Comparing Bottom-up with Top-down Parsing Architectures for the Syntax Definition Formalism from a Disambiguation standpoint

Jurgen J. Vinju
NWO-I Centrum Wiskunde & Informatica
TU Eindhoven, The Netherlands

Abstract

Context-free general parsing and disambiguation algorithms are threaded throughout the research and engineering career of Eelco Visser. Both our Ph.D. theses featured the study of "disambiguation." Disambiguation is the declarative definition of choices among different parse trees, derived using the same context-free grammar, for the same input sentence.

This essay highlights the differences between syntactic disambiguation for context-free general parsing in a top-down architecture and a bottom-up architecture. The differences between top-down and bottom-up are mainly observed as practical aspects of the software architecture and software implementation. Eventually, the concept of data-dependent context-free grammar brings all engineering perspectives of disambiguation back into a conceptual (declarative) framework independent of the parsing architecture. The novelty in this essay is the juxtaposition of three general parsing architectures from a disambiguation point of view: SGLR, SGLL, and DDGLL. It also motivates design decisions in the parsing architectures for SDF{1,2} and Rascal with previously unpublished detail. The essay falls short of a literature review and a tool evaluation since it does not investigate the disambiguation methods of the many other parser generator tools that exist. The fact that only the implementation algorithms are different between the compared parsing architectures, while the syntax definition formalisms have practically the same formal semantics for historical reasons, nicely "isolates the variable" of interest.

We hope this essay lives up to the enormous enthusiasm, curiosity, and drive for perfection in syntax definition and parsing that Eelco always radiated. We dearly miss him.

2012 ACM Subject Classification Software and its engineering → Syntax

Keywords and phrases parser generation, context-free grammars, GLR, GLL, algorithms, disambiguation

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.12

Acknowledgements This essay would not have been possible without the past teamwork with Eelco Visser, Paul Klint, Jan Heering, Jeroen Scheerder, Mark van den Brand, Chris Verhoef, Alex Sellink, Pieter Olivier, Hayco de Jong, Georgios (Rob) Economopoulos, Martin Bravenboer, Tijs van der Storm, Joost Visser, Merijn de Jonge, Jørgen Iversen, Arnold Lankamp, Ali Afroozeh, Anastasia Izmaylova, Bas Basten, Martin Bravenboer, Adrian Johnstone and Elizabeth Scott on the topic of context-free general parsing and disambiguation and supporting technologies. However, any error or inconsistency in the following is all mine.

1 Introduction

This essay focuses on qualitative differences in the design and implementation of Syntax Definition, Parser Generation, and Disambiguation between two classes of Parsing algorithms: GLR and GLL. Disambiguation is a function of an entire Parsing Architecture with the goal of reducing the set of Parse Trees (the Parse Forest) to exactly one using declarative definitions. Extensions of the Syntax Definition Formalisms (languages for context-free grammars) allow expressing preferences between different Parse Trees as produced by the Grammar. It is helpful to see Disambiguation as orthogonal to Parsing, where the latter produces Parse Trees and the former removes them again. However, in actual Parsing Architectures, this distinction is virtually invisible due to efficiency considerations. In this essay, we emphasize declarative and correct parsing over efficiency.

1.1 History of Disambiguation with the SDF

A brief and selective history of disambiguating context-free grammars written in the "Syntax Definition Formalism" (SDF) is due first. We focus on the parsing architectures of the Syntax Definition Formalism (SDF) [18, 15, 19] and its later incarnations SDF2 [26, 45] (a part of the "new" ASF+SDF Meta-Environment [7, 12] and of StrategoXT [13]), and its further parallel offspring in Rascal [24, 25] and Spoofax [22] (SDF3 [4]). This history motivates the comparison of parsing architectures later and establishes their origins and dependencies for the sake of full disclosure. We are not comparing independently designed artifacts.

Already in the first SDF from the early 1980's its users wrote context-free grammars in a BNF-like format [18], plus regular extensions such as lists and optionals (a.k.a. "EBNF"), plus disambiguation declarations. The non-terminal notation of SDF was taken from the meta notation used in Paul Klint's PhD thesis [23]. These definitions were then used to generate parsers and other useful language tooling such as unparsers, syntax highlighters, structure editors, etc. SDF used general parsing algorithms from the very start. Initially, it was based on Jay Earley's algorithm [16], but SDF switched to Tomita's GLR parsing algorithm [37] quickly. Earley's and GLR would allow any kind of grammars —not just LL(1) or LR(1) or LALR, but *any*. This was a unique and stimulating feature for a parser generator, since putting together rules from different modules would also work (i.e. parsing would happen) and often it would do the right thing. SDF with GLR as its underlying execution mechanism offered the powers of modularity, language embeddings, and compositionality of grammars that were eminently useful and also deemed elegant.

However, ambiguity and its cure disambiguation are neither modular nor compositional. By this, we mean that two arbitrary unambiguous modules when composed can easily become ambiguous and that two arbitrary modules with disambiguation constructs without parsing errors could when composed, easily generate spurious parsing errors. Ambiguity of context-free grammars is generally undecidable (with and without disambiguation constructs) [36], and this conundrum is the main motivation for all our research into the "diagnostics and treatment" of ambiguity in the SDF world [43]. Ambiguity made the SDF sometimes hard to use, despite its elegance and compositionality, or perhaps because of it. On top of this, the non-determinism of SDF grammars (ambiguous or not) is also a source of inefficiency of Tomita's GLR. The same disambiguation constructs that were proposed would also have a positive effect on efficiency as well.

Rewinding, this history starts with the implementation of SDF which was documented in Jan Rekers' thesis on incremental general parser generation in 1992. SDF existed before this and was documented in the technical reports of the ESPRIT project "GIPE - Generating Interactive Programming Environments" and GIPE II in ESPRIT 2 ([18] not digitized). SDF and the initial implementations of SDF2 were implemented using ASF+SDF itself, which was implemented in "LeLisp" [15]. The scanner in SDF was non-deterministic as well: scanning would produce all possible tokens at a given input position, from which the non-deterministic parsers could choose. This was necessary to achieve the desired modularity and compositionality of real programming languages like PL/I, Pascal, and COBOL. Disambiguation was featured in SDF by the associativity and priority declarations between rules, to help declare the binding strength of unary, binary, and n-ary expression operators without having to factor a grammar and without introducing any helper non-terminals. Writing and maintaining SDF grammars was attractive because due to these two disambiguation constructs the number of rules needed to define the syntax of a language was kept close to the number of actual constructs in the language. The interactions between the scanner and parser were sometimes unpredictable due to complex feature interactions with language composition and rules like longest match and keyword reservation. Incremental parsing was important at that time for feasibility of running complex and lively updated IDEs on small and slow machines. Wilco Koorn and Jan Rekers extended the GLR algorithm for substring parsing [32] to that end, and also to improve error recovery

and auto-completion IDE features. This pushed the interaction between the parsing and the scanning algorithms to the limit.

Eelco's Ph.D. thesis on SDF2 in 1997 followed [45], and we completed an implementation of its parsing and disambiguation mechanisms (SGLR and PGEN) in C and ASF+SDF in 2000 together with Jeroen Scheerder and Mark van den Brand [7]. A main driving force at that time was the COBOL grammar by Chris Verhoef and Ralf Lämmel [27], as well as the "Island Grammars" by Leon Moonen [29] that kept pushing the boundaries of what was possible with declarative disambiguation. Peter Mosses and the Action Notation [9] (design and implementation) we considered important to satisfy as our "customer on-site." Eelco had been inspired by Solomon and Cormack [33] and introduced "scannerless parsing" to SDF by removing the entire scanner from the architecture, thereby introducing lexical ambiguity to the playground of SDF2. This design decision removed the aforementioned hard-to-predict interactions between a non-deterministic scanner and a non-deterministic parser.

Eelco's thesis contains the mitigations necessary to make scannerless parsing workable. These are two new disambiguation constructs: *follow restrictions* for longest match and *reject rules* for keyword reservation [39]. He also introduced a simplified representation of parse trees, with only three kinds of nodes: applications of grammar rules, unordered ambiguity clusters, and terminal characters. The grammar rules were represented in the abstract syntax of SDF2 as an algebraic data type in the ATerm format (Pieter Olivier, Hayco de Jong, Mark van den Brand, Paul Klint) [11, 10]. This algebraic format of constructors for parse trees, called AsFix2, was derived from earlier parse tree export formats (AsFix1) that were designed by Mark van den Brand to bootstrap ASF+SDF off of the LeLisp implementation [38] and as the intended exchange format between the various tools of the ASF+SDF Meta-Environment [7] that was under development at that time.

Since SDF2 was partly implemented in ASF+SDF itself, a bootstrap was required. Mark van den Brand and Pieter Olivier completed the LeLisp-independent ASF+SDF compiler in ASF+SDF in 2001, which included a compilation of the implementation of SDF2 and a re-implementation of the back-end table generator in C. My own thesis work included disambiguation for SDF2 [42]. That was in collaboration with Eelco, Jeroen Scheerder, and Mark van den Brand [39], on how to implement the filtering ideas in Eelco's thesis [26, 45]. The work with Rob Economopoulos was about integrating RNGLR [34] into SGLR in 2013 [17], which could simplify some of the filters but complicated others. Diagnosing ambiguity with Bas Basten was a parallel track [43, 5] (2011).

SDF2 as-is was used for many years by the ASF+SDF Meta-Environment, ELAN4 environment [12, 41], Action Environment [9] and StrategoXT [13] communities. In this essay we focus on the differences between the bottom-up parsing architecture of SDF2 as it was in the early 2010's and the Rascal top-down architecture in the same period.

SDF2's Scannerless Generalized LR parsing algorithm (SGLR) is by Eelco Visser [45]. What makes it different from its predecessors, namely Rekers' fixed version [31] of Tomita's GLR [37], is the semantics or implementation of disambiguation filters specific to scannerless parsing and specific to declarative definitions of operator precedence and associativity in expression grammars. SGLR is the parsing architecture made ready to implement SDF2, the syntax definition formalism from the thesis of Eelco Visser. The predecessor of SDF2, SDF had similar disambiguation constructs for operator precedence, but not for disambiguating lexical syntax. Hence Eelco named "SGLR": "Scannerless GLR", and the efficient and declarative disambiguation of lexical syntax is its core contribution.

Lexical ambiguity aside, what we never really solved (at that time) was the context-free ambiguity problem in general. Context-free general parsing produces multiple parse trees, sometimes, and it is hard to predict when. And, SGLR offers specific disambiguation constructs for specific kinds of ambiguity [5], but it can not solve arbitrary ambiguity. So, we experimented (from the very start)

with far more general disambiguation constructs, such as the "multiset filter" algorithm [26, 45]. The multiset filter lifts the strict partial order of priority rules to entire parse trees by considering them "sets of rules" and applying a strict partial order of sets of partially ordered elements on entire trees. The more advanced the filters became, the less declarative and the more heuristic they became. Also, new disambiguation filters tended to be hard to implement correctly (more on this later) or efficiently (they all became back-end tree filters). Every new disambiguation concept added to SGLR required almost the effort of a PhD thesis. At the end of the 2010's, we found a resting point, eventually, to be able to filter parse trees using general purposes tree manipulators, such as Stratego, ASF+SDF, and Rascal—a.k.a. semantics directed disambiguation [8].

The second bootstrap stage of Rascal in 2011 provided us with a choice again. We had the opportunity to start from scratch in terms of parsing architecture. Having wrestled with the GLR algorithm for decades, we decided to flip the perspective and go top-down to Scott and Johnstone's GLL [35]. The one and only motivation was the simplicity and elegance of the new architecture, promising also simple and elegant disambiguation filters. A scannerless version of GLL [35] with disambiguations based on all the filters of SDF2 was indeed produced for Rascal 0.4.x. After this experimenting with new filters became really easy, intuitive, and fast. It is the goal of the current essay to substantiate this story.

The PhD thesis of Izmaylova and Afroozeh contains the idea of mapping the semantics of disambiguation filters to (lexical) constraints in data-dependent context-free grammars [1]. The team of Jim, Mandelbaum, and Walker had shown with their Jakker parser generator that data-dependent grammars are an elegant formalism for expressing the unambiguous syntax of programming languages [20, 21]. Moreover, such data-dependent grammars with lexical constraints seem to effectively model almost every hack we have seen in hand-written top-down parsers as well. Even the offside rule in Haskell, which is described as a hack of introducing an extra token in the token stream in an error state, can be simulated using a DDCFG in Iguana [2], and also symbol tables such as used in the scanning of C programs are expressible in data-dependent context-free grammars [1].

2 Comparing Syntax Definition Formalisms

These are the three main disambiguation constructs in SDF2 and Rascal:

- **Priorities and associativity** for binding strength of operators in expression languages;
- **Reject rules** for keyword reservation
- **Follow restrictions** for longest and first match

2.1 Associativity and Priority Disambiguation

In Figure 1 the use of priority and associativity declarations is shown for the same language "Exp" in both SDF2 and Rascal. Associativity can be applied to a single rule, e.g. $*$, or a group of rules ($-$ and $+$). We see a binary ordering $>$ which is to be interpreted as a strict partial order (transitive, irreflexive, asymmetric) between rules.

In both formalisms, each priority and associativity declaration defines a set of disallowed derivation steps in the respective grammar. Namely for "left" associative binary recursive rules a rule in the same group must not be derived on the right-hand side non-terminal of the rule. Vice versa for "right" associativity. If a rule is not binary recursive (i.e. it is of the form `X ::= X ... X` then the filter has *no effect*.

Also, both formalisms share similar semantics for the priority relation. If production A has a higher priority than another production B, then never shall B be a direct child of A. No non-terminal of A will be recognized using the application of rule B. SDF2 priority rules have no effect unless the two rules in question are shaped like so: `X ::= ε X ... | ... X ε` or so: `X ::= ... X ε | ε X ...`.

```
module ExpInSDF2
context-free syntax
  Id          -> Exp
  "(" Exp ")" -> Exp {bracket}

context-free priorities
  Exp "[" Exp "]" -> Exp <0> >
  Exp "*" Exp     -> Exp {left} >
  { left:
      Exp "+" Exp -> Exp
      Exp "-" Exp -> Exp }
```

```
module ExpInRascal
syntax Exp
  = Id
  | bracket "(" Exp ")"
  | Exp "[" Exp "]"
  > left Exp "*" Exp
  > left ( Exp "+" Exp
         | Exp "-" Exp
         );
_
_
```

Figure 1 Comparing expression language definition between SDF2 and Rascal; minor meta-syntax differences but conceptually the same.

```
module RejectInSDF2
context-free syntax
  Id           -> Exp
  Keyword      -> Id {reject}
  "begin"      -> Keyword
  "end"        -> Keyword
```

```
module RejectInRascal
syntax Exp = Id \ Keywords;
keyword Keywords
  = "begin"
  | "end"
  ;
```

Figure 2 Comparing reject rules between SDF2 and Rascal.

The recursive positions must overlap at a left-most or right-most position (or their pre/postfixes must derive the empty sequence ε).

For SDF2, if the user makes a mistake and defines a priority relation that is reflexive or symmetric ($A > B$ and $B > A$), then the filter may remove too many derivations from the generated parser with parse errors as a result. Also, SDF2 removes *all* nested derivations of B under A if $A > B$, including the position between brackets in `Exp "[" Exp "]"`. This would make it impossible to write `a[1+2]`. And so SDF2 users write $< 0 >$ before that rule to limit the filter to the first argument position and ignore it for the second. For Rascal the set of generated filter positions is filtered itself; only the positions that are guaranteed to generate ambiguity are filtered and the other positions are ignored. If the priority relation is not a partial order, the user is provided with an error message.

In short: the semantics of associativity and priority disambiguation is described in terms of derivation step filters on the grammar level. We refer to Eelco's thesis [45], this paper on ambiguity diagnostics [5], and this on disambiguating expression grammars [3], which explain the above formally.

The correctness of the semantics of these constructs relies on the guaranteed ambiguity of binary expression operators that are left and/or right-recursive, such that the filter does not remove the last derivation from a Parse Tree [3]. So, their implementation should prevent the effects of such derivations or remove them, somewhere in the respective parsing architecture in order to implement this filtering behavior.

2.2 Reject Rules

Figure 2 depicts the two styles of reserving keywords as a disambiguation mechanism. The `reject` tag was introduced by Eelco Visser in 1997. The semantics is that any subsentence recognized by *any* derivation for `Id` which can also be recognized as `Keyword`, at that same input position and having the same length, must be filtered. As described, the mechanism extends context-free grammars to include the *intersection* of context-free non-terminals and it should be possible to generate parsers for the famous non-context-free example $a^n b^n c^n$. Later we read why this was not accomplished with SDF2.

```
module RestrictionsInSDF2
lexical syntax
  [A-Za-z][A-Za-z0-9]* -> Id
lexical restrictions
  Id -/- [A-Za-z0-9]
```

```
module RestrictionsInRascal
lexical Id
  = [A-Za-z][A-Za-z0-9]*
  !>> [A-Za-z0-9];
_
```

Figure 3 Comparing follow restriction between SDF2 and Rascal.

The Rascal reflection of this design is the \ operator that removes the language of `Keywords` from *that specific* use of `Id` in `Exp`. The difference is thus that Rascal defined a derivation step filter for a specific position of `Id` in a specific rule, while SDF2 defined a filter for all uses of `Id` in any rule. Nevertheless, the expressive power would be the same, if the Rascal designers had not limited the keyword non-terminals to generate non-empty and finite languages only.

2.3 Follow restrictions

Here in Figure 3 we even more clearly step out of the realm of context-free grammars. Follow restrictions in Rascal and SDF2 express, literally, constraints on what comes *after* the character yield of a recognized non-terminal. In SDF2 we define a filter that removes all derivations of `Id` anywhere if the single character that would follow it in the input is a member of the character class `[A-Za-z0-9]`. Although this is not a local property of a Parse Tree, it is a local input of most parsing algorithms that move from left to right through the input. The next character or token in the stream is usually referred to as the "lookahead token." SDF2 also has multi-character lookahead tokens for restrictions, which lead to the same semantics (but an arguably much more complex implementation).

In Rascal the semantics is similar, but we define the derivation filter not for all instances of the non-terminal, but only for the position in the rule that the restriction is applied. Next to this Rascal features the complement: a follow requirement declares that a non-terminal *must* be followed by a certain character class, and their analogous duals: precede restrictions and requirements. Rascal, like SDF2, also features multiple look-ahead characters.

While explaining the semantics of the three disambiguation constructs we used only ideas such as Grammar, Derivation steps, Parse Trees, and Input sentences. There is no distinction between top-down and bottom-up because we have yet to dive into the implementations of these filters.

3 Comparing Parsing and Disambiguation Algorithms

We are comparing the *parsing and disambiguation* algorithms SGLR and SGLL as they were part of the parsing architectures for SDF2 and Rascal as they were in 2010. We will show details of the SGLR implementation in C and ASF+SDF and in Java and Rascal of the Rascal implementation.

Figure 4 shows the parsing and disambiguation architecture of SDF2 with SGLR in it. As you can see a disambiguation filter may end up filtering a rule from a grammar completely, modify the SLR parse table to prevent certain derivations to occur at all, or inject itself in the parser run-time to prevent parser driver operations that have the same effect. Eventually, every filter could be implemented by a post-parse tree transformation.

Figure 5 shows a different architecture because there is no intermediate stage for parse table generation. However, Rascal does have a grammar rule merging step while loading a new grammar that captures some of the partial evaluation that is done while building parse tables as well.

The two premises of implementing SDF2's disambiguation constructs are [26, 39]

- efficiency: implement filters as early as possible in the parsing architecture; preventing non-determinism is better than fixing ambiguity later. For Rascal, we adopted a similar but less

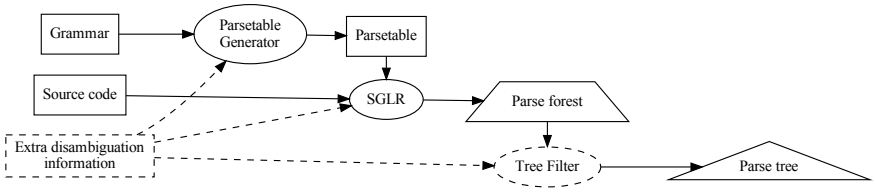

Figure 4 Bottom-up SDF2 architecture based on SGLR

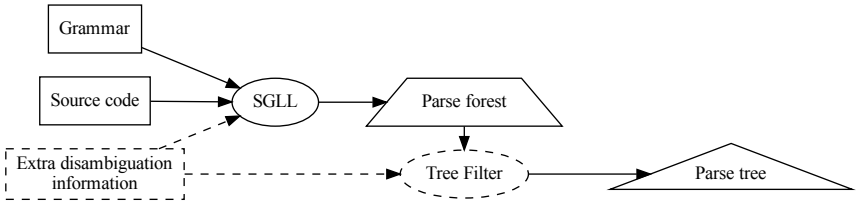

Figure 5 Top-down Rascal architecture based on SGLL

far-fetching dogma: better filter while predicting than filter while accepting a derivation.

- grammar neutrality: do not change the shape of the grammar rules, such that also the shape of the Parse Trees is unaffected. This adds to the predictability of the shape of the forests as well as efficiency since tree structure does not need to be reconstructed. For Rascal, this is also a core design constraint.

3.1 Implementing priority and associativity

In a bottom-up parser, it is possible to *completely prevent* the creation of derivations that do not satisfy the constraints generated from associativity and priority declarations[1] The major vehicle for this is the parse table generator.

SDF2 uses DeRemer's SLR(1) [14] table construction. An SLR(1) parse table is something most students must be able to generate from a grammar by hand in their Compiler Construction course. The table represents a state machine for a pushdown automaton driver that will recognize a string or not, using the information in the table. One could look at the table as a partially evaluated parsing algorithm (typically Earley's algorithm [16]) where the grammar is interpreted but the input sentence is left open. Eelco's idea is that any **reduce** P_1 actions going out of a state can be completely removed if said state witnesses that P_2 is its parent at the wrong position according to $P_2 > P_1$.

However, SLR(1)'s follow sets are defined on non-terminals and not production rules, so they do not represent rules that clearly. Every goto action out of a state may represent a union of rules for different non-terminals and different positions in different rules of the same non-terminal. Seeing that we use a non-deterministic parsing driver, Eelco observed that it was possible to redefine SLR follow sets per production instead of per non-terminal; and this enabled a full implementation of the priority and associativity semantics. The overhead is that different rules would exit a state on sometimes the same follow set to the same state, but at least never could an illegal transition be made anymore. The

[1] Later Peter Mosses found out that there are cases where SDF2 is theoretically incomplete for expressing binding strength using single derivation step filters and that required the development of a new theory and new implementations. This is out of the scope of the current paper.

breakthrough here was that this solution, on top of Rekers' version of Tomita's GLR, could deal with *any* context-free grammar and not only the LR(k) class.

The actual code that implements this filter in SDF2 is written in ASF+SDF and C (against ApiGen-generated ATerm interfaces). The expensive part is the transitive closures and cartesian products part, which was ported to C after the declarative version in ASF+SDF proved to be a bottleneck. This implementation derives triples (P_1, pos, P_2) that explain for every rule P_1 at which position pos the other rule P_2 should be disallowed. The worst-case size of the set of these triples is quadratic in the number of original production rules. Later the relatively simple SLR table generator takes this set as an additional argument and surgically does not add reductions if they occur in the set (Figure 6).

The Rascal implementation of the same filter uses a comparable triplet computation but is written in (higher-level) Rascal. The Rascal code also first proves ambiguity for left-most and right-most recursive positions, instead of applying the filter to all arguments of every production. With SGLL, the set of triplets is not used at parser generation time but during the prediction stages of the top-down parsing algorithm (Figure 7). Rascal's SGLL (designed and implemented by Arnold Lankamp) filters rules the moment they are applied by looking at their parent node. At this moment of rule reduction, the triplet set is queried. This has the exact same effect as the SDF2 implementation, at the cost of a hash-table lookup. In fact, the SGLL implementation numbers every production with a low integer and uses a small array to look up all the conflicting rules at a certain position. In a correct SGLL or SGLR implementation, all these solutions for priority and associativity filtering have the same effect of removing reductions without breaking the algorithm, at the cost of extra bookkeeping.

The SGLL algorithm prevents certain recursive steps dynamically while the SDF2 algorithm filters certain derivations. From a slight distance, both algorithms simulate a grammar transformation that would introduce non-terminals for every production rule and disallow certain rules based on the same constraint triplets. A factored grammar, such as we see when using LALR parsers, would probably not be much different. One could say that by requiring not to change the grammar, we have to simulate those grammar changes on a *lower level of abstraction* to achieve the same effect. Accidentally, a similar "set of integer production rules representation" used by Arnold Lankamp in SGLL, Eelco had envisioned earlier with "character-class grammars" [44]. There each non-terminal would also be represented by a set of active production rules for that level in the grammar and removing a production would entail implementing a filter on the grammar level.

The SDF2/SGLR solution requires theory: does the implementation satisfy all the constraints derived from the semantics of the formalism? Well, only if we change the concept of what an SLR(1) table is a bit, such that it fits. The solution does not generalize to other standard table formats, like LALR(1). The Rascal/SGLL solution sits very tightly on the concept of top-down parsing where recursion on the way down models prediction and coming back up models acceptance/reduction; it can be added to any (G)LL(1) algorithm implementation technique.

3.2 Implementing reject

The way we write SDF2 `reject` rules already leaks something about the implementation strategy of keyword reservation. We simply schedule the rejected rule along with the rest of the grammar. Then, when *all alternatives* for the `Id` non-terminal have finished, we check if one of them was accidentally a `reject` rule and if so we remove all those derivations from the computation.

With this elegant approach, Eelco had found a near-optimal solution [45]. We do not have to start a whole other parser with its own stack. Instead, we surf on the non-deterministic graph-structured stack of Tomita's algorithm and pay only a low overhead of recognizing an additional alternative.

However, this implementation had to be revisited and revised many times between the years 2000 and 2005, and eventually, we had to admit general non-terminals could not be "rejected" by this algorithm. The problem with `reject` is very much akin to the original bug in Tomita's algorithm, which

```
1  ATermList shift_prod(ItemSet items, int prodNr) {
2    Item item, newitem;
3    PT_Symbol symbol;
4    ATermList newvertex = ATempty;
5    ItemSetIterator iter;
6    PT_Production prod = PGEN_getProductionOfProdNumber(prodNr);
7
8    symbol = PT_getProductionRhs(prod);
9
10   ITS_iteratorPerDotSym(items, symbol, &iter);
11   while (ITS_hasNext(&iter)) {
12     item = ITS_next(&iter);
13     assert(PT_isEqualSymbol(symbol, IT_getDotSymbol(item)));
14     newitem = IT_shiftDot(item);
15     if (newitem != NO_ITEM
16         && !PGEN_isPriorityConflict(item, prodNr)) {
17       newvertex = ATinsert(newvertex, IT_ItemToTerm(newitem));
18     }  }
19
20   return newvertex; }
```

Figure 6 C code snippet in the SDF2 parser generator with a single surgical addition predicate on line 16 to implement associativity/priority filtering. Small intervention: big impact.

occurred with hidden left recursion. Sometimes the graph-structured stack would miss reductions and Farshi fixed that [30] with an additional stage to search for the missing reductions. The `reject` implementation very much depends on the algorithm identifying a moment where all rules for the restricted non-terminal `Id` have reduced, but the original algorithm for it did not achieve this. The reject "aspect" of SGLR is scattered in different places making it hard to theorize what its effect really is. And so sometimes rules would continue even though they should have been rejected with spurious ambiguity as a result. After several experiments, the diagnosis was left-nullable rules for the restricting could lead to "escaped" reductions. Further complicating the algorithm with yet more searching on top of Farshi's fix was deemed inefficient.

The reject filter in Rascal is either an implementation of an `ICompletionFilter` or an `IEnterFilter`. The latter prevents going into a production when it is predicted while the former removes the effect of recognizing a production when it is completed. This is the general filtering scheme, which gives access to the input character array, the start, and end index of the currently recognized input, etc. Other (static) information, like which non-terminal or production is captured by the object that implements the filter. The reject filter in SGLL simply compares the input subsentence with a given list of keywords that are not allowed and fails on a match. A more complex implementation could parse the substring using another parser (or recognizer). Setting up a nested parser is not as complex or expensive in Java as it was in SGLR's C version.

The parameters of a completion filter in SGLL (see Figure 8) make explicit what can be safely used as information to filter without breaking the algorithm's assumptions. The dynamic programming techniques that are used to stay in polynomial time for an exponential number of parse trees, or even cubic, are predicated upon the identification of reusable parse stacks for reusable subsentences. When context information breaks into this equation, it breaks the underlying assumptions for sharing computations leading to false positives (spurious derivations) as well as false negatives (spurious filtering of derivations). Every new filter introduced requires new theory. The definition of `ICompletionFilter` and `IEnterFilter` in Rascal's SGLL mitigate this by allowing any filter based on the given information and guaranteeing algorithmic correctness. If you need something more, it's back to the drawing board just as with SGLR.

```
1  private void handleEdgeListWithRestrictions(...) {
2    firstTimeRegistration.clear();  firstTimeReductions.clear();
3    for (int j = edgeSet.size() - 1; j >= 0; --j) {
4      AbstractStackNode<P> edge = edgeSet.get(j);
5      int resultStoreId = getResultStoreId(edge.getId());
6
7      if (!firstTimeReductions.contains(resultStoreId)) {
8        if (firstTimeRegistration.contains(resultStoreId)) continue;
9
10       firstTimeRegistration.add(resultStoreId);
11
12       if (!filteredParents.contains(edge.getId())) {
13         AbstractContainerNode<P> resultStore = null;
14         if (edgeSet.getLastVisitedLevel(resultStoreId) == loc)
15           resultStore = edgeSet.getLastResult(resultStoreId);
16         ... /* elided error recovery code */
17         resultStore.addAlternative(production, resultLink);
18       } else {
19         AbstractContainerNode<P> resultStore = edgeSet.getLastResult(resultStoreId);
20         stacksWithNonTerminalsToReduce.push(edge, resultStore);
21       } } } }
```

Figure 7 Java code snippet in Rascal's SGLL parser run-time, with a single additional predicate on line 14 to filter associativity/priority violations. Again: small intervention; big impact.

```
1  public interface ICompletionFilter {
2    boolean isFiltered(int[] input, int start, int end, PositionStore positionStore); }
```

Figure 8 Rascal's SGLL completion filter interface code.

3.3 Implementing follow restrictions

The final filter, however innocuous, proved to be another grand challenge for implementing in SDF2 and SGLR. Firstly, the basic implementation for single-character lookahead was simply to remove the given characters from the lookahead sets in the SLR(1) table [45]. Secondly, multiple character lookahead would be implemented by dynamically filtering reductions in the inner parser loop. With the right internal administration that associates the lookaheads with every production rule, the code change in the algorithm is minimal.

However, at the time we were using the ATerm library for representing parse trees with its *maximal sharing* ability. Parse tree nodes that were structurally equal, would always be shared. Sharing was thus based on the contents of the trees' sub-nodes, and not on the context. This design decision is efficient in the context of ambiguity where lots of sub-nodes would be structurally equal (whitespace). With a theory of context-free grammars this all works fine. With the theory of context information with follow restrictions, maximal sharing breaks the parser. To fix the bugs, we ended up introducing an additional lookup table for ambiguous parse trees where the starting position in the input sentence became part of the lookup key.

Having learned from the experience, the Rascal SGLL implementation used its own intermediate SPPF-like [37] data structure for parse forests, which is then serialized to AsFix Parse Trees after the parse is done. The aforementioned `ICompletionFilter` can be used to implement follow restrictions in a single line of code. `IEnterFilter` would be used for the precede variants. If you'd start from scratch to implement SGLR in Java, for example, you would use the same trick. Indeed the JSGLR code by Karl Trygve Kalleberg in Spoofax, uses an ambiguity hash-table that also includes the start position of every tree.

Feature	Grammar	Implementation
SDF2 Priority / Associativity	$P_1 > P_2$, `left`, `right`	1. Compute constraint triplets 2. SLR(1) follow-sets per rule 3. Filtering goto's in SLR(1)
Rascal Priority / Associativity	$P_1 > P_2$, `left`, `right`	1. Compute constraint triplets 2. Filter forest edge creation
SDF2 Keyword reservation	`Kw -> Id {reject}`	1. Grouping reductions 2. Filter reductions
Rascal Keyword reservation	`Id \ Kw`	1. `ICompletionFilter`
SDF2 Longest Match	`Id -/- [A-Z]`	1. Goto Filter on follow sets 2. Reduction filter for k lookahead
Rascal Longest Match	`Id !>> [A-Z]`	1. `ICompletionFilter`

Table 1 Overview of the disambiguation implementations for either SDF2 (bottom-up) or Rascal (top-down).

3.4 Top-down disambiguation is easier to get right

Table 1 summarizes how the three declarative implementation constructs were implemented in either a top-down (Rascal) or bottom-up (SDF2) parsing architecture. Once you have the theory straight, the implementation of a disambiguation filter seems a surgical incision in either algorithm: a conditional around the scheduling of the next step. If only this were true.

Implementing the Reject and Follow Restriction filters has proven to be complex for the SGLR architecture while it is straightforward in SGLL. Moreover, the priorities and associativity filter on the parse table level can never be complete, even though it is elegant. Small changes in the parse table, such as filtering a goto edge, may break the conditions under which parse stacks or parse tree nodes can be shared later in the GLR algorithm. These semantic links are platonic, in the sense that they do not lead to explicit dependencies on the source code level of SGLR, however, when not taken care of complex bugs do arise. Concretely, the addition of Follow Restrictions to the SLR table construction algorithm broke many of the underlying assumptions of the SGLR implementation and required the reconsideration of all of its internal data structures. On the other hand, for SGLL a follow restriction was a simple conditional while scheduling the next algorithmic step.

The reject filter for SGLR proved to be even more difficult to implement. The actual filter operation is to remove all other reductions for a non-terminal in the presence of a "rejected" one for the same sub-sentence and non-terminal. The SGLR algorithm does not schedule reductions in such a way that a clear moment arises when all possible reductions for a subsentence have been collected. Sometimes graph stack nodes are processed already for further reductions (chain rules), and that way a tree would escape that would otherwise have been filtered. Sometimes the rejected stack node itself would be processed too early, letting later nodes escape. The first problem was solved by changing the GLR algorithm to group reductions in the same starting position of the input. The latter problem was solved by disallowing more complex non-terminals to be rejected, limiting them to finite non-nullable languages. The reject filter in SGLL is a simple reduction filter.

We conclude that top-down is much easier to experiment with and extend. Bottom-up could be faster due to partial evaluation, but still, additional bookkeeping and less sharing are required to filter correctly. Rascal's SGLL in Java is as fast as the SDF2's SGLR in C.

4 Perspective on contextual disambiguation with DDCFGs

Let's step back from the comparison made between top-down general and bottom-up general parsing with disambiguation and zoom out to the general problem of disambiguation.

- All three disambiguation constructs use *context information*.
- Each of the three disambiguation constructs is an *ad-hoc* extension of the SDF [43, 6].
- Each disambiguation construct deeply impacts the parsing algorithm.

Never mind that they are easier to implement in GLL, but what is the best way of formulating the next disambiguation construct on the SDF level? Say we want to support the offside rule [28]. How to implement it in (S)GLL? Disambiguation always adds "context" to the algorithm of constructing parse trees as compared to "context-free" grammars. At the LDTA conference in 2011, Trevor Jim and Yithzak Mandelbaum demonstrated the utility and elegance of data-dependent context-free grammars with their Yakker parser generator [21, 20]. Much earlier, Mark van den Brand in his PhD thesis [40] also demonstrated that parse-time semantic predicates can be used elegantly and efficiently to disambiguate (lexical and context-free) ambiguity.

A Data-dependent Constraint Grammar is a context-free grammar with three major extensions: (a) the non-terminal on the left-hand side of any rule may receive additional data parameters, (b) every symbol on the right-hand side may be conditional on said data parameters using constraint formulas, and (c) data from the input sentence or of syntax trees already processed may be passed as parameters to non-terminals or constraints. Typical "data" would be the character string of a sub-sentence, the start and ending position of every rule, the current indentation level, etc. Typical formulas would be integer arithmetic (for layout positioning), and string (in)equality, but in general, any predicate *without side-effects* written in the host programming language is ok.

Afroozeh and Izmaylova [2, 1] mapped all of SDF's and Rascal's disambiguation mechanisms to their Iguana formalism which is based on data-dependent grammars, and then immediately added many more disambiguation constructs. For example, with Iguana it is possible to declaratively express the offside rule and many other "two-dimensional" layout constraints for programming languages such as Haskell and Python. Iguana is a top-down parsing architecture, as the reader might expect. It is also possible to implement data-dependent grammars on top of Earley's algorithm [21, 16].

However, the semantics of Disambiguation remains the same whether you implement your DDCFG parsing algorithm in a top-down or a bottom-up data-dependent context-free general framework. The formal semantics of data-dependent context-free grammars acts as a virtual machine for disambiguation constructs, making it easier to reason about correctness independent of the implementation in a complex parsing algorithm [1].

5 Conclusion

First, contextual disambiguation is a pleonasm. Second, it is arguably easier to design and implement (new) disambiguation constructs with GLL than with GLR. Third, data-dependent context-free grammars add the level of formality and generality that we were always searching for when inventing new disambiguation schemes (as exemplified by the Jakker and Iguana Parsing Architectures). We conclude that a *top-down* implementation of *data-dependent* context-free parsing is the way to go for Rascal as well as SDF3.

References

1 Ali Afroozeh and Anastasia Izmaylova. One Parser to Rule Them All. In *Proceedings of the 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2015, pages 151–170. ACM, 2015. doi:10.1145/2814228.2814242.

2 Ali Afroozeh and Anastasia Izmaylova. Iguana: a practical data-dependent parsing framework. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, pages 267–268. ACM, 2016. doi:10.1145/2892208.2892234.

3 Ali Afroozeh, Mark van den Brand, Adrian Johnstone, Elizabeth Scott, and Jurgen J. Vinju. Safe specification of operator precedence rules. In *International Conference on Software Language Engineering (SLE)*, LNCS. Springer, 2013.

4 Luís Amorim and Eelco Visser. *Multi-purpose Syntax Definition with SDF3*, pages 1–23. 09 2020. doi:10.1007/978-3-030-58768-0_1.

5 Bas Basten and Jurgen Vinju. Parse forest diagnostics with dr. ambiguity. In *International Conference on Software Language Engineering (SLE)*, LNCS. Springer, 2011.

6 Bas Basten and Jurgen Vinju. Parse forest diagnostics with Dr. Ambiguity. In *International Conference on Software Language Engineering (SLE)*, LNCS. Springer, 2011.

7 Mark G.J. van den Brand, Arie van Deursen, Jan Heering, Hayco A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul. Klint, Leon Moonen, Pieter .A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

8 Mark G.J. van den Brand, Steven Klusener, Leon Moonen, and Jurgen J. Vinju. Generalized Parsing and Term Rewriting - Semantics Directed Disambiguation. In Barret Bryant and João Saraiva, editors, *Third Workshop on Language Descriptions Tools and Applications*, Electronic Notes in Theoretical Computer Science. Elsevier, 2003.

9 Mark van den Brand, Jørgen Iversen, and Peter Mosses. An Action Environment. *Electr. Notes Theor. Comput. Sci.*, 110:149–168, 12 2004.

10 Mark van den Brand and Paul Klint. ATerms for manipulation and exchange of structured data: It's all about sharing. *Information & Software Technology*, 49:55–64, 01 2007.

11 Mark van den Brand, Paul Klint, Hayco de Jong, and Pieter Olivier. Efficient annotated terms. *Software—Practice & Experience*, 30(2), January 2000.

12 Mark van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. Environments for term rewriting engines for free! In *Proceedings of the 14th International Conference on Rewriting Techniques and Applications*, RTA'03, page 424–435, Berlin, Heidelberg, 2003. Springer-Verlag.

13 Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1):52–70, 2008. Special Issue on Second issue of experimental software and toolkits (EST). doi:https://doi.org/10.1016/j.scico.2007.11.003.

14 Frank DeRemer. Simple lr(k) grammars. *Commun. ACM*, 14(7):453–460, 1971. doi:10.1145/362619.362625.

15 Arie Van Deursen, Jan Heering, and Paul Klint. *Language Prototyping: An Algebraic Specification Approach: Vol. V*. World Scientific Publishing Co., Inc., USA, 1996.

16 Jay Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13:94–102, February 1970. doi:http://doi.acm.org/10.1145/362007.362035.

17 Giorgios R. Economopoulos, Paul Klint, and Jurgen J. Vinju. Faster scannerless GLR parsing. In Oege de Moor and Michael I. Schwartzbach, editors, *Compiler Construction (CC)*, volume 5501 of *Lecture Notes in Computer Science*, pages 126–141. Springer, 2009. doi:http://dx.doi.org/10.1007/978-3-642-00722-4_10.

18 J. Heering and P. Klint. A syntax definition formalism. Technical report, 1986. ESPRIT'86: Results and Achievements, page 619–630.

19 Jan Heering, Paul R. H. Hendriks, Paul Klint, and Jan Rekers. The syntax definition formalism SDF — reference manual. *SIGPLAN Not.*, 24(11):43–75, nov 1989. doi:10.1145/71605.71607.

20 Trevor Jim and Yitzhak Mandelbaum. Delayed semantic actions in yakker. In Claus Brabrand and Eric Van Wyk, editors, *Language Descriptions, Tools and Applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding*, page 8. ACM, 2011. doi:10.1145/1988783.1988791.

21 Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and algorithms for data-dependent grammars. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 417–430. ACM, 2010. doi:10.1145/1706299.1706347.

22 Lennart C.L. Kats and Eelco Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. *SIGPLAN Not.*, 45(10):444–463, oct 2010.

23 Paul Klint. *From SPRING to SUMMER: design, definition and implementation of programming languages for string manipulation and pattern matching*. PhD thesis, Technische Hogeschool Eindhoven, March 1982.

24 Paul Klint, Tijs van der Storm, and J.J. Vinju. EASY meta-programming with Rascal. In João Fernandes, Ralf Lämmel, Joost Visser, and João Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *LNCS*, pages 222–289. Springer Berlin / Heidelberg, 2011.

25 Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177. IEEE Computer Society, 2009. doi:http://doi.ieeecomputersociety.org/10.1109/SCAM.2009.28.

26 Paul Klint and Eelco Visser. Using filters for the disambiguation of context-free grammars. In G. Pighizzini and P. San Pietro, editors, *Proc. ASMICS Workshop on Parsing Theory*, pages 1–20, Milano, Italy, 1994. Tech. Rep. 126–1994, Dipartimento di Scienze dell'Informazione, Università di Milano.

27 Ralf Lämmel and Chris Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

28 P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9:157–166, March 1966.

29 Leon Moonen. Generating robust parsers using island grammars. *Proceedings Eighth Working Conference on Reverse Engineering*, pages 13–22, 2001.

30 Rohman Nozohoor-Farshi. Handling of ill-designed grammars in tomita's parsing algorithm. In *Proceedings of the First International Workshop on Parsing Technologies*, pages 182–192, Pittsburgh, Pennsylvania, USA, August 1989. Carnegy Mellon University. URL: https://aclanthology.org/W89-0219.

31 J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

32 Jan Rekers and Wilco Koorn. Substring parsing for arbitrary context-free grammars. In *Proceedings of the Second International Workshop on Parsing Technologies*, pages 218–224, Cancun, Mexico, February 13-25 1991. Association for Computational Linguistics. URL: https://aclanthology.org/1991.iwpt-1.25.

33 D. J. Salomon and G. V. Cormack. Scannerless NSLR(1) parsing of programming languages. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI 1989, pages 170–178. ACM, 1989. doi:http://doi.acm.org/10.1145/73141.74833.

34 Elizabeth Scott and Adrian Johnstone. Right nulled GLR parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, jul 2006.

35 Elizabeth Scott and Adrian Johnstone. GLL parsing. *ENTCS*, 253(7):177 – 189, 2010. Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009).

36 Thomas A. Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

37 M. Tomita. *Efficient Parsing for Natural Languages. A Fast Algorithm for Practical Systems*. Kluwer Academic Publishers, 1985.

38 Mark van den Brand, Jan Heering, Paul Klint, and Pieter A. Olivier. Compiling language definitions: The ASF+SDF compiler. *CoRR*, cs.PL/0007008, 2000. URL: https://arxiv.org/abs/cs/0007008.

39 Mark van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002*, volume 2304 of *LNCS*, pages 143–158. Springer, 2002.

40 Mark G. J. van den Brand. *PREGMATIC - a generator for incremental programming environments*. PhD thesis, Radboud University Nijmegen, 1992.

41 Mark G. J. van den Brand, Pierre-Etienne Moreau, and Christophe Ringeissen. The ELAN Environment: an Rewriting Logic Environment based on ASF+SDF Technology. In *Workshop on Language Descriptions,*

Tools and Applications - LDTA'02, volume 65/3 of *Electronic Notes in Theoretical Computer Science*, page 7 p, Grenoble, France, April 2002. Colloque avec actes et comité de lecture. internationale. URL: https://hal.inria.fr/inria-00101028.

42 J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, Universiteit van Amsterdam, November 2005.

43 Jurgen J. Vinju. SDF disambiguation medkit for programming languages. Technical Report SEN-1107, Centrum Wiskunde & Informatica, 2011. http://oai.cwi.nl/oai/asset/18080/18080D.pdf.

44 Eelco Visser. From context-free grammars with priorities to character class grammars. In Mieke Brune Arie van Deursen and Jan Heering, editors, *Dat Is Dus Heel Interessant, Liber Amicorum dedicated to Paul Klint*. Centrum Wiskunde & Informatica and IVI Universiteit van Amsterdam, 1997.

45 Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, Universiteit van Amsterdam, 1997.