

OOPEG: An Object-Oriented Parser Generator Based on Parsing Expression Grammars

A Master's Thesis by
Jacob Korsgaard and Jørgen Ulrik B. Krag
d626a

Department of Computer Science
Aalborg University

July 30, 2010

Title:

OOPEG: An Object-Oriented Parser Generator Based on Parsing Expression Grammars

Abstract:

This thesis documents the development of OOPEG, a Parsing Expression Grammar (PEG) based parser generator written in C#. We find that other compiler-compilers have dense input syntax, because of the inclusion of semantic actions, labeling and disambiguation markup.

We have found that PEGs are very suitable for building parser generators as our grammars not only define the syntax of the language concisely, but also explicitly define the parsing procedure and by using very little markup we can generate concise, safe and easily usable object-oriented abstract syntax trees.

OOPEG features an object-oriented modular parsing engine that uses packrat parsing to achieve sufficient performance to be used in a production environment. To make OOPEG more accessible we provide a tutorial for creating a simple language, as well as integration with the Visual Studio 2010 IDE.

Project theme:

Programming Technology

Project period:

Spring Semester 2010,
1st February 2010 to 30th July 2010

Project group:

d626a

Participants:

Jacob Korsgaard
Jørgen Ulrik Balslev Krag

Supervisor:

Bent Thomsen

Print run:

4

Pages:

96

Appendices (number, type):

1 CD-ROM with source and binaries

The contents of the thesis are freely available, however publishing (with source) must happen only by agreement with the authors.

This thesis was developed by project group d626a at the Aalborg University, Department of Computer Science, in the spring of 2010.

When external sources are used, the reference for the source is written in square brackets e.g. [1], where 1 corresponds to an entry in the bibliography, which is located at the end of the thesis.

Bold and *italic* are used to emphasize words, no specific convention is applied.

When citations are used it is explicitly noted as a citation.

A CD-ROM is attached to this thesis which contains the source code of the implemented software.

We would like to extend our thanks to our supervisor Bent Thomsen at Aalborg University. We would like thank Progressive Media for their corporation and feedback.

Jacob Korsgaard

Jørgen Ulrik Balslev Krag

1. Introduction	11
1.1. Modern Compilers	12
1.2. Modern Parser Generators	12
1.3. Object-Oriented Abstract Syntax Trees	13
1.4. The Visitor Pattern	14
1.5. Background and Problem Statement	15
1.6. Thesis Organization	16
2. Parsing Expression Grammars and packrat parsing	17
2.1. PEG Advantages	18
2.2. PEG Disadvantages	19
2.3. Packrat Parsers	20
2.3.1. Left Recursion	21
3. Compiler-Compilers	23
3.1. Lex / Yacc	23
3.2. ANTLR	26
3.3. SableCC	28
3.4. Xtext	32
3.5. Rats!	34
3.6. Observations	35
3.6.1. Grammar Syntax	35
3.6.2. Semantic Actions and PEGs	35
3.6.3. Improving the Learning Curve	36
4. The Object-Oriented Parsing Expression Grammar (OOPEG) Parser Generator	37
4.1. Parsing Engine	37
4.2. Parser Generator	38
5. OOPEG Tutorial	39
5.1. Input Grammar	40
5.1.1. Choosing Rule Types	42
5.2. Invoking the Parser	42
5.3. Implementing Semantics	43
5.4. Putting It All Together	46
5.5. Advanced Usage	47
5.5.1. Using Multiple Grammars	48
5.5.2. Command-line Invocation of OOPEG	48

6. Syntax Tree Construction	49
6.1. Concrete Syntax Tree	49
6.2. Abstract Syntax Tree	51
6.2.1. Choice Index	52
6.3. Reduced Abstract Syntax Tree	53
6.3.1. Pass-Through Rules	54
6.3.2. Tree-Types	55
6.4. Syntax Tree Representation	55
6.4.1. Node Representation	56
6.4.2. Representing Parent-Child Relations	56
6.4.3. Representing Tree-Types	58
6.5. Decorating the AST	59
6.6. Error Handling	60
7. Parsing Engine	61
7.1. Parsing Expressions and Operators	61
7.2. Rules and Non-terminals	63
7.2.1. Creating AST Nodes	63
7.2.2. Non-terminals	65
7.2.3. Parsing a Rule	67
7.2.4. Memoization	67
7.3. Sub-trees using Tree-Types	68
7.4. Left-Recursive Rules	69
8. Parser Generator	71
8.1. Grammar Specification Language	71
8.1.1. Additional Operators	71
8.1.2. Naming Scope Rules	72
8.1.3. Defining Tree-Types	73
8.1.4. Defining Rules	73
8.2. Syntactic Analysis	74
8.3. Contextual Analysis	74
8.3.1. Declaration Pass	74
8.3.2. Name Resolving	75
8.3.3. Left Recursion	75
8.3.4. Inferring Typed Child Relations	75
8.4. Code Generation	76
8.4.1. Tree-Types and Visitors	76
8.4.2. AST Classes	77
8.5. Future Improvements	78
8.5.1. Prefix Capture Check	78
8.5.2. C# Using Directive	78

8.5.3. Assembly References	78
8.5.4. Errors by Non-terminals	79
8.5.5. Extended Modularity	79
8.5.6. More Visual Studio Integration	80
8.5.7. Grammar Optimization Tools	80
9. Optimizing Performance	81
9.1. Instantiating Type-Safe Nodes	81
9.2. Reducing Instantiation Count	82
9.3. Grammar Optimization	83
9.4. Commit Operator	84
10. Discussion	85
10.1 Multiple Target Languages	85
10.2 Inline Semantic Actions	85
10.3 Future Research	86
10.3.1 Left Recursion Elimination	86
10.3.2 Optimized Selection of Choice Alternatives	86
10.3.3 Error Recovery	87
10.3.4 Optimizing Grammars For Better Cache Usage	87
10.3.5 Automatic Commit Operator	88
11. Conclusion	89
11.1 Building ASTs	89
11.2 Parsing Engine	90
11.3 Learning Curve and Integration	91
A. The OOPEG Grammar for OOPEG	95

Introduction

In this thesis we examine the possibilities and difficulties presented by basing an object-oriented parser generator tool on *Parsing Expression Grammars* (PEGs)[7]. The goal for the parser generator is to be used in a production setting at the video game developer Progressive Media, but the tool should be general enough to appeal to other compiler developers and enthusiasts.

A parser generator is a tool that generates code for parsing a programming language using a grammar specification of that language. Such a grammar is usually a *Context-Free Grammar* (CFG) and written in languages resembling *Backus-Naur Form* (BNF) or *Extended BNF* (EBNF). In this thesis we replace these grammars with PEGs and investigate the implications.

On the surface CFGs and PEGs seem very similar as they both can be described by a set of rules of the form:

$$\textit{Symbol} \leftarrow \textit{Expression}$$

where *Symbol* is a non-terminal symbol and *Expression* is a parsing expression. Parsing expressions are built using *terminals*, *non-terminals* and *operators*. Non-terminals refer to other rules; terminals are symbols from the input language; and operators represent the operations we can perform in an expression, such as sequencing, choosing and repeating.

The main difference between the two grammars is that - contrary to PEGs - CFGs can be ambiguous. This is because CFGs are used to **generate** languages where PEGs **recognize** languages. Source code parsers are used to recognize programs and cannot be ambiguous, which means that even though we can define an ambiguous grammar using BNF or EBNF we cannot generate a parser for such grammars and we would not want to. Such grammar specifications need to be annotated with disambiguation information or have convoluted rules to resolve disambiguation, which is for example the case for the common solution to the dangling else problem.

Another difference between PEGs and CFGs is the languages they can recognize. A language such as $\{a^n b^n c^n \mid n \geq 1\}$ cannot be recognized by a CFG, but is recognizable by a PEG because of its syntactic predicates that provide unlimited lookahead, see chapter 2 on page 17.

1.1. Modern Compilers

Parsing is part of the first phase of a compiler, which is a tool that converts code written in a source language into code written in the target language. Legacy compilers do this by parsing a source code file, breaking it down into the syntactical constructs of the parsed language and performing a single semantic action whenever each construct is encountered in the input. The problem with this approach is that it only allows a single pass over the source code and that actions can only be performed in the order the constructs of the language are matched. A language like C uses constructs like forward declarations to make it possible to parse the language in a single pass, but languages like C# and Java requires multiple passes over the source code.

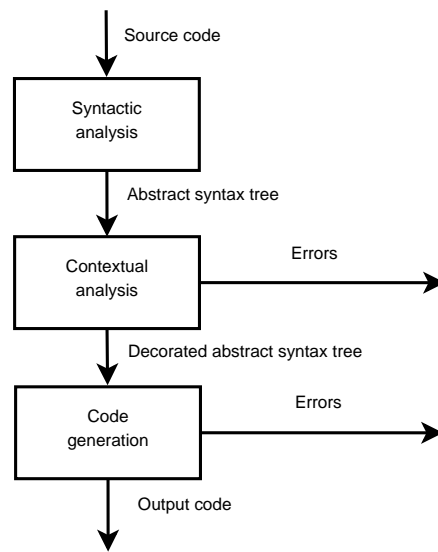


Figure 1.1.: Data flow in a typical modern compiler.

A modern compiler usually parses the input code and generates an internal representation of the code called an Abstract Syntax Tree (AST). This is done in the **syntactic analysis** phase of the compiler, see Figure 1.1. The AST is passed over many times during the **contextual analysis** while various semantic checks are performed, such as enforcing scope rules and type checking. During the contextual analysis the AST is decorated and manipulated before it is used in **code generation**. Compilation of C# source code requires many passes on the ASTs as explained by a senior developer from the C# compiler team, Eric Lippert, on his blog[17].

1.2. Modern Parser Generators

A modern parser generator should generate the code needed to do the *syntactic analysis*. We can choose two different ways of building ASTs: one is to have semantic actions for every parsing rule in the grammar and have the language designer add code to build

his own ASTs in the grammar, this can however litter the input grammar as shown in chapter 3 on page 23; the other alternative is to automatically build an AST based on the *parse tree*.

The parse tree represents the syntactic structure of the parsed input, the leaves of the tree are terminals and the interior nodes are non-terminals. A parse tree is also called a **Concrete Syntax Tree** (CST). It can be inconvenient working with CSTs as they can potentially contain many nodes that have no semantic value. So any parser that automatically builds CSTs should have a utility to reduce such trees to ASTs. Such functionality is described in chapter 6 on page 49.

1.3. Object-Oriented Abstract Syntax Trees

If a generated parser builds ASTs automatically then it becomes the responsibility of the parser generator to define the representation of ASTs and how they are traversed, decorated and manipulated by the compiler developer.

A common way of representing an AST is by using object-oriented programming. This is because, if the AST is built correctly it can be very concise. We can for example represent arithmetic expressions in an AST using the object-oriented *Composite* design pattern[10], where the leaf nodes are literals and interior nodes are operators. All nodes in the AST are of the type `Expression` and binary operators have references to the left and right operand both of which are also `Expression` types. This recursive design can be used to build arbitrarily complex expressions using few types. For example, we can represent "1+2*3" including operator precedence in an object hierarchy as seen on Figure 1.2.

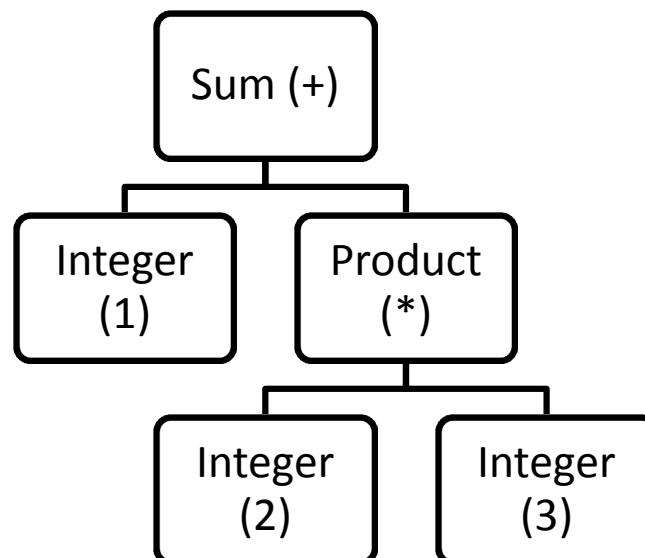


Figure 1.2.: An example AST for $1+2*3$, each box is an object and the lines are references.

All `Expression` objects have at least a single common operation which is *evaluate*, which we can use to calculate the result of the expression. The *evaluate* operation has separate behavior for every type of node in the tree, an `Integer` returns its integer value, a `Sum` returns the sum of the value of its two operands and `Product` returns the product of its operand values.

This means that given any reference to an `Expression` we want to be able to perform the *evaluate* operation. In object-oriented programming we have basically two choices in where to put the operation[23]:

Data-centered We can put the operation on the data by implementing an abstract or virtual function on `Expression` and all sub-types.

Operation-centered Alternatively, we can put the operations in a separate class which has the specific functionality for every sub-type of `Expression`. This approach is called the **Visitor** pattern and is preferable as it the clean separation of data and operations performed on the data.

1.4. The Visitor Pattern

It is problematic for us to use the data-centered approach when the classes for the ASTs are generated by a parser generator. Because the classes for the AST are auto-generated any changes made by the language designer will be overwritten if he makes changes to the original grammar and re-generate the AST classes. Furthermore, adding a method in all classes in a type hierarchy for every task you want to perform clutters the classes.

The Visitor pattern is designed to relieve these problems. A visitor is an interface that defines a `Visit` method for all the exact types in a type hierarchy. The base class in the type hierarchy defines an abstract `Accept` method that all sub-types have to implement. In each of the implementations there is a single call to the `Visit` method on the *visitor* that takes the exact type of the sub-type.

Given an object reference by the base class type, we can call the `Accept` method with a visitor and have the `Visit` method for the exact type of the object be called on the visitor.

Visitors are often used to do tree traversal, a traversing visitor executes some logic for a node and then calls `Accept` on the children of the node. If all nodes have a single list of children of the base type, a common pattern is to implement an abstract traversal adapter class. All `Visit` methods on the traversal class simply call `accept` on the children of the currently visited node. The traversal class can then be derived from and the methods for the nodes the new class is interested in can be overridden, for all other types of nodes the adapter will simply continue traversing the tree. This results in very concise tree traversal classes that only have logic for the parts of the tree we are interested in.

1.5. Background and Problem Statement

This project is a continuation of our previous project [15], in which we were studying possibilities for using programming technology to improve the process of developing cross-platform video games. In that project we were examining how we could ease having a single source code base and generate code for multiple different target devices. During that project we identified three major areas of further interest, one of which was the further development of an object-oriented PEG parser we had implemented. We recognized the potential of using PEGs as the core in a compiler-compiler tool and chose to follow up on that subject.

In the previous project and this project we are working with the video game developer Progressive Media. At Progressive Media they have many small languages that are used in unison to generate a multitude of target languages. Small languages are used for doing preprocessing, for defining assets, for scripting by designers and more. Having a tool that eases the design, implementation and maintenance of languages could be of great value to the company. As with any production setting, a core requirement for any tool is that it performs satisfactory in terms of both usability and execution time. Optimizing both ease of use and performance is therefore essential.

We implement the tool in the C# language because it is the object-oriented language of choice for Progressive Media and ourselves. Also, once completed we would like Progressive Media to use and help maintain the parser generator. The tool would also benefit greatly if it integrates well with the tool chain used at Progressive Media which is based on Visual Studio and the MSBuild build tool.

Finally, while we are targeting Progressive Media as the users of our tool, we would like it to be easily accessible and usable by any programming language developer or enthusiast.

To summarize, the purpose of our thesis is:

- We intend to examine existing compiler-compiler tools and identify desirable properties applicable to a PEG based parser generator.
- We intend to design and build an object-oriented PEG based parser generator having said properties, as well as identify the techniques needed to do so.
- We want the tool to perform well enough for it to be viable in a production environment such as at Progressive Media.
- Finally, we want to identify areas of interest for future development of the tool and PEG based compiler-compilers in general.

1.6. Thesis Organization

The rest of this thesis is organized as follows:

Chapter 2 provides an introduction to parsing expressions grammars and packrat parsing.

Chapter 3 examines which lessons and design ideas can be learned from existing compiler-compilers.

Chapter 4 outlines the core visions and requirements for the OOPEG parser.

Chapter 5 contains a tutorial on how to use OOPEG to generate an interpreter for a small language.

Chapter 6 details how syntax trees are constructed in OOPEG.

Chapter 7 details the implementation of the OOPEG parsing engine.

Chapter 8 describes the implementation of the parser generator in OOPEG as well as possible future improvements for OOPEG.

Chapter 9 describes techniques used to improve OOPEG's performance.

Chapter 10 contains a discussion of possible future work.

Chapter 11 contains our conclusion.

Parsing Expression Grammars and packrat parsing

The following description of Parsing Expression Grammars is based on the one in [15] with corrections and additions.

A Parsing Expression Grammar (PEG) is a grammar for describing formal languages. A PEG is basically a schematic form of a recursive descent parser. Which means the procedure for parsing input is clearly readable from the definition of said language in PEG. A PEG consists of a set of rules similar to Backus-Naur Form (BNF) or Extended Backus-Naur Form (EBNF) on the form:

Symbol \leftarrow *Expression*

Each rule is an instruction for parsing a string and expressions can be constructed from operators defined in table 2.1 on the next page. Other rules can be referenced by using the symbol of the rule as a non-terminal. If a rule matches the input it will indicate success and consume the matched input (unless the entire rule is a predicate). Otherwise it will indicate failure.

A PEG can never be ambiguous because the PEG choice operator is ordered. A parser created from a PEG will attempt each alternative in a choice in order. If an alternative successfully parses the remaining choices will not be tried. If a parsing error occurs while parsing an alternative, the parser will backtrack and attempt the next alternative in the choice.

It is possible for a PEG to indicate success when parsing a string without consuming all the input. If you need to verify that a PEG recognizes the whole string you will have to check that the top-level rule from the PEG indicates success when applied to the string and that all the input from the string is consumed. An alternative solution is to add a predicate rule to the grammar that states that no further input must exist. This can be done by appending the expression '! .' to the starting rule of the grammar. In this case the rule will fail if there is input remaining.

Listing 2.1 shows an example PEG that recognizes arithmetic expressions, by parsing the input starting from the Start rule.

Listing 2.1: An example PEG for recognizing arithmetic expressions.

```
Start  <- Sum;
Sum    <- Product ( ( '+' / '-' ) Product )*;
Product <- Value ( ( '*' / '/' ) Value )*;
Value  <- [0-9]+ / '(' Sum ')';
```

A / B / C / D	Ordered Choice: Try each of the rules A, B, C and D in order. If one of the rules matches, consume what the rule matched and indicate success. If none of the rules matches indicate failure.
A B C D	Sequence: Apply each rule in order, consuming input for each. If one rule fails to match reset input position back to before starting parsing of the sequence and indicate failure. Otherwise indicate success.
&E	And Predicate: Match against rule E, but do not consume any input. Indicate success if the rule matched and failure otherwise.
!E	Not Predicate: Match against rule E, but do not consume any input. Indicate failure if the rule matched and success otherwise.
E+	One or More: Match E as many times as possible, consuming input each time a match is made. Indicate success if one or more matches were made, otherwise indicate failure.
E*	Zero or More: Match E as many times as possible. Always indicate success.
E?	Zero or One: Try to match E and consume input if match is made. Indicate success.
[abcd]	Character Class: Matches any of the characters 'a', 'b', 'c' or 'd'. If match is made consume input and indicate success. Otherwise indicate failure.
[c1-c2]	Character Range: Matches any character in the range between c_1 and c_2 . If match is made consume input and indicate success. Otherwise indicate failure.
'Some text'	Literal String: Match the literal string given. If match is made consume input and indicate success. Otherwise indicate failure.
.	Any Character: Match any single character. Will always consume input and indicate success, unless at the end of input where it will indicate failure.

Table 2.1.: Table of PEG operators.

2.1. PEG Advantages

The major advantage of PEGs over CFGs is that they are never ambiguous. For example most programming languages have `if-then` and `if-then-else` constructs. If these constructs are defined naturally in a context-free grammar the string "`if a then if b then s1 else s2`" can be interpreted in two different ways. The ordered choice of a PEG removes this ambiguity by always accepting the first possible match for the input.

Another advantage of PEGs over CFGs is that PEGs provide unlimited look-ahead by using predicates. An example of a language that can not be expressed using a context-free grammar, but can be expressed by the usage of PEG predicates, is $\{a^n b^n c^n | n \geq 1\}$ as described in [7]. A PEG for recognizing this language is shown in listing 2.2.

Listing 2.2: A PEG for recognizing $\{a^n b^n c^n | n \geq 1\}$.

```
Start <- &(AB 'c') 'a'+ BC;
AB <- 'a' AB? 'b';
BC <- 'b' BC? 'c';
```

This means that PEGs are able to express some languages that CFGs can not. However precisely what PEGs can express is still an open question[7].

Another example of a language that can be expressed with PEGs and not with CFGs is the `block-skip` language mentioned in [1], which handles indentation rules similar to those of the `Python` programming language directly in the grammar.

Another advantage of PEGs are that they are easy to extend and modify. The composable nature of the rules and operators in a PEG makes it possible to merge two PEGs into a new one if there are no clash between the symbol names used on the two PEGs. This is used extensively by the `Fortress` language, which is based on the *Rats!* [11] parser. The idea of giving the programmer the tools to modify the syntax of a language to have it resemble their problem domain is something that the designer of `Fortress`, Guy L. Steele, advocated in his *Growing a Language* keynote [22].

Finally, the absence of ambiguity in PEGs allow the exact behaviour of a PEG parser to be directly readable from its specification, easing debugging and optimization tasks.

2.2. PEG Disadvantages

Using PEGs to define languages and construct parsers may ease the process of creating and maintaining a parser, but there are certain pitfalls one should be aware of. Creating a PEG that contains left recursion will lead to an infinite loop in the parser. Therefore, it is not trivial to convert a CFG based language specification to a PEG, as such a specification often uses left recursion to create repetition. The same repetitions can be created in PEGs by using the zero-or-more repetition operator or by converting them to right recursion. *Rats!* has implemented such conversions for direct left recursion[11], however according to [24] no implementation exists where similar conversion is done for indirect left recursion.

Another problem with PEG definitions is hidden prefix capture, which occurs when one of the first alternatives in an ordered choice makes it impossible for later alternatives to ever be considered. For example, the PEG in listing 2.3 can never match the string `'++n'`, because it will commit to the first branch of the choice by matching the first `'+'` and then try to match the character range `a-z` and then fail. Since the choice construction will only try the other rules in the choice if a rule fails we will get no backtracking here as the first alternative of the choice succeeded.

Listing 2.3: Example of hidden prefix capture in a PEG.

```
Rule <- ('+' / '++') [a-z];
```

In this case the solution is pretty simple, simply reversing the order of the literals used in the choice to fix the problem. However, hidden prefix capture can occur in a way where the actual literals involved are defined in non-terminal referenced rules, which can make it very difficult to find.

Another problem with PEGs is spacing. A traditional parser will have a tokenizer that automatically removes unwanted whitespaces from the input. This is not the case for PEGs, we must explicitly define spacing in any PEG we define. Usually a spacing rule or operator is used to consume zero-or-more whitespaces where needed. The problem is that we can end up defining the spaces rule many places in the grammar littering it and making it harder to read. A good rule to follow is to insert space rules after literals and other terminal operators. Listing 2.4 shows the PEG from listing 2.1 on page 17 modified to include spacing.

Listing 2.4: PEG with added spacing.

```
Start <- S Sum;
Sum <- Product ( ( PLUS / MINUS ) Product )*;
Product <- Value ( ( MULT / DIV ) Value )*;
Value <- Number / LPAR Sum RPAR;
Number <- [0-9]+ S;

S <- [ \n\t]*;
PLUS <- '+' S;
MINUS <- '-' S;
MULT <- '*' S;
DIV <- '/' S;
LPAR <- '(' S;
RPAR <- ')' S;
```

2.3. Packrat Parsers

One of the problems with a simple implementation of a PEG-based parser is that, because of backtracking and unlimited lookahead, it could exhibit exponential time performance with regards to input. This can be remedied by using a packrat parser.

Using a packrat parser ensures linear time performance by using memoization to ensure a rule is never parsed more than once for each position in the input. It does this by caching the results of parsing a rule at a position. Before the parser begins parsing a rule it checks the cache for an existing result and uses that instead if it exists. If it has not already been cached, parsing continues as normal and the result of calling the rule is cached on completion.

It should be noted that packrat parsers achieve their performance at the cost of increased memory usage as the cache can potentially end up containing one entry for each type of rule for each position in the input. Additional information on packrat parsing can be found in[6].

2.3.1. Left Recursion

An interesting aspect of packrat parsers is that they can be extended to support left recursion. How it can be done is shown in [24], however the method defined there will break the linear time guarantee of the packrat parser. It should also be noted that a PEG-based packrat parser extended with left recursion will be able to parse an extended class of languages which is a superset of the PEG parsable languages. A formal definition of this class of languages is still undefined[24].

Compiler-Compilers

In this chapter we will be taking a look at existing compiler-compiler tools. We want to find out how the tools are used to build parsers from a grammar specification and how to use the output from the parsers. We will be using this knowledge to support the development of our own parser generator.

The compiler-compiler tools we will be looking at are:

- Lex / Yacc
- ANTLR
- SableCC
- Xtext
- Rats!

These tools have been chosen to represent different approaches with regards to parsing algorithm, handling of semantic actions and parse trees, interfaces between different parts of the compilation process, ease of integration into other systems and grammar syntax readability.

3.1. Lex / Yacc

Lex and Yacc are two separate programs that can be used together to generate a compiler. Lex is a lexical analyzer generator that generates a lexical analysis program from an input of regular expressions and corresponding program fragments. Yacc is a compiler compiler tool that takes a grammar specification in a notation similar to BNF as input and generates a program for parsing the specified input. The Yacc input specification allows the user to specify code to be executed when each production of the grammar is recognized. Listings 3.1 and 3.2 on the next page show the input needed for Lex and Yacc to create a calculator for simple arithmetic expressions.

Note that the issue of ambiguity is addressed by declaring '+' , '-' , '*' and '/' as left-associative. Precedence is decided by the order in which rules are defined so that the last rule defined will have the highest precedence. In this case it means that division and multiplication has higher precedence than subtraction and addition.

Listing 3.1: Lex input for generating a simple calculator.

```
%{
#include <stdlib.h>
#include "y.tab.h"
#ifndef YYSTYPE
#define YYSTYPE int
#endif
extern YYSTYPE yylval;
%}
%%
[0-9]+      {
            yylval = atoi(yytext);
            return INTEGER;
        }
[+*/*()]    return *yytext; /* Operators */
[ \t\n\r]*  /* ignore whitespaces */
%%
int yywrap(void) {
    return 1;
}
```

Listing 3.2: Yacc input for generating a simple calculator.

```
%{
#include <stdio.h>
int yylex(void);
void yyerror(char *);
%}
%token INTEGER
%left '+' '-'
%left '*' '/'
%%
program:
    Expr          { printf("%d\n", $1); }

Expr:
    INTEGER      { $$ = $1; }
    | Expr '+' Expr { $$ = $1 + $3; }
    | Expr '-' Expr { $$ = $1 - $3; }
    | Expr '*' Expr { $$ = $1 * $3; }
    | Expr '/' Expr { $$ = $1 / $3; }
    | '(' Expr ')'
    ;

%%
void yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
}

int main(void) {
    yyparse();
    return 0;
}
```

To generate the parser program both Lex and Yacc have to be invoked. Listing 3.3 shows the commands needed to do so on a unix or linux system.

Listing 3.3: Invoking Lex and Yacc

```
# Use yacc to generate y.tab.c and y.tab.h
yacc -d calculator.y
# Use lex to generate lex.yy.c
lex calculator.l
# Compile the output from both lex and yacc into the resulting parser
cc lex.yy.c y.tab.c -o calculator
```

Lex and Yacc have existed since 1975[16][13]. The original implementation of Lex and Yacc is written in C and generates a LALR based parser written in C code. Lex and Yacc are different from the other tools we examine, because lexical analysis phase is separated from the parser in the program Lex. This allows the lexer to be easily replaced, at the cost of having to maintain two input files.

One major problem with Lex and Yacc is the amount of code that has to be written in the input specification. When encountering an error in the resulting program that error can either originate from the code generated by Lex, Yacc or other external code handwritten by the user. This causes the input specification to be hard to read because the syntax and semantic actions are mixed together and difficult to discern. For example if a multi-line semantic action contains code that has similar syntax to the rule definition syntax as shown in listing 3.4.

Listing 3.4: Example of how semantic actions can be mistaken as rules.

```
%{
#define Value 1
}%
%token Int;
%%
Expr:
    Int '+' Int { $$ = $1
    | Value - Value
    ;
    }
```

Another issue with Lex and Yacc is that they provide no framework for generating and working with parse trees or ASTs. This can be done by combining Lex and Yacc with a tree-building tool, but this again adds to the complexity of tracking down errors in the resulting program.

Tools similar to Lex/Yacc exist for many different languages. Examples include JavaCC for Java[14], fslex/fsyacc for F# (part of the F# distribution) and C# CUP/C# lex for C#[12]. Lex and Yacc are also often referenced from the documentation of other compiler-compilers not based on them, in order to make it easy for people used to the Lex/Yacc input system to migrate to the tool in question. Therefore, despite their age Lex and Yacc are still relevant in the world of compiler-compilers.

More information about Lex and Yacc can be found in their respective papers[16][13].

3.2. ANTLR

ANTLR is a compiler-compiler tool written in Java that uses LL(*) parsing. It generates a lexer and a parser from a single input file, written in an Extended Backus-Naur Form (EBNF) like syntax. It can output parser code in multiple languages including Java, C, C# and Python. An example of the syntax needed to generate a simple parser for sum expressions is show in listing 3.5.

Listing 3.5: ANTLR input for generating a simple calculator.

```
grammar calculator;

tokens {
    PLUS    = '+' ;
    MINUS   = '-' ;
}

/* PARSER RULES */
expr returns [int value]
    : i=integer {$value = $i.value;}
    (
        PLUS i=integer {$value += $i.value;}
        |
        MINUS i=integer {$value -= $i.value;}
    )*;

integer returns [int value]
    : NUMBER { $value = Integer.parseInt($NUMBER.text); };

/* LEXER RULES */
NUMBER : (DIGIT)+;
WHITESPACE : ( '\t' | ' ' | '\r' | '\n' | '\u000C' )+ { $channel = HIDDEN; };
fragment DIGIT : '0'..'9' ;

@members {
    public static void main(String[] args) throws Exception {
        calculatorLexer lex = new calculatorLexer(new ANTLRFileStream(args[0]));
        CommonTokenStream tokens = new CommonTokenStream(lex);

        calculatorParser parser = new calculatorParser(tokens);

        try {
            parser.expr();
        } catch (RecognitionException e) {
            e.printStackTrace();
        }
    }
}
```

Running ANTLR with this input will generate a class for the lexer and one for the parser. The code inside the @members{} block will be placed in the parser class, here we put the entry point for our Java program. Compared to Lex/Yacc the input is more compact, but the input still appears cluttered because of the semantic rules and it is hard to read the input grammar as a specification for the language it represents. To illustrate, the corresponding *expr* production from the grammar is shown in listing 3.6 on the facing page without its semantic actions.

Another major difference from Lex/Yacc is that ANTLR uses LL(*) parsing, which means that the input grammar can not have left recursion. The ANTLRWorks tools

provided with ANTLR v3 can assist the developer in converting left-recursive grammars to grammars using repetitions such as the one in listing 3.6.

An alternative to using inline semantic rules is to have ANTLR automatically generate an abstract syntax tree (AST). This can be done by either using ANTLR's AST tree operators, or by using inline tree rewrite rules.

When using the *operator method* the following operators can be used:

Operator	Description
!	Do not include node or sub-tree (if referencing a rule) in sub-tree
^	Make node root of sub-tree created for entire enclosing rule even if nested in a subrule

Table 3.1.: AST generation operators in ANTLR.

Any token or rule reference not annotated with an operator will simply be added as children to current sub-tree. Listing 3.6 how operators can be used to create an AST for the calculator example.

Listing 3.6: ANTLR calculator without semantic actions but with AST operators instead.

```
/* PARSER RULES */
expr : NUMBER ( ( PLUS | MINUS ) ^ NUMBER ) *;
```

The *rewrite rule method* allows specifying rewrite rules in the grammar that will change the output made by the parser. They are more expressive than the operators and allow the creation of imaginary nodes and reordering of children nodes. Listing 3.7 shows how an AST could be created for the calculator example using rewrite rules.

Listing 3.7: ANTLR calculator with AST rewrite rules.

```
/* PARSER RULES */
expr :
  (l=NUMBER->$l) // Store current result in $expr
  (
    (
      PLUS r=NUMBER -> ^(PLUS $expr $r) // Make a PLUS node
    |
      MINUS r=NUMBER -> ^(MINUS $expr $r) // Make a MINUS node
    )
  )
  ) *;
```

Rewrite rules also allows for grouping of similar items into lists, semantic predicates for deciding which nodes to build and various methods for cloning or copying data between created nodes. The tree nodes generated by ANTLR contains a token representing the root of the sub-tree as well as a list of children. Which means the generated sub-tree is not strongly typed. ANTLR provides some helper classes for traversing and manipulating the generated trees as well as a framework for creating your own trees.

More information about ANTLR can be found in [19] and the ANTLR documentation Wiki [2].

3.3. SableCC

SableCC is an LALR(1)-based compiler-compiler. The grammar is specified in an EBNF-like syntax. SableCC is different from the other tools in that it does not allow any semantic actions in the input specification, only syntax. Instead, SableCC generates a statically typed AST type hierarchy based on the parsing rules defined in the grammar. Any semantic actions must be performed on the ASTs after parsing has completed using the visitor design pattern.

Figure 3.1 illustrates how using SableCC to develop a parser results in a different work flow than using a traditional Lex/Yacc style parser with inline semantic actions.

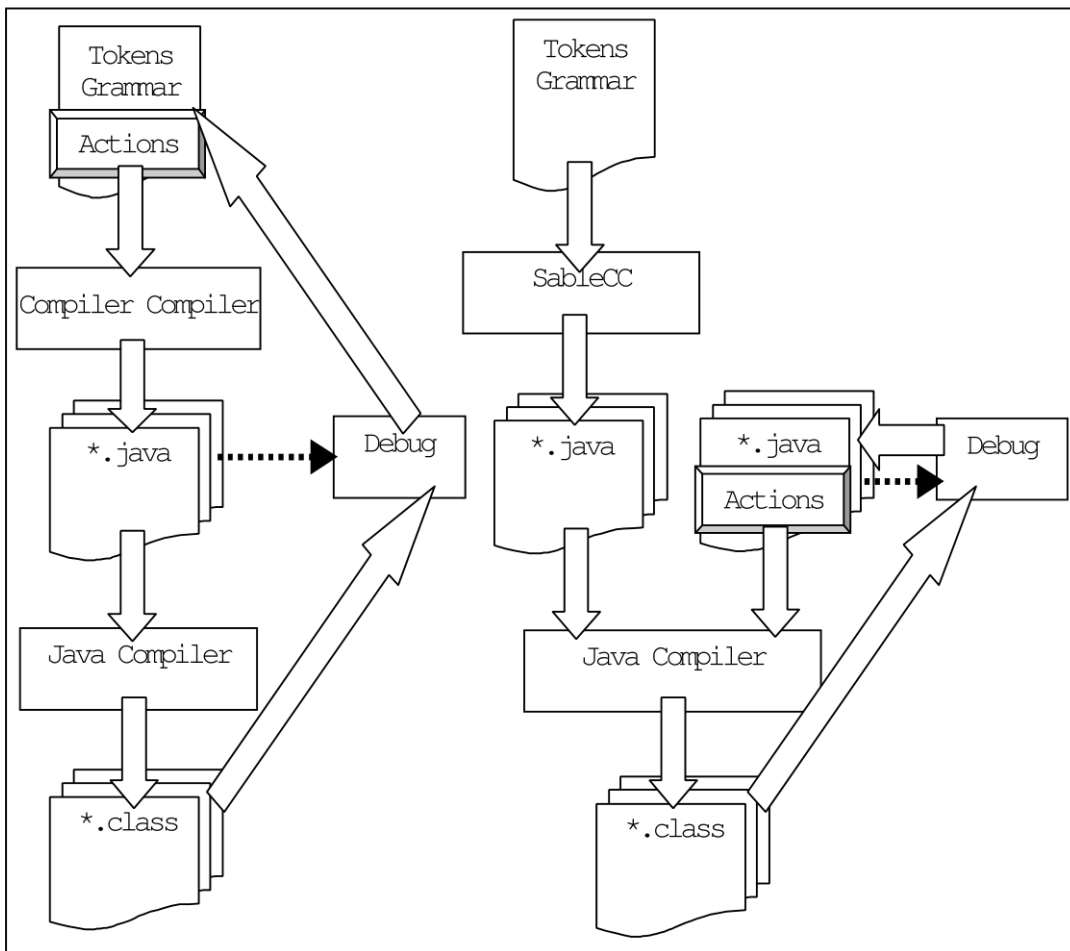


Figure 3.1.: Traditional versus SableCC compiler development. Source: [9].

When developing using SableCC the code for the syntactic analysis is kept separate from the code for contextual analysis and code generation. This means that the code for the syntactic analysis will only have to be generated if changes are made to the syntax of the language being developed. This shortens the debugging cycle when working on the contextual analysis or code generation parts of the compiler as changes can be tested without invoking SableCC.

A SableCC grammar for a simple calculator is shown in listing 3.8.

Listing 3.8: SableCC input for generating a simple calculator.

```
Package calculator;

Helpers
  digit = ['0' .. '9'] ;

Tokens
  integer = digit+;
  plus = '+';
  minus = '-';
  whitespace = ' ' | 10;

Ignored Tokens
  whitespace;

Productions
  program =
    {add} [left]:program plus [right]:integer |
    {sub} [left]:program minus [right]:integer |
    {int} integer;
```

When SableCC is run with this input it will generate a lexer, a parser and the necessary classes for generating and traversing the AST. Note that in the example, choices in the program production have to be explicitly named to allow SableCC to generate a separate class for each choice. It is also worth noticing name annotations which will be converted into methods on the resulting AST node for easy access to children in the AST. The above example will generate the following AST node classes:

PProgram Parent class for all nodes generated from the program production.

AAddProgram Class for nodes generated by the add choice of the program production.

ASubProgram Class for nodes generated by the sub choice of the program production.

AIntProgram Class for nodes generated by the int choice of the program production.

Token classes For each token mentioned in the grammar a class with the name of the token prefixed by a "T" will be created.

When the parser is run it will build an AST consisting of instances of the above classes. We implement semantic actions for our grammar using a visitor based on SableCC generated classes. Such a visitor is shown in listing 3.9.

Listing 3.9: Visitor for the SableCC calculator.

```
public class Interpreter extends AnalysisAdapter {
    public int result = 0;

    @Override
    public void caseAAddProgram(AAddProgram node) {
        // Execute left side, storing its result in result
        node.getLeft().apply(this);
        // Add the right side
        result += Integer.parseInt(node.getRight().getText());
    }

    @Override
    public void caseASubProgram(ASubProgram node) {
        // Execute left side, storing its result in result
        node.getLeft().apply(this);
        // Subtract the right side
        result -= Integer.parseInt(node.getRight().getText());
    }

    @Override
    public void caseAIntProgram(AIntProgram node) {
        // Store the integer in result
        result = Integer.parseInt(node.getInteger().getText());
    }
}
```

The code needed to run the SableCC-generated parser and apply the visitor is shown in listing 3.10.

Listing 3.10: Code to run the SableCC calculator.

```
public class CalculatorTest {
    public static void main(String[] args) {
        try {
            // Run lexer and parser
            Lexer lexer = new Lexer (new PushbackReader(new FileReader(args[0]), 1024));
            Parser parser = new Parser(lexer);
            Start ast = parser.parse() ;

            // Run our visitor on the generated AST
            Interpreter interp = new Interpreter () ;
            ast.apply(interp);

            // Print out the result
            System.out.println("Result: " + interp.result);
        }
        catch (Exception e) {
            System.out.println (e) ;
        }
    }
}
```

SableCC can also do CST to AST transformations. There are two steps to applying this transformation to an existing grammar: First a new grammar for the AST must be created in the input file, excluding nodes and productions that are no longer needed; Secondly rules have to be added to the productions of the CST grammar defining how

the AST nodes should be created. Listing 3.11 shows an example of this for a grammar for a slightly more advanced calculator.

Listing 3.11: Example of SableCC CST to AST transformations.

```
Productions
  expr {-> expr} =
    {add} [left]:expr add [right]:factor {-> New expr.add(left.expr, right.expr)}
    | {sub} [left]:expr sub [right]:factor {-> New expr.sub(left.expr, right.expr)}
    | {factor} factor {-> factor.expr};

  factor {-> expr} =
    {mul} [left]:factor mul [right]:value {-> New expr.mul(left.expr, right.expr)}
    | {div} [left]:factor div [right]:value {-> New expr.div(left.expr, right.expr)}
    | {value} value {-> value.expr};

  value {-> expr} =
    {number} number {-> New expr.number(number)}
    | {parens} left_paren expr right_paren {-> expr.expr};

Abstract Syntax Tree
  expr = {add} [left]:expr [right]:expr
    | {sub} [left]:expr [right]:expr
    | {mul} [left]:expr [right]:expr
    | {div} [left]:expr [right]:expr
    | {number} number;
```

In the CST to AST annotations we can either create and return a new node, or we can return a node of the same AST type created deeper in the hierarchy. In this example the factor and value productions are eliminated using these annotations, resulting in a more concise AST. We can also generate lists of nodes in the AST, listing 3.12 shows how this can be done for the arguments passed to a function call.

Listing 3.12: Example of list creation with SableCC CST to AST transformations.

```
/* This example assumes expr to be defined as in the previous example */
Productions
func {-> func} =
  {function} [name]:identifier left_paren [args]:arg_list right_paren
    {-> New func.function(name, [args.expr])};

arg_list {-> expr*} =
  {single} [arg]:expr {-> [arg.expr]}
  | {multiple} [arg]:expr comma [rest]:arg_list{-> [arg.expr, rest.expr]};

Abstract Syntax Tree
  func = {function} [name]:identifier [args]:expr*;
```

One thing worth noticing is that the AST grammar can not refer to rules defined in the CST grammar. This means that every node you want in the AST has to be defined in the AST grammar. This can potentially lead to having many rules defined both in the CST grammar and the AST grammar, which can make the grammar harder to maintain.

More information about SableCC can be found in [9].

3.4. Xtext

Xtext is a complete language development framework which focuses on making it easy to work with a language once it has been developed. It will not only generate parsers and ASTs, but also allows for the generation of a sophisticated Eclipse-based development environment for a language. The runtime components generated by Xtext are based on the Eclipse Modelling Framework, making them easy to integrate with other Eclipse-based tools. Going into details with everything Xtext can do is out of the scope of this thesis. Therefore, the discussion of Xtext will be limited to its input grammar language, cross-referencing feature and how Xtext handles AST construction. Other information about the Xtext framework can be found on the Xtext homepage[4] as well as in [5].

Listing 3.13 shows an example input grammar for an Xtext-based language, taken from the Xtext documentation.

Listing 3.13: Example of an Xtext grammar.

```
grammar org.eclipse.xtext.example.Domainmodel
  with org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.eclipse.org/xtext/example/Domainmodel"

DomainModel :
  (elements+=Entity)*;

Type:
  DataType | Entity;

DataType:
  'datatype' name=ID;

Entity:
  'entity' name=ID ('extends' superType=[Entity])? '{'
    (features+=Feature)*
  '}';

Feature:
  name=ID ':' type=TypeRef;

TypeRef:
  referenced=[Type] (multi?='*')?
```

Xtext uses an ANTLR back-end for parsing so the syntax is similar to the one used in ANTLR. At the start of the grammar the contents of another grammar is included. This allows for working with grammars in a modular fashion, allowing often used productions to be put in a module and imported into other grammars. The grammar imported in this example will provide the production for the ID used later in the grammar.

Xtext parsers generate statically-typed ASTs. It must be explicitly defined in the grammar which nodes should be put in the tree and which fields should be available on each type of node. This is done by prefixing anything that should be available in the generated nodes with a name followed by one of the three operators '=', '+=' and '?='. Examples of

this in listing 3.13 on the preceding page. The semantics of the operators are as follows:

"name=ID" on line 13 This will create a field on the `DataType` node with the name 'name', containing an `ID` node.

"elements+=Entity" on line 7 This will create a list field on the `DomainModel` node with the name 'elements', containing `Entity` nodes.

"multi?='*'" on line 24 This will create a boolean field named 'multi' on the `TypeRef` node. The field will contain the value `true` if a star was matched at the end of the rule and `false` otherwise.

A special construct in Xtext grammars is cross-references. An example of a cross-reference can be seen on line 16 where the assignment `superType=[Entity]` specifies that a cross-reference to another `Entity` should be added. The full syntax for a cross-reference is `[TypeName|RuleCall]` where the rule `ID` will be used if the `RuleCall` part is not specified. When the parser encounters a cross-reference it will parse the `RuleCall` rule and store the result as a name internally. When parsing is done the linker will establish the proper reference using the stored name as well as a table containing instances of the referenced type. In order for this to work the type referenced with `TypeName` must have a 'name' field that can be used to compare with the temporarily stored name in the cross-reference. Xtext provides a full framework for working with cross-references, including scoping rules and `import`-statements with wildcards. This makes it easy to generate compilers with simple name resolving and type checking.

Xtext allows rewriting of the generated AST by using *Assigned Actions*. They work in a way similar to the AST rewriting operators from ANTLR, except they have been defined more explicitly. Listing 3.14 shows an example of how the generated AST for a binary expression can be rewritten.

Listing 3.14: Example of AST rewriting in Xtext.

```
Expression :
  {Operation} TerminalExpression ({Operation.left=current}
  op='+' right=Expression)?;

TerminalExpression returns Expression:
  '(' Expression ')' |
  {IntLiteral} value=INT;
```

The semantics of the action `{Operation.left=current}` are equivalent to the code shown in listing 3.15: A new `Operation` node is created and its `left` field is assigned to the currently matched `Operation`. Afterwards `current` is set to the newly created `Operation`.

Listing 3.15: Semantics of AST rewriting in Xtext.

```
Operation temp = new Operation();
temp.setLeft(current);
current = temp;
```

The Xtext rewrite rules make it possible to generate left-associative parse trees even though the grammar makes use of right-recursion. Left-associative parse trees are important for left-associative arithmetic expressions.

3.5. Rats!

Rats! is a compiler generator based on packrat parsing. Grammars for Rats! are written in a PEG-like syntax. Listing 3.16 shows an example of the input grammar for a simple calculator in Rats!.

Listing 3.16: Example of a simple calculator in Rats!.

```
public Sum Sum = Spacing head:Int tail:SumTail* { yyValue = new Sum(head, tail.list()); };
SumTail SumTail = op:('+ ' / '- ') Spacing i:Int { yyValue = new SumTail(op, i); };
Integer Int = 1:Digits Spacing { yyValue = new Integer(1); };
String Digits = [0-9]+;
transient void Spacing = (' ' / '\t' / '\f' / '\r' '\n' / '\r' / '\n')*;
```

Each production in the grammar specifies a return type. The semantic action for each rule must produce an object of this type and a class for each type has to be implemented by the user. In the above example two user-defined classes (`Sum` and `SumTail`) are used as well as the internal Java class `Integer`. Rats! uses labels to identify the different parts of a production in the semantic rules. Anything that appears as a repetition will be stored in nested instances of a generic `Pair<T>` class. A collection of nested pairs can be converted to a list by called the method `list()` on the top-level `Pair`, as done in the example with `tail.list()`. Rules in Xtext grammars can be prefixed with modifiers that change their semantic meaning. In the example the modifiers `public`, `void` and `transient` are used. `public` exposes a production as a public method in the generated parser. `void` tells the parser that a production should not generate a node and `transient` disables the memoization of the production. For more information about modifiers available in Rats! grammars see the Rats! paper[11].

The major disadvantage of the above grammar is that the user has to implement the classes used for the parsing tree himself. A way to avoid this is to use Rats!'s special generic type, which will create a generic parsing tree, however this tree will not be strongly typed. When using generic productions Rats! also allow for direct left recursion. This allows the creation of a parser with Yacc-style semantic actions like the one shown in listing 3.17 on the next page.

Rats! also provide a modules system as seen in Xtext. However unlike Xtext Rats! allows for the modification of rules in an imported grammar. This makes it easy to extend parsers made with Rats! with both syntax and semantic actions. This feature is used in the Fortress language that uses Rats! for its internal parser. Many of the basic

Listing 3.17: Example of using generic nodes in Rats!.

```

public generic Program = Spacing Sum;

generic Sum =
  Sum '+' Spacing i:Int { RatsTest.Result += (Integer)(i.get(0)); } /
  Sum '-' Spacing i:Int { RatsTest.Result -= (Integer)(i.get(0)); } /
  i:Int { RatsTest.Result = (Integer)(i.get(0)); };
generic Int = l:Digits Spacing { yyValue = GNode.create("Int", new Integer(1)); };
String Digits = [0-9]+;
transient void Spacing = (' ' / '\t' / '\f' / '\r' '\n' / '\r' / '\n')*;

```

language features in Fortress are implemented as Rats! modules that can be easily modified or extended[20].

3.6. Observations

3.6.1. Grammar Syntax

During the examination of the above tools, the most interesting issues is how the grammars for all of the input grammars easily become littered when labels and semantic actions or tree rewriting rules are added. This results in difficult to read grammars and thereby hindering maintenance. Some of the noise can be avoided by using the SableCC approach and move semantic actions away from the grammar and work exclusively on ASTs. This still leaves the problem of labels and tree rewriting rules. In our opinion the problem with labels and rewrite rules is that the syntax used form them is too verbose. In the SableCC example nearly everything gets a label and each rule is defined in three different places: Once for the actual rule, once for its AST counterpart and once in the tree rewriting rule. It would be preferable if some of this information could be inferred or combined.

3.6.2. Semantic Actions and PEGs

Apart from littering the input grammars, inline semantic actions cause additional problems when used with PEGs and other left-derivative parsers. Consider the naive approach to specifying semantic actions on an example PEG grammar shown in 3.18.

Listing 3.18: Example of wrong order of execution of semantic actions.

```

Sum <-
  i:Int SumTail* { result = i; }
SumTail <-
  ('+' i:Int) { result += i; }
  /
  ('-' i:Int) { result -= i; }

```

In this grammar, the order of execution of the semantic actions will be wrong: First the semantic actions in the `SumTail` rule will be executed and only when they have all

been processed will the semantic action of the `Sum` rule be executed, overwriting the value stored in `result`. In the examined tools this problem is addressed in two ways: In ANTLR and Xtext semantic rules can be defined anywhere in the grammar, allowing the initial integer to be stored before the tail of the expression is processed. In Rats! the problem is solved by using either a user-defined tree or the generic tree to store temporary results in. However, both of these approaches adds additional notation to grammar, which makes it harder to read.

Another problem with semantic actions in a PEG-based parser is that PEGs allow backtracking. This means that it has to be possible to undo semantic actions whenever the parser backtracks. The easy way to solve this problem is to generate a tree from the parsing and then discard the sub-tree generated by rules that fail. With this approach a successful parse will result in a parse tree containing only the nodes matching the input. This tree can then be traversed in order to perform semantic actions.

3.6.3. Improving the Learning Curve

When being introduced to a new tool, a good tutorial provides a starting point from which one can get up and running. Once the tutorial has been completed, an understanding of how the tool works has been achieved and the examples from the tutorial can be used as a base for the project the developer originally had in mind.

Some of the tools we have been examining do not provide tutorials. This makes it much harder to understand the tool and get started implementing a language using it. While we could gather the information from other sources, not having a tutorial or introduction made it much more difficult than necessary.

Since there are many compiler-compiler tools to choose from, not having a good introducing tutorial may cause potential users to choose a different tool. Which is why we think it is very important to provide a good tutorial on how to use our parser generator.

The Object-Oriented Parsing Expression Grammar (OOPEG) Parser Generator

In this chapter we will outline the core visions and requirements we have for our parser generator OOPEG.

In the observations section from the previous chapter we found that in the existing tools, the input grammars easily become very verbose and therefore difficult to read and understand. We also found that doing semantic actions during parsing can cause problems with PEGs because of parsing order and backtracking.

Because of these observations and because modern compilers do multiple contextual analysis passes as described in the introduction chapter 1 on page 11, we decided to not have any semantic actions in the input grammar. This is what makes OOPEG a parser generator and not a compiler-compiler. OOPEG generates the semantic analysis phase of a compiler, and tools to support the remaining phases, which are instead implemented in the target language C#.

There are two components of OOPEG, one is the object-oriented PEG-based **parsing engine** and the other is the **parser generator**.

4.1. Parsing Engine

The parsing engine should implement all the operators defined in chapter 2 on page 17 and should apply packrat parsing in order to achieve linear time performance. Parsers should be built using an object-oriented approach so that each operator in the PEG is represented by an instance of a class. This allows for the later addition of new operators as well as the possibility to change the compiler at runtime. Also the operator objects can be composed into arbitrarily complex PEG parsers, as well as allow the combination of multiple separate PEGs using simple object references.

The parsing engine should be able to generate statically typed ASTs and do CST to AST conversions during the parse process, such that the compiler developer will only have to work with nodes that have semantic value after the parsing.

Since OOPEG is going to be used in the production environment at Progressive Media it is important that the parsing engine performs reasonably well. As packrat parsing is applied performance should not only be measured by speed, but also by heap utilization

which can easily become a bottleneck in packrat parsers[11]. Furthermore, OOPEG parsers should be thread safe so that it is possible to run multiple parsers at the same time.

4.2. Parser Generator

The parser generator builds parsers that utilize the parsing engine. It does this by parsing an input grammar and composing the parsers from the parsing engine operators.

The input used by OOPEG should be as close to the formal PEG language definition as possible. The input grammar of OOPEG should be purely declarative, specifying only what the parser should do, not how it should do it. Furthermore, annotations used to provide semantic information such as CST to AST conversion should be kept at an absolute minimum. It is important that the syntax used to describe CST to AST transformations becomes less verbose than the one used by SableCC.

OOPEG should be implemented by bootstrapping. This means that OOPEG should itself be an OOPEG parser that takes OOPEG grammars as input and outputs the components mentioned above. This approach makes it easier to later replace parts of the implementation, for example to generate code for another programming language than C#.

It is our wish that OOPEG should be easy to use and easy to get started with. Therefore we provide a tutorial on how to create a parser with OOPEG. This tutorial is presented in chapter 5 on the next page.

The three chapters following the tutorial will detail the implementation of OOPEG, however they should not be necessary to read in order to use OOPEG.

OOPEG Tutorial

This tutorial will show how to use OOPEG to create a calculator program that can perform simple arithmetic calculations, store results in variables and print out results. We assume the reader knows object-oriented programming and particularly C#.

The calculator will be able to run programs like the one shown below:

```
var x = 1;
var y, z = 3;
y = z - x;

output x;
output z - y - x;
output (3 - 2 - 1) + 11 * 7;
```

In order to get started with creating the calculator, OOPEG has to be installed. This is done by installing a Visual Studio 2010 template for OOPEG, which can be found at <http://felizk.dk/oopeg/>. OOPEG is not supported in Visual Studio versions earlier than 2010. The template can be installed by double-clicking on the downloaded .vsf file and accepting to install it. During the installation you will get a warning about the template not being signed, feel free to ignore it.

Once the template has been installed, create a new project in Visual Studio 2010 and select Visual C# => Oopeg Parser as the project type. Use the name OopegTutorial for the project. This will create a mock-up project with a simple parser that parses the string "Start here!". You can build the project and run it to test that everything works.

5.1. Input Grammar

In order to create our calculator we first have to create a grammar for it. Open the `Grammar1.peg` file in the project and change it to the contents shown in listing 5.1 on the facing page. You can rename the grammar file to something else than `Grammar1.peg` if you like.

The first line specifies the namespace used for this parser. All the code generated by OOPEG will be put into a C# namespace with the name defined here.

The following two lines define the two trees we will use to represent our source code. Unlike other tools OOPEG does not use one big AST representing the entire source code for implementing semantic actions. Instead it is possible for the user to specify different trees with different meanings. In this case the `StatementTree` will hold statements such as declarations, assignments and output commands and the `ExpressionTree` will hold arithmetic expressions.

Note the underscore `'_'` which is special OOPEG operator for matching zero or more **spaces**. Since OOPEG is based on PEGs, spacing has to be put explicitly in the grammar where it is needed. A way to achieve spacing rules similar to those from a tokenizing parser is to add a spacing operator at the start of the grammar and after every literal operator, as we have done in this tutorial example.

Next the `Start` rule is defined. The `'<-'` assignment indicates that this rule should create a node in the AST, however this node should not be part of one of the explicitly defined trees. Nodes defined in this way will typically be accessed through a member on their parent node, as will be the case with the `Identifier` node mentioned below.

The `Statement` rule defines that a statement should be one of `Declaration`, `Assignment` or `Output` followed by a semicolon. However, as this rule does not provide any additional semantic information, we are not interested in creating a node for this rule. Therefore, it is defined as a **pass-through** rule using the `'='` syntax.

The rules `SingleDeclaration`, `Assignment` and `Output` is where we have the data we actually want to be present as nodes in our statement tree. Therefore, the rules are defined using **tree-type** syntax `'<: StatementTree <-'`, causing them to create nodes for the statement tree. Note that since the `Statement` and `Declaration` rules are defined as *pass-through* rules using the `'='` syntax, the generated `StatementTree` nodes will "bubble up" through the parse tree and end up as the children of the `Start` node in the final AST.

The `ArithmeticExpression` rule is the top rule in our arithmetic expressions. This rule shows how **left recursion** can be used. Also note that the choice between `PLUS` and `MINUS` has been made as an **inline** rule. This is done to expose it so it can be used in the code generation later on.

ArithmeticExpression is part of the ExpressionTree tree, but it has been marked **cullable** using the '?:' syntax. This means that the rule should *not* generate a new node if it has exactly one child of type ExpressionTree. This case occurs if the top choice of ArithmeticExpression select the alternative 'Product', which would produce an ArithmeticExpression with a single Product node and nothing else, we are not interested in that so we cull it.

The Product rule shows the standard method for making repetitions using PEGs.

Listing 5.1: The OOPEG grammar used in the tutorial.

```
namespace OopegTutorial {
  tree StatementTree;
  tree ExpressionTree;

  Start <-
    _ Statement+ ![\u0000-\uFFFF];

  Statement =
    (Declaration / Assignment / Output) SEMI;

  Declaration =
    VAR SingleDeclaration (COMMA SingleDeclaration)*;

  SingleDeclaration : StatementTree <-
    Identifier (EQUALS ArithmeticExpression)?;

  Assignment : StatementTree <-
    Identifier EQUALS ArithmeticExpression;

  Output : StatementTree <-
    OUTPUT ArithmeticExpression;

  /* Arithmetic expressions */
  ArithmeticExpression ? : ExpressionTree <-
    ArithmeticExpression (PlusMinus <- PLUS / MINUS) Product / Product;

  Product ? : ExpressionTree <-
    Value (Tail <- (STAR / SLASH) Value)*;

  Value =
    Number / VariableReference / LPAR ArithmeticExpression RPAR;

  Number : ExpressionTree <-
    [0-9.]+ _;

  VariableReference : ExpressionTree <-
    Identifier;

  Identifier <- [A-Za-z] [A-Za-z0-9]* _;

  /* Tokens */
  LPAR == '(' _;
  RPAR == ')' _;
  STAR == '*' _;
  SLASH == '/' _;
  PLUS == '+' _;
  MINUS == '-' _;
  SEMI == ';' _;
  EQUALS == '=' _;
  COMMA == ',' _;

  VAR == 'var' _;
  OUTPUT == 'output' _;
}
```

The `Number` and `VariableReference` rules generate `ExpressionTree` nodes and are therefore defined using the tree-type annotation `: ExpressionTree <-`.

The `Identifier` rule states that nodes should be created for it (it is defined using `<-`), but it does not belong to a tree. This is done so `Identifiers` will appear as members on the nodes that refer to them, such as `VariableReference` and `Assignment`.

The rest of the file defines **macros** that are used to parse literals and spaces. The difference between a macro `'==`' and a *pass-through* rule `'=`' is that macros do not show up in the parser that is generated from the grammar. They are like macros in C and are expanded in all the places they are used before the parser is generated. Because of the expanding nature of macros, you cannot have cyclic macros, because then they cannot be expanded before code generation.

5.1.1. Choosing Rule Types

When deciding how to define rules while constructing a language consider the following:

Does the rule have any semantic value for the language? If it does, use the `<-` assignment, if not then make it a pass-through rule using `'=`'.

Is the rule part of a tree? If it is mark it using the tree-type syntax `: TypeOfTree <-`.

Is the rule part of a tree, but should not create a node in some cases? Then you use the cullable `'?: TypeOfTree <-`' syntax, this means that the rule will not create a node if it has only a single child node of the same tree-type. This is often the case for binary operators or other rules that need two children to have any semantic value.

5.2. Invoking the Parser

In order to invoke the parser from our newly created grammar change the following:

```
Grammar1 root;  
if (Grammar1.Grammar.TryParse("Start Here!", out root))
```

in the `Program.cs` file into:

```
Start root;  
if (Start.Grammar.TryParse("var x = 1 + 2 * 3; output x;", out root))
```

Note: If you have used another project name than `OopegTutorial` for your project you will have to change the namespace used in the `Program.cs` file to `OopegTutorial` or add an `using OopegTutorial;` statement in the top of the file.

You should now be able to compile and run the parser used for the calculator. Try changing the input string used in the `TryParse` method so that the parsing will fail and the parser will print out errors.

5.3. Implementing Semantics

Now our parser can parse the language used for our calculator and tell us if we are using correct syntax or not. We also want our calculator to be able to actually perform the calculations and output results. In order to do this we have to write a visitor class that works on the AST generated by the parser. In this case we will actually need two visitors since we have defined two tree-types in our grammar, `StatementTree` and `ExpressionTree`, which will each need their own visitor to process their nodes.

The code for these two visitors are shown in listing 5.2 and 5.3 on the following page.

Listing 5.2: The visitor used to process statements.

```
namespace OopegTutorial
{
    public class StatementVisitor : StatementTreeDepthFirstAdapter
    {
        private Dictionary<string, SingleDeclaration> environment = new Dictionary<string,
            SingleDeclaration>();
        private ArithmeticVisitor aVisitor;
        public StatementVisitor()
        {
            aVisitor = new ArithmeticVisitor(environment);
        }

        public override void VisitSingleDeclaration(SingleDeclaration node)
        {
            string id = node.Identifier.ToString().Trim();
            if (environment.ContainsKey(id))
            {
                string message = string.Format("Identifier '{0}' already defined at {1}", id,
                    node.Origin.LineAndPosition(environment[id].Index));
                throw new EvaluationException(message, node.Origin.LineAndPosition(node.Index));
            }
            else
                environment[id] = node;

            if (node.ExpressionTree != null)
                node.AssignValue(node.ExpressionTree.Accept(aVisitor));
        }

        public override void VisitAssignment(Assignment node)
        {
            string id = node.Identifier.ToString().Trim();
            if (environment.ContainsKey(id))
                environment[id].AssignValue(node.ExpressionTree.Accept(aVisitor));
            else
                throw new EvaluationException("Cannot assign to undefined variable: " + id, node.
                    Origin.LineAndPosition(node.Index));
        }

        public override void VisitOutput(Output node)
        {
            Console.WriteLine(node.ExpressionTree.Accept<double>(aVisitor));
        }
    }
}
```

Listing 5.3: The visitor used to evaluate expressions.

```

namespace OopegTutorial
{
    class ArithmeticVisitor : ExpressionTreeEvaluator<double>
    {
        private CultureInfo ci;
        private Dictionary<string, SingleDeclaration> environment;

        public ArithmeticVisitor(Dictionary<string, SingleDeclaration> environment)
        {
            ci = new CultureInfo("en-US");
            this.environment = environment;
        }

        public double EvalProduct(Product node)
        {
            double result = node.SubTreeChild.Accept(this);

            foreach (var p in node.TailChildren)
            {
                switch (p.ChoiceIndex)
                {
                    case 0:
                        result *= p.ExpressionTree.Accept(this);
                        break;
                    case 1:
                        result /= p.ExpressionTree.Accept(this);
                        break;
                    default:
                        break;
                }
            }

            return result;
        }

        public double EvalArithmeticExpression(ArithmeticExpression node)
        {
            ExpressionTree left = node.GetSubTreeChild(0);
            ExpressionTree right = node.GetSubTreeChild(1);

            if (node.PlusMinus.ChoiceIndex == 0)
                return left.Accept(this) + right.Accept(this);
            else
                return left.Accept(this) - right.Accept(this);
        }

        public double EvalNumber(Number node)
        {
            return double.Parse(node.ToString(), ci);
        }

        public double EvalVariableReference(VariableReference node)
        {
            string id = node.Identifier.ToString().Trim();
            if (environment.ContainsKey(id))
            {
                if (environment[id].HasBeenAssigned)
                    return environment[id].Value;
                else
                    throw new EvaluationException("Use of unassigned identifier: " + id, node.
                        Origin.LineAndPosition(node.Index));
            }
            else
                throw new EvaluationException("Undefined identifier: " + id, node.Origin.
                    LineAndPosition(node.Index));
        }
    }
}

```

Since we want to execute our statements in the order they appear in the code we can let the visitor used for statements inherit from the `StatementTreeDepthFirstAdapter` class. This means that the visitor will by default visit all the nodes in the generated `StatementTree` in a depth first manner. We will only have to implement methods for those nodes which we are actually interested in.

The visitor used for evaluating arithmetic expressions inherits from `ExpressionTreeEvaluator<double>`. This means that the visiting methods should each return a `double` value, which is what we want our arithmetic expressions to evaluate to. Unlike the `DepthFirstAdapter` used for statements this base class has no automatic system for traversal of the AST. We must therefore manually call the `Accept()` method on the sub-nodes we want to be evaluated in each method.

The calculator has to keep track of the assignment of variables. This is done using a string-indexed dictionary that contains the `SingleDeclaration` node that declared the variable. Since we have to know whether a variable has been assigned or not we decorate the AST nodes for `SingleDeclarations` with a boolean value `HasBeenAssigned`. This is done by using partial classes as shown in listing 5.4.

Listing 5.4: Code for AST decoration.

```
namespace OopegTutorial
{
    public partial class SingleDeclaration
    {
        public bool HasBeenAssigned = false;
        public double Value = 0;

        public void AssignValue(double value)
        {
            HasBeenAssigned = true;
            Value = value;
        }
    }
}
```

A few things should be noted about how the visitors work. First, the `StatementVisitor` invokes the `ArithmeticVisitor` whenever it has to use the value of an `ExpressionTree` node. This way only the `StatementVisitor` has to be run on the AST in order to run our calculator. Secondly, the `EvalProduct` and `EvalArithmeticExpression` methods on the `ArithmeticVisitor` shows two ways of implementing code for a node representing repetitions: `EvalProduct` is for normal PEG repetitions and `EvalArithmeticExpression` is for left recursion. It should also be noted that `EvalArithmeticExpression` does not have to implement the case where there is only one `Product` childnode present as a `ArithmeticExpression` node with only one child will have been automatically removed from the tree.

Finally, in order to have the parser give nice error messages when variables are not handled properly we add an `EvaluationException` which will contain both the error message and the line and position it occurred on. The code for the `EvaluationException` class is shown in listing 5.5.

Listing 5.5: `EvaluationException` class used in the tutorial.

```
namespace OopegTutorial
{
    public class EvaluationException : Exception
    {
        public LineAndPosition lp;
        public EvaluationException(string errorMessage, LineAndPosition lp)
            : base(errorMessage)
        {
            this.lp = lp;
        }

        public override string Message
        {
            get
            {
                return lp.ToString() + ": " + base.Message;
            }
        }
    }
}
```

5.4. Putting It All Together

In order to invoke our new visitors, change the `Program.cs` to what is shown in listing 5.6 on the next page.

This will change our parser to read input from a file called `Input.txt`. Add this file to the Visual Studio project as a text file, select it in the solution explorer and select `Copy if newer` under the `Copy to Output Directory` property in the `Properties` window. This will copy the file to the output directory before running the parser, allowing you to edit the input for your parser directly in Visual Studio.

Note that now that we are using a file as an input we have to handle errors ourselves. In this case it is done by creating an instance of `ConsoleErrorLogger` and passing it to `TryParse`. If the parsing fails we can use `logger.WriteToConsole();` to print all errors to the console. If not working in a console environment it is possible to develop your own `ErrorLogger`.

When the program is run it should output the following:

```
Parse Success, running visitor
1
0
77
```

The complete source code can be found at: <http://felizk.dk/oopeg/>

Listing 5.6: Code for invoking the calculator.

```

namespace OopegTutorial
{
    class Program
    {
        static void Main(string[] args)
        {
            Start root;
            ConsoleErrorLogger logger = new ConsoleErrorLogger();
            if (Start.Grammar.TryParse(Context.FromFile("Input.txt"), logger, out root))
            {
                Console.WriteLine("Parse Success, running visitor");
                StatementVisitor sv = new StatementVisitor();
                try
                {
                    root.StatementTreeChildren.Accept(sv);
                }
                catch (EvaluationException e)
                {
                    Console.WriteLine("Error: " + e.Message);
                }
            }
            else
            {
                logger.WriteToConsole();
                Console.WriteLine("Parse Failed");
            }
            Console.ReadLine();
        }
    }
}

```

5.5. Advanced Usage

Some modern languages are designed to use indentation to indicate blocks of code. Examples of such languages are Python, F# and Haskell. CFGs can not be used to express the block structure of such languages. This problem is typically solved using ad-hoc parsing or a preprocessor that has to be run on the source code before it can be parsed using a CFG. In [1] the indentation problem is formalized and the block-skip language is introduced as a prototype language for expressing the problem. Andersen shows how a PEG can make use of predicates to correctly parse the block-skip language. An OOPEG implementation of the block-skip language is shown in listing 5.7 on the following page.

This example shows how predicates can be used for unlimited lookahead in PEGs, allowing the definition of languages that are not possible to define using CFGs. It is also a good example on how using treetypes and pass-through rules makes it easy to create a simple AST from a complex grammar. All that is needed to get an AST with properly nested nodes is to mark the `Block` and `Skip` rules as belonging to the `Statement` tree. Defining all the other rules as pass-through rules further reduces the amount of nodes in the AST so it ends up containing only block and skip nodes and nothing else.

Listing 5.7: OOPEG grammar for the block-skip language.

```
namespace blockskip
{
    tree StatementTree;

    Start <- (Block / Skip '\n' / '\n')*;
    Block : StatementTree <- BEGIN_BLOCK STMTS;

    Skip : StatementTree <- 'skip';

    BEGIN_BLOCK = 'block' '\n'+;

    INDENT_LEQ_REC = SPACE INDENT_LEQ_REC SPACE / Block / Skip '\n'+;
    INDENT_G_REC = SPACE INDENT_G_REC SPACE / Block SPACE / Skip '\n' SPACE;

    INDENT_G = SPACE INDENT_G SPACE / BEGIN_BLOCK SPACE / Skip '\n' SPACE;

    SINGLE_LINE_BLOCK = &(SPACE* BEGIN_BLOCK) !(INDENT_G);

    STMTS = !SINGLE_LINE_BLOCK (LINE / LASTLINE);

    LINE = &INDENT_LEQ_REC !INDENT_G_REC SPACE+ (Block / Skip '\n'+) STMTS;

    LASTLINE = SPACE+ (Block / Skip !Skip !'block' '\n'*);

    SPACE = ' ';
}

```

5.5.1. Using Multiple Grammars

Additional grammars can be added to a OOPEG Visual Studio project by right-clicking on the project, selecting Add => New item... and selecting the Grammar template. You can refer to rules defined in other files, however rules residing in another namespace must be referred with their fully qualified name,

5.5.2. Command-line Invocation of OOPEG

If you want to use OOPEG outside of visual studio, you can use the command line program `Oopeg2.exe`. `Oopeg2.exe` is used as follows:

```
Usage: Oopeg2.exe OutputDir GrammarFile [GrammarFile2 GrammarFile3 ...]
```

When run, `Oopeg2.exe` will generate the parser(s) for the specified grammar(s) in the `OutputDir`. If `-r` is specified for the `OutputDir` argument OOPEG will output the source code file for each grammar in the same directory as the grammar file.

`Oopeg2.exe` can be downloaded from <http://felizk.dk/oopeg/>.

Syntax Tree Construction

This chapter describes the methods we have used to generate syntax trees from parsing expression grammars. We start out with generating a concrete syntax tree and proceed to introduce techniques that allow us to reduce the size of the generated tree and cull away uninteresting nodes.

6.1. Concrete Syntax Tree

One approach is to build a complete parse tree or Concrete Syntax Tree (CST), this is done by creating nodes in the tree for every operator used. We create leaf nodes for every terminal parsed and tree nodes for the aggregate operators. We do not create nodes for syntactic predicates or anything matched by their children. If an operator fails to parse correctly we create an error node which can be used to provide proper error messaging on unsuccessful parsing attempts. Every node contains starting position and length of the substring they represent. For aggregate operators these indexes span the entire text matched by child operators.

We create nodes for the following operators:

Sequence We create a sequence node for all sequences that are not the direct child of non-terminal nodes.

Choice *Choice* nodes are marked with an index of which choice was successfully matched. It has references to unsuccessful matches as well for error handling purposes.

Repetition For every repetition we create a *Repetition* node, all the children of a repetition node are Sequence nodes.

Terminals Matched terminals such as character ranges and literals are also represented by nodes.

Non-terminals Non-terminal nodes are Sequence nodes that are marked with the rule they represent.

This approach will allow us to uniquely identify and access every part of the matched input by the operators used to match it. Accessing and using such a CST in C# is very verbose and difficult to read. Take the following example grammar:

```
Start <- A+ ('b' / 'c')*;
A <- 'a';
```

Matching this grammar against "aabc" yields the CST seen on Figure 6.1.

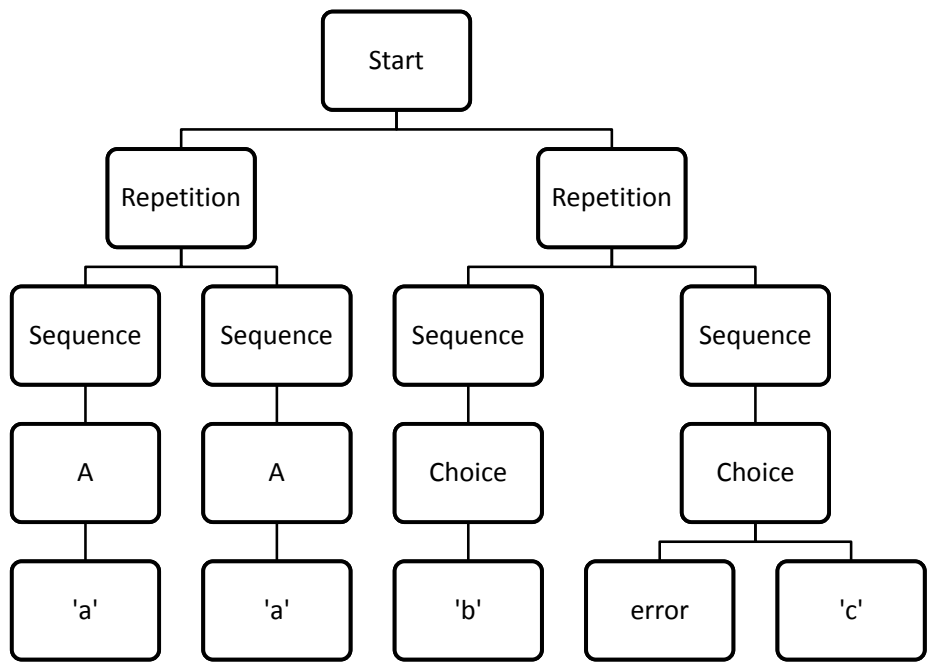


Figure 6.1.: CST (or parse-tree) for "aabc".

Looking up all matched 'c's using C# would look something like this:

```
var start = Start.Parse("aabc");
foreach(var sequenceNode in start[1])
{
    if(sequenceNode[0].ChosenIndex == 1)
    {
        var cNode = sequenceNode[1];
        // work with the 'c' node.
    }
}
```

It is obvious that several of these nodes could be left out, especially the extra sequences. However for consistent lookup according to the grammar, the sequences are necessary.

This approach is very fragile to changes in the grammar, when a rule is changed there is a risk that it invalidates indexes and thereby code. Such errors could be difficult to find, for example if our previous Start rule was changed to:

```
AltStart <- A+ ('e' / 'f')* ('b' / 'c')*
```

The code we wrote would now be working on all 'f' nodes instead of 'c' possibly without compile errors or runtime exceptions.

Another concern with this naive approach is memory usage when creating nodes for every operator. A repeated character range (e.g. [0-9]+) would cause a Sequence node and a character range node to be created for every index matched by the expression. An approach we used to reduce the amount of nodes created was to mark some rules as to not create child nodes. This results in *token*-like nodes that have no child nodes but represent the input matched by the expression of a Rule, which is useful for rules such as those matching whitespaces.

6.2. Abstract Syntax Tree

The CST approach is what is needed if we want a precise image of how an input was matched using a PEG parser. However, it is difficult to work with and contains much information that we may or may not be interested in.

The next step is to allow us to select what it is we are interested in from our grammar and only generate that data for further use. The first thing we need is a construct that groups and names interesting expressions, which we already have in *rules*.

If we simply only create nodes for every matched rule we will have reduced our tree quite significantly, however we will have lost some structure of the tree. For example, if we have the grammar:

```
Start <- A+ '?' A+ ':' A+;
A <- 'a';
```

Start nodes would have a list of A children, however it would not be possible to know which A nodes correspond to which repetition. One solution is to add more rules to name and capture the individual lists of A nodes:

```
Start <- Predicate '?' True ':' False;
Predicate <- A+;
True <- A+;
False <- A+;
A <- 'a';
```

However, this approach may litter the name scope and give rise to name clashing or longer names such as StartPredicate causing the rule to be:

```
Start <- StartPredicate '?' StartTrue ':' StartFalse;
StartPredicate <- A+;
StartTrue <- A+;
StartFalse <- A+;
A <- 'a';
```

Taking a note from popular regular expression implementations we wanted to add named capturing groups to alleviate this problem to be able to do something like the following:

```
Start <- (Predicate: A+) '?' (True: A+) ':' (False: A+)
```

These capturing groups are very similar to rules, and to keep things simple we can simply have rules be inlineable:

```
Start <- (Predicate <- A+) '?' (True <- A+) ':' (False <- A+);
```

The inline rules are not accessible from other rules in the definition and therefore do not litter the naming scope.

6.2.1. Choice Index

There may also be rules in which it would be advantageous to know the result of a choice;

```
Sum <- Int '+' Int / Int '-' Int;  
Int <- [0-9]+;
```

The meaning of the Sum node is changed based on which choice was matched, but the resulting AST would be identical. For these cases we propose annotating nodes with the result of the first top-level choice in the expression. If more granularity is needed such as in the following case:

```
Sum <- Int (('+' / '-') Int)*;  
Int <- [0-9]+;
```

Inline rules can be used to name the repeated expression:

```
Sum <- Int (Tail <- ('+' / '-') Int)*;  
Int <- [0-9]+;
```

In which case the Tail node would be annotated with the result of the choice operator.

There is still one scenario where we want to create nodes for terminals and operators, which is when an error occurs. The error nodes are attached to each non-terminal node, even if the non-terminal node successfully parsed. How the error nodes are used in error handling can be seen in section 6.6 on page 60.

6.3. Reduced Abstract Syntax Tree

We have reduced the syntax tree from a CST to an AST, however there is another abstraction level we want to address. Consider the following grammar:

```
Sum    <- Product ('+' Product)*;
Product <- Value ('*' Value)*;
Value  <- SubExpr / Int;
SubExpr <- '(' Sum ')';
Int    <- [0-9];
```

Parsing the input "3*(2+2)" we would yield the AST seen on the left in Figure 6.2. There are many nodes in that AST which have no semantic value, they are only created as a result of the rules of the grammar. If we were to remove all those nodes we would end up with our desired reduced AST which is picture on the right in Figure 6.2. To achieve this reduced AST we introduce two concepts: **Pass-through rules** and **Tree-Types**.

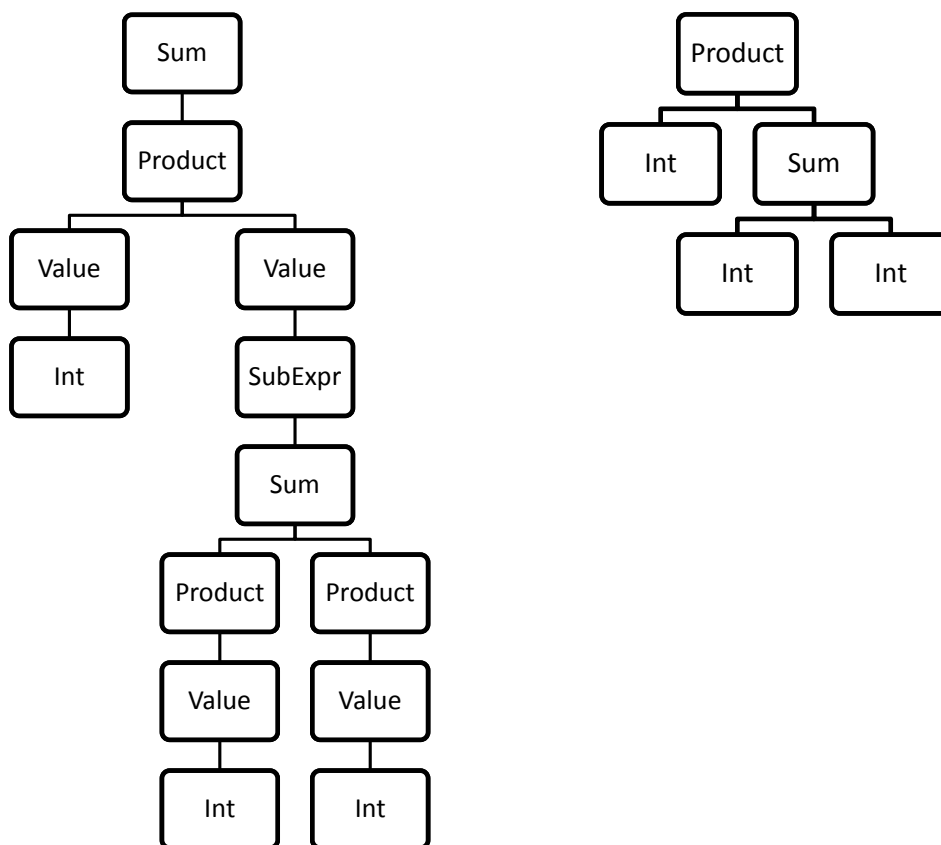


Figure 6.2.: AST for "3*(2+2)" on the left. The right AST is our desired Reduced AST for the same expression.

6.3.1. Pass-Through Rules

There is some syntax we need to match which does not have any semantic meaning. The rules that match such empty syntax should not contribute to the final AST. We refer to these rules as **pass-through** rules and denote them by using a different assignment operator '=':

```
Sum    <- Product ('+' Product)*;
Product <- Value  ('*' Value)*;
Int    <- [0-9];
Value  = SubExpr / Int;
SubExpr = '(' Sum ')';
```

Value is simply a choice between two expressions and does not have any semantic value. Its primary role is to avoid redundancy in the *Product* rule. Having a *SubExpr* node does not contribute any information to this particular AST and can also be a pass-through rule.

The nodes that are created as a result of executing the expression of a pass-through rule are added to the node created from the rule that refers to the pass-through rule, meaning the *Product* rule will have *Sum* and *Int* children instead of *Value* children. The resulting AST is shown in Figure 6.3.

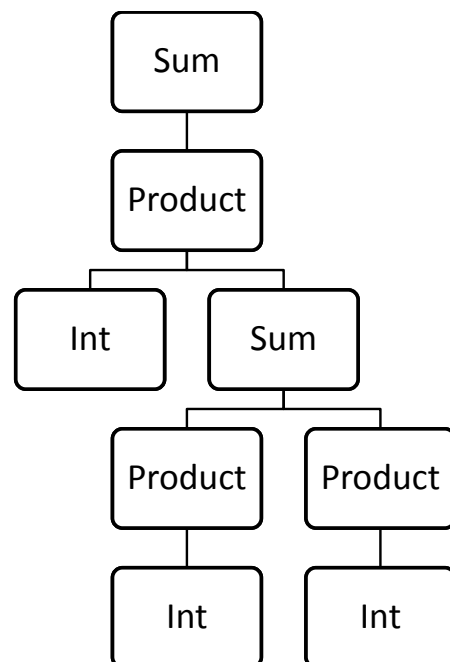


Figure 6.3.: AST for "3*(2+2)" after introduction of pass-through rules.

6.3.2. Tree-Types

To reduce the AST even further, we introduce the concept of *Tree-Types*. In an AST there are often groups of nodes that are intimately related. Examples of such groups are expressions, statements and declarations. In our running example we have defined the grammar of an expression, one that we want to be able to evaluate once we have our AST. We want the AST nodes relevant to that evaluation and only those.

To group nodes, we use tree-types. By declaring a rule as having the type Expression we are able to distinguish it from others and group it with other Expressions.

```
tree ExpressionTree;

Sum      : ExpressionTree <- Product ('+' Product)*;
Product  : ExpressionTree <- Value ('*' Value)*;
Int      : ExpressionTree <- [0-9];
Value    = SubExpr / Int;
SubExpr  = '(' Sum ')';
```

This alone does not reduce our tree any further, but it allows us to add a specific constraint on when a node should be created. In our latest AST seen in 6.3 on the facing page, the nodes we are not interested in are the binary operators that have only one child, specifically only one child of type Expression. We therefore introduce the **cullable** notation "?:" in the place of ":" in the type definition to denote that the Expression node should not be created if it has only a single Expression child. With this addition the final grammar becomes:

Listing 6.1: A grammar using cullable tree-types.

```
tree ExpressionTree;

Sum      ? : ExpressionTree <- Product ('+' Product)*;
Product  ? : ExpressionTree <- Value ('*' Value)*;
Int      : ExpressionTree <- [0-9];
Value    = SubExpr / Int;
SubExpr  = '(' Sum ')';
```

And the result of parsing "3*(2+2)" matches our desired AST as seen on the right in Figure 6.2 on page 53.

6.4. Syntax Tree Representation

Now we know which nodes we want the parser to create, we can start examining how they should be represented. The representation of the syntax tree and its nodes directly affects the speed of construction, ease of use and memory usage of the AST.

As we are targeting C#, we will examine how we should design the object relations between nodes of the AST as well as the class hierarchy for representing nodes.

6.4.1. Node Representation

First, for every node we need to know which rule created the node. For this we have three basic options:

ID Value: We can use a unique identifier for every rule, such as a string, and annotate every node using the ID for the rule that created it.

Object Reference: If rules are represented using objects, we can add a reference to the rule on each node.

Typed Nodes: Finally, we can have a unique type of AST node for every rule, in the form of a class.

Using a string to represent the rule types has the advantage that we can easily add a new type of node, even at runtime. A disadvantage is that it would be easy to corrupt such a tree by creating nodes that have invalid identifiers.

Using object references, we can be certain that the node was created by a rule and is valid and thereby that the nodes themselves are valid. However, we cannot get any compile time verification of the validity of the tree as a whole. Because, we do not have the support of the type system we can easily corrupt the tree by mistake.

Therefore, we find that giving each rule its own type of node to create is the safest and most suitable design.

6.4.2. Representing Parent-Child Relations

We find that there are two basic approaches we can use to represent parent-child relations. One is to use either arrays or a lists to represent the children of a node. The other is to have the relations between nodes be represented on the nodes themselves, by giving nodes references to their siblings in a linked-list type of approach. This way a reference to an AST node is a reference to a list of AST nodes.

The advantages and disadvantages are the same as between using a linked-list or an array list, with one exception. With regular dynamic linked-lists, the representation of the list is disjoint from the content of the list meaning that to add a new item to the lists means allocating memory for the list node. With AST nodes this is not the case, we already allocated that memory for the node. In C# or Java if we use arrays or lists for children, we would have to allocate the array separately after allocating the AST node.

With the amount of nodes that are allocated during the construction of the AST, doubling the amount of heap memory allocations needed is a big deal (see the performance chapter 9 on page 81) and should be avoided.

Therefore, we find that representing lists of AST nodes, such as a list of children to an AST node, should be done using a linked-list sibling approach.

6.4.2.1. Inferring Typed Children

Given that each rule creates nodes of unique types, we can infer the types of generated nodes' children, how many of each it can have and implement constraints to ensure the tree sanity. Having type-strong references to child nodes also makes traversing the tree much easier.

Given the following rule:

```
Method <- Modifier? Type Identifier '(' (Parameter (',' Parameter)*)? ') ' Body
```

We can see that if `Method` is parsed successfully, the new node will have the following child nodes:

- Zero-or-one **Modifier**
- One **Type**
- One **Identifier**
- Zero-or-more **Parameters**
- One **Body**

This assumes that none of the non-terminals refer to pass-through or tree-typed rules. By examining the non-terminals, repetitions and choices of a parsing expression, we can explicitly determine the minimum and maximum amount of each type of node.

If a non-terminal is a pass-through rule we can treat its parsing expression as if it was inlined in the referring rule and apply the same rules. If the non-terminal is a pass-through rule and is directly or indirectly recursive using other pass-through rules, the maximum of each type of node the pass-through rule can generate is the same as with repetitions of one-or-more.

The result we want is easy and type safe access to the children of the node. The following class node signature is similar to what we want to achieve:

```
class Method : AST
{
    public Modifier Modifier { get; } // returns null if there is no modifier
    public Type Type { get; }
    public Identifier Identifier { get; }
    public IEnumerable<Parameter> ParameterChildren { get; }
    public Body Body { get; }
}
```

This would allow us to access members of nodes as such:

```
Class classNode = GetClassNode();
foreach( var method in classNode.MethodChildren )
{
    foreach( var parameter in method.ParameterChildren )
    {
        // Do something.
    }
}
```

6.4.2.2. Accounting for Tree Manipulation

By giving nodes a reference to their parent, sibling and first child, we can completely represent an AST, with the possibility to traverse up as well as down. However, the list of children a node can have would not be completely safe. We can determine which types of children a node can have after parsing has completed, but we may want to allow the user of the AST to manipulate the tree by adding and removing nodes.

If we represent the children of an AST node as a list of AST nodes, modifying the AST means adding and removing children in these lists. If the lists are not type safe, it would be possible to add a child to an AST node that during parsing could not end up with such a child. We can use typed child inference to provide shorthand properties that return all nodes of a specific type in the child list. Furthermore, we can hide the functionality of adding and removing AST children and provide type safe add and remove methods on the AST node types. An advantage of using only a single list of AST nodes is that we can implement tree search functionality on the AST class easily and thus avoid cluttering generated nodes with such logic.

Alternatively, we can populate the AST node classes with typed and amount constrained lists for every type of child node. As lists of nodes are represented by a single node reference it does not require more memory allocations only more memory space. This leads to smaller lists that only contain the nodes that the typed accessor properties refer to, which could potentially mean faster access and search in the AST. The implementation would be more complicated however, as care needs to be taken to update parent relations, also searching algorithms need to be implemented on each AST class to search through all of a nodes child lists. Furthermore, initial population of the lists also needs to be implemented on the AST classes.

Finally, there is the question if it is allowed for a node to be represented twice in an AST. If we discard parent relations it is possible for an AST node to be represented in several child lists. The advantage is that the implementation of add and remove is much simpler and that code duplication is nearly free. A big disadvantage is that the tree becomes a graph as it would be possible to have cyclic references.

6.4.3. Representing Tree-Types

Tree-types exist to represent related rules, which means to represent related types. Also, because of the **cullable** construct any reference to a cullable rule could be a reference to another node of the same tree-type.

This corresponds well to the features provided by type polymorphism, as such we can use abstract classes or interfaces to represent tree-types. If a rule has a tree-type the node it generates will implement the interface or abstract class representing that tree-type.

When inferring the types of children an AST class has, in the case that they are tree-types the reference should use the tree-type class instead of the exact class.

Another consideration we can make regarding tree-types and sub-trees is whether or not we want the representation of the relations between sub-tree nodes to be separate from the rest of the AST. Because the nodes are related and a sub-tree node such as a binary expression node needs to refer to its sub-tree children, we may want to have a shortcut reference to those nodes. The problem is summarized in the following example:

```
tree Tree;  
  
Root : Tree <- 'a' (SubRule <- 'b' Leaf)*;  
Leaf : Tree <- 'c';
```

Should Root nodes have a direct reference to Leaf nodes or should we do a search in SubRule children nodes for Leaf nodes when we need them. The problem with searching is that there may be instances where you would never find one and could potentially risk searching through a large unrelated part of the AST.

The problem with having separate references between sub-tree nodes, is that those references would need to be created during AST construction, and subsequently maintained if the AST is modified after construction. The benefit is that any traversing algorithm would need only to traverse the actual sub-tree.

6.5. Decorating the AST

In the contextual analysis phase of a compiler it should be possible to decorate an AST with custom data. The simplest way of representing these decorations is as fields on the AST classes, however if the developer changes the AST classes after generation, those changes will be lost if the classes are generated again.

Using C# we can remedy this problem by generating `partial` classes. When a class is partial it is possible to extend them in other source files. This way a compiler developer can extend a generated class with new fields and methods without risking the changes being overwritten. Using partial classes also allows data-centered operations on tree-types, by adding an abstract method to the partial abstract tree-type class and implementing said method on all AST classes of that tree-type.

Another approach to decorating an AST is to simply have a hash-table keyed by the object reference of the ASTs, such an approach is better if the decoration is temporary.

6.6. Error Handling

Every time an operator fails to parse a piece of the input, we would like to log that error to report to the user. Instead of having error logging be a separate system or tree, we can have errors be a part of the AST during construction. An advantage of this approach is that we preserve the structure of the choices taken causing the error and can print a rule trace for every error.

With the rule AST classes having several references to different types of children, a memory optimization to do for errors is to have a separate class for error nodes. This separate class would have a reference to its child error nodes and be annotated with the rule that failed to parse.

Having errors be a part of the AST also allows us to use the error nodes to do instrumentation on the ordering of choices. As a result of the grammar, the ordering of options in a choice may be suboptimal for parsing a given type of input. For example, given the following grammar:

```
Product <- Int (('%' / '/' / '*') Int)*;  
Int <- [0-9]+;
```

If it turns out that the typical input for this grammar contains mostly '*' products, it would be advantageous for the grammar to be rearranged to attempt that operator first. By logging all the errors, even if the input is parsed correctly, and preserving the structure of the error we can identify such excessive backtracking by noting the amount of errors that specific operator caused.

Parsing Engine

This chapter will detail the implementation of our C# packrat parsing engine. The parsing engine is used by OOPEG and the the parsers it generates. The parsing engine has the syntax tree construction features described in the previous chapter 6 on page 49.

7.1. Parsing Expressions and Operators

A parsing expression is composed of operators. Operators such as *literals*, *character ranges* and *non-terminals* are leaf operators and the operators *sequence*, *choice*, *repetition* and *predicates* are aggregate operators as their result is based on the results of their child operators.

From an object oriented point of view each operator can be represented as an object and aggregate operators as containers of operator objects. Each object can be populated with the information it needs to perform the logic it represents - such as the string a *literal* should match. Which means that given any operator, we should be able to parse some input using the expression it represents. This turns the operators into the actual parser.

The recursively composable nature of the operators calls for the object-oriented *Composite* design pattern[10]. An aggregate operator have references to other operators which can also be aggregate operators. Which means we define a base class for all operators that allows us to begin parsing some input given any operator:

```
public abstract class GrammarOperator
{
    public abstract Result Parse(Context context);
}
```

The **input** is the string it needs to parse. The `Context` class is a wrapper on this string along with the current position in the input, the memoization cache and some additional info we will detail as it becomes relevant.

The **output** of the parsing is whether or not it succeeded in parsing and the result of parsing. The result of parsing in our case means the AST nodes that were created by the operator and its children. This data is encapsulated in the `Result` value:

```
public struct Result
{
    public bool Success;
    public AST ASTList, ASTListEnd;
}
```

Because many results will be generated during parsing, keeping the size of the result value as low as possible is essential to improve performance.

It is worth noting now that the `Context` is a reference type, and because of memoization we made it mutable. This means we could put all results of parsing on the `Context` and have it change state during parsing instead of returning `Result` values. However, in doing so we risk complicating the logic for operators - especially for composing and returning AST nodes - because of backtracking in the parser. If such an approach is used we need to be able to save the state of the current AST building process at some points and return to that state. For example, an assertion operator should not generate AST nodes, so it needs to be able to remove all nodes that were generated in its sub-expression. With the `Result` approach, this is a simple matter of returning a new `Result` value and without any AST nodes in the list. If the construction is on `Context` the assertion operator needs to save the AST state somehow and revert after the completion of parsing the sub-expression. It can be accomplished by having a list of AST nodes and saving the length of the list before parsing the sub-expression and after parsing removing all nodes that were added. But the logic needs to be implemented in all affected operators.

This design allows us to define the standard PEG operators as well as easily add specialized operators. Following is the *literal* operator as an example:

```
public class Literal : GrammarOperator
{
    public string String { get; set; }

    public override Result Parse(Context context)
    {
        if (String.Length > context.Available) {
            // CreateError creates an ErrorAST node and puts it in a failed result.
            return context.CreateError(this);
        }

        // Compare string to the input character by character.
        for (int i = 0; i < String.Length; i++)
            if (String[i] != context.input[i + context.index])
                return context.CreateError(this);

        context.index += String.Length;
        return new Result(true);
    }
}
```

Aggregate operators such as sequence and choice can use the variable parameter argument feature of C# to allow for creation of complex parsing expressions using a single C# expression:

```
public class Sequence : GrammarOperator
{
    public Sequence(params GrammarRule[] rules) { this.Rules = rules; }
    public GrammarRule[] Rules { get; private set; }
    public override Result Parse(Context context) { /* ... */ }
}

public static void Main(string[] args)
{
    // ('a' / 'b') 'c'
    var pegExpression =
        new Sequence(
            new Choice(
                new Literal("a"),
                new Literal("b")
            ),
            new Literal("c")
        );

    Result result = pegExpression.Parse( new Context("abc") );
    /* Do something with result */
}
```

Aggregate operators can concatenate the results of child expressions to build lists of AST nodes. For example the *sequence* operator would call `Parse` on each of its child operators and then concatenate all the returned AST node lists into a new list which it will return with the `Result` value.

7.2. Rules and Non-terminals

Rules are named parsing expressions that are referable from other expressions using non-terminals. For simplicity, consistency and because we have inline-rules, rules are `GrammarOperators`.

7.2.1. Creating AST Nodes

It is the job of rules to create AST nodes. After having successfully executed its parsing expression an AST node is created and the AST list from the `Result` is added as children on the new AST node. Because we want rules to create AST nodes of different types and have the logic for parsing a rule in the `Rule` class, we need to be able to tell the `Rule` class which node class to instantiate.

Instantiation using Type-Parameter Constraints We can instantiate using the C# type parameter constraint 'new()', this allows us to invoke the empty constructor of the type parameter:

```
public class Rule<T> : GrammarOperator
    where T : AST, new()
{
    protected AST CreateNode()
    {
        return new T();
    }
    /* ... */
}
```

However, during an optimization phase it was found that `new T()` uses type reflection and dynamic constructor invocation. This kind of reflective execution is notoriously slow in the .NET framework and accounted for more than half of spent CPU cycles according to our profiling tools, see chapter 9 on page 81 for more.

Instantiation using Inheritance We can also use sub-typing and inheritance, by declaring the `Rule` class abstract and having an explicit rule class for every AST node. This means we can have an abstract `CreateNode` method which instantiates the correct type of node:

```
public class A : Rule
{
    /* ... */
    public override AST CreateNode()
    {
        return new ANode();
    }
}
```

This approach uses virtual functions to solve the problem and is much faster and still leaves the implementation for parsing a rule in the common abstract class.

Instantiation using Delegates A third approach is to use delegates and simply pass a function to the constructor of a `Rule` whose task it was to create a node:

```
delegate AST NodeCreationHandler();

public class Rule
{
    public Rule( NodeCreationHandler nodeCreator ) { /* ... */ }
    protected NodeCreationHandler nodeCreator;
    protected AST CreateNode()
    {
        return nodeCreator != null ? nodeCreator() : null;
    }
}

// Instantiation using lambda.
var rule = new Rule( () => new A() );
```

By using delegates we can avoid having extra explicit classes for our rules, it also opens the `Rule` class up for interesting future usage, such as creating new rules at runtime that generate nodes that are merely named as opposed to explicitly typed.

Instantiation using Runtime Code-Generation It is possible to generate new code at runtime and get the simplicity of the type parameterized `Rule` while getting the performance of the other solutions. The easiest way to do this is to use *Expression Trees* from the .NET framework version 3.5 and later. We can define functions using expression trees and then compile and execute them at runtime:

```
public static Rule Create<T>() where T : AST, new()
{
    // Get the empty constructor of the type.
    var constructor = typeof(T).GetConstructor( Type.EmptyTypes );

    // Create the Expression tree ( A single lambda expression with an instantiation in it )
    var exTree = Expression.Lambda<NodeCreationHandler>( Expression.New( constructor ) );

    return new Rule( exTree.Compile() ); // where Rule takes a NodeCreationHandler
}
```

An older, complicated and more powerful approach is to use `Reflection.Emit` to generate intermediate language code on the fly.

7.2.2. Non-terminals

There are a few ways we can implement non-terminals. We can associate each non-terminal with an identifier, such as a string or integer value, and use it to lookup the rules in a hash table at parsing time. However, this is not as safe as we would like it, the identifier could for example refer to a non existing rule.

Because our rules are `GrammarOperators` we can choose to have a direct object reference in the expression instead of using non-terminals:

```
// A <- 'a'
var a = new Rule( () => new A(), new Literal("a" ) );

// E <- 'b' A
var e = new Rule( () => new E(),
    new Sequence(
        new Literal("b"),
        a
    )
);
```

However this approach does not allow for cyclic references, in the example `A` cannot refer to `E` as it has not been defined yet.

In order to allow cyclic references, we need to declare rules before we assign them parsing expressions:

```
var A = new Rule( () => new A() );
var E = new Rule( () => new E() );

A.Expression = new Sequence(new Literal("a"), E );
E.Expression = new Sequence(new Literal("c"), A );
```

Having to explicitly instantiate rules before they are used is unwieldy when the amount of rules increases, it also requires us to have a single point of declaration of all rules and their relations; an approach that is not very modular.

Our solution to this is to borrow from the *Singleton* pattern. Using a *Singleton* pattern we can simplify creating expressions with non-terminals by hiding the requirement to declare the rule before assigning it an expression. We can even go a step further and put the *Rule Singleton* on the class that represents the AST node it creates:

```
public class E : AST
{
    private static Rule _grammar;
    public static Rule Grammar
    {
        get
        {
            if(_grammar == null)
            {
                _grammar = new Rule( () => new E() );
                _grammar.Expression = new Sequence( new Literal("c"), A.Grammar );
            }
            return _grammar;
        }
    }
}

public class A : AST
{
    private static Rule _grammar;
    public static Rule Grammar
    {
        get
        {
            if(_grammar == null)
            {
                _grammar = new Rule( () => new A() );
                _grammar.Expression = new Sequence( new Literal("a"), E.Grammar );
            }
            return _grammar;
        }
    }
}
```

In the above example there is a cycle between A and E, however if A.Instance is accessed first it will create an instance of itself before it calls E.Instance which means that when E calls A.Instance its _instance is not null and the instance is returned. This solves the declare first problem and also increases the cohesiveness of the code.

It is worth noting that without synchronization of the instantiation of the rules, this approach is not thread-safe.

7.2.3. Parsing a Rule

For simplicity, we begin by excluding details for memoization, *pass-through* rules and *tree-types* when describing the parsing and AST creation procedure.

Listing 7.1: Parse method for rule without memoization, pass-through or tree-types.

```
Result Parse(Context context)
{
    Result subResult = Expression.Parse(context);

    var node;
    if(subResult.Success)
        node = CreateNode();
    else
        node = context.CreateError(this);

    node.SetChildren( subResult.ASTList );
    return new Result( subResult.Success, node );
}
```

The first step for a rule is to call `Parse` on its sub-expression to get the `Result` value. If the sub-expression parsed successfully we create a typed AST node otherwise we create an error node. The newly created node has the nodes in the result attached as children. Finally, we create and return a new `Result` value with our newly create node.

To create the typed AST nodes we call the `CreateNode` function on the `Rule` class. If `CreateNode` returns `null` we interpret this as the rule being *pass-through* and proceed to simply return the result of parsing the sub-expression instead of creating a new `Result` value.

7.2.4. Memoization

As described in chapter 2 on page 17, we can use packrat parsing to achieve linear time performance. This means we need to be able to save the result of parsing a specific rule at a specific index of our input and use a lookup method to retrieve previously cached results using constant time complexity.

We need to be able to uniquely identify every rule for the memoization to work. The approach we use is to have the constructor of the base `Rule` class generate a execution time unique unsigned integer ID. By generating the ID at construction we avoid having to generate a hash for the rules every time we use them. This approach allows us to use an unsigned 64bit integer (C# `ulong`) to represent a key in our cache, where the high-order 32bit is the ID of the rule and the lower-order bits represent the index in the input. With the ID determined, our cache is simply a hash-map from an `ulong` to a cached result using the .NET base class library class `Dictionary<ulong, CacheResult>`.

At the beginning of the `Parse` method of `Rule` we generate the key, look it up in the dictionary and if a result is cached we return the result. It is very important to remember to also advance the index on the `Context` if the result was a success. If a result was not previously cached on the key, we parse as usual, but remember to insert the result in the dictionary before returning.

7.3. Sub-trees using Tree-Types

As mentioned in the section 6.4.3 on page 58, we can choose to build sub-trees separately and achieve a performance boost when working with the trees after parsing.

To do this we add another list of nodes to the `Result` value:

```
public struct Result
{
    public bool Success;
    public AST ASTList, ASTListEnd;
    public SubTree SubTreeList, SubTreeListEnd;
}
```

When parsing a rule, if the AST node that was created has a tree-type we add the sub-tree nodes from the sub-expression `Result` list as sub-tree children of the newly created node. If the newly created node does not have a tree-type, we return the sub-tree node list unchanged with the result of parsing the rule.

For every tree-type we generate an abstract class that sub-tree nodes inherit from. The class inherits from the `SubTree` class which is our node type for sub-tree nodes that allows us to distinguish regular AST nodes from nodes that are part of sub-trees without knowing the exact tree-type of the node. The `SubTree` class also contains the references that allow us to build sub-trees separately.

7.3.0.1. Cullable Tree-Types

If a rule has a tree-type and is marked **cullable**, we check to see if the sub-tree list from the sub-expression result contains only a single node of the same tree-type. If that is the case, we refrain from creating a new node and simply return the sub-tree node.

This approach however causes problems with memoization. When retrieving successful results from the cache we advance the current index on the `Context` by the length of the node retrieved from the cache. But when storing sub-tree nodes we can not be certain that the length of the sub-tree node is the same as the length of a created node:

```
tree ExpressionTree;
Sum ? : ExpressionTree <- Value ('+' Value);
Value <- '(' Int ')';
Int : ExpressionTree <- [0-9]+;
```

Using this grammar to parse "(1)" starting from `Sum` would result in a sub-tree with only a single `Int` node in it (matching "1") and advance the index by 3. Suppose that because of backtracking we parse `Sum` again at the same index, it would retrieve the `Int` node from the cache and advance the by the length of the node which is 1, causing unpredictable behaviour. As a result of this we save the length along with the result in a `CachedResult` value in the cache.

A different approach would be to have the `Result` also contain the resulting index of running `Parse` on an operator instead of having it as a mutable member of the `Context` class.

7.4. Left-Recursive Rules

Some grammars benefit from being able to use left-recursive rules. But because PEG parsers are LL parsers, they do not support neither direct or indirect left recursion. It is possible to work around left recursion by converting it to the equivalent right-recursion or repetition. This does cause problems with the AST generated by such rules. Consider the rules for parsing binary sum expressions shown in listing 7.2.

Listing 7.2: Examples of recursion/repetition for binary expressions.

```
Integer <- [0-9]+;
RightRecursiveSum <- Integer '-' RightRecursiveSum / Integer;
TailSum <- Integer (Tail <- '-' Integer)*;
LeftRecursiveSum <- LeftRecursiveSum '-' Integer / Integer;
```

They are all capable of parsing the input string "3-2-1", but they will all create different parse trees. `RightRecursiveSum` will generate a right-associative parse tree interpreting the input string as $(3-(2-1))$. If the expression is computed in this way it will produce the wrong result. `TailSum` will generate a non-binary parse tree with each sum node containing a head integer and a list of tail nodes. While semantics that produce the right result for this type of tree can be written they are not as straight forward as those for a binary parse tree. The `LeftRecursiveSum` rule will produce a left-associative parse tree that interprets the expression as $((3-2)-1)$, which will produce the correct result.

Since left-associative binary expressions are present in most programming languages, having a construct that supports them directly makes sense. While simple PEG parsers do not support left recursion, packrat parsers do. In [24] it is shown how a packrat parser can be modified to support left recursion. OOPEG has been modified to support left recursion using this technique.

For OOPEG we have added a new `GrammarOperator`, `LeftRecursiveRule`, that inherits from `Rule`. The main trick taken from [24] is that before a left-recursive rule is parsed we enter an error state into the cache at the key of the left-recursive rule. When the rule then recurses it will grab the error value from the cache and fail the recur-

sive choice. The result of parsing is then put back in the cache and parsing is attempted again. This is repeated until an attempt either fails or does not advance the index further than a previous attempt.

When one left recursion rule is already performing this greedy expansion at an index, all other left-recursive rules called on that index are parsed without using the cache at all. This along with the greedy expansion, that performs parsing on the same index with the same rule more than once, are the main contributors to this solution not having linear time performance.

Parser Generator

With the parsing engine properly defined and implemented we can begin generating parsers from grammar specifications.

The parser generator will generate the first step of a compiler in the form of a complete **syntactic analysis** phase for a language. It will also generate a few tools to ease development of the *contextual analysis* and *code generation* phases.

This chapter will describe the parser generator in detail including: the input language, semantic analysis, contextual analysis and code generation.

8.1. Grammar Specification Language

As mentioned in chapter 4 on page 37 we want the definition of PEGs for our parser to resemble the original PEG specification as defined in [7]. The complete grammar for the input file can be seen in the appendix. There are three areas where we differ from regular PEGs:

Additional Operators A few new parsing expression operators were added to either improve performance of parsing or ease the definition of grammars.

Naming Scope Rules To avoid name clashings, ease code generation and add modularity we have added namespaces and scope rules.

Rule Annotations To support our tree construction features, rules have a few different assignment operators and tree-type annotations.

Tree-Types Tree-types are explicitly defined in our grammars.

8.1.1. Additional Operators

Inline Rules As described in chapter 6 on page 49, we have added the inline rule definition:

```
// B is an inline rule.  
A <- 'a' (B <- 'b');
```

This operator has the same parsing semantics as using a non-terminal. The difference is in name scoping only, as B can not be called by other rules.

Literal List In our previous study of PEGs we found that there are some situations where a grammar calls for a choice between a list of literals, for example choosing between keywords:

```
Keyword <- 'private' / 'public' / 'protected' / 'int' / 'internal' / 'abstract' / 'void';
```

Using a choice as above is problematic. Each string is matched from start to finish, which can be avoided by using a dictionary like structure. Also, as in the above case, there is the potential for prefix capture errors, 'internal' will never be matched as 'int' will match first and end the choice.

Using a literal list operator, we can generate an efficient lookup data structure for the literals and also avoid prefix matching by always matching as many characters as possible:

```
Keyword <- { 'private', 'public', 'protected', 'int', 'internal', 'abstract', 'void' };
```

Spaces Shorthand '_' Matching zero or more whitespaces is a common occurrence in PEGs. To make this less intrusive in the definition we have introduced a shorthand with the underscore character: '_'

```
Assign <- '=' _;
```

Commit '\$' The '\$' character is used to explicitly specify a commit point. A commit point is analogous to Prolog's cut operator '!' in that it prevents backtracking. All choices taken up to the index where the operator is encountered will not backtrack. Furthermore, because the parser can never return from its current index, we can empty the memoization cache up to the current index.

As with the cut operator from Prolog, this operator should be used sparingly as it can lead to unpredictable behavior. However, it can cut down on memory usage during parsing quite significantly. A successful scenario where we used the operator was when parsing declarations in C#. Because of the keywords used in declarations we could be sure that backtracking was unnecessary after successfully parsing a declaration.

Expanding on the idea of using commit statements is mentioned in the future work section 10.3 on page 86.

8.1.2. Naming Scope Rules

Being able to group rules into hierarchical namespaces reduces name clashing and makes it easier to define and read related rules. Because our parser generator generates for C# we decided to simply copy its namespaces and type scope rules. For every rule we are going to be generating a C# class and using the same type scope rules makes

it easier to generate the code and to understand for the users. The following grammar will generate a single class `Start` in the `Calculator` namespace:

```
namespace Calculator
{
    Start <- 'start';
}
```

Referring to rules in other namespaces follows same convention and rules as in C#. For our initial version we have not added the `using` statement from C#, which means referring to a rule in a different namespaces is done using fully qualified names:

```
namespace Literals
{
    Number <- [0-9]+;
}
namespace Calculator
{
    Start <- Literals.Number '+' Literals.Number;
}
```

8.1.3. Defining Tree-Types

Because each tree-type is also represented in the generated code by a C# class we explicitly define them in the specification. The following defines a tree-type `ExpressionTree` in the `Calculator` namespace:

```
namespace Calculator
{
    tree ExpressionTree;

    Int : ExpressionTree <- [0-9]+;
}
```

We could infer which tree classes to create directly from their usage and not explicitly define them. However, in the spirit of C# and type-strong languages we decided upon the explicit specification of each tree. It would be easy for users to misspell a tree-type annotation and we want to catch such errors at compilation.

8.1.4. Defining Rules

Contrary to the original specification for PEGs, we have different ways of defining rules that denotes how AST creation is handled for a rule. The syntax we use was shown in chapter 6 on page 49. Referring to tree-types follow the same scope rules as referring to other rules.

Beyond pass-through, tree-typed and cullable rules, we also added macros. A macro uses the `'=='` assignment operator and are parsing expressions that are expanded all the places they are referenced at parser generation time. This means that cyclic macros are not allowed.

8.2. Syntactic Analysis

Syntactic analysis is provided by the parser generator itself. The parser generator is bootstrapped and uses the generated **syntactic analysis** to parse its input language.

An example of invoking the generated parser and getting a reference to the AST follows:

```
var context = Context.FromFile( "input.peg" );

// Start is a rule in the input language.
Start root;

if (!Start.Grammar.TryParse(context, out root))
{
    // Parsing Failed.
}
else
{
    // Parsing Succeeded. Do something with "root".
}
```

Bootstrapping was accomplished by first implementing the syntactic analysis in hand using the parsing engine. Then once the parser generator was able to generate parsers, we generate the parser for itself and replace the handwritten version.

The most important concepts of the output of our grammar language are the two main tree-types: `ADeclaration` and `AExpression`. Declarations of tree-types, rules and namespaces are `ADeclaration` nodes and all parsing expression operators such as literals, choice, sequence, predicates and non-terminals are `AExpressions`.

For each tree-type we generate visitor interfaces and classes, these are further explained later in the code generation section 8.4 on page 76 in this chapter.

8.3. Contextual Analysis

The contextual analysis verifies the semantics of the input language. We need to check the integrity of the grammar from our input.

8.3.1. Declaration Pass

Specifications of PEGs allow for forward references to other rules, which means we do not need or want to forward declare rules that are being used by non-terminals.

This means we need to do a declaration pass in which we register all namespaces, rules and trees and put them in a symbol table by their fully qualified name, so that we later can refer to them by using our naming scope rules.

Due to child inference and pass-through rules, the top node of our AST - `Start` - has a list of `ADeclaration` children which are all the top-level declarations in our input. We can simply use a visitor to visit all the sub-tree nodes and register all the declarations we visit.

8.3.2. Name Resolving

Once we have our symbol table, we can resolve all names in tree-type annotations and non-terminals. We need to throw errors if there are references to non-existing rules and we decorate each non-terminal node with a reference to the rule they represent.

For this we use an `ADeclaration` visitor that keeps track of which symbols are available in each namespace at each rule. For each rule, we resolve the tree-type if it has any and then use an `AExpression` visitor to traverse the expression on the rule and resolve all non-terminals.

8.3.3. Left Recursion

Each rule must be checked for direct and indirect left recursion. If a rule is left-recursive we mark it as such, so that we can generate a left recursion `GrammarOperator` for the rule.

Because we have already resolved non-terminals we can use an `AExpression` visitor that can jump from non-terminals to their referred rule. By tracking which rules we have already visited we can find left recursions. The visitor has a custom traversing algorithm, which stops traversing a sequence if it encounters a sub-expression that consumes input.

8.3.4. Inferring Typed Child Relations

Finally, we infer the types of children each rule generated node can have. We can do this using a visitor on the parsing expressions. See previous section on inferring children 6.4.2 on page 56 for more info.

Once the child relations have been found they are saved on the rule AST nodes.

8.4. Code Generation

With our completely decorated tree we are ready to begin generating code. As we are generating C# source code we have a few options.

One is to simply use a string or `StringBuilder` to represent code we are generating and simply append code directly to the string during generation. This approach is very simple and easy to implement, it does however lack a few features such as handling indentation easily. Also, with this implementation, it can be very difficult to discern from the source code what the output of the generation will look like.

Another approach also used by SableCC is to use a templating system. Such templating systems are widely applied in the web application development field, where it is a big advantage to be able to separate the application logic from the design of the interface. With templates, the focus is on what to output instead of how to output it. Usually a model-view-controller type pattern is used, where the model is the data we need to display, the view is the template and the controller is a link between the two that only exposes the properly formatted parts of the model that the view needs.

An advantage of the templating approach is also that we can easily replace the view component to output something different, for example changing from generating C# to generating Java code.

We have used both approaches in the parser generator. For generating the parsing rules we use a string approach, because each operator in the rule is barely a line of code. For generating AST and tree-type classes we use the templating system *Text Template Transformation Toolkit* (T4), which integrates seamlessly in Visual Studio 2010 and was originally designed to generate source code[18]. The classes we generate have much code in common with a few key identifiers difference and are perfect candidates for templating systems.

8.4.1. Tree-Types and Visitors

For tree-types we need to generate an abstract class that enables the use of visitors on the sub-types. This means adding abstract `Accept` methods.

In addition to generating visitor interfaces for tree-types, we also generate evaluator interfaces. Evaluators are visitors whose `visit` method returns a value and are used for expressions that can be evaluated to a single value.

Both visitors and evaluators come in type parameterized versions that allow for type safe return values for evaluators and for parameters to be sent to visit methods. Each of the type parameterized variations call for a compatible `Accept` method on the tree-type class and subsequently on the AST classes as seen in listing 8.1 on the facing page.

Listing 8.1: Example classes and visitors for tree-type Expr with only one rule A

```

public class Expr : SubTree
{
    public abstract void Accept(ExprVisitor visitor);
    public abstract void Accept<T>(ExprVisitor visitor, T tParam);
    public abstract T Accept(ExprEvaluator evaluator);
}
public class A : Expr
{
    public override void Accept(ExprVisitor visitor) { visitor.Visit(this); }
    public override void Accept<T>(ExprVisitor visitor, T t) { visitor.Visit(this, t); }
    public override T Accept(ExprEvaluator evaluator) { evaluator.Eval(this); }
}
public interface ExprEvaluator<T>
{
    public T Eval( A a );
}
public interface ExprVisitor
{
    public void Visit( A a );
}

```

Finally, we also generate a depth first tree traversal adapter. As mentioned in section 1.4 on page 14 these adapters makes it a little easier to traverse and visit all nodes in a tree:

```

public abstract class ExprDepthFirstAdapter : ExprVisitor
{
    public virtual void Visit( A node )
    {
        // Visit all subtree children
        Expr current = node.SubTreeChild as Expr;
        while(current != null)
        {
            current.Accept(this);
            current = current.SubTreeSibling as Expr;
        }
    }
}

```

8.4.2. AST Classes

For every rule we generate a class and on that class we have the static singleton property that contains the rule GrammarOperator. If a rule is pass-through, the class we generate is static such that it can never be instantiated.

We also generate a C# expression for the rule's parsing expression. To do this we use an AExpression visitor to visit all operators in the expression. For every operator we generate code to instantiate the operator and for aggregate operators we visit its children and their generated instantiations become arguments for the constructor for the aggregate operator.

In each of these classes that are not static, we also have to generate accessors and lists for our inferred typed children as described in 6.4.2 on page 56.

8.5. Future Improvements

We have identified some possible future improvements for the parser that we did not have the time to do.

8.5.1. Prefix Capture Check

As mentioned in chapter 2 on page 17 on PEGs, PEGs are susceptible to prefix capture errors that can be difficult to debug. We can detect simple instances of prefix capture such as `'+' / '++'`. However, because of predicates, completely verifying that there are no prefix captures is an undecidable problem. Determining if there is prefix capture in A/B corresponds to finding out if $(\&A)B$ can match anything which corresponds to finding the intersection between $L(A)$ and $L(B)$ which is undecidable. For more info on this problem see [21].

8.5.2. C# Using Directive

We have implemented the C# naming scope rules. However, we have yet to implement the **using** directive.

```
using Oopeg.Identifiers;
```

The using directive allows the use of types (or in our case rules) in other namespaces without using the fully qualified name. It corresponds to importing the types to the current naming scope.

The directive also allows the aliasing of a type in a scope:

```
using Identifier = Oopeg.Identifiers.ID;
```

In this example the symbol `Identifier` is an alias for `Oopeg.Identifiers.ID`. Without the using directive, any type of modularity will be very verbose.

8.5.3. Assembly References

C# code is able to refer to existing class libraries through assembly references. It should be possible to do this for OOPEG files as well. We can use reflection at compile-time to verify that we can get a reference to a `GrammarOperator` from a public static property or method on a class. We could even require that the actual property, method or field that should be used is annotated with an attribute. When generating code from OOPEG files we can load the referenced assemblies and do reflection and verify that all references to external rules can be resolved to a `GrammarOperator`.

Another great benefit from this approach is that you can implement custom `GrammarOperators` and refer to them by name. We could even extend the syntax for non-terminals to allow

for parameters for custom `GrammarOperators`, such as:

```
// A custom operator called with different parameters
Start <- CustomOperator<42, "some string", NonTerminal, A / B >;
```

One downside is that for us to be able to refer to `GrammarOperators` or rules that are defined in the same assembly as the code we are generating (e.g. in the same project) we need to implement a C# parser for the other files in the project, because we cannot simply use reflection.

8.5.4. Errors by Non-terminals

In the current implementation we generate error nodes for every operator that fails, which corresponds to the CST construction paradigm though only for errors. We believe it should be possible to reduce these nodes to only be generated for the non-terminals that fail. In doing so we remove a great deal of node instantiations. Instead of getting the specific operator that failed we can generate an error based on the expression of the rule. The simple example is for a rule:

```
Start <- 'a' / 'b';
```

If such a rule were to fail parsing, we can generate an error message from the expression without knowing the exact point where the rule failed. We can simply write "expected 'a' or 'b'".

Inferring such error messages from the expressions of rules would allow us to refrain from creating CST nodes for every error encountered. It may however be that it is more desirable to know exactly how and where the node failed.

Another approach would be not to generate error nodes for operators when running the parser initially, but in the event that parsing failed for the input, simply reparse the rules that failed while generating error nodes from operators.

8.5.5. Extended Modularity

The module system in OOPEG allows grammars to be defined in their own namespace, and also makes it possible to refer to rules from another grammar present in the same project. This module system could be extended to something like the system seen in `Rats!`, where it is possible not only to refer to other rules, but also to modify them in a way similar to aspect oriented programming. This makes it possible to extend the functionality of an already defined language, like it is done in the `Fortress` project[20]. Extending OOPEG with such a feature would make it an even stronger representative of PEG parsing as it underlines the benefit of being able to combine two grammars into one.

8.5.6. More Visual Studio Integration

The project and item templates we have created for integration with Visual Studio have the advantage of working without the user having to install any programs or plugins that require administrator privileges.

If we want more integration with Visual Studio we would need to create a Visual Studio extension. This would allow us to have syntax highlighting and other more advanced features, such as an AST browser data visualizer for inspecting an AST through the Visual Studio debugger. We could also envision an extension that adds breakpoint and inspector support for OOPEG parsers such that setting a breakpoint on a rule or operator would could pause execution and allow the inspection of the current parse stack and other interesting values.

8.5.7. Grammar Optimization Tools

We mentioned in section 6.6 on page 60 that the error nodes system we implemented can be used to provide the users with at tool for optimizing his grammar. Tools for visualizing the AST and how it parsed a specific input with profiling of which rules failed and how, would help the parser implementer to optimize and improve his parser.

We implemented a simple unfinished tree visualizer for a completely parsed AST using Windows forms and type reflection, so implementing such tools is definitely possible and would be a valuable addition to our parser generator.

Optimizing Performance

We want OOPEG to perform fast enough to be a viable tool for Progressive Media to use in their code conversion efforts. To accomplish this we have been profiling our code during development and identified the parts of the program that spend the most CPU cycles.

We used Visual Studio's CPU sampling performance profiler and our own implementation of a C# 3.0 language parser. We ran the parser on a few big C# source files of between 500Kb and 900Kb, and between 16.000 and 30.000 lines of code.

9.1. Instantiating Type-Safe Nodes

The first bottleneck we encountered was related to instantiating our tree nodes. As previously mentioned in this thesis we were using C#'s empty constructor type parameter constraint `new()`. This construct invokes the `System.Activator.CreateInstance` method of the base class library, which is a reflective method and is much slower than regular instantiations.

We did a comparison test on instantiating 1.000.000 objects using three different methods.

Generic Using the generic type parameter constraint `new()`.

Derived Deriving from an abstract factory class and implementing a method for instantiating a type specific node.

Delegate Using a delegate function to instantiate the node.

The Derived and Delegate implementations performed comparably by instantiating one million nodes in 0.06 seconds on an Intel Core 2 Duo 3.0 GHz. The Generic implementation spent 1.09 seconds to do the same task on the same machine.

9.2. Reducing Instantiation Count

Once the huge reflection bottleneck was removed, most CPU cycles were spent simply instantiating nodes. In the beginning of development we were generating the nodes for the entire CST. By generating nodes only for the AST we saved on instantiations quite significantly.

In earlier versions of OOPEG every node had a list to contain children. The node was instantiated and sent down through the parsing expression of the rule. Rules were responsible for putting the nodes they generate into the child list of the parent. This meant that we needed to instantiate a node even if we subsequently culled that node because of tree type rules.

Turning around the order of instantiation and having the result of parsing any expression contain a list of nodes, meant we could avoid instantiating nodes we were never going to use.

In the syntax tree construction chapter 6 on page 49 we made the claim that having a list or array as a field on a node for containing children would cause extra memory allocations and slow down the construction of the AST. We have run a simple test in which we instantiate 10.000.000 nodes using the sibling, array and list data representation to verify this claim. For every node we instantiate we also add a single child. This is mainly because the `List<T>` type refrains from initializing its internal representation until something is put into the list. The classes we instantiate are:

```
abstract class Node {}

class SiblingNode : Node
{
    public Node Sibling, FirstChild;
}

class ListNode : Node
{
    public List<Node> Children = new List<Node>();
}

class ArrayNode : Node
{
    public Node[] Children = new Node[1];
}
```

The results are as expected; instantiating a single node that has references to children and siblings is the fastest with ~ 2.75 seconds; using array nodes causes a single extra memory allocation for the array and takes a total of ~ 5.04 seconds; with the internal representation of `List<T>` being an array it will invariably be slower than using an array, and it is by taking ~ 6.9 seconds if we instantiated the list with an initial capacity of 1. If we instead use the default initial capacity of 10, it takes ~ 9.9 seconds for us to instantiate the nodes.

9.3. Grammar Optimization

The performance of a parser is invariably based on the grammar of the language it is parsing. During our profiling sessions we identified a few very slow constructs in our C# specification. The following rule for example:

```
RealLiteral
  <- [0-9]* '.' [0-9]+ ExponentPart? RealTypeSuffix?
    / [0-9]+ ExponentPart RealTypeSuffix?
    / [0-9]+ RealTypeSuffix;

ExponentPart <- [eE][+-]?[0-9]+;
RealTypeSuffix <- [fF] / [dD] / [Mm];
```

When the C# parser needs to parse an expression, one of the first choices it attempts is whether or not the expression is a literal. One of the first literals is this `RealLiteral`. The problem with this rule is that it has bad cache utilization. It will manually attempt to parse `[0-9]` three times for every single attempt at parsing a literal. This check is only for `RealLiterals` even though integer literals should also begin with a number. By simply putting the `[0-9]+` expression in a separate cacheable rule we can improve performance significantly.

```
RealLiteral
  <- Number? '.' Number ExponentPart? RealTypeSuffix?
    / Number ExponentPart RealTypeSuffix?
    / Number RealTypeSuffix;

ExponentPart <- [eE][+-]? Number;
RealTypeSuffix <- [fF] / [dD] / [Mm];
Number <- [0-9]+;
```

Now the actual number is only parsed once and subsequently fetched from the cache. Before introducing these sorts of optimizations to our C# literal parsers, our C# parser spent 95% of samples parsing these rules, after the optimization it fell to 5%.

Syntactic predicates can also be used to exclude entire branches of choices. If we know that for any rule in a choice to match, the input has to match a very simple predicate, such as for any number literal to be a match the next character should be a `[0-9]`. We can put this predicate before attempting to match the number literals and avoid trying all the different permutations of number literals.

9.4. Commit Operator

In chapter 8 on page 71 we introduced the commit operator `$`. Once we have optimized our instantiations and our grammar, the main bottleneck becomes simply caching and retrieving cached values. By introducing the commit operator which prevents backtracking of any sort from where the commit operator was matched, we can choose to simply clear our cache when we parse the operator. Clearing the cache means clearing our hash-table which reduces the amount of collisions for a while and thus improves performance, it can also help to keep the size of the cache low in general.

Discussion

During the implementation of OOPEG we made choices to narrow the scope of this thesis. We will now examine what could have been done differently and identify open problems that could be addressed in future work.

10.1. Multiple Target Languages

Some of the parser generator tools we have encountered, such as ANTLR, have multiple target languages, meaning that the tool can output parsers written in different languages. The choice of using C# for target language for OOPEG was based on the needs of Progressive Media, as well as our own experience with C#. Adding additional target languages to OOPEG could be done by implementing more of the templates used for code generation. A potential problem though is that the current parsers generated by OOPEG depend on our `Oopeg2` .NET library containing the parsing engine. This makes it very easy to generate parsers for .NET languages, but targeting other languages such as Java would require either the `Oopeg2` library to be re-implemented in Java or changing the code generation system to also output an implementation of the parsing engine. However, having every generated parser contain the core functionality of the parsing engine might lead to problems when using two parsers in the same project, hindering the possibility of merging two parsers into one.

10.2. Inline Semantic Actions

One of our major design decisions was to leave out the inline semantic actions present in many of the tools we have examined. We dislike inline semantic actions because they clutter the input grammar, however they might prove useful when creating small domain specific languages with very simple semantics, such as the calculator examples that were used throughout this thesis. While adding semantic actions as an option in OOPEG could be done it would introduce a lot of new problems, like a new code generation system for processing them as well as a way to address the issue of wrong order of execution explained in section 3.6.2 on page 35.

10.3. Future Research

Basing parsers on PEGs is a relatively new idea, which means that a limited amount of research has been done in the area. In the next section we will highlight some open problems we have encountered during our work on the OOPEG implementation.

10.3.1. Left Recursion Elimination

In OOPEG left recursion is handled by using the method defined in [24]. However, it could be interesting to examine whether it would be possible to implement an algorithm similar to the one used for removing left recursion in CFGs for LL parsers[3]. Applying the existing algorithm to PEGs is not possible since the algorithm depends on choices being unordered. If an algorithm can be found it would also be interesting to look into how it could be applied as part of a parser in a way such that the tree generated by the parser would still conform to the left recursion defined in the grammar.

10.3.2. Optimized Selection of Choice Alternatives

The backtracking of choices makes PEGs perform slower than traditional parsers. Even when using packrat parsing multiple different rules might still be tried for each position in the index. It is often the case that partial parsing of one alternative in a choice will exclude the possibility of the some of the remaining alternatives ever being parsed successfully. For example consider the choice shown in listing 10.1.

Listing 10.1: Example of mutually exclusive choice alternatives.

```
Start <-  
  'namespace' Identifier LEFTBRACKET NamespaceContent RIGHTBRACKET /  
  'class' ClassContent /  
  'interface' InterfaceContent;
```

If the first alternative fails to parse after having parsed the first literal - 'namespace' - successfully, trying to parse the remaining alternatives would not be necessary, as they both start with something other than 'namespace'. It would be interesting to examine if it is possible to automatically identify such choices in a grammar and then implement an early out mechanism for them in the parser code generated for the choice, as this could lead to a speed-up of the parsing.

Note that the example does not illustrate all the aspects of the problem. It might be the case that the partially parsed rule does not disable the possibility of parsing all of the other alternatives, or that the alternatives it disables depends on how far it got before its own parsing failed. An implementation should therefore make it possible for an earlier alternative to disable other alternatives depending on how far it got before it failed.

Like the prefix capture problem from section 8.5.1 on page 78, this problem may

be undecidable. However, it would still be worth making the optimization in the less complex cases where it is possible to detect, such as the case where all the alternatives in a choice start with a terminal operator.

The *Rats!* parser has an implementation of the simplest case of this optimization, allowing choices between literals to be changed into switch statements[11]. It would be interesting to see if the optimization can be applied in a more general manner, perhaps by using something similar to the *rete algorithm*[8] used for pattern matching in functional programming.

10.3.3. Error Recovery

Some CFG-based parser generator tools provides error recovery in the parsers they generate, typically done by replacing, deleting or inserting tokens until the parsing can continue[3]. Having error recovery allows the parser to potentially identify multiple problems in the code which can then be fixed before parsing is attempted again. Being able to remove multiple errors for each parsing run saves development time and is therefore a valuable feature.

Since a PEG-based parser does not have a token stream this approach can not be used here. It would be interesting to examine which methods of error recovery could be applied in a PEG-based parser. One approach could be to allow top-level repetitions in the language to to retry parsing of the inner expression in the repetition whenever it fails. For example consider running the PEG shown in listing 10.2.

Listing 10.2: Example of repetition error recovery.

```
Start <- (Statement ';'')+;  
Statement <- 'a' / 'b' / 'c';
```

If a parser for this PEG was run with the input `'a;bc;'` the repetition in the start statement would fail to parse because of a missing semicolon after the b. However if the inner expression of the repetition was allowed to retry from the input position where the last attempt failed it could still parse the last `'c;'` statement in the input correctly. This approach is similar to deleting tokens in a CFG-based parser until something that is know occurs. It might be possible to look at other forms of error recovery from CFG-based parsers and implement similar systems for PEGs.

10.3.4. Optimizing Grammars For Better Cache Usage

As shown in the example in section 9.3 on page 83 the design of grammar can greatly affect the performance gained from the memoization. It would be interesting to examine if such grammar optimizations could be performed automatically.

This could possibly be done by identifying inline operators in the grammar that would

potentially be parsed multiple times for the same input position and automatically factor those operators out into a separate, cacheable rule, such as the `[0-9]+` character class from the example.

10.3.5. Automatic Commit Operator

The commit operator mentioned in section 9.4 on page 84 speeds up parsing by disallowing backtracking from the current input position and purging the cache used for memoization.

It would be interesting to examine whether placement of commit operators could be done automatically. For example, consider the PEG shown in 10.2 on the previous page. Whenever this PEG has successfully parsed the inner expression in the repetition in the start rule, it will not be possible for it to backtrack through the input that has been just parsed. Therefore an automatic commit could be done at the end of the inner statement, allowing the cache to be cleared for additional performance. This would only work for a top-level repetition as repetitions further down in the rule hierarchy might be subject to backtracking for a parent choice rule.

It is possible that the choice analysis needed for the optimized selection of alternatives - mentioned in a previous section - could be used to identify alternatives that would never succeed by backtracking and therefore would allow commits to be placed automatically deeper in the rule hierarchy.

Conclusion

After having examined some existing compiler-compiler tools, we found the most detrimental property of a compiler-compiler tool to be complicated and difficult to read input files. Most input grammars have dense syntax littered with semantic actions and labels. We also found that semantic actions are not very desirable in PEG-based compiler-compilers as implementing and understanding it is made difficult by the backtracking and parsing order.

Based on this knowledge we built OOPEG as a parser generator for generating the *syntactic analysis* for compilers, leaving the implementation of *contextual analysis* and *code generation* to be done in OOPEG's target language C#.

This approach restricts the syntax of our input files to only contain instructions for parsing input and building ASTs. We introduced a few new constructs for parsing to improve performance and to support the construction of simpler ASTs.

11.1. Building ASTs

We found that building CSTs is unfeasible as it spends too much memory and is difficult to work with after construction. So we create AST nodes for every rule in the PEG by default. Every AST node we generate has its own type and we infer the types and amounts of the children a node can have. This allows us to generate type-strong access and modification of ASTs to ensure sanity and ease development. To trim our AST we have added new syntax to our PEG definitions that concisely describe how nodes should be created.

Marking rules as **pass-through** using the '=' assignment operator cause rules to not create AST nodes, instead any nodes that are generated during the parsing of the rule's sub-expression are bubbled up to the caller of the *pass-through* rule. This allows us to mark purely syntactical rules as having no semantic value, and as such avoid having them in the generated AST.

Tree-types are used to group related rules. A tree-type corresponds to a base-class for all sub-tree nodes generated by tree-type rules. Using tree-types allows sub-trees to be constructed using the concise *Composite* design pattern. We can then auto-generate visitors and evaluators to allow easy operation-centered AST operations.

With tree-types we also introduced the **cullable** syntax, which cause a tree-typed rule to not create a node if its sub-expression created only a single node of said tree-type. This single concise rule was shown to be very powerful, as it could simplify expression trees significantly by not creating binary operator nodes that have no right operand.

Instead of having one very large visitor for all rules in a grammar we specifically generate a visitor for every tree-type. This means that a visitor for an expression tree only has to implement logic for expressions and not for every other rule in the grammar. It also means that we can provide an evaluator visitor class that visits nodes and returns values which is very valuable for expressions.

Adding only very little extra syntax to our input grammars we are able to generate trimmed polymorphic object-oriented ASTs and specialized visitors while preserving the readability of PEGs. using the partial class feature of the C# language we can extend AST classes without modifying the auto-generated code and thereby decorate the trees very easily and without compromising the ability to regenerate the parser.

11.2. Parsing Engine

The parsers we generate are based on our object-oriented modular parsing engine. We create parsing expressions using objects conforming to the *Composite* design pattern which allow us to create arbitrarily complex expressions using very few operators. It also allows us to implement new and custom parsing operators for improved performance or special needs.

One new special operator we added is the **Commit** operator which is based on Prologs *cut* operator. This operator prevents backtracking and can improve performance significantly in some cases. Because, it disallows backtracking we can clear the cache when we encounter the operator, which reduces cache collisions and memory usage during parsing. We found it very useful for C# member declarations.

By exploiting the *Singleton* design pattern, we can have rules refer to each other by object reference during their construction without resolving cyclic references. This allows us to put a single static property containing all parsing logic for a rule on the AST classes we generate, maximizing cohesiveness.

This modular approach to building parsers also allow us to compose a new parser from rules from existing grammars. We even envision how this can be done by referencing such rules in already compiled .NET assemblies.

The test parsers we constructed were shown to perform acceptably after some optimization. We were able to use our custom C# parser to parse approximately 15.000 lines pr. second. Compared to a SableCC C# implementation we found, ours took twice as long to parse the same input, however we have also identified many possibilities for further optimization of our parsing engine, the parsers we generate and our own C#

grammar. We find this to be very acceptable when applied in a production environment, especially if the parser is continually improved.

We identified areas of interest for future improvements of OOPEG and future research into Parsing Expression Grammars. Most interestingly is the study of optimizing choices to avoid backtracking and avoid attempting futile choices, which if even partially solved would make a tool such as OOPEG much stronger and faster.

11.3. Learning Curve and Integration

We found that existing compiler-compilers often lack proper documentation and are difficult to get started with. With this in mind we put effort into writing a simple tutorial and we made a simple installer for Visual Studio 2010 project and item templates to get developers started. Our tool integrates neatly with Visual Studio and offers one click parser generation and compilation from within the IDE for rapid development and test. Most PEGs and examples we have listed in this thesis can even be copied directly into an OOPEG grammar file, compiled and tested.

In conclusion, we have found that PEGs are very suitable for building parser generators as our grammars not only define the syntax of the language concisely, but also explicitly defines the parsing procedure and with very little markup can generate concise, safe and easily usable object-oriented ASTs.

- [1] Søren Grønnegaard Andersen. When grammars are not enough. Master's thesis, Aarhus University, Aarhus, Denmark, 2010. 19, 47
- [2] Tree construction - antlr 3 - antlr project. <http://www.antlr.org/wiki/display/ANTLR3/Tree+construction>. 27
- [3] Andrew W. Appel and Jens Palsberg. Modern Compiler Implementation in Java. Cambridge University Press, New York, NY, USA, 2003. 86, 87
- [4] Heiko Behrens. Xtext. <http://www.eclipse.org/Xtext/>. 32
- [5] S. Efftinge and M. Völter. oAW xText: A framework for textual DSLs. In Workshop on Modeling Symposium at Eclipse Summit, 2006. 32
- [6] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, pages 36–47. ACM, 2002. 20
- [7] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. ACM SIGPLAN Notices, 39(1):122, 2004. 11, 18, 19, 71
- [8] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem* 1. Artificial intelligence, 19(1):17–37, 1982. 87
- [9] Etienne M. Gagnon and Laurie J. Hendren. Sablecc, an object-oriented compiler framework. Technology of Object-Oriented Languages, International Conference on, 0:140, 1998. 28, 31
- [10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. ECOOP'93-Object-Oriented Programming, pages 406–431, 1993. 13, 61
- [11] Robert Grimm. Better extensibility through modular syntax. SIGPLAN Not., 41(6):38–51, 2006. 19, 34, 38, 87
- [12] "Samuel Imriská". "c# cup manual". "<http://www.seclab.tuwien.ac.at/projects/cuplex//cup.htm>". 25
- [13] Stephen C. Johnson. Yacc: Yet another compiler-compiler. Technical report, 1975. 25
- [14] Viswanathan Kodaganallur. Incorporating language processing into java applications: A javacc tutorial. IEEE Software, 21:70–77, 2004. 25
- [15] Jacob Korsgaard and Jørgen Ulrik Balslev Krag. A study on improving mobile game development using meta-programming and memory management techniques. <https://services.cs.aau.dk/public/tools/library/>

- files/rapbibfiles1/1262799521.pdf. 15, 17
- [16] M.E. Lesk and E. Schmidt. Lex - a lexical analyzer generator. Computing science technical report, 39:128, 1975. 25
- [17] Eric Lippert. How many passes? <http://blogs.msdn.com/b/ericlippert/archive/2010/02/04/how-many-passes.aspx>. 12
- [18] Microsoft. Code generation and text templates. <http://msdn.microsoft.com/en-us/library/bb126445.aspx>. 76
- [19] T.J. Parr and R.W. Quong. ANTLR: A predicated-LL (k) parser generator. Software: Practice and Experience, 25(7):789–810, 1995. 27
- [20] S. Ryu. Parsing fortress syntax. In Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, pages 76–84. ACM, 2009. 35, 79
- [21] Sylvain Schmitz. Modular syntax demands verification. Technical report, LABORATOIRE I3S, UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS, 2006. 78
- [22] G.L. Steele. Growing a language. Higher-Order and Symbolic Computation, 12(3):221–236, 1999. 19
- [23] Mads Torgersen. The expression problem revisited. In ECOOP 2004 Object-Oriented Programming, volume 3086 of Lecture Notes in Computer Science, pages 1–44. Springer Berlin / Heidelberg, 2004. 14
- [24] Alessandro Warth, James R. Douglass, and Todd Millstein. Packrat parsers can support left recursion. In PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pages 103–110, New York, NY, USA, 2008. ACM. 19, 21, 69, 86

The OOPEG Grammar for OOPEG

```

namespace Oopeg2
{
  namespace ParserGenerator
  {
    tree ADeclaration;
    tree AExpression;

    Start <- _ (Declaration / COMMENT)*;

    Declaration = (Namespace / Tree / RuleDeclaration) $;

    Namespace : ADeclaration <- 'namespace' SPACE Identifier _ (DOT Identifier _)* LCURLY (
      Declaration / COMMENT)* RCURLY;

    RuleDeclaration : ADeclaration
      <- Rule SEMI;

    Tree : ADeclaration
      <- 'tree' SPACE Identifier _ SEMI;

    Rule : AExpression
      <- Identifier _ (EQUALS / ASSIGN / TreeAnnotation? ARROW) ChoiceExpression;

    TreeAnnotation
      <- ('?:' / COLON) _ Identifier (DOT Identifier _)* _;

    ChoiceExpression ? : AExpression
      <- SequenceExpression (SLASH SequenceExpression)*;

    SequenceExpression ? : AExpression
      <- SingleExpression+;

    RepetitionExpression : AExpression
      <- PrimitiveExpression Repetition;

    AssertionExpression : AExpression
      <- '&' PrimitiveExpression;

    NAssertionExpression : AExpression
      <- '!' PrimitiveExpression;

    SubRule : AExpression
      <- Identifier _ (DOT Identifier _)*;

    Literal = '\\'' (Value : AExpression <- ~((\\'|\\\\\\\\|\\^')+~) '\\'' _;

    Regex <- '~' (Pattern : AExpression <- ~((\\|~|\\\\\\\\|\\^~')+~) '~' _;

```

```

LiteralList ? : AExpression
  <- LCURLY Literal (COMMA Literal)* RCURLY;

Spaces : AExpression
  <- ' ' _;

Commit : AExpression
  <- '$' _;

// Possible error following causes fail: !Rule+
SingleExpression = NAssertionExpression / AssertionExpression / RepetitionExpression /
  PrimitiveExpression;
PrimitiveExpression = Literal / Range / Commit / Spaces / Regex / LiteralList / SubRule /
  SubExpression ;

SubExpression = LPAR (Rule / ChoiceExpression) RPAR;

Repetition
  <- LCURLY Integer _ COMMA (Integer _)? RCURLY
    / STAR
    / PLUS
    / QUESTION;

Identifier
  <- [a-zA-Z][a-zA-Z0-9_]*;

Integer <- [0-9]+;

Range : AExpression <- '[' (Invert <- '^')? ( UnicodeClass / CharRange / Char)+ ']' _
  Repetition?;
Char <- {'\a', '\t', '\r', '\v', '\f', '\n', '\e', '\\'} / '\x' [0-9a-fA-F]{2,} / '\u'
  [0-9a-fA-F]{4,} / '\]' / '\[' / [^\n\r\]];
UnicodeClass <- '\c' {'Lu', 'Ll', 'Lt', 'Lm', 'Lo', 'Mn', 'Mc', 'Me', 'Nd', 'Nl', 'No', '
  Zs', 'Zl', 'Zp', 'Cc', 'Cf', 'Cs', 'Co', 'Pc', 'Pd', 'Ps', 'Pe', 'Pi', 'Pf', 'Po', '
  Sm', 'Sc', 'Sk', 'So', 'Cn'};
CharRange <- Char '-' Char;

COMMENT == ('/' [^\n\r]* / ~/\.*?\/) _;
SPACE == [ \f\n\r\t\v]+;
ASSIGN == '=' _;
EQUALS == '==' _;
RPAR == ')' _;
LPAR == '(' _;
LCURLY == '{' _;
RCURLY == '}' _;
SLASH == '/' _;
DOT == '.' _;
COMMA == ',' _;
STAR == '*' _;
PLUS == '+' _;
QUESTION == '?' _;
ARROW == '<-' _;
SEMI == ';' _;
COLON == ':' _;
}
}

```