recursive alternatives. The LR automata also simulates such a process by allowing left-recursive nonterminal transitions only when other non-left-recursive alternatives reduce on top of the stack.

Our biggest surprise was the discovery of the relationship between GLL and Johnson's CPS recognizers, which turned out to be basically the same algorithm when dealing with left recursion. The handling of left-recursion in GLL and Johnson's CPS recognizers is dynamic and happens at the parser runtime, and the parser runs over the same original grammar. This makes GLL and Johnson's CPS recognizers runtime easier to understand compared to the runtime of a parser based on LR automata. We believe the more intuitive runtime model and flexibility of GLL and Johnson's recognizers makes them the general parsing algorithm of choice.

General parsing in practice

In Chapter 1 we discussed the spectrum of parsing algorithms based on their usage. Most compiler front-ends have hand-written recursive-descent parsers or LALR parsers generated by Yacc. General parsing algorithms have been mainly used in language engineering tools such as ASF+SDF, Rascal and Spoofax, where expressiveness and ease of use are important.

We suspect two factors have been in play that general parsing algorithms have not become mainstream. First, they are mostly intended to be used as part of a language workbench (ASF+SDF, Rascal and Spoofax), and many of their features are not available as a standalone library. Moreover, general parsers have gained a bad reputation for being slow. We believe Iguana addresses both of these issues. It is available as a standalone library, and as we have shown in Chapter 6, Iguana has good performance and memory footprint. There is still a long way to go to make Iguana mature to be used in practice, but we can predict that it can become a viable general parsing tool and compete with mature tools such as ANTLR 4.

The future of parsing

When discussing the future of parsing we should distinguish between the theoretical and practical aspects. From the theoretical perspective, in the last two decades we observed various variants and improvements to the GLR parsing algorithm, which also led to the development of GLL. We expect the same pace of theoretical contribution, in the form of extensions or improvements to the current parsing algorithms. From a practical perspective, we expect that most parsers for the front-end of compilers continue to be written by hand in the form of recursive-descent parsers, as it always pays off in terms of performance and producing good error messages to do the initial investment of writing the parser by hand.

For other applications of parsing such as designing domain-specific languages and reverse engineering, tools based on general parsing algorithms can become more popular. Besides improving performance, making disambiguation easier for the end user can increase the adaptation of such tools. Ambiguity is a difficult topic, and any effort in reporting ambiguities in a more understandable manner to the user and providing suggestions for disambiguation can be a good step towards this goal.

Bibliography

[1] A. Aasa. Precedences in Specifications and Implementations of Programming Languages. In *Selected Papers of the Symposium on Programming Language Implementation and Logic Programming*, PLILP '91, pages 3–26. Elsevier, 1995. (cited on pages 116, 122, and 148)

[2] M. D. Adams. Principled Parsing for Indentation-sensitive Languages: Revisiting Landin's Offside Rule. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 511–522. ACM, 2013. (cited on page 94)

[3] A. Afroozeh and A. Izmaylova. Faster, Practical GLL Parsing. In *Proceedings of the 24th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS)*, CC '15, pages 89–108. Springer, 2015. (cited on pages 13, 78, 79, 82, 83, 139, and 152)

[4] A. Afroozeh and A. Izmaylova. One Parser to Rule Them All. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! '15, pages 151–170. ACM, 2015. (cited on pages 155, 171, and 175)

[5] A. Afroozeh and A. Izmaylova. Operator Precedence for Data-dependent Grammars. In *Proceedings of the ACM SIGPLAN Symposium/Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 13–24. ACM, 2016. (cited on page 164)

[6] A. Afroozeh, M. van den Brand, A. Johnstone, E. Scott, and J. J. Vinju. Safe Specification of Operator Precedence Rules. In *Proceedings of the 6th International Conference on Software Language Engineering*, SLE '13, pages 137–156. Springer, 2013. (cited on pages 52 and 157)

[7] A. V. Aho, S. C. Johnson, and J. D. Ullman. Deterministic Parsing of Ambiguous Grammars. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN*

Symposium on Principles of Programming Languages, POPL '73, pages 1–21. ACM, 1973. (cited on pages 53, 61, 98, 120, and 123)

[8] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006. (cited on pages 1, 10, 12, 13, and 98)

[9] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., 1972. (cited on pages 5 and 10)

[10] J. Aycock and N. Horspool. Faster Generalized LR Parsing. In *Proceedings of the 8th International Conference on Compiler Construction, Held as Part of the European Joint Conferences on the Theory and Practice of Software (ETAPS)*, CC'99, pages 32–46. Springer, 1999. (cited on pages 9 and 52)

[11] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS®: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 625–634. IEEE Computer Society, 2004. (cited on pages 6 and 98)

[12] S. Billot and B. Lang. The Structure of Shared Forests in Ambiguous Parsing. In *Proceedings of the 27th Annual Meeting on Association for Computational Linguistics*, ACL '89, pages 143–151. Association for Computational Linguistics, 1989. (cited on page 7)

[13] M. Bravenboer, E. Tanter, and E. Visser. Declarative, Formal, and Extensible Syntax Definition for aspectJ. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 209–228. ACM, 2006. (cited on page 47)

[14] K. Clarke. The Top-down Parsing of Expressions. Technical report, Dept. of Computer Science and Statistics, Queen Mary College, 1986. (cited on pages 71, 122, 147, and 201)

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. (cited on page 9)

[16] N. A. Danielsson and U. Norell. Parsing Mixfix Operators. In *Proceedings of the 20th International Conference on Implementation and Application of Functional Languages*, IFL'08, pages 80–99. Springer, 2011. (cited on page 147)

[17] F. L. DeRemer. *Practical Translators for LR(k) Languages*. PhD thesis, Massachusetts Institute of Technology, 1969. (cited on pages 34, 56, and 120)

[18] J. Earley. An Efficient Context-free Parsing Algorithm. *Commun. ACM*, 13(2):94–102, Feb. 1970. (cited on pages 3, 7, 56, 93, 98, 121, 174, and 175)

[19] G. Economopoulos, P. Klint, and J. Vinju. Faster Scannerless GLR Parsing. In *Proceedings of the 18th International Conference on Compiler Construction, Held As Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, CC '09, pages 126–141. Springer, 2009. (cited on page 53)

[20] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. Layout-Sensitive Generalized Parsing. In *Proceedings of the 5th International Conference on Software Language Engineering*, SLE '12, pages 244–263. Springer, 2012. (cited on page 94)

[21] E. L. Favero. The Simple and Powerful yfx Operator Precedence Parser. *Softw. Pract. Exper.*, 37(14):1451–1474, Nov. 2007. (cited on page 147)

[22] K. Fisher and R. Gruber. PADS: A Domain-specific Language for Processing Ad Hoc Data. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 295–304. ACM, 2005. (cited on page 63)

[23] R. W. Floyd. Syntactic Analysis and Operator Precedence. *J. ACM*, 10(3):316–333, July 1963. (cited on pages 13, 120, and 147)

[24] B. Ford. Packrat Parsing:: Simple, Powerful, Lazy, Linear Time (Functional Pearl). In *Proceedings of the 7th ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36–47. ACM, 2002. (cited on page 200)

[25] B. Ford. Parsing Expression Grammars: A Recognition-based Syntactic Foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122. ACM, 2004. (cited on pages 5, 53, 147, and 200)

[26] R. A. Frost, R. Hafiz, and P. Callaghan. Parser Combinators for Ambiguous Left-recursive Grammars. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages*, PADL'08, pages 167–181. Springer, 2008. (cited on page 198)

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995. (cited on page 164)

[28] S. E. Ganz, D. P. Friedman, and M. Wand. Trampolined Style. In *Proceedings of the 4th ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 18–27. ACM, 1999. (cited on page 14)

[29] P. Gazzillo and R. Grimm. SuperC: Parsing All of C by Taming the Preprocessor. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 323–334. ACM, 2012. (cited on page 95)

[30] J. Gibbons and N. Wu. Folding Domain-specific Languages: Deep and Shallow Embeddings (Functional Pearl). In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 339–347. ACM, 2014. (cited on page 208)

[31] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 1st edition, 1996. (cited on page 34)

[32] J. Gosling, B. Joy, G. L. Steele, Jr., G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013. (cited on pages 51, 87, 139, 163, 175, and 197)

[33] D. Grune and C. J. Jacobs. *Parsing Techniques: A Practical Guide*. Springer Publishing Company, Incorporated, 2nd edition, 2010. (cited on pages 7, 10, 13, and 209)

[34] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The Syntax Definition Formalism SDF–Reference Manual–. *SIGPLAN Not.*, 24(11):43–75, Nov. 1989. (cited on pages 3, 4, 20, 56, 59, 67, and 152)

[35] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2006. (cited on page 156)

[36] G. Hutton. Higher-order Functions for Parsing. *Journal of Functional Programming*, 2(3):323–343, July 1992. (cited on pages 25, 94, and 174)

[37] G. Hutton and E. Meijer. Monadic Parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, July 1998. (cited on pages 25, 94, and 174)

[38] A. Izmaylova, A. Afroozeh, and T. v. d. Storm. Practical, General Parser Combinators. In *Proceedings of the ACM SIGPLAN Symposium/Workshop on Partial Evaluation and Program Manipulation*, PEPM '16, pages 1–12. ACM, 2016. (cited on page 206)

[39] T. Jim and Y. Mandelbaum. Efficient Earley Parsing with Regular Right-hand Sides. 253(7):135 – 148, 2010. LDTA'09. (cited on page 94)

[40] T. Jim and Y. Mandelbaum. A New Method for Dependent Parsing. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, ESOP '11, pages 378–397. Springer, 2011. (cited on page 146)

[41] T. Jim, Y. Mandelbaum, and D. Walker. Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 417–430. ACM, 2010. (cited on pages 6, 21, 56, 59, 63, 64, 65, 91, 93, 122, 130, 131, 152, 154, and 175)

[42] M. Johnson. The Computational Complexity of GLR Parsing. In *Generalized LR Parsing*, pages 35–42. Springer US, 1991. (cited on pages 7, 36, 52, 78, 185, and 192)

[43] M. Johnson. Memoization in Top-down Parsing. *Comput. Linguist.*, 21(3):405–417, Sept. 1995. (cited on pages 6, 10, 25, 175, 182, and 203)

[44] S. C. Johnson. Yacc: Yet Another Compiler-Compiler. AT&T Bell Laboratories, `http://dinosaur.compilertools.net/yacc/`. (cited on pages 1, 34, 53, 56, and 120)

[45] A. Johnstone and E. Scott. Modelling GLL Parser Implementations. In *Proceedings of the 3rd International Conference on Software Language Engineering*, SLE'10, pages 42–61. Springer, 2011. (cited on pages 45 and 79)

[46] A. Johnstone, E. Scott, and G. Economopoulos. Generalised parsing: Some costs. In *Proceedings of the 13th International Conference on Compiler Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, CC '04, pages 89–103. Springer, 2004. (cited on pages 52 and 121)

[47] A. Johnstone, E. Scott, and M. van den Brand. Modular Grammar Specification. *Sci. Comput. Prog.*, 87:23–43, 2014. (cited on pages 67 and 146)

[48] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 805–824. ACM, 2011. (cited on page 95)

[49] L. C. Kats, E. Visser, and G. Wachsmuth. Pure and Declarative Syntax Definition: Paradise Lost and Regained. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 918–932. ACM, 2010. (cited on pages 4, 20, 59, 121, and 152)

[50] P. Klint, R. Lämmel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380, July 2005. (cited on pages 3 and 98)

[51] P. Klint, T. v. d. Storm, and J. Vinju. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 9th IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 168–177. IEEE Computer Society, 2009. (cited on pages 87, 159, and 171)

[52] P. Klint, T. van der Storm, and J. J. Vinju. EASY Meta-Programming with Rascal. Leveraging the Extract-Analyze-SYnthesize Paradigm for Meta-Programming. In *Proceedings of the 3rd International Summer School on*

Generative and Transformational Techniques in Software Engineering, GTTSE '09. Springer, 2010. (cited on page 113)

[53] P. Klint and E. Visser. Using Filters for the Disambiguation of Context-free Grammars. In *Proceedings of the ASMICS Workshop on Parsing Theory*, pages 1–20. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano, 1994. (cited on pages 101 and 122)

[54] D. E. Knuth. On the Translation of Languages from Left to Right. *Information and control*, 8(6):607–639, 1965. (cited on pages 1, 7, 34, 56, and 120)

[55] P. J. Landin. The Next 700 Programming Languages. *Commun. ACM*, 9(3):157–166, Mar. 1966. (cited on page 61)

[56] B. Lang. Deterministic Techniques for Efficient Non-Deterministic Parsers. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 255–269. Springer, 1974. (cited on page 7)

[57] D. Leijen. Parsec, A Fast Combinator Parser. Technical Report 35, Department of Computer Science, University of Utrecht (RUU), 2001. (cited on pages 25 and 198)

[58] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.02. Technical report, Inria, 2014. (cited on pages 4, 51, 120, and 124)

[59] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed Translations. *J. Comput. Syst. Sci.*, 9(3):279–307, 1974. (cited on page 93)

[60] S. Marlow. Haskell 2010 language report, 2010. (cited on pages 62 and 90)

[61] S. McPeak and G. C. Necula. Elkhound: A Fast, Practical GLR Parser Generator. In *Proceedings of the 13th International Conference on Compiler Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, CC '04, pages 73–88. Springer, 2004. (cited on pages 6, 34, 52, 98, 146, and 174)

[62] A. Melnikov. Collected Extensions to IMAP4 ABNF, April 2006. `http://tools.ietf.org/html/rfc4466`. (cited on page 56)

[63] Microsoft Corporation. C# Language Specification Version 5.0, June 2013. `http://www.microsoft.com/en-us/download/details.aspx?id=7029`. (cited on pages 51, 63, 65, and 88)

[64] A. Moors, F. Piessens, and M. Odersky. Parser Combinators in Scala. Technical report, Katholieke Universiteit Leuven, 2008. (cited on page 198)

[65] M.-J. Nederhof. A New Top-down Parsing Algorithm for Left-recursive DCGs. In *Proceedings of the 5th International Symposium on Programming Language Implementation and Logic Programming*, PLILP'93, pages 108–122. Springer, 1993. (cited on page 201)

[66] P. Norvig. Techniques for Automatic Memoization with Applications to Context-free Parsing. *Computational Linguistics*, 17(1):91–98, 1991. (cited on pages 9 and 181)

[67] R. Nozohoor-Farshi. GLR Parsing for ϵ-Grammers. In M. Tomita, editor, *Generalized LR Parsing*, pages 61–75. Springer US, 1991. (cited on pages 7 and 52)

[68] M. Odersky. *The Scala Language Specification, version 2.9.* Programming Methods Laboratory, EPFL, 2014. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`. (cited on page 23)

[69] T. Parr and K. Fisher. LL(*): The Foundation of the ANTLR Parser Generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436. ACM, 2011. (cited on pages 30, 32, and 163)

[70] T. Parr, S. Harwell, and K. Fisher. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 579–598. ACM, 2014. (cited on pages 9, 20, 30, 32, 53, 71, 146, 147, 163, and 201)

[71] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992. (cited on pages 7, 22, and 98)

[72] T. Ridge. Simple, Efficient, Sound and Complete Combinator Parsing for All Context-Free Grammars, Using an Oracle. In *Proceedings of the 7th International Conference on Software Language Engineering*, SLE '14, pages 261–281. Springer, 2014. (cited on page 175)

[73] D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) Parsing of Programming Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '89, pages 170–178. ACM, 1989. (cited on pages 47, 60, 67, 146, and 156)

[74] A. C. Schwerdfeger and E. R. Van Wyk. Verifiable Composition of Deterministic Grammars. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 199–210. ACM, 2009. (cited on page 3)

[75] E. Scott and A. Johnstone. Generalized Bottom Up Parsers With Reduced Stack Activity. *Comput. J.*, 48(5):565–587, Sept. 2005. (cited on page 9)

[76] E. Scott and A. Johnstone. Right Nulled GLR Parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, July 2006. (cited on pages 7 and 52)

[77] E. Scott and A. Johnstone. Recognition is not Parsing - SPPF-style Parsing from Cubic Recognisers. *Sci. Comput. Program.*, 75(1-2):55–70, Jan. 2010. (cited on page 159)

[78] E. Scott and A. Johnstone. GLL Parse-tree Generation. *Science of Computer Programming*, 78(10):1828–1844, 2013. (cited on pages 3, 4, 6, 9, 35, 36, 41, 45, 49, 56, 58, 59, 65, 78, 79, 82, 83, 91, 98, 113, 121, 159, 160, 174, 175, 202, and 203)

[79] E. Scott and A. Johnstone. GLL syntax analysers for EBNF grammars. *Science of Computer Programming*, 166:120–145, 6 2018. (cited on page 171)

[80] E. Scott, A. Johnstone, and R. Economopoulos. BRNGLR: A Cubic Tomita-style GLR Parsing Algorithm. *Acta informatica*, 44(6):427–461, 2007. (cited on pages 9, 59, 121, 175, 185, and 203)

[81] D. Spiewak. Generalized Parser Combinators. `http://www.cs.uwm.edu/~dspiewak/papers/generalized-parser-combinators.pdf`, March 2010. (cited on page 202)

[82] M. Thorup. Controlled grammatic ambiguity. *ACM Trans. Program. Lang. Syst.*, 16(3):1024–1050, May 1994. (cited on page 116)

[83] M. Thorup. Disambiguating grammars by exclusion of sub-parse trees. *Acta Informatica*, 33(5):511–522, 1996. (cited on page 116)

[84] M. Thorup. Disambiguating Grammars by Exclusion of Sub-Parse Trees. *Acta Inf.*, 33(6):511–522, 1996. (cited on pages 122 and 148)

[85] M. Tomita. *Efficient Parsing for Natural Language*. Kluwer Academic Publishers, 1985. (cited on pages 3, 4, 7, 34, 52, 56, 59, 78, 81, 98, 121, 174, and 185)

[86] L. Tratt. Direct Left-recursive Parsing Expression Grammars. Technical Report EIS-10-01, School of Engineering and Information Sciences, Middlesex University, Oct. 2010. (cited on page 200)

[87] M. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw., Pract. Exper.*, 30(3):259–291, 2000. (cited on page 159)

[88] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. *Electr. Notes Theor. Comput. Sci.*, 44(2):3–8, 2001. (cited on page 6)

[89] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Olivier. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Trans. Program. Lang. Syst.*, 24(4):334–368, July 2002. (cited on pages 7, 34, 52, 174, and 188)

[90] M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *Proceedings of the 11th International Conference on Compiler Construction, Held as Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)*, CC '02, pages 143–158. Springer, 2002. (cited on pages 4, 20, 48, 53, 56, 60, 67, and 156)

[91] E. R. Van Wyk and A. C. Schwerdfeger. Context-aware Scanning for Parsing Extensible Languages. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07, pages 63–72. ACM, 2007. (cited on pages 31, 60, 66, 67, 142, and 156)

[92] E. Visser. From context-free grammars with priorities to character class grammars. In A. van Deursen, M. Brune, and J. Heering, editors, *Dat Is Dus Heel Interessant, Liber Amicorum dedicated to Paul Klint*, pages 217–230. CWI, 1997. (cited on page 116)

[93] E. Visser. From Context-Free Grammars With Priorities to Character Class Grammars. Technical report, University of Amsterdam, 1997. (cited on page 148)

[94] E. Visser. Scannerless Generalized-LR Parsing. Technical report, University of Amsterdam, 1997. (cited on pages 22, 47, 99, 107, 122, and 156)

[95] E. Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, 1997. (cited on pages 20, 22, 53, 56, 60, 67, 94, 122, and 123)

[96] P. Wadler. How to Replace Failure by a List of Successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *Proceedings of a Conference on Functional Programming Languages and Computer Architecture*, FPCA '85, pages 113–128, 1985. (cited on page 15)

[97] P. Wadler. Comprehending Monads. In *Proceedings of the ACM Conference on LISP and Functional Programming*, LFP '90, pages 61–78. ACM, 1990. (cited on page 180)

[98] A. Warth, J. R. Douglass, and T. Millstein. Packrat Parsers Can Support Left Recursion. In *Proceedings of the ACM SIGPLAN Symposium/Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '08, pages 103–110. ACM, 2008. (cited on page 200)

[99] D. A. Watt. Rule splitting and attribute-directed parsing. In *Semantics-Directed Compiler Generation*, pages 363–392, 1980. (cited on page 93)

[100] W. A. Woods. Transition Network Grammars for Natural Language Analysis. *Commun. ACM*, 13(10):591–606, 1970. (cited on pages 58, 78, and 94)

Summary

This thesis presents the design and implementation of practical general top-down parsers. Top-down parsing, usually in the form of recursive-descent parsing, is efficient, easy to understand and is widely used to manually write parsers for real programming languages. However, due to the difficulty of dealing with left recursion, top-down parsers always suffered from expressiveness problems compared to their bottom-up counterparts.

In many applications of parsing, such as designing domain-specific languages, where factors such as expressiveness and ease of use are important, general parsing algorithms are considered. General parsing algorithms support all context-free grammars and can present all the ambiguities in a compact parse forest format. Most of the research on developing general parsers has been devoted to bottom-up parsers, most notably on the Generalized LR (GLR) parsing algorithm. GLR is a mature parsing algorithm, however, due to its bottom-up nature, it is hard to understand and modify. There have been a number of efforts to increase the expressive power of recursive-descent parsing by using various backtracking techniques, but a solution to the problem of left recursion proved to be difficult.

The Generalized LL (GLL) parsing algorithm, which has been developed by Scott and Johnstone in 2010, provided a viable alternative to GLR. GLL parsers support all context-free grammars (including left-recursive ones), are recursive-descent like, and their runtime has a one-to-one relationship with the grammar. This makes GLL attractive for parsing programming languages. The other very important, and less widely known work, on generalizing recursive-descent parsing is Mark Johnson's work on Continuation-Passing Style (CPS) recognizers. Our work in this thesis has been inspired by both GLL and Johnson's CPS recognizers, and as we discovered, these two algorithms are so similar in the way they handle left recursion that we can consider them basically the same algorithm.

GLL is a relatively new general parsing algorithm that has not been yet widely used in practice. Working with Johnson's recognizers led us to propose a more efficient Graph Structured Stack (GSS) for GLL, which resembles traditional function memoization. Our modified version of GLL parsing is not only faster, but also easier

to understand, and laid out the foundation for our work on data-dependent grammars.

We found data-dependent grammars to be an excellent abstract intermediate language for implementing various disambiguation constructs without knowledge of a specific parsing technology. As data-dependent grammars are rather low-level, we have also provided high-level disambiguation constructs, e.g., for operator precedence and indentation-sensitivity, that desugar to data-dependent grammars. We discuss the application of our technique to resolve various ambiguities, for example, the ones found in indentation-sensitive languages such as Haskell and Python, conditional directives in C#, `typedef` ambiguity in C, and intricate cases of operator precedence in OCaml.

A significant part of this thesis is dedicated to the semantics and implementation of disambiguation rules for operator precedence. Disambiguation rules for operator precedence are among the most important disambiguation constructs, and are perhaps the most difficult to get right. We introduce a derivation-based semantics for operator precedence disambiguation that is independent of the underlying parsing technique, and is safe, i.e., does not remove sentences from the language when there is no ambiguity. In addition, our operator precedence semantics can deal with so-called *deep* operator precedence cases that often occur in functional programming languages such as OCaml. Our safe specification of operator precedence rules is implemented by an automatic grammar rewriting process that preserves the shape of the parse trees, conforming to the original ambiguous grammar. This rewriting, however, could lead to very large grammars, which motivated us to propose an alternative implementation that is based on data-dependent grammars and resolves the operator precedence ambiguities at runtime.

Our last contribution is general parser combinators that can deal with all context-free grammars and produce a binarized Shared Packed Parse Forest (SPPF) in cubic time and space. Our general parser combinators have the flexibility and expressiveness of traditional parser combinators, and the performance guarantee of general parsing algorithms. Our general parser combinators are based on Johnson's CPS recognizers. We extend Johnson's CPS recognizers to achieve recognition in cubic time, and extend the resulting cubic CPS recognizers to parsers that construct a binarized SPPF. We used our cubic CPS parsers as the basis for Meerkat, a general parser combinator library in Scala.

In the course of this thesis, we have developed Iguana, our GLL-based data-dependent parsing framework. Iguana's extensible data-dependent grammar API allows the user to easily add new disambiguation constructs or modify existing ones. We have used Iguana to parse various real programming languages, such as Java, C#, Haskell, and a significant subset of OCaml. The results of our performance evaluation show that Iguana is practical for parsing real programming languages, and in terms of performance, is comparable to a mature parsing tool such as ANTLR.

Samenvatting

Dit proefschrift presenteert het ontwerp en de implementatie van praktische, algemene top-down parsers. Top-down parsers, meestal in de vorm van recursive-descent parsers, zijn efficiënt, gemakkelijk te begrijpen en worden veel gebruikt om handmatig parsers voor echte programmeertalen te schrijven. Vanwege de moeilijkheid om links-recursie te behandelen, leiden top-down parsers altijd aan expressiviteitsproblemen in vergelijking met hun bottom-up tegenhangers.

In veel toepassingen van parseren, zoals het ontwerpen van domeinspecifieke talen, waarbij factoren zoals expressiviteit en gebruiksgemak belangrijk zijn, worden algemene parseringsalgoritmen gebruikt. Algemene parseringsalgoritmen ondersteunen alle context-vrije grammatica's en kunnen alle ambiguïteiten presenteren in de vorm van een compact bos van bomen. Het meeste onderzoek naar het ontwikkelen van algemene parsers is gericht op bottom-up parsers, in het bijzonder op het gegeneraliseerde LR-algoritme (GLR). GLR is een volwassen parseringsalgoritme, maar vanwege zijn bottom-up karakter, is het moeilijk te begrijpen en te modificeren. Er zijn in het verleden een aantal pogingen ondernomen om de expressieve kracht van recursive-descent parseren te vergroten door verschillende backtrackingtechnieken te gebruiken, maar een oplossing voor het probleem van links-recursie blijkt moeilijk te zijn.

Het gegeneraliseerde LL-algoritme (GLL), dat in 2010 door Scott en Johnstone is ontwikkeld, biedt een levensvatbaar alternatief voor GLR. GLL parsers ondersteunen alle context-vrije grammatica's (inclusief links-recursieve), ze lijken op recursive descent en hun runtime heeft een één-op-één relatie met de grammatica. Dit maakt GLL aantrekkelijk voor het parseren van programmeertalen. Ander, zeer belangrijk maar minder bekend werk, over het generaliseren van recursive-descent parsers, is het werk van Mark Johnson over CPS-herkenners gebaseerd op continuaties (Continuation Passing Style). Ons werk in dit proefschrift is geïnspireerd door zowel GLL als de CPS-herkenners van Johnson, en, zoals we hebben ontdekt, zijn deze twee algoritmen zo vergelijkbaar in de manier waarop ze links-recursie verwerken dat we ze in principe als hetzelfde algoritme kunnen beschouwen.

GLL is een relatief nieuw algemeen parseringsalgoritme dat in de praktijk nog niet veel is gebruikt. Het werken met de herkenners van Johnson heeft ons ertoe gebracht

om een efficiëntere Graph Structured Stack (GSS) voor GLL voor te stellen, die lijkt op het traditionele memoriseren van functieresultaten. Onze gewijzigde versie van GLL parseren is niet alleen sneller, maar ook gemakkelijker te begrijpen, en heeft de basis gelegd voor ons werk op het gebied van data-afhankelijk grammatica's.

We hebben ontdekt dat data-afhankelijk grammatica's een uitstekende abstracte tussentaal vormen voor het implementeren van verschillende disambigueringsconstructies zonder kennis van een specifieke parseringstechnologie. Omdat data-afhankelijke grammatica's van een nogal laag-niveau zijn, stellen we ook hoog-niveau disambigueringsconstructies voor, bijvoorbeeld voor prioriteitsregels van operatoren en indentatiegevoeligheid, die naar data-afhankelijke grammatica's kunnen worden vertaald. We bespreken de toepassing van onze techniek om verschillende ambiguiteiten op te lossen, bijvoorbeeld die gevonden in indentatiegevoelige talen zoals Haskell en Python, conditionele directieven in C#, de bekende `typedef` ambiguiteit in C, en ingewikkelde gevallen van operatorprioriteiten in OCaml.

Een belangrijk deel van dit proefschrift is gewijd aan de semantiek en implementatie van disambigueringsconstructies voor prioriteiten van operatoren. Disambigueringsconstructies voor operatorprioriteiten behoren tot de belangrijkste disambigueringsconstructies en zijn misschien het moeilijkste om correct te krijgen. We introduceren een afleidinggebaseerde semantiek voor operatorprioriteiten die onafhankelijk is van de onderliggende parseringstechniek, en veilig is, d.w.z. geen zinnen uit de taal verwijdert als er geen ambiguïteit is. Bovendien kan de semantiek voor operatorprioriteiten omgaan met zogenaamde *diepe* gevallen van operatorprioriteit die vaak voorkomen in functionele programmeertalen zoals OCaml. Onze veilige specificatie van regels voor operatorprioriteiten wordt geïmplementeerd door een automatisch grammaticaherschrijfproces dat de vorm van de afleidingsbomen behoudt, conform de oorspronkelijke ambigue grammatica. Deze herschrijving kan echter leiden tot zeer grote grammatica's, wat ons motiveerde om een alternatieve implementatie te ontwikkelen die is gebaseerd op data-afhankelijke grammatica's en die ambigue operatorprioriteiten kan oplossen tijdens runtime.

Onze laatste bijdrage bestaat uit algemene parsercombinators die alle context-vrije grammatica's kunnen verwerken en een binair Shared Packed Parse Forest (SPPF) in kubische tijd en ruimte kunnen produceren. Onze algemene parsercombinators hebben de flexibiliteit en expressiviteit van traditionele parsercombinators en de runtimegarantie van algemene parseringsalgoritmen. Onze algemene parsercombinators zijn gebaseerd op de CPS-herkenners van Johnson. We breiden Johnson's CPS-herkenners uit om herkenning in kubische tijd te bereiken en breiden de resulterende kubische CPS-herkenners uit naar parsers die een binair SPPF construeren. We hebben onze kubische CPS-parsers gebruikt als basis voor Meerkat, een algemene parsercombinatorbibliotheek voor Scala.

In de loop van dit proefschrift hebben we, tenslotte, Iguana ontwikkeld, een op GLL gebaseerde data-afhankelijk parseringsraamwerk. Met Iguana's API (Application Programmers Interface) voor uitbreidbare data-afhankelijke grammatica's kan de gebruiker gemakkelijk nieuwe disambigueringsconstructies toevoegen of bestaande constructies wijzigen. We hebben Iguana gebruikt om verschillende echte programmeertalen te parseren, zoals Java, C#, Haskell en een groot deel van OCaml. Onze prestatieanalyses tonen aan dat Iguana praktisch bruikbaar is voor het parseren van echte programmeertalen en qua prestaties kan concurreren met een volwassen parseringstool zoals ANTLR.

Titles in the IPA Dissertation Series since 2016

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices*. Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization*. Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems*. Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs*. Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement*. Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security*. Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning*. Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections*. Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors*. Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT).* Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data.* Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities.* Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too.* Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod.* Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages.* Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01

S. Darabi. *Verification of Program Parallelization.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-02

J.R. Salamanca Tellez. *Coequations and Eilenberg-type Correspondences.* Faculty of Science, Mathematics and Computer Science, RU. 2018-03

P. Fiterău-Broştean. *Active Model Learning for the Analysis of Network Protocols.* Faculty of Science, Mathematics and Computer Science, RU. 2018-04

D. Zhang. *From Concurrent State Machines to Reliable Multi-threaded Java Code.* Faculty of Mathematics and Computer Science, TU/e. 2018-05

H. Basold. *Mixed Inductive-Coinductive Reasoning Types, Programs and Logic.* Faculty of Science, Mathematics and Computer Science, RU. 2018-06

A. Lele. *Response Modeling: Model Refinements for Timing Analysis of Runtime Scheduling in Real-time Streaming Systems.* Faculty of Mathematics and Computer Science, TU/e. 2018-07

N. Bezirgiannis. *Abstract Behavioral Specification: unifying modeling and programming.* Faculty of Mathematics and Natural Sciences, UL. 2018-08

M.P. Konzack. *Trajectory Analysis: Bridging Algorithms and Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2018-09

E.J.J. Ruijters. *Zen and the art of railway maintenance: Analysis and optimization of maintenance via fault trees and statistical model checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-10

F. Yang. *A Theory of Executability: with a Focus on the Expressivity of Process Calculi.* Faculty of Mathematics and Computer Science, TU/e. 2018-11

L. Swartjes. *Model-based design of baggage handling systems.* Faculty of Mechanical Engineering, TU/e. 2018-12

T.A.E. Ophelders. *Continuous Similarity Measures for Curves and Surfaces.* Faculty of Mathematics and Computer Science, TU/e. 2018-13

M. Talebi. *Scalable Performance Analysis of Wireless Sensor Network.* Faculty of Mathematics and Computer Science, TU/e. 2018-14

R. Kumar. *Truth or Dare: Quantitative security analysis using attack trees.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-15

M.M. Beller. *An Empirical Evaluation of Feedback-Driven Software Development.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2018-16

M. Mehr. *Faster Algorithms for Geometric Clustering and Competitive Facility-Location Problems.* Faculty of Mathematics and Computer Science, TU/e. 2018-17

M. Alizadeh. *Auditing of User Behavior: Identification, Analysis and Understanding of Deviations.* Faculty of Mathematics and Computer Science, TU/e. 2018-18

P.A. Inostroza Valdera. *Structuring Languages as Object-Oriented Libraries.* Faculty of Science, UvA. 2018-19

M. Gerhold. *Choice and Chance - Model-Based Testing of Stochastic Behaviour.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-20

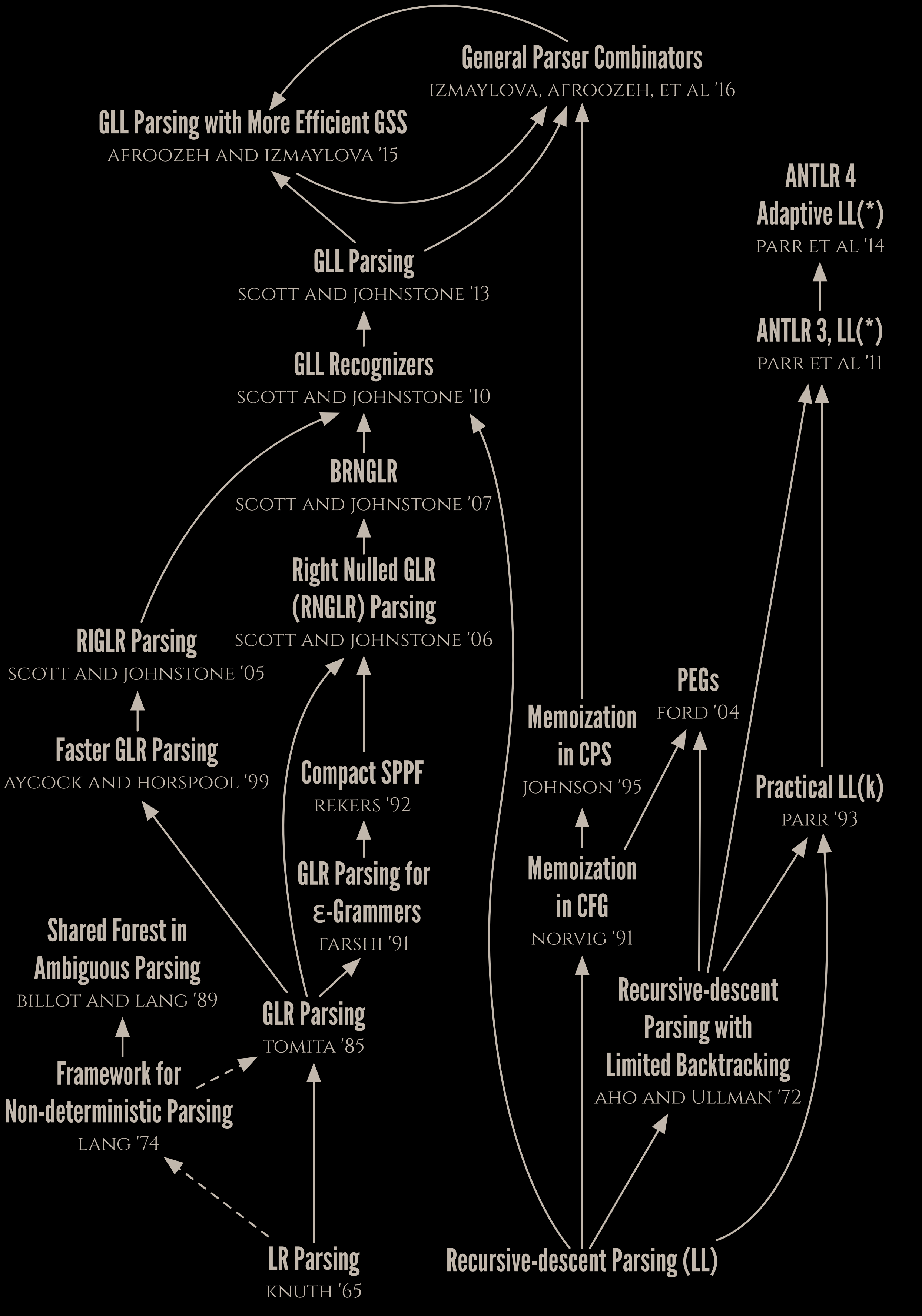
A. Serrano Mena. *Type Error Customization for Embedded Domain-Specific Languages.* Faculty of Science, UU. 2018-21

S.M.J. de Putter. *Verification of Concurrent Systems in a Model-Driven Engineering Workflow.* Faculty of Mathematics and Computer Science, TU/e. 2019-01

S.M. Thaler. *Automation for Information Security using Machine Learning.* Faculty of Mathematics and Computer Science, TU/e. 2019-02

Ö. Babur. *Model Analytics and Management.* Faculty of Mathematics and Computer Science, TU/e. 2019-03

A. Afroozeh and A. Izmaylova. *Practical General Top-down Parsers.* Faculty of Science, UvA. 2019-04



General Parser Combinators

IZMAYLOVA, AFROOZEH, ET AL '16

GLL Parsing with More Efficient GSS

AFROOZEH AND IZMAYLOVA '15

ANTLR 4

Adaptive LL(*)

PARR ET AL '14

GLL Parsing

SCOTT AND JOHNSTONE '13

ANTLR 3, LL(*)

PARR ET AL '11

GLL Recognizers

SCOTT AND JOHNSTONE '10

BRNGLR

SCOTT AND JOHNSTONE '07

Right Nulled GLR

(RNGLR) Parsing

SCOTT AND JOHNSTONE '06

RIGLR Parsing

SCOTT AND JOHNSTONE '05

PEGs

FORD '04

Memoization

in CPS

JOHNSON '95

Faster GLR Parsing

AYCOCK AND HORSPOOL '99

Compact SPPF

REKERS '92

Practical LL(k)

PARR '93

Memoization

in CFG

NORVIG '91

GLR Parsing for

ε-Grammers

FARSHI '91

Shared Forest in

Ambiguous Parsing

BILLOT AND LANG '89

GLR Parsing

TOMITA '85

Recursive-descent

Parsing with

Limited Backtracking

AHO AND ULLMAN '72

Framework for

Non-deterministic Parsing

LANG '74

LR Parsing

KNUTH '65

Recursive-descent Parsing (LL)