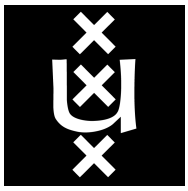

University of Amsterdam

Master Thesis Software Engineering

Measuring the quality of domain-specific language implementation approaches: Java versus ANTLR

Author:
Jeroen van Schagen
jeroen.v.schagen@gmail.com

Supervisor:
Dr. Tijs van der Storm
storm@cwi.nl

Host organization:
Centrum Wiskunde & Informatica (CWI)

Publication status:
Public domain

September 18, 2009

Measuring the quality of domain-specific language implementation approaches: Java versus ANTLR

Jeroen van Schagen
MSc. Software Engineering
University of Amsterdam, Netherlands
Centrum Wiskunde & Informatica, Netherlands
Supervised by Tijs van der Storm
jeroen.v.schagen@gmail.com

September 18, 2009

Abstract. Usage of a domain-specific language (DSL) leads to significant benefits to end-users, however the development and maintenace process can be quite costly. Various approaches to implementing a DSL are described in the literature, but developers are often left clueless about which approach is most beneficial for their situation. This study is part of a global research project that compares DSL implementation approaches, to compose guideliness that assist developers selecting the most beneficial implementation approach. In this particular study, the traditional "compiler" approach in Java is compared to the ANTLR "compiler generator". Using ANTLR to generate a parser leads to significantly fewer development- and maintenance effort, but performance and error messages are worse. Generating a checker and interpreter with ANTLR has potential, but is too limited in its current condition.

Keywords. Domain-specific language, Implementation approach, Compiler generator, Grammar metrics

1 Introduction

A Domain-Specific Language (DSL) is a language that is designed towards a specific domain. Usage of DSLs can lead to substantial gains in expressiveness and ease of use, when compared to a general-purpose language [MHS05]. By offering domain-specific notations, a DSL can greatly enhance productivity and in some cases enable end-user programming [KLBM08]. The advantages of DSLs are supported by studies of [HB88] and [KMB+96].

Despite the clear advantages that DSLs offer, the decision to develop a DSL is rarely taken [MHS05]. Developing a DSL is complicated, requiring both domain knowledge and language development experience. Due to a lack of guidelines and experienced reports on DSL development, developers often resort to non-linguistic approaches [KLBM08]. Which is unfortunate, as a DSL designed for a well-defined domain and implemented with adequate tool support can significantly reduce development- and maintenance effort [vDK98].

Compiler generators can be used to facilitate the DSL implementation process. A compiler generator automates the construction of various language processing tools based on a formal language description, also known as the *grammar*. This grammar is a collection of rules, consisting of one or more alternatives, that describe the structure of a particular language. Having a concise grammar is useful for the sake of documentation and early detection of inconsistencies [BS86]. Compiler generators supposedly reduce implementation effort [MHS05], enhance maintenance [vDK98] and result in more evolvable software [BLS98]. Nevertheless, actual advantages of compiler generators have never been empirically studied.

This study is part of a global research project, in which DSL implementation approaches, as categorized in [MHS05], are investigated. By performing a quantitative comparison between DSL implementation approaches, guidelines can be constructed that

assist DSL developers in selecting the most suitable implementation approach. In this study, the qualities of a manual compiler approach in Java, that will be referred to as the "vanilla" approach, and a compiler generator approach using ANTLR are measured and analyzed. Using both approaches the WAEBRIC [vdS09] language will be implemented, resulting in a parser, type-checker and interpreter.

Related work

Comparison of DSL implementation approaches is a rather new field of research. T. Kosar [KLBM08] conducted the only known study that compares approaches by implementing the same language. Implementation approaches are evaluated based on effective lines of code normalized by a language factor, performance, development effort and user satisfaction. Maintenance effort remains undiscussed, while maintenance roughly consumes up to 80% of the project resources [LST78].

Kosar concludes that usage of an embedding approach, by extending a host language, requires the least development effort. However, due to the inherited notations, an embedded implementation is less domain-specific, more complex and thus harder to use. Embedding is suggested for a DSL with a small set of users, and when notation should not be strictly obeyed. Both compiler and compiler generator approaches managed to obtain an optimal notation. Compiler generator implementations contain a factor 10 fewer lines of code for syntax, but no notable difference in semantics. Varieties in performance were neglectable.

The remainder of this article is structured as follows. Section 2 presents the research questions, hypothesis and explains the conducted experiment in detail. Execution of the experiment is described in section 3. Gathered results are displayed and rationalized in section 4, followed by an evaluation on both approaches in section 5. Finally, the research questions are answered in section 6.

2 Research approach

To examine the benefits that an ANTLR generated DSL implementation has over a vanilla Java implementation, both approaches are applied in a formal experiment. Comparison variables are minimized by implementing the same DSL and utilizing one developer for both approaches. Using both approaches the WAEBRIC language is implemented, resulting in a parser, type-checker and interpreter.

2.1 WAEBRIC

WAEBRIC is a domain-specific language for generating XHTML markup, based on reusable function building-blocks [vdS09]. WAEBRIC offers compact and user-friendly syntax with powerful semantics, allowing users to efficiently write websites. A simple program written in WAEBRIC look as follows:

```
module welcome
import util;

site
  welcome.html: welcome("Jeroen");
end

def welcome(name)
 html {
  header("Welcome");
  p "Welcome " + name + "!" ;
 }
end
```

WAEBRIC programs always start with a module definition that corresponds to the basename of the file, in this case "welcome.wae". This moduel contains a site definition stating that "welcome.html" should consists of the evaluated output of function *welcome* with a string valued argument. Inside function *welcome* the markup *html* is specified. Because *html* is not defined as function, it is recognized as XHTML tag. Markups can be nested in a sequence of function calls and XHTML tags. By using a block, enclosed in curly bracers, function call *header*, defined in module *util*, and tag *p* are nested behind *html*. Evaluating the above specified program results in the following XHTML code:

```
<html>
 <title>Welcome</title>
 <p>Welcome Jeroen!</p>
</html>
```

WAEBRIC also offers common programming constructs, such as **if** and **else** for conditionals, **each**

for iterations, **let** for object bindings and **yield** for additional markup parametrization.

2.2 Research question

During this study the following research question will be investigated:

Does an ANTLR generated WAEBRIC implementation have a higher quality, than one manually developed in vanilla Java?

Where implementation quality is decomposed in the following attributes: functionality, maintainability, performance and usability. Each of these quality attributes is below discussed in more detail.

2.2.1 Functionality

Functionality describes how-much the implementation conforms to the reference implementation. To measure functionality, a test set of WAEBRIC programs is defined. Functionality is determined by the percentage of programs that successfully build, which will be referred to as acceptance. More acceptance is considered better as it shows higher conformance to the specification. Lower acceptance might indicate that various notations or constructions are particularly difficult, or impossible, to express using that particular approach.

Hypothesis. Functionality will likely be similar for both implementations. ANTLR has proven its syntactical capabilities by implementing languages similar to WAEBRIC. The semantics are implemented in Java code snippets, which offers identical capabilities as a vanilla implementation.

2.2.2 Maintainability

Domains frequently change, meaning the DSL has to evolve in order to stay beneficial. The ease at which a language can be modified or extended is called maintainability. Maintainability will be measured using metrics, offering an objective and quantitative comparison between the implementations.

Metrics are standard practice for general programming languages, but these metrics are not always applicable to a grammar based language such as the ANTLR grammar. ANTLR grammars formally describe syntax using Extended Backus-Naur Formalism (EBNF) notation. EBNF defines production rules, or productions, where an identifier symbol, called the non-terminal (NT), is respectively assigned to a sequence of terminal (T) and NT symbols. Terminal symbols are literal strings that cannot be decomposed further in the grammar, without losing their meaning. The maintainability of EBNF specifications will be measured according to the metric suite proposed in [PM04]. Semantics are described in action code embedded between EBNF productions. This action code is written in plain Java and will be analyzed accordingly.

Table 1 shows the names and abbreviations of all calculated metrics. How each metric is calculated and interpreted will be discussed below.

Metric	Abbr.
Non-commented Lines of Code	NCLOC
Method Lines of Code	MLOC
McCabe Complexity	MCC
Halstead Effort	E
Method count	#M
Average size of a method	AVS
Maintainability Index	MI
Class count	#C

Table 1: Maintainability metrics

NCLOC measures the total non-blank and non-comment lines of code. NCLOC is measured, as it remains the default size metric, which can also be easily visualized by readers. Lower NCLOC values will be considered better as shorter implementations are typically easier to maintain.

MLOC measures the NCLOC that are positioned inside a method, constructor included. For similar reasons as NCLOC lower will be considered better.

MCC [McC76] counts the number of flows through a method. Methods with higher MCC values are psychologically harder to understand, and require more test cases to obtain complete coverage, resulting in us to consider them worse.

Methods without branches have a MCC of 1, considering there is only one flow available. When branching occurs due to an if, for, while, do, case and catch statement, ? ternary operator, && or || logical operator the MCC value is incremented by

one. Branches in an EBNF notation are caused by | or, ? optional and * + closure operators [PM04].

E [Hal77] determines the mental effort required to develop or maintain a program. Implementations that require fewer effort will be considered better. E is calculated using the following formula:

$$E = \frac{\mu_1 \eta_2 (\eta_1 + \eta_2) log_2(\mu_1 + \mu_2)}{2\mu_2}$$
μ_1 = number of unique operators
μ_2 = number of unique operands
η_1 = occurrence of operators
η_2 = occurrence of operands

There are many variations to how operator and operand can be interpreted. During this study the definition of Nandy[1] is used for Java specifications. ANTLR supports $\rightarrow$ $\wedge$! rewrite, .. range, $\sim$ negations, $=\triangleright?$ predicate and {} [] action operators, besides regular EBNF notation. Which leads to an extended definition of [PM04] that states as follows:

operator $\in$ { | , ? , * , + , $\rightarrow$, $\wedge$, ! , .. , $\sim$, $=\triangleright?$, {} , [] }
operand $\in$ NT $\cup$ T

#M counts the number of methods, and is a default size metric. ANTLR uses an input grammar to generate a recursive descent parser, where each production rule is implemented in a unique method. Because of this direct relation between production and method, the #M of a grammar is calculated by counting productions [PM04].

AVS calculates the average LOC per method by dividing MLOC to #M. Average method size is comparable to the average amount of alternatives, recognized by the '|' symbol, inside a production [PM04]. Methods with fewer LOC are generally easier to understand and are thus considered better.

MI [OH94] is a hybrid metric that represents the maintainability of a software system in single value. Systems with a higher MI value are supposedly more maintainable and thus considered better. MI has never been claimed the most accurate indicator of maintenance, nor has it been validated entirely. Nevertheless it offers a working model used to quickly and easily predict software maintainable, according to the following formula:

$$MI = 171 - 3.42 \ln(\text{avgE}) - 0.23 MCC - 16.2 \ln(AVS)$$
avgE = average E per method

#C counts the amount of class definitions. #C is a default size metrics which can strictly be applied to object oriented languages, meaning it cannot be mapped to ANTLR grammars.

Hypothesis. Maintainability will likely be better in the ANTLR implementation, as a grammatical notation is more expressive and thus shorter then verbose Java. It is common knowledge that shorter programs are often easier to maintain. Additionally the maintainer can purely focus on the WAEBRIC language definition, without having to maintain an architecture and internal data structures.

2.2.3 Performance

Performance describes the speed of processing a WAEBRIC program, measured by execution time in milliseconds (ms). Where a longer execution time will naturally be considered worse. Execution time is measured overall and per processing phase.

Hypothesis. Performance of the ANTLR implementation will likely be better overall. Many theoretical knowledge went into developing ANTLR, thus it is likely to contain many optimizations that the vanilla implementation does not have.

2.2.4 Usability

Usability describes the ease at which end-user can use the implementation for building WAEBRIC programs. While writing a program, end-uses want to receive well formed error messages to locate faults. The quality of error messages is determined based on the following quality attributes: cause of error, position of error and possible solution.

Hypothesis. Error message quality will likely be higher in the vanilla implementation, as they are completely customized towards WAEBRIC while ANTLR has to stay supportive to any language.

3 Implementation

Processing of a DSL program is separated into several phases, each implemented into a seperate com-

[1] http://www.scribd.com/doc/99533/Halsteads-Operators-and-Operands-in-C-C-JAVA-by-Indranil-Nandy

ponent. Figure 1 demonstrates the different components and their interaction.

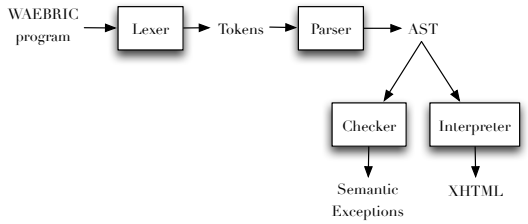

Figure 1: Components

Processing starts at the lexer, where an input program is converted into tokens. These tokens are used by the parser to generate a uniform data type, here known as the Abstract Syntax Tree (AST). Before further processing takes place, the type-checker validates that the AST is sementically correct. Afterwards, the interpreter converts the AST into one or more XHTML documents. Each component is below described in more detail, including the encountered challenges during implementation.

3.1 Lexer

Lexical analyzers, also known as *lexers* or *scanners*, convert a character stream in a collection of tokens. Tokens are sequences of characters that have a collective meaning. Character sequences forming a token are known as a *lexemes*, some examples are:

```
lexeme "abc" is an identifier token
lexeme "123" is a natural number token
lexeme "'@bc" is a symbol token
```

Separating lexical analysis from parsing leads to a cleaner overall design. This seperation of concerns also allows comment and layout to be discarded during lexical analysis to simplify parsing [ASU86].

3.1.1 Vanilla

In the vanilla implementation lexical analysis works as follows. First a token pattern is selected, based on the first input character. Following this pattern, input characters are retrieved and stored as lexeme. When a pattern is matched, the lexeme is placed inside a token object along with the token sort and position in the input. Incase the input is unable to match any pattern, an exception is thrown and the lexer continues from its current position.

3.1.2 ANTLR

While most lexer generators use regular expressions to describe token patterns, ANTLR uses a lexical grammar. ANTLR grammars support EBNF notation and uses a grammar dialect derived from Yacc [Joh75], where production identified in lowercase letters represents a syntactical definition and one in uppercase letter a token definition. Code fragment 1 shows a selection of WAEBRIC token definitions:

```
IDCON: LETTER (LETTER | DIGIT | '-')* ;
NATCON: DIGIT+ ;
SYMBOLCON: '\'' SYMBOLCHAR* ;
fragment SYMBOLCHAR: ~( '\u0000'..'\u001F'
  | ' ' | ';' | ',' | '>' | '}' | ')' ) ;
```

Code snippet 1: Lexical grammar

Productions always contain a single NT symbol on the left hand side, positioned left of the colon. On the right hand side, also known as the production body, a sequence of T symbols, surrounded by single quotes, and NT symbols is respectively assigned to this NT. For example, in the above grammar NT IDCON consists of a sequence of NT symbols DIGIT and LETTER and T symbol '-'. Conclusively, a semicolon character is used to mark the production ending. Productions that start with the **fragment** keyword are not derived in tokens, but can be referenced by other token definitions.

3.1.3 Challenges

Context sensitive definitions

During implementation the problem was encountered, that WAEBRIC defines lexemes which belong to multiple tokens. For example, the lexeme `''abc''` is recognizable as both text and string. In these situations context information needs to be used to determine the actual token sort. Strings are only usable between a **comment** keyword and semicolon symbol, in other situations the lexeme should be recognized as text. In the vanilla implementation strings are recognized by comparing the previously stored token to a **comment** keyword. EBNF grammars are context-free, as productions

cannot be restricted to particular context. ANTLR offers semantic predicates, which work as follows:

```
STRCON: {inComment} ?=> '"' STRCHAR* '"' ;
```

Code snippet 2: Semantic predicate

Semantic predicates are boolean expressions that can disable alternatives at parse time. Incase the predicate evaluates false its associated alternative will be ignored. A false *inComment* will cause the only alternative in STRCON to be ignored, leaving TEXT as only matchable production.

Context start- and end tokens are used to maintain actual context information, as demonstrated in code fragment 3. Whenever a **comment** keyword is matched boolean field *inComment* is set true, to later be reset false after matching a semicolon.

```
COMMENT: 'comment' { inComment = true; } ;
SEMICOLON: ';' { inComment = false; } ;
```

Code snippet 3: Context information

3.2 Parser

Parsers verify the structure of a source program and convert this program into an intermediate data structure, formally known as the *syntax tree*, for further analysis and processing.

3.2.1 Vanilla

For the vanilla implementation a recursive descent LL(k) parser is constructed, processing input from Left-to-right using Left-most derivation. Recursive descent parsers are top-down parsers, where each production is implemented in a seperate method. Implementing a recursive descent parser is straight forward, and results in a program that closely resembles the grammar it recognizes [ASU86].

During the top-down construction of a syntax tree, each production is treated as rewritting step. From a start symbol, designated by the grammar, each rewrite step replaces a NT with their production body. LL parsers use the Left-most derivation, where the most left positioned NT is rewritten first. Using left-most derivation leads to left associative trees, where the following structure is applied:

```
1+2+3 = (1+2)+3
```

LL(k) parsers got a fixed lookahead *k*. Lookahead refers to the scanning ahead one or more tokens in the input, in order to determine which production alternative should be used. Code fragment 4 shows a LL(k) grammar, where cdata statements can be recognized with k=1. This is because only one *stm* alternative begins with a **cdata** keyword. Distinguishing echo statements requires k=2, as expressions and embeddings are distinguishable on the first token. In the vanilla implementation alternatives are distiniguished by looking a specific amount of tokens ahead, this amount varies per production.

```
stm: 'cdata' expression ';'
   | 'echo' expression ';'
   | 'echo' embedding ';'
   ;
```

Code snippet 4: LL(k) grammar example

3.2.2 ANTLR

Traditional LL(k) parsers only support a select class of grammars. To support a larger class of grammars, ANTLR recently introduced the LL(*) parsing technique. While LL(k) parsers scan a fixed amount of tokens ahead, LL(*) allows the lookahead to continue until it runs out of input. The obvious advantage of LL(*) is that a developer does not have to calculate k, which is a difficult and time consuming process. In some cases its impossible to calculate k, such as the grammar in code fragment 5 where *stm* has an arbitrary number of *markup*.

```
stm: markup+ expression ';'
   | markup+ embedding ';'
   ;
```

Code snippet 5: Non-LL(k) grammar example

When the vanilla parser is unable to distinguish between alternatives, due to an unbound *k*, we use "backtracking". Instead of distinguishing between alternatives in advance, backtracking guesses which alternative is recognized by selecting the first possible alternative. Whenever an exception occurs the origional state is restored and second alternative selected, continuing until the production is matched.

LL(*) is able to distinguish between alternatives by using an efficiently coded Deterministic Finite Automata (DFA). Lookaheads in a DFA are small and efficient, because they do not descent deep into the rule invocation chains [Par07]. LL(*) parsers are able to recognize more grammars without backtracking, which often leads to better performance.

3.2.3 Syntax trees

Syntax trees can be either concrete, i.e. a complete represtation of the input, or abstract, which offers a highly processed and concensed version of the input. ASTs ignore information that is not required for further processing, resulting in a compact representation that is easier to understand and process, while lowering time and space overheads [Gri07].

Syntax trees are either static or dynamic typed, which embodies a trade-off between flexibility and type-safety. The vanilla parser constructs a statically typed syntax tree, which arranges nodes into a hierchy of data structures that closely resemble the grammar. Each production has a unique node class, with the alternatives embedded as sub-class. Figure 2 shows a fraction of our vanilla syntax tree, where class *module* is corresponsive to production *module*. Inside the class *module*, concrete references are maintained to its children nodes.

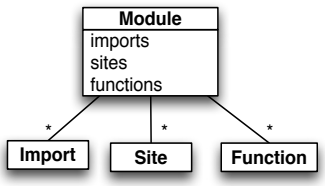

Figure 2: Statically typed AST

Using a static typing enforces type-safety, but at the cost of flexibility. Because the syntax tree data structures are tightly coupled to the grammar, modifications to the grammar automatically require changes to the data structures.

ANTLR generated parsers construct a dynamically typed syntax tree, which only relies on a single data structure. Dynamic typing makes the syntax tree flexible and easy to extend, but, due to the lack of concrete types, the tree is difficult to construct and process. Figure 3 shows the class diagram of a dynamically typed syntax tree, consisting of a single class *Node*. Instead of storing type information in concrete node classes, the variable *type* is used.

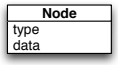

Figure 3: Dynamically typed AST

3.2.4 Challenges

Left recursion

Both implementations are unable to accept left-recursive grammars, where one or more productions are left-recursive. The production p is left-recursive when the left-most symbol in any alternative, either directly or indirectly, references to p. WAEBRIC has various left-recursive productions, for example:

```
predicate: predicate '&&' predicate
         | ...
         ;
```

Code snippet 6: Left recursive production

This production is equal to the recursive descent parse method *predicate* shown in code snippet 7. Running *predicate* invokes an infinite recursion, eventually causing a stack overflow exception.

```
void predicate() {
  predicate();
  match("&&");
  predicate();
  ...
}
```

Code snippet 7: Predicate

Left-recursion is solved by left factoring, where two or more alternatives are placed inside a single alternative by merging a common prefix. Code snippet 8 features the left factored production of *predicate*, where NT symbol *nonRecursivePredicate*, consisting of all non left-recursive alternatives, is used as common prefix. Remainders of the recursive alternatives are placed behind, allowing zero or more occurences.

```
predicate: nonRecursivePredicate
           ( '&&' predicate )* ;
```

Code snippet 8: Left factoring

WAEBRIC is unrecognizable by both implementations without the use of left-factoring. However, because left-factoring merges the syntax of multiple alternatives, the implementation becomes more complex and harder to understand.

Import

WAEBRIC programs are modular as definitions from other modules can be included by using import constructs. To simplify further analysis and processing, all imported modules are parsed and stored inside the AST. For both implementations the dependant module information is retrieved by executing a new parser instance for each import directive and combining their output AST with the actual AST. Because the vanilla implementation uses static typing AST information can simply be merged by using accessors and modifiers for concrete node types. The ANTLR implementation uses a dynamic typing, which as earlier stated complicates tree processing and construction. Rather than manually inserting the imported module AST, ANTLR offers rewrite rules.

Rewrite rules describe the desired structure of an AST, based on production symbols. Each alternative, whether it is in a subrule or in the outermost rule level, is able to specify a rewrite rule. Rewrite rules always begin with a single → symbol, as demonstrated in the *import* production of code fragment 9. Upon recognition of an import construct, the string return value *path* of production *moduleId*, is used to parse the dependant module. The resulting tree, an instance of CommonTree, is then respectively placed inside the import node.

```
imprt: 'import' moduleId ';'
  -> ^( 'import' moduleId
          { parseFile($moduleId.path) }
      ) ;
```

Code snippet 9: Import

3.3 Type-checker

Semantic analysers, here known as type-checkers, traverse the AST to verify its semantic correctness. The semantic violations are provided to WAEBRIC end-users as warning messages, without influencing the interpreter output. During semantic analysis the following checks are conducted:

1. **Undefined functions.** Function call f always requires a function f defined in the module or one of its transitive imports, or f should be a tag defined in the XHTML 1.0 Transitional
2. **Arity mismatches.** Function calls should always contain an equal amount of arguments as the corresponding function definition
3. **Duplicate function definitions.** Function definitions can never have the same name
4. **Undefined variables.** Variable expression x should always be traceable back to an enclosing let-definition or function parameter
5. **Non-existing module reference.** Import directive i requires a corresponding file *i.wae*

3.3.1 Vanilla

Syntax tree traversal in the vanilla implementation is, as traditionally, done using the visitor pattern [GHJV95]. The visitor pattern allows type-safe tree traversal through double dispatch. Each visitor implements from a common interface *V*, that contains a visit(N) method for each AST node *N*. Inside each *N* an accept(V) method is specified for invoking the visitors. Traversal strategies are specified inside the visit methods, which offers flexible traversals but also entangles navigation code with concrete visitor functionality. Code snippet 10 shows a fragment of the vanilla type-checker, where variable references are checked for variable definition exceptions.

Using the visitor pattern seperates language processing algorithms from the syntax tree data structure. Language processing functionality, such as type-checking, is implemented in concrete visitors. This seperation of concerns for one offers the ability to add and modify processing logic without having to modify the data structure. Nevertheless, the visitor pattern creates a tight coupling between interface V and the data structure. Changes to the data

```
public void visit(VarExpression expr) {
  if(! isDefinedVariable(expr.getId())) {
    exceptions.add(
      new UndefinedVariableException(
       expr.getId()));
  }
}
```

Code snippet 10: Visitor pattern

structure automatically require modifications to V and its concrete visitor implementations. Because WAEBRIC is a stable language, which is unlikely to change, this was not considered a problem.

3.3.2 ANTLR

ANTLR offers type-safe traversal of dynamically typed trees by using tree grammars. Inside this tree grammar an AST structure is formally described using a grammatical notation similar to the parser grammar. To execute actions on specific nodes, action code can be embedded inside the related production. Based on a tree grammar, ANTLR generates a tree walker which on execution performs a depth-first tree traversal and any specified actions.

Code snippet 11 shows the type-checking of variable expressions, by using tree grammars. First, a variable expression is formally described as production *varExpression*, which consists of a single NT symbol IDCON. Whenever the currently traversed AST node matches *varExpression*, the action code between curly bracers is executed. Inside the action code production symbols are accessable by $*name*.

```
varExpression: IDCON {
  if(! isDefinedVariable($IDCON)) {
    exceptions.add(new
      UndefinedVariableException($IDCON));
  }
} ;
```

Code snippet 11: Tree grammar

3.3.3 Scoping

WAEBRIC allows the definition of variables and functions, that can again be referenced by variable expressions and function calls. The enclosing context in which a definition can be referenced is called a *scope*. Scopes contain dictionary data structures, known as *symbol tables*, which maps identifiers to a definition. Because WAEBRIC allows both variables and functions to be defined, two symbol tables need to be used. Scopes in WAEBRIC are stackable, where a reference is considered valid incase its identifier is contained in any of the scopes.

Scoping in the vanilla implementation is handled by the *Environment* class. This environment consists of a variable- and function table, including accessors and modifiers, and an optional parent environment. When a definition cannot be found in the current environment, the parent environment is checked. This hierarichal structure allows scope stacking, while maintaining a simple interface.

ANTLR supports a scoping meganism which automatically generates a scope stack, where symbol tables are declared using Java syntax. Code snippet 12 shows the definition of scope *Env*, containing a symbol table for variables and functions. ANTLR then generates an *Env* specific scope stack.

```
// ANTLR
scope Env {
  Map<String, Integer> vars;
  Map<String, Integer> funcs;
}

// Generated JAVA
Stack<Env_scope> Environment_stack;
static class Env_scope {
  Map<String, Integer> vars;
  Map<String, Integer> funcs;
}
```

Code snippet 12: Scope mechanism

Scopes are assignable to production rules. Whenever a production that uses this *Env* scope, such as the *eachStatement* specified in code snippet 13, is recognized a new scope instance is pushed into the stack and popped after traversing the production. Before using the symbol tables, they need to be manually initialized in Java syntax. By specifying initialization code inside the @init construct, tables are initialized before the production is traversed.

Inside the action code symbol tables can directly be accessed by $Env::*name*. Code snippet 14 speci-

```
eachStatement
  scope Env;
  @init {
    $Env::funcs = new HashMap();
    $Env::vars = new HashMap();
  } : ... ;
```

Code snippet 13: Production scope

fies a function which checks for function definitions, based on an identifier. Each scope is checked using a for-loop, starting at the latest defined scope.

```
boolean isDefinedFunc(String name) {
  for(int i=$Env.size()-1; i>=0; i--) {
    if($Env[i]::funcs.containsKey(name)) {
      return true;
    }
  } return false;
}
```

Code snippet 14: Scope accessor

3.3.4 Challenges

Unordered function definitions

WAEBRIC allows functions to be defined in any order. Meaning that function *f1* may call function *f2*, eventhough *f2* is defined later. Unordered function definition cause no difficulties in the vanilla implementations, as any function definition can be retrieved in advance from AST node *module*. Tree grammars, on the contrary, perform a depth-first traversal where a function call is possible to be traversed earlier then its definition. In order to support unordered function definitions, an additional tree walker, refered to as the loader, was made. This loader retrieves all function definitions and passes them to the type-checker and interpreter.

3.4 Interpreter

Interpretation is the final processing phase, where the syntactically and semantically correct AST is evaluated into one or more XHTML document(s). Document formatting and output is done using the JDOM[2] XML API.

[2]http://www.jdom.org

3.4.1 Challenges

Data passing

Sometimes the evaluation of a construct depends on the evaluation of its children, in these cases data passing is required. Not-predicates, for example, are evaluated as the negation of its sub-predicate evaluation. To evaluate not-predicates, evaluation information of the sub-predicate has to be passed.

Evaluations for the vanilla implementation are, as earlier stated, performed using the visitor pattern. In the visitor pattern an interface is specified that contains a visit(N) method for each node *N*. To provide convinient data passing, a generic return type *T* is specified for each visit(N). Figure 4 shows our global visitor design, consisting of the traditional visitor interface *IVisitor* and abstract class *NullVisitor*. This *NullVisitor* minimizes interface overhead for concrete visitors by providing a base implementation for each visit(N), performing a depth-first traversal and returning **null**. Visitors extend from *NullVisitor*, using a concrete type *T*, overwritting any methods relevant to their purpose.

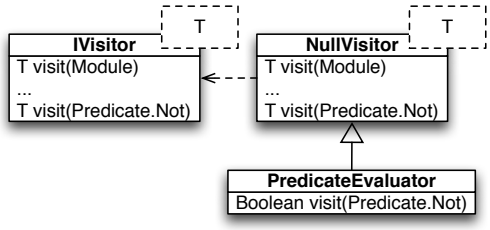

Figure 4: Generic visitor

To minimize the casting of return types, each evaluation type has a seperate visitor implementation. Code snippet 15 displays the *PredicateEvaluator*, which evaluates predicates into booleans.

```
class PredicateEvaluator
    extends NullVisitor<Boolean> {
  @Override
  Boolean visit(Predicate.Not not) {
    return ! not.getPred().accept(this);
  }
}
```

Code snippet 15: Generic return type

Inside, the evaluation of not-predicates is specified. Not-predicates are evaluated by negating its sub-predicate evaluation, retrieved by calling *accept* on the argumented AST node.

Each node has an *accept* method that, as earlier mentioned, is used to invoke the visitor using dynamic dispatch. Data passing is made convinient by letting the *accept*, specified in node N, return an instance T, retained from invoking visit(N). Code snippet 16 shows the *accept* implementation, which specifies a return type T using generic methods.

```
<T> T accept(IVisitor<T> visitor) {
  return visitor.visit(this);
}
```

Code snippet 16: Generic accept method

A tree grammar offers common return and parameter constructs to pass information. Code snippet 17 shows the evaluation of predicates using tree grammars. Inside this snippet, the production *predicate* defines a boolean return value *eval*, that is computed inside the action code. Evaluation information of sub-predicate p is accessable by $\$p.eval$.

```
tree grammar WaebricInterpreter;

predicate returns [Boolean eval] :
  '!' p=predicate { $eval = !$p.eval; }
  ... ;
```

Code snippet 17: Return construct

Both approaches offer an elegant solution, as evaluation information is maintained locally. Another approach would be to maintain a class variable for each evaluation type. Class variables make the code harder to understand and error prone as the variable can be modified from anywhere in the class.

Control flow structures

Tree walkers perform a depth-first tree traversal, which is insufficient for implementing a branched control flow structure. In an if-statement, for example, the sub-statement should only be executed when its predicate evaluates true. Tree walkers traverse the sub-statement regardless.

Fortunately, traversal over the sub-statement can be skipped by replacing NT symbol *statement* with a wildcard '.' symbol, as proposed in [3]. Wildcard symbols tell the tree walker to match the next node, and its children, causing no action code to be executed. After traversing the if-statement, shown in code snippet 18, the predicate will be evaluated. Incase *eval*, a boolean return value specified in production *predicate*, is true the sub-statement will be traversed. The sub-statement is traversed by telling the tree walker to go to the tree location where the sub-statement begins using `input.seek(int)` and invoking the generated function *statement*.

```
ifStatement
  @init{ int ti = 0; }
  : ^( 'if' predicate
    { ti = input.index(); } . ) {
      int curr = input.index();
      if($predicate.eval) {
        input.seek(ti);
        statement();
        input.seek(curr);
       }
    } ;
```

Code snippet 18: If-statement

Nevertheless, using wildcards and seek functions to implement control flow structures, results in a program that is harder to understand, debug and verify. Index information also needs to be maintained per branch, which adds to the complexity.

3.4.2 Yield statement

WAEBRIC offers a yield statement to parameterize functions with additional markup. These markup arguments will be output where, one or more, yield statement(s) are placed. A simple WAEBRIC program using yield statements might look as follows:

```
def main() func title "hello" ; end
def func() html yield ; end
```

Function *main* describes a markup statement, that contains a function call *func*, HTML tag *title* and textual expression "hello". Because the function *func* contains a yield statement, all markups placed behind function call *func* are parameterized. Inside

[3] http://www.antlr.org/wiki/display/CS652/Tree-based+interpreters

function *func*, the output of yield is now equal to TITLE "HELLO". Which leads us to the evaluation:

```
<html><title>hello</title></html>
```

Evaluation of the markup statement FUNC TITLE "HELLO" starts at the first markup element, function call *func*. Before evaluating function call *func* we check if function *func* contains any yield statements. According to the above specified definition it does, so the markup statement remainder, TITLE "HELLO" in this case, is parameterized. During the evaluation of *func*, each yield statement will trigger the evaluation of markup chain TITLE "HELLO".

After interpreting function call *func*, remainders of the markup statement TITLE "HELLO" should not be evaluated again, as they were parameterized. In the vanilla implementation this is simply done by abruptly ending the markup statement evaluating with a **return** construct. In a tree grammar this is more complicated, as the correct index value has to be set for the next construct to be traversed. Because markup statements got an arbitrary number of markups, its end index is difficult to determine. To simplify the skipping of markup statement remainders, we rewritten markup statements into a nested AST hierarchy, as shown in code snippet 19.

```
markupStatement: markup markupChain
  -> ^( MARKUP_STM markup markupChain ) ;

markupChain: expression ';'
  -> ^( MARKUP_CHAIN expression )
  | ...
  | markup markupChain
  -> ^( MARKUP_CHAIN markup markupChain ) ;
```

Code snippet 19: Rewrite nested hierchy

With the specification of rewrite rules, markup statement FUNC TITLE "HELLO" is parsed into the AST shown in figure 5. This nested tree hierchy, allows markup statement remainders to be skipped, from anywhere in the markup chain, by matching a single wildcard symbol. Code snippet 20 shows how the detection of a yield statement causes the remainder of a markup statement to be matched, by calling the *matchAny* wildcard function. Then the **return** construct is invoked to end markup statement evaluation prematurely.

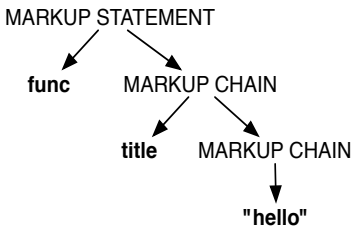

Figure 5: Markup chain

```
markupChain:
  ^( MARKUP_CHAIN markup {
      if($markup.yield) {
        matchAny(input); // Wildcard
        match(input, Token.UP, null);
        return retval; // End traversal
      }
    } markupChain ) ... ;
```

Code snippet 20: Markup remainder skip

In order to build a nested AST, backtracking had to be enabled. LL(*) parsers are unable to process tail recursion, or recursive rule invocation, where the last element of an alternative in the body of production p rewrites back to p. This is because generation of the DFA would invoke an infinite recursion. Enabling backtracking likely results in worse performance, but simplifies the implementation. Another approach would be storing the end index of each markup chain and using seeks, which makes the implementation more difficult to understand.

4 Results

4.1 Functionality

Both implementations are fully compliant to the reference implementation, scoring 100% acceptance on the functionality suite.

4.2 Maintainability

4.2.1 Lexer and parser

Maintainability results of both parser and lexer implementations are shown in table 2 and 3, where G stands for grammar, AC for action code and PC for

pre-amble code. Where pre-amble code is Java syntax positioned outside of a production rule. Java metrics were obtained by using Metrics[4] and Testwell CMT[5] tools. Because no ANTLR metric tools exists, grammar metrics were calculated by hand.

	Lexer		Parser	
Metric	Core	Token	Core	AST
NCLOC	291	235	1025	2046
MLOC	214	64	717	569
MCC	3.81	1.06	2.94	1.03
E	$1.3E^6$	$2.7E^5$	$2.2E^6$	$1.9E^6$
MI	99	137	100	141
#M	26	49	79	514
AVS	8.2	1.3	9.1	1.1
#C	2	11	13	88

Table 2: Java maintainability results

	Lexer			Parser		
Metric	G	AC	PC	G	AC	PC
NCLOC	29	14	4	98	8	15
MLOC	25	14	0	83	8	12
MCC	3.28	1	1	4.04	1.5	2
E	$1.9E^4$	1567	76	$4.1E^4$	2043	5692
MI	121	143	134	106	127	101
#M	29	8	0	23	2	1
AVS	5.24	1.8	0	10.87	3.5	12
#C	n.a.	0	0	n.a.	0	0

Table 3: ANTLR maintainability results

Both the NCLOC, MLOC and E are significantly in favor of ANTLRs grammatical approach. Due to the expressiveness of EBNF notation in comparison to verbose Java, roughly 10 times fewer NCLOC are needed to implement syntax, which supports the findings earlier mentioned in [MHS05]. MCC, which measures the average number of flows threw methods, is lower in the vanilla parser because it specified more methods, having a unique parser method per alternative. The ANTLR lexer has fewer MCC and AVS due to its expressive token pattern notation. MI indicates that both implementations are maintainable, as both score above above 85 [OH94], but ANTLR is clearly in favor.

Another distinctive result is that most vanilla NCLOC were measured in the AST package. This is due to the static typing, where each production has a unique AST node class and production alternative are embedded as sub-class. Node classes are purely used as model, consisting of fields, accessors and modifiers. Thus, most NCLOC are likely spent on class and function definitions, which explains the significantly lower MLOC. Because accessors and modifiers can often be generated by an IDE, the specified E is not entirely representative.

[4]http://metrics.sourceforge.net

[5]http://www.verifysoft.com/en_cmtx.html

4.2.2 Checker and interpreter

Table 4 and 5 features our maintainability results of both the checker and interpreter. Unlike the parser, there are no significant differences in MLOC. Infact, the ANTLR checker contains more MLOC than the vanilla checker. This is because tree grammars are forced to respecify the entire syntax tree structure, resulting in significant amounts of duplicate code. Due to the, earlier mentioned, *NullVisitor* the vanilla checker only has to override any relevant visit methods, which also explains the lower #M. Because the interpreter requires more Java code, the disadvantage of duplicate grammar code is less obvious. The ANTLR interpreter has fewer MLOC, as its navigation code for depth-first traversals is specified more expressive in EBNF notation.

	Checker		Interpreter
Metric	Core	Scope	Core
NCLOC	211	79	734
MLOC	106	35	493
MCC	1.44	1.63	2.03
E	$2.0E^5$	$1.3E^5$	$2.1E^6$
MI	110	127	103
#M	18	16	68
AVS	5.9	2.2	7.3
#C	7	1	5

Table 4: Java maintainability results

	Checker			Interpreter		
Metric	G	AC	PC	G	AC	PC
NCLOC	85	35	84	125	185	164
MLOC	72	35	42	112	185	109
MCC	2.75	1.6	1.75	2.96	2.7	2.41
E	$2.2E^4$	$6.9E^4$	$1.5E^5$	$5.4E^4$	$8.0E^5$	$3.5E^5$
MI	117	120	118	113	96	106
#M	24	10	12	23	18	17
AVS	6.54	3.5	3.5	6.74	10.3	6.4
#C	n.a.	0	6	n.a.	0	1

Table 5: ANTLR maintainability results

To normalize the aspect of language mixture in the ANTLR implementation, ANTLR grammars and embedded Java syntax were considered as one language. Table 6 shows the global metric results for the ANTLR implementation, where E is calculated by combining operators and operants of both the grammar, action code and pre-amble code.

These combined metric results show that the vanilla implementation requires significantly fewer E, and has a lower MCC. The reason that ANTLR

scores worse on these metrics is directly relatable to the earlier mentioned duplicate code in tree grammars. MI indicates no significant differences between the maintainability both implementations.

Metric	Lexer	Parser	Checker	Interpr.	Loader
NCLOC	47	121	204	474	132
MLOC	43	102	149	406	105
MCC	2.78	3.77	2.24	2.72	2.47
E	$4.1E^4$	$1.6E^5$	$6.7E^5$	$2.9E^6$	$1.4E^5$
MI	123	103	112	100	116
#M	37	26	46	58	36
AVS	4.12	10	4.97	7.94	5.13
#C	0	0	6	1	0

Table 6: ANTLR global maintainability

4.3 Performance

Performance results are displayed in figure 6. The numeral presentation can be found in Appendix A. Benchmarks were executed on a MacBook Pro 5.1 with the following specifications: Intel Core 2 Duo (2.4 GHz), 4GB DDR3, using Mac OSX Leopard, SUN JDK 1.6 and ANTLR 3.1.3.

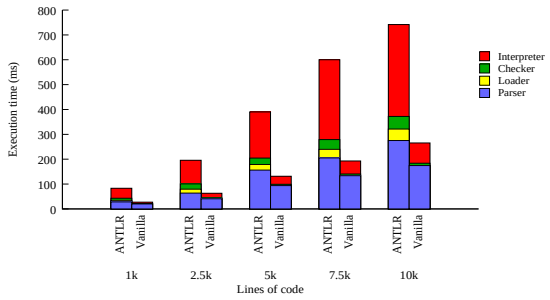

Figure 6: Performance

Each performed benchmark, regardless of program size, is won by the vanilla implementation. In the vanilla parser we measured a consistent speedup of 1.5 over ANTLR. Both approaches use backtracking to recognize markup statements, so any performance differences must be caused by the parsing technique. Tree grammars are consistently outperformed by a factor of 5, this is due to the additional syntax tree validation during traversal.

4.4 Usability

Code snippet 21 shows the error message obtained from parsing the following WAEBRIC program:

```
module test;
def func1
def func2 end
```

Both error messages contain information about the line number, problem and possible solution. What error message is better remains largely subjective, however the vanilla error message provides additional context information, such as "function end" instead of "END", which should make the error easier to understand for end-users. In case a specific error format is desired, error handling functionality can be overwritten in action code. The semantic error are presented similar for both implementations.

```
// ANTLR
line 3:0 missing END at 'def'

// Java
Unexpected token found: "def" KEYWORD
(line: 3, character: 0). Was expecting
a function end, use the expected "end".
```

Code snippet 21: Error messages

5 Evaluation and discussion

To determine whether the usage of ANTLR results in a higher quality WAEBRIC implementation than vanilla Java, we decomposed our evaluation in the earlier mentioned quality attributes: functionality, maintainability, performance and usability.

5.1 Functionality

Both approaches obtained a full conformance, as expected in our earlier stated hypothesis. However, we experienced several notable differences while debugging. Using ANTLR to develop parsers clearly introduces fewer bugs. Inconsistencies in the grammar are automatically checked for during generation, eliminating most programming errors. Bugs can still be introduced by specifying incorrect WAEBRIC syntax, but in our case the grammar could almost entirely be copied from our documentation, which leaves little space for mistakes.

Both checker and interpreter implementations introduced a similar amount of bugs, as both use Java syntax to specify semantics. Debugging was harder

in ANTLR, due to the dynamic syntax tree typing, which is harder to read, eventually making it more difficult to determine what is causing the defect.

5.2 Maintainability

Developing a parser using ANTLR results in a more maintainable implementation then vanilla, supporting our hypothesis. Due to the EBNF notation, which is more expressive then Java, implementations are significantly shorter and easier to understand. ANTLR grammars specify languages on a higher abstraction level, where developers do not require any extensive language development or parsing knowledge to do their jobs. However, in order to use ANTLR, developers have to learn the ANTLR grammar language. To reduce this learning curve and enhance the productivity of grammar development, ANTLR offers "ANTLRWorks", an IDE providing syntax highlighting, path visualization and debugging functionality on grammars.

ANTLR tree grammars provide and innovative approach, with much potential, to traversing syntax trees. Because the tree grammar language itself is a DSL, specifically designed towards traversing syntax trees, significant gains can be made in expressiveness. Data passing, for example, is handled elegantly by using common return and parameter constructs. These constructs can, as earlier stated, be somewhat simulated in the visitor pattern with generics, but lead to extensive code overhead in the node and visitor classes. Scoping functionality is also supported by default, which saves the developers from making any related design decisions, and thus reducing effort. Nevertheless, the ANTLR checker and interpreter were measured less maintainable then their vanilla implementations.

Causes that tree grammars score worse on maintainability are directly related to the amount of duplicate code. Each tree grammar is forced to respecify the entire syntax tree in order to validate its structure, as argumented in [6]. This is, of course, a questionable design decision, because the parser is already responsible for validation. Regardless, the more maintainable solution would be to let tree grammars extend from a parser grammar, allowing the attachment of action code by using a **super** construct, similar to the Ometa[7] parser generator. By using inheritance, syntax tree specifications are centralized, which should lead to a less error-prone and easier to maintain implementation.

In the earlier mentioned maintainability results language mixture is disregarded, which would favor the vanilla implementation even further. ANTLR grammars consist of grammatical EBNF notation and embedded Java syntax. Attaching Java code to your implementation offers a great amount of functionality, but should also be used with caution as it reduces readability. Switching between languages, formally known as "context switching", is expensive, making the implementation harder to understand. How much the maintainability is reduced by context switching remains uninvestigated, but the frequency of language swaps should be minimized.

Tree grammars provide a clean separation between action code, described in Java syntax, and navigation code, specified using EBNF. Nevertheless, the depth-first traversal strategy of tree grammars is insufficient for interpreting control flow structures, which require "hacked" navigation functionality entangled between action code in order to work. To make tree grammars beneficial for more complicated languages, ANTLR should allow alternative traversal strategies to be specified more elegantly, preferably without the use of action code.

One of the clear advantages that tree grammars offer is that the syntax tree data structure and visitor interface do not have to be developed and maintained, which saves a significant amount of effort. Tree grammars are currently an excellent solution for simple tree manipulation and regular depth-first interpretations, when optimal performance is not required. By supporting language specific traversal strategies and reducing duplicate code, tree grammars could well compete with the visitor pattern.

5.3 Performance

Performance benchmarks show that the ANTLR implementation is consistently slower then vanilla, regardless of program size and processing phase, which rejects our hypothesis. Causes that ANTLR performs worse than expected seems to be related to its parsing algorithm and additional validation during tree traversals. Regardless, the differences in execution time are so slim that they should be barely noticeable by end-users.

[6] http://antlr.org/article/1100569809276
[7] http://tinlizzie.org/ometa

5.4 Usability

Between both implementations, the error messages showed a high amount of similarities. Both error messages display the line number, related token and possible solution. Determining what error message is better remains a subjective activity. In this study we considered the vanilla error messages better, because they provide additional context information and are presented in a, to us, more readable format, which supports our hypothesis.

5.5 Threats to validity

During our study we encountered several threats to validity, these are discussed below.

5.5.1 Vagueness of quality attributes

Most quality attributes are subjective and cannot purely be determined based on objective metrics. Due to this vagueness, we will never be able to prove, with certainty, that any specific metric provides a correct indication of a quality attribute. Which in turn makes our conclusions disputable.

To maintain a certain quality, we've only made use of metrics that are widely accepted and used in literature. However, we also applied grammar metrics, which is a rather new field of research, first described in [PM04]. These grammar metrics are still used on a very limited scale, and little literature is available on the subject, which makes their correctness slightly uncertain. Instead of purely basing our conclusions on metrics, we have also taken personal experience into consideration.

5.5.2 Optimal implementation

In order to obtain a fair comparison between the approaches, both implementations need to be optimal. Implementations that are not optimal will result in worse metrics results, leading to incorrect conclusions. To optimize the vanilla implementation, we frequently discussed design decisions with language experts at the CWI. While using ANTLR, we made frequent use of the antlr-intrest [8] mailing list. Regardless of the taken precautions, we cannot be entirely certain of an optimal implementation.

[8] http://www.antlr.org/pipermail/antlr-interest

5.5.3 External validity

Results presented in this study are based on implementing WAEBRIC. However, it is disputable if these comparisons can be generalized to implementing any language. WAEBRIC was selected due to its similarities, both syntactical and semantic, with commonly used programming languages, making it representative to a large class of languages. In order to draw global conclusions, a wider variety of languages need to be implemented and analyzed.

6 Conclusion

Our work shows that generating a parser with ANTLR requires significantly fewer development- and maintenance effort, but result in slightly worse performance and error messages. Due to its expressive EBNF notation, our ANTLR parser required roughly 10 times fewer NCLOC, which supports the findings from [KLBM08]. ANTLR allows languages to be described on grammatical level, relieving the developers and maintainers from parsing techniques and architectural decisions, significantly reducing the amount of effort and knowledge required. We suggest ANTLR for parser construction in any situation where an optimal performance is not required and ANTLR errors are sufficient.

Using ANTLR tree grammars to implement a type-checker and interpreter is an innovative idea with much potential, but is not used optimally in its current condition. Tree grammars can significantly reduce the amount of effort required, as no syntax tree data structure and visitor interface have to be developed and maintained. Inside the tree grammar, elegant solutions are provided for data passing and scoping. However, its depth-first traversal strategy is insufficient for interpreting control flow structures, which requires action code hacking. Tree grammars also contain a significant amount of duplicate code, which makes the implementation more error prone and harder to maintain. We suggest tree grammars for simple tree manipulations and depth-first interpretations, in other situations we recommend sticking to the visitor pattern.

ANTLR parsers can be used together with a visitor implementation, by manually constructing a statically typed syntax tree in the action code.

Future work

Before any global guidelines can be composed, further research is required by implementing more languages and comparing different approaches. With this study we hope to trigger researchers in doing similar work, as much remains to be investigated.

Acknowledgements

First and foremost, thanks to Tijs van der Storm and Jurgen Vinju for providing excellent guidance. I would also like to thank my CWI-SEN1 colleagues for making the stay enjoyable and motivating. Last but certainly not least, special thanks to Michel de Graaf and Lars de Ridder for providing a constant stream of epic jokes and ridiculous remarks.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. 1986.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 143, Washington, DC, USA, 1998. IEEE Computer Society.

[BS86] R. Bahlke and G. Snelting. The psg system: from formal language definition to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, 1986.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-wesley professional computing series, 1995.

[Gri07] Robert Grimm. Declarative syntax tree engineering (or one grammar to rule them all). Syntax trees, 2007.

[Hal77] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[HB88] R.M. Herndon and V.A. Berzins. The realizable benefits of a language prototyping language. *IEEE Transactions on Software Engineering*, 14(6):803–809, 1988.

[Joh75] Stephen Johnson. Yacc: Yet another compiler compiler. *Computing Science Technical Report*, 32:1–35, 1975.

[KLBM08] Toma Kosar, Pablo Lopez, Pablo Barrientos, and Marjan Mernik. A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390–405, 2008.

[KMB+96] R. B. Kieburtz, L. McKinney, J. M. Bell, J. Hook, A. Kotov, J. Lewis, D. P. Oliva, T. Sheard, I. Smith, and L. Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th International Conference on Software Engineering*, pages 542–553, 1996.

[LST78] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[McC76] Thomas J. McCabe. A complexity measure. In *IEE Transactions on Software Engineering*, volume 2, pages 308–321, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[MHS05] Marjan Mernik, Jan Heering, and Antony Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.

[OH94] Paul Oman and Jack Hagemeister. Construction and testing of polynomials predicting software maintainability. *Journal of Systems and Software*, 24(3):251–266, 1994.

[Par07] Terence Parr. *The defnitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf, 2007.

[PM04] J.F. Power and B.A. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):405–426, 2004.

[vDK98] Arie van Deursen and Paul Klint. Little languages: little maintenance? *Journal of Software Maintenance: Research and Practice*, 10(2):75–92, 1998.

[vdS09] Tijs van der Storm. Waebric: a little language for markup generation. Language reference, 2009.

A Performance

Table 7 and 8 present a numeral representation of our benchmark results, discussed in section 4.3.

LOC	Parser	Checker	Interpreter	Total
1000	20.2	1.15	5.85	27.2
2500	40.72	3.72	18.66	63.1
5000	93.66	5.44	32.2	131.3
7500	133.73	6.73	52.33	192.79
10000	174.5	9.46	81.39	265.35

Table 7: Java execution times

LOC	Parser	Loader	Checker	Interpreter	Total
1000	28.09	6.46	8.34	40.33	83.22
2500	63.62	15.77	21.45	95.07	195.91
5000	156.2	22.62	25.69	185.66	390.17
7500	205.06	34.7	39.12	321.37	600.25
10000	265.07	46.08	51.03	369.34	731.52

Table 8: ANTLR execution times