Insert here your thesis' task.

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

New Ruby parser and AST for SmallRuby

Bc. Jiří Fajman

Supervisor: Ing. Marcel Hlopko

18th February 2016

Acknowledgements

I would like to thank to my supervisor Ing. Marcel Hlopko for perfect cooperation and valuable advices. I would also like to thank to my family for support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 18th February 2016 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Fajman, Jiří. *New Ruby parser and AST for SmallRuby.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2016.

Abstract

The aim of this thesis is to design, implement and test a new Ruby parser for SmallRuby. As a validation, the new parser should parse sources of selected Ruby projects. I took the AST from Rubinius. The lexical analyzer is written by hand and I created a complex tool for generating syntax analyzers from Bison grammar file. The project contains an AST testing infrastructure.

Keywords Parser, Ruby, Smalltalk, SmallRuby, AST, Rubinius, MRI, lexical analysis, syntax analysis, compiler-compiler construction, Yacc, Bison

Abstrakt

Cílem této práce je navrhnout, implementovat a otestovat nový Ruby parser pro SmallRuby. Parser jsme validovali oproti zdrojovým kódům vybraných projektů v Ruby. V návrhu AST jsme vycházeli z Rubinia. Lexikální analyzátor jsme vytvořili zcela od základu a také vyvinuli generátor syntaktických analyzátorů pro gramatiky ve formátu Bison pro Smalltalk/X. Součástí projektu je i testovací infrastruktura pro validaci generovaného AST.

Klíčová slova Parser, Ruby, Smalltalk, AST, Rubinius, MRI, lexikální analýza, syntaktická analýza, compiler-compiler, Yacc, Bison

Contents

List of Figures

List of Tables

Introduction

Today's world is full of computers and computer software. Most of people cannot imagine everyday life without smart phones, tablets, notebooks and many other devices. They start the day by opening mobile email client and social network application to check new messages and events, then enter the car and find their way to work on GPS.

All of these activities require software to run, but it must have been created somehow. All programs written in a programming language must have been programmaticaly read and parsed before they could have been further processed, i.e. to create executable files or interpret the source code. And this is the purpose of a parser.

The aim of the thesis is to design, implement, and test a new Ruby parser for SmallRuby. Ruby is a modern programming language, where almost everything is treated as an object. It is intended to be comfortable for users, therefore it offers a wide range of language constructs. Ruby is a very flexible language, where a lot of constructs can be written in more than one way. However, this flexibility makes it much more difficult for programmers to parse Ruby.

We take a look at Ruby language and its main features and evaluate current most important implementations of Ruby. There are a lot of different implementations of Ruby. They are implemented in various programming languages and designated for different platforms. They also vary a lot in performance.

After choosing the most suitable implementation for SmallRuby, we analyze its structure and introduce a custom implementation of a parser in Smalltalk. A parser consists of two main parts, the lexical analysis and syntax analysis. We highlight the most significant and challenging parts of both these parser components and test their functionality. We also focus on technologies used by these implementations to generate lexical analyzers.

Next, we will design and implement a flexible and reusable tool for automated generating of syntax analyzers. This tool is supposed to generate

a source code for Smalltalk/X, which is a perspective dialect of Smalltalk. Development of all parts of the project will be guided by test-driven development whenever possible to prevent issues with the core parts of the new Ruby parser.

Then we will test the parser against real Ruby projects and validate the generated AST with that of MRI. Within this part we will also design and implement a framework for validating the AST against the reference MRI parse tree.

Finally, we will measure and evaluate parser's performance. It includes selection of criteria for testing inputs, parsing the inputs with both the new Ruby parser and MRI, and comparation and evaluation of the results.

The structure of the thesis is as follows:

- Chapter 1: General introduction to Ruby and its implementations
- Chapter 2: Overview of syntax analyzer generation tools
- Chapter 3: Design and implementation of a lexical analyzer
- Chapter 4: Syntax analysis and parsing techniques
- Chapter 5: Designing a tool for generating syntax analyzers
- Chapter 6: Validation of the parser and performance analysis

Chapter 1

Ruby

In this chapter we will highlight the most important features of Ruby language. After that we will walk through various available implementations of Ruby and then choose one, which is most suitable for this assignment and use with SmallRuby. The selected implementation will be used as a model for creating a new Ruby parser in Smalltalk/X. Finally we will elaborate on parsing techniques, which are used in given implementation of Ruby.

The development of Ruby as a programming language started in 1993 by Yukihiro Matsumoto, who is also known as "Matz." Ruby was released to the public in 1995. The MRI implementation was considered the reference of Ruby until the specification of Ruby language in 2011. [19]

The author of Ruby selected useful parts of his favorite programming languages. He took inspiration from Perl, Smalltalk, Eiffel, Ada and Lisp and formed a new language that combines functional programming style with imperative programming.

Ruby was intended to be a language, which is comfortable for the Ruby programmer. It offers a variety of language constructs and a majority of them can be written in two or more ways.

1.1 Features of Ruby

1.1.1 Everything is an object

"I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python," said Yukihiro trying to find an ideal syntax for a completely new programming language. He wanted to be able to give properties and actions to everything in code, therefore everything in Ruby is an object. Properties are called instance variables and actions are caller methods. As an explanation of the Ruby's pure object-oriented approach, let us use the most common demonstration which performs actions on numbers:

```
3.times { print "Hello Ruby World!" }
```

The line above performs print of a string and this action is repeated 3 times. Many operations on numbers in Ruby are quite readable for a programmer. Widespread programming languages often don't consider numbers and other basic data types to be objects and need to use a different construct e.g. for repeated actions. [6]

1.1.2 Flexibility

The intention of Ruby is not to restrict the programmer. Thus a lot of Ruby features can be redefined in almost any way. These parts can be modified or even removed at runtime.

As an example, there is a built in class called `Numeric` which represents a numeric data type. Although you can use its builtin methods to perform for example subtraction, there is a possibility to add a custom method directly to the built in `Numeric` class. Let us look at the example:

```
class Numeric
  def minus(x)
    self.-(x)
  end
end

y = 4.minus 2
```

The result of the example is obviously the same as if we wrote a more straightforward

```
y = 4 - 2
```

The class definition begins with keyword `class` followed by an alphanumeric identifier, which defines name of a given class. The body of the class goes after a new line or a semicolon. Unlike in other programming languages, you are not forced to add a semicolon at the end of each statement. You can use a new line instead, however semicolons are valid line separators too.

A method call is prefixed with keyword `def` followed by a method name and method parameters. Both class and method (and other block-like structure) body is followed by keyword `end`. There is an alternative way how to define blocks. You can surround a block body with curly braces { and }.

As opposed to for example Java, you can split definition of a class to as many files as you like. There are several more ways to combine source code from different locations, we will take a closer look at it later in section 1.1.4.

Similarly to the line separator above, there are more ways how to invoke a method call. The first option is to add a list of parameters after the method

name followed by a space, i.e. not to surround the list of parameters. This option has some restrictions. There is no problem when you write simple commands, however if you use it in a more complex code, complication with associativity of operators will soon arise.

This is why Ruby syntax preserves the second option, which means to surround method arguments with parenthesis. This option can be used everywhere and unlike the first possibility does not face any associativity problems.

Predefined operators in Ruby such as +, -, *, / and others are just a "syntactic sugar," but they can you can redefine them as any other methods.[6]

1.1.3 Blocks

Another powerful feature of Ruby are blocks. In Ruby you can pass to a method many types of argument. Block is a special type of argument that can also be passed to a method.

Basically, it is a sequence of code, which can be executed by the invoked method, thus the method's behaviour can be controlled by the block. A block can take parameters like a regular method. Typical examples of controlled methods are those working with an array:

map – creates a new array containing values returned by the block

select – creates an array containing only those elements, for which the block returns true

delete_if – deletes every element for which the block evaluates to true

Let us take a look at a more specific example, which appends an exclamation mark after each element:

```
["a", "b", "c"].map {|x| x + "!" }   #=> ["a!", "b!", "c!"]
```

Finally, the block given as a parameter can be invoked using keyword `yield`. [7]

1.1.4 Mixins

Almost every modern language has support for inheritance. Ruby has support for inheritance as well. However it does not support multiple inheritance, only single inheritance is allowed. But there is a good reason for this.

Instead of multiple inheritance, Ruby introduces the concept of modules. A `module` is a collection of methods, which is in Objective-C also known as Category. The purpose of multiple inheritance is to gain methods of more than one superclass, therefore classes can use methods introduced by a specific module. Modules are a kind of mixins, because methods of a module are "mixed" into the class.

Inside of a class definition it is possible to mix methods of a module using keyword `include` followed by the module name. Of course, it is allowed to include more than one module. The concept of modules avoids many problems related with multiple inheritance.

Prohibition of multiple inheritance, however, does not limit the possibility to use multilevel inheritance. Multilevel inheritance allows a subclass to become a parent class another one, thus there may be more than one level of inheritance. [6]

1.1.5 Variables

In Ruby there are three basic types of variables:

- local variable
- instance variable
- global variable

Local variables are limited to the scope of a current block or function. They cannot be used outside of this scope except if they are assigned to for example an instance variable. Local variables, unlike the other types of variables, does not use any prefix symbol, so both variables and methods may have the same name. There are a few exceptions, where the name cannot be same for both a variable and a method. Methods can be suffixed by exclamation mark, quotation mark or equals sign, while each of these indicates a specific behavior.

Instance variables are prefixed with character @, thus there is no need to use keyword self. Scope of these variables is limited to the scope of the class instance and they can be referenced not only from the class methods. [6]

The last type are global instances. These are prefixed with dollar sign and may be referenced from anywhere in the namespace.

There is no need to declare a variable in advance or even declare its data type. They can be assigned a value without previous declaration, but you cannot read a value of an undeclared variable, this will raise an exception.

1.1.6 Naming conventions

Every programming language has its naming conventions. Class names and constants should be a sequence of upper case words. For variables and method names Ruby suggests to use a "snake case." It is a sequence where every word begins with a lower case letter and each two words are separated by an underscore.

A speciality of Ruby are method names ending with a quotation or exclamation mark. The first one indicates a getter returning a boolean value, while

the latter one does not create a new instance and applies all modifications on self.

1.1.7 Summary

In summary, Ruby is an interpreted programming language, which considers everything as an object. It has support for simple inheritance with support of modules, whose methods can be included into any class.

Ruby is very flexible as it allows redefinition of a lot of its parts, allows to choose form more possible forms of a block definition or a different function call syntaxes. There is also support for blocks which can affect behavior of a method.

However, flexibility and comfortable language constructs are the reasons, why the syntax of Ruby is so extensive and complicated.

1.2 Ruby implementations

1.2.1 Rubinius

When we talk about a modern language platform which supports a number of programming languages, we must mention The Rubinius Language Platform. Whereas the purpose of this thesis is to create a Ruby parser, the features not relating to Ruby language are beyond our scope.

The most interesting features are the following:

1. virtual machine
2. garbage collector
3. JIT
4. concurrency support
5. multiple operating systems support

Rubinius platform includes its own bytecode virtual machine and a generational garbage collector. A frequently used feature to increase performance of the virtual machine is just-in-time compiler (JIT). Basically, it refers to translation of a source code to some kind of binary code that occurs after a program begins its execution.

Often not the whole source code is translated, only the most frequently used parts. The main purpose of JIT is to increase the speed of program execution, especially when the code is translated into instructions, which are direcly executable on the underlying hardware. [12]

The platform uses native operating system threads to satisfy concurrency support. In Rubinius there is no global interpreter lock. One of important

advantages of Rubinius is multiple operation systems support. Rubinius can run on various Unix/Linux operating systems and on Mac OS X. In time of writing this text, Rubinius has no support for Microsoft Windows.

These are the main parts of Rubinius:

- Ruby parser
- Ruby bytecode compiler
- Ruby core library
- C-API compatibility for native C extensions

An interesting thing is that a considerable part of Rubinius is written in Ruby. The Ruby core library is almost entirely in Ruby. The debugger, the Ruby bytecode compiler and other tools are also written in Ruby. The Ruby parser, however, is defined using Bison grammar file combined with pure C. In addition, the parser of Rubinius written in C references Ruby core, which is written in Ruby. [25]

1.2.2 JRuby

Another implementation of Ruby language is JRuby. JRuby is distributed as a free software licensed under the BSDL license. This implementation is also very specific, because it is completely written in Java.

JRuby provides a complete syntax for Ruby, all of the core native classes. It has its own implementation of most of the Ruby Standard Libraries. The most of standard libraries are own complements of Ruby files. But a few of them that depend on C language-based extensions have been completely reimplemented. These standard libraries are not complete (last checked in January 2016). [5]

1.2.2.1 Differences between JRuby and MRI

In a perfect world, all Ruby implementations would behave absolutely the same in all situations. However because this is very hard if not impossible to achieve, there exist some minor differences. Many of differences are caused by differences between C/C++ and Java. Because the list of differences is quite long, I will mention only some of them.

In JRuby it is not possible to run native C extensions. This is often being resolved by porting popular libraries to Java or with a help of FFI (FFI has become a popular alternative to binding to C libraries).

Time precision is also an issue caused by Java. Since it is not possible to obtain precision in microseconds in JVM, Time.now.usec cannot return time values with as high precision as MRI does.

Also there is a difference in the invocation of external processes. This difference can be seen only on Microsoft Windows. When using function `system`, JRuby tries to be "smarter" than MRI. If the invoked file is not binary executable, MRI requires you to specify the file suffix to run requested script. JRuby can manage to run the script even without having the suffix. An example:

```
system('script')
system('script.bat')
```

The second expression works fine on both implementations, but in the first case the script is run only on JRuby. On MRI it fails to find the file.

Another advantage of JRuby is "true parallelism." JRuby offers really good distribution of work among all physical processor threads. [14]

The parser of JRuby is based on Bison parser generator, which has been ported to Java. Then the parser has been generated using Jay parser. In fact, in JRuby there are two parsers. One for 1.8 and one for 1.9 mode.

JRuby includes a hand-written lexical analyzer. This implementation of lexer is used in both JRuby parsers, with some version-specific tweaks. [16]

Chapter 2

Compiler-Compiler

2.1 Yacc

Generally, the input of every computer program has a structure. In terms of theoretical computer science we can think of this structure as of an input language, which is accepted by that computer program. An input language may be from a very simple one, e.g. a sequence of numbers, to a very complex one, e.g. a complete programming language.

One way to describe an input language is to use Yacc. Yacc is a general tool to describe the input to a computer program. First of all the user specifies the structures of the input and then Yacc tries to recognize these structures. When one of the specifies structures is recognized, there may be added a code to be performed.

The Yacc user writes a specification of an input process in a specific Yacc format. This format consists of three main parts:

1. declarations
2. rules
3. programs

The declarations part typically contains grammar symbol definitions including precedence information for grammar symbols and instructions, which rule to use when a parsing conflict occurs. However, this section may be empty.

The rules section can contain one or more grammar rules. A grammar rule has the following syntax:

```
A  :  BODY ;
```

A is a name of a nonterminal and body can be made up of zero or more literals and nonterminals. A colon and a semicolon are Yacc puctuation. [27]

2.1.1 Tokens

The programs part includes the input subroutine, which reads the input sequence of characters and provides the next basic input item. This part of a parser is called lexical analyzer.

The next basic item is called token. "A token is a group of characters having collective meaning." The token can be a word or a puctuation mark. In case of a grammar describing a programming language the token may be an identifier name, a keyword, e.g. if, else, while, class...

The token is then passed to the syntactic analyzer, which tracks the sequence of input tokens and typically creates higher lever structures and performs corresponding actions – the user supplied code. [23]

2.1.2 Abstract Syntax Tree

These higher level structures are typically combined to form an *abstract syntax tree* (hereinafter referred to as AST). The AST created from the source code of a programming language captures behaviour of the given source code in such a way that is comfortable and easy for later processing. Generally we want to have a node for every language construct. The leaves of AST usually represent operands associated with their parent node.

The process of turning the source code into AST can be seen in figure 2.1.

When the input is turned into an abstract syntax tree, it can be processed in any way. In the most cases they are optionally passed to an optimization mechanism and then translated to computer native code (for example C language) or to a bytecode (for example Java) or directly interpreted.

Translation to the native code is used for example in C language. Bytecode form is utilized for example in Java, which is then performed by a virtual machine.

Virtual machines are, compared to the native language execution, a bit slower, however, virtual machines are designed to work on many different platforms interpreting the same program bytecode. Consequently there is no need to recompile the source code. In case of Java JVM takes place (Java Virtual Machine). SmallTalk languages uses the virtual machine as well. [17]

2.1.3 Yacc summary

Summarized, Yacc is a general purpose tool to expose input structure to a computer program. The Yacc user specifies grammar symbols, which appear in the grammar, and grammar rules that define particular structure of the input. Yacc then turns these specifications into a source code of a common programming language. In case of Yacc, C or C++ language is used. Generated source code can be compiled by a normal compiler, e.g. gcc or g++.

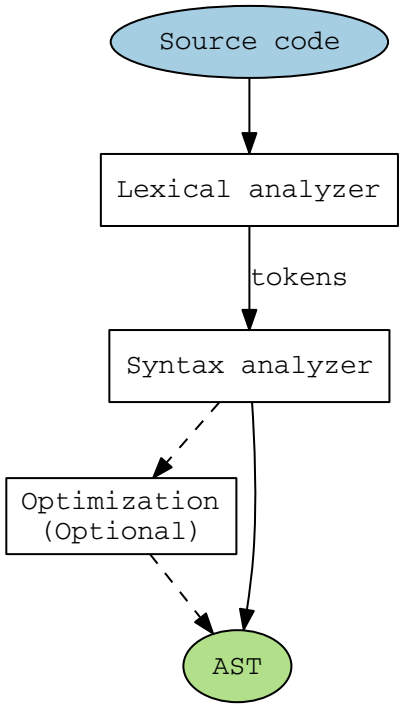

Figure 2.1: Simple parser pipeline

2.2 Bison

Bison is a general purpose tool to generate a deterministic LR or generalized LR parser using LALR(1) parser tables. As a source, Bison takes an annotated context-free grammar. [9]

With Bison, we are able to create a wide range of language parsers from simple desk calculators to such a complicated programming languages as Ruby. Bison compiler-compiler is directly used in Rubinius Melbourne, which is one of the Ruby language implementations.

Rubinius is based on a context-free grammar written in Bison. The author of this grammar is Yukihiro Matsumoto, who is also known as Matz. He was contributing to the grammar development since 1993. [25]

One of great advantages of Bison is that it is backward compatible with Yacc compiler-compiler. Therefore every parser in Yacc, which is written properly, should work well with Bison without any changes. This is really good to know for the everyone who is familiar with Yacc, because he ought to

be able to write parsers in a way he is used to with little trouble. The only requirement is to be fluent in C or C++ language. [9]

2.2.1 Bison grammar

Although Bison is backwards compatible with Yacc, its structure differs in some points. As opposed to Yacc grammar file, Bison grammar file has three main sections with one nested subsection, which is optional:

```
%{
C declarations
%}

Bison declarations

%%
Grammar rules
%%

Additional C code
```

In both Yacc and Bison, there has to be a special sequence of characters which separates each section. It must be a sequence, which is not used in regular C/C++ file or in Java source code. For this purpose, the authors of Bison choose two percent characters %% on a separate line. This sequence separates three main sections, %{ and %} surround the C declarations part.

The exact form of the Bison specification file is important for this thesis, because we are going to process the grammar file programatically to create the syntactic analyzer part of the Ruby parser and an AST generator.

Bison by default works with C/C++ languages, but there is also an experimental feature, which offers a possibility to use Java with Bison. For the purpose of this thesis, we will focus on the first possibility, which is also used in Rubinius Melbourne to describe the grammar of Ruby language. For additional information about using Java with Bison, please visit [10]. Further details about the Bison format will be described in section 5.3.

Chapter 3

Lexical analysis

For the new Ruby parser, I derived the design of the Rubinius lexical analyzer. And because the implementation of the new Ruby parser is really complex, I will discuss the most important parts of the tokenizer, especially the main loop, which is responsible for recognizing input characters and determining their purpose.

3.1 Branches

In programming languages, there are two types of jump statements. These are known as conditional and unconditional branches. The first type performs branch only under certain conditions, while the second type jumps unconditionally, regardless of the any condition.

A `goto` statement or unconditional branch can be avoided in high-level programming languages, but this can hardly be done in low-level programming languages. This is because the high-level programming constructs such as `if` command are translated into low-level programming language equivalent of goto. [31]

A `goto` statement allows us to perform an unconditional jump from the `goto` command to a statement equipped with a label. A `goto` statement is sometimes used in C language to increase performance of critical programs, such as system drivers. We can see an example of this goto statement on figure 3.1. [1]

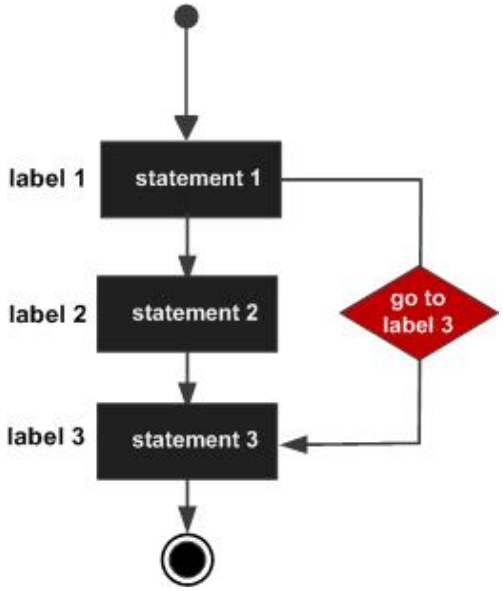

Figure 3.1: Example of a goto statement

However, goto statements are always recommended not to be used, because Its usage implies some disadvantages. In particular, their usage in a program considerably decreases code readability.

There are two types of jumps in the code:

- Forward jump
- Backward jump

As a goto can perform both forward or backward jump, it is significantly harder for a programmar to trace the program flow. [4]

3.2 Goto in SmallTalk

In the source code of Rubinius there we encounter many occurences of a goto statement. Almost whole main function of its lexical analyzer, the `yylex`, is written using goto statements. SmallTalk, however, does not support goto statements, so we must use a different construct to simulate the original functionality.

For this purpose, we will use exception handling mechanism instead of goto statement, which is not present. Let's create class `Goto`, whose purpose is to encapsulate the block of code, which should be executed next, and to raise an exception containing this block, so that this code block can be executed by the exception handler.

Furthermore, we define these code block in advance even before entering the code sequence, which handles goto statements, and then we pass them with the exception. This allows us to write blocks of code only once, save them into local variables and then invoke when needed without redefining them on all places they are used on.

```
|blockRetry blockNormalNewline ... |
blockRetry := [
    "code after original retry label"
    ...
].
blockNormalNewline := [
    ...
    Goto block: blockRetry.
].

"goto loop"
...
```

In the previous example, we have transformed code snippets, whose beginnings were originally marked with labels (an identifier followed by a colon)

into SmallTalk code blocks. These block are stored in local variables and will be passed as arguments to a method of Goto class. This method then raises an exception containing the block reference. Goto exception handler loop will be discussed later in section 3.3.

Now, let's focus on the line which realizes goto parts:

```
Goto block: aBlock.
```

This new bit of code seems similar to the original C language code. It is obvious that SmallTalk has no built-in Goto construct, therefore we have to create a new class named Goto.

Java or C++ handles exceptions in a way that the block, that is supposed to throw an exception, is surrounded by a try-catch block. This is very similar in SmallTalk. Smalltalk exception handling only use on: do: method on a block instead of try-catch.

The major difference, however, is discovered when we need to raise an exception. Java, C++ and similar programming languages utilize keyword `throw` to raise an exception. SmallTalk does not contain `throw` keyword, but makes use of `Error` class. This class comprises `signal` method, which substitutes keyword `throw`. The method causes an Error class (or its descendant class) to be raised.[3][2]

And because our goto substitute uses exceptions, we need to derive the native Error class. The `block:` is a class method that arranges creation of a new instance of Goto and stores given block reference in instance variable. Then the `signal` message is sent and our goto handling loop takes place.

3.3 Exception handling loop

At this moment we are able to raise an exception with appropriate block to be executed, but the exception is not caught. This is done by goto loop:

```
"Goto loop"
block := retryBlock.
[true] whileTrue: [
    [block value] on: Goto do: [ :goto |
        block := goto block.
    ].
].
```

First, we store code block, which will be executed first, to a local variable `block`. Then we enter an endless loop. Every time we run the current block a Goto exception is thrown to perform the unconditional jump.

Now we need to make sure that we leave the loop sometime. The output of the `yylex` function is an integer value, therefore we can directly return a value from one of the local blocks and leave the whole method.

3.4 Different levels

Now we may perform a basic goto command in SmallTalk. However, Rubinius does not use only basic jumps.

```
start:
if( [ condition ] ){
    [ action A ]
    innerLabel:
    [ action B ]
}
[ action C ]
```

In `yylex` function, we can find a label even in the middle of an if statement. Thus we need to take care of these situations:

1. condition is met;
2. condition is not met;
3. jump to the label in the if statement.

In the first case, we reach the label and we naturally continue behind it, in a more formal way we follow actions this order: action A, followed by action B, action C.

The second case causes to jump after the if block, thus executes only Action C. The last case jumps right into the middle of the if block regardless the condition. This executes only action B followed by action C.

The problem is that the original implementation allows as many labels in one block as needed, whilst our SmallTalk implementation of a goto block cannot contain another labeled block.

This can be figured out by separating blocks by the end of inner block and adding extra gotos before the inner label and to the end of the inner block.

We can illustrate the situation as follows:

```
start:
if( [ condition ] ){
    [ action A ]
    goto innerLabel;
    innerLabel:
    [ action B ]
    goto end;
}
goto end:
end:
[ action C ]
return result;
```

This may look a bit complicated or needless, but in this form we can straightforwardly transform the code into SmallTalk using our `goto` construct.

```
|startBlock innerBlock endBlock|
startBlock := [
    ( [ condition ] )ifTrue: [
        [ action A ]
        Goto block: innerBlock.
    ].
    Goto block: endBlock.
].
innerBlock := [
    [ action B ]
    Goto block: endBlock.
].
endBlock := [
    [ action C ]
    ^ result
].
```

This the exact transcription of the previous schematics. Finally, we only have to set `startBlock` as entry point and enter the loop. Using this separation of code parts, we don't have to repeat pieces of code twice or more times.

Chapter 4

Syntax analysis

Now, when we have created a tokenizer, we can continue and create the second part of the parser – the syntax analyzer. The syntax analyzer or parser takes tokens from a tokenizer in a form of token streams. The parser analyzes input tokens and validates them against syntactic rules to detect, whether the input sequence is a part of a input language or not. [13]

4.1 Grammar

Every parser has well-defined its input language – syntax, which can be accepted and further processed by the parser. This syntax is usually defined using a *grammar*. There are many types of grammar, where each one has defined its capabilities and limits. We will take a closer look on Context-Free Grammar.

A context-free grammar is a set of four components:

- set of terminals
- set of nonterminals
- set of productions
- starting symbol

A terminal is a basic symbol, from which the language is formed. Every product of a tokenizer is a terminal. The task of the syntax analysis is to determine, whether the input sequence of terminals is correct. Many people use lowercase letters for terminals. [13]

A nonterminal is a more abstract grammar symbol. We can imagine them as syntax variables which denote acceptable sequences of terminals. The nonterminals define sets of strings that are generated by the grammar. The set

of nonterminals is a list of all nonterminals that can appear in the grammar. A capital letter is often used as a nonterminal name.

The third component is a set of productions. Productions are also often called grammar rules. The grammar rules specify the way in which nonterminals and terminals can be combined to form another nonterminals. Each production is made of a left side, an arrow, and a sequence of terminals and nonterminals called right side of a production. The left side consists of a nonterminal from the set of nonterminals. We can define more than one rule with the same left side.

There may arise a situation, when an empty string is a part of the language generated by the grammar. In this case we need a special symbol to define an empty right side of the production. In other words, we can produce the left side terminal without reading any input.

We need to have an entry point, otherwise we don't know from where to start the generation of strings. This entry point is called `starting symbol`. The starting symbol is a nonterminal, from which the parser derives other nonterminals and terminals. [13]

4.1.1 Derivation

In order to determine whether the input string is a part of the language generated by a grammar, we *derive* the input string. A derivation is basically applying a sequence of grammar rules on the input string. It is be formed by applying one or more production rules.

In parsing, we have to take two decisions for a sentential form of the input:

- which nonterminal to replace
- which rule to apply

We have two options from where to begin derivation of the right side of a production rule. If we decide to start from the left side, this procedure is called left-most derivation. We take the first symbol of the rule from the left and proceed to the next symbol until we reach the right-most symbol.

However, we have the option to start from the right side and this procedure is called right-most derivation or right-sentitial form. For purpose of this thesis we stick to the left-most derivation. [13]

4.1.2 Ambiguity

Here is an example of a simple context-free grammar:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow id \end{aligned} \tag{4.1}$$

In example 4.3 there is a set of three production rules. Consider the following input:

$$a - b + c \tag{4.2}$$

Provided that E is a starting symbol, we have to choose one rule with nonterminal E on the left side. However, all three rules have symbol E on the left side. Therefore we have to choose only one of them at the moment.

This is the case when a different order of applying the rules gives a completely different result and a different parse three, that is why ambiguity in grammar is not desired. There are two factors, which affect the form of an abstract syntax tree:

- operator associativity
- operator precedence [13]

4.2 Operator associtativity

Assume that the first rule constructs a node representing operation add, where the left child node is a left operand and the right child stands for the right operand. The second rule behaves the same, except that the operation is not addition but subtraction.

If we first apply the rule with addition, we will receive a parse tree with operation add as a root node and operation subtraction as his left child. If we visualize the parse three using parenthesis, we will receive $(a - b) + c$.

In the case we first apply the subtraction rule, we will have the subtraction operator as the root of the parse three and the result will be $a - (b + c)$.

Both these situations can be seen on fig. 4.1. The first case if placed on the left side of the figure.

If the operand has operators on both sides (which is the situation in the example), the side on which the operator takes the operand depends on the associativity of these operators. There are three options for the associativity:

- left-associative
- right-associative
- nonassociative

If the operator is left-associtative, the operand will become the right part of the left operator, in case of right-associative operator, the operand will be part of the left node of the operator to the right from the operand. [13]

However, there is also the third option, which means that the operator is not associative. The meaning of this associativity option is to deny any

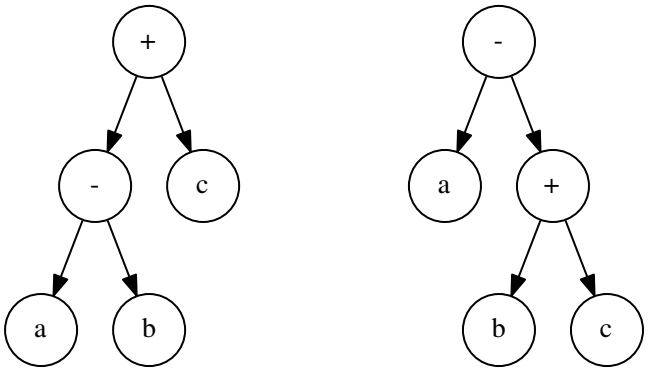

Figure 4.1: Example of operator associativity

associativity decisions with this operator and to raise an error whenever the need of associativity decisions arises.

For the purpose of creating a parse tree, we usually consider operations like addition, subtraction, multiplication and division to be left-associative. This behavior can be seen on fig. 4.1 on the left side of the figure.

4.3 Operator precedence

We are familiar with both results of the previous example, because we consider the add and subtract operations to have the same priority. But in Ruby there are many operators that have a different priority, therefore we need a mechanism to resolve this.

The key for this situation is the operator precedence. Imagine a grammar with two operators with a different priority, say addition and multiplication.

As an example we will use the following grammar:

$$\begin{aligned} E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow id \end{aligned} \tag{4.3}$$

As the input we use the following expression:

$$a + b * c \tag{4.4}$$

Provided that E is a starting symbol, we have to choose one rule with nonterminal E on the left side. However, all three rules have symbol E on the left side. Therefore we have to choose one of them.

We can use the first rule to gain $E + E$. Then the first E, after using the third rule, stands for a and the second one denotes $b * c$. We use the second rule on the second E to produce $b * c$ and we finally match the whole input.

Although we must use all these rules, we can use them in a different order. Imagine that we use the second rule first. Then the first E substitutes $a + b$ and the second one c. Rule number one is used to match $a + b$ and the third one matches single identifiers. We have also parsed the whole string, but the result is different.

We can visualize results using parenthesis. In the first case we have first matched addition operator and then the multiplication operator. Thus the result expression would look like $a + (b * c)$. The second case takes the multiplication operator first, so we gain $(a + b) * c$.

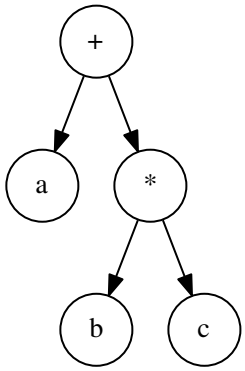

Figure 4.2: Example of operator precedence

Naturally, we are familiar with the first case, because we give the multiplication operator a "higher priority" than the addition operator.

Now, we must force the grammar to use the first case, because the second one would give us an incorrect result. We say that multiplication operator has a *higher precedence* than addition operator. The operator precedence forces the grammar to always prioritize multiplication over addition and use the form $a + (b * c)$, which is illustrated in fig. 4.2. [13]

4.4 Parsing techniques

When we search the literature for parsing techniques, we can find a lot of different techniques, but in general, there are only two techniques to do parsing – the rest are only technical details.

The first method tries to imitate the original parsing process by rederiving the sentence from the start symbol. This technique starts at the root of the parse tree and continues downwards to the leaves. This method is called *top-down*.

A common form of a top-down parsing technique is a recursive-descent parsing. It uses a recursive procedures to process the input. The greatest disadvantage of this form is backtracking. It means that the parser tries to derive a production, and when the derivation fails, it tries to restart the process using a different grammar rule of the same production. This method may process the input string more than once to find the right production.

On the contrary, the second technique tries to rollback the production process and to reduce up the sentence back to the starting symbol. It quite natural that this method is called *bottom-up*. Bottom-up parsing is most commonly done by a shift-reduce parser.

As the parser name indicates, it is based on two unique actions: shift and reduce. The shift refers to the advancement of the input pointer to the next symbol and shifting the symbol onto the symbol stack. When the parser finds a complete grammar rule (specifically the right side of the rule), it is replaced by the left side of the rule. This action is called reduce. [28] [13]

Each parsing techniques has its pros and cons. Top-down parsers are easier to be written by hand, but they need a restricted grammar, for example which does not contain left-recursive rules. Bottom-up parsers, on the other hand, can handle a larger class of grammars. They have no problem with left-recursive rules. They are also suitable for automatic parser generation.

The parsing techniques are a very complex topic, which is far beyond the scope of this thesis. For more information see [29] or [28].

Chapter 5

Smalltalk Compiler-Compiler

It is possible to rewrite the whole Ruby grammar from Bison format to Smalltalk/X by hand. However, it is also possible to create a complete Bison format parser and let the parser transform the grammar and create a whole new parser for us. There is a lot of reasons for this:

1. complexity
2. time requirements
3. human factor
4. actualizations
5. reusability

It is possible to rewrite the whole Rubinius bottom-up grammar, which has more than 3300 lines of code with references to helper functions having another more than 2500 lines of code written in pure C, by hand. But this process is really complex and a human would have serious problems with transforming that large grammar to the corresponding result form. Moreover after that there are many source code snippets in the grammar, which have to be translated to the target language as well. [26]

Doing the transformation by hand would be also very time-consuming. This work may take a lot of months, if not years. Creating of a custom compiler is also really time-consuming and challenging, but still not having time requirements as huge as the ready-made grammar transformation.

There is also a relevant reason for automated grammar processing – the human factor. A computer with a correctly written program may be considered error free. A human, however, does make mistakes, especially when performing many simple actions for a long time, it is almost impossible not to make a mistake and this kind of mistake is really hard to find. Of course there

will arise many human-factor mistakes during creation of a custom parser, but usually these can be tracked down faster.

And finally after creation of a Ruby parser by hand will surely be released a new version of Ruby and refactoring of the current grammar may be confusing and complicated. As Smalltalk Compiler-Compiler (hereinafter referred to as STCC) automates original grammar processing, it is possible to automatically transform a new grammar even after release of a new Ruby version. A programmer does not need to care about new Ruby syntax, he only manages to translate newly added code snippets from C to the target language – Smalltalk.

The custom Smalltalk Compiler-Compiler (hereinafter referred to as STCC) will not help us with translation of tokenizer nor small blocks of code included in the grammar, however it does transform any grammar we want into a parsing automaton, which is be able to recognize an input defined by a given grammar and further process it (create a parse tree, directly perform the source code or anything else). [18]

And because we need to create a tool for automated parser generation, we also need a parsing technique, which is suitable for automated parser generation. On the attached CD, there is an implementation of a top-down recursive-descent parser, which reads simple arithmetic expressions including operator precedence. It is located in file `Ruby_ExpressionGrammar.st`.

As we can see, the implementation contains while loops, which are not easy to generalize for a generating tool. It also performs a lot of backtracking, which reduces the parser's performance. After a consultation with the thesis supervisor we decided to choose a bottom-up parsing technique, which is much more suitable for automated parser generation.

5.1 Main Features

But what should the STCC be able to do? Here is a list of the most significant features that should be supported:

- parse a source grammar
- create a parser from the grammar
- export parser

First of all, STCC takes an input grammar, usualy from a source file, and save it in an internal structures. Then it should perform essential transformations on the grammar.

After that STCC is ready to make use of the grammar and begin transformation to a parsing automaton. The output parser realizes parsing using the parsing automaton, which determines, what symbols have been already read

and what symbols are allowed to be next according to the grammar. Every step of a parsing automaton may be accompanied by an action – a piece of code which performs additional actions to define the output form – in our case these actions generate the parse tree.

Now, when we have defined out parsing automaton, we can straightforwardly turn a set of automaton states into a functional parser in a programming language of our choice. [18]

As a methodology for STCC development, I have chosen test-driven development. The STCC project contains tests for all core features of the Smalltalk compiler-compiler and many other tests are performed additionally in Smalltalk environment using automated tests. The Ruby part tests are based on RSpec gem. For more information about RSpec, see [24].

5.2 Modularity

We try to make the STCC as flexible as possible. Therefore we split separate parts of a parser into modules. With modular programming, we separate concerns such that modules perform logically discrete operations. Each module has a well-defined function with a minimal side effect interaction between them. Almost every module of the application should be replaceable by a different implementation of the module, which preserves the defined interface, using which the modules communicate. [21]

In the section 5.2 we have defined three main goals of the custom compiler-compiler. We will split the STCC into modules exactly according to these operations. There is a significant profit of separating concerns into modules.

The first module is meant to read an input grammar from a Bison grammar file and then save it into an internal structure. In the future we may decide to change the source of a grammar and not to read it from a file or even not to use Bison file format. In this case we can simply create a different implementation of this module and adapt it to read a grammar from a different source.

The second module does the most difficult job. It is passed a grammar from the first module and is supposed to convert this grammar into a bottom-up parser. By replacing this module we can change the parser type to a different one, even from a bottom up to a top-down parser.

Finally, the third module takes input of the previous module and generates a final parser in Smalltalk. Even this module can be substituted. The result of this substitution is that we can change the target language, in which is the result parser generated, change into any other programming language.

Now I will go through modules in sequence and elaborate on the implementation and other information about each module.

5.3 Bison Parser

The first module takes the input Bison grammar file and parses it to fill in the inner structures of STCC. Bison grammar file is divided into four sections in a fixed order, but only two of them are relevant for the compiler-compiler.

The last part is dedicated for a tokenizer code, whose translation cannot be easily processed by a computer anyway, therefore we don't have to read this part at all.

Rubinius grammar is written in C and C needs to declare every variable or function in advance. The first part of grammar file is intended to hold these function prototypes and variable declarations. And because these declarations are used only in a tokenizer and grammar code snippets, which as well must to be translated by hand, the first part is irrelevant as well.

The remaining two parts include grammar itself and definitions of tokens attributes (such as precedence or associativity) respectively.

5.3.1 State Pattern

For reading of sectioned input file we can use design pattern called state. A State is a behavioral design pattern, which allows behavior of an object to be changed only by replacing its state object. The state object is changed, when internal state of the object changes.

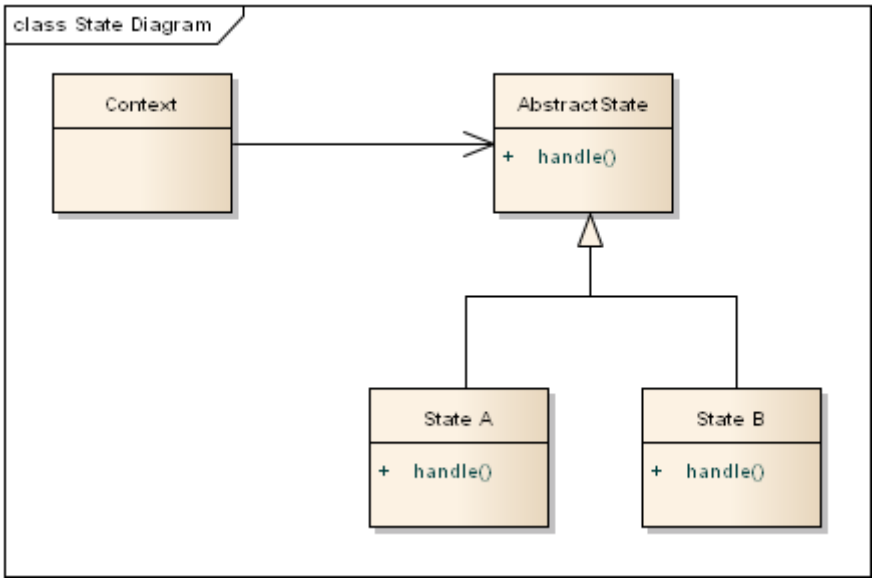

Figure 5.1: State design pattern class diagram

A class diagram in fig. 5.1 illustrates the situation. An object (which is also called "context") has a reference to a state object. This object is represented

by an abstract class with one abstract public method `handle`. This method is implemented in subclasses to define different behavior in different states. [30]

To make the Bison module more transparent, we divide it according to the grammar file structure into three main states with following purposes:

1. skip C declarations
2. parse tokens definition
3. parse grammar rules

The first reason why I have chosen these main states is because they copy the structure of an grammar file. The second reason is that every iteration through these states has only one direction – when you leave a state, you cannot return to it. This fact makes the grammar parser easier to understand and eventually alter its behavior, change the order of states or perform any other modifications in the future.

Let us create a BisonParser class, which will hold an instance of a ParserState class. The BisonParser input method will take string input (typically content of a grammar file) and distribute it to the current parser state. Every parser state has a privilege to change state of the BisonParser, when it encounters grammar section separator. Bison uses `%%` as a section separator.[9]

5.3.2 Parsing a grammar file

Because a compiler-compiler, unlike the output parser, can afford to be a little less optimized for performance, we can use regular expressions to match individual parts of a grammar.

Each state includes an ordered set of tuples, where each tuple is made of a regular expression and a block. Regular expression is tested against current line of code or its part. A block is performed only when input string matches corresponding regular expression. For additional control, this block has a mechanism to confirm processing of the input or to reject it. In case of rejection the testing continues with the next regular expression as if the regular expression was not matched.

Even inside of a ParserState we can notice that input cannot be always tested against the same set of regular expressions. Imagine a usual C multiline block comment. Because we process each line of input separately, there must be a mechanism to identify, if we are for example inside of a comment block or not. This is why each parser state can have more than one set of tuples and is able to switch between them.

Sometimes regular expression utilizes only a part of line and needs to pass the rest of the line to another regular expression. An example of this can be the definition of a grammar rule, where we do not repeat the left side nonterminal and use pipe notation instead:

```
rule            : right_side1
                | right_side2
                ;
```

Terminals, nonterminals and other grammar symbols on the right side, represented by `right_side1` and `right_side2` respectively are both parsed the same way. The diffence is in the prefix, which is parsed differently. For this reason, every block in this tuple can inform the BisonParser not to continue on the next line, but continue on the current line on whatever position the block needs.

The first state is very simple – it only skips everything placed before the declarations part of the file. When the declarations separator is found, it changes the state of Bison grammar parser to the next one.

The second state is a bit more complex. This state can be seen in fig. 5.2. This state starts in a default set of rules (called "main" in the picture). The main set is both the entry point as well as the exit point of the parser declarations state. In the declarations part there may be lines modifiyng default behavior of the parser. The Rubinius grammar file contains three of them:

- `%pure-parser`
- `%parser_param {rb_parser_state* parser_state}`
- `%lex_param {rb_parser_state* parser_state}`

The `%pure-parser` option forces the output parser to be *reentrant*. "A reentrant program is one which does not alter in the course of execution." Reentrancy is an important feature whenever it is possible to execute the program asynchronously. In case of having more thread in the application, a nonreentrant program must use means of synchronization such as mutexes or semaphores to prevent undefined behavior.

The actual Bison implementation is written in pure C and often uses statically allocated variables for communication with `yylex`. The STCC implementation elaborated in this thesis does not communicate that way, therefore this option is ignored and is preserved for compatibility reasons only.

The other two options, `%parser_param` and `%lex_param`, define parser state structure and data type. The STCC will not use them and they are skipped.[9]

The declarations part may also contain definition of the union. C language is a "strongly typed" programming language, which requires a precise defition of data types. And because not all tokens' values fit into one data type, there must more data types allowed as a return type. A union is a good solution because every token can return only one data type and a union uses the same memory block for all data allowed types and thereby reduces allocating of unnecessary memory space.

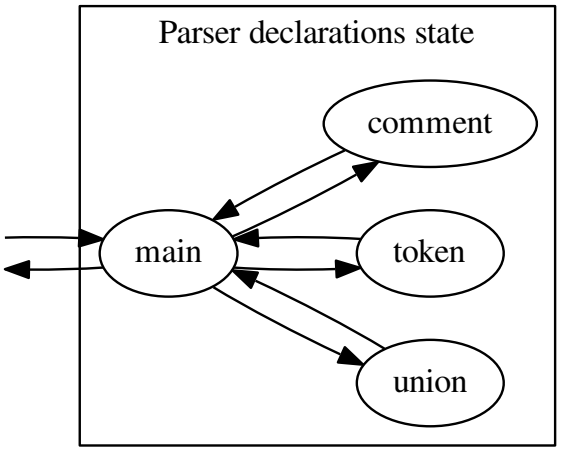

Figure 5.2: Transitions inside of the parser declarations state

```
%union {
    VALUE val;
    NODE *node;
    ID id;
    int num;
    const struct vtable* vars;
}
```

The union for return values is identified by `%union` and uses C-like notation to specify allowed data types. Smalltalk is not a strongly typed programming language and all return values can be simply referenced by one variable. Hence a union definition is useless for Smalltalk and will be ignored as the previous options.

The union definition, however, is written over more than one line, so it is not so trivial to skip it. When the *main* set of tuples encounters beginning of union definition, it switches to another set of tuples, which in the figure 5.2 uses label *union*. This set has a simple task – it iterates over lines and searches for the end of the union definition. When it finds the ending curly brace, it activates the default set of tuples.

Because the Bison grammar allows C-like block comments, the STCC must expect them as well:

```
/*
 *      precedence table
 */
```

Because these comments may take more than one line as in the example, we must detect the beginning of the comment and then switch to another set, which searches for the ending sequence. Then it switches back to the default set. [26]

The most important part is the declaration of symbols which can appear in the grammar. The definition of terminal and nonterminal symbols can be single or multiline and specifies terminal names and optionally its data type, which is marked with angle brackets.

This part should also contain any terminal symbol properties, such as associativity and precedence. Terminal associativity has reserved a three keywords:

- %left
- %right
- %nonassoc

These associativity values are explained in detail in section 4.3. Terminal precedence is specified by the order of declarations. The latter declaration means a higher precedence. Usualy more of terminal symbols can have the same precedence value. In that case these symbols are placed on the same line.

A complex example including most of features we talked about follows.

```
%token <num>  tREGEXP_END
%type <node> singleton strings
%token tUPLUS           /* unary+ */

/*
 *      precedence table
 */

%nonassoc  modifier_if
%left  keyword_or keyword_and
%right keyword_not
```

When the *main* set meets the section delimiter, the Bison grammar parser changes its state to grammar rules state.

5.3.3 Parsing grammar rules

The grammar rules state consists of four sets of tuples as illustrated in fig. 5.3. The entry point is called main and represent the left side of a grammar rule.

The left side of the rule consists of a nonterminal. This nonterminal produced after successful reduction of the right side, i.e. the grammar symbols. Bottom up grammar actions will be discussed in detail in section 5.4.1 on the page 38, which is dedicated to bottom up parser actions.

The left and right side are delimited by a colon. Another often used notation defines more rules for one nonterminal without redefinition of the left side. This is done by delimiting right side symbols by a pipe. The end of the sequence of rules is indicated by a semicolon.[9]

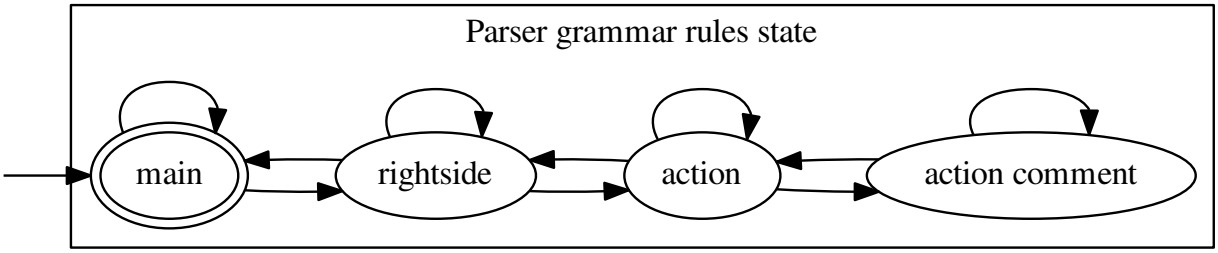

Figure 5.3: Transitions inside of the grammar rules state

The *main* set of regular expressions is displayed, unlike the other sets, as an ellipse with two peripheries. In the notation of state automatons this signifies an *accept state*. An accept state is a state, in which an automaton must end in order to accept the input.

After a parser grammar file section delimiter is found, we have everything we need to generate a parser from the input grammar. Therefore we can ommit the rest of the grammar file. In other words parsing of the grammar file ends here. Every rule definition must be ended properly with a semicolon, so the only correct state to end in is the *main*. This is why it is displayed as an accept state. [13]

5.3.4 Grammar symbols

Main set has a transition to the *rightside* set, which represent the right side of the rule. The right side can consist of a combination of the following symbols:

- terminal
- nonterminal
- action
- epsilon

A *terminal* is a symbol that can be returned from the tokenizer. It can be a single character or even a keyword or identifier. A *nonterminal* can be replaced by a sequence of grammar symbols according to the grammar rules.

To create a parse tree, we need a construct to tell the parser what should be the result of a rule reduction or execute some additional code at the given position. For this purpose Bison grammar supports actions. An *action* is a grammar symbol, which contains a piece of code to be executed at its position in the rule. Action execution is a separate topic and will be also discussed later in section 5.4.1. [9]

The rightside set can encounter an action. In this case the Bison grammar parser will perform a transition to the *action* set, which reads the content of the action. The action can also contain a comment block, which is handled in set called *action comment*. The only way to the accept state is to go backwards through all states it went through in a reverse order. These transitions can be seen in diagram 5.3.

The right side of the rule can also be empty. This is useful when there may be zero or more occurences of a symbol. Typically, we mark an empty rule with an epsilon symbol ε. Because of subsequent processing of the grammar we will use epsilon symbol only with a transition to the parser accept state.

5.4 Parser Builder

The second module is the one which consumes most of the computation time during parser creation. It has the most difficult task of all three modules – to turn the input grammar into a parsing automaton. This is done by iterating over all grammar rules to generate bottom up parser states and transitions between them.

For this task we need to be familiar with the following concepts:

- LR items
- closure function
- goto function
- algorithm for set-of-items construction

We will begin with definition of an LR item, which is needed for all other concepts mentioned before.

Definition 1. *An LR item of a grammar is a production rule augmented with a position marker (a dot) somewhere within its right hand side.*

In other words, an LR item is a set of a left side (a nonterminal which is the result of the rule application), a right side (a set of grammar symbols mentioned in section 5.3.4) and a position marker.

The position marker is usually visualised as a dot in the right side of the rule. It can be placed to the beginning of the right side, to its end or somewhere in between its grammar symbols. The position marker is important for the parser builder because it defines, which part of the grammar rule is already read and which part is still expected to appear in the input.

Now let us look at the example of LR items (in STCC implementation it is also called a *dot rule*). The production $A \to XYZ$ (A, X, Y, Z are nonterminal symbols) yields the following set of LR items:

$A \to .XYZ$
$A \to X.YZ$
$A \to XY.Z$
$A \to XYZ.$

Definition 2. *The closure(I) is a set of items I for a grammar G is the set of items constructed from I using two rules:*

1. *Initially, every item in I is added in closure* (I)
2. *If* $A \to \alpha.B\beta$ *in closure* (I) *and* $B \to \gamma$ *is a production, then add item* $B \to .\gamma$ *to I if its not already there. We apply this rule until no more new items can be added to closure* (I)

The closure function helps us to compute a set of LR items, which we can use in given situation. The STCC uses the closure function to create states of the parser. The result of the closure function represents one state that can be accessed using parser actions (bottom up parser actions are discussed later in section 5.4.1).

The creation of a closure is inspired by determinization process, i.e. transforming a nondeterminate automaton to a determinate automaton. At the time of creating the parser we do not know, what input we will parse, therefore we consider all possibilities. More information about determinization can be found at [29].

Now when we have states, we need to create transitions between them. The key for this is the goto function, which tells us what to do when a specific grammar symbol appears.

Definition 3. *For a set of items I and a grammar symbol X, the function* $goto\,(I, X)$ *is defined as the closure of set of all items* $[A \to \alpha X.\beta]$ *such that* $[A \to \alpha.X\beta]$ *is in I.*

When a terminal symbol appears in the input, we must know whether we can move to a different state or do something else. When we reach the end of the dot rule, we can – under certain circumstances – "substitute" the right side of the rule with the nonterminal on the left side. This means that we can also pass a nonterminal to a goto function.

The goto function signifies that we want to move from state I to the state J using a terminal or nonterminal symbol X. The result is a closure of the rules we gain from rules in state I, which has grammar symbol X right after the position marker.

If there were no such rules with the position marker before X, the function would return an empty set of LR items. An empty set means that the parser should not move to another state, but perform another parser action instead.

The last thing we need to do before we can use the state construction algotithm is grammar augmentation. Let S be the starting nonterminal symbol. Then add a new rule $S' \rightarrow S$ to the grammar rules and make S' the new starting symbol. The result grammar is called *augmented grammar*. This is a mechanism to enable the accept action, which will be introduced in the next section.

Now we can introduce the algorithm for set-of-items construction. We start with the closure of the first rule, which in case of augmented grammar is $S' \rightarrow .S$, and use the goto function for every grammar symbol which can possibly follow. We repeat this action for every new state until no more states can be added. [18]

5.4.1 Parser Actions

The main task of the result parser is to decide what to do based on the input. For this reason we need to create an action table, which has defined action for each state and each grammar symbol. The parser has five possible actions, from which it must choose exactly one to be the next:

- Shift
- Reduce
- Goto
- Accept
- Error

The *shift* action is used when we need to read the next token from the input. It reads the next token and pushes it on the top of the symbol stack. The second effect is that it pushes a new state onto the state stack. This means that the next decision, which action to perform, will be related to the new state.

The shift action can be performed when we have a terminal a on the input and the state I_i, which is on the top of the state stack, contains item $A \to \alpha.a\beta$ and $goto\,(I_i, a) = I_j$ is a nonempty closure. [18]

When we reach the end of a rule, we can perform a *reduction*. It means that we remove all items on the right side from the symbol stack and replace them with the nonterminal on the left side of the rule, which is being reduced.

The number of symbols removed from the symbol stack is usually equal to the count of the symbols on the right side, unless the last symbol is an action. In that case remove $n-1$ symbols, where n is the length of the right side. We also remove the same count of states from the state stack, because we need to move to the state where the rule shifting begins.

You may notice that I mentioned only action symbol on the last position of the rule. Why not for example action symbol on the first position?

The answer is simple. It is not possible to have an action symbol on a different position then as the last symbol of a rule. This is caused by using *markers*. See section 5.4.2.

When we reduce a rule, we gain the nonterminal on the left side of the reduced rule. When we have a terminal symbol, we shift the input to the next symbol and move to another state. After reduction, when we have a nonterminal symbol, we do not shift the input and we only move to another state. This is called *goto* action.

In fact, the goto action is used only after a successful reduction and can be considered a part of the shift action. However we need it to be a separate action, because we use it in the action table of a state, where we tell the parser what action to choose when a specific symbol arrives.

When we have to decide what action to perform based on the symbol, which cannot be used for neither shift nor reduce, we perform an *error* action. This action usualy starts the failover mechanism which leads to error recovery.

The last action is *accept*. This action is invoked when the parser read the whole input string and decided that the string is generated by the grammar, and therefore it should be accepted. Accept state is added to the parser additionally and can be reached by shifting end-of-input token, which is usualy written as ϵ. [11]

5.4.2 Markers

It is possible to have an action symbol everywhere in the grammar rule. If it is placed to the end of the rule, we execute the action after shifting of the penultimate symbol ("end-rule action").

But when the action symbol is placed in any other position than the last ("mid-rule action"), we expect the action symbol to be executed on its position, even when the rule has not been shifted completely. [9]

This is realized by markers. The mid-rule action is replaced by a *marker*. Marker is a new nonterminal symbol, which has only one grammar rule and

this rule contains only only grammar symbol – the action being replaced by the marker.

This rule is an epsilon production, and consequently shifts no symbol from the input. It is directly reduced, that means the action of the action symbol is performed and then we gain a nonterminal and move to the state, where the action is next in the rule. [11]

Now the action is performed at the correct position and we can use the marker nonterminal to go to the next state.

5.4.3 Resolving Conflicts

The syntax analyzer looks at state's action table if it contains the grammar symbol which is to be used. If the table contains action for the symbol, the action is performed. Otherwise, the default action is performed instead.

The error action is initially set as a default action for a new state. Then the parser builder looks for the rules, which can be reduced. The first found rule replaces the error in the default action.

When two or more reductions are available, we must use the *look-ahead token* and decide correct reduction accordingly. If two or more reductions share a common *follow*, then we face a complication. This situation is called *reduce-reduce conflict* and may indicate a serious error in the grammar. If this conflict occurs, the parser should choose one of valid next steps.

In case of reduce-reduce conflict, it chooses to reduce by the grammar rule that appears first in the grammar (for more information about look-ahead token and follow function, see [9] and [18]).

After assignment of reduce actions, we find all shift actions. As in case of reductions, there may be a situation, when the place in the action table we need for shift action is already occupied by a reduction. This situation is called *shift-reduce conflict* and is resolved using associativity and precedence settings.

Every terminal symbol may be assigned a precedence and associativity values, as discussed in section 4.2. Every grammar rule can be also assigned these properties. The rule takes the precedence value from its rightmost terminal symbol or can be explicitly defined in the grammar rule.

The general steps in this case are the following:

1. If the terminal has the higher precedence, we choose to shift.
2. If the rule has the higher precedence, we choose to reduce.
3. If both terminal and the rule have the same precedence, then we use associativity to decide:
   a) If the terminal is left associative, we choose to reduce.
   b) If the terminal is right associative, we choose to shift.

c) Terminals may be declared to be nonassociative, in which case we raise an error if the associativity is needed to resolve the parsing conflict.

If precedence values do not help to solve the conflict, the shift-reduce is resolved by choosing to shift. [15]

And, finally, there is a third possibility, a shift-shift conflict, but it cannot happen as far as the state generation is based on shifting a specific grammar symbol.[20]

5.5 Parser Exporter

Now we have a complete parsing automaton. The only thing left to do is to generate source code for the target programming language, which is in our case Smalltalk.

The interface of the parser exporter module is designed to be flexible and universal for use with other programming languages than Smalltalk. The Smalltalk parser exporter consists of three classes:

- SmallTalkBUParserExporter
- TokenProvider
- ActionTranslator

The output of the module is Smalltalk class file containing declaration of the parser class and definitions for its methods. The exporter is divided into two main parts – header and body.

The header part includes initialization of the parser, instance variable declarations with getters and setters. The header part can be the same for many different bottom up parsers.

The body is generated dynamically based on parser states and transitions between them. It contains the main loop, which handles parser states and stacks needed for its activity.

The steps of parser are the same as those of the automaton. First, we try to find an action for a specific symbol. If it is not found, we proceed to the default action.

A goto action is very easy to implement. It only pushes a new state onto the state stack. A shift action does the same thing and above that it shifts the input to the next token. The method for shifting the input can be changed by parameter `next_token` of the parser exporter.

The error action outputs the error message and stops parsing of the input. In future releases, this may initiate error recovery mechanisms. Method invoked by Smalltalk on error can be customized by changing parameter

`error_call`, or it is possible to override method `raise_error` which renders the error handling code and perform whatever action is needed.

When we want to signalize that parsing is successfully finished, we perform accept action. Every time we reach the accept action we are sure that the symbol on the top of the symbol stack is ϵ indicating the end of input. It does not carry any value, therefore we can safely remove it from the stack. The top of the symbol stack is now the output of the whole parsing process, unless it was redefined by grammar actions.

The last action and the most complicated one is reduction. First the parser checks the symbol type of the last symbol from the grammar rule being reduced, if it is an action. The end-rule action does not carry a value and takes no space on the symbol stack. What more, we should perform this action before removing symbols from the symbols stack, because this action usualy needs to work with these symbols.

Then we can safely remove all rule symbols from the symbol stack (except the end-rule action) and also the same count of the states from the state stack. The parser exporter needs identifiers of grammar symbols to generate the parser. This is covered by `TokenProvider` class.

5.5.1 Action Translation

For convenience, the parser exporter offers two ways how to provide source code for grammar actions. Intuitively, we can write the source code right into the grammar. There is, however, the second and more convenient way to do this.

When we run the STCC for the first time, it generates an action file suffixed with .cfg.dist, which contains a list of all grammar actions from the grammar. Now we can fill in the action source code.

The .cfg.dist is meant to be a template for the .cfg file. This is for safety reasons. When we run the STCC again, this file is overwritten by a new template and all customizations would be lost.

The original Bison parser uses placeholders for manipulating symbols from the symbol stack. It uses `$1` for the first parameter, `$2` for the second... It uses `$$` to mark the output of the action. In Smalltalk `$` is a reserved symbol, so the STCC uses `@@` and `@1` instead.

When the actions are filled in, we can run the STCC again to use them in the final parser. Before these grammar actions are rendered into the output source code, they are escaped. The .st format uses the exclamation mark as a special symbol, therefore we must use two exclamation marks instead. This is done automatically by STCC.

5.5.2 Dot Exporter

For a better visual inspection of the output, the STCC creates two graphical files containing the parsing automaton. First, the automaton is described by .dot format. Then it can be converted to any graphical format, STCC uses vector .ps format.

In figure 5.4 there is an example of a simple grammar, which will be turned into a parser. It processes an expression made of integer numbers and operator add.

The operator is left-associative, therefore it will evaluate the expression from left to right. The grammar action placed at the end of the third rule directly computes the result number. This grammar allows an empty string as well.

```
%token tINTEGER
%left '+'

/* Grammar follows */
%%
program         : arg
                | /* empty */
                ;

arg             : arg '+' arg { @@ := @1 + @3. }
                | tINTEGER
                ;
```

Figure 5.4: Example of a simple expression grammar

The state diagram generated from this grammar can be seen on page 60. There are also steps needed to parse an example input $1 + 2 + 3$.

Parsing starts in state 0. Lookahead token is terminal $tINTEGER$, so we shift and move to state 1. From state 1 the only possible action is to reduce to gain arg nonterminal and get back to state 0 (default action). But now we use arg's goto action to get to state 3. We shift + and then shift $tINTEGER$ to receive second arg.

Then we goto from state 5 to 6 and reduce $arg + arg$ to gain arg. We reduce because we set + to be left-associative. If we chose right-associativity, we shifted + instead.

Now we are back in state 0, because reduction has 3 symbols. After that we shift + and immediately $tINTEGER$. Again reduce $arg + arg$ and we can also reduce nonterminal program, which is the only symbol in the starting rule.

We goto using *program* and the last thing we need to do is to verify that we already reached the end of input. If we did so, we are ready to accept the input and finish parsing.

Chapter 6

Evaluation

In this chapter, we will examine capabilities of the whole parser and test it against real Ruby projects such as Rubinius Std Lib, Bundler, Rspec, Rails and Discourse.

The second part of the chapter is focused on testing performance of the parser and choice of testing data.

6.1 Validation

We suggest the best metric to measure quality of the parser is to parse real Ruby projects to find out, how much of the projects is the new Ruby parser able to parse. Then we will discuss possibilities of testing the result parse tree against MRI.

For automated testing of a whole project, I have created a validation framework, which encapsulates the process of collecting results from the parser and their evaluation. It crawls recursively through directories and searches for all Ruby files in the project. These files are individually passed to a mechanism, which manages the communication with the parser and retrieves the results.

The communication mechanism can be easily customised by subclassing the `Validator` class and overriding corresponding methods. The entry point of the framework is `ParserValidator`, which is used to evaluate these Ruby projects: Rubinius Std Lib, Bundler, Rspec, Ruby on Rails and Discourse. The results can be seen in table 6.1.

It is important to understand the real meaning of the results. The first row shows Rubinius Std Lib with 47.61 %. It does not mean that the parser is able to parse only 47.61 % of constructs in the project, but approximately 52.39 % of source files contain one or more constructs that are not supported by the new parser yet.

The parser cannot continue after an invalid code sequence is found. Failover mechanisms of the new Ruby parser are left as a part of future work. Because every Ruby file contains hundreds of language constructs, the boolean result is

project	parsed files	files count
Rubinius Std Lib	47.61 %	3468
Bundler	26.53 %	297
Rspec	28.85 %	213
Ruby on Rails	45.77 %	2080
Discourse	51.92 %	1767

Table 6.1: The new Ruby Parser with real projects source code

not sufficient. Therefore the metrics computes percentage of the parsed code before the error occured.

The advantage of this technique is that it reflects the portion of code, which was really parsed. This metrics has a disadvantage, because the result is dependent on the position of unsupported construct in the source file. It also does not take the file size into account.

The next step is to design a testing infrastructure for comparing the AST of the new Ruby parser with MRI. The fact is that the AST of MRI and the new Ruby parser, which is based on the AST of Rubinius, are not the same and they vary in many aspects.

As an example, we use the following simple input: `abc = 15`

The new Ruby parser AST

```
[
    [0] :NODE_LASGN,
    [1] "abc",
    [2] [
        [0] :NODE_NUMBER,
        [1] 15,
        [2] nil,
        [3] nil
    ],
    [3] nil
]
```

MRI AST

```
[
    [0] [
        [0] :assign,
        [1] [
            [0] :var_field,
            [1] [
                [0] :@ident,
                [1] "abc"
            ]
        ],
        [2] [
            [0] :@int,
            [1] "15"
        ]
    ]
]
```

We can see that both parse trees have many common parts. The first element in MRI is a block of statements, which actually contains only one statement. The Rubinius detects that the array contains only one statement and does not include the array node.

The second difference is that Rubinius uses a different name for assignment node. The MRI parse tree has also a greater depth, because MRI uses wrapping nodes, which contain only one child node and gives no additional information.

From the previous analysis we can see that the parse trees are convertible. I have created a testing tool that traverses both trees simultaneously and compares nodes according to their type.

It is not possible to create a comparator, which does not need to take the node type into account. Therefore we need to create a comparing mechanism for every node type, where each node type has many different forms based on the specific language construct. The testing tool is a part of the future work on this project, because the number of situations which need to be tested individually is really high and cannot be generated automatically.

The result percentage from all projects together using the current implementation is 2.63 %. There is no meaningful way to compare the parse trees partially, therefore the results of the comparator are boolean. This fact significantly decreases the percentage of successfully validated files, because the major part of Ruby files are quite long and use many different language constructs.

6.2 Performance

Another important measure of program quality is its performance. We need to know how long it will take to parse a file of given parameters. In the grammar there are many language constructs of Ruby and each of them has is tokenized and processed by the grammar differently taking a different amount of time.

As an example, we look at processing of two tokens, a comma and a plus sign. A comma is used as a parameter separator for example in both function declaration and function call. When the tokenizer finds a comma character in the input, it does not have to do any other decisions and returns a comma token. When a comma token appears in the look-ahead token, the syntax analyzer almost always chooses to shift, which takes much less effort then to for example reduce.

A plus sign, unlike a comma, is much more difficult for the tokenizer to process. The first thing is the tokenizer state. The tokenizer goes through a number of different states and behaves according to the current state. If a plus sign appears after an operator, it can be followed by either `@`, `=` or other character. It can be occurence of a `+=` operator, an instance variable, which is in Ruby prefixed with `@`, or a plus sign. The last variant has two forms: an unary plus and a binary plus. A syntax analyzer also must be able to handle all these possibilities.

Despite the different time demands of different language constructs, the following performance tests will be based on a number of tokens in the testing

token count	MRI time	new Ruby parser time	ratio
250	0.034877701	0.229	0.1523043712
500	0.212531985	0.453	0.4691655298
750	0.370353954	0.703	0.5268192802
1000	0.536990125	0.915	0.5868744536
1250	0.722713367	1.107	0.6528576034
1500	0.884904387	1.135	0.7796514423
1750	1.04379763	1.574	0.6631497014
2000	1.218045643	1.760	0.6920713881

Table 6.2: Average MRI and new Ruby parser evaluation time in seconds to number of tokens

source code.

I will measure time needed for the MRI to create a parse tree from testing set of input files. Then parse the same set of input files with the new Ruby parser and compare data with those provided by MRI. For all testing input files, I will use language constructs, which have similar time demands to parse. Every input file will be evaluated five times and the average value will be the result.

Time of the new Ruby parser will be measured with built-in Smalltalk/X method `Time millisecondsToRun:`. To measure time of MRI, I will use two Ruby gems: *Benchmark* to measure elapsed time and *Ripper* to retrieve a parse tree of the MRI result. Because Smalltalk's minimal time unit is a millisecond, the differences between time measuring tools are negligible. [22][8]

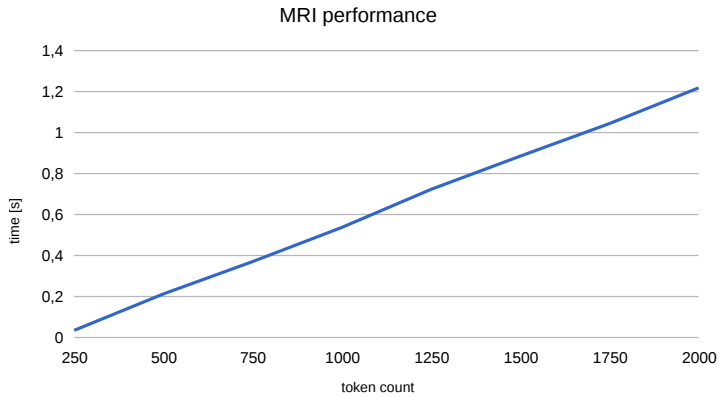

Figure 6.1: Time required by MRI to create a parse tree

First, look at figure 6.1. According to the graph, the time elapsed by MRI for parsing a source code increases linearly with the number of tokens to parse.

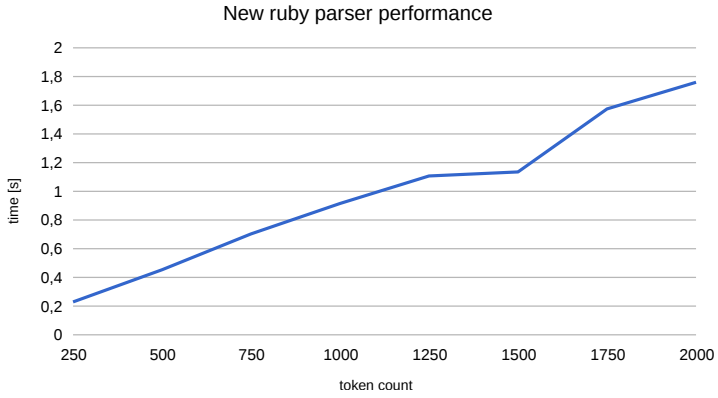

Figure 6.2: Time required by the new Ruby parser to create a parse tree

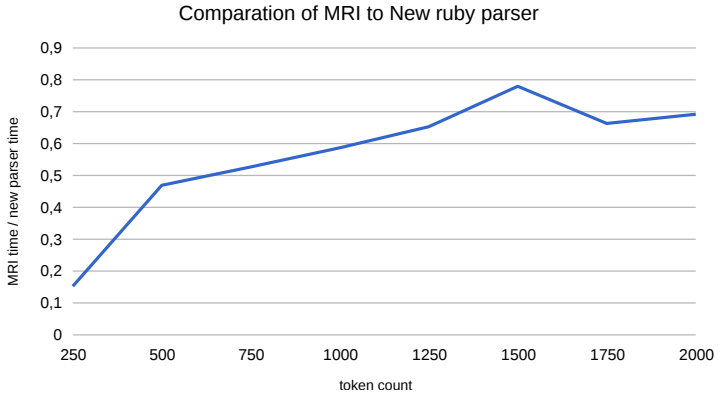

Figure 6.3: MRI and New ruby parser time ratio

All LR(n) parsers really do have a linear complexity depending on the number of tokens.

For a tokenizer, every token is processed individually, therefore its complexity is $O(n)$, where n is the number of tokens. And the syntax analyzer does not do any parse tree scans, it performs a predefined number of steps. The number of steps is dependant on the context, where each possible situation is defined during creation of the parser and has a constant number of steps. The syntax analyzer is neither dependant on the number of tokens to be parsed. Therefore the whole parser has linear time complexity. [18]

Now we will compare the MRI results with those of the new Ruby parser. According to the figure 6.2 the new Ruby parser has also a linear time dependence on the number of tokens. The curve is almost a straight line except of the point 1500. With 1500 tokens to parse I measured repeatedly a lower value

than expected, but this deviation is most probably an inaccuracy in measurment. The MRI graph does not happen to have this deviation. However with 1750 tokens the time value fits the straight line again.

The third graph shows ratio of MRI time and new Ruby parser time. We can see that the ratio converges to neighborhood of 0.7. This means that the new Ruby parser, according to the selected input data sets, elapses only 1.4 times MRI time. The last graph does not show a constant function as expected. This is caused by initialization procedures.

The new Ruby parser performance can be improved in many ways. For example the tokenizer can be rewritten in C. Because the whole syntax analyzer is generated by STCC, it is very easy to change its parts on one place and let the STCC regenerate the whole syntax analyzer.

6.3 Future work

The new Ruby parser is a perspective project with a wide variety of possibilities for future development. It is crucial to continue in the lexical analyzer development. There are still language constructs, which are not successfully recognized. This task is connected with the STCC tool, which can be extended as well.

Another feature, which significantly improves the impression of the parser is failover. A failover mechanism enables the parser to continue parsing even after a syntax error occurence. This feature would also increase the number of the parsed files. The new Ruby parser also needs to be integrated with SmallRuby.

The fact that the new Ruby parser is only 1.4 times slower than the reference implementation of Ruby is a good result. But the performance may still be improved.

In some cases, the tokenizer uses exceptions to control the program flow. Because Smalltalk exceptions are a powerful tool with a lot of options, it also significantly slows down the execution of the program. Removing of these exceptions can help to improve performance of the parser.

The speed of the syntax analyzer can be also improved. Smalltalk does not provide functionality similar to switch construct known from other programming languages. Therefore it must be substituted with an equivalent construct.

Smalltalk/X allows to write parts of code directly in C++. Rewriting of the core parts of the parser into C/C++ can also increase execution speed.

Conclusion

In this thesis we described a new Ruby parser written in Smalltalk. We described high level features of the Ruby language, with focus on syntactical structure. The research continues with an overview of currently most used Ruby implementations.

I have chosen Rubinius to be most suitable for the new Ruby parser, mainly because its tokenizer and grammar are not so much dependent on the other parts of the interpreter.

The design of the new Ruby parser is derived from Rubinius. Because almost every Ruby grammar is described by Bison format, I highlighted some of Yacc's and Bison's features in a separate chapter.

Every modern parser has two parts: a lexical analysis and a syntax analysis. I have created a hand-written tokenizer, which turns the input source code into terminal symbols, which can be later used in the syntax analysis.

Instead of creating a lexical analyzer by hand, I have created the Smalltalk Compiler-Compiler (STCC), a tool for generating syntax analyzers from grammars in the Bison format. This tool can be used to generate a parser from many grammars, not only from the Ruby grammar.

I have created STCC for many reasons. First, automated processing of a grammar eliminates the human factor. Also it has much higher time requirements to create a syntax analyzer by hand than to create an automated tool. And after finishing the analyzer there will surely be released a new version of Ruby and it would be very complicated to integrate new features. With STCC it is possible to directly modify the grammar and let the analyzer be regenerated automatically.

The tool is also very flexible and can be adapted to different types of grammar, to create a parser using a different parsing technique or even to generate a parser in a different programming language than Smalltalk.

I have created an extensive infrastructure, which is needed by both lexical and syntax analyzers to work properly. The abstract syntax tree builder by both tokenizer and syntax analyzer. The tokenizer to create leaf nodes

which represent terminal symbols including position in the file and the syntax analyzer to combine leaf nodes into more complex node hierarchy.

Testing is also an essential part of software development. I used test-driven development throughout the project. The tokenizer has test cases for almost every input code sequence based on the Rubinius source code. The STCC and syntax analyzer functionality is checked by a set of test cases as well.

The syntax analyzer has also a testing infrastructure which compares the new Ruby parser output with the AST of MRI. The AST of MRI and Rubinius (which is used by the new Ruby parser) are not the same, however they are convertible. The testing infrastructure takes these differences into account and verifies the output of the new Ruby parser against MRI.

I have also created performance analysis, which compares speed of the new Ruby parser with the speed of MRI. According to the testing results, the new Ruby parser is approximately only 1.4 times slower than MRI and still offers a lot of ways, how to improve its speed.

Bibliography

[1] C++ goto statement. 2015, [Online; 30 March 2015]. Available at: http://www.tutorialspoint.com/cplusplus/cpp_goto_statement.htm

[2] Exceptions — GNU Smalltalk. 2015, [Online; 31 March 2015]. Available at: http://smalltalk.gnu.org/wiki/exceptions

[3] Java — exception handling. 2015, [Online; 31 March 2015]. Available at: http://www.tutorialspoint.com/java/java_exceptions.htm

[4] What is goto statement in C language. 2015, [Online; 30 March 2015]. Available at: http://www.webcodeexpert.com/2013/08/what-is-goto-statement-in-c-languageits.html

[5] About JRuby. 2016, [Online; 9 January 2016]. Available at: https://github.com/jruby/jruby/wiki/AboutJRuby

[6] About Ruby. 2016, [Online; 10 January 2016]. Available at: http://www.ruby-lang.org/en/about/

[7] Array in Ruby. 2016, [Online; 10 January 2016]. Available at: http://ruby-doc.org/core-1.9.3/Array.html

[8] Benchmark. 2016, [Online; 13 February 2016]. Available at: http://ruby-doc.org/stdlib-1.9.3/libdoc/benchmark/rdoc/Benchmark.html

[9] Bison 3.0.4. 2016, [Online; 31 January 2016]. Available at: http://www.gnu.org/software/bison/manual/bison.html

[10] Bison Java Parsers. 2016, [Online; 9 January 2016]. Available at: http://www.gnu.org/software/bison/manual/html_node/Java-Parsers.html

[11] Bottom-Up Parsing. 2016, [Online; 8 February 2016]. Available at: http://dragonbook.stanford.edu/lecture-notes/Stanford-CS143/08-Bottom-Up-Parsing.pdf

[12] A Brief History of Just-In-Time. 2016, [Online; 9 January 2016]. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.97.3985&rep=rep1&type=pdf

[13] Compiler Design – Syntax Analysis. 2016, [Online; 20 January 2016]. Available at: http://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm

[14] Differences Between MRI And JRuby. 2016, [Online; 9 January 2016]. Available at: https://github.com/jruby/jruby/wiki/DifferencesBetweenMriAndJruby

[15] IBM: General Programming Concepts: Writing and Debugging Programs. 2016, [Online; 8 February 2016]. Available at: https://www-01.ibm.com/support/knowledgecenter/ssw_aix_53/com.ibm.aix.genprogc/doc/genprogc/yacc_ambiguous_rules.htm%23ivj4280gaco

[16] JRuby Internal Design. 2016, [Online; 10 January 2016]. Available at: https://github.com/jruby/jruby/wiki/JRubyInternalDesign

[17] Kaleidoscope: Implementing a Parser and AST. 2016, [Online; 3 January 2016]. Available at: http://llvm.org/docs/tutorial/LangImpl2.html

[18] Language Processors. 2016, [Online; 25 January 2016]. Available at: http://www.iis.ee.ic.ac.uk/yiannis/lp/LPLecture11bw.pdf

[19] Matz Ruby developers and RubySpec. 2016, [Online; 10 January 2016]. Available at: http://rubinius.com/2014/12/31/matz-s-ruby-developers-don-t-use-rubyspec/

[20] ML-Yacc User's Manual. 2016, [Online; 8 February 2016]. Available at: https://www.cs.princeton.edu/~appel/modern/ml/ml-yacc/manual.html

[21] Modular Programming. 2016, [Online; 25 January 2016]. Available at: http://c2.com/cgi/wiki?ModularProgramming

[22] Ripper. 2016, [Online; 13 February 2016]. Available at: http://ruby-doc.org/stdlib-2.0.0/libdoc/ripper/rdoc/Ripper.html

[23] The role of lexical analyzer. 2016, [Online; 3 January 2016]. Available at: http://www.univ-orleans.fr/lifo/Members/Mirian.Halfeld/Cours/TLComp/l3-0708-LexA.pdf

[24] RSpec: Behaviour Driven Development for Ruby. 2016, [Online; 8 February 2016]. Available at: `http://rspec.info/`

[25] The Rubinius Language Platform. 2016, [Online; 6 January 2016]. Available at: `https://github.com/rubinius/rubinius/blob/master/README`

[26] Rubinius Ruby Parser – GitHub. 2016, [Online; 24 January 2016]. Available at: `https://github.com/rubinius/rubinius-melbourne`

[27] Yacc: Yet Another Compiler–Compiler. 2016, [Online; 2 January 2016]. Available at: `http://dinosaur.compilertools.net/yacc/`

[28] Grune, D.; Jacobs, C. J. H.: *Parsing Techniques – A Practical Guide*. Ellis Horwood, Chichester, England, 1990, ISBN 0-13-651431-6.

[29] Hopcroft, J. E.; Motwani, R.; Ullman, J. D.: *Introduction to Automata Theory, Languages, and Computation*. Delhi: Pearson Education, 2000, ISBN 81-7808-347-7.

[30] Pecinovský, R.: *Návrhové vzory*. Computer Press, 2007, ISBN 978-80-251-1582-4.

[31] Streib, J. T.: *Guide To Assembly Language: A Concise Introduction*. Springer Science & Business Media, 2011, ISBN 978-0-85729-271-1.

Appendix A

List of abbreviations

AST Abstract Syntax Tree

BU parser Bottom Up parser

TD parser Top Down parser

ST Smalltalk

STX Smalltalk/X

STCC Smalltalk Compiler-Compiler

PS PostScript

MRI Matz's Ruby Interpreter or Ruby MRI

Appendix B

Example state diagram

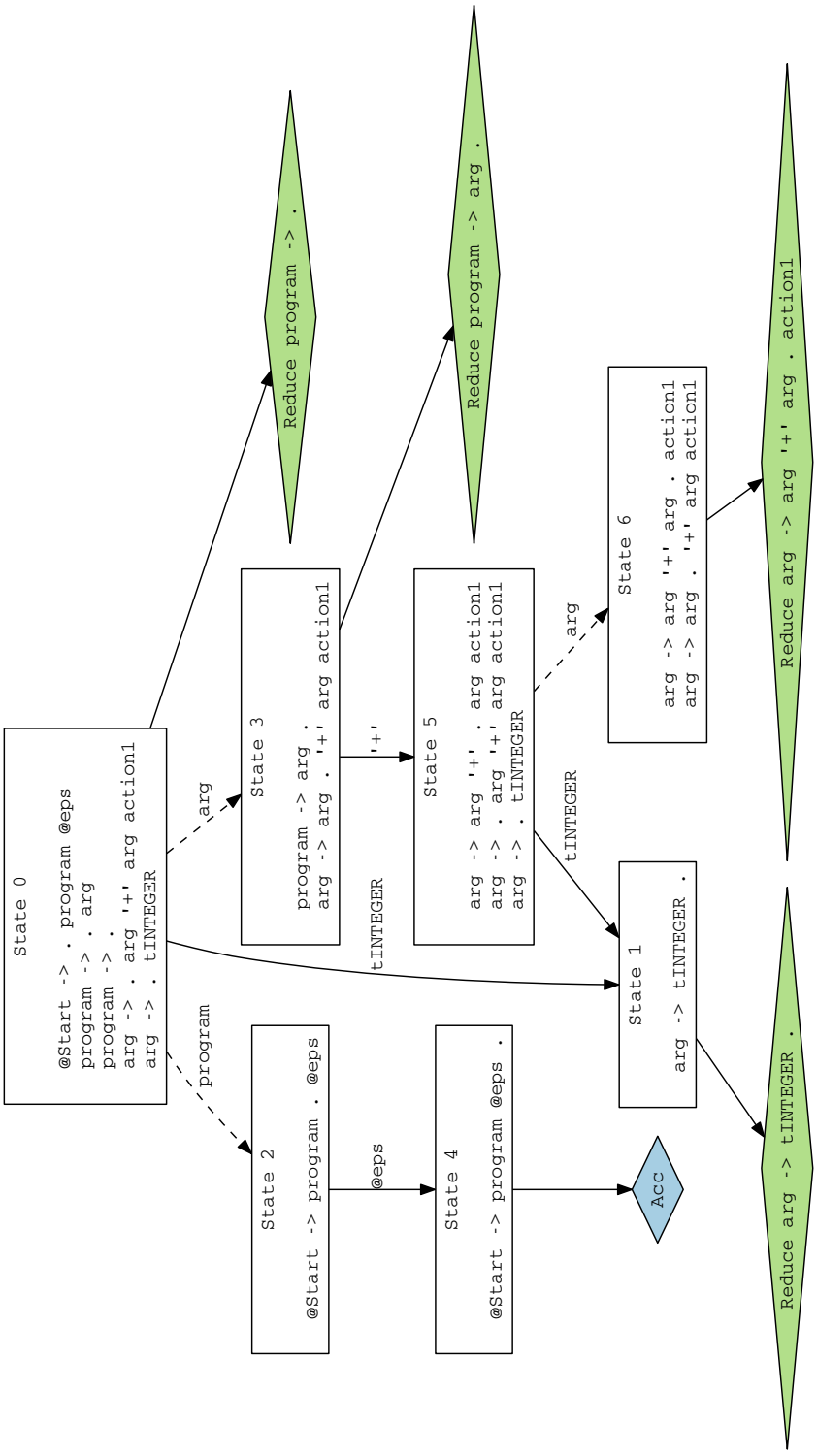

Figure B.1: State diagram generated by STCC

Input: $1 + 2 + 3$
Parsing steps:

```
0: Shift 1
1: default - not found +
1: Reduce arg -> tINTEGER .
0: Goto 3

3: Shift 5

5: Shift 1
1: default - not found +
1: Reduce arg -> tINTEGER .
5: Goto 6
6: default - not found +
6: Reduce arg -> arg '+' arg . action1
0: Goto 3

3: Shift 5

5: Shift 1
1: default - not found EOF
1: Reduce arg -> tINTEGER .
5: Goto 6
6: default - not found EOF
6: Reduce arg -> arg '+' arg . action1
0: Goto 3
3: default - not found EOF
3: Reduce program -> arg .
0: Goto 2

2: Shift 4
4: default - not found EOF
4: Accept

result: 6
```

Appendix C

Content of CD

- `readme.txt` ..... a brief description of the CD
- `src`
  - `comparator` ..... AST structure comparator in Ruby
  - `stcc` ..... STCC tool source code in Ruby
  - `parser` ..... parser source code in Smalltalk/X
  - `thesis` ..... source code of the thesis in LaTeX format
- `text` ..... thesis text
  - `thesis.pdf` ..... thesis in PDF format