
Pós-Graduação em Ciência da Computação

Clayton Wilhelm da Rosa

# A Combinator Based, Certifiable, Parsing Framework

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Clayton Wilhelm da Rosa

# A Combinator Based, Certifiable, Parsing Framework

Dissertação apresentada ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

**Área de concentração**: Linguagens de Programação e Engenharia de Software.

**Orientador**: Prof. Dr. Márcio Lopes Cornélio.

Recife
2019

**Catalogação na Fonte**
Bibliotecário Vimário Carvalho CRB4/1204

R788c Rosa, Clayton Wilhelm da.
A Combinator based, certifiable, parsing framework / Clayton Wilhelm da Rosa. - 2019.
100 f.: il., fig.

Orientador: Prof. Dr. Márcio Lopes Cornélio.
Dissertação (Mestrado) - Universidade Federal de Pernambuco. CIN, Ciência da Computação. Recife, 2019.
Inclui Referências e apêndices.

1. Linguagem de programação. 2. Engenharia de software. 3. Software confiável. I. Cornélio, Márcio Lopes (orientador). II. Título.

005.13 CDD (22. ed.) UFPE-MEI 2019-136

**Clayton Wilhelm da Rosa**

**“A Combinator Based, Certifiable, Parsing Framework”**

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 28 de agosto de 2019.

**BANCA EXAMINADORA**

_____

Prof. Dr. Gustavo Henrique Porto de Carvalho
Centro de Informática/UFPE

_____

Prof. Dr. Rodrigo Geraldo Ribeiro
Departamento de Computação e Sistemas / UFOP

_____

Prof. Dr. Márcio Lopes Cornélio
Centro de Informática/UFPE
(**Orientador**)

## ABSTRACT

Parsers are ubiquitous software, much more common than one would normally take notice. Parsing spreads from simple command line functionalities to natural languages processing, to language composition. Parsing is also somewhat regarded as a solved problem in computation. However, that does not translate into reality, especially when considering their implementations, which can be complex and difficult to maintain. In the last decades, multiple tools have surged aiming to improve the process of parsing, from the now well established parser generators to recent interactive parsing frameworks, which try to reduce the knowledge requirements for the specification of parsers. Although these tools have their own merits, very little effort was put into their standardization and formal reliability. We try to address these issues by implementing a reliable and flexible parsing framework that is composed of a small and extensible library of parser combinators, and a reliable, easily verifiable, parser generator based on the standardized meta-syntax of the extended Backus-Naur notation. We also provide valuable insight into the implementation of the General LL parsing technique in a purely functional setup.



**Keywords**: Parsing. Functional programming. Software reliability.

## RESUMO

*Parsers* são softwares muito mais comuns do que normalmente nos damos conta. *Parsers* estão presentes nas mais diversas áreas, no processamento de linhas de comando, no processamento de linguagens naturais, ou ainda na composição de linguagens. O processo de *parsing* é considerado por muitos um problema já solucionado, porém isto não é inteiramente verdade, especialmente quando falamos das implementações de *parsers* que podem ser complexas e de difícil manutenção. Nas últimas décadas, muitas ferramentas que buscam facilitar o processo de *parsing* surgiram. Ferramentas como geradores de *parsers*, ou mais recentemente, *frameworks* interativos, que tentam reduzir a quantidade de conhecimento necessária para a especificação de *parsers*. Ainda que estas ferramentas tenham seus méritos, estas também apresentam algumas limitações. Estas ferramentas apresentam pouca ou quase nenhuma padronização entre si, além de não oferecerem garantias de confiabilidade. Nós buscamos mitigar estes problemas com a implementação de um *framework* para *parsing*, confiável e flexível. O *framework* é composto de uma biblioteca extensível de combinadores, e de um gerador de *parsers* que é facilmente verificável, e que se baseia na meta-sintaxe padrão da notação estendida de Backus-Naur. Além disso, nós apresentamos informações valiosas sobre a implementação do algoritmo GLL, sob uma perspectiva puramente funcional.



**Palavras-chave**: *Parsing*. Programação funcional. Software confiável.

## LIST OF FIGURES

## LIST OF TABLES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| **ADT** | Algebraic Data Type |
| **BNF** | Backus-Naur Form |
| **CFG** | Context-Free Grammar |
| **CFL** | Context-Free Language |
| **CPS** | Continuation-Passing Style |
| **DSL** | Domain-Specific Language |
| **EBNF** | Extended Backus-Naur Form |
| **FL** | Functional Language |
| **GLL** | Generalized LL |
| **GLR** | Generalized LR |
| **GSS** | Graph-Structured Stack |
| **NL** | Natural Language |
| **PEG** | Parsing Expression Grammar |
| **PL** | Programming Language |
| **RD** | Recursive Descent |
| **RegEx** | Regular Expression |
| **RG** | Regular Grammar |
| **RL** | Regular Language |
| **RRG** | Right Regular Grammar |

## CONTENTS

# 1 INTRODUCTION

> *If you think it is simple, then you probably misunderstood the problem.*
>
> Bjarne Stroustrup, *1997*

Software has become a ubiquitous element of our daily lives. In fact, without even noticing we, in one way or another, rely on software to fulfill just about every task. We rely on software for communication, transportation, paying bills, and the list goes on. While we take the availability and reliability of software for granted, more often then what should be expected we are remembered otherwise.

For many domains such as health care, automotive industry, and information security, errors and defects in the development of an application can lead to serious financial losses, or even threaten the users' lives. These so-called critical domains require a rigorous verification of their systems to provide reliable software.

Normally such verification is accomplished by the exhaustive analysis of the states and transitions of idealized software models. However, the analysis of bigger programs by such technique is limited by the growth in the number of states of the software models (1). Also, such idealized models neither represent the application code, where implementation errors can impair the software reliability (2), nor can detect defects introduced by external agents such as compilers and interpreters (3, 4).

For the majority of ordinary software, defects introduced by interpreters or compilers will cause little harm, specially when compared to the defects related to erroneous implementation (5). Critical software on the other hand should not neglect this sort of defects, since they can nullify any reliability guarantee given by techniques such as model verification and static analysis (5).

For this reason, various research projects are trying to achieve precise formalizations of semantics and implementations of Programming Languages (PLs) (4, 5, 6) with the intent of improving critical software reliability in a more fundamental way. Even so, a fundamental component of any PL development, the parser, is sometimes neglected by these projects, which many times depend on non-reliable third party parsers. However, the same projects point out that the absence of a formally verified parser contributes for less reliable results (4, 7).

Let's take a step back from critical software and allow ourselves to contemplate parsing from a less strict and demanding perspective. It is very likely that programmers overlook the fact that parsers are more common than one would expect. Parsing spreads from a simple AWK one-liner, to a JSON or markup library for any PL, to the processing of Domain-Specific Languages (DSLs) and Natural Languages (NLs), to a tool for language

composition, domain from where we borrow the very fitting sentence "Parsing: The Solved Problem That Isn't"[1].

Parsing has a long research history, with refined techniques and seen by many as a solved problem. However, some works (8, 9) argue that the reality is different, specially when considering the implementation of parsers, which can be complex and have a costly maintenance. The last decades popularized the use of parser generators as a tool to facilitate the build and maintenance of these software. Parser generators take a Backus-Naur Form (BNF) like high-level specification of a grammar and synthesize a parser for the given grammar. This way, in theory, we would only concentrate on specifying the syntax of a language.

Even though largely adopted in production, parser generators do not come without some pitfalls. Parser generators rely on their own particular syntax for the specification of grammars, mostly without concern for compatibility or standardization. Also, they require some knowledge of how their underlying parsing algorithm works, largely because of the limitations of parsing techniques, a classical example is the necessity for factorization of left-recursive grammars. At last, parser generators do not provide any formal guarantees of their reliability.

Most recently, various works (8, 10, 11, 12) have put a great effort towards the implementation of more "accessible" parser generators and frameworks, aiming to reduce the knowledge requirements of theories such as formal languages, making the user distant from the parser inner workings. This is achieved in two major ways. First, they provide tools that help the user solve common recurrent problems, such as ambiguity detection and resolution. Second, a more general solution, is the application of general parsing algorithms, which can cope with any grammar specified by the user.

In the light of initiatives such as the aforementioned, we aim to provide an easy to verify, compact, and flexible, parsing framework. A solution that critical and "regular" software projects would benefit from. The framework is composed by an intuitive parsing combinators library and a parser generator that synthesizes parsers based on a standardized, simple, input meta-syntax.

## 1.1 CONTRIBUTIONS

Before listing the contributions of this work we must present a disclaimer. This work originally had more ambitious objectives, it was, in fact, supposed to implement a parser generator that would synthesize fully general parsers, what would have been one of the first adaptations of the recently discovered Generalized LL (GLL) (13) parsing technique in a purely functional setup. Alas, the intrinsic imperative nature of the algorithm im-

[1] The title of an article written by Laurence Tratt on the use of parsers in the context of language composition. The article can be found at <https://tratt.net/laurie/blog/entries/parsing_the_solved_problem_that_isnt.html>.

paired us from achieving our goal, for now. An in depth discussion of the design decisions and adversities encountered along this project execution will be given throughout this work. The main objective of this work is to provide a reliable and easily certifiable parsing framework. Recognizing that the term certifiable may lead to misunderstandings, we clarify. The development of this work focus on establishing a reliable foundation that will allow/facilitate a formal verification of its implementation.

We have two specific objectives.

- To generate parsers capable of recognizing a reasonable subset of Context-Free Languages (CFLs). The definition and manipulation of grammars must be simple and standardized. Also, the generated parser code must be human readable, easy to maintain and reason about.
- To explore the adaptation and implementation of the GLL parsing algorithm on the purely Functional Language (FL) Haskell, and provide valuable insight for future works.

## 1.2 OUTLINE

This work is organized as follows. In Chapter 2, we review the main concepts on formal languages, principles regarding the parsing of these languages. We also and introduce a standard notation for specifying syntax, and cover the main concepts of the language we chose for this work implementation. Chapter 3 covers the implementation of combinators and provides insight into our efforts trying to adapt our combinators to the GLL technique. In Chapter 4, we elaborate on the organization, limitations, and reliability of our parser generator. We follow with the presentation and discussion on validation of our solution. We present our conclusions, discuss related research, and propose future work in Chapter 5.

# 2 LANGUAGES, PARSING, AND TOOLS

> *Language shapes the way we think, and determines what can we think about.*
>
> Benjamin Lee Whorf,

Although very familiar, the term language is difficult to define. From the study of speech perspective, one could say that a language is an auditory and motor system for the communication between individuals. Dictionaries broadly define a language as "a system of sounds, words and rules for communication". In this work when we refer to a language we do so from the formal languages perspective, where the definition is as follows (14, 15).

**Definition 2.0.1: Language**

A set of strings (or words) over a set $\Sigma^*$, where $\Sigma^* = \{\varepsilon\} \cup \Sigma$.

The symbol $\varepsilon$ represents the empty string, and $\Sigma$ is the language alphabet. An alphabet is a finite, non-empty set of symbols. For most occidental languages $\Sigma$ would be some variation of the Latin alphabet, for Japanese the set of *kana*, and binary words are build over $\Sigma = \{0, 1\}$.

At this point, most textbooks on formal languages would present a set of concepts such as the length of a word, concatenation, pre-, su-, and infixes. However, as they will not contribute for the understanding of this work, they will not be covered here. That said, a concept is missing.

If a language is a set of strings, how do we build those strings? Even if a word, in any human language, can ultimately be seen as an arbitrary displacement of the symbols of its alphabet, that is not true for all of what we may want to write. A real number, for example, cannot be written as "1234," (with a comma, but no real fragment). Taking this idea a step further, what about sentences? In most human languages one cannot just arrange words randomly in a sentence and expect it to make any sense. The same is true in a PL, we cannot write any sequence of words and expect it to be a program.

What is implied above is that we require a set of rules on how to structure our languages. There are many types of languages, with varying expressiveness, and in Sections 2.1 and 2.2 we present some of them and their main concepts, such as the mentioned structuring rules.

## 2.1 REGULAR LANGUAGES

Normally, the formal definition of Regular Languages (RLs) require the definition of what is a finite automaton, and how it works, but since we do not directly deal with automata

in this work, for conciseness, we will not present such concepts. Despite that, the definition of an RL is given bellow (14, 15).

**Definition 2.1.1: Regular Language**

A language $L$ is regular *iff* there exists a finite automaton $A$ such that $L = L(A)$, where $L(A)$ is the set of all words accepted by $A$.

Even though definition 2.1.1 relies on the automata theory, it does not lessen the understanding of this work, since, at this point, we are more interested on how the structure of an RL can be represented, than on how it can be recognized.

### 2.1.1 Regular Expressions

Regular Expressions (RegExs) are algebraic mechanisms for defining RLs, the words/lexemes we recognize as part of our languages (14, 15). RegExs are simple compositions of the symbols in a given alphabet and three operators '+', '*', and juxtaposition[1]. parentheses can also be used to group sub-expressions, in similar fashion to when we write arithmetic expressions. The precedence of the operators is, from highest to lowest: '*', juxtaposition, and '+'.

Operations on RegExs have and interesting correspondence to set operations on RLs. Given $r$ and $r'$, RegExs, this correspondence is described in Table 1.

Table 1 – RegEx operations to set operations correspondence.

| | **RegEx** | **Set** |
|---|---|---|
| 1 | $L(r + r')$ | $L(r) \cup L(r')$ |
| 2 | $L(rr')$ | $\{s_r s_{r'} \mid s_r \in L(r) \wedge s_{r'} \in L(r')\}$ |
| 3 | $L(r^*)$ | $\bigcup_{i=0}^{n} L^i(r)$ |

This correspondence is not particularly important for this work, we present it and do not look back at it. What should be noticed is that RegExs and sets present two different intuitions. RegExs show us **how**[2] to recognize (build) the words of a language, while sets tell us **what** words are part of a language. A similar phenomenon holds between grammars and sets, as well.

In line 1 (Table 1) the RegEx says "match either $r$ or $r'$", while its corresponding set states "the valid words of the language defined by $/r + r'/$ are the valid words of the language defined by $r$, and also the valid words of the language defined by $r'$". In line 2

[1] Often times this operator is called the dot operator '·', however when defining RegExs this operator is not actually written down, therefore we use the juxtaposition.

[2] RegExs are definitions/specifications of RLs, but we can read a RegEx this way "match a 0 followed by a 1, then match either any number of 0's or any number of 1's, for $/01(0^* + 1^*)/$. This is a sequential description of **how** to recognize the words of a language.

we have "match $r$ followed by $r'$", and the corresponding set definition is similar to a Cartesian product, but instead of pairs it is composed of the juxtaposition of the valid words in $L(r)$ with the valid words in $L(r')$. Finally, (3) represents repetition, it specifies $n$ juxtapositions ($n \geq 0$) of the string defined by $r$; its corresponding set is the union of all sets of words of size $i$ ($L^0(r) = \{\varepsilon\}$), which are juxtapositions of $r$.

For a better comprehension of what and how RegExs work, in Table 2 we give a RegEx $r$, a textual description of what language it defines, and examples of valid words for $L(r)$.

Table 2 – RegEx examples.

| **RegEx** | $L(r)$ | **Examples** |
|---|---|---|
| $000 + 111$ | The words 000 and 111[3]. | 000, 111 |
| $01(0^* + 1^*)$ | All words that have any number of 0's, or any number of 1's, prefixed by 01. | 01, 010000, 011 |
| $0 + 1(0 + 1)^*$ | The Language of all binary numbers [4]. | 0, 1, 0001, 010101 |
| $01^*$ | All words with an arbitrary number of 1's prefixed by a single 0[5]. | 0, 01, 011...1 |
| $(0 + 1)^*00(0 + 1)^*$ | Any word that has at least one occurrence of consecutive 0's. | 000, 001, 100, 1001010 |

### 2.1.2 Regular Grammars

Grammars are a mathematical mechanism for describing languages, formal languages. They tell us if a word or sentence is well-formed in a certain language (15). A grammar translates verbal rules such as "a sentence is composed by a subject, followed by a predicate" into a formal structure of the form.

$$SENTENCE \rightarrow SUBJECT\ PREDICATE$$
$$SUBJECT \rightarrow ARTICLE\ NOUN$$

Each line of this structure is called a **rule** or **production**. To the left of the arrow symbol, left-hand side, we define the syntactical category, also called **variable** or **non-terminal**, $SENTENCE$. The arrow itself is a separator called **production symbol**, it indicates the definition of a rule. On the right-hand side of the production symbol, there is a sequence of $n \geq 0$ syntactical elements, which compose a $SENTENCE$, in this example, $SUBJECT$ and $PREDICATE$ (14).

[3] RegExs 000 and 111 are examples of juxtaposition of three symbols, 0 or 1.
[4] This RegEx illustrates a very common pattern $/rr^*/$, which exists for grammars as well, and will have its own operator defined in Section 2.4.
[5] Note that the '*' operator applies only to the very first RegEx to its left, in this case 1, and not 01.

**Definition 2.1.2: Grammar**

Is a quadruple $G = (V, T, S, P)$, where
$V$ is a finite set of nonterminals,
$T$ is a finite set of symbols called **terminals**,
$S \in V$ is the so called **start variable**, and
$P$ is a finite set of rules.

Formally a grammar is defined as we see in definition 2.1.2. If we were to define our sentence grammar formally, we could have $V = \{SENTENCE,\ SUBJECT,\ ARTICLE,\ NOUN,\ PREDICATE\}$, $T = \{a, the\}$ if $ARTICLE$ was to be defined as

$$ARTICLE \rightarrow\ a \mid the$$

$a$ and $the$ are "constant" elements of the grammar and do not label rules, they exist by themselves, while nonterminals are defined in terms of other elements of the grammar. $S = SENTENCE$, which on a top-down, left-right reading of the grammar is the first rule to be defined. Finally our set of rules would look like as follows.

$$P = \left\{ \begin{array}{rl} SENTENCE \rightarrow & SUBJECT\ PREDICATE, \\ SUBJECT \rightarrow & ARTICLE\ NOUN, \\ ARTICLE \rightarrow & a, \\ ARTICLE \rightarrow & the, \\ NOUN \rightarrow & \ldots, \\ PREDICATE \rightarrow & \ldots, \\ PREDICATE \rightarrow & \ldots \end{array} \right\}$$

Before we move on to the specifics of Regular Grammars (RGs), we discuss grammar operators. In Table 3, we present the two basic grammar operators.

Table 3 – Grammar operators.

| | |
|---|---|
| $a \mid b$ | Alternative or choice |
| $a\ b$ | Sequence or juxtaposition |

First, the alternative operator indicates that a rule will hold for any sequence of elements in $(V \cup T)^*$ separated by '|' in its right-hand side. This operator is actually an abbreviation, so we can, for example, write the article production of the sentence grammar as given by 2.2, instead of 2.1.

$$\begin{array}{l} ARTICLE \rightarrow\ a \\ ARTICLE \rightarrow\ the \end{array} \qquad (2.1) \qquad\qquad ARTICLE \rightarrow\ a \mid the \qquad (2.2)$$

The alternative operator corresponds to the '+' operator used by RegExs. The article rule would be $/a + the/$ if defined as a RegEx. The article production also illustrates the use of the sequence operator, which relates to juxtaposition in RegExs.

Consider the terminal *the*, the second alternative of $ARTICLE$ (2.2). This terminal can be seen as the juxtaposition of three other symbols, $t$, $h$, $e$, and so the article rule could be written as.

$$ARTICLE \rightarrow a \mid t\ h\ e$$

Notice that the sequence operator has greater precedence, the alternative operator extends, to its left and right, as far as a sequence goes, this means it will **not** match something like $/(a \mid t)\ h\ e/$. In fact, the use of parentheses as grouping mechanism is not allowed in basic grammar notation (see Section 2.4).

The '*' operator does not have a grammar corresponding. At least, not a direct corresponding symbol in the grammars' syntax (again, we defer to Section 2.4). Still, grammars are capable of representing the same set of languages by means of recursion.

As we have seen, '*' represents repeated juxtaposition of the RegEx immediately to its left. Now, remember the binary number RegEx from Table 2, we present a corresponding grammar (2.3), where we can clearly see the same $/rr^*/$ repetition pattern, with the $DIGITS$ rule being the recursive corresponding of $r^*$.

$$\begin{aligned} BNUM &\rightarrow\ DIGIT\ DIGITS \\ DIGITS &\rightarrow\ DIGIT\ DIGITS \mid \varepsilon \\ DIGIT &\rightarrow\ 0 \mid 1 \end{aligned} \tag{2.3}$$

This correspondence allows us to rely on a single mechanism for recognition of the whole spectrum of languages we are interested in. Even if this correspondence was discussed in terms of a more general concept of grammar, it will become clear that it holds for the more restrict grammars, which we will work with.

Now that we covered the main concepts regarding grammars, what is an RG? As the name implies it is a grammar and therefore all that has been introduced so far holds for RGs, except for restrictions to its structure, which makes them regular. In definition 2.1.3 we formalize the concept of an RG.

**Definition 2.1.3: Regular Grammar**

A grammar $G = (V, T, S, P)$ is regular if all its productions are either

| **left-linear** | | **right-linear** |
|---|---|---|
| $A \rightarrow Bx,$ | or | $A \rightarrow xB,$ |
| $A \rightarrow x.$ | | $A \rightarrow x.$ |

where $A, B \in V$ and $x \in T^*$.

Definition 2.1.3 states that any right-hand side of an RG can have at most one nonterminal, and it must be either the left-most or right-most element of a sequence, otherwise a rule can only be defined as a sequence of $n \geq 0$ terminals (15).

In this work we use RG as a synonym of Right Regular Grammar (RRG), which are right-linear grammars, their productions closely resemble RegExs and this resemblance actually translates into an interesting theoretical result that states "RRGs generate RegExs" (15), what reinforces our assertions regarding the correspondence between RegExs and grammars, in particular RGs.

To conclude this section, we recall grammar 2.3, which, as it stands, is not regular, and, to illustrate the structures of RGs, we show a regular version of it.

$$\begin{aligned} BNUM &\rightarrow\ 0\ DIGITS \mid 1\ DIGITS \\ DIGITS &\rightarrow\ 0\ DIGITS \mid 1\ DIGITS \mid \varepsilon \end{aligned} \tag{2.4}$$

## 2.2 CONTEXT-FREE LANGUAGES

While very useful for describing simple patterns, which have very useful applications, and are specially important for this work when laying down the foundations of formal languages, RLs are very limited. Indeed, simple languages formed by balanced parentheses or palindromes, are not regular.

For the study of PLs and NLs, we present the definitions of Context-Free Grammar (CFG) (2.2.1) and CFL (2.2.2), followed by the introduction of mechanisms necessary for the understanding of the process of parsing.

**Definition 2.2.1: Context-free Grammar**

A grammar $G = (V, T, S, P)$ is context-free if all its productions have the form

$$A \rightarrow x,$$

where $A \in V$ and $x \in (V \cup T)^*$.

Notice that CFGs have no restrictions to the structure of their right-hand side, neither on the number of nonterminals nor on their positioning.

**Definition 2.2.2: Context-free Language**

A language $L$ is context-free *iff* there exists a CFG $G$ such that $L = L(G)$.

### 2.2.1 Derivation Trees

So far we referred to grammars solely as a notation, but grammars can also be used to describe and visualize how we can recognize a sentence as part of a language, that is, if a sentence satisfies a given set of grammar rules.

The **derivation** is a process for defining the language of a grammar (14). We proceed from the top of the grammar, the start symbol, expanding it to one of its right-hand side rules. From the resulting sequence of elements, we choose one nonterminal and expand it by one of its rules, this substitution process goes on until we have a sequence composed only of terminals (14). The intuition behind this process is to search amidst the rules for the ones that will lead us to the token(s) in our sentence, if we fail to do so, the sentence is not recognized as part of a language.

Let $w = 0101$ be a four bit binary and $G$ the binary number grammar 2.3. The derivation of $w$ according to $G$ is illustrated by the following sequence of steps[6].

$$\begin{aligned} BNUM &\Rightarrow DIGIT\ DIGITS \Rightarrow 0\ DIGITS \Rightarrow 0\ DIGIT\ DIGITS \Rightarrow 01\ DIGITS \\ &\Rightarrow 01\ DIGIT\ DIGITS \Rightarrow 010\ DIGITS \Rightarrow 010\ DIGIT\ DIGITS \\ &\Rightarrow 0101\ DIGITS \Rightarrow 0101 \end{aligned} \tag{2.5}$$

We start at $BNUM$ and expand it into $DIGIT\ DIGITS$, the only rule defined by $BNUM$. Next we substitute $DIGIT$. At this point, we have a choice, either to derive the terminal 0 or 1. We chose the former since $w$ starts with it. Then we have to substitute $DIGITS$. Again, we must choose, this time between $DIGIT\ DIGITS$ and $\varepsilon$; we chose the first one, because choosing an empty rule would leave us with zero nonterminals to expand and terminate the derivation not recognizing $w$[7]. The remaining derivations follow the same pattern, until we reach the last derivation, when we substitute $DIGITS$ by $\varepsilon$ and conclude that $w \in L(G)$.

Presented as it is, derivation 2.5 is actually dependent on the order we choose for our substitutions[8]. In our derivation we chose to always substitute the first nonterminal to the left of a rule, such derivation is called a **left-most derivation**. If we choose to do the opposite the derivation is called a **right-most derivation**.

An order independent, more intuitive and very useful way of expressing CFG derivations is to use derivation trees, mostly known as **parse trees**. **Parsing** is the process of describing the structure of a sentence through the derivations of a grammar (15).

---

6 The symbol '⇒' indicates a derivation, a substitution of a nonterminal.

7 The derivation process is basically a hit and miss procedure, it is possible to improve it, but we will not cover how to do so.

8 Different orders of substitution will produce structurally different derivations.

**Definition 2.2.3: Parse Tree**

A tree $\tau$ is a parse tree of a sentence for a grammar $G = (V, T, P, S)$ if it satisfies the following conditions (14, 15).

1. The root of $\tau$ is labeled by $S$, the starting rule of $G$.
2. For all $n$, interior node of $\tau$, $n \in V$.
3. Every leaf $l$ of $\tau$ is in $T \cup \{\varepsilon\}$. If $l = \varepsilon$, then $l$ is the only child of its parent.
4. If a node $A \in V$ have its children labeled $a_1, a_2, \ldots, a_n$, then $P$ must contain a rule

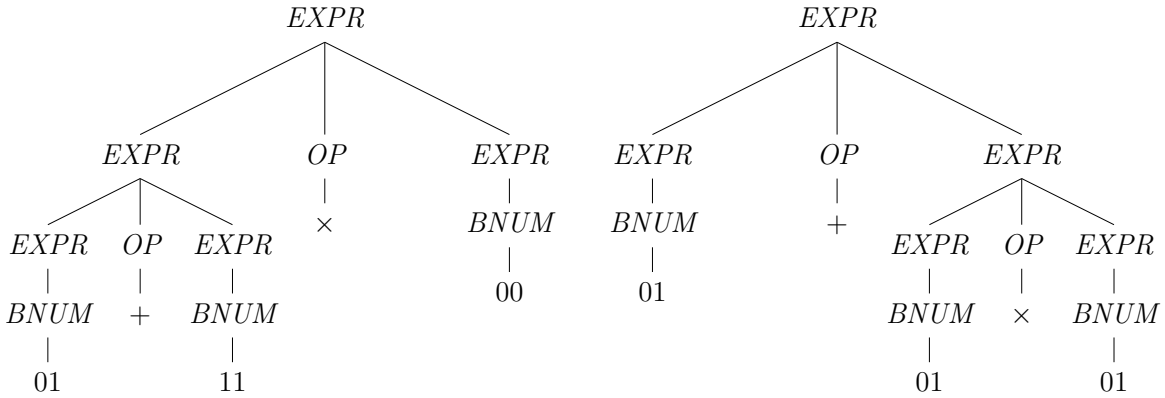
$$A \rightarrow \ a_1 \ a_2 \ \ldots \ a_n$$

Lets define a set of rules (2.6) of a binary arithmetic expressions grammar as follows,

$$\begin{aligned} EXPR &\rightarrow \ EXPR\, OP\, EXPR \mid BNUM \\ OP &\rightarrow \ + \mid \times \end{aligned} \tag{2.6}$$

where $BNUM$ is defined by grammar 2.3. The derivation of, say $w = 01 + 11 \times 00$, is given by the parsing trees illustrated in Figure 1.

Figure 1 – Ambiguous derivation trees.




(a) Left (b) Right

It is easy to see that trees 1a and 1b are parse trees of $w$ for grammar 2.6, according to Definition 2.2.3 [9]. The important thing to notice is that there are two valid parse trees for the same sentence $w$. This phenomenon is called **ambiguity** and it is one important concept for this work. Grammars for which some sentences can have multiple parse trees are called ambiguous (14, 15), and parsing sentences for these grammars present us with

[9] Keep in mind that in order to keep their size reasonable trees 1a and 1b are a little loose with the derivation of the $BNUM$ rule.

a challenge. What is the "correct" result of the parsing? We will revisit the ambiguity topic later when implementing our combinators, in Chapter 3.

Parse trees are particularly important for the implementation of parsers in general, they are the output of the application of parsing functions to a source input. Parse trees represent the structure of a language and can be naturally processed by programs such as analyzers and translators (14).

## 2.3 PARSING

As hinted in Section 2.2.1, **parsing** is the process of finding a sequence of derivations which leads us to conclude that a certain sentence is part of a language or not, based on a set of rules specified by a grammar (15). Until now we have talked about parsing from a lose and "generic" perspective, sometimes referred as exhaustive search parsing (15). For practical purposes, we need a more strict and organized technique that can give us an efficient implementation of a parser, and that is the topic we will cover in this section.

The **parser** is the component of a compiler[10] responsible for carrying out the parsing task. A parser will not only assert the membership of a sentence in a language, but also "organize" the components of the sentence into a structure that resembles the one defined by the grammar of the language, doing so by producing the data structure equivalent of the parse tree seen in Section 2.2.1 (15, 16, 17).

Parsing techniques are commonly discriminated into two categories, each with different advantages and capabilities, details that will not be covered due to the great variety of parsing techniques. **Top-down** parsers parse the input stream performing left-most derivations, they build the parse tree from the root, the grammar start symbol, down to the leafs, the grammar terminals (18). **Bottom-up** parsers map their input stream into the reverse of a right-most derivation, the idea is to "fill in" the internal nodes of the parse tree, starting from its leafs up to the root (16, 18).

In addition to being categorized in one of the above sets of parsing techniques, a parser can also be a general one, meaning that they are applicable to the whole set of CFGs (19, 16). In this work we explore the implementation of GLL parsers in a referentially transparent FL. First, in Section 2.3.1, we look at the basic Recursive Descent (RD) parsing technique, which will serve as foundation for presenting, in Section 2.3.3, the more sophisticated GLL technique.

This work will not cover bottom-up parsers, neither go into some of the details normally associated with top-down parsing, which will not contribute to the understanding of what is core to this work.

[10] Roughly speaking, a compiler is a program that translates a given text in some source language into text in a target language. Strict and more accurate descriptions of a parser can be found in the vast literature on this topic, some referenced in this work.

### 2.3.1 Recursive Descent Parsing

RD parsers are normally described as members of a larger set of top-down parsers called LL(1), because they scan the input stream from **L**eft to right (the first L of LL) and produce a parse tree from **L**eft-most derivations (second L), while using only one (1) element of the input stream, the one being matched, to unambiguously determine the control flow of the parsing procedure (17, 20).

As implied by the name, an RD parser relies on the use of recursive functions to fulfill its task. The intuition behind this technique is quite simple, the idea is to represent elements of a grammar by corresponding elements of source code, achieved by following a few basic directives (17, 18).

1. Each rule $R$ in the grammar is implemented by a function in the parser code. The right-hand side of $R$ specifies the structure of its corresponding function;

2. Sequences of terminals and nonterminals correspond to matches against the input stream and function calls, while alternatives correspond to conditionals in the code;

3. A terminal on the right-hand side of $R$ is matched against the input. In case of a match, we advance to the next element of the input stream, and to the next element on the right-hand side of $R$, otherwise an error must be reported;

4. A nonterminal is represented as a call to its corresponding function. We wait for the return of the called function. When it returns we continue to the next element on the right-hand side of $R$;

5. The process continues until there are no more elements to the right-hand of $R$ that need to be handled.

To illustrate how these directives can generate a parser, let us consider the simple arithmetic expressions grammar 2.7. For rule $FACTOR$ of this grammar, we would derive a function like the one described by Algorithm 2.1.

$$\begin{aligned} EXPR &\rightarrow EXPR\ ADD\ TERM \mid TERM \\ ADD &\rightarrow + \mid - \\ TERM &\rightarrow TERM\ MUL\ FACTOR \mid FACTOR \\ MUL &\rightarrow * \mid / \\ FACTOR &\rightarrow (\ EXPR\ ) \mid NUM \end{aligned} \tag{2.7}$$

By directive 1, each rule of the grammar is implement by a function of the parser, therefore we define the function FACTOR, line 1 of parser 2.1.

Directive 2 states that alternatives in the grammar correspond to conditionals in the code. We use the case statement to check if the next element of the input matches the

**Algorithm 2.1** Parses the rule $FACTOR$ of grammar 2.7.

```
1: function FACTOR
2:     case input of                          ▷ input is a shared stream
3:         '(':
4:             MATCH('('), EXPR, MATCH(')')
5:         num:
6:             NUM
7:         otherwise:
8:             ERROR
```

first terminal expected by the first alternative in the right-hand side of $FACTOR$, which is '(', or if it matches a number[11], first terminal expected by the second alternative of the $FACTOR$ rule, represented in the code as the identifier *num*, in line 5.

Lines 4 and 6 of Algorithm 2.1 are also derived from directive 2. They represent a sequence of terminals and nonterminals for each alternative of the $FACTOR$ rule. The calls to function MATCH are derived from directive 3, matching their arguments against the input. Functions EXPR and NUM are products of directive 4, and upon call will try to parse their corresponding rules. Directive 5 is self-explanatory. We now finish this section.

RD parsers are quite attractive for a couple of reasons. These parsers, as we saw, can be generated from simple directives and are quite suitable for handwriting, also the directives we have presented are mechanical in nature and therefore can be automated (18, 20). Not only that, but if we look at Algorithm 2.1, we can spot the similarities between the structure of the code and the structure of the rules of grammar 2.7. This last property is quite useful when we consider the verification and maintenance of a parser. They are human readable parsers, especially when compared to table driven techniques (13).

### 2.3.2 Drawbacks of RD Parsers

Even though RD parsers are quite attractive for their simplicity and close resemblance to the structure of grammars, as some other LL(1) parsers, they are not widely used as a production solution (18). From the drawbacks of RD parsing two are quite notorious and limit the capabilities of the technique.

The first limitation of the basic RD technique is due to a phenomenon called the **common prefix**. Common prefixes happen when two alternatives for the same grammar rule begin with the same sequence of grammar elements (18, 20). This is sometimes called a **prediction conflict**, since it is not possible to predict, based only on the input

[11] The matching of a number is loosely presented here. A stricter matching would require the definition of first and follow sets, intuitive concepts, but that would be troublesome to introduce, and would improve the understanding of the technique very little. We could also rewrite the function, removing the second case of the conditional, but the idea is to show how the directives work, keeping the structure of the function closer to what is described by them.

element being matched, which alternative will lead to a successful parsing (20). To clarify how common prefixes can be an issue for the implementation of RD parsers we use grammar 2.8.

$$\begin{aligned} IF \rightarrow\ & if\ EXPR\ then\ STMTS \\ |\ & if\ EXPR\ then\ STMTS\ else\ STMTS \end{aligned} \tag{2.8}$$

For grammar 2.8 we would derive a parser like the one shown in Algorithm 2.2.

**Algorithm 2.2** RD parser with prediction conflict.

```
1: function IF
2:     case input of
3:         if:
4:             MATCH(if), EXPR, MATCH(then), STMTS
5:         if:
6:             MATCH(if), EXPR, MATCH(then), STMTS, MATCH(else), STMTS
7:         otherwise:
8:             ERROR
```

Now, which is the right case to match? The one in line 3 of Algorithm 2.2, or the one in line 5? Well, most PLs that implement similar case constructs will match only one of its alternatives, normally the first one from the top. This is not what we want, because our parser would not recognize any source with an *else* particle, what does not correspond to the language specified by grammar 2.8.

To solve this problem we could look ahead on the input to learn what sentence is being parsed, with or without an *else* clause. But, how far do we need to look ahead? The answer is, there is no way to know, the *else* token we are looking for could lie (or not) anywhere ahead in the input (20), making this a not really practical solution.

Another possible solution is to modify the parser function IF into something like described by Algorithm 2.3.

**Algorithm 2.3** RD parser avoiding prediction conflict.

```
1: function IF
2:     MATCH(if)
3:     EXPR
4:     MATCH(then)
5:     STMTS
6:     if input = else then
7:         MATCH(else), STMTS
```

Whilst Algorithm 2.3 is a practical solution to the problem and one that actually recognizes the language specified by grammar 2.8, there are some catches. Not only do we need to modify the parser structure, what may be a quite onerous task on bigger and more complex parsers, but this parser also lost its resemblance to grammar 2.8 structure.

In fact, the structure of Algorithm 2.3 corresponds to the structure of another grammar, one that cannot be described by basic BNF syntax (18).

The second RD parsing weakness lies on recursion. Recursion is intrinsic to grammars, but not all recursions present us with an obstacle. What concerns us is left-recursion for the implementation of RD parsers.

From its name, one can figure that **left-recursion** is a particular case of recursion to the left of a sequence. Left-recursion can however be found in two forms. Rules $EXPR$ and $TERM$ of grammar 2.7 are examples of **direct left-recursion**, the nonterminal being defined is also the left-most element of at least one of its alternatives (20). **Indirect left-recursion** happens when there exists $N \overset{+}{\Rightarrow} \alpha N \beta$, where $\alpha$ can be empty, i.e. there exists a sequence of one or more derivations that consume no input, leading to a left-most derivation of rule $N$ (17, 18).

The problem with left-recursion in RD parsers is illustrated by Algorithm 2.4, which implements a parser for rule $EXPR$ of grammar 2.7.

**Algorithm 2.4** Left-recursive RD parser.

```
1: function EXPR
2:     case input of
3:         expr:
4:             EXPR, ADD, TERM
           ⋮
5:         otherwise:
6:             ERROR
```

At line 4 of Algorithm 2.4 we can see the sequence of function calls corresponding to the first alternative of rule $EXPR$ of grammar 2.7. We have identified that the element of the input currently being parsed can be derived from $EXPR$, therefore we immediately proceed to call function EXPR, the first of the sequence. Notice this is a recursive call to the same parser function, so we will again decide which of the alternatives to derive. However, we have not advanced in the input, since no terminal has been matched, thereby we fall again to line 4, where we call function EXPR again, once more without consuming any input, and so on.

It should be clear that the recursive pattern described by Algorithm 2.4 is one of nontermination; the execution of the parser will continue indefinitely. As for common prefixes the solution to left-recursion is to modify the parser, which is the same as to modify the grammar and implement a completely different parser.

Other possible disadvantages of RD are discussed when we present the implementation in Chapter 3. For now, in Section 2.3.3, we discuss a technique that allow us to overcome the limitations of the basic RD parsing technique, while maintaining its attractive properties.

### 2.3.3 GLL Parsing

As explained before, RD parsers are attractive because their control flow closely resemble the structure of their respective grammar, alas the set of grammars accepted by such parsers is limited. GLL is an RD-like parsing technique that addresses the aforementioned RD limitations by borrowing some of the tools used by Generalized LR (GLR), a natural language, bottom-up parsing technique developed by Masaru Tomita (19).

General parsing techniques normally incur less efficient implementations, but GLL parsers, as GLR parsers, have the property of having a linear performance as their grammars get closer to being deterministic, while their implementations maintain an almost one-to-one correspondence to the grammar structure (13), which speaks of the need for efficient human-readable parsers (16, 13).

We use grammar 2.9 as reference to introduce how GLL works.

$$\begin{aligned} S &\rightarrow ASd \mid BS \mid \varepsilon \\ A &\rightarrow a \mid c \\ B &\rightarrow a \mid b \end{aligned} \tag{2.9}$$

To build an RD parser for grammar 2.9 would give us something like Algorithm 2.5.

**Algorithm 2.5** Common RD parser for grammar 2.9.

```
1:  I                                              ▷ The input stream
2:  i ← 0                                          ▷ The current input index
3:  function PARSE
4:      S
5:  function S
6:      case I[i] of
7:          {a, c}:
8:              A, S, MATCH(d)
9:          {a, b}:
10:             B, S
11: function A
12:     case I[i] of
13:         a:
14:             MATCH(a)
15:         c:
16:             MATCH(c)
17:         otherwise:
18:             ERROR
19: function B                                     ▷ It has the same structure as function A
20:     ⋮
```

As we can see, grammar 2.9 is not LL(1) and therefore Algorithm 2.5 will not behave correctly, given the common prefixes of rules $A$ and $B$, which can easily be found at lines 7 and 9 of the parser.

The GLL technique addresses this issue by way of a couple of simple measures. First, function calls are substituted by labels and the parser control flow is explicitly carried by stack operations, and *goto* statements. The labels are also used for partitioning functions corresponding to non-LL(1) grammar rules. Lastly, the technique introduces the use of a mechanism called descriptor (13).

A **descriptor** is a triple of the form $(L, s, i)$, where $L$ is a label, $s$ is a stack, and $i$ is an index of input $I$. Basically, the set of descriptors $\mathscr{R}$ is used to record each possible parsing choice for any given rule, and control the termination of the parser (13).

Algorithm 2.6 illustrates how Algorithm 2.5 is modified by the GLL technique, so that it becomes capable of parsing the language defined by grammar 2.9.

For a better comprehension of how GLL works, using Algorithm 2.6, we go through some of the steps of parsing the string $I$ = "aad$", where '$' is a special symbol that marks the end of the input. To begin with, as most programming languages, we index our input starting at 0 and as expected initialize our input pointer, $i$, with 0. The descriptors set is empty and the current call stack contains the single special pair $L_0^0$. Since label $L_0$ will serve as our control, or dispatch loop, we must guarantee that we return to it at some point, and that is the purpose of initializing $s$ with $L_0^0$[12].

After initializing our control variables we proceed to label $L_S$. Here, as in the basic RD parser 2.5, we can see code instances for the three alternatives of rule $S$, from grammar 2.9, as well as the conflict between its first and second alternatives, at lines 3 and 4, respectively. What happens for the GLL parser 2.6, though, is that instead of trying to parse one of the alternatives, by immediately calling their corresponding parsing function, we add one descriptor for each of them to $\mathscr{R}$, and fall to label $L_0$ with $\mathscr{R} = \{(L_{S_1}, [L_0^0], 0), (L_{S_2}, [L_0^0], 0)\}$.

At label $L_0$ the descriptor $(L_{S_1}, [L_0^0], 0)$ is removed from $\mathscr{R}$. Each of the descriptors in $\mathscr{R}$ serve as a parsing state register of sorts, where the current label $L$ indicates what parsing procedure is to be applied, the index $i$ indicates from which position we should parse the input, and $s$ is the call stack for this particular procedure. As indicated by the descriptor just removed from $\mathscr{R}$, we jump to label $L = L_{S_1}$.

Once at label $L_{S_1}$ we take two simple actions, first we push the pair $L_1^0$ to $s$, and then proceed to $L_A$. Pushing $L_1^0$ before we go to $L_A$ is how we keep track of where we should return to, after parsing rule $A$. For this reason label $L_1$ is one of the so-called **return labels**. If $L_1$ is a return label, $L_{S_1}$ is what is called an **alternate label**, the label for the first alternative of $S$.

Label $L_A$ corresponds to the parsing function A from Algorithm 2.5. It will match the input against the expected terminal, perform a POP operation, and return to $L_0$. Label $L_A$ is a representative of the last category of labels used by GLL parsers, it is a **nonterminal label**, which is self-explanatory.

[12] $L_0^0$ is a syntax sugar for the pair $(L_0, 0)$, where the second element of the pair is an input index.

**Algorithm 2.6** GLL parser for grammar 2.9.

```
1:  i ← 0, 𝓡 ← ∅, s ← [L_0^0]
2:  L_S:
3:      if I[i] ∈ {a, c} then (L_S1, s, i) is added to 𝓡
4:      if I[i] ∈ {a, b} then (L_S2, s, i) is added to 𝓡
5:      if I[i] ∈ {d, $} then (L_S3, s, i) is added to 𝓡
6:  L_0:
7:      if 𝓡 is not empty then
8:          (L, s, i) ← 𝓡                          ▷ A descriptor is removed from 𝓡
9:          if L = L_0, s is empty, and i = |I| then
10:             SUCCESS
11:         else
12:             go to L
13:     else
14:         FAILURE
15: L_S1:
16:     PUSH(L_1^i, s), go to L_A                  ▷ L_χ^i is syntactic sugar for the pair (L_χ, i)
17: L_1:
18:     PUSH(L_2^i, s), go to L_S
19: L_2:
20:     if I[i] = d then
21:         MATCH(d)
22:         POP(s, i, 𝓡)
        go to L_0
23: L_S2:
24:     PUSH(L_3^i, s), go to L_B
25: L_3:
26:     PUSH(L_4^i, s), go to L_S
27: L_4:
28:     POP(s, i, 𝓡), go to L_0
29: L_S3:
30:     POP(s, i, 𝓡), go to L_0
31: L_A:
32:     if I[i] = a then
33:         MATCH(a)
34:         POP(s, i, 𝓡)
35:         go to L_0
36:     else if I[i] = c then
37:         MATCH(c)
38:         POP(s, i, 𝓡)
39:     go to L_0
40: L_B:
41:     ⋮
```

Before we go back to $L_0$ it is worth noticing that the POP function used by GLL is a little unconventional. While a conventional pop operation would take a stack as parameter, remove the top element and return it as result, the pop action used by Algorithm 2.6 takes the current call stack and input pointer, as well as the set of descriptors $\mathscr{R}$. What happens is that POP will indeed remove the top element of $s$, which, at this point, would be $L_1^0$, but it will also create a descriptor of this return point in the parsing procedure and add it to $\mathscr{R}$. We add descriptor $(L_1, [L_0^0], 1)$ to $\mathscr{R}$ as consequence of the POP in $L_A$, where $L_1$ is the label at the top of $s$, $[L_0^0]$ is the remainder of the call stack after the pop, and $i = 1$ after matching the first '$a$' in the input.

Back at $L_0$ we remove descriptor $(L_{S_2}, [L_0^0], 0)$ and go on to repeat the same series of steps we performed for the first descriptor we removed from $\mathscr{R}$, except we start at $L_{S_2}$ and finish at label $L_B$ instead of starting at $L_{S_1}$ and finishing at $L_A$.

After processing the first two descriptors in $\mathscr{R}$, we have completed the early steps of parsing the conflicting alternatives of rule $S$, with $\mathscr{R} = \{(L_1, [L_0^0], 1), (L_3, [L_0^0], 1)\}$. We will then remove the first of these descriptors and continue the parsing from label $L_1$ as we did before. This process will go on until we eventually have $\mathscr{R} = \{(L_0, [], 3), (L_2, [], 2)\}$. Removing $(L_0, [], 3)$ will satisfy the conditional at line 9, and successfully finish parsing our input.

Whilst the GLL technique as illustrated by Algorithm 2.6 is capable of handling ambiguous grammars, it is not general, given that left-recursion remains a drawback. Say, for example, we needed to build a parser for some grammar similar to 2.10

$$S \rightarrow \ S\beta \mid \ \ldots \tag{2.10}$$

In the same way as Algorithm 2.6, we would have a label $L_S$ were we could potentially add a descriptor $(L_{S_1}, [L_0^0], 0)$ to $\mathscr{R}$, for the first alternative of $S$. After dispatched to parse the first alternative of $S$ we would find ourselves executing line 8 of Algorithm 2.7.

**Algorithm 2.7** Left-recursive GLL labels for grammar 2.10.

```
1: L_S:
2:     if I[i] ∈ some matching set of terminals then
3:         (L_S1, s, i) is added to R
4:     ⋮
5: L_0:
6:     ⋮
7: L_S1:
8:     PUSH(L_1^i, s), go to L_S
```

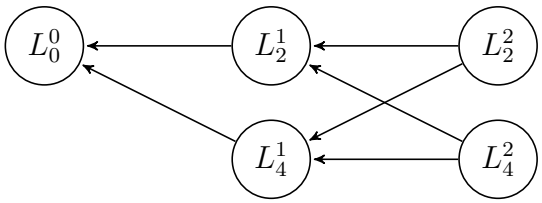
At label $L_{S_1}$ we would update our call stack to $s = [L_0^0, L_1^0]$ and immediately return to $L_S$. Even though we have a new call stack, we have not advanced the input pointer.

This means we would once more satisfy the conditional at line 2, adding the "same"[13] descriptor as before, such that $\mathscr{R} = \{\ldots, (L_{S_1}, [L_0^0, L_1^0], 0)\}$. When we later remove this descriptor from $\mathscr{R}$ we will repeat the exact same steps just described. As result, we have $\mathscr{R} = \{\ldots, (L_{S_1}, [L_0^0, L_1^0, L_1^0], 0)\}$, leading into an infinite recursion.

Besides the remaining issue with left-recursion, as presented, the GLL technique can create a number of descriptors exponential in the size of the input (13). To cope with this issue the GLL algorithm employs a few extra machinery. The core modification needed is to replace the traditional call stack used on the descriptors by what is called a Graph-Structured Stack (GSS) (19, 13, 21).

The GSS is a directed cyclic graph[14], a shared data structure that allows the combination of all call stacks into a single structure (13, 21). For set $\mathscr{R} = \{(L_{S_3}, [L_0^0, L_2^1, L_2^2], 2), (L_{S_3}, [L_0^0, L_2^1, L_4^2], 2), (L_{S_3}, [L_0^0, L_4^1, L_2^2], 2), (L_{S_3}, [L_0^0, L_4^1, L_4^2], 2)\}$, created when parsing "aad$" with Algorithm 2.6, its four stacks are combined into the graph depicted by Figure 2, as a consequence we can reduce the size of $\mathscr{R}$, by treating descriptors that share the same call stack top element as a single descriptor, so that $\mathscr{R} = \{(L_{S_3}, L_2^2, 2), (L_{S_3}, L_4^2, 2)\}$.

Figure 2 – Simple GSS example.




Finally, to allow the GLL to process left-recursive grammars, we also need some auxiliary tools to help keep track of the operations performed on the GSS and $\mathscr{R}$ structures. First, a list of sets $U_i = \{(L, u) | (L, u, i),$ *has already been added to* $\mathscr{R}\}$ that allow us to cut out left recursion before it "explodes". Before adding a descriptor to $\mathscr{R}$, we check if it was already added for the current input index. A set $\mathscr{P}$, which stores all pairs of labels and input indexes for which a pop action has been executed. It will be consulted every time a new node is added to the GSS, to guarantee that all the needed descriptors are added to $\mathscr{R}$ (13, 21).

The addition of this extra machinery will not really affect most of the intuition behind the algorithm, its core components and properties, or execution flow, for this reason we do not explore the GLL technique any further. In depth insight into its operation can be found in works such as (13, 21), details on how to operate using the GSS, auxiliary structures $U_i$ and $\mathscr{P}$, a set of directives for parser generation, and more.

[13] Despite the fact that their stacks are actually different, having the same label and input index is enough, in this case, for calling them the same.

[14] This is for GLL, the original work of Tomita (19) defines the GSS as being acyclic.

To finish this section, we observe that the GLL algorithm is designed for imperative languages, and even for those languages its implementation will incur the need for adaptations, since unrestricted **goto** statements, used in the GLL pseudo code, are not at all common. Also, the algorithm heavily relies on shared, mutable data structures (21). These factors present us with a difficult challenge, re-imagining how to implement the GLL algorithm by using a functional PL, specially a referentially transparent one, such as Haskell. We discuss the details of this GLL implementation in Chapter 3.

## 2.4 EXTENDED BACKUS-NAUR FORM

In previous sections, when we had to show examples of grammars we used a notation common to some books on the theory of formal languages. While fairly common and not "harmful" in any way, this notation is not ideal for this work. This section covers the notation chosen as input format for our parser generator, which provides convenient mechanisms for the practical specification of languages.

A syntactic metalanguage is a notation for defining the syntax of a language. Despite their importance, many different notations do exist, what leads to misunderstandings and non-appreciation of the advantages provided by rigorous notations (22).

Since the introduction of the Algol 60, PLs have followed the tradition of "defining themselves" formally. The BNF notation was introduced as the syntactic metalanguage for the definition of Algol 60. It has been adopted, extended and/or slightly altered by many ever since (22). The existence of many extended or modified notations for BNF, however, leads to a series of problems such as:

- Having to adapt to many different notations can be quite confusing. Every time you may want to learn some detail about a language syntax, you will find yourself looking at a different metalanguage;
- Also, different metalanguages often make use of particular, special notation or features. This can hinder the understanding of what is being defined, and ultimately, make those metalanguages unsuitable for the definition of another language;
- Finally, the lack of consistent formalism impairs the development of tools for processing meta languages.

The ISO/IEC 14977 notation is an effort to define the standard for BNF-based metalanguages, compiling the most common BNF extensions. The ISO 14977 standard is used as the input format for our parser generator, as well as a guide for the implementation of our combinators, and as reference for defining this work own Extended Backus-Naur Form (EBNF) parser. We now take a brief look at its core components. We start by defining the main components of the ISO 14977 EBNF.

| | |
|---|---|
| **Sequence** | An ordered list of zero or more items. |
| **Sub-sequence** | A sequence within another. |
| **Non-terminal** | A syntactic element of the language. |
| **Meta-identifier** | The name of a non-terminal symbol. |
| **Start symbol** | A non-terminal that does not occur in any other syntax rule. |
| **Sentence** | A sequence of symbols representing the start symbol. |
| **Terminal** | A sequence of one or more characters forming an irreducible element of a language. |

Before we proceed, some notes about the definition of these components. The sequence component is presented above as it is on the ISO standard, but the requirement of some sort of order between its items is an odd one, specially when no order criteria is provided. For this reason, we will ignore it. Also, it states that the length of a sequence can be null. Since grammar rules correspond to parsing functions, defining a rule as an empty sequence of elements would compare to the definition of a function without a body, therefore, a sequence must contain at least one item.

Regarding the start symbol, the ISO EBNF does not anticipate any indirect recursion over the start symbol, a restriction we overlook in this work. It does not affect in any way the definition of input grammars, nor the implementation of combinators.

Next, we present the set of operators and symbols defined by the ISO EBNF, and give an informal description of their semantics. We divide them into two sets. In Table 5, we list stand-alone operators and symbols, while in Table 6, we list the balanced ones.

Table 5 – ISO EBNF stand alone symbols/operators.

| | |
|---|---|
| * | repetition-symbol |
| , | concatenate-symbol |
| \| | alternative-symbol |
| = | defining-symbol |
| ; | terminator-symbol |

In Table 5, the concatenate- and alternative-symbol correspond directly to the alternative and sequence operators from CFGs (in Table 3), with the exact same semantics. The defining-symbol, as implied by its name, indicates the definition of a rule (non-terminal), it unifies the notation for symbols such as '::=', '→', or ':'. The terminator-symbol is just a punctuation indicating the end of a rule definition.

The repetition-symbol needs a more elaborate explanation. It must not be confused with the RegExs '*' operator (introduced in Section 2.1.1), it is a binary operator for defin-

ing the finite concatenation ("repetition") of a sub-sequence ß, such that $n^*ß \equiv ß_1, \ldots, ß_n$, where $n \in \mathbb{N}$[15].

Finally, in regard to the precedence of the operators in Table 5, their precedence is as laid out by the table, from the highest precedence at the top, to the lowest, at the bottom row. However, the termination-symbol is not an operator, and defining-symbol also does not need to be seen as one.

Table 6 – ISO EBNF balanced symbols.

| | | | |
|---|---|---|---|
| ' | single-quote-symbol | | |
| " | double-quote-symbol | | |
| ( | start-group-symbol | end-group-symbol | ) |
| [ | start-option-symbol | end-option-symbol | ] |
| { | start-repeat-symbol | end-repeat-symbol | } |

Table 6 lists a set of grouping symbols. Lets start with both quotation marks, single-quote- and double-quote-symbol. These symbols are used to define, group together, a terminal. A terminal started by a single quote must end with a single quote, the same applies to double quotes. Quoted sub-sequences can exist inside another using the alternate quotation, "Quote this, 'quote that'", but at no point the same quotation symbol can be used in a sub-sequence of itself, "Quote 'this, "quote that"'".

The next three balanced pair of symbols define the grouping of rule structures, meaning that any element used on a non-terminal definition can be used inside of these grouping symbols, except for defining-symbol, and termination-symbol. These structures defined inside of a grouping pair of symbols act as anonymous rules definitions, terminated by one of the end-*-symbols.

The only semantic addition made by these symbols has to do with precedence and grouping. Consider the following definition $id = (letter \mid digit), 31^*(letter \mid digit);$, although ',' has precedence over '|', because the alternative between letter and digit occurs inside '( )', choosing one of them must be resolved before the sequencing, also the repetition, in the second element of the sequence, is applied over the hole choice and not only its first element, such that we have

$$id = (letter \mid digit), (letter \mid digit)_1, \ldots, (letter \mid digit)_{31};$$

instead of

$$id = letter \mid digit, letter_1, \ldots, letter_{31} \mid digit;.$$

The described grouping and precedence modification effects also hold for the remaining two grouping symbols, but both add their own extra semantic twist.

[15] If greater clarification is needed. $\mathbb{N} = \mathbb{N}_0 = \mathbb{N} \cup \{0\}$.

First, the *-option-symbols define an optional rule structure, all that is defined between '[ ]' may be matched at most once. Finally, the *-repeat-symbols define a similar repetition pattern as the repetition-symbol from Table 5, but it defines a non-deterministic number of repetitions of whatever is described between '{ }'. Instead of a finite one, it will match a pattern for as long as it happens in the input stream.

To illustrated some of the EBNF functionalities and syntax we recall grammar 2.8. Its parser, Algorithm 2.3, was said to actually correspond to another grammar, one that could not be described by basic BNF notation. We introduce the EBNF grammar corresponding to Algorithm 2.3, using the syntactic metalanguage described in this section.

```
if = 'if', expr, 'then', stmts, [ 'else', stmts ];
```

More examples to help understanding the ISO EBNF notation can be found in Chapter 4, and Appendixes A and C. Details regarding the implementation of our EBNF parser and combinators, as well as their limitations, are discussed in Chapter 3 and Section 4.2, respectively.

## 2.5 HASKELL

Before we proceed to the discussion of our implementation and results, we introduce what we believe are necessary concepts for the understanding of this work implementation and discussion. We present the fundamental concepts of Haskell, a general purpose, purely functional PL.

We take a moment to consider the term "purely functional". It is because of this purity that the so-called pure FLs are regarded as tools for writing secure, error-free, systems (23). The term itself is controversial, and there is no agreement on its meaning (24). For the purposes of this work, we focus on the fact that pure FLs center around the notion of evaluation of expressions, rather than the modification of the state of a program (25). We do not dwell on this topic any longer, and defer to works such as (24) for deeper and more accurate discussion.

To center around the evaluation of expressions means that the programmer defines functions in terms of equations, which obey normal mathematical principles, and that determine the value of a function for an arbitrary input (23, 26). This idea is sometimes called computation by calculation, because the role of the underlying machine is to act as a simple evaluator (calculator), except that we can increase the power of this evaluator by introducing new definitions (25, 26).

### 2.5.1 Expressions and Types

Let us start our Haskell tour by defining some fundamental concepts. First, if we are to perform calculations, what are the objects of these calculations? As already hinted, in the

beginning of this section, we perform calculations on expressions. **Expressions** are used to denote values, which can be atomic, indivisible, such as a number or a character, or can be structured, composed of smaller elements, such as lists, tuples and functions (25, 26). **Values** are expressions resulting of an evaluation (25).

Every well-formed Haskell expression has a type (25). **Types** are collections or sets of values, which are in some way similar to each other (23, 25, 26). For example, although `'a'` and `'b'` are different characters, both are elements of the set of all characters, and we can apply the same kind of operations over them, such as capitalization. Some Haskell values and their associated types are illustrated in Table 7.

Table 7 – Examples of values and their types.

| | | |
|---|---|---|
| `101` | `::` | `Integer` |
| `'a'` | `::` | `Char` |
| `['a', 'b', 'c']` | `::` | `[Char]` |
| `(['o','n','e'], 1)` | `::` | `([Char], Integer)` |

The symbol '`::`' can be read as "has type" or "is a(n)". The last two items of the table are Haskell's basic structured types, the first is a list of characters and the second a tuple of two elements (a pair), the first element a list of characters, the second an integer number.

A list is a homogeneous sequence of elements, all elements must be of the same type, enclosed by brackets, and separated by commas (27). Lists may have an arbitrary number of elements, from zero up to $\infty$, including other lists. In this case its type would be indicated by `[[T]]`, for a list of lists of `T`. If lists are homogeneous and possibly infinite, tuples are possibly heterogeneous and finite, with elements enclosed by parentheses and separated by commas (27). Tuples of many elements have type `(T1, ..., TN)`.

For a FL the most important kind of expressions is the function. A function `f` is a relation, a rule of correspondence, from an element of type `A` into a value of type `B`. This relation is expressed by a type signature of the form `f :: A -> B`. Table 8 shows examples of functions and their corresponding types.

Table 8 – Examples of functions and their types.

| | | |
|---|---|---|
| `odd` | `::` | `Integer -> Bool` |
| `toUpper` | `::` | `Char -> Char` |
| `length` | `::` | `[a] -> Int` |

There are a couple of new elements worth noticing in Table 8. First, we can see that the type names in the signature of functions are separated by the operator `(->)`. The arrow operator is used to define function types, it takes two types as parameters and

gives us a function type. Functions with more than two types can be defined by extending the arrow notation `T1 -> ... -> TN`, where `TN` indicates the return type of a function.

Another new element can be noticed in the type signature of function `length`. Instead of specifying its parameter type with a type name, as the other functions, `id` uses a variable, which is called a `type variable`. Haskell expressions are polymorphic by default, the type variable `a` indicates the polymorphic nature of function `length`. A type variable can be substituted by any Haskell type, "allowing" the parametrization of a function, meaning that `length` can calculate the length of any list (27).

### 2.5.2 Definitions

Haskell programs are composed of a number of definitions, which associate an identifier (name) to an expression (23). For example.

```
pi :: Float
pi = 3.1416
```

The first line of the above definition declares `pi` to be of type `Float`, and the second line associates (binds) the identifier `pi` to the value 3.1416. The second line of the definition of `pi` is called an equation (25).

Now suppose we want to use `pi` to calculate the area of a circle. We can define a function `circleArea` as follows.

```
circleArea   :: Float -> Float
circleArea r = pi * square r
    where
        square n = n * n
```

Notice that function `circleArea` is defined in the same way as `pi`, except that it takes a parameter `r :: Float`, and its body, the expression to the right of the symbol '=', is slightly more elaborate. The body of `circleArea` tells us that the area of a circle is given by the product of `pi` and the square value of `r`. We observe that `square` is a local definition, introduced by the use of the keyword `where`[16]. The function `square` takes a number `n` and multiplies it by itself. If we look closer, we see that `square` is applied to its argument without parentheses, common in many other PLs. In Haskell, function application is indicated by spaces between the function name and its arguments. We show some examples of function applications in Table 9.

First function, `sort`, takes a list of numbers and return a list of numbers in ascending order. The second function takes a pair, not a pair of arguments, but a single tuple composed of two elements, and returns the first of them. And at last, the function `apply`,

[16] There are other ways to introduce local definitions, such as the `let ... in ...` construction, which can be found in Chapter 3.

takes two arguments, a function over numbers and a number, then it applies the first argument to the second.

Table 9 – Haskell function applications.

| | |
|---|---|
| `sort` | `[1, 0, 3, 2]` |
| `fst` | `(x, y)` |
| `apply` | `(\n -> n * n) 0` |

There are a couple of things worth discussing about the function `apply`. As mentioned, it takes another function as an argument. In Haskell, functions can be handled as any other object, such as numbers (23). A function that takes another function as argument, or returns a function as its result, is called a **high-order function**. Next, the function, argument of apply, is a **lambda-expression**, or simply **lambda**, an anonymous definition of a function. A lambda starts with a backslash (\) symbol, followed by a list of arguments separated by spaces, then the definition symbol '`->`', and an expression. Finally, notice that the lambda-expression is enclosed within parentheses to guarantee that `apply` will see the whole expression as a single parameter.

One last note on the definition of functions. We mentioned that the binding of an identifier to an expression can be called an equation, this has some meaning that goes beyond just nomenclature. A Haskell function can be defined by multiple equations. Consider for example the definition of addition over natural numbers in axiomatic set theory. If we were to define its equivalent in Haskell, we could do so as follows.

$$+(m, 0) = m \ \forall m \in \mathbb{N};$$
$$+(m, S(n)) = S(+(m, n)) \ \forall m, n \in \mathbb{N}$$

```
add :: Nat -> Nat -> Nat
add m Zero     = m
add m (Succ n) = Succ (add m n)
```

We see that there are two equations for the definition of addition, both in set theory and Haskell. The first equation defines the addition of any number $m$ to zero to be equal to $m$. The second equation is defined recursively, for the sum of a number $m$ and the successor ($S$, `Succ`) of another number $n$, is the successor of the sum of $m$ and $n$. Simplifying, for each iteration of `add` in which we reduce $n$ by one, we add one to $m$.

Now that we have two equations with the same name, which one should we use when calling `add`? Functions defined by this sort of equational reasoning are chosen by pattern matching over the structure of their arguments values. The syntactic expressions of values, called patterns, are used to choose, on a top-down order, which is the appropriate equation (27). For example, if we call `add a Zero`, the first equation will be applied, since the second parameter matches the pattern defined in the first equation. On the other hand, if we call `add Zero a`, we will fall to the second equation, which will then be applied,

assuming that `a` is of the form `Succ (...)`[17].

One interesting aspect of the addition of natural numbers, previously defined, is the natural number type and its structure, which closely relates to the definition of natural numbers in set theory. How is that possible? Does Haskell defines natural numbers in such an "unconventional" manner, instead of using digits?[18] What happens is that Haskell allows us to define our own types, a topic we discuss next.

We walk or way up on the definition of types in Haskell by first introducing the idea of type synonyms. As its name implies, **type synonyms** is a mechanism for giving alternative names for types, possibly more suggestive, clearer names, for an already existing type (26). Now, remember Table 7, some of the types used to describe those values are "suitable" for having type synonyms. In fact, the third line of Table 7, the list of characters, has a common type synonym defined by Haskell. Also, the last item of Table 7 could, for example, represent some element of an associative list. The code bellow illustrates how these synonyms would be defined.

```
type String = [Char]
type Entry  = (String, Int)
```

We can use the synonyms in the example above to rewrite the last two entries from Table 7 as shown in Table 10.

Table 10 – Values and their types with type synonyms.

| | | |
|---|---|---|
| `"abc"`[19] | `::` | `String` |
| `("one", 1)` | `::` | `(String, Integer)` |

We can further elaborate on the idea of type synonyms and their utility. Consider the type `Entry` we just defined. Say for example that we want to represent any type of entry indexed by a string, we could do that by introducing a type variable, rewriting our type `Entry` as follows.

```
type Entry a = (String, a)
```

We can use this type synonym to define other synonyms, such as the associative list in which an `Entry` would be stored.

```
type Assoc a = [Entry a]
```

[17] More information and examples on pattern matching can be found in the references used in this section.

[18] Some Haskell implementations actually provide a similar implementation of natural numbers.

[19] A list of characters can be written as one would normally write a string in most PLs.

Although type synonyms can prove to be quite useful, they do not actually define new types, only aliases. **Algebraic Data Types (ADTs)** are Haskell's mechanism for defining new types. ADTs resemble grammar rules in the structure of their definitions. They are composed of a left-hand side called type constructor, which defines a new type (similar to nonterminals), and a right-hand side composed of an "enumeration" of value constructors, which specify what values a certain type may have (alternatives).

To illustrate the definition of an ADT, we recall the `Nat` type, previously used to illustrate the definition of functions, which can be defined in the following manner.

```
data Nat = Zero | Succ Nat
```

What this definition tells us is that a natural number is either zero, or the successor of another natural number. Any natural number could be constructed as described in Table 11.

Table 11 – Natural numbers and their corresponding digits.

| `Nat` | Digit |
|---|---|
| `Zero` | 0 |
| `Succ Zero` | 1 |
| `Succ (Succ Zero)` | 2 |
| `Succ (Succ (Succ Zero))` | 3 |
| ⋮ | ⋮ |

Types where at least one of its value constructors is defined in terms of the type constructor, in the case of `Nat` its second value constructor, are called recursive types. The recursive pattern of type `Nat` should be clear in the values listed in Table 11.

To finish this section we would like to observe that Haskell offers yet another device for defining types. It is similar to ADTs in the sense that it is composed of a left-hand side, where the type constructor is defined, and a right-hand side, where the value constructor is defined, except that it can only have a single value constructor. We show an example of this mechanism in the following code, a slightly different version of the associative list introduced earlier.

```
newtype Assoc a = Assoc [Entry a]
```

Despite the restriction to the number of value constructors, the `newtype` construction provides an easy way to define a simple type that can be made into an element of a type class, the main topic of our next section.

### 2.5.3 Type Classes and Monads

If a type is a class of values similar to each other according to some criteria, normally the kind of operations we can perform over such values (23), **type classes** define a class of types, similar in regards to the functions that can be applied over the values of a type within such class. For example, the type class `Eq` defines a set of types that can, in some way, be compared for equality. Specifically, it specifies the fundamental common behaviors amongst the instances of a type class. The definition of the type class `Eq` follows.

```
class Eq a where
    (==) :: a -> a -> Bool
```

One way of reading the above class definition is as "a type `a` is an instance of class `Eq` if the operation `(==)` can be defined for type `a`" (25). To illustrate how a type can be made into an instance of a type class, we implement an instance of the type `Assoc`, introduced at the end of Section 2.5.2, for the type class `Eq`.

```
instance Eq a => Eq (Assoc a) where
    (Assoc [])     == (Assoc [])     = True
    (Assoc [])     == (Assoc _)      = False
    (Assoc _)      == (Assoc [])     = False
    (Assoc (x:xs)) == (Assoc (y:ys)) = x == y && xs == ys
```

What the implementation of `Assoc` as instance of `Eq` tells us is, two empty associative list are equal. An empty associative list is not equal to another list that is not empty. And finally, an associative list is equal to another, if each of its elements is equal to the corresponding element in the other list[20]. This instance says something like "type `Assoc` is an instance of class `Eq` and this is how the operation `(==)` is performed on it".

Type classes can also be used to enforce restrictions on certain operations. This is sometimes referred to as **qualified types**, which is a solution for when one might need or prefer to limit a type variable (polymorphic type) to a smaller class of possibilities (25). This happens on the first line of our `Assoc` instance of `Eq`, where we say "are instances of class `Eq` only those associative lists composed of elements of type `a`, such that `a` can also be compared for equality".

A particularly notorious Haskell type class is called `Monad`. In Haskell a **monad** is a parametrized type `m`, an instance of the type class `Monad`, which has its fundamental behaviors characterized by the following functions (28).

```
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b
```

[20] Notice that our associative list is actually a common list of pairs, so the comparisons of the elements are perfomed in an ordered fashion, from the left to right, something that might not happen on an actual implementation of an associative list.

The `return` function states that a value of type `a` can be lifted, put in the context of type constructor `m`. Operator `(>>=)`, also known as bind, states that computations over values within some context identified by `m` can be sequentialized.

To clarify these ideas of context and computations within contexts, consider the monad `Maybe`, which we will see later in Chapter 3. The type `Maybe` represents uncertainty, whether a computation returns a value or not, it is a way to represent partial functions via types. Imagine for example some collection of type `c`, and that we want to define a function that retrieves some element of said collection. Such a function cannot return a value that is not in the collection, but a function in Haskell always returns a value and therefore we need some way of adding extra context to this computation. The type `Maybe` provides us such context by means of two constructors, `Just` for when a function returns a value for a given argument, `Just c` for our collection, and `Nothing` otherwise. There is a catch to this "trick" though, we cannot operate directly over values wrapped within a context. A value of type `Maybe Int` is not the same as a value of type `Int`, for instance, we cannot directly add (`+`) values of the former. Functions such as `(>>=)` provides us with a tool for reaching for the values within the context of a monad. The bind function in particular, allows us to apply operations to those values sequentially, while being capable of "storing" the intermediate results.

With this brief introduction to monads, we conclude our tour of Haskell. Although a simple language in terms of its syntax, Haskell has plenty of semantic aspects that are unconventional for those not used to functional programming. Yet, we do not wish to dwell to long on its details, and possibly complicate the understanding of concepts necessary for this work comprehension. We encourage the reader, if interested or in need, to take a look at the references utilized throughout this section.

# 3 ON THE IMPLEMENTATION OF GLL COMBINATORS

> *Three witches you shall meet, along the road to your fate. The first is twilight, the second is night, and the third is the coming of day.*

The Sword, *2010*

To start this chapter we take a moment to consider its epigraph. This quote was intended as a metaphor for the general structure in which the few articles (9, 21, 29) discussing the implementation of the GLL parsing technique, in a mostly functional manner, are organized.

Those works normally discuss their implementation of GLL as a three step process. First, the implementation of standard parser combinators, where the basic and main concepts are covered. This section would be analogous to the first witch mentioned in the epigraph, the twilight.

The second step would be to formalize extra machinery necessary to achieve the implementation of a general, GLL like, set of combinators. At this point the concept of Continuation-Passing Style (CPS) combinators and the basic notions of memoization are introduced. This step is analogous to the night.

And finally, the memoization of parsing combinators and continuations, as a tool for coping with left-recursion and ambiguity is discussed. This third step would finally lead us to the dawn of a new parsing tool, one that adapts a strictly imperative technique into a set of composable functions for parsing.

Although this work fails to provide GLL like combinators, as the underlying parsing tool of our parser generator, we still cover their implementation and choose to do so in the same format described above. We believe it is a good constructive strategy to provide insights into the pitfalls of GLL implementation in a pure FL, such as Haskell.

## 3.1 STANDARD PARSER COMBINATORS

Parser combinators are simple high-order functions used to represent grammar constructions such as choice, repetition, and so on (30, 31). In this context, a parser is both the combination of many parser combinators, but also each of these smaller components. Therefore, a parser is normally defined as a function, or a function type.

```
type Parser i a = i -> [(i, a)]
```

As defined above, a parser is a function from some input `i` into a list of pairs, which are composed by the remaining unconsumed input, and some result of type `a`. The return being a list makes it possible to represent the failure of a parser, by yielding an empty

list, and also the non-deterministic nature of grammars, returning a list of results, one for each alternative of a rule, for example.

We take a slightly different approach. A parser is defined as a class of types, in this case a class of function types (32):

```
class Parser m where
    parse          :: m a -> String -> [(String, a)]
    fullParse      :: m a -> String -> [a]
    fullParse p i = [ a | ("", a) <- parse p i ]
```

where the function `parse` is the main interface for the application of a parser, and `fullParse` applies a parser but does not return partial results of that parser application, it only returns results for parsers that consume the whole input.

The use of a type class, instead of a data type or type alias, allows simple extensions to the concept of parser, whilst maintaining a consistent interface to its different definitions.

Another particular characteristic of this combinators implementation is their separation into two "sets". The first corresponds to a monadic implementation of a parser instance, similar to what happens in many other works. The second component implements a stricter version of the combinators to be used as the main back-end tool for parser generation. This separation of the combinators into two components allows the use of the combinators apart from the parser generator, as a library for the implementation of parsers, similar to what is proposed by many works on functional parsing.

Our implementation of the basic idea on parser combinators starts with the definition of a "standard" parser.

```
newtype Std a = Std (String -> [Res a])
-- where
type Res a = (String, a)
```

It should be clear by now why this is called a standard parser, it is a simple parametrization of the general concept of parser presented in the beginning of this section. A parser is a function from an input string into a list of results.

The next step is to make this function type an instance of `Parser` and `Monad`. The implementation of the standard parser instance is quite simple.

```
instance Parser Std where
    parse (Std p) = p
```

The implementation of the `parse` function consists of the simple extraction of the actual parsing function from its constructor.

Now, the instance of `Monad` for the standard parser is defined as follows[1].

[1] Instances of the `Monad` type class are required to be instances of the `Functor` and `Applicative` type classes. We chose to not cover their implementations, since it does not contribute to the understanding of how parsing combinators work.

```
instance Monad Std where
    -- return :: a -> Std a
    return a    = Std $ \i -> [(i, a)]
    -- (>>=) :: Std a -> (a -> Std b) -> Std b
    Std p >>= f = Std $ \i ->
        concat [ q i' | (i', a) <- p i , let Std q = f a ]
```

First, `return` defines a parser that always succeeds for an arbitrary value `a`. The parser just makes `a` the second element of its return value, and consumes no input. The bind operator takes a standard parser `p` and applies it to an input `i`, then the function `f` is applied to the second element of each result produced by `p`. The application of `f` results in another parser `q` which is then applied to the remaining input `i'`. We use list comprehensions to define our bind operator[2].

Despite being an instance of `Monad`, our standard combinators are not very useful. For changing this, first we define the basic grammar operators listed in Table 3, Chapter 2.

```
class GrammO m where
    (<:>) :: m a -> m b -> m (a, b)
    (<|>) :: m a -> m a -> m a
```

Operator `(<:>)` corresponds to the sequence operator, and `(<|>)` corresponds to the alternative operator. The implementation of both operators is given bellow.

```
instance GrammO Std where
    -- (<:>) :: Std a -> Std b -> Std (a, b)
    p <:> q = do a <- p
                 b <- q
                 return (a, b)
    -- (<|>) :: Std a -> Std a -> Std a
    p <|> q = Std $ \i -> parse p i ++ parse q i
```

The sequence operator takes two parsers `p` and `q`. Parser `p` is applied first and its result is stored in `a`; then `q` is applied, its result is stored in `b`. Finally, we return a pair `(a, b)`, a binary representation of a sequence[3]. Longer sequences are represented by nested pairs, for example, see the following application.

```
a <:> b <:> c <:> d ⇒+ ((a, b), c), d)
```

We note that nesting occurs to the left, because of the left-associativity of the `(<:>)` operator. The sequence combinator fails to parse an input if any of its elements fail, think of it as an `and` over parsers.

[2] For simplicity, list comprehensions can be seen as syntactic sugar for iterators that always produce a list of values.

[3] Tuples are the "default" data type for sequences representation in many works on functional parsing.

The alternative operator defines a simple concatenation of the returns from its parameters, parsers `p` and `q`. One can think of it as an `or` over parsers. Let the results of applying `p` and `q` be the lists `xs` and `ys`, respectively, then, we have:

```
xs ++ ys = x0: ... :xn−1:ys
xs ++ [] = xs
[] ++ ys = ys
[] ++ [] = []
```

An alternative combinator will only fail if both of its parameters fail. The same reasoning can be extend to rules with an arbitrary number of alternatives.

With this we have covered all of the instances implementations. Before we proceed to discuss the implementation of the combinators synthesized by the parser generator, we define the structure of parse trees, which result from the applications of these combinators.

```
data ParseTree = Eps
               | Token Tk
               | Seq   L R
               | Rule  Label Alt
```

A `ParseTree` is composed of at least one of the following elements.

`Eps`: Represents an empty production;

`Token`: A leaf node that corresponds to a grammar terminal. The parameter for this constructor is a string `Tk`;

`Seq`: A grammar rule is composed by at least one sequence of $n \geq 1$ juxtapositions of elements. This juxtaposition is represented by the `Seq` constructor. `Seq` is a binary sub-tree with (`L`)eft and (`R`)ight branches. The recursive structure of `Seq` allows for arbitrarily big sequences, building an unbalanced `ParseTree`;

`Rule`: Corresponds to the parsing of one of the (`Alt`)ernatives of the grammar rule identified by `Label`. `Rule` acts as a sort of container for the elements of `ParseTree`.

The first "parse tree generating" combinator we define is the empty one.

```
eps :: Std ParseTree
eps = success Eps
```

Parser `eps` always succeeds[4] with an "empty" `ParseTree` as result. The name `eps` is short for epsilon, a letter from the Greek alphabet used in this work, and textbooks on formal languages, to represent an empty production.

We are still to define a combinator which consumes input.

[4] The function success is an alias for the previously defined monadic function `return`.

```
term    :: String -> Std ParseTree
term "" = error "Terminals must be non-empty strings!"
term s  = Std $ \i ->
    let n  = length s
        s' = take n i
        i' = drop n i
     in if s == s' then [(i', Token s')]
                   else [ ]
```

The combinator `term` takes a string `s` and matches it against the first `n` characters of input `i`, where `n` is the length of the parameter `s`. In case of a match, the Token `s` (or `s'`) is returned. Unlike most common parser combinators, no parametrization of a terminal value is allowed, strings are the only literal values matched or generated, as defined by the ISO EBNF standard.

The alternative operator utilized by the parser generator is the same as the one defined as instance of the `GrammO` type class, while the sequence operator is slightly different.

```
sqnc     :: Std ParseTree -> Std ParseTree -> Std ParseTree
sqnc p q = p <:> q >>= \(x, y) -> return $ Seq x y
```

The `sqnc` combinator uses operator `(<:>)`, previously defined, to parse the input, taking the resulting sequence pair and converting it into a sequence value of the `ParseTree` type.

Finally, the last of the basic combinators.

```
rule        :: Label -> Std ParseTree -> Std ParseTree
rule l alts = Std $ \i -> label l $ parse alts i
    where
        -- label :: String -> [Res ParseTree] -> [Res ParseTree]
        label l rs = [ (i', Rule l t) | (i', t) <- rs ]
```

The `rule` function acts as a wrapper combinator of sorts. It takes a label `l`, which will identify a set of alternatives `alts`. The function `label` takes the result of parsing the alternatives of `rule` and "encapsulates" them into a new result. If the original result was a failure, `label` behaves like the identity function. Otherwise, it will take the resulting parse tree `t` and "put it inside" a new parse tree `Rule` labeled by `l`. No change to the structure of `t` occurs.

As will be discussed in Chapter 4, one of the measures to provide an easily verifiable implementation of a parser generator is to define a set of combinators describing a one-to-one mapping between EBNF operators and parser combinators. To achieve this one-to-one relation we need a few extra combinators.

The missing combinators are the ones corresponding to the definition of optional sub-rules ('[ ]'), as well as unbound ('{}') and finite repetition ('*'). We define their relative combinators as follows.

First, the optional (`opt`) combinator, where we first make use of the `rule` parser, by way of its infix notation `(=!>)`.

```
opt   :: Std ParseTree -> Std ParseTree
opt p = "Optional" =!> p <|> eps
```

The use of a standard label, in this case "`Optional`", will allow pattern matching over sub-trees generated by this parser in the implementation of the parser generator, see Chapter 4 for details.

The `closure` combinator, which implements unbound repetition.

```
closure   :: Std ParseTree -> Std ParseTree
closure p = "Closure" =!> p' <|> eps
    where
        p' = p # closure p
```

Here we see the infix alias (`#`) for the sequence (`sqnc`) combinator.

```
times     :: Int -> Std ParseTree -> Std ParseTree
times 0 _ = eps
times 1 p = p
times n p = "Repetition" =!> p'
    where
        p' = p # (n - 1) *. p
```

Finally, the combinator for finite repetition, where the operator `(*.)` is the infix version of the function `times`.

## 3.2 CPS COMBINATORS

Pushing onwards in our path to the implementation of GLL combinators we take our second step, rewriting the introduced parser combinators into CPS combinators. Intuitively, as suggested by its name, a continuation is a function that tell us what to do next, or rather what to do next with the result of a previous computation.

A continuation can have various forms[5]. It can be defined as a simple identity function type `a -> a`, or be something a bit more general `a -> b`, or even incorporate some type of context `a -> m b`. If we recall the monadic `bind` definition, from Section 3.1, we can observe the pattern `a -> m b` as the second parameter of that function. We define our continuation parser type in a similar fashion[6].

```
newtype KP m a = KP (forall b. (a -> m b) -> m b)
```

In the same way that the function `f`, parameter of the standard parser monadic `bind`, takes the result from a previous parser application and returns a new parser, a continuation

[5] The term "form" is loosely used here to refer to a general idea of a function behavior, based on his type alone, and not his actual body.

[6] The use of `forall` in the definition of `KP` requires the ghc extension `RankNTypes`. We try to avoid "language extensions" as much as possible. This is the only one in the whole implementation.

parser `KP` takes a value of type `a` and returns a parser `m b`, which will then continue the parsing process.

A continuation is made into a parser by the implementation of the `parse` interface from the `Parse` type class.

```
instance (Monad m, Parser m) => Parser (KP m) where
    parse (KP p) i = parse (p return) i
```

CPS functions do not return directly to their callers, instead, they take an extra parameter, a continuation that will take their computed result (28). In the implementation of the `parse` function, the monadic `return` acts as the continuation argument of parser `p`. In fact, any other function of type `a -> m b` could be used.

Much of the code presented in Section 3.1 is replicated by the CPS combinators, one could compare their implementations, the sources are available at <https://bitbucket.org/claytown/hgll-re>. Because of this similarity, we choose not to list the entirety of the code for the implementation of the CPS combinators, and discuss only what we consider to be the most relevant components, significantly different parsers.

We start with the implementation of the `Monad` instance.

```
instance Monad (KP m) where
    -- return :: a -> (KP m) a
    return v   = KP $ \k -> k v
    -- (>>=) (KP m) a -> (a -> (KP m) b) -> (KP m) b
    KP p >>= f = KP $ \k -> p (\a -> let KP q = f a in q k)
```

The function `return`, instead of returning a `Res a`, it applies a continuation `k` to its parameter `a`. If we substitute `k` by our continuation `return`, from `parse`, we have `return v`, a parser of type `m b`.

In the `bind` function, the continuation, argument of `p`, is a lambda that takes the computed value `a`, from `p`, and applies `f` to it, the resulting parser `q` is applied to the continuation `k`, closing the sequence.

Even as CPS functions, if we compare the combinators and descriptions just given, to the standard combinators, we can notice their great similarity, despite some differences in code. If the standard and CPS combinators are so similar, why use CPS? Continuations can be used as a tool to achive backtracking functionality in a functional language, and backtracking is a way for achieving non-determinism in the implementation of parser combinators (28, 33). Also, the combination of continuations and memoization allows top-down parsers to terminate even when processing left recursive grammars (33).

## 3.3 MEMOIZED COMBINATORS

Before going into details on the limitations of this work to deliver GLL combinators, we try to understand how combinators relate to GLL, a technique that could not seem more

apart. If we first look at the relation between traditional, imperative, RD parsing, this relation should be more obvious than one would expect.

Conventional RD parsers translate grammars into functions and conditionals. Combinators do the same, and so does GLL, only that it makes the control flow of the parser explicit. Looking back at the structure of a GLL descriptor, its first element, the label, is just a reference to a function. The second element of a descriptor, a stack of labels, is the explicit representation of a parser call stack, storing references to where this parser must return. The third element is an index to the remaining input to be processed by a parser.

Now imagine the combinators as GLL descriptors. The reference to a parsing function, is the combinator itself. The call stack of a parser is implicitly controlled by the system or interpreter. The input index is the only constant parameter of a parser combinator definition. Further on, the so-called return labels represent procedures applied after the return of some other parsing procedure. Their relation to the continuations introduced in Section 3.2 should be intuitive enough.

The difference between the combinators discussed so far and GLL lies in two main elements. First, they do not apply a especial procedure for dispatch control, such as label $L_0$. Also, they lack the tools to cut out left-recursion, especially memoization.

To our knowledge, all works that implemented the GLL technique utilizing combinators did it in what could be called a "mostly functional" manner. Although their combinators are "functional", they rely on non-functional machinery to purge left-recursion. However, a recent work, which we discuss in Section 5.1, has an approach that is different from these previous works.

Mark Johnson (33) introduced a technique that combined continuations and memoization to handle left-recursion in top-down parsers based on combinators. Johnson's technique can be related to most of the works we discuss in this section, what complicates the understanding of what makes a set of combinators GLL. We elaborate on this topic in Section 5.1, for now we focus on the works derived from Johnson's approach.

Johnson starts by defining a set of combinators, very similar to the set we implement in Section 3.1. After presenting the setbacks of his set of combinators, he introduces a memoization procedure, which at first aims only to reduce the number of redundant computations performed by backtracking parsers.

This first memoization procedure is a simple wrapper function that takes a parser (unmemoized) and returns a memoized version of it. The memoized parser is a function which defines a shared data structure that maps the parser arguments into its result. Every time a memoized parser is called, it checks if a result was previously computed for his set of arguments. If true, the stored value is returned, otherwise the parser is applied to its arguments and the result is stored in the shared map before it is returned.

The memoization just described cannot cope with left-recursion because the unmemoized parser is called before an entry is created in the map, thus leading into non-

termination. The Scheme code bellow comes from Johnson's paper and illustrates this situation. The unmemoized parser `fn` is called in the first line, while the map update should happen in the second line.

```
(let ((result (apply fn args)))
(set! alist (cons (cons args result) alist))
```

To be able to implement left-recursive grammars both, the original set of combinators and the memoization machinery must change. A new set of CPS combinators is implemented. Finally, memoization is adapted to accommodate continuations and multiple results per parser, since CPS combinators introduce non-deterministic behavior.

The memoization still defines and relies on a map data structure, still indexed by the arguments of a parser, except that now each entry of the map is composed by a pair of sets. One is a set of results. The other one is a set of all continuations passed as argument to the parser identified by the entry index.

When a memoized procedure is called it checks whether it has already been called with the current set of arguments. If not, the current continuation argument is added to the memoization map. Then the unmemoized parser is called to process the input. Its result will be passed to another continuation which will then store the result[7].

If the memoized parser was previously called for a particular set of arguments the current continuation argument is stored, and the continuation is applied to each of the results in the corresponding map entry. No call is made to the unmemoized procedure.

Even though some of the details about the memoization procedure were hid, the main concepts for handling left-recursion were covered. We can question why do we spend such an effort on a technique that predates the GLL algorithm by about 15 years? Well, because most of the works that implement GLL combinators make use of the same or very similar concepts.

Johnson's technique is attractive not only because of its simplicity and generality, but also, because of the way memoization is implemented and utilized, which helps to preserve a certain familiarity, common to the implementation of combinators amongst many works. It helps to reason about their code. Also, at the time of the development of this work, we were not aware of any other technique for the functional implementation of GLL.

We have not succeeded to implement GLL in Haskell. We discuss the pitfalls in what follows. Let us start with considerations regarding shared mutable states. In his work Spiewack (21) states "The primary motivation for the mutable state was convenience not necessity.". We dare to disagree.

To address this affirmation, let us revisit the implementation of parser combinators, forget about monads, type classes, and so on. Let us take a look at a simple deterministic parser, similar to the first set of combinators from Johnson (33), and others (21, 29, 34).

[7] Some steps are omitted for simplicity. Please refer to the original article if necessary (33).

```
type Parser a = String -> Maybe (String, a)
```

A parser is a function that takes an input `i`, and may or may not return a result. After we finished implementing all our standard combinators, we want to extend them by introducing continuations. Here we use the simplest continuation type possible.

```
type K a = a -> a
```

We redefine our parser type to accommodate the new parameter.

```
type Parser a = String -> K a -> Maybe (String, a)
```

Now our GLL parser combinators are almost done, we just need to think about memoization. First, we need a data structure, where to store continuations and parser results.

```
type MemoTable a = Map Args ([K a], [a])
```

Our memo data structure is a `Map`, where each entry is indexed by a set of parser arguments `Args`. Each entry in `MemoTable` is composed of a pair. The first element of the pair is a list of all the continuations passed to a certain parser. The second element is the `Set` of values computed by this parser. The only thing missing is the memoization procedure, we call it `memo`.

```
1  memo   :: Parser a -> Parser a
2  memo p = do
3      let table = Map.empty
4      \Args k -> case Map.lookup Args table of
5          Just (ks, as) -> do -- p has been called with Args before
6              let ks'   = k:ks
7              Map.insert Args (ks', as) table
8              fmap k as
9          _             -> do
10             let (ks, as) = ([], [])
11             let ks'      = k:ks
12             Map.insert Args (ks', as) table
13             p Args $ \a ->
14                 if a `elem` as then
15                     applyKs ks' a
16                 else do
17                     let as' = a:as
18                     Map.insert Args (ks', as') table
19                     applyKs ks' a
```

The implementations of `memo` that we show, are not valid Haskell code, they are an attempt to emulate Johnson's memoization procedure in Haskell, as close as possible to his work. We elaborate on the setbacks we encounter when trying to adapt it to this work.

Starting at Line 3, we define our memo table, which may be updated at Lines 7, 12 and 18. Well, Haskell does not support mutable data, every data "modification" actually corresponds to the creation of a new data structure representing the updated state of a computation. Line 3 is the representation of what happens in many other mostly functional implementations: the definition of a local, mutable "persistent" data structure, which is consulted and updated at each call to a parser. This allows a much more convenient implementation of the combinators, which are almost not affected by the introduction of memoization.

Spiewack (21) observes in his work: "If we were implementing GLL in Haskell we would likely return a modified `Trampoline` [...] rather than modifying its data structures in-place"[8]. This is how one would implement stateful computations in a pure FL, we must explicitly thread the computation state around.

Again we need to modify the `Parser` type.

```
Parser a = String -> MemoTable a -> K a -> (MemoTable a, [(String, a)])
```

`Parser` now takes a `MemoTable` as argument, but also returns a `MemoTable`, and has a non-deterministic behavior, indicated by the return of a list of results instead of a `Maybe`. We modify `memo` to accommodate these changes.

```
1  memo   :: Parser a -> Parser a
2  memo p = \Args table k ->
3      case Map.lookup Args table of
4          Just (ks, as) -> -- p has been called with Args before
5              let ks'    = k:ks
6                  table' = Map.insert Args (ks', as) table
7               in (table', fmap k as)
8          _             ->
9              let (ks, as) = ([], [])
10                 ks'      = k:ks
11                 table'   = Map.insert Args (ks', as) table
12              in p Args table' $ \a ->
13                     if a `elem` as then
14                         (table', applyKs ks' a)
15                     else
16                         let as'     = a:as
17                             table'' = Map.insert Args (ks', as') table'
18                          in (table'', applyKs ks' a)
```

It looks like we solved our problems regarding mutable state, it seems Spiewak's observation on the matter was correct.

[8] `Trampoline` is Spiewak's implementation of label $L_0$. Is a class that defines important control data structures, and is responsible for the dispatch of parsing procedures. In a way, this is where memoization happens.

However, if we look carefully at the latter `memo` code we can notice some type related issues. Remember the definition of the continuation type. It is an identity type function `a -> a`. Now consider Lines 7, 14, and 18, where we return the parsing result. In the second element of those pairs, we either apply a continuation to a list of values of type `a`, or apply a list of continuations to a value of type `a`, in both cases returning `[a]`. But this is not the expected type `(MemoTable a, [(String, a])`. We modify the continuation type to fix this problem.

```
type K a = a -> [(String, a)]
```

This is not quite right as expected. Let us first talk about function `applyKs`, which can be found at Lines 14 and 18. Function `applyKs`, as its name implies, applies each continuation from list `ks'` to value `a`. Now that a continuation returns a list, `applyKs` returns `[[(String, a)]]`. A simple fix would be `concat $ applyKs ks' a`, which would "flatten" the return of `applyKs` into a single list of results, the expected return type.

Another problem, this with greater implications, can be observed at Line 12. The parser `p` application takes advantage of the fact that a parser takes a continuation, passing `p` a continuation responsible for memoizing result `a` (the lambda expression after the `'$'` operator) and propagate it to the continuations that where arguments of `p` before. The problem has to do with the return type of both parser `p` and continuation argument, they must match, but they do not.

We could try and make the types of `p` and continuations "work out" in a couple of ways. Changing the return type of the continuation to match the parser return would not work, since the return pair would have type.

```
(MemoTable a, [(MemoTable a, [(String, a)])])
```

Trying to make the parser type match this would only lead to a circular issue.

Maybe we could try to implement the last six lines of `memo` in a different way.

```
1  let (table'', rs) = p Args table' c
2      a             = snd $ head rs
3      as'           = a:as
4      table'''      = Map.insert Args (ks', as') table''
5   in (table''', applyKs ks' a)
```

This solution assumes a continuation `c :: a -> (MemoTable a, [(String, a)])` exists. Not only that, it also assumes the `head` (first) element of the list of results `rs` to be the appropriate pair to extract value `a` from, and still the problem remains. We could extract a result `r` from an arbitrarily chosen element from those returned `applyKs`, like we did for `a`, but we have no guarantees that the chosen `r` is the correct return of the memoized version of parser `p`.

After all we discussed so far, we only addressed issues in the implementation of memoization. Assuming the solutions proposed previously implemented a reliable version of `memo`. Now consider the implementation of the alternative combinator, which was implemented in Section 3.1 as a simple concatenation of the results of its parameters parsers `p` and `q`. Imagine then the implementation of the alternative combinator for the memoized parser type we just developed, remember, we have to "combine" two results of type `(MemoTable a, [(String, a)])`.

Starting with the second element of the pair, it seems really simple, we can just apply `(++)`, as we did for the standard combinator. What about the first element of the results, how can we combine or choose one `MemoTable`? What if our parser type is just wrong? What if `Parser` returned something like `[(MemoTable a, String, a)]`? This type of result implies that each parser procedure would carry its own instance of a memo table. If we consider the example of the implementation of `alt`, this seems promising, we could simply apply `(++)` without having to worry about MemoTable. Still, the type issue of memoization would remain, happening again at the last element of the result tuple. If we try to apply the continuations stored in the memo table we end up with some type `[[a]]`. What about type `[MemoTable a, (String, a)]`? Here we loose track of the appropriate value `a` to pass to `applyKs`, can we choose one arbitrarily? We stop here, going through all imaginable "solutions" is simply unviable.

Another very impactful feature from imperative and hybrid languages is represented in the code of `memo` by the keyword `Args`. Memoization implemented in some dialects of Scheme takes advantage of the ability to index data structures by a list of parameters[9], which is key to avoid left-recursion, allowing the memoization procedure to identify each parser uniquely, avoiding calls to parsers already encountered. Spiewack relies on the fact that Scala functions are objects and can be identified by their references, also to avoid left-recursion.

Afroozeh *et al.* avoid indexing the memo table with references to parsers by having each "parser" implement their own internal memoization. Again, this is only possible because of mutable data structures, combined with the ability to extend types and define continuations as functions of type `Unit`.

Haskell also has a type unit, which is normally used in monadic contexts to indicate that a function produces side effects, and that its return is disposable or that we are not interested in it. The `State` monad is the "default" mechanism for the representation of stateful computations.

Remember, the `State` monad is only a tool for abstraction of the explicit threading of a computation state. If anything, it makes it harder to accommodate continuations. Also, the original motivation leading us to consider a solution using `State` was the possibility

[9] We are not sure of the inner workings of such PLs, but is reasonable to believe this is possible via some sort of memory reference.

of abstracting the problematic return type of continuation `K` with Haskell unit type `()`.

However, in the scenery where we would use `State` as a solution, the memoization of a parser state would return something like `State ... [a]`, whilst the return of a side effect continuation wold be `State ... ()`. This represents the idea that we are interested in the values returned by a parser, at the same time that we are only interested in the side effects of continuations, and this is conflicting. Another interesting phenomenon that may arise from a solution based on `State` has to do with the storage of states in the memo table, in other words, a `State` may store instances of itself.

After all these struggles, we could not find a solution for the type mismatches between the many combinators and memoization components. Some GLL implementations do not face this sort of type issue, since their supporting languages are Scheme (33) and Racket (29). For the works of Spiewack and Afroozeh *et al.*, which are implemented in Scala, we argue that being able to utilize shared, mutable, data structures associated with some other features, is not only a convenience, but is core to allow the implementation of this sort of memoization.

# 4 A COMBINATOR BASED PARSER GENERATOR

> *No amount of source-level verification or scrutiny will protect you from using untrusted code.*
>
> Ken Thompson, *1984*

In Chapter 3, we showed the implementation of a set of parser combinators for the ISO EBNF standard, and provided insight about pitfalls in the naive implementation of GLL combinators in a purely functional setup.

We dedicate this chapter to the discussion of the contributions related to the parser generator and its reliability. In Section 4.1 we discuss how the generator implementation is organized, and why we believe this implementation is not only intuitively reliable, but also suitable for formal verification. Section 4.2 show results produced by the parser generator, as well as evidence of its reliability.

## 4.1 PARSER GENERATOR

Since one of the objectives we have is to provide an easily verifiable parser generator, we deliberately tried to organize and implement the components of our parsing solution in a systematic way.

The implementation of combinators, as mentioned in Chapter 3, is separated in two levels. The first level defines instances of monad and parser related type classes, where combinators are polymorphic according to those type classes specification. The second level is where we mostly define aliases, or utilize operators defined in the first level, to implement the combinators necessary to represent the ISO EBNF set of operators.

This separation has two main benefits. First, combinators defined in the first level can be utilized independently, for example, as a library for writing parsers via combinators, and can be extended without affecting components in the second level. Second, this flexibility contributes for the reliability goal, once this base is verified as reliable, changes to the second level or any sort of extension, will not affect it.

The set of combinators defined at level two is very small, only the few necessary to establish a one-to-one correspondence with the operators defined by the EBNF standard. Working with an exact one-to-one correspondence helps to provide reliability, establishing what we call a principle of reliability, which we discuss towards the end of this section. Alas, this small set of combinators has its down side, which we discuss later in this section.

The set of combinators utilized by the parser generator produces as output a parse tree that represents the syntax of a language, which is read from an EBNF input file. The parser that processes these inputs is designed to closely resemble what is specified in

the ISO standard. This way one could effectively assert its reliability consulting the ISO documentation.

We use this opportunity to make some notes on where it was not possible, or we have chosen not, to be compliant to the standard. First, the EBNF standard defines a very small set of characters, any character not specified by the standard cannot be used as part of the input for the generator. It should be obvious that this is very limiting. Luckily Haskell provides a way around this limitation. Any Unicode character can be specified as part of a language lexical, if it is defined as a Haskell hexadecimal escaped char, `\xHHHH`, where `H` is a hexadecimal digit. An example of this feature is given is Section 4.2

While the host language may provide advantages it may also be the cause of limitations. If any character in the syntax of a language is also one of the characters Haskell expects to be escaped within a string, then this character must be escaped in the input file. For example, a backslash must be defined as the escaped character `'\\'`. The parser will fail, or produce an erroneous combinator, otherwise[1].

The ISO EBNF defines what is called a special sequence, which allows the arbitrary specification of syntactical elements. Say we wanted to define a language based on the works of Tolkien, we could define a special sequence like this `?Any Tengwar script Tengwö found in the books of Tolkien?`. Although it is possible to support a limited set of special sequences, we cannot allow this sort of looseness in the input of the generator.

Another feature we do not support is the definition by exception or difference. For example, a variable could be defined as `identifier - keyword`. This feature could actually be implemented as a binary combinator that only succeeds if its first argument succeeds and the second fails, but for now it is not supported.

The removal of special sequences and set differences has an important consequence, every rule has to explicitly specify its alternatives. One illustration of this limitation is given by rules `{first,second}TerminalCharacter` in the sources listed in Section 4.2, the only difference between these two rules is the swapping of the non-terminals `*QuoteSymbol`. However, since the definition by exception is not allowed, they must redundantly define each of their alternatives.

One last note on the EBNF standard support: since the underlying language of the parser and parser generator is Haskell, the meta-identifiers defined in the input must also be valid Haskell identifiers. This is further commented in Section 4.2.

Now, continuing with the implementation of the EBNF parser, and parser generator. The ISO standard divides the introduction of the EBNF meta syntax in three parts, each step closer to the actual syntax. We do something similar with the definition of the input parser. The first module defines the basic character set of the EBNF, next the remainder of the lexical is defined, and finally the syntax rules.

We have the same organization for the parser generator, and the relation between

[1] The double quote character is an exception, since it is internally treated as an special case.

parser and generator is similar to the relation between grammars and action objects from Perl 6[2]. What this means is, for each parser rule defined in a parser module, there is a corresponding generator function in a generator module.

This organization of parser and generator gives us a one-to-one relation. The structure of the generator code reproduces, almost exactly, the structure of the parser, a property assured by pattern matching over the structure of a parse tree. This should guarantee an indirect one-to-one relation between the input EBNF and its respective generated parser.

Although not formally verified, at least for now, we can establish a principle of reliability to build upon. For a complete reference, the following code illustrates the generator definition and the properties mentioned above, the source can be compared with the corresponding parser code in Section 4.2.

```
1  gDefinitionsList (Rule "DefinitionsList" t) = case t of
2      (Seq l r) -> gSingleDefinition l ++ gStar list r
3          where
4              list (Seq _ d) = cAltOp ++ gSingleDefinition d
```

The pattern matching at the first line of `gDefinitionList` guarantees that we will fail to generate anything except the specified rule. This pattern matching happens for all of the generator functions. Such pattern matching also allows us to match the inner alternatives according to their expected structure, exactly as specified by the EBNF parser.

There is only one pattern for `gDefinitionList` to match internally (at Line 2), since the EBNF rule `definitionList` has only one alternative. The body of the case equation `(Seq l r)` corresponds exactly to the structure of `definitionList` right-hand side, which is defined as follows.

```
singleDefinition, {alternativeSymbol, singleDefinition};
```

The function `gSingleDefinition` corresponds to the non-terminal `singleDefinition`, which is followed by the unbound repetition (`gStar`) of the sub-sequence `list`. The sub-sequence represented by `list` in the generator (at Line 4) corresponds to the sub-sequence within curly braces in the grammar above. The function `gStar` is responsible for the generation of many occurrences of the pattern described by `list`, for as long as it occurs in the sub-tree `r`, if at all. The generator `gStar` is also defined by patter matching corresponding to the structure defined by combinator `star` (or `closure`). In the body of `list`, the function `cAltOp` stands for (c)ombinator (Alt)ernative (Op)erator, which is an alias for combinator `(<|>)`.

[2] More information is available at <https://docs.perl6.org/language/grammars>.

## 4.2 VALIDATION

To provide a proof of concept, and what we dare to call evidence of reliability for the implemented parser generator, we propose a simple experiment[3].

As exposed in Section 4.1, our parser generator takes an EBNF specification and outputs the corresponding parser, which is composed of a series of combinators. We parse the input EBNF file with a parser defined utilizing the same set of combinators generated as output. It was also argued that this property is important to establish a principle of reliability for the parser generator.

From the arguments presented in Section 4.1, it is reasonable to presume that, if we feed the parser generator with an specification of the EBNF standard, the generated parser should greatly resemble the EBNF parser we defined by hand, since the later ought to be as close of a transcript of the standard as possible, within the limitations of the parser combinators.

The remaining of this section presents the results obtained from this experiment, as well as comments on some particular differences between generated and manually defined parsers. We split the presentation of the source in the same way we organized our implementation, for better readability. We have also formated the generated code for presentation as well as readability[4].

Starting with the EBNF characters set specification. We first list the EBNF input, followed by the generated code, then the manually written code. Commentary is spread between these elements, whenever needed.

```
letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k'
       | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v'
       | 'w' | 'x' | 'y' | 'z'
       | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K'
       | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V'
       | 'W' | 'X' | 'Y' | 'Z';

decimalDigit = '0' | '1' | '2' | '3' | '4'
             | '5' | '6' | '7' | '8' | '9';

concatenateSymbol     = ',';
definingSymbol        = '=';
alternativeSymbol     = '|' | '/' | '!';
endGroupSymbol        = ')';
endOptionSymbol       = ']' | '/)';
endRepeatSymbol       = '}' | ':)';
firstQuoteSymbol      = "'";
repetitionSymbol      = '*';
```

[3] For the sake of simplicity, no further considerations on the meaning and connotation of the term experiment were maid.

[4] No other alteration was made to the generated code.

```
19  secondQuoteSymbol     = '"';
20  exceptSymbol          = '-';
21  specialSequenceSymbol = '?';
22  startGroupSymbol      = '(';
23  startOptionSymbol     = '[' | '(/';
24  startRepeatSymbol     = '{' | '(:';
25  terminatorSymbol      = ';' | '.';
26
27  otherCharacter = spaceCharacter
28                 | ':' | '+' | '_' | '%' | '@' | '&' | '#'
29                 | '$' | '<' | '>' | '\\' | '^' | "" | '~';
30
31  spaceCharacter = '\x0020';
32
33  horizontalTabulationCharacter = '\t';
34  verticalTabulationCharacter   = '\v';
35
36  formFeed       = '\f';
37  carriageReturn = '\r';
38
39  newLine = {carriageReturn}, '\n', {carriageReturn};
```

Here the definitions of rules `letter` (Line 1) and `decimalDigit` (Line 8) illustrate the necessity for explicit specification of characters sets, since there is no support for elaborate RegExs, built-in sets of characters, or special sequences.

At Line 29 we observe an example of the necessity for escaping certain characters, due to how Haskell strings work. If a character being defined in the EBNF must be escaped in a Haskell string, then it must be escaped in the EBNF as well. Lines 31 and 33 illustrate the ability to define Unicode characters and non-printable "control" characters, respectively. Again, within Haskell capabilities.

```
1  letter = "letter" =|>
2           t "a" <|> t "b" <|> t "c" <|> t "d" <|> t "e" <|> t "f"
3       <|> t "g" <|> t "h" <|> t "i" <|> t "j" <|> t "k" <|> t "l"
4       <|> t "m" <|> t "n" <|> t "o" <|> t "p" <|> t "q" <|> t "r"
5       <|> t "s" <|> t "t" <|> t "u" <|> t "v" <|> t "w" <|> t "x"
6       <|> t "y" <|> t "z"
7       <|> t "A" <|> t "B" <|> t "C" <|> t "D" <|> t "E" <|> t "F"
8       <|> t "G" <|> t "H" <|> t "I" <|> t "J" <|> t "K" <|> t "L"
9       <|> t "M" <|> t "N" <|> t "O" <|> t "P" <|> t "Q" <|> t "R"
10      <|> t "S" <|> t "T" <|> t "U" <|> t "V" <|> t "W" <|> t "X"
11      <|> t "Y" <|> t "Z"
12
13 decimalDigit = "decimalDigit" =|>
14                t "0" <|> t "1" <|> t "2" <|> t "3" <|> t "4"
15            <|> t "5" <|> t "6" <|> t "7" <|> t "8" <|> t "9"
```

```
16
17  concatenateSymbol     = "concatenateSymbol"     =|> t ","
18  definingSymbol        = "definingSymbol"        =|> t "="
19  endGroupSymbol        = "endGroupSymbol"        =|> t ")"
20  endOptionSymbol       = "endOptionSymbol"       =|> t "]" <|> t "/)"
21  endRepeatSymbol       = "endRepeatSymbol"       =|> t "}" <|> t ":)"
22  firstQuoteSymbol      = "firstQuoteSymbol"      =|> t "'"
23  repetitionSymbol      = "repetitionSymbol"      =|> t "*"
24  secondQuoteSymbol     = "secondQuoteSymbol"     =|> t "\""
25  exceptSymbol          = "exceptSymbol"          =|> t "-"
26  specialSequenceSymbol = "specialSequenceSymbol" =|> t "?"
27  startGroupSymbol      = "startGroupSymbol"      =|> t "("
28  startOptionSymbol     = "startOptionSymbol"     =|> t "[" <|> t "(/"
29  startRepeatSymbol     = "startRepeatSymbol"     =|> t "{" <|> t "(:"
30  terminatorSymbol      = "terminatorSymbol"      =|> t ";" <|> t "."
31  alternativeSymbol     = "alternativeSymbol"     =|>
32      t "|" <|> t "/" <|> t "!"
33
34  otherCharacter = "otherCharacter" =|>
35                   spaceCharacter
36               <|> t ":" <|> t "+" <|> t "_" <|> t "%" <|> t "@"
37               <|> t "&" <|> t "#" <|> t "$" <|> t "<" <|> t ">"
38               <|> t "\\" <|> t "^" <|> t "" <|> t "~"
39
40  spaceCharacter = "spaceCharacter" =|> t "\x0020"
41
42  horizontalTabulationCharacter = "horizontalTabulationCharacter" =|>
43      t "\t"
44  verticalTabulationCharacter   = "verticalTabulationCharacter"   =|>
45      t "\v"
46
47  formFeed       = "formFeed"       =|> t "\f"
48  carriageReturn = "carriageReturn" =|> t "\r"
49
50  newLine = "newLine" =|>
51      closure (carriageReturn) # t "\n" # closure (carriageReturn)
```

A note on a redundant pattern we can observe on the generated parser. Each function definition is followed by a labeled rule with the exact same name of that function, what may cause some strangeness. Those labels are a flexible way to allow secure pattern matching. Notice that they are used in the handwritten parser as well.

```
1  letter = "Letter" =!>
2           t "a" <|> t "b" <|> t "c" <|> t "d" <|> t "e" <|> t "f"
3       <|> t "g" <|> t "h" <|> t "i" <|> t "j" <|> t "k" <|> t "l"
4       <|> t "m" <|> t "n" <|> t "o" <|> t "p" <|> t "q" <|> t "r"
5       <|> t "s" <|> t "t" <|> t "u" <|> t "v" <|> t "w" <|> t "x"
```

```
6         <|> t "y" <|> t "z"
7         <|> t "A" <|> t "B" <|> t "C" <|> t "D" <|> t "E" <|> t "F"
8         <|> t "G" <|> t "H" <|> t "I" <|> t "J" <|> t "K" <|> t "L"
9         <|> t "M" <|> t "N" <|> t "O" <|> t "P" <|> t "Q" <|> t "R"
10        <|> t "S" <|> t "T" <|> t "U" <|> t "V" <|> t "W" <|> t "X"
11        <|> t "Y" <|> t "Z"
12
13  decimalDigit = "DecimalDigit" =!>
14                 t "0" <|> t "1" <|> t "2" <|> t "3" <|> t "4"
15              <|> t "5" <|> t "6" <|> t "7" <|> t "8" <|> t "9"
16
17  concatenateSymbol     = "ConcatenateSymbol"     =!> t ","
18  definingSymbol        = "DefiningSymbol"        =!> t "="
19  endGroupSymbol        = "EndGroupSymbol"        =!> t ")"
20  endOptionSymbol       = "EndOptionSymbol"       =!> t "]" <|> t "/)"
21  endRepeatSymbol       = "EndRepeatSymbol"       =!> t "}" <|> t ":)"
22  exceptSymbol          = "ExceptSymbol"          =!> t "-"
23  firstQuoteSymbol      = "FirstQuoteSymbol"      =!> t "'"
24  repetitionSymbol      = "RepetitionSymbol"      =!> t "*"
25  secondQuoteSymbol     = "SecondQuoteSymbol"     =!> t "\""
26  specialSequenceSymbol = "SpecialSequenceSymbol" =!> t "?"
27  startGroupSymbol      = "StartGroupSymbol"      =!> t "("
28  startOptionSymbol     = "StartOptionSymbol"     =!> t "[" <|> t "(/"
29  startRepeatSymbol     = "StartRepeatSymbol"     =!> t "{" <|> t "(:"
30  terminatorSymbol      = "TerminatorSymbol"      =!> t ";" <|> t "."
31  alternativeSymbol     = "AlternativeSymbol"     =!>
32      t "|" <|> t "/" <|> t "!"
33
34  otherCharacter = "OtherCharacter" =!>
35                   spaceCharacter
36               <|> t ":" <|> t "+" <|> t "_" <|> t "%" <|> t "@"
37               <|> t "&" <|> t "#" <|> t "$" <|> t "<" <|> t ">"
38               <|> t "\\" <|> t "^" <|> t "`" <|> t "~"
39
40  spaceCharacter = "SpaceCharacter" =!> t " "
41
42  newLine = "NewLine" =!>
43      star carriageReturn # t "\n" # star carriageReturn
44
45  horizontalTabulationCharacter = "HorizontalTabulationCharacter" =!>
46      t "\t"
47  verticalTabulationCharacter   = "VerticalTabulationCharacter"   =!>
48      t "\v"
49
50  formFeed       = "FormFeed"       =!> t "\f"
51  carriageReturn = "CarriageReturn" =!> t "\r"
```

Looking at the machine generated (starting at Page 61) and handwritten (starting at Page 62) sources, barely no difference can be noticed. For the differences that do exist we provide clarification.

We start with the difference on the rule operators. A closer look reveals that the generate code defines non-deterministic rules, via operator `(=|>)`, while the manual implementation uses its deterministic variant `(=!>)`. For the parser generation to work we need a deterministic parse of the input, whilst we aim to generate a non-deterministic parser as output, therefore, the difference.

Another difference related to the definition of rules has to do with their labels. Labels from handwritten rules start with an upper-case character, instead of a lower-case, they are otherwise identical. This happens because labels are generated from the rule meta-identifier defined in the input file, which must be a valid Haskell identifier, the same is true for the generated Haskell function. Extra steps could be added to the generation process to fix this issue.

Notice that the rule `newLine` is located at different points of the Haskell sources, this is not at all an issue. The generated `newLine` function occurs in the exact relative location where `newLine` is specified by the EBNF. Another difference can be observed in the `newLine` rule, but we leave the discussion for later in this section.

One last note, the `spaceCharacter` rules are different, they define different terminals. Actually, the hexadecimal sequence `\x0020` is the Unicode codification of the space character, so both rules define the same terminal. Again, the generated code corresponds exactly to what is specified by the input, in this case, a hexadecimal codification.

```
1  terminalString = firstQuoteSymbol, firstTerminalCharacter
2                 , {firstTerminalCharacter}, firstQuoteSymbol
3                 | secondQuoteSymbol, secondTerminalCharacter
4                 , {secondTerminalCharacter}, secondQuoteSymbol;
5
6  firstTerminalCharacter = letter
7                         | decimalDigit
8                         | concatenateSymbol
9                         | definingSymbol
10                        | alternativeSymbol
11                        | endGroupSymbol
12                        | endOptionSymbol
13                        | endRepeatSymbol
14                        | exceptSymbol
15                        | repetitionSymbol
16                        | secondQuoteSymbol
17                        | specialSequenceSymbol
18                        | startGroupSymbol
19                        | startOptionSymbol
20                        | startRepeatSymbol
```

```
                        | terminatorSymbol
                        | otherCharacter;

secondTerminalCharacter = letter
                        | decimalDigit
                        | concatenateSymbol
                        | definingSymbol
                        | alternativeSymbol
                        | endGroupSymbol
                        | endOptionSymbol
                        | endRepeatSymbol
                        | exceptSymbol
                        | firstQuoteSymbol
                        | repetitionSymbol
                        | specialSequenceSymbol
                        | startGroupSymbol
                        | startOptionSymbol
                        | startRepeatSymbol
                        | terminatorSymbol
                        | otherCharacter;

integer                 = decimalDigit, {decimalDigit};
metaIdentifier          = letter, {metaIdentifierCharacter};
metaIdentifierCharacter = letter | decimalDigit;
```

The corresponding generated code.

```
terminalString = "terminalString" =|>
                   firstQuoteSymbol # firstTerminalCharacter
                                    # closure (firstTerminalCharacter)
                                    # firstQuoteSymbol
               <|> secondQuoteSymbol # secondTerminalCharacter
                                     # closure (secondTerminalCharacter)
                                     # secondQuoteSymbol

firstTerminalCharacter = "firstTerminalCharacter" =|>
                           letter
                       <|> decimalDigit
                       <|> concatenateSymbol
                       <|> definingSymbol
                       <|> alternativeSymbol
                       <|> endGroupSymbol
                       <|> endOptionSymbol
                       <|> endRepeatSymbol
                       <|> exceptSymbol
                       <|> repetitionSymbol
                       <|> secondQuoteSymbol
                       <|> specialSequenceSymbol
```

```
22                        <|> startGroupSymbol
23                        <|> startOptionSymbol
24                        <|> startRepeatSymbol
25                        <|> terminatorSymbol
26                        <|> otherCharacter
27
28  secondTerminalCharacter = "secondTerminalCharacter" =|>
29                              letter
30                         <|> decimalDigit
31                         <|> concatenateSymbol
32                         <|> definingSymbol
33                         <|> alternativeSymbol
34                         <|> endGroupSymbol
35                         <|> endOptionSymbol
36                         <|> endRepeatSymbol
37                         <|> exceptSymbol
38                         <|> firstQuoteSymbol
39                         <|> repetitionSymbol
40                         <|> specialSequenceSymbol
41                         <|> startGroupSymbol
42                         <|> startOptionSymbol
43                         <|> startRepeatSymbol
44                         <|> terminatorSymbol
45                         <|> otherCharacter
46
47  integer = "integer" =|> decimalDigit # closure (decimalDigit)
48
49  metaIdentifier = "metaIdentifier" =|>
50      letter # closure (metaIdentifierCharacter)
51
52  metaIdentifierCharacter = "metaIdentifierCharacter" =|>
53                              letter
54                         <|> decimalDigit
```

The generated code shown above corresponds to the remainder of the EBNF lexical specification. It is quite long due to rules `*TerminalCharacter`, because of that, and because we want to provide some mechanized evidence of the similarity between generated and written parsers, instead of showing the code of the handwritten parser we show the output of the sources differences for both parsers[5].

```
1  1c1
2  < terminalString = "terminalString" =|>
3  ---
4  > terminalString = "TerminalString" =!>
```

[5] The difference check was performed with the command `diff`, provided by GNU package diffutils <https://gnu.org/software/diffutils>.

```
3c3
<                                    # closure (firstTerminalCharacter)
---
>                                    # star firstTerminalCharacter
6c6
<                                     # closure (secondTerminalCharacter)
---
>                                     # star secondTerminalCharacter
9c9
< firstTerminalCharacter = "firstTerminalCharacter" =|>
---
> firstTerminalCharacter = "FirstTerminalCharacter" =!>
28c28
< secondTerminalCharacter = "secondTerminalCharacter" =|>
---
> secondTerminalCharacter = "SecondTerminalCharacter" =!>
47c47
< integer = "integer" =|> decimalDigit # closure (decimalDigit)
---
> metaIdentifier = "MetaIdentifier" =!> letter # star metaIdentifierCharacter
49,52c49
< metaIdentifier = "metaIdentifier" =|>
<     letter # closure (metaIdentifierCharacter)
<
< metaIdentifierCharacter = "metaIdentifierCharacter" =|>
---
> metaIdentifierCharacter = "MetaIdentifierCharacter" =!>
54a52,53
>
> integer = "Integer" =!> decimalDigit # star decimalDigit
```

This might look like a lot of more differences than expected, but this output is misleading. From Line 1 down to Line 20, all the indicated differences are related to characters case, deterministic versus non-deterministic operators, and aliases, all of which were previously clarified. The remaining differences regard the formating of rule metaIdentifier, and the swapping of positions in source, between rules `integer` and `metaIdentifier`.

The next sources are listed one immediately after the other, with no interruptions.

```
syntax = syntaxRule, {syntaxRule};

syntaxRule =
    metaIdentifier, definingSymbol, definitionsList, terminatorSymbol;

definitionsList =
    singleDefinition, {alternativeSymbol, singleDefinition};

singleDefinition =
```

```
    syntacticFactor, {concatenateSymbol, syntacticFactor};

syntacticFactor = [integer, repetitionSymbol], syntacticPrimary;

syntacticPrimary = optionalSequence
                 | repeatedSequence
                 | groupedSequence
                 | metaIdentifier
                 | terminalString
                 | emptySequence;

optionalSequence = startOptionSymbol, definitionsList, endOptionSymbol;
repeatedSequence = startRepeatSymbol, definitionsList, endRepeatSymbol;
groupedSequence  = startGroupSymbol, definitionsList, endGroupSymbol;
emptySequence    = eps;
```

```
syntax     = "syntax"     =|> syntaxRule # closure (syntaxRule)
syntaxRule = "syntaxRule" =|>
    metaIdentifier # definingSymbol # definitionsList # terminatorSymbol

definitionsList = "definitionsList" =|>
    singleDefinition # closure (alternativeSymbol # singleDefinition)

singleDefinition = "singleDefinition" =|>
    syntacticFactor # closure (concatenateSymbol # syntacticFactor)

syntacticFactor = "syntacticFactor" =|>
    opt (integer # repetitionSymbol) # syntacticPrimary

syntacticPrimary = "syntacticPrimary" =|>
                    optionalSequence
                <|> repeatedSequence
                <|> groupedSequence
                <|> metaIdentifier
                <|> terminalString
                <|> emptySequence

optionalSequence = "optionalSequence" =|>
    startOptionSymbol # definitionsList # endOptionSymbol

repeatedSequence = "repeatedSequence" =|>
    startRepeatSymbol # definitionsList # endRepeatSymbol

groupedSequence = "groupedSequence" =|>
    startGroupSymbol # definitionsList # endGroupSymbol

emptySequence = "emptySequence" =|> eps
```

```
1  syntax     = "Syntax"     =!> syntaxRule # star syntaxRule
2  syntaxRule = "SyntaxRule" =!>
3      metaIdentifier # definingSymbol # definitionsList # terminatorSymbol
4
5  definitionsList = "DefinitionsList" =!>
6      singleDefinition # star (alternativeSymbol # singleDefinition)
7
8  singleDefinition = "SingleDefinition" =!>
9       syntacticFactor # star (concatenateSymbol # syntacticFactor)
10
11 syntacticFactor = "SyntacticFactor" =!>
12      opt (integer # repetitionSymbol) # syntacticPrimary
13
14 syntacticPrimary = "SyntacticPrimary" =!>
15                       optionalSequence
16                   <|> repeatedSequence
17                   <|> groupedSequence
18                   <|> emptySequence
19                   <|> metaIdentifier
20                   <|> terminalString
21
22 optionalSequence = "OptionalSequence" =!>
23     startOptionSymbol # definitionsList # endOptionSymbol
24
25 repeatedSequence = "RepeatedSequence" =!>
26     startRepeatSymbol # definitionsList # endRepeatSymbol
27
28 groupedSequence = "GroupedSequence" =!>
29     startGroupSymbol # definitionsList # endGroupSymbol
30
31 emptySequence = "EmptySequence" =!> t "eps"
```

The differences between generated and written code we can observe on both sources above are the same we observed along this section, but we have two differences left to cover. Both instances of generated code, for the first two components of this work EBNF definition, have single non-terminals occurring between parentheses, for example `(decimalDigit)`, while hand written code has no parentheses for the same productions.

The reason for the surrounding parentheses is the same reason why the sequence `(integer # repetitionSymbol)`, and others, occur within parentheses in the generated and manually written codes above. Such sub-rules are preceded by a function implementing optional or repetition operators. The generator guarantees the correct application of these functions by grouping the argument sub-rule within parentheses.

Another note regarding the grouped sub-rules and the function preceding them in the code. Notice that the unbound repetition operator has different names from generated to manually written code; the former uses `closure`, whereas the latter uses `star`. Both are

equivalent; `star` being an alias of `closure`.

Finally, we believe that multiple evidences and arguments are provided in this section as means to support the argument about reliability made in Section 4.1. There are differences between the EBNF parser generated and the one written for this work, but they are few and harmless in regards to the parser semantics. Also, the similarity between the input grammar and its corresponding generated parser is remarkable, which is another indicator of reliability.

This work includes a series of appendixes where more examples of grammars and their respective generated parsers can be found, they are too long and no further benefit would come from listing them here. Nevertheless, these specifications and their corresponding parsers, are useful to validate the developed parser generator.

# 5 CONCLUSIONS

> *Life gets boring, someone invents another necessity, and once again we turn the crank on the screwjack of progress hoping that nobody gets screwed.*
>
> Larry Wall, *1997*

In this work we developed a reliable parser generator based on combinators. We took steps in the direction of standardization of the generator input, enforcing the use of an actual standard, and by doing so, we avoid introducing yet another particular syntaxes. Our generated code is actually human readable, a desirable property that is hard to find when considering parser generators. The generated parser closely resembles the input grammar, with combinators corresponding to the set of operators defined by the ISO EBNF standard.

We also provided a small core of basic monadic combinators compatible with most of what is already established in the area of functional parsing. This basic set can be extended, and utilized for manual specification of parsers, without affecting the parser generator reliability.

Although we have failed to achieve an implementation of combinators capable of parsing the full set of CFGs, namely the left-recursive ones, this failure lead us to a different type of contribution, exposing the pitfalls of the implementation of Johnson's memoized combinators in a pure FL. This might be useful as a reference for future works considering a purely functional implementation of GLL, and to motivate a better understanding of a technique we believe, due to its simplicity, is ideal for the implementation of general combinators, preserving their code from modification and consequent complexity increase.

## 5.1 RELATED WORK

Parser combinators have a long ongoing research history. Even before the introduction of the GLL technique much has been done to improve their efficiency, extend the set of supported grammars, provide better error handling, and so on. In this section we focus on works close to the method and techniques we succeeded or failed to apply, mostly, recent works influenced by the introduction of GLL.

We start by pointing the reader to Section 3.3, where a lengthy discussion on the characteristics of other, GLL-based, implementations of parser combinators can be found. Those are recent works that surged since the introduction of GLL, which will not be discussed any further, but their nature have another aspect worth mentioning.

Most of the recent implementations of GLL combinators, achieve their goal of generality by adapting Johnson's memoization procedure (33). A recent work from the the

original GLL authors acknowledges the relation between both techniques (35). We mentioned this interesting relation between techniques separated by fifteen years of research in Section 3.3.

With that in mind, it seems strange that no implementation of memoized general combinators has been accomplished in Haskell. It is true that parser combinators that support left recursion do exist in Haskell (35, 36), but to the best of our knowledge, none that applies Johnson's technique. As an example of this claim, we take a look at Frost *et al.* memoized combinators, and the recent work of Binsbergen *et al.*

The work of Frost *et al.* (36) develops a set of combinators capable of parsing ambiguous and left-recursive grammars. The combinators do not incorporate continuations, and use memoization as a tool to reduce complexity alone. The memoization applied by Frost *et al.* requires the input of the combinators to be a numeric index of the input. The support to left recursion is achieved by the addition of extra context in the form of a counter. The number of calls to each parser at a certain input position is counted and if the number of calls goes out of bound their execution is interrupted. Direct and indirect recursion are treated differently, each solution adds complexity not only to types and memoization, but to the code of some of the combinators as well.

As a late discovery in the development of this work, we came upon the work of Binsbergen, Scott, and Johnstone, the last two the authors of the original work on GLL. The work of Binsbergen *et al.* is the first GLL implementation in Haskell, and it uses a completely different approach from other functional implementations.

The work of Binsbergen *et al.* is based on the principle that a grammar can be extracted from combinator expressions and then be given a stand alone parser (35). They define a parser procedure equipped with GLL machinery, similar to what is described in Section 2.3.3, to parse the generated grammars. A set of what is called "BNF combinators", which as reinforced by the authors are not the same as parser combinators, is provided as an "embedded DSL for describing syntax" (35). All the extra machinery used by this strategy is something we are trying to avoid, in order to preserve reliability and reduce the effort necessary for a possible verification.

In regards to functional parser generators. Most of the effort on improving parsing technology in functional programming seems to revolve around combinators alone; we say that from a Haskell perspective, but it is a reasonable assumption to extrapolate. There are a few Haskell parser generators such as Happy and Peggy. Both generators work with an embedded DSL approach, with different syntaxes; Peggy being based on Parsing Expression Grammars (PEGs). Happy and Peggy provide limited support to bigger sets of grammars via GLR and elimination of left-recursion, respectively (37, 38).

Parser generators like Happy and Peggy not only do not push for any sort of compatibility in their input format, they also produce code difficult to comprehend or argue about and that is specific to a certain PL, or a small set of languages. To support code

generation for more languages also adds complexity. Not to mention the greater complexity of their implementations, which makes it harder to establish any sort of reliability. Although the parsers generated by this work generator produces Haskell code, one can easily interface with the output of the generated parsers, since it is a simple to process data format, which can even be parametrized. See Section 5.2 for future work.

## 5.2 FUTURE WORK

With some improvements we believe this work can become a solid alternative for the current available parsing tools. Although we are not sure if the originally chosen technique for coping with left-recursion is actually appropriate for a purely functional implementation, other general techniques are available, and as long as they do not affect the combinators complexity in any harmful way, we intend to support the complete set of CFGs.

To provide better evidence of the generator reliability we have a couple of options, from a manual proof of some of the generator properties, to the application of a mechanized solution such as QuickCheck, which would also facilitate later verification of possible extensions.

Other branch of future work regards the consolidation of the generator as a production tool. Most of the parser generator limitations, discussed in Chapter 4, are solvable and we believe can be implemented without any major setbacks. We also want the generator to be a tool with a broader reach, by providing some parametrization of the parser output. One could for example configure the generator to synthesize parsers which would format their output as a JSON, which could be easily glued to projects, possibly, written in other programming languages.

# REFERENCES

1 D'SILVA, V.; KROENING, D.; WEISSENBACHER, G. A survey of automated techniques for formal software verification. *Transactions on Computer-Aided Design and Systems*, IEEE, v. 27, n. 7, p. 1165–1178, June 2008.

2 MALECHA, G.; RICKETTS, D.; ALVAREZ, M. M.; LERNER, S. Towards foundational verification of cyber-physical systems. In: *Science of Security for Cyber-Physical Systems Workshop*. Wien, AT: IEEE, 2016.

3 YANG, X.; CHEN, Y.; EIDE, E.; REGEHR, J. Finding and understanding bugs in C compilers. In: *Conference on Programming Language Design and Implementation*. New York, NY, US: ACM, 2011. p. 283–294.

4 BODIN, M.; CHARGUERAUD, A.; FILARETTI, D.; GARDNER, P.; MAFFEIS, S.; NAUDZIUNIENE, D.; SCHMITT, A.; SMITH, G. A trusted mechanised JavaScript specification. In: *Symposium on Principles of Programming Languages*. New York, NY, US: ACM, 2014. p. 87–100.

5 LEROY, X. Formal verification of a realistic compiler. *Commun. ACM*, ACM, New York, NY, US, v. 52, n. 7, p. 107–115, July 2009.

6 WANG, F.; SONG, F.; ZHANG, M.; ZHU, X.; ZHANG, J. Krust: A formal executable semantics of Rust. *CoRR*, abs/1804.10806, April 2018.

7 MAFFEIS, S.; MITCHELL, J. C.; TALY, A. An operational semantics for JavaScript. In: *Asian Symposium on Programming Languages and Systems*. Berlin, Heidelberg, DE: Springer, 2008. p. 307–325.

8 PARR, T.; FISHER, K. S. LL(*): The foundation of the ANTLR parser generator. In: *Conference on Programming Language Design and Implementation*. New York, NY, US: ACM, 2011. p. 425–436.

9 AFROOZEH, A.; IZMAYLOVA, A. One parser to rule them all. In: *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. New York, NY, US: ACM, 2015. p. 151–170.

10 AFROOZEH, A.; IZMAYLOVA, A. Iguana: A practical data-dependent parsing framework. In: *Proceedings of the 25th International Conference on Compiler Construction*. New York, NY, US: ACM, 2016. p. 267–268.

11 SCHRÖER, F. W. *ACCENT, A Compiler Compiler for the Entire Class of Context-Free Grammars*. 2006. On the internet: <http://accent.compilertools.net>. (Technical Report). Accessed: 08, jul. 2019.

12 LEUNG, A.; SARRACINO, J.; LERNER, S. Interactive parser synthesis by example. In: *Conference on Programming Language Design and Implementation*. New York, NY, US: ACM, 2015. p. 565–574.

13 SCOTT, E.; JOHNSTONE, A. GLL parsing. *Eletronic Notes in Theoretical Computer Science*, v. 253, n. 7, p. 177–189, July 2010.

14 HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. *Introduction to Automata Theory, Languages, and Computation*. 3. ed. Boston, MA, US: Addison-Wesley, 2007.

15 LINZ, P. *An Introduction to Formal Languages and Automata*. 5. ed. Sudbury, MA, US: Jones & Barlett, 2012.

16 AHO, A. V.; ULLMAN, J. D. *The Theory of Parsing, Translation, and Compiling*. 5. ed. Englewood Cliffs, NJ, US: Prentice-Hall, 1972. v. 1.

17 MOGENSEN, T. Æ. *Basics of Compiler Design*. Copenhagen, DK: Department of Computer Science, University of Copenhagen, 2009.

18 LOUDEN, K. C. *Compiler Construction Principles and Practice*. 1. ed. Boston, MA, US: Cengage Learning US, 1997.

19 TOMITA, M. *Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems*. 1. ed. Norwell, MA, US: Kluwer Academic Publishers, 1986.

20 FICSHER, C. N.; CYTRON, R. K.; LEBLANC, R. J. *Crafting a Compiler*. 1. ed. Boston, MA, US: Addison-Wesley, 2010.

21 SPIEWAK, D. *Generalized Parser Combinators*. 2010. On the internet: <https://dinhe.net/~aredridel/.notmine/PDFs/Parsing/>. Accessed: 12, aug. 2019.

22 ISO/IEC 14977:1996(E). Information Technology – Syntactic Metalanguage – Extended BNF. Genève, CH, 1996.

23 THOMPSON, S. *Haskell the craft of functional programming*. 3. ed. Harlow, Essex, GB: Pearson, 2011.

24 SABRY, A. What is a purely functional language? *Journal of Functional Programming*, Cambridge University Press, v. 8, n. 1, p. 1–22, January 1998.

25 HUDAK, P. *The Haskell School of Expression*. 1. ed. Cambridge, GB: Cambridge University Press, 2000.

26 BIRD, R. *Introduction to Functional Programming using Haskell*. 2. ed. Hertfordshire, GB: Prentice Hall, 1998.

27 HUTTON, G. *Programming in Haskell*. 1. ed. Cambridge, GB: Cambridge University Press, 2007.

28 FISCHER, S. *Reinventing Haskell Bactracking*. Christian-Albrechts Univerity of Kiel, 2009.

29 ØYE, V. *General Parser Combinators in Racket*. 2012. On the internet: <https://epsil.github.io/gll>. Accessed: 12, aug. 2019.

30 HUTTON, G. High-order function for parsing. *Journal of Functional Programming*, Cambridge University Press, v. 2, n. 3, p. 323–343, July 1992.

31 HUTTON, G.; MEIJER, E. *Monadic Parser Combinators*. School of Computer Science and IT, University of Nottingham, 1996.

32 LJUNGLÖF, P. *Pure Functional Parsing an Advanced Tutorial*. Licentiate Thesis — Departament of Computer Science, Chalmers University of Technology and Göttenborg University, 2002.

33 JOHNSON, M. Memoization in top-down parsing. *Association for Computational Linguistics*, Providence, RI, US, v. 21, n. 3, p. 405–417, September 1995.

34 AFROOZEH, A.; IZMAYLOVA, A.; STORM, T. van der. Practical, general parser combinators. In: *Workshop on Partial Evaluation and Program Manipulation*. New York, NY, US: ACM, 2016. p. 1–12.

35 BINSBERGEN, L. T. van; SCOTT, E.; JOHNSTONE, A. GLL parsing with flexible combinators. In: *Proceedings of ACM Conference*. New York, NY, US: ACM, 2018.

36 FROST, R. A.; HAFIZ, R.; CALLAGHAN, P. Parser combinators for ambiguous left-recursive grammars. In: *Practical Aspects of Declarative Languages*. Berlin, Heidelberg, DE: Springer, 2008. p. 167–181.

37 MARLOW, S.; GILL, A. *The Parser Generator for Haskell*. 2011. On the internet: <https://www.haskell.org/happy>. Accessed: 08, jul. 2019.

38 TANAKA, H. *The Parser Generator for Haskell*. 2011. On the internet: <https://tanakh.github.io/Peggy>. Accessed: 08, jul. 2019.

# APPENDIX A – JSON EBNF

```
json = element;

value = object
      | array
      | string
      | number
      | 'true'
      | 'false'
      | 'null';

object = '{', members, '}'
       | '{', ws, '}';

members = member, ',', members
        | member;

member = ws, string, ws, ':', element;

array = '[', elements, ']'
      | '[', ws, ']';

elements = element, ',', elements
         | element;

element = ws, value, ws;

string = '"', characters, '"';

characters = character, characters
           | eps;

character = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k'
          | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v'
          | 'w' | 'x' | 'y' | 'z'
          | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K'
          | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V'
          | 'W' | 'X' | 'Y' | 'Z'
          | digit
          | '\x0020'
          | '!' | '#' | '$' | '%' | '&' | "'" | '(' | ')' | '*' | '+' | ','
          | '-' | '.' | '/' | ':' | ';' | '<' | '=' | '>' | '?' | '@' | '['
          | ']' | '{' | '}' | '|' | '~'
          | '\x007f'
          | '\\', escape;
```

```
45
46  escape = '"' | '\\' | '/' | 'b' | 'f' | 'n' | 'r' | 't'
47         | 'u', hex, hex, hex;
48
49  hex = digit
50      | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
51      | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';
52
53  number = integer, fraction, exponent;
54
55  integer = '-', onenine, digits
56          | onenine, digits
57          | '-', digit
58          | digit;
59
60  digits = digit, digits
61         | digit;
62
63  digit = '0' | onenine;
64
65  onenine = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';
66
67  fraction = '.', digits | eps;
68
69  exponent = 'e', sign, digits
70           | 'E', sign, digits
71           | eps;
72
73  sign = '+' | '-' | eps;
74
75  ws = '\x0020', ws
76     | '\n', ws
77     | '\r', ws
78     | '\t', ws
79     | eps;
```

# APPENDIX B – GENERATED JSON PARSER

```
1  json = "json" =|> element
2
3  value = "value" =|>
4            object
5        <|> array
6        <|> string
7        <|> number
8        <|> t "true"
9        <|> t "false"
10       <|> t "null"
11
12 object = "object" =|>
13             t "{" # members # t "}"
14         <|> t "{" # ws # t "}"
15
16 members = "members" =|>
17               member # t "," # members
18           <|> member
19
20 member = "member" =|> ws # string # ws # t ":" # element
21
22 array = "array" =|>
23            t "[" # elements # t "]"
24       <|> t "[" # ws # t "]"
25
26 elements = "elements" =|>
27               element # t "," # elements
28           <|> element
29
30 element = "element" =|> ws # value # ws
31
32 string = "string" =|> t "\"" # characters # t "\""
33
34 characters = "characters" =|>
35                 character # characters
36             <|> eps
37
38 character = "character" =|>
39                t "a" <|> t "b" <|> t "c" <|> t "d" <|> t "e" <|> t "f"
40           <|> t "g" <|> t "h" <|> t "i" <|> t "j" <|> t "k" <|> t "l"
41           <|> t "m" <|> t "n" <|> t "o" <|> t "p" <|> t "q" <|> t "r"
42           <|> t "s" <|> t "t" <|> t "u" <|> t "v" <|> t "w" <|> t "x"
43           <|> t "y" <|> t "z"
44           <|> t "A" <|> t "B" <|> t "C" <|> t "D" <|> t "E" <|> t "F"
```

```
45          <|> t "G" <|> t "H" <|> t "I" <|> t "J" <|> t "K" <|> t "L"
46          <|> t "M" <|> t "N" <|> t "O" <|> t "P" <|> t "Q" <|> t "R"
47          <|> t "S" <|> t "T" <|> t "U" <|> t "V" <|> t "W" <|> t "X"
48          <|> t "Y" <|> t "Z"
49          <|> digit
50          <|> t "\x0020"
51          <|> t "!" <|> t "#" <|> t "$" <|> t "%" <|> t "&" <|> t "'"
52          <|> t "(" <|> t ")" <|> t "*" <|> t "+" <|> t "," <|> t "-"
53          <|> t "." <|> t "/" <|> t ":" <|> t ";" <|> t "<" <|> t "="
54          <|> t ">" <|> t "?" <|> t "@" <|> t "[" <|> t "]" <|> t "{"
55          <|> t "}" <|> t "|" <|> t "~"
56          <|> t "\x007f"
57          <|> t "\\" # escape
58
59  escape = "escape" =|>
60            t "\"" <|> t "\\" <|> t "/" <|> t "b"
61        <|> t "f" <|> t "n" <|> t "r" <|> t "t"
62        <|> t "u" # hex # hex # hex
63
64  hex = "hex" =|>
65         digit
66     <|> t "a" <|> t "b" <|> t "c" <|> t "d" <|> t "e" <|> t "f"
67     <|> t "A" <|> t "B" <|> t "C" <|> t "D" <|> t "E" <|> t "F"
68
69  number = "number" =|> integer # fraction # exponent
70
71  integer = "integer" =|>
72             t "-" # onenine # digits
73         <|> onenine # digits
74         <|> t "-" # digit
75         <|> digit
76
77  digits = "digits" =|>
78            digit # digits
79        <|> digit
80
81  digit = "digit" =|> t "0" <|> onenine
82
83  onenine = "onenine" =|>
84             t "1" <|> t "2" <|> t "3" <|> t "4" <|> t "5"
85         <|> t "6" <|> t "7" <|> t "8" <|> t "9"
86
87  fraction = "fraction" =|>
88              t "." # digits
89          <|> eps
90
91  exponent = "exponent" =|>
```

```
92              t "e" # sign # digits
93          <|> t "E" # sign # digits
94          <|> eps
95
96  sign = "sign" =|> t "+" <|> t "-" <|> eps
97
98  ws = "ws" =|>
99        t "\x0020" # ws
100   <|> t "\n" # ws
101   <|> t "\r" # ws
102   <|> t "\t" # ws
103   <|> eps
```

# APPENDIX C – JAVA 1.7 SYNTACTICAL EBNF

```
qualifiedIdentifier     = identifier, {'.', identifier};
qualifiedIdentifierList = qualifiedIdentifier, {',', qualifiedIdentifier};


compilationUnit = [[anotations], 'package', qualifiedIdentifier, ';']
                , {importDeclaration}, {typeDeclaration};

importDeclaration = 'import', ['static'], qualifiedIdentifier, ['.*'];

typeDeclaration = classOrInterfaceDeclaration | ';';

classOrInterfaceDeclaration =
    {modifier}, (classDeclaration | interfaceDeclaration);

classDeclaration = normalClassDeclaration | enumDeclaration;

interfaceDeclaration = normalInterfaceDeclaration | annotationTypeDeclaration;


normalClassDeclaration = 'class', identifier, [typeParameters]
                       , ['extends', ttype], ['implements', typeList]
                       , classBody;

enumDeclaration = 'enum', identifier, ['implements', typeList], enumBody;

normalInterfaceDeclaration = 'interface', identifier, [typeParameters]
                           , ['extends', typeList], interfaceBody;

annotationTypeDeclaration = '@', 'interface', identifier, annotationTypeBody;


ttype = (basicType | referenceType), {'[', ']'};

basicType = 'byte'
          | 'short'
          | 'char'
          | 'int'
          | 'long'
          | 'float'
          | 'double'
          | 'boolean';

referenceType =
    identifier, [typeArguments], {'.', identifier, [typeArguments]};
```

```
45
46  typeArguments = '<', typeArgument, {',', typeArgument }, '>';
47  typeArgument  = referenceType
48                | '?', [('extends' | 'super'), referenceType];
49
50
51  nonWildcardTypeArguments = '<', typeList, '>';
52
53  typeList = referenceType, {',', referenceType};
54
55  typeArgumentsOrDiamond = typeArguments | '<', '>';
56
57  nonWildcardTypeArgumentsOrDiamond = nonWildcardTypeArguments | '<', '>';
58
59  typeParameters = '<', typeParameter, {',', typeParameter}, '>';
60  typeParameter  = identifier, ['extends', bound];
61
62  bound = referenceType, {'&', referenceType};
63
64
65  modifier = annotation
66           | 'public'
67           | 'protected'
68           | 'private'
69           | 'static'
70           | 'abstract'
71           | 'final'
72           | 'native'
73           | 'synchronized'
74           | 'transient'
75           | 'volatile'
76           | 'strictfp';
77
78  annotations = annotation, {annotation};
79  annotation  = '@', qualifiedIdentifier, ['(', [annotationElement], ')'];
80
81  annotationElement = elementValuePairs | elementValue;
82
83  elementValuePairs = elementValuePair, {',', elementValuePair};
84  elementValuePair  = identifier, '=', elementValue;
85  elementValue      = annotation
86                    | expression1
87                    | elementValueArrayInitializer;
88
89  elementValueArrayInitializer = '{', [elementValues], [','], '}';
90
91  elementValues = elementValue, {',', elementValue};
```

```
92
93
94  classBody = '{', {classBodyDeclaration}, '}';
95
96  classBodyDeclaration = ['static'], block
97                       | {modifier}, memberDecl
98                       | ';';
99
100 memberDecl = methodOrFieldDecl
101           | 'void', identifier, voidMethodDeclaratorRest
102           | identifier, constructorDeclaratorRest
103           | genericMethodOrConstructorDecl
104           | classDeclaration
105           | interfaceDeclaration;
106
107 methodOrFieldDecl = ttype, identifier, methodOrFieldRest;
108 methodOrFieldRest = fieldDeclaratorsRest, ';'
109                   | methodDeclaratorRest;
110
111 fieldDeclaratorsRest = variableDeclaratorRest, {',', variableDeclarator};
112
113 methodDeclaratorRest = formalParameters, {'[', ']'}
114                      , ['throws', qualifiedIdentifierList], (block | ';');
115 voidMethodDeclaratorRest =
116     formalParameters, ['throws', qualifiedIdentifierList], (block | ';');
117
118 constructorDeclaratorRest =
119     formalParameters, ['throws', qualifiedIdentifierList], block;
120
121 genericMethodOrConstructorDecl =
122     typeParameters, genericMethodOrConstructorRest;
123 genericMethodOrConstructorRest =
124       (ttype | 'void'), identifier, methodDeclaratorRest
125     | identifier, constructorDeclaratorRest;
126
127
128 interfaceBody = '{', {interfaceBodyDeclaration}, '}';
129
130 interfaceBodyDeclaration = {modifier}, interfaceMemberDecl
131                          | ';';
132
133 interfaceMemberDecl = interfaceMethodOrFieldDecl
134                     | 'void', identifier, voidInterfaceMethodDeclaratorRest
135                     | interfaceGenericMethodDecl
136                     | classDeclaration
137                     | interfaceDeclaration;
138
```

```
interfaceMethodOrFieldDecl = ttype, identifier, interfaceMethodOrFieldRest;
interfaceMethodOrFieldRest = constantDeclaratorsRest, ';'
                           | interfaceMethodDeclaratorRest;

constantDeclaratorsRest = constantDeclaratorRest, {',', constantDeclarator};
constantDeclaratorRest  = {'[', ']'}, '=', variableInitializer;

constantDeclarator = identifier, constantDeclaratorRest;

interfaceMethodDeclaratorRest =
    formalParameters, {'[', ']'}, ['throws', qualifiedIdentifierList];

voidInterfaceMethodDeclaratorRest =
    formalParameters, ['throws', qualifiedIdentifierList];

interfaceGenericMethodDecl =
    typeParameters, (ttype | 'void'), identifier, interfaceMethodDeclaratorRest;


formalParameters = '(', [formalParameterDecls], ')';

formalParameterDecls = {variableModifier}, ttype, formalParameterDeclsRest;

variableModifier = 'final' | annotation;

formalParameterDeclsRest = variableDeclaratorId, [',', formalParameterDecls]
                         | '...', variableDeclaratorId;


variableDeclaratorId = identifier, {'[', ']'};


variableDeclarators = variableDeclarator, {',', variableDeclarator};
variableDeclarator  = identifier, variableDeclaratorRest;

variableDeclaratorRest = {'[', ']'}, ['=', variableInitializer];

variableInitializer = arrayInitializer | expression;

arrayInitializer =
    '{', [variableInitializer, {',', variableInitializer}, [',']], '}';


block = '{', {blockStatement}, '}';

blockStatement  = localVariableDeclarationStatement
                | classOrInterfaceDeclaration
```

```
186                 | [identifier, ':'], statement;
187
188 localVariableDeclarationStatement = localVariableDeclaration, ';';
189
190 localVariableDeclaration = {variableModifier}, ttype, variableDeclarators;
191
192 statement = block
193           | ';'
194           | identifier, ':', statement
195           | statementExpression, ';'
196           | 'if', parExpression, statement, ['else', statement]
197           | 'assert', expression, [':', expression], ';'
198           | 'switch', parExpression, '{', switchBlockStatementGroups, '}'
199           | 'while', parExpression, statement
200           | 'do', statement, 'while', parExpression, ';'
201           | for, '(', forControl, ')', statement
202           | 'break', [identifier], ';'
203           | 'continue', [identifier], ';'
204           | 'return', [expression], ';'
205           | 'throw', expression, ';'
206           | 'synchronized', parExpression, block
207           | 'try', block, (catches | [catches], finally)
208           | 'try', resourceSpecification, block, [catches], [finally];
209
210 statementExpression = expression;
211
212
213 catches = catchClause, {catchClause};
214
215 catchClause =
216     'catch', '(', {variableModifier}, catchType, identifier, ')', block;
217
218 catchType = qualifiedIdentifier, {'|', qualifiedIdentifier};
219
220 finally = 'finally', block;
221
222 resourceSpecification = '(', resources, [';'], ')';
223
224 resources = resource, {';', resource};
225 resource  =
226     {variableModifier}, referenceType, variableDeclaratorId, '=', expression;
227
228
229 switchBlockStatementGroups = {switchBlockStatementGroup};
230 switchBlockStatementGroup  = switchLabels, blockStatements;
231
232 switchLabels = switchLabel, {switchLabel};
```

```
switchLabel  = 'case', (expression | enumConstantName), ':'
             | 'default', ':';

enumConstantName = identifier;


forControl = forVarControl
           | forInit, ';', [expression], ';', [forUpdate];

forVarControl =
    {variableModifier}, ttype, variableDeclaratorId, forVarControlRest;

forVarControlRest =
      forVariableDeclaratorsRest, ';', [expression], ';', [forUpdate]
    | ':', expression;

forVariableDeclaratorsRest =
    ['=', variableInitializer], {',', variableDeclarator};

forInit   = forUpdate;
forUpdate = statementExpressions;

statementExpressions = statementExpression, {',', statementExpression};


expression = expression1, [assignmentOperator, expression1];

assignmentOperator = '='
                   | '+=' | '-=' | '*=' | '\='
                   | '&=' | '|='
                   | '^=' | '%='
                   | '<<=' | '>>=' | '>>>=';

expression1 = expression2, [expression1Rest];

expression1Rest = '?', expression, ':', expression1;

expression2 = expression3, [expression2Rest];

expression2Rest = {infixOp, expression3}
                | 'instanceof', ttype;


infixOp = '||' | '&&' | '|' | '&'
        | '==' | '!='
        | '<' | '>' | '<=' | '>='
        | '<<' | '>>' | '>>>'
```

```
280         | '+' | '-' | '*' | '/' | '%' | '^';
281
282 expression3 = prefixOp, expression3
283             | '(', (expression | ttype), ')', expression3
284             | primary, {selector}, {postfixOp};
285
286 prefixOp = '++' | '--'
287          | '!' | '~'
288          | '+' | '-';
289
290 postfixOp = '++' | '--';
291
292
293 primary = literal
294         | parExpression
295         | 'this', [arguments]
296         | 'super', superSiffix
297         | 'new', creator
298         | nonWildcardTypeArguments
299         , (explicitGenericInvocationSuffix | 'this', arguments)
300         | qualifiedIdentifier, [identifierSuffix]
301         | basicType, {'[', ']'}, '.', 'class'
302         | 'void', '.', 'class';
303
304
305 parExpression = '(', expression, ')';
306
307 arguments = '(', [expression, {',', expression}], ')';
308
309 superSiffix = arguments
310             | '.', identifier, [arguments];
311
312 explicitGenericInvocationSuffix = 'super', superSiffix
313                                 | identifier, arguments;
314
315
316 creator = nonWildcardTypeArguments, createdName, classCreatorRest
317         | createdName, (classCreatorRest | arrayCreatorRest);
318
319 createdName = identifier, [typeArgumentsOrDiamond]
320             , {'.', identifier, [typeArgumentsOrDiamond]};
321
322 classCreatorRest = arguments, [classBody];
323
324 arrayCreatorRest =
325     '[', ( ']', {'[', ']'}, arrayInitializer
326          | expression, ']', {'[', expression, ']'}, {'[', ']'} );
```

```
327
328
329 identifierSuffix = '[', ({'[', ']'}, '.', 'class' | expression), ']'
330                   | arguments
331                   | '.', ( 'class'
332                          | explicitGenericInvocation
333                          | 'this'
334                          | 'super', arguments
335                          | 'new', [nonWildcardTypeArguments], innerCreator )
336                   ;
337
338 explicitGenericInvocation =
339     nonWildcardTypeArguments, explicitGenericInvocationSuffix;
340
341 innerCreator =
342     identifier, [nonWildcardTypeArgumentsOrDiamond], classCreatorRest;
343
344
345 selector = '.', ( identifier
346                 | explicitGenericInvocation
347                 | 'this'
348                 | 'super', superSiffix
349                 | 'new', [nonWildcardTypeArguments], innerCreator )
350          | '[', expression, ']';
351
352
353 enumBody = '{', [enumConstants], [','], [enumBodyDeclarations], '}';
354
355 enumConstants = enumConstant, {',', enumConstant};
356 enumConstant  = [annotations], identifier, [arguments], [classBody];
357
358 enumBodyDeclarations = ';', {classBodyDeclaration};
359
360 annotationTypeBody = '{', {annotationTypeElementDeclaration}, '}';
361
362 annotationTypeElementDeclaration = {modifier}, annotationTypeElementRest;
363
364 annotationTypeElementRest = ttype, identifier, annotationMethodOrConstantRest
365                           | classDeclaration
366                           | interfaceDeclaration
367                           | enumDeclaration
368                           | annotationTypeDeclaration;
369
370 annotationMethodOrConstantRest = annotationMethodRest
371                                | constantDeclaratorsRest;
372
373 annotationMethodRest = '(', ')', ['[', ']'], ['default', elementValue];
```

# APPENDIX D – GENERATED JAVA 1.7 PARSER

```
qualifiedIdentifier = "qualifiedIdentifier" =|>
    identifier # closure (t "." # identifier)
qualifiedIdentifierList = "qualifiedIdentifierList" =|>
    qualifiedIdentifier # closure (t "," # qualifiedIdentifier)


compilationUnit = "compilationUnit" =|>
      opt (opt (anotations) # t "package" # qualifiedIdentifier # t ";")
    # closure (importDeclaration) # closure (typeDeclaration)

importDeclaration = "importDeclaration" =|>
    t "import" # opt (t "static") # qualifiedIdentifier # opt (t ".*")

typeDeclaration = "typeDeclaration" =|> classOrInterfaceDeclaration <|> t ";"

classOrInterfaceDeclaration = "classOrInterfaceDeclaration" =|>
    closure (modifier) # (classDeclaration <|> interfaceDeclaration)

classDeclaration = "classDeclaration" =|>
    normalClassDeclaration <|> enumDeclaration

interfaceDeclaration = "interfaceDeclaration" =|>
    normalInterfaceDeclaration <|> annotationTypeDeclaration

normalClassDeclaration = "normalClassDeclaration" =|>
      t "class" # identifier # opt (typeParameters) # opt (t "extends" # ttype)
    # opt (t "implements" # typeList) # classBody

enumDeclaration = "enumDeclaration" =|>
    t "enum" # identifier # opt (t "implements" # typeList) # enumBody
normalInterfaceDeclaration = "normalInterfaceDeclaration" =|>
      t "interface" # identifier # opt (typeParameters)
    # opt (t "extends" # typeList) # interfaceBody

annotationTypeDeclaration = "annotationTypeDeclaration" =|>
    t "@" # t "interface" # identifier # annotationTypeBody


ttype = "ttype" =|> (basicType <|> referenceType) # closure (t "[" # t "]")

basicType = "basicType" =|>
             t "byte"
         <|> t "short"
         <|> t "char"
```

```
45          <|> t "int"
46          <|> t "long"
47          <|> t "float"
48          <|> t "double"
49          <|> t "boolean"
50
51  referenceType = "referenceType" =|>
52      identifier # opt (typeArguments) # closure (t "." # identifier
53                                                  # opt (typeArguments))
54
55  typeArguments = "typeArguments" =|>
56      t "<" # typeArgument # closure (t "," # typeArgument) # t ">"
57  typeArgument  = "typeArgument"  =|>
58                  referenceType
59              <|> t "?" # opt ((t "extends" <|> t "super") # referenceType)
60
61  nonWildcardTypeArguments = "nonWildcardTypeArguments" =|>
62      t "<" # typeList # t ">"
63
64  typeList = "typeList" =|> referenceType # closure (t "," # referenceType)
65
66  typeArgumentsOrDiamond = "typeArgumentsOrDiamond" =|>
67      typeArguments <|> t "<" # t ">"
68
69  nonWildcardTypeArgumentsOrDiamond = "nonWildcardTypeArgumentsOrDiamond" =|>
70      nonWildcardTypeArguments <|> t "<" # t ">"
71
72  typeParameters = "typeParameters" =|>
73      t "<" # typeParameter # closure (t "," # typeParameter) # t ">"
74  typeParameter  = "typeParameter"  =|> identifier # opt (t "extends" # bound)
75
76  bound = "bound" =|> referenceType # closure (t "&" # referenceType)
77
78
79  modifier = "modifier" =|>
80              annotation
81          <|> t "public"
82          <|> t "protected"
83          <|> t "private"
84          <|> t "static"
85          <|> t "abstract"
86          <|> t "final"
87          <|> t "native"
88          <|> t "synchronized"
89          <|> t "transient"
90          <|> t "volatile"
91          <|> t "strictfp"
```

```
92
93  annotations = "annotations" =|> annotation # closure (annotation)
94  annotation  = "annotation"  =|>
95      t "@" # qualifiedIdentifier # opt (t "(" # opt (annotationElement) # t ")")
96
97  annotationElement = "annotationElement" =|> elementValuePairs <|> elementValue
98
99  elementValuePairs = "elementValuePairs" =|>
100     elementValuePair # closure (t "," # elementValuePair)
101 elementValuePair  = "elementValuePair"  =|> identifier # t "=" # elementValue
102
103 elementValue = "elementValue" =|>
104                 annotation
105             <|> expression1
106             <|> elementValueArrayInitializer
107
108 elementValueArrayInitializer = "elementValueArrayInitializer" =|>
109     t "{" # opt (elementValues) # opt (t ",") # t "}"
110
111 elementValues = "elementValues" =|>
112     elementValue # closure (t "," # elementValue)
113
114
115 classBody = "classBody" =|> t "{" # closure (classBodyDeclaration) # t "}"
116
117 classBodyDeclaration = "classBodyDeclaration" =|>
118                         opt (t "static") # block
119                     <|> closure (modifier) # memberDecl
120                     <|> t ";"
121
122 memberDecl = "memberDecl" =|>
123                methodOrFieldDecl
124           <|> t "void" # identifier # voidMethodDeclaratorRest
125           <|> identifier # constructorDeclaratorRest
126           <|> genericMethodOrConstructorDecl
127           <|> classDeclaration
128           <|> interfaceDeclaration
129
130 methodOrFieldDecl = "methodOrFieldDecl" =|>
131     ttype # identifier # methodOrFieldRest
132 methodOrFieldRest = "methodOrFieldRest" =|>
133                      fieldDeclaratorsRest # t ";"
134                 <|> methodDeclaratorRest
135
136 fieldDeclaratorsRest = "fieldDeclaratorsRest" =|>
137     variableDeclaratorRest # closure (t "," # variableDeclarator)
138
```

```
139 methodDeclaratorRest = "methodDeclaratorRest" =|>
140       formalParameters # closure (t "[" # t "]")
141     # opt (t "throws" # qualifiedIdentifierList) # (block <|> t ";")
142
143 voidMethodDeclaratorRest = "voidMethodDeclaratorRest" =|>
144       formalParameters # opt (t "throws" # qualifiedIdentifierList)
145     # (block <|> t ";")
146
147 constructorDeclaratorRest = "constructorDeclaratorRest" =|>
148     formalParameters # opt (t "throws" # qualifiedIdentifierList) # block
149
150 genericMethodOrConstructorDecl = "genericMethodOrConstructorDecl" =|>
151     typeParameters # genericMethodOrConstructorRest
152 genericMethodOrConstructorRest = "genericMethodOrConstructorRest" =|>
153         (ttype <|> t "void") # identifier # methodDeclaratorRest
154     <|> identifier # constructorDeclaratorRest
155
156
157 interfaceBody = "interfaceBody" =|>
158     t "{" # closure (interfaceBodyDeclaration) # t "}"
159
160 interfaceBodyDeclaration = "interfaceBodyDeclaration" =|>
161                             closure (modifier) # interfaceMemberDecl
162                         <|> t ";"
163
164 interfaceMemberDecl = "interfaceMemberDecl" =|>
165         interfaceMethodOrFieldDecl
166     <|> t "void" # identifier # voidInterfaceMethodDeclaratorRest
167     <|> interfaceGenericMethodDecl
168     <|> classDeclaration <|> interfaceDeclaration
169
170 interfaceMethodOrFieldDecl = "interfaceMethodOrFieldDecl" =|>
171     ttype # identifier # interfaceMethodOrFieldRest
172 interfaceMethodOrFieldRest = "interfaceMethodOrFieldRest" =|>
173         constantDeclaratorsRest # t ";"
174     <|> interfaceMethodDeclaratorRest
175
176 constantDeclaratorsRest = "constantDeclaratorsRest" =|>
177     constantDeclaratorRest # closure (t "," # constantDeclarator)
178 constantDeclaratorRest  = "constantDeclaratorRest" =|>
179     closure (t "[" # t "]") # t "=" # variableInitializer
180
181 constantDeclarator = "constantDeclarator" =|>
182     identifier # constantDeclaratorRest
183
184 interfaceMethodDeclaratorRest = "interfaceMethodDeclaratorRest" =|>
185       formalParameters # closure (t "[" # t "]")
```

```
186     # opt (t "throws" # qualifiedIdentifierList)
187
188 voidInterfaceMethodDeclaratorRest = "voidInterfaceMethodDeclaratorRest" =|>
189     formalParameters # opt (t "throws" # qualifiedIdentifierList)
190 interfaceGenericMethodDecl = "interfaceGenericMethodDecl" =|>
191       typeParameters # (ttype <|> t "void") # identifier
192     # interfaceMethodDeclaratorRest
193
194
195 formalParameters = "formalParameters" =|>
196     t "(" # opt (formalParameterDecls) # t ")"
197
198 formalParameterDecls = "formalParameterDecls" =|>
199     closure (variableModifier) # ttype # formalParameterDeclsRest
200
201 variableModifier = "variableModifier" =|> t "final" <|> annotation
202
203 formalParameterDeclsRest = "formalParameterDeclsRest" =|>
204         variableDeclaratorId # opt (t "," # formalParameterDecls)
205     <|> t "..." # variableDeclaratorId
206
207
208 variableDeclaratorId = "variableDeclaratorId" =|>
209     identifier # closure (t "[" # t "]")
210
211
212 variableDeclarators = "variableDeclarators" =|>
213     variableDeclarator # closure (t "," # variableDeclarator)
214 variableDeclarator  = "variableDeclarator"  =|>
215     identifier # variableDeclaratorRest
216
217 variableDeclaratorRest = "variableDeclaratorRest" =|>
218     closure (t "[" # t "]") # opt (t "=" # variableInitializer)
219
220 variableInitializer = "variableInitializer" =|> arrayInitializer <|> expression
221
222 arrayInitializer = "arrayInitializer" =|>
223       t "{" # opt (variableInitializer # closure (t "," # variableInitializer)
224     # opt (t ",")) # t "}"
225
226
227 block = "block" =|> t "{" # closure (blockStatement) # t "}"
228
229 blockStatement = "blockStatement" =|>
230                    localVariableDeclarationStatement
231               <|> classOrInterfaceDeclaration
232               <|> opt (identifier # t ":") # statement
```

```
233
234 localVariableDeclarationStatement = "localVariableDeclarationStatement" =|>
235     localVariableDeclaration # t ";"
236
237 localVariableDeclaration = "localVariableDeclaration" =|>
238     closure (variableModifier) # ttype # variableDeclarators
239
240 statement = "statement" =|>
241         block
242     <|> t ";"
243     <|> identifier # t ":" # statement
244     <|> statementExpression # t ";"
245     <|> t "if" # parExpression # statement # opt (t "else" # statement)
246     <|> t "assert" # expression # opt (t ":" # expression) # t ";"
247     <|> t "switch" # parExpression # t "{" # switchBlockStatementGroups # t "}"
248     <|> t "while" # parExpression # statement
249     <|> t "do" # statement # t "while" # parExpression # t ";"
250     <|> for # t "(" # forControl # t ")" # statement
251     <|> t "break" # opt (identifier) # t ";"
252     <|> t "continue" # opt (identifier) # t ";"
253     <|> t "return" # opt (expression) # t ";"
254     <|> t "throw" # expression # t ";"
255     <|> t "synchronized" # parExpression # block
256     <|> t "try" # block # (catches <|> opt (catches) # finally)
257     <|> t "try" # resourceSpecification # block # opt (catches) # opt (finally)
258
259 statementExpression = "statementExpression" =|> expression
260
261
262 catches = "catches" =|> catchClause # closure (catchClause)
263 catchClause = "catchClause" =|>
264       t "catch" # t "(" # closure (variableModifier) # catchType # identifier
265                                                                 # t ")"
266     # block
267
268 catchType = "catchType" =|>
269     qualifiedIdentifier # closure (t "|" # qualifiedIdentifier)
270
271 finally = "finally" =|> t "finally" # block
272
273 resourceSpecification = "resourceSpecification" =|>
274     t "(" # resources # opt (t ";") # t ")"
275
276 resources = "resources" =|> resource # closure (t ";" # resource)
277 resource  = "resource"  =|>
278       closure (variableModifier) # referenceType # variableDeclaratorId
279     # t "=" # expression
```

```
280
281 switchBlockStatementGroups = "switchBlockStatementGroups" =|>
282     closure (switchBlockStatementGroup)
283 switchBlockStatementGroup  = "switchBlockStatementGroup"  =|>
284     switchLabels # blockStatements
285
286 switchLabels = "switchLabels" =|> switchLabel # closure (switchLabel)
287 switchLabel  = "switchLabel"  =|>
288                t "case" # (expression <|> enumConstantName) # t ":"
289            <|> t "default" # t ":"
290
291 enumConstantName = "enumConstantName" =|> identifier
292
293
294 forControl = "forControl" =|>
295              forVarControl
296          <|> forInit # t ";" # opt (expression) # t ";" # opt (forUpdate)
297
298 forVarControl = "forVarControl" =|>
299       closure (variableModifier) # ttype # variableDeclaratorId
300     # forVarControlRest
301
302 forVarControlRest = "forVarControlRest" =|>
303         forVariableDeclaratorsRest # t ";" # opt (expression) # t ";"
304                                                              # opt (forUpdate)
305     <|> t ":" # expression
306
307 forVariableDeclaratorsRest = "forVariableDeclaratorsRest" =|>
308     opt (t "=" # variableInitializer) # closure (t "," # variableDeclarator)
309
310 forInit = "forInit" =|> forUpdate
311 forUpdate = "forUpdate" =|> statementExpressions
312
313 statementExpressions = "statementExpressions" =|>
314     statementExpression # closure (t "," # statementExpression)
315
316
317 expression = "expression" =|>
318     expression1 # opt (assignmentOperator # expression1)
319
320 assignmentOperator = "assignmentOperator" =|>
321                      t "=" <|> t "+=" <|> t "-=" <|> t "*=" <|> t "\="
322                  <|> t "&=" <|> t "|="
323                  <|> t "^=" <|> t "%="
324                  <|> t "<<=" <|> t ">>=" <|> t ">>>="
325
326 expression1 = "expression1" =|> expression2 # opt (expression1Rest)
```

```
327
328 expression1Rest = "expression1Rest" =|>
329     t "?" # expression # t ":" # expression1
330
331 expression2 = "expression2" =|> expression3 # opt (expression2Rest)
332
333 expression2Rest = "expression2Rest" =|>
334                     closure (infixOp # expression3)
335                <|> t "instanceof" # ttype
336
337
338 infixOp = "infixOp" =|>
339            t "||" <|> t "&&" <|> t "|" <|> t "&"
340            <|> t "==" <|> t "!="
341            <|> t "<" <|> t ">" <|> t "<=" <|> t ">="
342            <|> t "<<" <|> t ">>" <|> t ">>>"
343            <|> t "+" <|> t "-" <|> t "*" <|> t "/" <|> t "%" <|> t "^"
344
345 expression3 = "expression3" =|>
346                prefixOp # expression3
347            <|> t "(" # (expression <|> ttype) # t ")" # expression3
348            <|> primary # closure (selector) # closure (postfixOp)
349
350 prefixOp = "prefixOp" =|>
351             t "++" <|> t "--"
352         <|> t "!" <|> t "~"
353         <|> t "+" <|> t "-"
354
355 postfixOp = "postfixOp" =|> t "++" <|> t "--"
356
357
358 primary = "primary" =|>
359            literal
360        <|> parExpression
361        <|> t "this" # opt (arguments)
362        <|> t "super" # superSiffix
363        <|> t "new" # creator
364        <|> nonWildcardTypeArguments
365         #  (explicitGenericInvocationSuffix <|> t "this" # arguments)
366        <|> qualifiedIdentifier # opt (identifierSuffix)
367        <|> basicType # closure (t "[" # t "]") # t "." # t "class"
368        <|> t "void" # t "." # t "class"
369
370
371 parExpression = "parExpression" =|> t "(" # expression # t ")"
372
373 arguments = "arguments" =|>
```

```
374     t "(" # opt (expression # closure (t "," # expression)) # t ")"
375
376 superSiffix = "superSiffix" =|>
377                arguments
378            <|> t "." # identifier # opt (arguments)
379
380 explicitGenericInvocationSuffix = "explicitGenericInvocationSuffix" =|>
381                                   t "super" # superSiffix
382                                <|> identifier # arguments
383
384
385 creator = "creator" =|>
386            nonWildcardTypeArguments # createdName # classCreatorRest
387        <|> createdName # (classCreatorRest <|> arrayCreatorRest)
388
389 createdName = "createdName" =|>
390       identifier # opt (typeArgumentsOrDiamond)
391     # closure (t "." # identifier # opt (typeArgumentsOrDiamond))
392
393 classCreatorRest = "classCreatorRest" =|> arguments # opt (classBody)
394
395 arrayCreatorRest = "arrayCreatorRest" =|>
396      t "[" # (  t "]" # closure (t "[" # t "]") # arrayInitializer
397             <|> expression # t "]"
398    # closure (t "[" # expression # t "]") # closure (t "[" # t "]"))
399
400 identifierSuffix = "identifierSuffix" =|>
401         t "[" # (closure (t "[" # t "]") # t "." # t "class" <|> expression)
402                 # t "]"
403     <|> arguments
404     <|> t "." # (  t "class"
405                 <|> explicitGenericInvocation
406                 <|> t "this"
407                 <|> t "super" # arguments
408                 <|> t "new" # opt (nonWildcardTypeArguments) # innerCreator)
409
410 explicitGenericInvocation = "explicitGenericInvocation" =|>
411     nonWildcardTypeArguments # explicitGenericInvocationSuffix
412
413 innerCreator = "innerCreator" =|>
414     identifier # opt (nonWildcardTypeArgumentsOrDiamond) # classCreatorRest
415
416 selector = "selector" =|>
417             t "." # (  identifier
418                     <|> explicitGenericInvocation
419                     <|> t "this"
420                     <|> t "super" # superSiffix
```

```
421                     <|> t "new" # opt (nonWildcardTypeArguments) # innerCreator)
422           <|> t "[" # expression # t "]"
423
424
425 enumBody = "enumBody" =|>
426       t "{" # opt (enumConstants) # opt (t ",") # opt (enumBodyDeclarations)
427     # t "}"
428
429 enumConstants = "enumConstants" =|>
430     enumConstant # closure (t "," # enumConstant)
431 enumConstant  = "enumConstant"  =|>
432     opt (annotations) # identifier # opt (arguments) # opt (classBody)
433
434 enumBodyDeclarations = "enumBodyDeclarations" =|>
435     t ";" # closure (classBodyDeclaration)
436
437 annotationTypeBody = "annotationTypeBody" =|>
438     t "{" # closure (annotationTypeElementDeclaration) # t "}"
439
440 annotationTypeElementDeclaration = "annotationTypeElementDeclaration" =|>
441     closure (modifier) # annotationTypeElementRest
442
443 annotationTypeElementRest = "annotationTypeElementRest" =|>
444         ttype # identifier # annotationMethodOrConstantRest
445     <|> classDeclaration
446     <|> interfaceDeclaration
447     <|> enumDeclaration
448     <|> annotationTypeDeclaration
449
450 annotationMethodOrConstantRest = "annotationMethodOrConstantRest" =|>
451                                      annotationMethodRest
452                                  <|> constantDeclaratorsRest
453
454 annotationMethodRest = "annotationMethodRest" =|>
455     t "(" # t ")" # opt (t "[" # t "]") # opt (t "default" # elementValue)
```