University of Amsterdam

Faculty of Science

Master's Thesis

Two implementation techniques for Domain Specific Languages compared:

OMeta/JS vs. JavaScript

Author:
Nickolas Heirbaut

Host organization:
Centrum Wiskunde & Informatica

Date approval:
8 October 2009

Supervisor:
Dr. Tijs Van Der Storm

Contents

Abstract

During the development of a domain-specific language, little attention is given to its implementation. The variety of implementation techniques is however large including preprocessing, embedding, (extensible) compiler / interpreter, application generators and commercial off-the-shelf tools. Choosing a suitable implementation technique is essential as it can affect the total effort required to implement a DSL largely, not to mention end-user productivity, efficiency and maintainability. In most cases, trade-offs has to be made when choosing the proper technique. While these trade-offs are much discussed among implementors, they are compared over diverse application domains and without the support of empirical evidence making an objective comparison difficult, if not impossible.

The purpose of this paper is to provide an in-depth quality analysis for two implementation techniques with JavaScript as target language. The first implementation involves the application of traditional GPL interpreter techniques to develop a lexer, parser, typechecker and interpreter. The second implementation applies the compiler-generator technique where certain phases of the interpreter technique are automated using OMeta/JS - a tool particularly well-suited for building language implementations - to generate a lexer, parser and a typechecker based on a formal specification.

Comparison shows that the interpreter technique results in higher performance and functionality, while the OMeta implementation is more maintainable and requires significant less effort from the implementor. As a result, we propose to use the OMeta implementation for rapidly prototyping domain-specific languages or when error reporting is not important. The interpreter technique is best applicable for DSLs targeted to production environments or when high performance is desired.

The research provided in this thesis is a contribution to the DSL Benchmark Implementation project which aims to provide an extensive quality overview of various implementation strategies, each carried out in different programming languages. The project is an important addition to the field of domain-specific languages and improves and extends earlier work on DSL implementation techniques.

Preface

This thesis is the result of my graduation project at the Centrum Wiskunde & Informatica (CWI) in fulfillment of the thesis requirements for the degree of Master in Software Engineering at the University of Amsterdam.

In this preface I would like to thank everybody involved in the program Master Software Engineering for the valuable knowledge they provided me on the many aspects of software engineering. Special thanks to Prof. dr. Paul Klint for offering a chance to contribute to the DSL Benchmark Implementation Project at the CWI, and Dr. Tijs Van Der Storm for his valuable feedback and supervision during the entire project.

I would also like to thank the author of OMeta/JS, Alessandro Warth, for reviewing the OMeta/JS implementation and answering all my questions regarding OMeta.

Finally a note of thanks to my parents for the opportunity they gave me to follow this program and for their moral support throughout the program.

Nickolas Heirbaut,
Sint-Niklaas, September 2009

Chapter 1

Introduction

1.1 Motivation

Since the emergence of domain-specific languages (DSLs) in the late 50's (APT, BNF), many articles have been written on the development of DSL's. It is however only recently that research has been done into the implementation of DSLs including preprocessing, embedding, (extensible) compiler / interpreter, application generators and commercial off-the-shelf tools [21].

Choosing a suitable implementation technique is essential as it can affect the total effort required to implement a DSL largely, not to mention end-user productivity, efficiency and maintainability. In most cases, trade-offs has to be made when choosing the proper technique. While these trade-offs are much discussed among implementors, they are compared over diverse application domains and without the support of empirical evidence making an objective comparison difficult, if not impossible.

A recent study from Kosar et al. [16] has provided the first empirical results by comparing ten different DSL implementation approaches on a quantitative manner. In particular, implementation effort and end-user productivity was measured as a base for cost-benefit analysis. There are however other measures that should be considered when choosing a suitable approach such as the level of functionality, maintainability and efficiency. In response to this, the Centrum Wiskunde & Informatica (CWI) in Amsterdam started the DSL Benchmark Implementation project which aims to provide an extensive quality overview of various implementation strategies, each carried out in different programming languages. Each implementation strategy in this project implements the same reference DSL Waebric, a language for generating xHTML markup.

The research provided in this thesis is an addition to the DSL Benchmark Implementation Project and analyzes the quality of two DSL implementation techniques with JavaScript as target language. The first implementation involves the use of traditional

GPL interpreter techniques to develop a lexer, parser, checker and interpreter. The second implementation applies the compiler-generator technique which is similar to the previous technique except that certain phases are automated using language development tools or so-called compiler-compilers such as ANTLR, JavaCC, Yacc and OMeta. In this research, OMeta/JS is used to generate a lexer, parser and checker based on a formal specification similar to BNF notation.

1.2 Research questions

Based on the foregoing, the following research question is defined:

RQ1 How does the quality of a DSL implementation in OMeta/JS relates to the quality of a hand-written JavaScript interpreter?

To define the term *quality* in the previous research question, the following subquestions should be answered as well:

RQ1.1 How maintainable is each implementation in terms of lines of code, method size, McCabe complexity, Halstead Effort and Maintainability Index?

RQ1.2 How efficient is each implementation in terms of duration of regression tests?

RQ1.3 How functional is each implementation in terms of acceptance tests, supported notation and error reporting?

1.3 Organization of this thesis

The organization of this thesis is as follows. Chapter 2 briefly introduces the notion of domain-specific languages, the compiler-generator OMeta and the reference DSL Waebric. Chapter three provides an overview of the research method that was applied in the case study. The implementation of the interpreter and compiler-generator technique are discussed in chapter 4 and 5 respectively. Chapter 6 discusses the results extracted from each implementation whereas chapter 7 analyzes these results. At last, chapter 8 provides concluding remarks based on the analysis phase.

Chapter 2

Background and context

2.1 Domain-specific languages

2.1.1 Introduction

A domain-specific language (DSL) is a small, usually declarative language specially designed for a particular application domain. A general-purpose language (GPL) such as Java or C on the contrary aims to target many application domains. In [3], the term *little* languages was introduced to identify a DSL as a language specialized to a particular problem domain which does not include many features found in conventional languages. A more formal definition of the term DSL was proposed in [34]:

> *"A domain-specific language (DSL) is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."*

DSLs are also known as special purpose, little, small, specialized, task-specific or application languages. The notations provided in DSLs are tailored towards the application domain, offering substantial gains in expressiveness and ease of use compared with GPLs for the domain in question, with corresponding gains in productivity and reduced maintenance costs [1]. Therefore, DSLs can be used without the need for knowledge about general programming, enabling end-user programming. The most common DSLs are HTML (markup), YACC (language grammar), GraphViz DOT (graph drawing), SQL (database), Make (incremental rebuilds), VHDL (hardware design), LaTeX (typesetting), MATLAB (technical computing) and BNF (syntax specification).

An important benefit of a DSL is that it allows solutions to be expressed at the level of the problem domain, enabling domain experts to understand, validate, modify and often

even develop DSL programs [34]. This, together with enhanced productivity, reliability, reusability, maintainability [33][15] and portability [11] make DSLs an interesting way to solve software engineering problems. The costs however, for designing, implementing and maintaining a DSL are one of the main disadvantages since expertise is required in both domain and language development. More disadvantages includes the potential loss of efficiency compared with hand-coded software, the costs of education for DSL users and the difficulty defining language constructs and scope [34].

2.1.2 Development phases

DSL development includes several phases which are discussed by Mernik et al. [21] in detail. A brief overview is given below:

- **Decision phase**
  A first step in the development of a DSL involves the identification of software which is eligible for DSL use. Mernik et al. proposes several patterns that help the identification process such as product lines, task automation, data structure traversal, GUI construction etc. If found, then the decision has to be made by a quantitative treatment of the trade-offs. Key factor herein is the return on investment of the DSL as it should pay for itself by more economical software development and/or maintenance later. Beside economic interests, the level of domain knowledge and programming expertise of the (end-)users should be taken into account when choosing to develop and implement a DSL.
- **Analysis phase**
  During this phase, the problem domain is identified and domain knowledge is gathered. Inputs are various sources of explicit or implicit domain knowledge, such as technical documents, knowledge provided by domain experts, existing GPL code and customer surveys. Most information is gathered informally, but sometimes domain analysis methodologies are used (e.g. FODA [Kang et al. 1990]). If code extraction is used, then the domain knowledge is mined from legacy GPL code by inspection and/or by using software tools.
- **Design phase**
  It's in the design phase that the actual DSL is shaped based on the information gathered in the analysis phase. The easiest way to design a DSL is to reuse an existing language, possible benefiting from easier implementation and familiarity for users. Three patterns can be distinguished. With the *piggyback* design pattern, only a part of the existing language is reused, while the *specialization* pattern reuses the existing language but limits it. The third technique reuses and expands the existing DSL with new features that address domain concepts. The results of the design phase is a DSL description than can be specified either formally (i.e. semantic definition) or informally (i.e. natural language, illustrative).
- **Implementation phase**
  In the fourth phase, the formal or informal specification is implemented. Care should be taken when choosing a suitable implementation as it can affect the total effort required to implement a DSL largely. The patterns identified by [Mernik] includes (extended) compiler/interpreter, application generator, preprocessor, embedding, Commercial Off-The-Shelf (COTS), or a combination of them.

2.1.3 Implementation strategies

The previous section introduced all phases of the DSL development process. In this section, a more closer look is given to the implementation phase based on earlier work of Spinellis [31] and Mernik et al. [21].

Interpreter/compiler

In an interpreter/compiler approach, the DSL is implemented in a general-purpose language (GPL) "from scratch" using standard GPL interpreter/compiler techniques. Interpreting or compiling includes phases such as lexical, syntactic and semantic analysis. Since the implementation is performed "from scratch", the technique is sometimes called the "vanilla" approach.

In the case of the interpreter, the DSL constructs are *recognized* and *interpreted* using standard fetch-decode-execute cycle. Interpreters are mostly used if execution speed is not important or if the language has a dynamic character such as JavaScript, Ruby or Python. Interpreters have the advantage of greater simplicity, greater control over the execution environment and easier extension over compilers.

If the compiler approach is applied, then constructs are *translated* to base language constructs and library calls, allowing complete static analysis on the DSL program/specification.

Note that Spinellis [31] argues that the development of DSLs is radically different from GPL development and therefore didn't included these techniques in his research. In [21] and [16] however, it is observed that such techniques are widely used in practice and have adopted these standard GPL techniques in their research. In this research, the observations of [21] and [16] are adopted as well.

Compiler generator

The compiler generator technique is similar to the compiler/interpreter technique, except that one or more phases are implemented using language development tools or so-called compiler-compilers or compiler-generators. Many phases of the compiler are similar to those of the interpreter, making compiler-generators applicable for interpreters as well. The implementation effort can be reduced since one or more phases are automated using tools, thus minimizing the disadvantages of standard interpreter or compiler techniques. Most common form of compiler generators are parser generators using BackusNaur Form (BNF) notation. Compiler generators are numerous and widely used such as ANTLR, Lex/Yacc, JavaCC and OMeta.

Extensible compiler/interpreter

The extensible compiler/interpreter technique extends traditional GPL compilers or interpreters with domain-specific optimization rules and/or domain-specific code generation. Interpreters tends to be easier extensible then compilers. However, extension of either one is not an easy task as the extensions usually turns out to be a very difficult and expensive because conventional compilers often lack extensibility and reusability [39]. Furthermore, caution is required on domain-specific notations as it can interfere with existing ones. Nevertheless, if extension is done properly, the implementor can benefit from the reuse of the complete existing compiler infrastructure. Examples of extensible compilers are Polyglot and JastAdd.

Preprocessing

The preprocessing technique translates the DSL to constructs in the existing language (the base language). The amount of preprocessing mainly depends on the nature of the preprocessor. Four subpatterns are considered:

- **Macro processing**
  Translation is performed using macro definitions which converts certain strings (macro calls) into special output. Examples are M4, GEMA, gpp and SNOBOL for general-purpose or CPIA for XML and HTML.
- **Source-to-source transformation**
  The DSL source code is transformed (translated) into base language source code. Program transformations can be defined as manual procedures or with the use of program transformation systems such as Stratego/XT, TXL, DMS and ASF+SDF.
- **Lexical processing**
  Lexical scanning is applied when only simple lexical scanning required, without the use of complicated tree based syntax analysis. Lexical processing is the lowest-level preprocessing. An example of it use is the textual inclusion of files. C preprocessor is the most common example of lexical processing.

  Due to the absent of semantic analysis, implementation is very easy. This is also the main disadvantage as static checking and optimizations at domain level are impossible. Static analysis on the other hand is limited to that done by the base language processor resulting in error reporting in terms of base language concepts.
- **Pipeline**
  The pipeline processor handles the sub languages of a DSL and translates them to the input language of the next stage.

Embedding

The embedded approach uses existing mechanisms in the GPL to build a library of domain-specific operations by defining new abstract data types and operators. This

technique is advisable when the DSL is both syntactically and semantically a subset of an existing host language. The advantage is that the compiler or interpreter of the GPL can be reused. Conversely, a less closer syntax notation and problematic error reporting is obtained. This approach has gained significant popularity in the functional programming community [12].

Commercial Off-The-Shelf (COTS)

Existing tools and/or notations are applied to a specific domain. For example, the general-purpose Powerpoint tool has been applied in a domain-specific setting for diagram editing [38]. XML is another example where a DTD or XML scheme are used to specify the grammar of the DSL. By building around existing tools, the effort required to implement the DSL can be reduced. In an XML setting for instance, the DOM and SAX parser can be used for parsing while XLST can be used for code transformation.

Hybrid

The hybrid method uses a combination of the previously discussed approaches in order to implement the DSL.

2.2 OMeta

2.2.1 Introduction

OMeta is an *object-oriented language* designed by Alessandro Warth particularly well-suited as a language development tool to create tokenizers, parsers, visitors and tree transformers. OMeta can be considered as a parser-generator and is therefore applicable in a compiler-generator approach. OMeta's key insight is the realization that all of the passes in a traditional compiler are essentially pattern matching operations [36]:

- a lexical analyzer finds patterns in a stream of characters to produce a stream of tokens;
- a parser matches a stream of tokens against a grammar (which itself is a collection of productions, or patterns) to produce abstract syntax trees (ASTs);
- a typechecker pattern-matches on ASTs to produce ASTs annotated with types;
- more generally, visitors pattern-match on ASTs to produce other ASTs;
- finally, a (naive) code generator pattern-matches on ASTs to produce code.

Figure 2.1 shows a grammar written in OMeta syntax that *recognizes* simple arithmetic expressions.

```
1  ometa Calculator{
2    digit    = '0' | ... | '9',
3    number   = digit+,
4    addExpr  = addExpr '+' mulExpr
5             | addExpr '-' mulExpr
6             | mulExpr,
7    mulExpr  = mulExpr '*' primExpr
8             | mulExpr '/' primExpr
9             | primExpr,
10   primExpr = '(' expr ')'
11            | number,
12   expr     = addExpr
13 }
```

Figure 2.1: OMeta Recognizer

2.2.2 Parsing Expression Grammar

OMeta is based on Parsing Expression Grammars (PEGs) which are used to describe formal languages in terms of sets of rules for *recognizing* strings in languages [8]. The

language of parsing expressions is shown in Table 2.1. Formally, a parsing expression grammar is described as a quadruple $G = (V_N, V_T, R, e_S)$ where V_N and V_T are disjoint sets of symbols known as non-terminals and terminals respectively, R is a finite set of parsing rules (production rules) of the form $(V_N \rightarrow R)$ and e_S is known as the start symbol expression ($e_S \notin V_N - V_T$).

Notation	Expression
Sequencing	$e_1\ e_2$
Prioritized choice	$e_1 \mid e_2$
Repetition ≥ 0	$e*$
Repetition ≥ 1	$e+$
Grouping	(e)
Negation	e
Look-ahead	$\&e$
Rule application	r
Character matching	$'x'$

Table 2.1: Inductive definition PEGs
e is a parsing expression and r is a non-terminal

Terminal symbols are literal strings forming the input of a formal grammar which cannot be broken down into smaller units using the rules of the grammar. By contrast, symbols that can be broken down into other symbols are called *non-terminal symbols*.

```
1 mulExpr = mulExpr '*' number
```

Figure 2.2: Production rule with one terminal and three non-terminals

The production rule shown in figure 2.2 has one terminal ('*') and three non-terminals (mulExpr, mulExpr and number).

Ordered choice operator

PEGs are stylistically similar to context-free grammars (CFGs) much like Extended Backus Naur Form (EBNF) notation. The fundamental difference is that a PEG's choice operator is ordered whereas CFGs use nondeterministic choice between alternatives. The choice operator lists alternative patterns to be tested in order, unconditionally using the first successful match. This is a very desirable property because it promises a unique interpretation of each parsing rule in the language resulting in an unambiguous grammar. Consider the following example with two production rules:

```
1 A -> a b | a
2 A -> a | a b
```

Figure 2.3: Illustration of prioritized choice in OMeta

Both production rules are similar but the choice operator is positioned differently. For CFGs, both production rules are identical. If the input to be recognized is 'a b' in the given example, then line 1 matches the first alternative whereas line 2 matches the second alternative. For PEGs line 1 and 2 are differently, moreover, the second alternative will never succeed because the first choice is always taken if the input string to be recognized begins with an 'a'. Thus, when 'a b' is used as input, then line 1 matches the first choice whereas line 2 also matches the first choice (since the input starts with 'a') but will fail because 'b' was not expected in that choice.

Semantic actions

A recognizer as presented in figure 2.1 can be transformed into a parser by adding semantic actions. These are specified using the → operator and written in OMeta's host language, usually the language in which OMeta implementation was written[1]. An action is a function or function object that is called if a match is found in the particular context where it is attached. In other words, the semantic action is executed if the corresponding rule matches the input.

An illustration of semantic actions is provided in figure 2.4 where JavaScript code is added to the semantic action of the production rule. The semantic action consists of an array whose first element is a string (e.g. 'add') and whose other elements are sub-trees (e.g. x). Note that the results of the non-terminals *exp* and *fac* are bounded to an identifier, x and y respectively, and are then accessed in the semantic action code. If no semantic action code is provided, then the value from the last successful match is returned. The semantic action code allows the creation of any kind of object beside arrays enabling the creation of user-defined Abstract Syntax Trees (ASTs).

```
1 addExpr = addExpr:x '+' mulExpr:y → ['add', x, y]
2           | addExpr:x '-' mulExpr:y → ['sub', x, y]
3           | mulExpr
```

Figure 2.4: Semantic Actions in OMeta

[1]OMeta has been implemented in JavaScript, Squeak Smalltalk and COLA supplemented with third-party implementations in C#, Python, Scheme, Lisp and Factor. All examples provided in this thesis are written in OMeta's JavaScript implementation, referred as OMeta/JS.

Semantic predicates

OMeta also supports semantic *predicates* used to determine the validity of applying a production. They are written in the host language and have access to the binded identifiers as well. An example is shown in figure 2.5 which describes a digit whose input has to be a character ranging from zero to nine. In JavaScript, the evaluation of a character is done using its ASCII character code value.

```
1 digit = char:d ?(d >= '0' && d <= '9')
```

Figure 2.5: Semantic predicate

2.2.3 OMeta: an extended PEG

The standard functionality given by PEGS has been extended in OMeta in a number of areas. OMeta not only operates on a stream of characters, it also allows pattern matching on *arbitrary datatypes* such as lists and strings. This make OMeta a suitable language for implementing transformations on unstructured data (e.g. parsers) as well as structured data (e.g. visitors)[35]. Figure 2.6 shows an interpreter written in OMeta performing pattern matching on lists which allows arithmetic operations. The corresponding input for this interpreter is provided by the output of the CalcParser in figure 2.1. It is also possible to combine a parser and interpreter by adjusting the semantic code to perform arithmetic operations immediately, skipping the parsing step.

```
1 interp = ['num' anything:x] -> x
2          ['add' interp:x interp:y] -> (x + y)
3          ['sub' interp:x interp:y] -> (x - y)
4          ['mul' interp:x interp:y] -> (x * y)
5          ['div' interp:x interp:y] -> (x / y)
```

Figure 2.6: Pattern matching on lists

Parameterized rules

Parameterized rules are another neat extension in OMeta which allows to define generic production rules by taking one or more arguments. A parameterized rule consists of a non-terminal extended with one or more parameters which are linked to a terminal or non-terminal in the grammar. Parameterized rules can used to create features that are not supported by the language. For example, OMeta does not support regular expressions but it can be mimicked using parameterized rules. Figure 2.7 shows a regular-expression-style character class using parameterized rules (cRange) enabling the

reuse of the semantic predicate.

```
1 digit           = cRange('0', '9'),
2 upperCaseLetter = cRange('A', 'Z'),
3 lowerCaseLetter = cRange('a', 'z'),
4 cRange :x :y    = char:c ?(c >= x && c <= y)
```

Figure 2.7: Parameterized rules

Higher-order rules

OMeta also provides a mechanism for implementing higher-order rules, i.e. rules that take other rules as argument. Higher-order rules are used in combination with parameterized rules and force the production rule passed in to be matched to the grammar. An example is given in figure 2.8 to create lists using a separator (e.g. apple;lemon;pineapples). The first production rule uses the non-terminal *listOf* and passes the non-terminal *fruitname* and the semicolon through. The third production line defines the non-terminal *listOf* with two arguments, *rule* and *sep*. The production body then applies the first argument (fruitname) to match the corresponding production rule given at line 2. Optionally, the semicolon separator and non-terminal fruitname are consumed and applied. This can be repeated unlimited. The separator token is differently applied using the token() construct since it is considered as a terminal which can not be matched to a non-terminal.

```
1 fruitList          = listOf(#fruitName, ';'),
2 fruitName          = letter+,
3 listOf :rule :sep = apply(rule) ( token(sep) apply(rule) )*
```

Figure 2.8: Higher-order production rules

Inheritance

Another interesting feature of OMeta is the ability to create new grammars that inherits from existing grammars. Using inheritance, the new grammar inherits all production rules of the base grammar, including the semantic predicates and code. The inherited production rules can be overridden if necessary to change its behavior. The rule application in the new grammar is obtained using the ($\wedge$) operator behaves exactly like a super-send in traditional OO-languages.

```
1 ometa ScientificCalculator <: Calculator{
2   mulExpr = mulExpr:x '*' expExpr:y -> (x * y)
3           | mulExpr:x '/' expExpr:y -> (x / y)
4           | expExpr,
5   expExpr = expExpr:x '^' ^primExpr:y -> Math.pow(x,y)
6           | ^primExpr
7 }
```

Figure 2.9: Grammar inheritance in OMeta

Figure 2.9 shows the implementation of a Scientific Calculator which extends from the basic calculator (figure 2.1) discussed previously. The scientific calculator adds functionality for exponentials by creating a new production rule *expExpr*. Due to the order of mathematical application, the exponential should be matched after multiplication. The mulExpr is for this purpose modified and refers now to expExpr instead to the primExpr. The expression rule itself is created at line 5 and accepts the exponential notation using the ^ operator. The code at line 6 provides a feedback mechanism to the primary expression so the order of mathematical application is continued. Note that the non-terminal primExpr is not defined in the ScienticCalculator but is inherited from the parent grammar. The non-terminal of a parent grammar is accessed using the super-send operator (^) such as on line 6.

2.3 Waebric

2.3.1 Introduction

Waebric is a domain-specific language for factoring xHTML webpages. Waebric was created as a response to the limitations of xHTML, template languages and WYSIWYG HTML editors. In Waebric, a website is factored much more like conventional programming languages using (parameterized) function declarations and corresponding calls instead of the conventional HTML tags (e.g.). In Waebric, markup elements are the equivalent of HTML tags and are specified using an identifier that corresponds to the xHTML element name optionally followed by parentheses (e.g. div()). The corresponding attributes of a xHTML tag are provided between the parentheses of the markup element in the form "`key = value`", e.g. div(width=200). The built-in operators *echo*, *comment* and *cdata* are used to produces text content, XML comments and CDATA sections respectively.

Apart from markup, Waebric also supports the creation of strings, symbols, lists and records (key/value). List can be iterated using the each statement while values of a recordlist are obtained using the dot-notation (e.g. product.price).

2.3.2 Example

A basic Waebric example is provided in figure 2.10. The example starts with the declaration of the module which states that the code should be contained in the file *homepage.wae*. After the module definition, code is supplied to define where the generated xHTML code should be outputted to the filesystem. This information is contained between the keyword *site* and *end*. In the example, the function *home* is called and executed and its result is stored in a file named *index.html*. If desired, more output files can be generated by specifying them in the site declaration.

The site definition is followed by the declaration of the *home* function containing one argument *msg*. The notation of a function declaration is similar to traditional programming languages. Waebric uses the keyword `def` followed by the function name and optional arguments, and closed with the keyword `end`. The body of a function is used to define the actual markup or to call other functions which on their turn generate markup.

Note that the notation of a function call is identical to the notation of a markup element. In fact, Waebric makes no difference in processing them. A function call is only a function call if there is a corresponding function declaration. Hence, if a function call is made to an undefined function then it is interpreted as a markup element. On the contrary, if a markup element is created but there is a function with the same name as the markup element, then the function is processed. Hence, the markup element is in such case a function call. This process is also referred as shadowing.

```
1  module homepage
2
3  site
4    index.html: home("Hello World!")
5  end
6
7  def home(msg)
8    html{
9      head title msg;
10     body echo msg;
11   }
12 end
```

Figure 2.10: Waebric example

To allow the generation of *nested* xHTML tags, markup elements are defined in a chain, such as shown at line 9 and 10 of figure 2.10. An xHTML element can be foreseen of multiple children using curly brackets (line 8-11). The Waebric program generates a *html* tag with two children, the *head* and *body* tag. Within the head tag there is a *title* tag nested which contains the value of the variable *msg*, here "Hello World!". The *body* tag contains the text "Hello World!" which was outputted directly using the echo statement. Figure 2.11 illustrates the generated xHTML code for the example in figure 2.10.

```
1  <html>
2    <head><title>Hello World!</title></head>
3    <body>Hello World!</body>
4  </html>
```

Figure 2.11: xHTML code of Waebric Example

2.3.3 Embedded markup

Until now, markup and datatypes were defined in a statement context (i.e. in the function body). Markup can also be defined inside text in order to highlight words, output the value of variables or to generate other markup (including function calls). If markup is used in such context, then it is referred as *embedded markup*. An example is provided in figure 2.12. The example echo's the message "Click here to contact us" where the word "here" is provided with a link to an e-mail address.

```
1 echo "Click <a(href='mailto:info@cwi.nl) 'here> to contact us"
```

Figure 2.12: xHTML code of Waebric Example

2.3.4 Juxtaposition

Special attention is required when using markup and datatypes with respect to markup juxtaposition. If a *chain* of markup elements is specified, then all elements are processed as markup apart for the last element which is processed as a variable. To override this behavior and use markup explicitly in the last element, one should use parentheses. For a single markup element without parentheses (e.g. div) in an embedded context, the rule is that it is processed as a variable whereas in a statement context it is processed as markup. In order to use markup in an embedded context, parentheses should be used. Note that a single markup element in a statement context cannot be transformed to a variable reference. Instead, the echo statement should be used with a string containing embedded markup. Figure 2.13 shows several examples to demonstrate the juxtaposition rules:

```
1 div;                 //markup
2 div div;             //markup, variable
3 div div();           //markup , markup
4 echo "<div>";        //variable
5 echo "<div()>";      //markup
6 echo "div div";      //markup, variable
7 echo "div div()";    //markup, markup
```

Figure 2.13: xHTML code of Waebric Example

2.3.5 Around parameterization

Example 2.10 illustrated how *data* was passed through a function using arguments in the function call. Waebric also supports *around* parameterization allowing *markup* to be passed through as well. The markup that needs to be passed through is not specified as an argument of the function call but by defining it right after the function call. This implies that all markup and data that is specified after a function call in the current statement is automatically passed through the function in question. The function itself is not required to use this markup, but if requested it can access the markup and data using the *yield* statement.

Example 2.14 illustrates how the yield statement is used. The main function first creates the layout structure by calling the layout function. The layout function itself creates an *html* element with a *head* and *body* element. The *body* element outputs the *yield* state-

ment which corresponds to the markup that was specified in the main function after the call of the layout function, here a call to the function *content*. The main advantage of such construction is that the layout function can now be reused by other web pages as well, eliminating the use of duplicate code.

```
1  def main
2    layout("Welcome to my homepage!") {
3      content();
4    }
5  end
6
7  // Create basic layout for the webpage
8  def layout(title)
9    html {
10     head {
11       css("blueprint.css", "screen, projection");
12       title title;
13     }
14     body yield;
15   }
16 end
17
18 //Create content for the webpage
19 def content
20   //Body content
21 end
```

Figure 2.14: Example yield statement

Chapter 3

Research method

This chapter discusses the experiment used in this thesis in order to provide an answer to the research questions defined in section 1.2. The main goal is to determine the quality of two DSL implementation techniques based on maintainability, functionality and efficiency metrics. The experiment is organized in three phases:

- Factoring an implementation of Waebric using the interpreter technique
- Factoring an implementation of Waebric using the compiler-generator technique
- Metric extraction for both implementations

The first technique chosen for this research is the interpreter approach in which traditional GPL interpreter techniques are applied to implement the DSL (vanilla implementation). The phases of this technique includes lexing, parsing, typechecking and interpreting. The second implementation applies the compiler-generator technique and involves the use of the language development tool OMeta to automate certain steps of the interpreter technique (OMeta implementation). In particular the lexing, parsing and typechecking phases are automated based on a formal grammar. A detailed description of each implementations is provided in chapter 4 and 5. To ensure optimal comparison, the following conditions were imposed with respect to the implementations:

- The implementations are developed by the same DSL implementor
- The implementations are carried out in the host language JavaScript
- The implementations implements the reference DSL Waebric
- The implementations endure a code review
- The implementations minimizes the use of libraries written in other languages as much as possible

- The implementations are based on the formal grammar specified in SDF provided by the CWI
- The optimal implementation is considered to be the ASD+SDF implementation provided by the CWI

After completion, the quality is measured for each implementation based on software metrics. A definition of software metrics is given by Paul Goodman [9]:

> *"Software metrics are the continuous application of measurement-based techniques to the software development process and its products to supply meaningful and timely management information, together with the use of those techniques to improve that process and its products"*

The software metrics used in this research are organized into three groups: maintainability, functionality and efficiency. Attention is needed when using metrics given to its definitions and counting rules as they are sometimes ambiguously defined in literature. This is especially true if comparison is made between different languages or if metrics are extracted with different tools or methodologies. Therefore, a standard definition and measurement technique for each metric is given in the following sections to create a uniform basis of estimate for each implementation.

3.1 Maintainability

Maintainability indicates the ease with which a program (i.e. an implementation) can be modified in order to correct defects, meet new requirements, make future maintenance easier or cope with a changed environment. Maintainability is generally measured using a combination of size and complexity metrics. The list of metrics considered in this experiment is provided in table 3.1. Note that class and package count are not measured since JavaScript is a prototype-based language in which classes or packages are not present.

The maintainability metrics are only measured on those part of the code which are factored by the implementor itself. This excludes external libraries and generated code as they are not part of the implementor's effort. Formal grammars are measured using the metric suite of Power and Malloy[28].

Metric description	
LOC	Number of Lines of Code
eLOC	Number of Effective Lines of Code
MCO	Method Count
AMS	Average Method Size
ECC	McCabe Extended Cyclomatic Complexity
EFF	Halstead Program Effort
VOL	Halstead Program Volume
MI	Maintainability Index

Table 3.1: Size and complexity metrics for program code

The following sections discusses the size and complexity metrics in respect to their counting rules and definitions. A classification is made between metrics for program code and grammar-based metrics since these metrics are gathered differently.

3.1.1 Program code metrics

Lines of code (LOC)

LOC measures the amount of source code lines (line break characters) in a software entity. It's one of the oldest metric found in literature and is still used routinely as the basis for measuring programmer productivity (LOC per programmer per month) [7]. LOC measures all lines in the software where code appears while comments and blank lines are excluded in this metric. A downside of this metric is that it's sensible for code conventions and formatting.

Effective Lines of Code

Effective Lines of Code (eLOC) is very similar to LOC but excludes standalone braces and parenthesis. The use of braces and parenthesis is a very common practice and is encouraged in some languages (e.g. Java [22]). Ignoring standalone parenthesis and braces is reasonable since it does not represent actual work performed by the programmer. LOC and eLOC are highly correlated and therefore redundant in an analytic context [30].

Method Count

Method count (MCO) is measured as the amount of subroutines or functions in the program. In JavaScript, functions are defined using the keyword *function*. By counting the total amount of occurrences of this keyword, the method count is obtained. Note that JavaScript is prototype-based in which classes are not present. Classes are mimicked in JavaScript using objects in order to modularize the code. Since these objects are specified identical to functions, the method count is increased when using mimicked classes.

Average Method Size

The average method size metric combines the LOC metric and MCO by measuring the average lines of code per method (AMS). A high AMS *might* indicate that (some) methods are too complex. AMS is calculated by diving the total lines of code (LOC) by the number of methods.

$$Average\ Method\ Size\ (AMS) = \frac{\#LOC}{\#Methods}$$

McCabe Cyclomatic Complexity

The most known method for measuring the complexity of a software system is the cyclomatic conditional complexity (CC) created by Thomas McCabe in 1976. It directly measures the number of linearly independent paths through a program's source code [20]. The computation of CC is based on the control-flow graph of a program:

$$McCabe's\ Cyclomatic\ Complexity\ (CC) = E - N + 2P$$

where

$$\begin{aligned} CC &= \text{Cyclomatic Complexity} \\ E &= \text{Number of edges of the graph} \\ N &= \text{Number of nodes of the graph} \\ P &= \text{Number of connected components} \end{aligned}$$

The complexity of a program is at least one since there is at least one execution path in each program. CC is independent of physical size, and complexity only depends on

the decision structure (branches) of a program [20]. An alternative for measuring CC is counting every decision point or branch in the program. The decision points considered in JavaScript include if, else if, switch, case, for, while, do while and catch.

While it is possible to count the complexity of the whole program, CC is mostly expressed as the average CC per method. The higher the complexity number, the more complex the code is. Based on such number, McCabe stated that a function with a complexity of more then 10 is considered to be complex as the risk of errors increases significantly.

Over time, Myers [23] extended McCabe's complexity metric by including Boolean operators in the decision count, the so-called Extended McCabe Cyclomatic Complexity (ECC). Whenever a Boolean operator (&& and ||) is found *within a conditional statement*, the ECC is additionally increased with one. The reasoning behind ECC is that a Boolean operator increases the internal complexity of the branch. The ECC metric is also applied to the formulas of Halstead and the Maintainability Index and therefore ECC is used throughout this research.

Halstead

The measures of the late Maurice Halstead were introduced in 1977. Halstead created several metrics which each have different meaning. Operators (e.g. arithmetical operators) and operands (e.g. variables, numbers and strings) are the fundamentals of Halstead's metrics:

Metric description	
n1	Number of distinct operators
n2	Number of distinct operands
N1	Total number of operators
N2	Total number of operands

Table 3.2: Halstead primitive metrics

An important question arise about the definition of operators and operands. When these metrics were designed, programming was procedural and in general monolithic. Halstead stated that the determination is intuitively obvious and requires no further explanation. Qutaish and Abran [29] pointed out that "Intuition is insufficient to obtain accurate, repeatable and reproducible measurement results." A first attempt to define operators and operands was made by Indranil Nandy who provided several guidelines for the languages ANSI C, C++ and JAVA. Most of these guidelines are used in other languages as well. Since no attempt were made in literature to define these constructs for JavaScript, we apply the guidelines of Java as much as possible onto JavaScript.

A first significant metric is Halstead's Length metric (LTH). It's the counterpart of the LOC metric and measures the size of a program. Research has shown that the LTH

metric is more accurate then LOC [17].

$$Program\ Length\ (LTH) = N1 + N2$$

Program Volume (VOL) is used to measure the information contents of a program, measured in mathematical bits. It is calculated using the program vocabulary and length:

$$Program\ Vocabulary\ (VOC) = n1 + n2$$
$$Program\ Volume\ (VOL) = LTH * \log_2(n)$$

Halstead Effort is another commonly used metric to express the time needed to implement or understand a program. It is not directly derived from the operators and operands but based on program's volume and difficulty:

$$Program\ Difficulty\ (DIFF) = \frac{n1}{2} * \frac{n2}{n2}$$
$$Program\ Effort\ (EFF) = VOL * DIFF$$

Criticism on Halstead is given in recent mathematical research [29] which observed that the structure and the size of a program changes the properties of the Halstead metrics. The research concluded that the original measures Volume, Difficulty and Effort can be reduced to the measures LTH and N2 under certain conditions. The validity of the LTH metric has been confirmed in the research.

Halstead introduced more metrics such as Halstead Time (estimated implementation time) and Halstead Delivered Bugs (probability of bugs) but are outside the scope of this thesis.

Despite the criticism on the Halstead measures, they are still widely used in software measurement tools and by many authors [24] [13] [25].

Metric description	
VOC	Halstead Program Vocabulary
LTH	Halstead Program Length
VOL	Halstead Program Volume
DIFF	Halstead Program Difficulty
EFF	Halstead Program Effort
TIME	Halstead Program Time
BUGS	Halstead Program Bugs

Table 3.3: Halstead derived metrics

Maintainability Index

The Maintainability index (MI) is a composite metric constructed by Oman and Hagemeister at the University of Idaho to indicate the overall system maintainability. It is

based on Halstead Volume (VOL) metric [10], Cyclomatic Complexity (ECC) [20] metric, average number of lines of code per module (LOC), and optionally the percentage of comment lines per module (COM). The notion *module* is considered to be the smallest unit of functionality. In JavaScript, the smallest unit is a function.

Two variants of the MI are available, one that considers comments (MI4) and one that does not (MI3).

$$MI3 = 171 - 5.2 * ln(avgVOL) - 0.23 * avgV(g) - 16.2 * ln(avgLOC)$$
$$MI4 = 171 - 5.2 * ln(avgVOL) - 0.23 * avgV(g) - 16.2 * ln(avgLOC) + 50 * sin(\sqrt{2.4 * perCM})$$

where

$$\begin{aligned} avgVOL &= \text{average Halstead Volume per module} \\ avgV(g) &= \text{average Extended Cyclomatic Complexity per module} \\ avgLOC &= \text{average count of Line of Code per module} \\ perCM &= \text{average Percent of Lines of Comments per module} \end{aligned}$$

Note that the original polynimial equations of MI used Halstead Effort instead of Volume. Due to lack of confidence in the Halstead Effort, the MI was modified in order to incorporate the use of Halstead's Volume. At the same time the equation was modified to limit the weight of comments as studies had shown that MI4 was overly sensitive for comments [18] [19].

In general, a higher MI value indicates better maintainability [26] [37]. The interpretation of the MI values is presented in table 3.4.

Maintainability	MI3 (without comments)	MI4 (with comments)
High	$MI3 >= 50$	$MI4 >= 85$
Moderate	N/A	$65 <= MI4 < 85$
Low	$MI3 < 5$	$MI4 < 65$

Table 3.4: Cutoff values Maintainability Index

3.1.2 Grammar metrics

Software metrics are usually applied to program code. The application of metrics onto grammar-based software is less common. In literature, the topic has been addressed since the mid 60's by Feldman [6] and Brauer [4] shortly after the introduction of grammar-based languages. The main problem with grammar-based software is the difference in abstraction compared with general-purpose languages. A grammar is based on production rules, terminals and non-terminals while program code is based on classes, functions and statements which complicates comparison.

In a more recent study of Power and Malloy, metrics were interpreted in a grammatical context [27] [28] which allows the mapping of grammar-based metrics onto metrics extracted from program code. Power and Malloy inspired their metrics on earlier work performed by Csuhaj-Varj and A. Kelemenov [5] and Brauer [4] who have applied these metrics to measure descriptional complexity of context-free grammars.

For the mapping of size and complexity metrics, Power and Malloy reasoned as follows. In program code, the concepts of control-flow graph and call graph are used to calculate most size and complexity metrics. If these concepts could be applied to a grammar as well, then it should be possible to reuse them. They continued by stating that a grammar can be considered as program, and a program consists out of a set of procedures with a body containing the control primitives of the language. By now corresponding procedures with non-terminals, and procedure bodies with the right-hand side of the production rule, they concluded that many size and complexity metrics are applicable to grammars as well.

The next sections discusses the metrics of Power and Malloy into more detail.

Number of terminals and non-terminals

According to Csuhaj-Varj et al [5], the number of *defined* non-terminals in a grammar (VAR) corresponds to the number of procedures.

$$Number\ of\ unique\ non-terminals\ (VAR) = \#N)$$
$$Number\ of\ unique\ terminals\ (TERM) = \#T$$

A larger number of non-terminals implies a greater maintenance overhead since changes to the definition of one may effect many others. In implementation terms, the size of the parse table is usually proportional to the number of terminals and non-terminals and can affect execution speed, especially if predictive parsing algorithms are used (e.g. OMeta).

Note that the VAR definition requires that the non-terminal is *defined* in the grammar. OMeta supports a mechanism to let a grammar inherit from an existing grammar. If a derived non-terminal is used in the new grammar, then it is not considered to be defined in that grammar, hence the restriction to *defined* non-terminal for the grammar in question. If the derived non-terminal is however *overridden*, then it is counted as a defined non-terminal.

McCabe Cyclomatic Complexity

Complexity in a grammatical context is measured by counting the number of non-circular execution paths in the grammar.

$$McCabe's\ Cyclomatic\ Complexity\ (CC) = \sum_{(n \to \alpha) \in P} mccabe(\alpha)$$

where

$$mccabe(v) = 0\ for\ v \in (N \cup T),$$
$$mccabe(f^k(\bar{x})) = 1 + mccabe(\bar{x})\ for\ k \in \{| * +\},$$
$$mccabe(f^k(\bar{x})) = mccabe(\bar{x})\ for\ k \in \{.\}$$

Note that under this formula the minimum complexity is zero, in contrast to the complexity measured in program code which has a minimum complexity of 1. The value is generally implemented as the number of branching operators found in a grammar [2]. In OMeta this corresponds to the alternative and repetition operators. The optional operator (?) is not included in OMeta.

Branch operators	
Alternative	|
Repetition ≥ 0	$*$
Repetition ≥ 1	$+$

Table 3.5: Branch operators in OMeta

Some confusion might arise regarding the repetition operator with separator. In some grammar syntaxes their is a special operator available for this kind of operations. If this operator is used, then the repetition is only valid if the specified delimiter is present between each repetition. The example below [2] shows an SDF production rule with the use of the repetition with separator:

```
1 "types" { TypeDefinition ";" }+ → TypeDefinitions
```

OMeta has a similar way to express repetition with separator:

```
1 TypeDefinitions = "types" listOf(#TypeDefinition, ";")
```

In [2] its argued that such constructs are built-in operators since it causes two possible execution paths in the grammar, increasing the ECC with one. It's interesting to observe that in OMeta the equivalent listOf() is not a build-in operator. In fact it is a *derived* non-terminal. The reasoning behind this is as follows. listOf() is a *derived higher order* production rule originating from OMeta's base code, a compiled grammar specifically designed for creating parsers. In this grammar, listOf is defined as a standard production rule, hence listOf can be defined as a non-terminal. One might argue that listOf is still

causing an extra branch in the production rule, and thus the ECC should be increased if listOf() is used. This is in contradiction with the purpose of higher order rules: obtaining reusability of code and thus reducing complexity. Consider the following example in OMeta:

```
1 Module = "module" IdCon ("." IdCon)* ModuleElement*,
2 Import = "import" IdCon ("." IdCon)*
```

The ECC in this example equals 3 as the iterative operator is used three times. A more optimal implementation is considered below where the $IdCon("."IdCon)*$ construction is replaced with a new non-terminal.

```
1 Module   = "module" ModuleId ModuleElement*,
2 Import   = "import" ModuleId,
3 ModuleId = IdCon ("." IdCon)*
```

In analogy with program code, the duplicate code is moved to a new subroutine for reusability. This results in a lower complexity $ECC = 2$. A similar result is obtained if *parameterized* production rules are used:

```
1 Module               = "module" ModuleId ModuleElement*,
2 Import               = "import" ModuleId,
3 ModuleId             = listOf(\#IdCon, "."),
4 listOf :rule :sep    = rule (sep rule)*
```

In the example above, the value of ECC remain status quo but the grammar is now capable of using other constructs that uses repetitions with a separator. In analogy with program code, the duplicate code is abstracted for use in other scenario's as well.
Finally, if higher order production rules are taken into consideration and a base grammar is inherited, then the result would be as follows:

```
1 Module   = "module" ModuleId ModuleElement*,
2 Import   = "import" ModuleId,
3 ModuleId = listOf(#IdCon, ".")
```

Here, the listOf() rule was removed as it is inherited from the parent grammar (not shown), causing the ECC to decline with one. Conversely, the complexity of the parent grammar is now increased with one. The analogy with program code is also applicable here: the generalized code is moved to the super class, making it accessible for all derived instances, hence complexity is removed from the program in question and moved to another part in the software.

Conclusion is that listOf() should not be considered as an operator but instead as a non-terminal, even if the non-terminal is defined in the parent grammar. In the latter, the complexity is not increased for the grammar in question.

Average RHS size

Average right-hand size (RHS) is the counterpart of average method size (AVS) for program code. It is estimated by counting the number of terminals and non-terminals in the body of each production and divide it by the amount of defined non-terminals (VAR). RHS is a formal alternative to measure the average lines-of-code (LOC) per method.

$$\text{Average RHS size (RHS)} = \frac{\sum_{n \to a \in P} size(\alpha)}{\#N}$$

Depending on the type of parser, a high RHS value might involve performance issues as more symbols must be placed on the parse stack. The RHS can easily be decreased by replacing some of it by a new non-terminal. Therefore, the RHS metric should always be considered in association with the amount of terminals and non-terminals.

Lines of code*

Measuring the lines of code for grammar-based software is not provided in the metric suite of Power and Malloy. Such metric is however widely used and useful for comparison. We therefore propose a new metric, LOC*, which counts all code within the grammar based on the average method size and the method count metrics of Power and Malloy earlier presented. By multiplying these two values, it is possible to count all code lines in the grammar. This provides a base for better comparison between the metrics for program code and grammar.

$$\text{Lines of Code* (LOC*)} = (VAR \times RHS)$$

Halstead

Halstead's Volume and Effort metrics are calculated as functions of the number of operators and operands a program contains. In a grammatical context, the operators corresponds to the the standard grammatical operations and the operands as the terminals and non-terminals in a given grammar. An overview of the operators and operands used in the calculation of Halstead's formulas is given in table 3.6.

Description	Operator	Example
Sequence		`Expr = Symbol1 Symbol2`
Alternative	|	`Expr = Symbol1 | Symbol2`
Repetition ≥ 0	$*$	`Text = '"' Letter* '"'`
Repetition ≥ 1	$+$	`Num = Digit+`
Negation	$\sim$	`Var = ~Keyword`
Lookahead	$\sim\sim$ or &	`ExprFollows = &Expr`
Semantic predicate	?	`LargeNum = Num:n ?  (n > 1000)`
Semantic action	$\rightarrow$ or !	`Sum = Num:a "+" Num:b → a + b`
Assignment operator	$=$	`Expr = Literal`
List	[...]	`["Expr" Symbol1 Symbol2]`
Parentheses rules	(...)	`Keyword = super(#Keyword) | "import"`

Table 3.6: Halstead operators and operands in OMeta

The sequence operator, also called the concatenation or juxtaposition operator is always implicit in OMeta/JS. It composes the various terminals and non-terminals into a production body. If OMeta would use explicit concatenation (using dot as operator), the result would look as follows:

```
1 Module = "module" . ModuleId . ModuleElement*
```

Four operators can be identified in the example above, the assignment operator, two sequence operators and one repetition operator.

Note that the operators for separating production rules (comma), grouping (parentheses) and binding (colon) are ignored as they do not perform operations on the operands nor complicate the grammar. The argument separators for "calling" parameterized non-terminals are also excluded since it's a specific notation for OMeta/JS not for OMeta in general. The hash symbol used for referring to non-terminals in some special cases is ignored for the same reason.

3.2 Efficiency

The efficiency of a program is determined with the use of performance tests which measures the execution speed in milliseconds for each implementation. The lower the execution time, the faster a Waebric program is converted to xHTML markup. Execution speed is measured on the main components of each implementation such as the lexer, parser, checker and interpreter to identify possible bottlenecks. Since the execution time of an implementation mostly depends on its input, 8 different test programs were developed with sizes ranging from 54 to 10.000 lines of code. Each test is executed 100 times to minimize the effects of background tasks of the operation system. The average execution time of these 100 tests is then calculated and used for analysis.

3.3 Functionality

Functionality of a Waebric program is measured using acceptance tests and the level of error reporting towards end-users.

The acceptances tests includes a collection of 104 Waebric programs which each exercises a particular functionality of the DSL and results in a failure or acceptance. Each implementation reads the Waebric program and converts it to xHTML markup. The results of each implementation is then compared with the results of the reference implementation in ASF+SDF[1] using a file comparison utility DiffMerge[2]. The test is accepted if no differences are found. The functionality metric is expressed as a percentage of the number of accepted tests versus the number of failed tests. The objective of this metric is to achieve 100% acceptance.

The second level at which functionality is measured is the support of error reporting. This is measured using a series of Waebric programs which each contains exactly one error. For each of these errors, the implementation should throw an (expected) error. The error message should meet the following requirements:

- Provide a correct description of what is causing the error
- Provide a description is terms of the problem domain (e.g. missing a semicolon after the statement)
- Provide the exact position of the error in the input file (line and column number)
- Provide a relevant part of the parser stack for error debugging

The more each error message includes these requirements, the better the level of error reporting is considered.

3.4 Hypotheses

> "Maintenance of a DSL implementation is reduced when using OMeta instead of traditional interpreter techniques." (1)

Formal grammars (e.g. BNF) allows the development of implementations at a higher abstraction level compared to third-generation programming languages such as COBOL, Java, JavaScript and C#. A formal grammar describes *what* the implementation should do rather than *how* this should be achieved. Abstracting away the implementation details allows the programmer to focus on the problem domain. This generally results

[1] `http://www.meta-environment.org`
[2] `http://www.sourcegear.com/diffmerge`

in a much smaller code base and thus maintenance effort can be reduced [1]. This might however be counteracted by the expressiveness that formal grammars yields as a result of higher abstraction. A higher expressiveness usually results in more complex code and thus affects maintainability negatively. Despite this observation, it is believed that the effects of a smaller code base are larger then the effects caused by higher expressiveness. The assumption is therefore made that maintainability of formal grammars is higher compared to general programming languages. Since the OMeta is based on formal grammars, it follows that the OMeta implementation results in a better maintainability.

> "A DSL implementation carried out with the interpreter technique has a higher performance compared to OMeta" (2)

In traditional interpreter techniques, implementors have great control on how the implementation is carried out. Such freedom is however not provided in compiler-generators since implementation details are deliberately abstracted away from the implementor. The implementor can utmost adjust a few global settings that affect the way the implementation is carried out, but not to such extent as with the interpreter technique. The limited freedom could have a negative effect on the performance of compiler-generators.

On the other hand, one might say that compiler-generators are specifically designed for implementing DSLs and utilize several mechanisms which improve performance positively. Compiler-generators do however need to balance between functionality and performance and it is possible that some performance improvements cannot be implemented as otherwise functionality would be lost. OMeta for instance has chosen not to implement the memoization feature for parameterized production rules which guarantee linear execution time as OMeta wants to keep its footprint small. Compiler generators might also generate code which contains a certain degree of overhead, resulting in even more performance loss.

But even if compiler-generators would include the most superior optimization techniques, then it would still be possible to implement these techniques in traditional interpreter techniques given the freedom of implementation they provide. It is therefore assumed that the interpreter technique provides better performance results than the compiler-generator techniques.

> "Closer syntax notation and better error reporting is obtained when applying the interpreter technique instead of OMeta" (3)

A higher abstraction might not only affects complexity and performance, it might also affects the level of error reporting. BNF for example does not support mechanisms that allows the integration or customization of error messages towards end-users. Instead, the end-user is left over to the generalized error messages provided by the compiler-generator itself. Such messages are usually experiences as cryptic and incomprehensible. This is

in contrast with the interpreter technique by which error messages can be implemented anywhere required and with clear and understandable messages in terms of the problem domain and in the language of the end-users. Since OMeta's syntax is similar to that of BNF [36], it is assumed that it's level of error reporting is equally and thus inferior to the vanilla implementation.

BNF also includes a flaw whereby symbols such as $[<, >, |, ::=]$ in the DSL might conflict with BNF's syntax itself. If OMeta derive these limitations, then the level of supported syntax notation might be at risk. Programming languages have no such limitations since they are supposed to read any character from plain text. Based on this, supported syntax notation is considered lower in OMeta.

3.5 Threats to validity

3.5.1 Waebric as reference DSL

To keep comparison proportional, the experiment uses a reference DSL that is implemented in both implementation approaches. One might ask how representative the proposed reference DSL Waebric is compared to other DSLs. The more representative Waebric is, the better the results of this research can be generalized and applied to other projects as well. With representative it is meant that the reference DSL shouldn't be too simple nor too complex compared to other DSLs. In order to identify the position of Waebric between other DSLs, a comparison is made based on the characteristics of Waebric's in relation with simple and complex DLSs.

Complex DSLs can be identified as languages which leans much more to GPLs then to DSLs. This is possible if the concepts it supports are similar to those of GPLs, if the application domain is comparable to that of a GPL or if understanding the DSL requires the same amount of effort as with GPLs. COBOL is a an example of such language where it is disputable whether it is a DSL or a GPL due to the wide application domain and extensive GPL concepts it includes. SQL on the other hand has no relation with GPLs regarding syntax notations or the concepts it contains, but is so extensive that its learning curve is probably equally steep to that of a GPL.

More simple DSLs are languages with a narrow application domain, limited syntax notations and/or flat learning curve. The DOT language could be categorized in this group due to its simple notation and very specific application domain (graph visualization)

To determine the position of Waebric, the following features where identified to allow comparison:

- Parameterized function declarations and function calls
- Limited control flow support including if-else and each statements

- Limited datatype support including numbers, string (text), symbols, lists and records
- Dotted-pair notation for field expressions
- Capability to import other Waebric programs
- Access to variables and functions in text (Embedded)
- Around parameterization of functions

It is noticeable that Waebric does support some of the features and concepts found in general-purpose languages such as function declarations, control flow and datatypes. This is however not to such extend that Waebric can be classified as a GPL since GPLs have much more features and concepts than Waebric such as file system access, classes/prototypes, customized datatypes, error handling, etc. Moreover, the concepts that are adopted by Waebric from GPLs are limited to the very basics. The limited support for control flow and limited computation with data as described in Waebric's description confirms this viewpoint [32].

Waebric's application domain is identical to that of HTML as they both target towards web development, more specifically (x)HTML markup. The extent of its domain is similar to that of DOT and SQL which have a specific application domain. Contrary, DSLs such as COBOL, VHDL and MATLAB have a much wider application domain.

A final comparison is made based on Waebric's learning curve. When Waebric is compared to languages such as SQL or COBOL, then it's fair to state that Waebric is easier to understand. Compared with more simple DSLs such as the DOT language, then Waebric is probably a lot harder to learn. This is especially true since the end-users of Waebric are mostly web developers who have little or no knowledge in general programming. Not only do they have to learn the concepts of Waebric, they are also required to understand the concepts of general programming which are included in Waebric such as the (parameterized) function declarations, around parameterization using the Yield statement, dotted-pair notation and datatypes.

Identifying the position of Waebric between other DSLs is not an easy task and probably requires a more extensive comparison than here provided. Nevertheless, comparison showed that Waebric features average characteristics regarding learning curve, expressiveness and application domain and that there are no extreme characteristics that might questioning its representativeness.

3.5.2 Mapping of metrics

In section 3.1.2 it was discussed that the metric suite of Power and Malloy was used in order to extract metrics from formal grammars such as used in OMeta. The metric suite

adapted object-oriented software metrics that are commonly used to measure program complexity and applied these metrics to measure complexity of formal grammars.

The research that was carried out by Power and Malloy validated these metrics based on a case study where four types of grammars where compared. Based on this, they stated that the metric suite allows integration in existing metrication processes. What the case study didn't included was a comparison between program code and grammar-based software in order to prove the equality of the metric suite with traditional software metrics. Such comparison is important to validate that there is a one-on-one mapping between the metric suite and traditional software metrics. The absence of this experiment might indicate that it was never the intent of the authors to allow such mapping. At least one example that contradictory the one-on-one mapping is the observation that the counting rules of the McCabe Cyclomatic complexity in Power and Malloy's metric suite are not identical to the original MCC, namely the minimum value is zero instead of one.

Drawing conclusions from metrics is extremely risky and requires many relativization during comparison, especially if metrics are compared over different levels of abstractions such as here presented in this research. The absence of empirical evidence that supports the one-on-one mapping may invalidate comparison. It is therefore important to take these issues into account during comparison and to draw conclusions with extreme caution, even if results show clear differences.

3.5.3 OMeta as compiler-generator

The list of available compiler-generators is large with more then hundreds if not thousands of different tools. Each of those compiler-generators utilizes different host languages, parsing algorithms, IDEs and/or syntax notations. OMeta for instance targets to languages such as JavaScript, Squeak, C#, Scheme, Lisp, Python or Ruby using a variant of Packrat parsers while Yacc is a LALR parser only targeted to C. Parsing algorithms have great impact on performance while syntax notation have impact on functionality and/or productivity. Maintainability is affected by the way how action code is organized (internal or external) while implementation effort is determined by the tool itself such as the presence or absence of an IDE. A comparison between compiler-generators is however beyond the scope of this thesis and belongs to the DSL Benchmark Implementation project as a whole in which this research is only an addition. This research will therefore not generalize its results to all compiler-generators since more research is required to confirm this.

Chapter 4

Vanilla implementation

This chapter discusses the implementation of Waebric using traditional interpreter techniques, further appointed as the "vanilla implementation". An overview of the different phases is provided in figure 4.1. The first phase involves *lexical analysis* or *lexing* in which the source of a Waebric program is read as a sequence of characters and converted into a sequence of tokens. A token is a *categorized* block of text such as a number, identifier, keyword or symbol. The second phase - *syntactic analysis* or *parsing* - verifies that the tokens form an allowable expression in respects to Waebric's syntax notation and converts the tokens into an internal data structure or Abstract Syntax Tree (AST). The final phase is concerned with *semantic analysis* in which the semantics of the internal data structure are validated (*typechecking*) and converted to xHTML markup (*interpreting*).

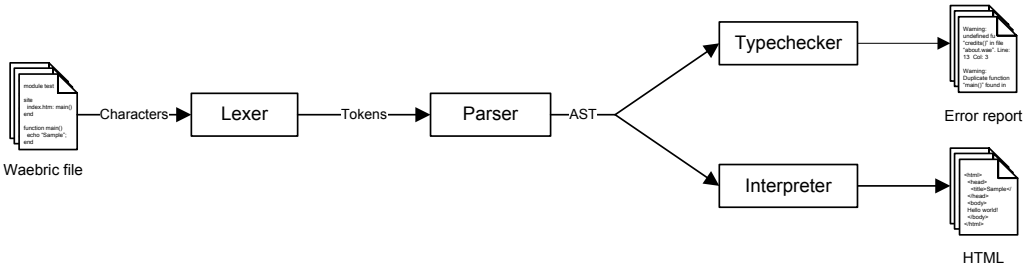

Figure 4.1: Phases of the vanilla implementation

A detailed description of each phase is provided in the following sections.

4.1 Lexical analysis

A lexical analyzer or simply lexer reads characters from the input, groups them into "lexemes", and produce as output a sequence of tokens for each lexeme in the source program [1]. A token is a pair consisting of a token name and an optional attribute (e.g. <**text**, "Hello world!">). The block of text corresponding to the token is known as a lexeme. Table 4.1 gives an overview of the tokens that are considered in Waebric. Note that whitespace and comments are ignored during tokenization.

Type	Example
Natural	47
Symbol	$+, \{,), >$
Single quote text	'Sample
Double quoted text	"Hello world"
Identifier	myVariable
Keyword	def, end, if, else, each

Table 4.1: Waebric Tokens

The most important consideration to perform lexical analysis before parsing is the simplicity it introduces at design level. The separation of lexical and syntactic analysis often allows to simplify at least one of these tasks and enables individual efficiency improvements on both components. Additionally, portability is enhanced since input-device-specific peculiarities can be restricted to the lexical analyzer.

4.1.1 Process

A simplified representation of the implementation is provided in figure 4.2. The process starts with evaluating the value of a character in order to determine which type of token it represents. Such evaluation is based on regular expressions or expected starting and/or ending character.

Once the lexer has determined which type of token it is dealing with, it starts tokenizing the character by storing its value in a token. Then, for each subsequent character that matches the expected input of this token, the character's value is appended to the existing token. From the moment that an unexpected character or ending character is found, tokenization of the current token stops and the token is added to a tokenlist.

Finally, a new character is read from the input stream and evaluated so a new tokenization process can take place. This process is repeated until all characters are consumed after which the tokenlist is passed on to the parser.

A character that is not recognized or not expected by the lexical analyzer results in a warning after which tokenization continues by evaluating the next character at the input.

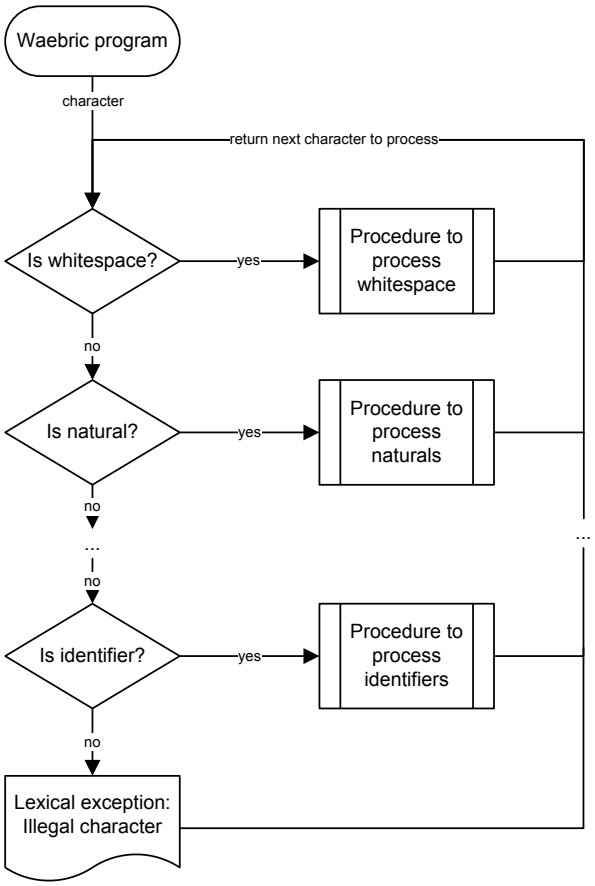

Figure 4.2: Flowchart lexical analysis

4.1.2 Example

Consider the following line of code from a Waebric program:

```
1   echo "Hello world!";
```

For this small example, the process of lexing is as follows:

1. Character *e* is read from the input and evaluated. Evaluation concludes that only identifiers and keywords start with a letter. A new token is created with *e* as value
2. The next character *c* is read from the input and evaluated. Since *c* is a valid character in an identifier/keyword, the character's value is added to the token. The token has now *ec* as value
3. Tokenization continues until a *space* is read from the input. A space is considered

to be invalid for an identifier/keyword, hence tokenization stops. The value of the current token is now *echo*

4. The value of the token is now compared against a list of reserved keywords in order to determine whether it is an identifier or a keyword.

5. The token is converted to a KeywordToken since *echo* is a reserved keyword. The token is added to the tokenlist.

6. Tokenization continues whereby the *space* character is ignored and a *double quote* is read from the input. The double quote is considered as the starting symbol of a Text block, hence a new TextToken is created with an empty string as value. The double quote itself is ignored.

7. All subsequent characters are processed and added to the value of the TextToken until a new double quote appears.

8. The TextToken has now *Hello world!* as value and is added to the tokenlist

9. The semicolon is read from the input, processed as a SymbolToken and added to the tokenlist.

4.1.3 Difficulties

Given the previous example, it is relatively simple to recognize tokens when they occur at the input. There are however circumstances in which the evaluation and tokenization is more complex due to Waebric's syntax notation. Consider the following example in which the output path of a Waebric program is defined in a *site definition*.

```
1 site
2   module/index.htm: main();   //1st site mapping
3   module/about.htm: about()   //2nd site mapping
4 end
```

The problem is that the reserved keyword *module* is used in the path of the output file whereby the lexer will categorize it as a KeywordToken. To prevent such behavior, the last processed KeywordToken is retrieved from the tokenlist and evaluated. If the value equals *site*, then it is assumed that the token is an IdentifierToken since it is part of an output path. While a path can contain keywords, the information that is provided after the path, e.g. *main()*, should *not*. Therefore, an additional condition is necessary which checks whether a colon was already processed. Even more complexity is introduced since more then one output path can be specified in a site definition by using multiple site mappings separated with a semicolon such as shown in the example. As a result, an extra condition was necessary on the semicolon.

While solution is relatively easy in this example, it does affect separation of concern since syntactical knowledge of Waebric is introduced at lexical stage instead of syntactic stage and thus affects portability. A possible solution to prevent such complexity lies in modifying the DSL by which the path should be surrounded by double quotes. In such case, the whole path would be recognized as a TextToken. If modification of DSL is out of order, then an additional lexer could be created which deals with this kind of complexity specifically. In the vanilla implementation is was chosen to leave complexity at lexical stage since no specific recommendations where made towards portability.

4.2 Syntactic analysis

The second phase of an interpreter involves syntactic analysis or parsing. The parser uses the first tokens produced by the lexical analyzer to create a tree-like intermediate representation (AST) that depicts the grammatical structure of the token stream [1].

4.2.1 Parsing algorithm

There are two commonly used methodologies to create parsers: top-down and bottom-up. Top-down parsers build the parse tree from the top (root) to the bottom (leaves) while bottom-up parsers start from the leaves and work their way up to the root. In either case, the input to the parser is scanned from left to right, one token at a time. The vanilla implementation uses a LL(*) parser which is based on the top-down methodology. A LL parses the input from **L**eft to right and constructs a **L**eftmost derivation of the sentence. In other words, a LL parser starts from the root element and then replaces each token with the most left token first. The * in LL(*) means that there is unlimited look-ahead capability.

4.2.2 Abstract Syntax Tree

The abstract syntax tree that is created by the parser captures the essential structure of the input in a tree form, while omitting unnecessary syntactic details [14]. Typically, each interior node in the AST represents an operation and the children of the node represent the arguments of the operation.

The creation of the AST in the vanilla implementation is performed using a bottom-up approach due to the choice for top-down parsing. This may look contradicting, but it is in fact the natural outcome when top-down parsing is applied. When parsing starts, a root node is created. The root node is supposed to produce a value based on the value of its children. The parser has however not visited the leaves of the root node and is therefore unable to produce a value at that moment. Instead, the value is produced after all children of that node are processed. As a result, leave nodes are the first nodes who

produces output and passes this through to their parent. The AST is in other words created bottom-up. The result is an object which contains one or more values retrieved from its children which on their turn contains values of their children and so on.

4.2.3 Design

The design of the parser includes nine subparsers which all process a particular part of the code. Subparsers can create instances of other subparsers using public interfaces in order to acquire data (i.e. AST objects) from child nodes. Figure 4.3 gives an overview of the different subparsers and the subparsers they instantiate.

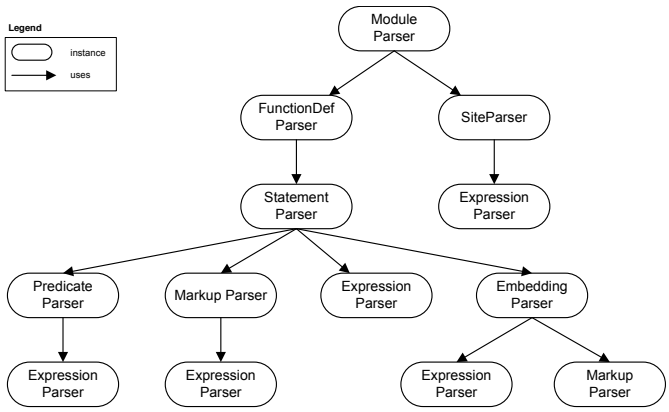

Figure 4.3: Overview of subparsers

Each subparser contains a variable which holds the token that it is currently being processed. The token provides functionality to retrieve the next or previous token in the tokenlist enabling infinite look-back and look-ahead. The token is updated throughout the entire parsing process, even when tokens are processed by instances of other subparsers. The latter is done by passing-in an instance of the parent parser to the subparser. The subparser updates the value of the parent parser after subparsing is completed and then returns the AST object back to the parent parser. The instance of the parent parser in the subparser is also used to prevent the creation of cyclic dependencies and allows separation of data and parsing algorithm, similar to the visitor pattern.

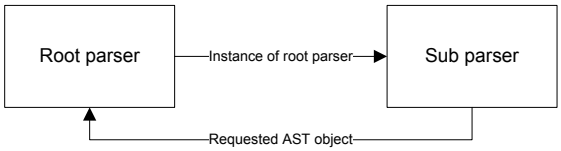

Figure 4.4: Relation parser and subparser

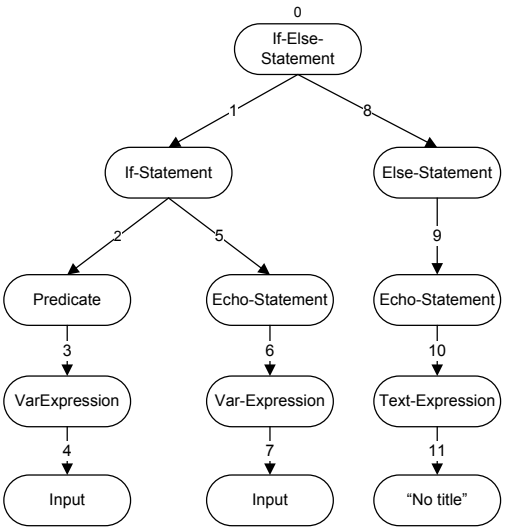

Figure 4.5: AST If-Else-Statement

```
1 if(input)
2   echo input;
3 else
4   echo "No title";
```

Figure 4.6: Syntax If-Else-Statement

Apart from processing tokens, a public interface also provides functionality to evaluate whether a particular token or sequence of tokens corresponds to a certain AST object. The Expression Parser for instance has a public function *isStartIfStatement* in order to evaluate whether the value of the supplied token corresponds to an *if-keyword*. More complex evaluations are included as well.

4.2.4 Example

Figure 4.6 illustrates a code example of a basic If-Else-Statement in Waebric syntax. A visual representation of the corresponding AST is provided in figure 4.5 and includes the sequence of parsing using the numbers on the edges.

The root element of the AST is an If-Else-Statement which contains two arguments, the If-Statement and the Else-Statement. The If-Statement includes two arguments, the predicate which evaluates the expression, and the statements that should be executed when the evaluation of the predicate succeeds. In the example their is only one statement that belongs to the If-Statement, namely an Echo-Statement. The Echo-Statement on its turn contains one argument, namely a value that equals the value of the variable *input*.

The Else-Statement is similar processed, however there is no predicate which needs to be evaluated and the value of the Echo-Statement is not a variable but a TextExpression.

The process of constructing the If-Else-Statement AST is given in figure 4.7.

Parsing starts with consuming the first token in the tokenlist, here $\langle \textbf{keyword}, if \rangle$. The parser now assumes that the following tokens are part of an If-Else-Statement and start

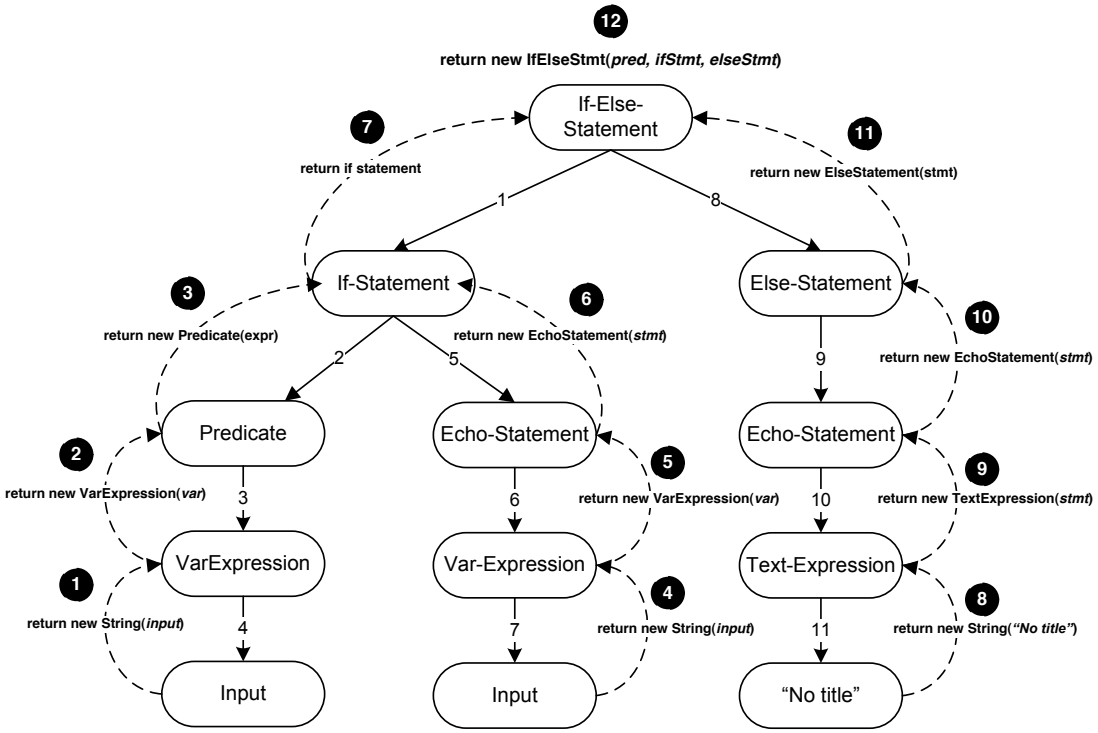

Figure 4.7: Process of creating an AST for the If-Else-Statement

parsing the arguments required to construct an If-Else-Statement, i.e. a predicate, an if-statement and an else-statement.

In Waebric, a keyword if is always followed by a predicate, hence this construction is parsed first. A predicate on its turn always starts with a left bracket and ends with a right bracket. Between the parenthesis, constructions are expected such as an Expression, NotPredicate, TypePredicate, OrPredicate or AndPredicate. As a result, a Predicate AST object has only one value (e.g. a NotPredicate) since left and right brackets are unnecessary syntactic details which are left out of the AST object.

The predicate parser starts matching the first expected token, the left bracket, to the current token $\langle \textbf{symbol}, leftbracket \rangle$ in the tokenlist. Since there is a match, the token is consumed and a new token is taken from the tokenlist. The parser evaluates this token, here $\langle \textbf{identifier}, input \rangle$, and concludes that it is a valid token and it should be processed as an Expression by the Expression parser.

The Expression Parser analyses the current token and concludes that it is a VarExpression since the token is a single identifier. As a result, the token is passed to the Variable parser which creates a new AST object of type VarExpression with as value *input*.

The VarExpression is returned to the Expression parser which on its turn returns the object to the Predicate parser. Note that the Expression Parser does not encapsulate

the value retrieved from the Variable parser into a new object.

The next step includes the encapsulation of the VarExpression object into a Predicate object in order to return it to the If-Else-Statement. Before the Predicate object is returned, the right bracket of the predicate construct must be matched. Since the token $\langle \textbf{symbol}, \textit{rightbracket} \rangle$, appears as next in the tokenlist, parsing succeeds and the object can be returned

The process continues similar for the two remaining arguments (If- and Else-Statement)

4.3 Semantic analysis

Until now, the Waebric program was validated on it's lexical and syntactic correctness. Lexical analysis was based on characters and text blocks (token) while syntactic analysis evaluated the sequence of these tokens in a grammatical context. The last phase involves semantic analysis by which the semantics of the parse tree are evaluated in order to understand its meaning.

Semantic analysis is performed at two levels, typechecking and interpreting. Typechecking involves the the validation of the syntactic well-formedness of the Waebric program while interpreting involves the creation of xHTML markup. Both components uses the parse tree as input.

The common concepts involved in semantic analysis are discussed in section 4.3.1 and 4.3.2 after which the typechecker and interpreter are discussed in section 4.3.3 and 4.3.4 respectively.

4.3.1 Environments

Lexical and syntactic analysis approached the Waebric program statically by storing data into a tokenlist or parse tree. Semantic analysis on the other hand processes its input dynamically since its data may change during execution. An example is a variable which is defined at the beginning of the program but is then changed further in the program. To store this kind of data, a new hierarchical structure was created, also appointed as *environment* objects.

Each environment object stores data such as the variable and functions it contains, the semantic errors that occurred in that environment or the dependencies that were used for that environment. Using a public interface, the interpreter or typechecker are able to add data to or retrieve data from the environment. The environment object itself allows the creation of new environments in order to create a hierarchy of environments with always one root environment containing all other environments.

4.3.2 Tree walking

Semantic analysis requires the traversal of the parse tree. The Visitor Pattern is a commonly used design pattern to perform such task as it separate the (traversal) algorithm from the data. The Visitor pattern enables the definition of a new operation on the AST structure without changing the AST objects themselves. Implementing the Visitor Pattern requires the creation of a visitor which describes how the AST structure should be traversed and the addition of a generic call-back function is each of the AST objects. The visitor contains functions for each AST object in the AST structure that should be traversed. The code that is supplied within these functions determines which child nodes of the corresponding AST object should be visited using the call-back function *accept*.

An example is given in figure 4.8 for a visitor implemented in Java. When the AST structure is passed in to the InterpreterVisitor, then the visitor uses polymorphism in order to distinguish which method in the visitor should be visited. In the current implementation, the AST object *module* would be passed-in, which causes the first visitor function to be executed. The code inside this visitor retrieves the child AST object *funcDef*, and forces this AST object to be traversed as well using the accept function. By passing in an instance of the current InterpreterVisitor using *this*, the accept function will be able to call back the visitor and process the function definition. When this happens, function two of the InterpreterVisitor is executed which contains more code to allow the traversal of the children of the function definition .

```
1  public class InterpreterVisitor{
2
3     public void Visit(Module module){
4        var funcDef = module.functionDefinitions[0]
5        funcDef.accept(this)
6     }
7     public void Visit(FunctionDef func){
8        //code
9     }
10    public void Visit(IfElseStmt stmt){
11       //code
12    }
13 }
```

Figure 4.8: Traditional visitor pattern

The implementation of the Visitor Pattern in JavaScript is however different since JavaScript does not provides polymorphism due to its weakly typed design. As a result, the Visitor Pattern was modified in order to benefit from the separation between data and algorithm. The modifications involves the creation of different visitor objects for each AST object such as presented in figure 4.9. Due to the modifications in the Vis-

```
function InterpreterVisitor(){

  function ModuleVisitor(env){
     this.env = env
     this.visit = function(module){
        var funcDef = module.functionDefinitions[0]
        funcDef.accept(new FuncDefVisitor(this.env))
     }
  }
  function FuncDefVisitor(env){
     this.env = env
     this.visit = function(funcDef){
        //code
     }
  }
  function IfElseStmt(env){
     this.env = env
     this.visit = function(ifElseStmt){
        //code
     }
  }
}
```

Figure 4.9: Modified Visitor Pattern

itor Pattern, the corresponding visitor object should be *instantiated* when calling the *accept* function instead of passing through the complete visitor using the *this* keyword. Note that the modifications do involve some code overhead compared to the traditional Visitor Pattern.

A second modification to the Visitor Pattern involves the use of parameterized visitor objects which accepts a variable in order to pass additional data. This modification is not specific for JavaScript since it can be implemented in the traditional pattern as well. Passing additional data is desirable in order to provide each visitor object access to the environment object to store semantic information such as discussed in section 4.3.1. The alternative involved the use of a global environment object for the complete visitor, but had the disadvantage that the currently processed environment should be updated continuously which is more error prone.

4.3.3 Typechecking

A typechecker evaluates the meaning of the parsing tree in order to provide warning messages towards the end-user when certain constructs in the language are not valid. An example of a semantic error is a duplicate function definition. A semantic error may affect the output of the interpreter, but it may never prevent the interpreter to produce

output. The following semantic errors are considered for Waebric [32]:

Undefined functions If for a markup-call, f , no function definition can be found in the current module or one of its (transitive) imports, and, if f is not a tag defined in the XHTML 1.0 Transitional standard, then this is an error.

Duplicate functions Multiple function definitions with the same name are disallowed.

Arity mismatches If a function is called with an incorrect number of arguments (as follows from its definition), this is an error.

Undefined variables If a variable reference x cannot be traced back to an enclosing let-definition or function parameter, this is an error.

Non-existing module If for an import directive import m no corresponding file m.wae can be found, this a an error.

The typechecker here presented implements the Visitor Pattern in order to traverse through the parsing tree and adds additional code inside the visitor objects to perform the typechecking. First, semantic information from the AST is stored inside the environments objects whereas later the semantic information is validated for generating previously mentioned errors. The visitor is built in such way that each node in the parse tree is visited, starting from the top.

The process of typechecking is illustrated using an example which validates the Waebric program for undefined variables. First, the typechecker collects all *defined* variables from the parse tree and adds them to the environments objects. In Waebric, variables are defined inside the formals of a function definition, in an each statement or in a variable binding construction. A code snippet is provided in figure 4.10 which collects variables from the arguments (formals) of a function definition.

```
1  function FuncDefVisitor(env){
2     this.env = env
3     this.visit = function(funcDef){
4        new_env = this.env.addEnvironment('func-def');
5        for (var i = 0; i < funcDef.formals.length; i++) {
6           var formal = funcDef.formals[i];
7           new_env.addVariable(formal);
8        }
9     }
10 }
```

Figure 4.10: Storing arguments of function as semantic information

After data collection, the typechecker visits the AST nodes at which a variable can be accessed (variable references) in order to evaluate whether that variable was earlier defined in the environment. If for some reason the variable isn't found in the environment (e.g. due to spelling mismatch), then a warning message is generated towards the end-user that the variable is undefined. Figure 4.11 illustrates the validation process of a variable reference using the VarExpressionVisitor.

```
1 function VarExpressionVisitor(env){
2    this.env = env
3    this.visit = function(variable){
4       //Check if variable exists
5       if (this.env.getVariable(variable) == null) {
6          this.env.addException(new UndefVarException(variable,
               this.env));
7       }
8    }
9 }
```

Figure 4.11: Validating whether a variable is defined

4.3.4 Interpreting

Interpreting is the final phase in the vanilla implementation and converts the parse tree to xHTML code. The interpreter is implemented similar to the typechecker using the Visitor Pattern and environment objects to store semantic information.

Process

Section 4.3.2 discussed that the parse tree is traversed with the use of the Visitor Pattern. What the pattern does not describes is in which sequence the AST nodes should be visited. For the purpose of the typechecker, the *complete* parsing tree was visited in a sequence similar to that of the LL parsers. Traversing through the complete parsing tree is however not desirable in the interpreter since certain AST nodes are not required to be visited. An example is an If-Else-Statement by which the interpreter evaluates the predicate in order to determine the branch that should be executed. If the complete parsing tree is traversed in an interpreter, then both branches would be visited resulting in unwanted xHTML markup. Therefore, the interpreter only visit AST nodes required for generating the correct xHTML markup. This is one for the main reasons why typechecking is not performed during interpreting itself.

An example of the visitor for an If-Else-Statements is given in figure 4.12. First, the value of the predicate is evaluated using an auxiliary subroutine. If evaluation succeeds,

then the If-Statement is visited and only xHTML code is generated which is provided in the If branch.

Note that apart from the *environment* object a second parameter is passed to the visitor object, namely the *dom* object to allow a visitor to generate xHTML code.

```
1 if (isValidPredicate(ifElseStmt.predicate, this.env)) {
2    ifElseStmt.ifStmt.accept(new StmtVisitor(this.env, this.dom));
3 } else {
4    ifElseStmt.elseStmt.accept(new StmtVisitor(this.env,this.dom));
5 }
```

Figure 4.12: Visiting AST nodes in the interpreter

The interpreter also differs from the typechecking visitor in such way that the start point of traversal is different. The typechecker always starts from the root element and traverses through all its children. The interpreter however starts traversal from the *main* function in the Waebric program. If a site definition is specified, then an additional traversal is performed for each output path specified in the site definition.

Markup generation

The JavaScript language is typically associated with a browser environment. In a browser, JavaScript is able to create or modify HTML, xHTML or XML code using the API of the Document Object Model (DOM). Initially it was meant to reuse this functionality in order to create the xHTML code in the interpreter. The DOM is however provided by the browser itself and not by JavaScript. As a result, using the DOM of the browser would prevent the use of the interpreter outside the browser using 'off-line' JavaScript engines such as Rhino or the V8 JavaScript Engine.

The first solution that came to mind included the use of an external library which provides similar functionality as the DOM. However, such libraries don't exists for JavaScript, probably because most JavaScript code is developed for a browser which already support xHTML creation natively using the DOM. The use of external libraries in other programming languages was also considered, in particular JDOM which is a Java implementation of the DOM. This would however limit the use of the interpreter to only Java-based JavaScript engines such as Rhino. Moreover, the research method in section 3 required that each implementation limited the use of external libraries written in other languages as much as possible.

Another solution included the use of String objects in order to construct the xHTML code. The disadvantage of such approach is that there is no possibility to traverse through the xHTML code or to retrieve attributes of certain xHTML elements and as a result would complicate the implementation of the interpreter.

Eventually the ENV-JS[1] library was chosen which simulates a complete browser environment using JavaScript code, including the DOM object. The browser environment is loaded in the JavaScript engine and then creates a global object *document* by which the DOM API can be accessed through. This allows the interpreter to generate markup.

Semantic errors

Semantic errors found during typechecking may not prevent the interpreter to produce its output. As a result, a recovery model was created which defines which action should be taken in case of a semantic error [32]. These actions should allow the interpreter to continue interpreting.

Undefined functions A markup-call to an undefined function is interpreted as if it was part of xHTML 1.0 transitional.

Duplicate functions Only the first function definition is processed. Subsequent function definitions with the same function name are skipped.

Arity mismatches If a function is called with more actual arguments than the number of formal arguments, the superfluous arguments are ignored. If the actual arguments are too few in number, the remaining formal parameters become undefined variables.

Undefined variables The value of the variable is evaluated as the string "undef".

Non-existing module The imported module is skipped.

[1] `http://ejohn.org/blog/bringing-the-browser-to-the-server`

Chapter 5

OMeta implementation

This chapter discusses the implementation of Waebric using the language development tool OMeta/JS. The process is similar to the vanilla implementation except that certain phases are automated, in particular lexing, parsing and typechecking. An overview is provided in figure 5.1. The lexer and parser are developed at once based on a formal specification. A second specification was developed in order to create the typechecker. The interpreter was reused from the vanilla implementation.

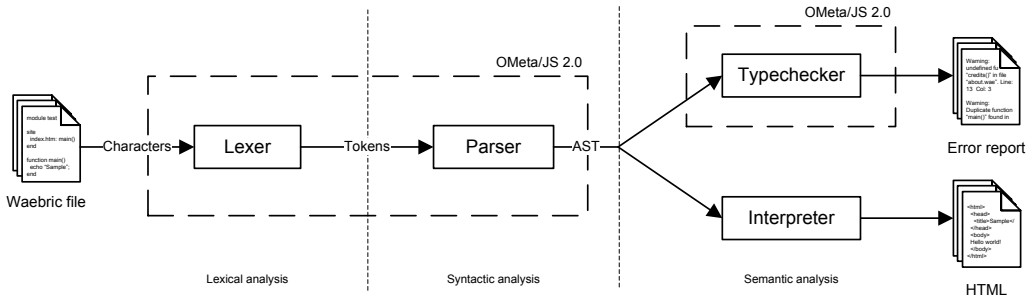

Figure 5.1: Overview of the OMeta implementation

5.1 Lexical and syntactic analysis

OMeta performs lexical and syntactic analysis at once and uses only a single specification in order to generate the parser. The parser itself is generated by the OMeta Base translator which takes as input the formal specification and generate as output the parser object. The generated parser is then accessed using a public interface in order to process the Waebric program.

The parser produces a parse tree based on the semantic action code provided in the

grammar. This enables the creation of user-defined parse trees, including statically and dynamically typed parse trees. The OMeta implementation uses a statically typed parse tree, identical to the one used in the vanilla implementation. This allows the reuse of the vanilla interpreter since it operates on the level of the parse tree.

The design of the OMeta parser is illustrated in figure 5.2.

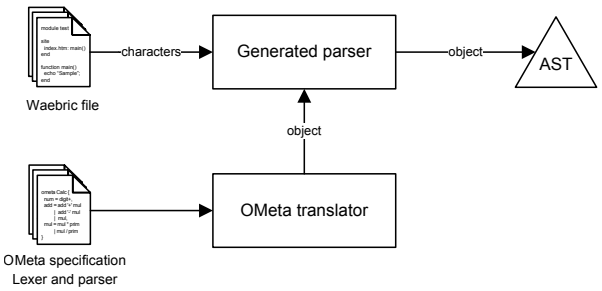

Figure 5.2: Design OMeta parser

5.1.1 Example

A subset of the grammar for the Waebric parser is shown in figure 5.3. The example contains 3 production rules whereof the first production rule is the root element. The root element consists out of the terminal *"module"* and the non-terminals *ModuleId* and *ModuleElements*. These non-terminals are defined by production rule 2 and 3 respectively. A *ModuleId* consists out of a series of identifiers separated by a dot. The *ModuleElement* is defined as the non-terminal Site, FunctionDef or Import using the choice operator.

```
1 Module        =  "module" ModuleId:id ModuleElement*:elements
2               -> new Module(id, elements),
3
4 ModuleId      =  listOf(#IdCon, "."):moduleId
5               -> new ModuleId(moduleId.join('.'))
6
7 ModuleElement =  Site | FunctionDef | Import
```

Figure 5.3: Code example OMeta parser

The matching process in OMeta is illustrated by applying the Waebric program in figure 5.4 to the grammar in figure 5.3. The parsing algorithm used in OMeta is a packrat parser which is similar to a recursive descent parser such as LL(k). As a result, OMeta parses the Waebric program using a top-down approach while the AST is created bottom-up.

The parser starts by matching the Waebric program to the root element, here the non-

terminal *module*. In this production rule, the first expected input is the non-terminal *module*. Since the input starts with the text module, there is a match.

Matching continues and expects the non-terminal *ModuleId*. Since this is a non-terminal, the second production rule is applied. In this production rule, a list of identifiers is expected separated with a dot. The input *homepage.cwi.nl* equals the expected input, and thus the non-terminal ModuleId is successfully matched. As a result, the semantic code is executed which stores the module name in a new AST object and returns it to the parent rule. As a result, the AST object is binded to the variable *id*.

```
1 module homepage.cwi.nl
2
3 def main
4   //markup
5 end
```

Figure 5.4: Waebric example

The last non-terminal is now matched in the first production rule whereby a collection of zero or more *ModuleElement* is expected. Since *ModuleElement* is a non-terminal, the third production rule is applied. This involves the matching of the non-terminal *Site*, *FunctionDef* or *Import*. For the given example, the non-terminal *FunctionDef* would be applied but is not shown in the grammar. If the *FunctionDef* is considered to be matched, then the value is returned to the parent rule and binded to the variable *moduleElements* in the first production rule. This variable is by default an array as a result of the * operator in order to store each module element separately.

Eventually, the complete root element is matched and the semantic action code can be executed. This involves the creation of a new AST object *Module* which takes as arguments the variables *id* and *elements*.

5.2 Semantic analysis

Semantic analysis in OMeta only includes the creation of a typechecker since the interpreter is lend from the vanilla implementation. The OMeta typechecker performs the same validation steps as the ones discussed in section 4.3.3 but uses a different mechanism to traverse through the parse tree. The following sections discusses this mechanism and illustrate its operation using a practical example.

5.2.1 Tree walking

In OMeta it is relatively simple to traverse through the parse tree since such functionality is natively supported by OMeta. Traversing is done by performing pattern matching on list objects. Lists are the very basic form by which a parse tree can be created, e.g. ['add', x, y]. An example was given in the background section in figure 2.6.

The main problem with list-based parse trees is that it prevents the implementor from using its own defined parse tree. This can be counteracted by creating two grammars, one grammar that generates the list-based parse tree and a second one which performs pattern matching on this parse tree to transform it into user-defined AST objects. The creation of a typechecker would involve the creation of a third grammar which performs pattern matching on the list-based parse tree in order to find semantic errors.

The presented solution does however introduce two new problems. At first, it requires the use of an intermediate grammar which converts the list-based parse tree to a user-defined parse tree. This might be considered as overhead since an extra grammar needs to be maintained. Secondly, it requires that the intermediate grammar and the typechecker grammar performs pattern matching on each production rule of the parser. This means that the intermediate and typechecker grammar are required to define the same amount of production rules as the parser. Whenever a modification is made to the grammar of the parser, it may require modification to both the intermediate and typechecker grammar, which is adversely for maintainability.

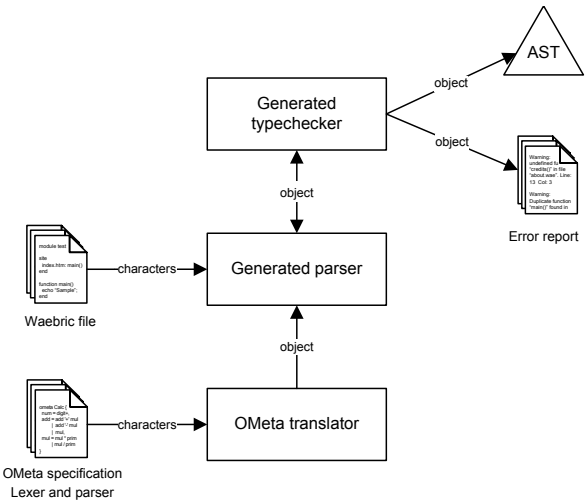

Figure 5.5: Design OMeta typechecker

At last, a complete other solution was considered which is eventually used for the creation of the typechecker. The solution involves the use of the inheritance structure of OMeta by which one grammar inherits from an existing grammar. Using inheritance,

the new grammar inherits all production rules of the base grammar, including the semantic predicates and action code. The inheritance also allowed the reuse of the parsing algorithm of the parser whereby a visitor pattern was unnecessary.

For the typechecker this means that a new grammar was created which inherits from the parser. Then, for each production rule wherefore a semantic action is required, i.e. gathering or validating semantic information, the production rule of the parser is overridden. Note that only semantic action code is needed for the typechecker to operate. In order to let the production body intact, i.e. identical to the one of the parent grammar, a super-send statement is used. The super statement prevents in such way any duplicate code.

The solution has the advantage that only two grammars need to be developed and maintained, and that the grammar of the typechecker is kept to the bare minimum since it only contains production rules for which validation was needed. Most effort was required to produce the action code. The action code was therefore transferred to external libraries, and function calls were made from the semantic action code to these libraries. This keeps the typechecker clear and maintainable.

5.2.2 Example

The operation of the typechecker is illustrated by performing a validation on undefined variables. The validation is divided into two phases: collection and validation of semantic information.

The collection of variables is performed by adding semantic action code at places where variables are defined, such as in the arguments of a function definition, also named as *formals*. First, the existing production rule is overridden by redefining the non-terminal *formals* at the left-hand-side of the rule and by apply the production body of the parent rule using the super-send operator at the right-hand-side. Then, the result of the rule application is binded to a variable. The value of the variable is produced by the semantic action code in the parser. The last phase adds semantic action code to the newly created production rule to add the variable to the current environment.

An example of data collection is given in figure 5.6 and adds for each formal a variable to the environment object. The last line of code in the action code is required by OMeta to return the original AST object back to the parent grammar.

```
Formals =  ^Formals:formals
        -> {
            //Add each variable in the formals to the environment
            for (var i = 0; i < formals.length; i++) {
               var formal = formals[i];
               WaebricOMetaValidator.environment.addVariable(
                  formal);
            }
            //Return the original AST object
            formals;
           },
```

Figure 5.6: Collection of semantic information

```
Expr =  Text:idText -> new TextExpr(idText)
     |  IdCon:idCon -> new VarExpr(idCon)
     |  NatCon:idCon -> new NatExpr(idCon)
     |  SymbolCon:idCon -> new SymbolExpr(idCon)
     ...
```

Figure 5.7: Parsing an expression in the OMeta parser

The last phase during typechecking validates the variables itself when they are used. In Waebric, a variable is part of the production rule *Expr* (see figure 5.7).

The intention is to add action code only to the variable expression but not to the other expressions. This is achieved by evaluating the type of AST object inside the semantic action code using *instanceof*. If evaluation succeeds, then an external library is used to determine whether the variable is defined in the current environment or not. A fragment of this external library is given in figure 5.8.

```
Expr =  ^Expr:expr
    -> {
        //Validate the VarExpression AST object
        if(expr instanceof VarExpr){
          validator.validateVarRef(expr, WaebricOMetaValidator)
        }
        //Return the original AST object
        expr;
       }
```

Figure 5.8: Validating a variable in the OMeta typechecker

```
function WaebricVariableValidator(parser){
  this.parser = parser;

  this.validateVariableReference = function(varExpr){
    //Search variable in current environment
    if (!this.parser.env.containsVar(varExpr.variable)) {
      //Create semantic exception
      var exception = new UndefVarException(varExpr.variable, this
          .parser.environment);
      parser.env.addException(exception);
    }
  }
}
```

Figure 5.9: Fragment external libraries OMeta typechecker

Chapter 6

Results

This chapter presents an overview of the metrics extracted from the vanilla and OMeta implementation. A classification is made between maintainability, performance and functionality.

6.1 Maintainability

The maintainability of each implementation is determined based on the metrics provided in section 3.1. The extraction of metrics is automated using the metric tool JSMeter[1]. The tool was modified in order to comply with the counting rules provides in section 3.1. The metrics for grammar-based code were gathered manually based on the metric suite of Power and Malloy.

The following sections discusses the most important maintainability measures into detail. A complete overview of all extracted metrics is given in appendix A.

6.1.1 Vanilla implementation

An overview of the maintainability metrics extracted from the vanilla implementation is given in table 6.1.

The measurements show that the parser is the largest component in the implementation with almost 1400 effective lines of code, followed by the interpreter with 792 lines. The sizes of the remaining modules are nearly identical with 431 lines for the lexer, 447 for the typechecker and 473 for the parse tree. The AST clearly uses more methods then other modules in relation with its size with a total of 131. The parser has a few more methods (163) but is almost 3 times larger. The high number of methods in the parser

[1] `http://code.google.com/p/jsmeter`

Metrics	Lexer	Parser	AST	Checker	Interpreter
Lines of code	538,0	1676,0	609,0	606,0	1.018,0
Effective lines of code	431,0	1.397,0	447,0	473,0	792,0
Method count	63,0	163,0	131,0	90,0	124,0
Average method size	8,5	10,3	4,7	6,7	8,2
Cyclomatic Complexity	3,0	3,3	1,9	1,9	2,6
Maintainability Index 3	83,5	74,8	104,2	91,3	83,1
Maintainability Index 4	124,9	114,2	148,0	131,5	123,5
Halstead Effort	21.901,2	357.691,5	46.005,3	73.821,8	200.908,5

Table 6.1: Maintainability metrics vanilla implementation

has its reflection on distribution of the code with an average method size of only 5. The parser on the contrary distributes its code less well with more than twice as many lines of code per method.

When the complexity of each module is considered, measurements shows that the parse tree and typechecker have the lowest complexity number of all modules with less then 2 branches per method on average. The interpreter has a slightly higher complexity number (2,6) while the lexer and parser have the highest complexity with an average of 3 branches per method.

The overall maintainability of a module was measured with the MI metric whereby a higher value indicates a better maintainability. The AST module provides the best results with an MI3 of 104 and an MI4 of 148 when considering comments. The second best result is on name of the typechecker with values of 91,3 (MI3) and 131,5 (MI4). The parser has the lowest maintenance index of all modules with a MI3 of 74,8 and a MI4 of 114,2. Still, MI results are well within the established cut-off values defined in table 3.4.

Finally, the Halstead effort was measured in order to establish an insight to the effort required to implement each module. The parser requires most effort of all modules with a value of over 350.000 which corresponds to a 50% of the total halstead effort for the vanilla implementation. The interpreter has a Halstead effort of approximately 200.000 resulting in a 28,6% share. The typechecker accounts for 10,5%, while AST and lexer accounts for the remaining 13,6%.

6.1.2 OMeta implementation

The results of the OMeta implementation are classified per module in order to keep results clear. The code of each module is measured in four categories. First, measurement is performed at the terminals and non-terminals of the formal specification that corresponds to the module using the counting rules of Power and Malloy. The second category measures the semantic information in the grammar such as action code and predicates. External libraries used in the semantic code are measured in the third category. A final classification is made for code which is written around the module such

as 'factory classes' or exception objects. The semantic action code that corresponds to a production rule is counted as one function.

Note that the LOC and eLOC measures for grammars are not based on the metric suite of Power and Malloy's. The LOC and eLOC metric are calculated using the counting rules for program code in section 3.1. The LOC* is a derived metric from the metric suite and provides better results to compare with program code. The LOC* metric is only listed for the grammar.

Parser

An overview of the metrics extracted from the OMeta parser is given in figure 6.2.

The grammar is the largest component in the implementation and consist out of 130 lines of code whereof 105 are effective. The LOC* metric indicates an even higher value with 384 lines of code. The size of the semantic action code and predicates is only slightly smaller compared to the grammar with a total of 92 effective lines. The size of the libraries is limited to only 71 effective lines and 10 methods.

The complexity of the grammar is slightly higher compared to the semantics it contains, but it should be noted that the grammar uses a different complexity measure then the three other categories of code where the complexity is at least one instead of zero. The maintainability of the semantics is the highest of all categories, followed by the additional code for the parser and the grammar itself. The libraries have the lowest Maintainability Index with a value of 120.6, which is still highly maintainable.

The Halstead metrics show that 61,8% of the effort is spent on the creation of the non-semantics of the grammar while the semantic code itself requires 27,6% of total effort. The external libraries and supporting code accounts for the remaining 10.4% effort.

Metrics	Grammar	Semantics	Libraries	Other
Lines of code	130,0	98,0	87,0	46,0
Lines of code*	384,0	N/A	N/A	N/A
Effective lines of code	105,0	92,0	71,0	36,0
Method count	60,0	76,0	10,0	7,0
Average method size	6,4	1,3	8,7	6,6
Cyclomatic Complexity	1,8	1,1	2,1	2,3
Maintainability Index 3	92,0	133,0	80,7	94,1
Maintainability Index 4	115,8	153,5	120,6	138,6
Halstead Effort	94.342,2	42.141,0	13.527,9	2.581,0

Table 6.2: Maintainability metrics OMeta parser

Typechecker

This section discusses the results in table 6.3 extracted from the OMeta typechecker.

What immediately strikes to the results is the low number of LOC and eLOC by which the typechecker was developed, particularly the grammar itself which counts only 11 effective lines of code. If LOC* is considered, then 22 lines of code are measured which is still low. The libraries represent the largest amount of code with a total of 170 effective lines. There is a total of ten production rules defined and based on the production body there are approximately 2 lines of code per method (AMS). Note that the LOC or eLOC measures are not used to establish the average method size in a grammar since AMS is calculated using the metric suite.

The complexity measure for the grammar equals zero since no branch operators were used in the grammar. The complexity of the remaining code is considered low, ranging between 1,7 to 2,3 branches per method. The Maintainability Index indicates that the grammar has highly maintainable code with MI3 set to 131,3. The Halstead measures indicates that most effort was required for constructing the semantics (53%) and the libraries (36,9%) while the grammar only required 2% in the total effort for this module.

Metrics	Grammar	Semantics	Libraries	Other
Lines of code	13,0	61,0	197,0	59,0
Lines of code*	22,0	N/A	N/A	N/A
Effective lines of code	11,0	35,0	170,0	46,0
Method count	10,0	10,0	20,0	9,0
Average method size	2,0	6,1	9,9	6,6
Cyclomatic Complexity	0,0	1,7	1,9	2,3
Maintainability Index 3	131,3	92,0	80,9	93,2
Maintainability Index 4	172,7	127,2	112,7	138,4
Halstead Effort	934,5	26.240,4	18.264,0	4.111,4

Table 6.3: Maintainability metrics OMeta typechecker

6.2 Efficiency

Execution speed was measured for each main component in the implementation by executing 8 Waebric programs ranging from 54 to 10.000 lines of code. Each test was carried out 100 times and the average number was taken for comparison. The higher the execution speed, the slower a Waebric program is interpreted to xHTML code.

The first series of tests uses three demo Waebric programs with sizes ranging from 54 to 967 lines of code. The second series of tests uses five Waebric programs with sizes ranging from 1000 to 10.000 lines of code whereby each line of code contains an identical echo-statement.

All tests were performed with the following computer configuration:

Computer configuration	
Processor	2 x Intel(R) Xeon(R) CPU E5420 @ 2.50GHz
Memory	2 x 2048MB DDR2 PC2-5300 ECC
Operation System	Microsoft Windows Server 2003 SBS
JavaScript engine	Rhino 1.7R2 (2009-03-22)
Java Virtual Machine	Java 1.6.0_15 (Sun Microsystems Inc)

Table 6.4: Computer configuration

Table 6.5 lists the execution speeds for the OMeta and vanilla implementation using three demo programs. The results of the vanilla lexer and parser are added together in order to compare the results with the OMeta parser equally.

The results show clear differences between each implementation, all of them in favor of the vanilla implementation. The vanilla lexer/parser is 2 to 6 times faster compared to its opponent in the OMeta implementation while the typechecker shows an even greater difference, up to a factor 16 for the largest Waebric program. The results of the interpreter are almost identical since this is a shared component.

Both implementations spend most time on executing the lexical and syntactic analysis with values ranging between 60 and 85 percent depending on input size. The execution time of the OMeta parser and typechecker are highly sensitive to their input with a factor 24 and 38. The interpreter is the least sensitive to program input with only a factor two difference.

Waebric program	**Lexer/Parser**		**Typechecker**		**Interpreter**	
	Vanilla	OMeta	Vanilla	OMeta	Vanilla	OMeta
Lava (967 LOC)	3.365,4	19.514,9	155,4	2.561.2	881,8	802,7
LDTA (269 LOC)	1.222,2	6.118,2	68,7	792,4	593,1	543,6
Menus (54 LOC)	354,4	812,7	34,2	65,9	424,7	445,2

Table 6.5: Comparison execution speed (ms) - series 1

The execution times for the vanilla and OMeta implementation under stress test are given in figure 6.1 and 6.2 respectively. The figure includes the relative execution times in order to examine the scalability of each component individually as input size grows. The absolute time values are given in figure 6.3.

The vanilla implementation executes the stress test in linear time whereby the smallest test file was interpreted in less then two seconds and the largest in 14 seconds. Most resources are spent on lexing with 45% to 75% of the total execution time, followed by the parsing which takes around 20 to 25%. The execution time of the typechecker is the lowest of all components and occupy only 2 to 7% of the total recourses. The remaining execution time is spend on interpreting and takes between 10 and 25%. Note that as the Waebric program grows, the relative execution time of the lexer increases in favor of the remaining components.

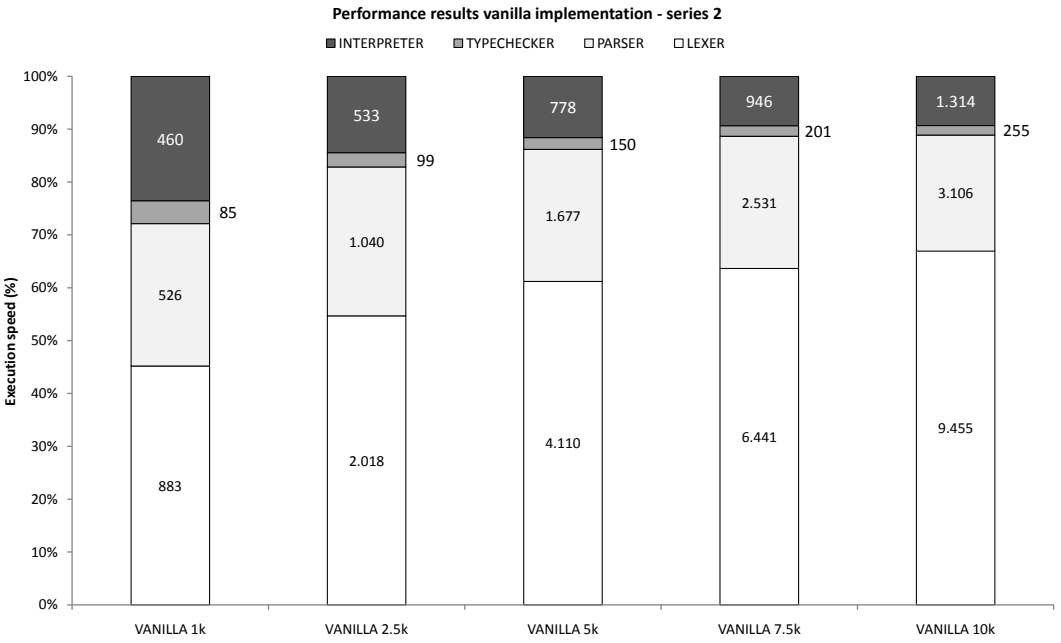

Figure 6.1: Execution time vanilla implementation

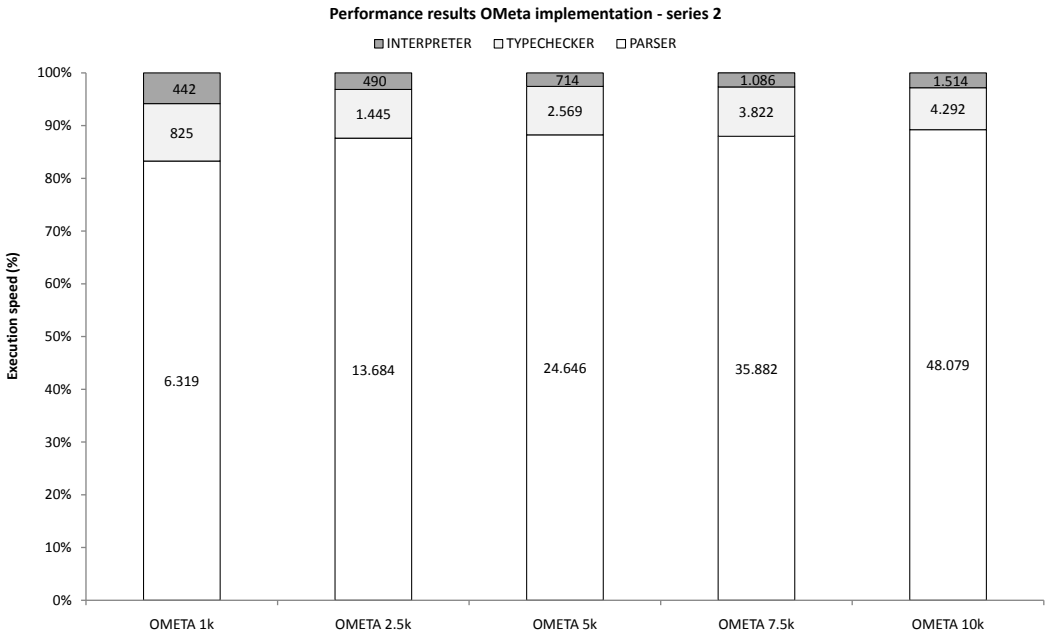

Figure 6.2: Execution time OMeta implementation

The results for the OMeta implementation are given in figure 6.2. The chart is mainly dominated by the execution times from the parser with values between 83 and 92% of the total time. The time ratio for each component remains relatevily stable over the different tests with only a small increase at the parser in favor of the typechecker and interpreter. The absolute values of the measures indicate that the OMeta implementation runs in linear time. The smallest test file was interpreted in 7,6 seconds whereas the largest test file required no less than 54 seconds.

The ratio of parsing in OMeta differs little from lexing/parsing in the vanilla implementation, while typechecking takes - relatively speaking - 2,5 times longer in the OMeta implementation. The execution time of the interpreter is equally in both implementation and since the vanilla implementation is faster in absolute numbers, it follows that the vanilla interpreter takes a larger portion in the total execution time, between 10 and 20% compared to 2 to 5% in OMeta.

An overview of the execution times per module and per implementation in absolute numbers is given in figure 6.3. The figure illustrates the linear progress of both implementations. It is also clear from the figure that the vanilla implementation is much faster then OMeta by a factor 4 in difference.

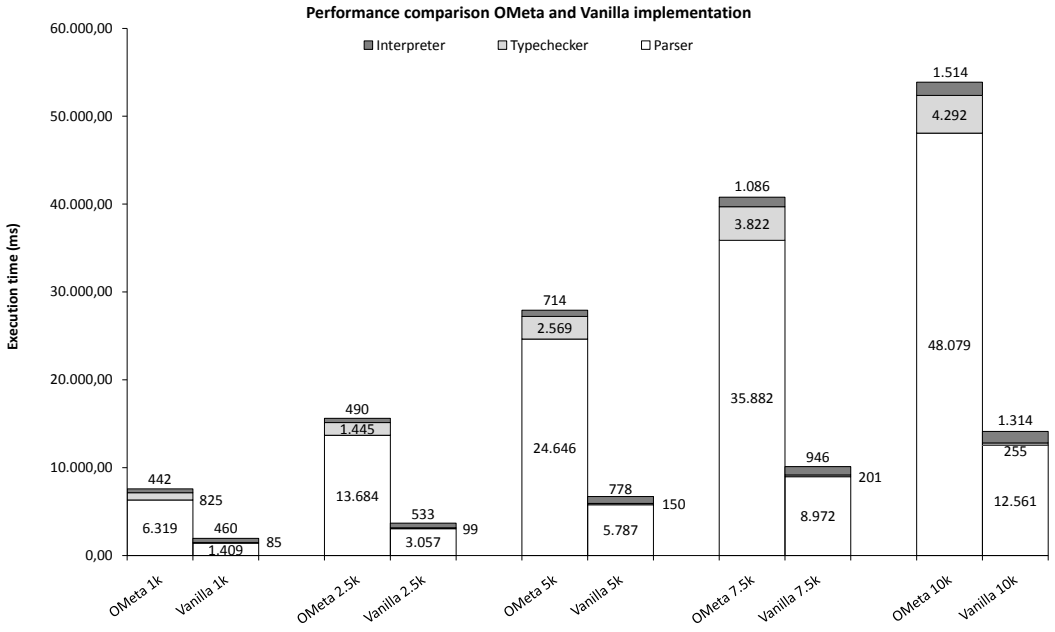

Figure 6.3: Comparison execution times

6.3 Functionality

Functionality is measured using a series of acceptance tests and the level of error reporting.

The acceptance tests are performed by running a collection of 104 acceptance tests in each implementation. Each of these tests exercises a particular functionality of the DSL and results in a failure or acceptance. Additionally, there are three demo Waebric programs added to the collection of acceptance tests to achieve full code coverage. Results show that each acceptance test succeeded in both implementations, inclusive the three additional Waebric programs. It follows that both implementations support the complete Waebric notation.

The second level at which functionality was measured is the support of error reporting towards the end-users. Results show that OMeta is only capable of throwing the line position of the error. The column number or a relevant part of the parser stack are not provided by OMeta and the only error description given towards the end-user is "matched failed". The error message may be 'correct', but it doesn't address the concrete problem, nor addresses it the error in the terms of the problem domain.

Better results are acquired in the vanilla implementation where 93% of the test files result in a correct error message with corresponding parser stack, line/column position and a description in terms of the problem domain. One error was unexpected using an illegal character in the symboltext while an error in an embedding context failed at lexical level and could therefore not provide a parser stack.

A complete evaluation of the error messages is provided in appendix A

Chapter 7

Analysis

This chapter analyzes and interprets the results presented in chapter 6. The analysis is categorized into maintainability, efficiency and functionality.

7.1 Maintainability

The maintainability metrics extracted from the implementations presents significant differences, especially for the lines of code and Halstead Effort metrics. The construction of the OMeta parser only required 304 effective lines of code* which is significant lower then its opponent in the vanilla implementation with more then 2200 lines of code, a factor 6.1 in difference. This can be explained by the higher abstraction level of the OMeta grammar which is more expressiveness compared to program code. A higher expressiveness usually results in more complex code and thus affects maintainability negatively. The level of abstraction can be counteracted using the LOC* metric which uses the method count (VAR) and average method size (RHS) metrics to calculate the lines of code in a grammar. When the LOC* metric is applied, the difference in code size decreases to a factor 5.7. In either case it's clear that the OMeta parser uses fewer lines of code which potentially results in more maintainable code.

The average method size indicates that the OMeta parser contains less code per function compared to the vanilla lexer/parser. The low value is partially the result of the semantic action code which creates the AST objects or performs limited string operations. If only the non-semantics of the grammar are considered, then the AMS metric is more realistic with values ranging between 6 and 7 lines of code per function. This number is relatively high since whitespaces between the terminals and non-terminals in a production body should be defined *explicitly*. Other formal specifications such as ANTLR or SDF ignore whitespace *implicitly* by defining the production rule as *context-free syntax*. This requires two languages, one for the lexical syntax and one for the context-free syntax. For this reason, OMeta has chosen not to implement this mechanism in order to create lexers

and parsers using the same language. The disadvantage is however that the size of the production body increases. The more terminals and non-terminals in a production rule, the more error prone the grammar. Note that this effect can be *minimized* by skipping whitespaces as much as possible at lexical production rules (i.e. rules that consume keywords, literals, etc).

The values of the Extended Cyclomatic Complexity metric show that the vanilla parser uses more branches compared to the OMeta parser. This is understandable since the vanilla parser requires many if/else statements to evaluate whether the tokens in the tokenlist are positioned in the correct sequence. These evaluations also allows the implementor to define user-defined error messages whenever an evaluation would fail. The downside is that complexity increases if the ECC metric is considered.

OMeta performs most evaluations implicit due to the higher abstraction level at which OMeta operates. The only complexity that is introduced inside the grammar is caused by the choice or repetition operator. Since these operators are not much used inside the grammar, it follows that the complexity is low.

The low complexity of the OMeta parser is also noticeable in the Maintainability Index and Halstead effort. The MI indicates higher maintainable code for the OMeta implementation while Halstead Effort indicate that the vanilla parser requires three times more development effort compared to the OMeta parser, which is in line with previous observations.

The maintainability of the typechecker is remarkable similar for both implementation. The only significant value is the lines of code metric which shows a difference of a factor 1,8 whereby OMeta has the smallest code base. The application of the LOC* metric has little or no impact on code size since most code is situated in the libraries and not in the grammar.

The cause of the limited code size in the OMeta typechecker is a direct consequence of the reuse of the tree walking mechanism from the parser using OMeta's inheritance capabilities. This has positive impact on maintainability since the traversal of the typechecker automatically adapts and evolves when the OMeta parser is changed. The vanilla typechecker on the other hand implemented the tree walking mechanism by itself using the visitor pattern through which more code was necessary. This has impact on maintainability twice. First there is more code to maintain, and second, if something changes at the parser (e.g. new or modified Waebric notations) it requires changes at the typechecker as well.

Finally, the Halstead metric indicates that the OMeta typechecker requires 30% less effort compared to the vanilla typechecker. This is however not significant enough to draw general conclusions.

7.2 Efficiency

From the results it's clear that OMeta translates Waebric programs at a much slower speed compared to the vanilla implementation. Despite the poor results, OMeta is still able to operate in linear time due to the use of a packrat parser [8] which memoizing the results of intermediate parsing steps to allow backtracking. [35].

The poor performance results might be attributed to the backtracking functionality that is provided by OMeta. Generally, backtracking affects performance negatively and should be avoided. The only grammar that uses backtracking is the OMeta parser which includes seven look-ahead operators which might have affect performance during tests. These operators are necessary to distinguish certain rules from each other and cannot be left out of the grammar. The look-ahead operators are almost always positioned at the end of a production to indicate that a rule may not be followed by a particular character or string. Since backtracking is only required on a string and not on a complete rule, it is believed that the look-ahead operators have little impact on performance. Moreover, the code review that took place with the creator of OMeta didn't pointed out harmful constructions in the grammar which might affect performance negatively.

Since backtracking is not the main cause for poor performance, it is assumed that OMeta's memoization mechanism is the culprit. The storage of intermediate parsing steps causes the size of the parsing table to be proportional with the number of terminals and non-terminals (operands) used in the grammar. Since the grammar of the OMeta parser contains 446 operands, it is believed that performance is affected negatively. The believe is strengthen by messages from members of the OMeta community which states that performance was never OMeta's top priority:

> "OMeta is not suited to high performance parsing (at least not the current unoptimized implementations). The beauty of OMeta is that it's so flexible, elegant and easy to implement ... but at the cost of performance."
>
> - John Leuner

In the vanilla implementation it's noticeable that most time is spend on lexing and not on parsing. The lexical phase is therefore the bottleneck in the vanilla implementation. A first attempt to improve the performance of the code was made during the project but resulted in an even worse performance. One optimization that was not carried out includes the use of stream readers to read characters from the input. The current implementation reads in the complete Waebric program using a Java library and afterwards passes it as string object to the lexer. It might be more efficiency to use input streams for this kind of operations so the complete string should not be kept in memory, but this would contradict with the requirements from section 3 which stated that the dependencies and functionality of external libraries should be kept to the bare minimum.

Despite the above observations is the vanilla implementation the most efficient of both implementations.

7.3 Functionality

The acceptance tests performed in section 6.3 demonstrated that the complete syntax notation is supported by both implementation. This conclusion is in variance with the hypothesis stated in section 3.4 which presumed that OMeta wouldn't be capable of supporting the complete syntax notation of Waebric. The hypothesis was based on the notion that BNF includes a flaw by which certain symbol characters in the DSL may conflict with BNF notation itself. During implementation is became clear that OMeta does not derives these limitation and, as a result, OMeta was able to achieve 100% acceptance.

The second level at which functionality was measured includes the level of error reporting. The vanilla implementation provided in 92% of the test cases the correct error message in terms of the problem domain with corresponding line/column position and parser stack. OMeta on the other hand was only capable of providing the end-user a message "Match failed!". The level of error reporting was forced up by manually by counting the newline characters in the semantic action code in order to output at least the line number at which parsing fails. The remaining functionality to provide a parser stack or to point out the production rule at which matching failed is however not supported by OMeta.

Chapter 8

Conclusions

This chapter provides concluding remarks on the research questions stated in section 1.2 based on the analysis presented in chapter 7.

8.1 Maintainability

RQ1.1	How maintainable is each implementation in terms of lines of code, method size, McCabe complexity, Halstead Effort and Maintainability Index?

The evaluation of the parser and typechecker indicates that the OMeta implementation provides better maintainable code then its opponents in the vanilla implementation. The large differences at the parser module suggest that OMeta provides the best solution to develop a highly maintainable parser with a minimum of resources. The results for the typechecker are less significant since most metrics are identical apart from the (e)LOC metric. However, when the reuse of the tree walker is taken into account, then the OMeta typechecker might be a better choice.

As a result, the assumption is made that the OMeta implementation is more maintainable then the vanilla implementation which confirms the hypothesis in section 3.4 which stated that maintenance is reduced when using OMeta instead of traditional interpreter techniques.

8.2 Efficiency

RQ1.2 How efficient is each implementation in terms of duration of regression tests?

The efficiency of the implementations was measuring using 8 regression tests with the objective to translate the Waebric program as fast as possible. Results clearly indicated that the vanilla implementation presents better performance results compared to OMeta/JS. This observation is in line with the hypothesis made in section 3.4. Despite better performance results for the traditional interpreter technique, performance is still remarkable poor due to JavaScript's implementation and the question might arise how useful a slow implementation is in a production environment.

8.3 Functionality

RQ1.3 How functional is each implementation in terms of acceptance tests, supported notation and error reporting?

Functionality was determined by measuring the level of acceptance and error reporting. The acceptance tests indicated no differences between the implementations, and as a result, functionality only depends on the level of error reporting.

Error reporting is significant better when using the traditional interpreter technique which provides error messages written in the language of the end-user and in terms of the problem domain. This is in shrill contrast with the OMeta implementation which is only capable of providing a generic error message "Match failed!" with corresponding line number. This makes OMeta probably not the best choice for a production environment as end-users are left in the dark. This viewpoint is supported in a technical report which states that the main purpose of OMeta is to *prototype* new programming languages or extensions to existing languages [35].

As a result of previous observations, it is concluded that the interpreter technique results in a better level of functionality and OMeta should only be chosen when error reporting is not of great importance (e.g. prototyping). This observation is only partially supported by the hypothesis in section 3.4 since it was presumed that the level of supported syntax notation in OMeta would be lower.

8.4 Overall quality

RQ1	How does the quality of a DSL implementation in OMeta/JS relates to the quality of a hand-written JavaScript interpreter?

Choosing a suitable DSL implementation technique is essential when developing a domain-specific language as it can affect the total effort required to implement the DSL largely, not to mention end-user productivity, efficiency and maintainability. The research presented in this thesis implemented the Waebric markup language using two commonly used techniques for domain-specific languages, the interpreter and compiler generator technique (OMeta/JS). The results gathered from these implementations exhibits large quality differences and show that there is no such thing as the most qualitative DSL implementation technique since each approach has its pros and cons.

The OMeta implementation has provided the best results in the field of maintainability as a result of minimal implementation effort and the highly maintainable code it yields. This comes with two downsides however, poor efficiency results - up to a factor four in difference - and the lack of good error reporting towards the end-user. This makes OMeta not useful for a production environment and OMeta should therefore be seen as tool for prototyping new programming languages or extensions to existing languages.

The results for the interpreter technique show that this implementation technique is more versatile compared to OMeta since it yields better performance and functionality while maintainability is still acceptable. If the cut-off values of the Maintainability Index and cyclomatic complexity are considered, then the vanilla implementation still has highly maintainable code. The main disadvantage of the interpreter technique is the time and effort required to implement a DSL. The Halstead metrics indicates that the interpreter technique requires twice as much effort compared to the implementation in OMeta.

Based on previous observations it is recommended to use OMeta/JS for rapidly prototyping domain-specific languages without the need for high performance and good error reporting, while the interpreter technique is best applicable for DSLs targeted to production environments or when high performance is desired.

Bibliography

[1] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, & Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Alves, T., and Visser, J. Metrication of sdf grammars. Tech. rep., Departamento de Informtica da Universidade Do Minho, Campus De Gualtar, Braga, Portugal, 2002.

[3] Bentley, J. L. Programming pearls: Little languages. *Communications of the ACM 29*, 8 (August 1986), 711–721.

[4] Brauer, W. On grammatical complexity of context-free languages. In *Proceedings of Mathematical Foundations of Computer Science* (1973), pp. 193–196.

[5] Csuhaj-Varj, E., and Kelemenov, A. Descriptional complexity of context-free grammar forms. *Theoretical Computer Science 112*, 2 (1993), 277–289.

[6] Feldman, J. A., Gips, J., Horning, J. J., and Reder, S. Grammatical complexity and inference. Technical report, Computer Science Departement Standford University, June 1969.

[7] Fenton, N. E., and Neil, M. Software metrics: roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering* (New York, NY, USA, 2000), ACM, pp. 357–370.

[8] Ford, B. Parsing expression grammars: a recognition-based syntactic foundation. *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2004), 111–122.

[9] Goodman, P. *Software Metrics, Best Practices for Successful IT Management*. Rothstein Associates Inc, 2004.

[10] Halstead, M. H. *Elements of Software Science*, vol. 7. Elsevier, 1977.

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

[12] Hudak, P. Building domain specific embedded languages. *ACM Computing Surveys 28A*, 4es (December 1996), 196.

[13] IBM, S. A. Rational asset analyzer. `http://publib.boulder.ibm.com/infocenter/rassan/v5r5/index.jsp?topic=/com.ibm.raa.doc/common/chalstd.htm`.

[14] Jones, J. Abstract syntax tree implementation idioms. *Pattern Languages of Program Design* (2003). Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003) http://hillside.net/plop/plop2003/papers.html.

[15] Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., LEwis, J., Oliva, D. P., Sheard, T., Smith, I., and Walton, L. A software engineering experiment in software component generation. *Proceedings of the 18th International converence on Software Engineering 18* (1996), 542–553.

[16] Kosar, T., E. Martnez Lpez, P., A, Barrientos, P., and Mernik, M. A preliiminary study on various implementation approaches of domain-spefic language. *ScienceDirect - Information and Software Technology 50* (April 2007), 390–405.

[17] Levitin, A. V. How to measure software size, and how not to. *Proceedings COMPSAC* (October 1986), 314–318.

[18] Liso, A. Software maintainability metrics model: An improvement in the coleman-oman model. *Crosstalk* (2001), 15–17.

[19] Lowther, B. The application of software maintainability metric models on industrial software systems,. Master's thesis, Department of Computer Science, University of Idaho, Moscow, 1993.

[20] McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 4 (December 1976), 308–320.

[21] Mernik, M., Heering, J., and M. Sloane, A. When and how to develop domain-specific languages. *ACM Computing Surveys 37* (2005), 316–344.

[22] Microsystems, S. Code conventions for the javatm programming language. `http://java.sun.com/docs/codeconv/html/CodeConventions.doc5.html`, September 1997.

[23] Myers, G. J. An extension to the cyclomatic measure of program complexity. *ACM SIGPLAN Notices 12*, 10 (October 1977), 61–64.

[24] Oman, P., Coleman, D., Ash, D., and Lowther, B. Using metrics to evaluate software system maintainability. *The Book Paradigm for Improved Software Maintenance 7*, 1 (January 1990), 39–45.

[25] Oman, P., and Hagemeister, J. Metrics for assessing a software system's maintainability. *Conference on Software Maintenance* (November 1992), 337–344.

[26] Oman, P., and Hagemeister, J. "constructing and testing of polynomials predicting software maintainability. *Journal of Systems and Software 24*, 3 (March 1994), 251–266.

[27] Power, J., and Malloy, B. Metric-based analysis of context-free grammars. In *Proceedings 8th International Workshop on Program Comprehension* (2000), pp. 171–178.

[28] Power, J. F., and Malloy, B. A. A metrics suite for grammar-based software: Research articles. *J. Softw. Maint. Evol. 16*, 6 (2004), 405–426.

[29] Qutaish, R. E. A., and Abran, A. An analysis of the design and definitions of halsteads metrics. *In Proceedings of the 15th International Workshop on Software Measurement (IWSM'2005) 15* (September 2005), 337–352.

[30] Rosenberg, J. Some misconceptions about lines of code. *Software Metrics, IEEE International Symposium on 0* (1997), 137.

[31] Spinellis, D. Notable design patterns for domain specific languages. *Journal of Systems and Software 56*, 1 (Feb. 2001), 91–99.

[32] Van Der Storm, T. *WAEBRIC: a Little Language for Markup Generation*, June 2009.

[33] van Deursen, A., and Klint, P. Tittle languages: Little maintenance? *Journal of Software Maintenance: Research and Practice 10*, 2 (December 1998), 75–92.

[34] Van Deursen, A., Klint, P., and Visser, J. Domain-specific languages: An annotated bibliography. *ACM 35(6)* (2000), 26–36.

[35] Warth, A. Experimenting with programming languages. Tech. rep., Viewpoints Research Institute, 2008.

[36] Warth, A., and Piumarta, I. Ometa: an object-oriented language for pattern matching. *DLS'07 DSL* (2007).

[37] Welker, K., Oman, P., and Atkinson, G. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice 9*, 3 (May 1997), 127–159.

[38] Wile, D. S. Supporting the dsl spectrum. *Journal of Computing and Information Technology 9*, 4 (2001), 263287.

[39] Zenger, M., and Odersky, M. Implementing extensible compilers. *Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL)* (2001).

Appendix A

Metrics

DSL BENCHMARK IMPLEMENTATION PROJECT

Maintainability

	LOC	ELOC	METHOD COUNT	AVG METHOD SIZE	Production Rules	Average size RHS	Total MCC	Avg MCC	Halstead Volume	Avg Halstead Volume	Halstead Effort	MI3	MI4
VANILLA IMPLEMENATION													
PARSER	**2.214**	**1.828**	**226**	*9,59*	*N/A*	*N/A*	**736**	*2,93*	*15607,35*	*187,01*	**379.862,69**	*78,54*	*119,35*
Lexical	538	431	63	8,54	*N/A*	*N/A*	191	3,03	8.333,20	132,27	21.901,17	83,53	124,86
Syntactic	1.557	1.294	151	10,31	*N/A*	*N/A*	517	3,42	36.222,25	239,88	336.060,35	74,57	113,78
Other	119	103	12	9,92	*N/A*	*N/A*	28	2,33	2.266,61	188,88	21.901,17	77,52	119,41
VALIDATOR	**606**	**473**	**90**	*6,73*	*N/A*	*N/A*	**169**	*1,88*	*9220,95*	*N/A*	**73.821,84**	*91,27*	*131,47*
Validator	606	473	90	6,73	*N/A*	*N/A*	169	1,88	9.220,95	102,46	73.821,84	91,27	131,47
INTERPRETER	**1.018**	**792**	**124**	*8,21*	*N/A*	*N/A*	**318**	*2,56*	*19969,98*	*161,05*	**200.908,95**	*83,08*	*123,51*
Interpreter	1.018	792	124	8,21	*N/A*	*N/A*	318	2,56	19.969,98	161,05	200.908,95	83,08	123,51
AST	**609**	**447**	**131**	*4,65*	*N/A*	*N/A*	**250**	*1,91*	*7620,87*	*58,17*	**46.005,28**	*104,16*	*147,94*
Abstract Syntax Tree	609	447	131	4,65	*N/A*	*N/A*	250	1,91	7.620,87	58,17	46.005,28	104,16	147,94
TOTAL (SUM)	**4.447**	**3.540**	**440**	*N/A*	*N/A*	*N/A*	**1.223**	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
TOTAL (AVG PER SUBMODULE	**741**	**590**	**95**	**8,06**	*N/A*	*N/A*	**246**	**2,52**	**13.938,98**	**147,12**	**116.766,46**	**85,69**	**126,83**
OMETA IMPLEMENATION													
PARSER	**361**	**304**	**153**	*22,99*	**60**	**6,43**	**225,00**	*7,24*	*N/A*	*N/A*	**152.592,01**	*99,99*	*132,14*
Grammar (excl. action code)	130	105	60	6,43	60	6,43	108,00	1,80	6.486,90	108,12	94.342,16	91,95	115,77
Grammar internal action code	98	92	76	1,29	*N/A*	*N/A*	80,00	1,05	5.117,73	67,34	42.140,98	133,23	153,52
Grammar external action code	87	71	10	8,70	*N/A*	*N/A*	21,00	2,10	1.869,02	186,90	13.527,86	80,72	120,64
Other	46	36	7	6,57	*N/A*	*N/A*	16,00	2,29	526,62	75,23	2.581,01	94,06	138,64
VALIDATOR	**330**	**262**	**49**	*6,13*	*10,00*	*2,00*	**76**	*1,48*	*1190,67*	*89,90*	**49.550,31**	*99,36*	*137,76*
Grammar (excl. action code)	13	11	10	2,00	10	2,00	0,00	0,00	228,44	22,84	934,53	131,33	172,67
Grammar internal action code	61	35	10	6,10	*N/A*	*N/A*	17	1,70	1.270,65	127,07	26.240,42	92,00	127,24
Grammar external action code	197	170	20	9,85	*N/A*	*N/A*	38	1,90	2.502,78	125,14	18.263,96	80,87	112,71
Other	59	46	9	6,56	*N/A*	*N/A*	21	2,33	760,80	84,53	4.111,40	93,23	138,44
INTERPRETER	**1.018**	**792**	**124**	*8,21*	*N/A*	*N/A*	**318**	*2,56*	*N/A*	*N/A*	**200.908,95**	*83,08*	*123,51*
Interpreter	1.018	792	124	8,21	*N/A*	*N/A*	318	2,56	19.969,98	161,05	200.908,95	83,08	123,51
AST	**609**	**447**	**131**	*4,65*	*N/A*	*N/A*	**250,00**	*1,91*	*N/A*	*N/A*	**46.005,28**	*104,16*	*147,94*
Abstract Syntax Tree	609	447	131	4,65	*N/A*	*N/A*	250	1,91	7.620,87	58,17	46.005,28	104,16	147,94
TOTAL (SUM)	**2.318**	**1.805**	**457**	*N/A*	**70**	**8,43**	**869**	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*	*N/A*
TOTAL (AVG PER SUBMODULE	**231,80**	**180,50**	**45,70**	**6,04**	**35,00**	**4,22**	**86,90**	**1,76**	**4.635,38**	**101,64**	**44.905,66**	**98,46**	**135,11**

DSL BENCHMARK IMPLEMENTATION PROJECT

Efficiency

	MODULE TOKENIZER	MODULE PARSER	MODULE TYPECHECKER	MODULE INTERPRETER	TOTAL
VANILLA LAVA.WAE					
Average	2.708,02	657,41	155,36	881,83	4.437,07
VANILLA LDTA.WAE					
Average	878,22	334,00	68,72	593,14	1.899,69
VANILLA MENUS.WAE					
Average	154,12	200,31	34,19	424,69	843,58
VANILLA 1k					
Average	882,53	526,24	85,04	459,52	1.957,85
VANILLA 2.5k					
Average	2.017,54	1.039,82	99,38	533,10	3.693,88
VANILLA 5k					
Average	4.110,02	1.676,88	150,30	778,31	6.719,09
VANILLA 7.5k					
Average	6.440,87	2.530,80	200,97	945,53	10.121,92
VANILLA 10k					
Average	9.455,14	3.106,28	254,81	1.314,40	14.134,99
OMETA LAVA.WAE					
Average	19.514,85		2.561,21	802,71	22.878,77
OMETA LDTA.WAE					
Average	6.118,20		792,35	543,61	7.454,16
OMETA MENUS.WAE					
Average	812,65		65,93	445,23	1.323,81
OMETA 1k					
Average	6.318,77		825,38	441,75	7.585,90
OMETA 2.5k					
Average	13.684,09		1.444,88	489,72	15.618,69
OMETA 5k					
OMETA 5k	24.646,12		2.568,89	713,56	27.928,57
OMETA 7.5k					
Average	35.881,59		3.822,23	1.085,94	40.789,76
OMETA 10k					
Average	48.079,44		4.292,37	1.514,26	53.886,07

DSL BENCHMARK IMPLEMENTATION PROJECT

Results error messages

	OMETA IMPLEMENTATION				VANILLA IMPLEMENTATION			
	Correctness error	**In terms of domain**	**Position (line/column)**	**Parser stack**	**Correctness error**	**In terms of domain**	**Position (line/column)**	**Parser stack**
No semicolon defined after statement ";"	✗	✗	!	✗	✓	✓	✓	✓
Unclosed formal in function definition ")"	✗	✗	!	✗	✓	✓	✓	✓
Explicit semicolon after last site mapping ";"	✗	✗	!	✗	✓	✓	✓	✓
Using a keyword as markup	✗	✗	!	✗	✓	✓	✓	✓
No separator site path/markuo	✗	✗	!	✗	✓	✓	✓	✓
Illegal character in SymbolText ">"	✗	✗	!	✗	✗	✗	✗	✗
Wrong type predicate (input.test?)	✗	✗	!	✗	✓	✓	✓	✓
No ending quote in a text expression	✗	✗	!	✗	✓	✓	✓	✓
No ending semicolon after the let-statement	✗	✗	!	✗	✓	✓	✓	✓
Unclosed function definition "end"	✗	✗	!	✗	✓	✓	✓	✓
Illegal character in text expression "<"	✗	✗	!	✗	✓	✓	✓	✓
Illegal character in formals "<"	✗	✗	!	✗	✓	✓	✓	✓
Unclosed embedding ">"	✗	✗	!	✗	!	!	✓	✗
Defining an each statement with a typo	✗	✗	!	✗	✓	✓	✓	✓
Wrong separator variable / list in each statement	✗	✗	!	✗	✓	✓	✓	✓