Expertise
and insight
for the future

Phuc Nguyen

An introduction to parsing context-free language

Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
Bachelor's Thesis
24 April 2020

Author Title Number of Pages Date	Phuc Nguyen An introduction to parsing context-free language 31 pages + 0 appendices 24 April 2020
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Smart System
Instructors	Kimmo Sauren

Context-free grammar is a fundamental tool in software technologies. It is present in almost all aspects of software developments, from construction of software in programming languages and compilers to applications such as data transmission, storage and presentation. This thesis aims to introduce context-free grammar and parsing context-free language to beginners by answering three questions – what context-free grammars are, what parsing context-free language means and how to parse context-free languages.

The motivation for context-free grammar is shown through its ordinary usages in daily signs, forms and numerical formulae. Having established that context-free grammar is used to describe structured information, its ability to do so is showcased further through examples in numerical expressions and programming languages.

Concrete examples and detailed illustrations are provided to show what parsing context-free language means and how to do so. To complete the introduction to parsing context-free language, an example parsing algorithm is provided. The algorithm is constructed on the idea of simulating left-most reduction on an input string and its implementation is based on a procedure called the DK-test, which is meant for determining if a context-free grammar is deterministic. This example parsing algorithm is rudimentary and limited in power, but it incorporates ideas adaptable to more advanced parsing algorithms, making it an ideal starting point for the study of parsing context-free languages.

Keywords	parsing, context-free language, context-free grammar, introduction to parsing

Contents

1 Introduction 1

2 Motivation for context-free grammar 2

3 Expressiveness of context-free grammar 8

3.1 Numerical expression 8

3.2 Functional expression 13

4 Context-free grammar 16

4.1 Definition of context-free grammar 16

4.2 Derivation and reduction 16

5 Parsing context-free language by simulating left-most reduction 20

5.1 Illustration of the parsing algorithm 21

5.2 The parsing algorithm 25

6 Conclusion 31

References 32

1 Introduction

This thesis aims at providing a full fundamental overview on how to parse context-free language, from motivation to minimal example implementation. Context-free language and context-free grammar may have faded into the background of modern software technologies, but they are the backbone of software development today. Almost all modern programming languages at the time of writing are described using context-free grammar either entirely or only for the context-free part of the language.

One of the most widely used programming languages – the programming language C – has its syntax described by a context-free grammar. Fundamental questions encountered when reading a piece of programming code in C, such as what is a number, string or whitespace. Which concepts are high-level compared to raw bits or plain text, can be answered by the syntax of the language [1], providing a well-defined and intuitive structure to interpret the language.

The same method is applied to describe other languages such as Java [2] and Haskell [3], which have even higher-level concepts than numbers and strings, such as functions, classes and modules. In addition, there are many other concepts in programming languages such as data declaration, function call, inheritance declaration, pattern matching etc. that need to be expressible by text and fit into their larger language semantically. Otherwise, today's programming practice would not have been so effortless. Another software facilitator constructed with context-free grammar is the data format JSON, which allows expressing structured data of record types in text. In fact, JSON is entirely described by a context-free grammar [4]. Compared to raw binary data, JSON is much more intuitive. JSON is widely used in web application and other applications for data storage and formatting, an example of which is glTF 2.0, a format for carrying 3D scenes and models utilizing JSON [5].

Context-free grammar brings a higher level of abstraction to the fundamental tools of software technology such as programming languages and data formats. Without context-free grammar, high-level programming would not have been possible because implementing high-level programming languages would have been much more difficult if not impossible. How can ideas such as “function”, “class” and “module” be expressed in text? How can strings of digits be distinguished from strings of punctuations and still fit into the meaningful scheme of the larger language? And most importantly, how can the answers

to these questions be documented in a well-defined structure so that they can easily be consumed and developed further? Context-free grammar is one answer to these questions. The goal of this thesis is to explain what context-free grammar is and produce a rudimentary algorithm to parse context-free language for learning purposes.

2 Motivation for context-free grammar

This section introduces context-free grammar through three examples. The first example is an opening hour sign, which contains only text but can tell a reader on which day of the week and during what time a shop is open. What gives meaning to the sign is not only the text but also the structure of the text. Figure 1 shows an example opening hour sign.

```
MON - WED : 09 - 11
THU - SAT : 09 - 23
   SUN : closed
```

Figure 1. Example of an opening hour sign

In figure 1, the opening hour sign has three lines with similar structure. The structure can be described as follows. Each line starts with a range of days of the week or just one day (where the days are **MON**, **TUE**, **WED**, **THU**, **FRI**, **SAT**, **SUN**) followed by a colon, followed by a range of hours in the day or the word **closed** (where the hours are **00**, **01**, **02**, **03**, **04**, **05**, **06**, **07**, **08**, **08**, **10**, **11**, **12**, **13**, **14**, **15**, **16**, **17**, **18**, **19**, **20**, **21**, **22**, **23**). Because of this known structure, the opening hour of the shop is known.

The second example is an application form detailing the applicant's name, age and address, shown in figure 2.

```
Name : Abc Xyz
Age : 10980
Address : Sector 8
```

Figure 2. Example of application form detailing applicant's name, age and address

The subtle difference between this application form and the opening hour sign in figure 1 is that the latter is finite in length, but the former is potentially infinite. Name, age and address are arbitrary depending on customs, countries and cultures, having no fixed length. In contrast, the opening hour sign described cannot have a line exceeding a certain length because of the fixed possible patterns for days of the week, hours of the day and the fixed structure which glues them together into ranges.

The potential growth in length of a name in the application form can be expressed by saying: a name is a sequence of characters. This is not clear in terms of being explicit—what is a sequence, are empty names allowed? Establishing that the application form does not allow undisclosed name or address but allows undisclosed age, their syntax is as follows. Name is a sequence of at least one character. A sequence of at least one character is either a single character or a single character followed by a sequence of at least one character. This is rather lengthy—a more notational description could be employed [6, p.103].

$$N \rightarrow S$$

$$S \rightarrow C$$

$$S \rightarrow CS$$

$$where\ N\ represents\ a\ name, S\ represents\ a\ sequence\ of\ at\ least\ one\ character\ and\ C\ represents\ any\ character$$

Figure 3 illustrates this syntax on the name in the application form from figure 2.

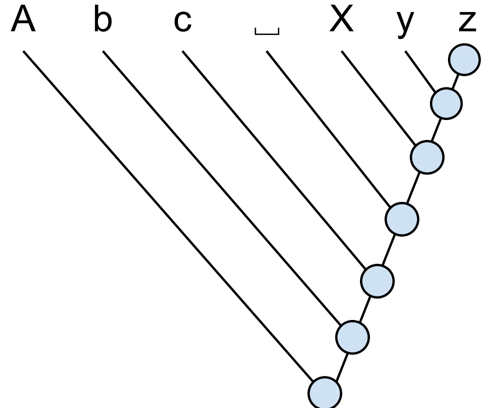

Figure 3. Structure of “a sequence of at least one character” (each circle represents a sequence of at least one character, which includes “z” because it is a sequence of one character)

This definition of character sequence is recursive, allowing its length to potentially grow to infinity from one. Figure 3 illustrates this on a sequence of seven characters, **Abc Xyz**, which is the character **A** followed by a sequence of characters starting with **b** and is followed by another sequence of characters starting with **c** and so on, ending with the sequence of a single character **z**. Because the applicant’s name is a sequence of at least one character, empty name is not allowed. In contrast, empty age is allowed, therefore, age can be described as a sequence of zero or more digits. Figure 4 illustrates this syntax of optional age on the age from figure 2.

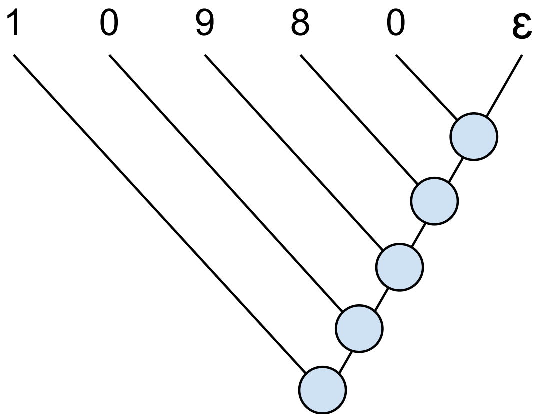

Figure 4. Structure of “a sequence of zero or more digits” (each circle represents a sequence of zero or more digits, ε denotes empty sequence)

The syntax of age and name are similar, except that an age can be empty while a name must have at least one character. Figure 4 shows this difference when compared to figure 3 through the position of ε compared to the position of the dot under character **z**.

The last example for context-free grammar is the simple numerical expression $3 * 5 + 6 \,/\, (4 * 8 + 2)$. At first glance this seems more structurally complex than the opening hour sign and application form in figure 1 and figure 2. The observed complexity comes from unclear grouping of the symbols. In figure 1 and figure 2, there is clear structuring of symbols into lines and left-right sections relative to the colons. One way to interpret $3 * 5 + 6 \,/\, (4 * 8 + 2)$ similarly is to dissect it with the same principle in mind—so that the components are clearly separated instead of lumping together in one line by considering $3 * 5 + 6 \,/\, (4 * 8 + 2)$ an addition of two terms: $3 * 5$ and $6 \,/\, (4 * 8 + 2)$.

$$+$$
$$3 * 5$$
$$6 \,/\, (4 * 8 + 2)$$

This agrees with the convention of operator precedence where $*$ and $/$ bind more strongly than $+$, or in another word, have higher precedence over $+$. This means, mathematically, $3 * 5 + 6 \,/\, (4 * 8 + 2)$ is equivalent to $(3 * 5) + (6 \,/\, (4 * 8 + 2))$, and not equivalent to $3 * (5 + 6) \,/\, (4 * 8 + 2)$. By the same convention, $*$ and $/$ have equal precedence and parentheses, $(\,)$, denote grouping. This leads to the full detailed structuring of $3 * 5 + 6 \,/\, (4 * 8 + 2)$ in figure 5.

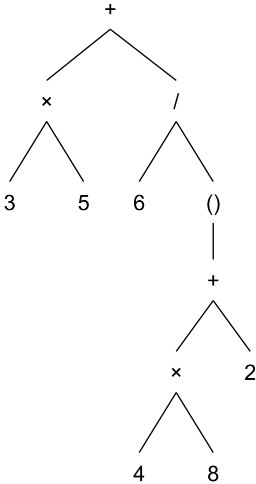

Figure 5. Structure of $3 * 5 + 6 \,/\, (4 * 8 + 2)$ considering operator precedence and parentheses grouping

A pattern emerges in figure 5. Operands of $+$ are either multiplications, divisions or numbers. Operands of $*$ or $/$ are either numbers or expressions in parentheses. These observations help define the syntax of $3 * 5 + 6 \,/\, (4 * 8 + 2)$. However, this description does not account for typical numerical expressions—for example, it does not say that operands of $+$ can also be numbers and other numerical expressions. Listing 1 shows an improved description, written in a similar style as the description of names in the application form, with the addition of the symbol '|' to express alternatives.

$$E \to T + T \mid F$$
$$T \to A * A \mid A/A \mid A$$
$$A \to N \mid (E)$$
$$N \to D \mid DN$$
$$D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$where$$
$$E\ represents\ a\ numerical\ expression,$$
$$T\ represents\ an\ operand\ of\ addition,$$
$$A\ represents\ an\ operand\ of\ multiplication\ or\ division,$$
$$N\ represents\ a\ natural\ number,$$
$$D\ represents\ a\ decimal\ digit$$

Listing 1. A symbolic description of a possible grammar for $3 * 5 + 6 \,/\, (4 * 8 + 2)$

Listing 1 shows a symbolic description of the grammar for $3 * 5 + 6 \mathbin{/} (4 * 8 + 2)$ where the grammatical components involved are denoted by capital letters (left-hand sides of the arrows) and the vertical bars separate the alternative forms of one grammatical component [6, p.109]. The rule for T — $T \rightarrow A * A \mid A/A \mid A$ — says it is either a multiplication, a division or an A, which is either a number or an expression in parentheses. An A is indivisible. This means that when reading $9109 + 6$ or $6 \mathbin{/} (8 + 4)$, mathematically, one does not interpret them as $910(9 + 6)$ or $(6/8) + 4$ respectively. (A could be thought of as "atom" for this indivisibility.) Finally, a number N is one or more decimal digits, and multiplication or division is only between two atoms. Applying this grammar on $3 * 5 + 6 \mathbin{/} (4 * 8 + 2)$ gives an illustration of its structure in figure 6.

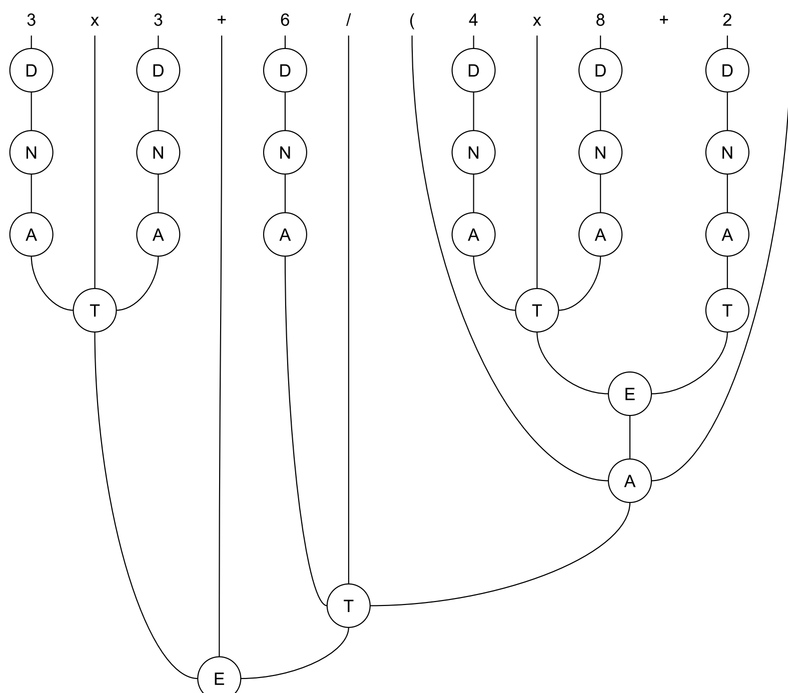

Figure 6. Structure of $3 * 5 + 6 \mathbin{/} (4 * 8 + 2)$ following the grammar in listing 1

By structuring $3 * 5 + 6 \mathbin{/} (4 * 8 + 2)$ as illustrated in figure 6, the numerical expression is "understood" because figure 6 shows exactly what the numbers are in the expression, what operations are done on them, what operations are done on the results of those operations.

These three examples show that three different "languages" – strings that tell opening hours of a shop, strings carrying an applicant's information and strings that express simple numerical operations – can be described similarly using what is called context-free

grammar [6, p.102-106]. Context-free grammar is intuitive, simple in description and powerful in application (as the three examples hinted at). Various programming languages can be described and are described at the time of writing by the principal of context-free grammar. Context-free grammar is complex enough to express complex programs, human-readable enough to be written and read by humans but also explicit enough to be interpreted and compiled by computers.

3 Expressiveness of context-free grammar

This section showcases the expressiveness of context-free grammar through two different mocked-up languages: a simple type of numerical expression found in calculators and functional expressions such as f(a, g(b), f(g)). By demonstrating the expressiveness of context-free grammar through these example languages, the components of a context-free grammar are informally introduced in their natural habitats.

3.1 Numerical expression

This section shows how numerical expressions with operator precedence can be described by a context-free grammar [6, p.103-106]. The type of numerical expression we aim to describe are numerical expressions with only integers, parenthesized expressions, exponentiation, negation, multiplication, division, addition, and subtraction. The simplest numerical expression is a positive integer such as 101, which we will customarily call a number in this section. In the grammar, negative integers such as -101 are treated specially, as a negation of a number, not directly a number.

Example 1. A few example numerical expressions are

(1) 101

(2) -101

(3) $101 + 101$

(4) $1 + 2 + 3$

(5) $(1 + 2) * 3 - 4$

The numerical expressions in example 1 (from (1) to (5)) increase in complexity. The key to seeing their structure is constructing them from a number and building up to a compound expression. For example, (1) is a number, 101, the simplest kind of numerical expression. (2) is a negated number, -101, which consists of a minus sign followed by a number. (3) is an addition between two numbers, which consists of a number followed by a $+$ sign, followed by another number. Addition can also be between other numerical expressions; this is hinted at by (5) showing a multiplication between parenthesized expressions $(1 + 2)$ and the number 3. A typical numerical expression is a compound expression made of sub-expressions combined by operators.

Employing a similar strategy, the key to constructing a grammar for numerical expression is to start with describing primitive expressions such as numbers and then describe how expressions can be combined into larger compound expressions, thus completing the description of all desired numerical expressions. However, to describe numerical expressions in this way, we must define what are "primitive" expressions. In fact, this "primitive" concept is a special case of operator precedence or binding strength. To clarify this, we examine the numerical expression $1 * 3 + 4$, which is interpreted as $(1 * 3) + 4$ but not $1 * (3 + 4)$. In mathematical convention, the multiplication operator $*$ has a higher precedence than the addition operator $+$. Another way of interpreting this is that the multiplication expression $1 * 3$ binds more strongly than the addition expression $3 + 4$, which is why the addition cannot "take" the operand 3 from the multiplication, hence $1 * 3 + 4$ is interpreted as $(1 * 3) + 4$ but not $1 * (3 + 4)$. With this, the grammar can be outlined by defining expressions with higher binding strength (precedence) such as numbers and numerical expressions in brackets first, then defining compound expressions with increasing precedence level such as exponentiation, negation, multiplication/division, addition/subtraction.

Numbers are written with digits, specifically we choose decimal digits $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. In the grammar, a number is represented by a symbol or variable N [6, p.102-104]. Later, when other components of numerical expression such as negation and multiplication etc. are described, they will be represented with their own variables. One interpretation for the word "variable" is that a number could be $1, 2, 100000292, 0, 123$ or any possible string of digits; similarly, if M represents any multiplication, it represents any possible multiplication expression. Under this interpretation, the word "variable" is an apt name for the symbols that represent a grammatical component in a grammar. A decimal digit is denoted by D.

$$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

This means a digit can be $0, 1, 2, 3, 4, 5, 6, 7, 8$ or 9. The vertical bar '|' expresses alternatives. The arrow '→' expresses the idea that given a variable D, D can expand to or derive $0, 1, 2, 3, 4, 5, 6, 7, 8$ or 9 [6, p.104]. With this, a number N as a single digit or multiple digits is described by

$$N \rightarrow D \mid DN$$

We can demonstrate the structure of a number according to $N \rightarrow D \mid DN$ by deriving a number from the variable N. In derivation, $A \Longrightarrow B$ means the string A derives the string B. A and B may contain any number of variables and other symbols, for example, $12N \Longrightarrow 120N$ because $N \Longrightarrow 0N$. [6, p.102]

$A\ derivation\ from\ left\ to\ right\ showing\ that\ N \Longrightarrow 31415$

N

$\Longrightarrow DN\ (because\ 31415\ is\ not\ a\ single\ digit)$

$\Longrightarrow 3N\ (because\ 31415\ starts\ with\ digit\ 3)$

$\Longrightarrow 3DN\ (because\ N \rightarrow DN)$

$\Longrightarrow 31N$

$\Longrightarrow 314N$

$\Longrightarrow 3141N$

$\Longrightarrow 3141D\ (because\ 31415\ has\ only\ 5\ digits\ and\ N \rightarrow D)$

$\Longrightarrow 31415$

Representing digits with D, numbers with N and other types of expressions with variables related to their precedence such as A for addition and subtraction collectively and E for any numerical expression, ordering the derivation rules of the variables by precedence from lowest to highest, the context-free grammar of numerical expression is constructed in listing 2.

$E \rightarrow A$	$E \rightarrow P_1$
$A \rightarrow A + M \mid A - M \mid M$	$P_1 \rightarrow P_1 + P_2 \mid P_1 - P_2 \mid P_2$
$M \rightarrow M * O \mid M \,/\, O \mid O$	$P_2 \rightarrow P_2 * P_3 \mid P_2 \,/\, P_3 \mid P_3$
$O \rightarrow -O \mid P$	$P_3 \rightarrow -P_3 \mid P_4$
$P \rightarrow W \wedge P \mid W$	$P_4 \rightarrow P_5 \wedge P_4 \mid P_5$
$W \rightarrow N \mid (E)$	$P_5 \rightarrow N \mid (E)$
$N \rightarrow D \mid DN$	$N \rightarrow D \mid DN$
$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$	$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Listing 2. Grammar of numerical expression capturing operator precedence (E represents a numerical expression) presented in two versions. On the left, the types of expressions are denoted by operators. On the right, they are denoted by precedence levels.

Per outline of the grammar for numerical expression in listing 2, the most strongly binding or highest precedence type of expression is placed at the bottom, the least at the top and a gradient of precedence in between. The two columns – left and right of listing 2 – present the same grammar with different sets of corresponding variables. The left is useful for the idea that each type of numerical expression is constructed with an operator in mind and capturing their precedence is capturing the precedence of the operators. The right column is useful for the idea that each type of expressions has a precedence without concerning the operators or their mathematical meaning. Together, the left and right columns capture the idea of operator precedence in numerical expressions. In addition, the right column shows that expression precedence or binding strength of arbitrary sense, not necessarily related to operators, can be captured in this type of grammar as well.

The key to seeing how listing 2 captures operator precedence is to ask what variable can derive what variable, and therefore, seeing what types of numerical expression can be operands of a specific operator and what types could not [6, p.105-106]. For example, knowing that unbracketed additions, such as $3 + 4$ cannot be operands of multiplication, helps explain why $1 * 3 + 4$ is interpreted as $(1 * 3) + 4$ and not $1 * (3 + 4)$. We express the "can derive" relationship of the variables in listing 2 through a diagram in listing 3, using the set of variables from right column of listing 1 because they are more precedence-related than those in the left column.

$$E \to P_1 \to P_2 \to P_3 \to P_4 \to P_5 \; or \; W \to N \to D$$
$$W \to E \; only \; as \; expressions \; in \; brackets \; (E)$$

Listing 3. “Can derive” relationship of variables in grammar of listing 1, right column, where $V_a \to V_b$ reads “V_a can derive V_b”

From listing 3, a variable representing expressions of precedence level a can only derive expressions of higher than or equal precedence level to a. In addition, all rules except that of W are similar to $P_1 \to P_1 * P_2 \mid P_1 / P_2 \mid P_2$ in that their operators only accept expressions of equal or higher precedence levels. These two facts are combined to show that an operator at precedence a can only accept expressions of precedence levels greater or equal to a. For example, operators of precedence 2 (P_2) are $*$ and $/$. They only accept expressions of higher or equal precedence because their operands, which are P_2 and P_3 in $P_2 \to P_2 * P_3 \mid P_2 / P_3 \mid P_3$, can only be P_2, P_3, P_4, P_5 but never P_1, because P_2 and P_3 can never derive P_1 according to listing 2 and listing 3. This means operands of multiplication and division cannot be additions or subtractions that are not in brackets. The same applies to other operators, completely capturing operator precedence in numerical expression.

In addition to operator precedence, there is the concept of associativity such as the idea that without parentheses, the expression $1 + 2 + 3 + 4 + 5$ is interpreted as $(((1 + 2) + 3) + 4) + 5$. Although this section (section 3.1) chose to focus on operator precedence, the grammar in listing 2 captures an associativity convention as well. However, this is different from the mathematical associativity of binary operators in mathematics, which is not captured by the grammar in listing 2. For example, it does not capture the equality

$$\begin{aligned} 1 + 2 + 3 + 4 + 5 &= \left(((1 + 2) + 3) + 4\right) + 5 \\ &= 1 + (2 + (3 + (4 + 5))) \end{aligned}$$

According to listing 2 (aided by listing 3), addition is left-associative, not right-associative. That is, for example, $1 + 2 + 3$ is an addition expression plus the number 3, not the number 1 plus an addition expression. The associativity captured in the grammar is purely grouping, right-associativity or left-associativity, not mathematical associativity namely, $(a \circ b) \circ c = a \circ (b \circ c)$ where $\circ$ denotes some binary operator. In listing 2, addition,

subtraction, multiplication, division ($+$, $-$, $*$, $/$) are left-associative while exponentiation (^) is right-associative.

3.2 Functional expression

This section gives a context-free grammar for mocked-up functional expressions such as f(a, g(b), f(g)) and introduces a few concepts and conventions of context-free grammar in the process. The structure of this kind of function expression is simple and less complex than that of numerical expressions described in section 3.1. However, applications of these functional expressions are various in practice; one example is the programming language Lisp [7].

The type of functional expressions to be captured in a context-free grammar in this example are those such as

$$f(a)$$
$$b(23, True)$$
$$gki(f(bcs), count(ns))$$

Defining a language through listing all strings in that language is one way to define a language. One of the definitions for “a language” is a set of strings. The set of all strings conforming to the grammar is the language of that grammar [6, p.103-104]. By giving a context-free grammar of functional expressions (listing 4), the set containing all of these functional expressions is captured.

$$F \rightarrow I(A)$$
$$A \rightarrow S \mid S, A$$
$$S \rightarrow F \mid I \mid L$$
$$I \rightarrow C \mid CI$$
$$L \rightarrow N \mid True$$
$$C \rightarrow a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z$$
$$N \rightarrow D \mid DN$$
$$D \rightarrow 0|1|2|3|4|5|6|7|8|9$$

$$The\ language\ variable\ is\ F.$$

$$The\ set\ of\ terminals\ is\ \{(,),comma,T,a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,$$
$$q,r,s,t,u,v,w,x,y,z,0,1,2,3,4,5,6,7,8,9\}.$$
$$The\ set\ of\ variables\ is\ \{F,I,A,S,L,C,N,D\}.$$

Listing 4. Grammar of mocked-up functional expressions such as $gki(f(bc),count(ns),2,True)$

The grammar of function expression in listing 4 looks more verbose than the grammar of numerical expression in listing 2, with extra elements such as the language variable, the set of terminals and the set of variables. All strings derived from the language variable (e.g. F in listing 4) form the language of the grammar. The set of terminals is usually called the alphabet of the language. It contains all the symbols composing the strings in the language. For a context-free grammar G with the set of terminals T, given any string s in the language of G, the only symbols present in s are the symbols in T and only the symbols in T. The set of terminals states precisely what symbols make up any string in the language. An example from a simplified English language is the set of symbols containing the English alphabet and punctuation marks. All words and sentences in this simplified English language are made up of symbols from this set and only symbols from this set; for example, “aesthetic” and “Hello, everyone!” contain only letters and punctuation marks. [6, p.102-106]

By convention, a context-free grammar is often specified by only stating its rules (see listing 2). All variables are identifiable by looking at the symbols on the left-hand side of the rules. After that, by elimination, knowing all the variables and the auxiliary symbols to structure the rules – the arrow and the vertical bar to denote substitution and alternatives, respectively – all terminals are known. Finally, the rule for the language variable (or start variable) is placed at the top as the first rule (top-to-bottom). The set of variables/non-terminals details all the variables used to represent components of the context-free grammar, thereby distinguishing between the symbols in the alphabet and the symbols for the variables, eliminating any ambiguity arising from the use of symbols to denote both terminals and variables in writing. The choice of the names “terminal” and “non-terminal” arises from the fact that terminal symbols such as $a, 0, x$ in listing 4’s grammar are not expandable to anything else, while variables such as F, N, L are expandable to strings of terminals and non-terminals. This property of non-terminals is the key to describing languages by a context-free grammar. For the remainder of this thesis, non-terminals will be called variables. [6, p.102-106]

If the grammar in listing 4 is put into words, the description is as follows. A functional expression is composed of an identifier followed by a list of arguments in parentheses. An identifier is a string of at least one character where the set of possible characters is the English alphabet. A list of arguments is either a single argument or multiple arguments separated by commas. A single argument can be any of three things: a functional expression, an identifier, or a literal. A literal is either a number or the string $True$. A number is a natural decimal number. (The variables' names are chosen to reflect what they represent, e.g. I for identifier, A for argument; however, their names are irrelevant, only what they expand to matters.)

The grammar in listing 4 describes a mocked-up type of functional expressions (the literals and the identifiers are limited). However, it is easy to see how it can be extended to capture more useful types of functional expressions (for example, the rules for literals can be extended to include string literals, numerical expressions, complete Boolean values).

4 Context-free grammar

4.1 Definition of context-free grammar

A context-free language G is a 4-tuple (V, A, R, S) where

1. V is the set of variables
2. A is the set of terminal symbols
3. R is the set of rules where each rule is a pair (v, s), v is a variable in V and s is a string of variables and terminals.
4. $S \in V$ is the language variable. All strings derivable from S form the language of context-free grammar G.

4.2 Derivation and reduction

The key idea to parsing context-free language in this thesis is to reduce a string bit by bit, step by step through a series of small reductions, reducing sub-strings to variables so as classify portions of the string by variables of the grammar [6, p.135-137]. A way to imagine this process is to imagine deriving a string from a variable through a series of intermediate derivations, then view the derivation in reverse to obtain the corresponding reduction. Every time, a variable v is expanded into a string w of variables and terminals in the derivation direction is when the string w is reduced to the variable v in the reduction direction [6, p.135-136]. This information can be represented pictorially and logically with a parse tree [6, p.102-103]. Parsing a string in a context-free language is to determine a parse tree involving that string and the language variable. The rest of this thesis will use a variant of parse diagram (see figure 7) to illustrate derivations and reductions.

Metropolia
University of Applied Sciences

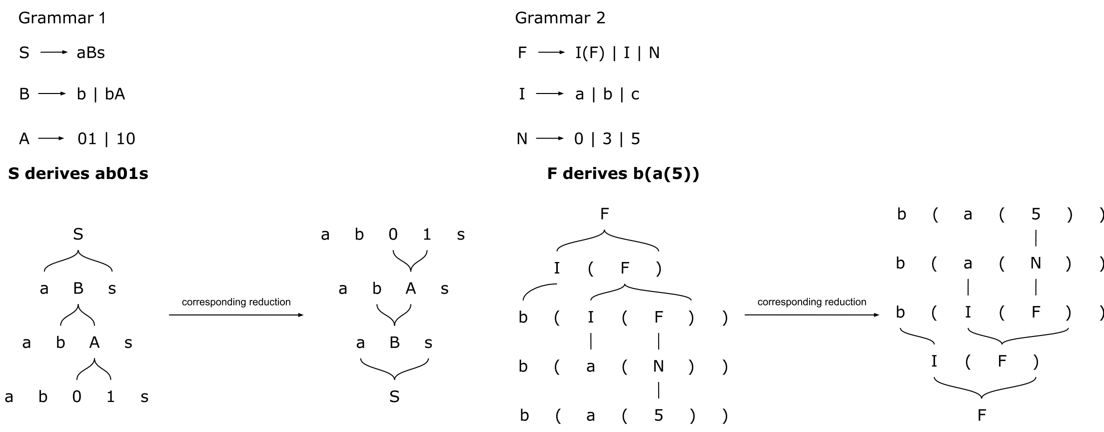

Figure 7. Derivations and their corresponding reductions

Figure 7 shows two strings being derived and then reduced. In a derivation, a variable s expanded to form a string. This is comparable to a noun deriving any noun such as “car”, “water”, “bird”, etc. In a reduction, a string and areas of that string are reduced to variables, effectively classifying them as those variables. This is comparable to “bright” being classified as an adjective, “cars” being classified as a noun or “run” being classified as a verb. Furthermore, the combination of a noun and a verb can form a sentence, such as “cars run” is a sentence. This is shown in figure 7 by repeated reductions. For example, the reduction process on the right half of figure 7 shows the substring “a(5)” in “b(a(5))” being classified as “a(N)”, which in turn is classified as “I(F)”, which in turn is classified as F, effectively classifying “a(5)” in “b(a(5))” as F. This is what parsing a string against a context-free grammar means – classifying areas of a string with variables from the context-free grammar until no area is left unclassified. This thesis shows how to achieve this classification process by simulating reductions.

However, for the purpose of parsing as a concrete algorithm, the general concept of reduction is not specific enough. Since there is generally more than one way to derive a string by a context-free grammar, there is generally more than one way to reduce a string in a context-free grammar (figure 8). [6, p.107-108] This leads to a problem when using reduction to parse a string. The interpretation of a string may vary depending on how it is reduced (figure 8).

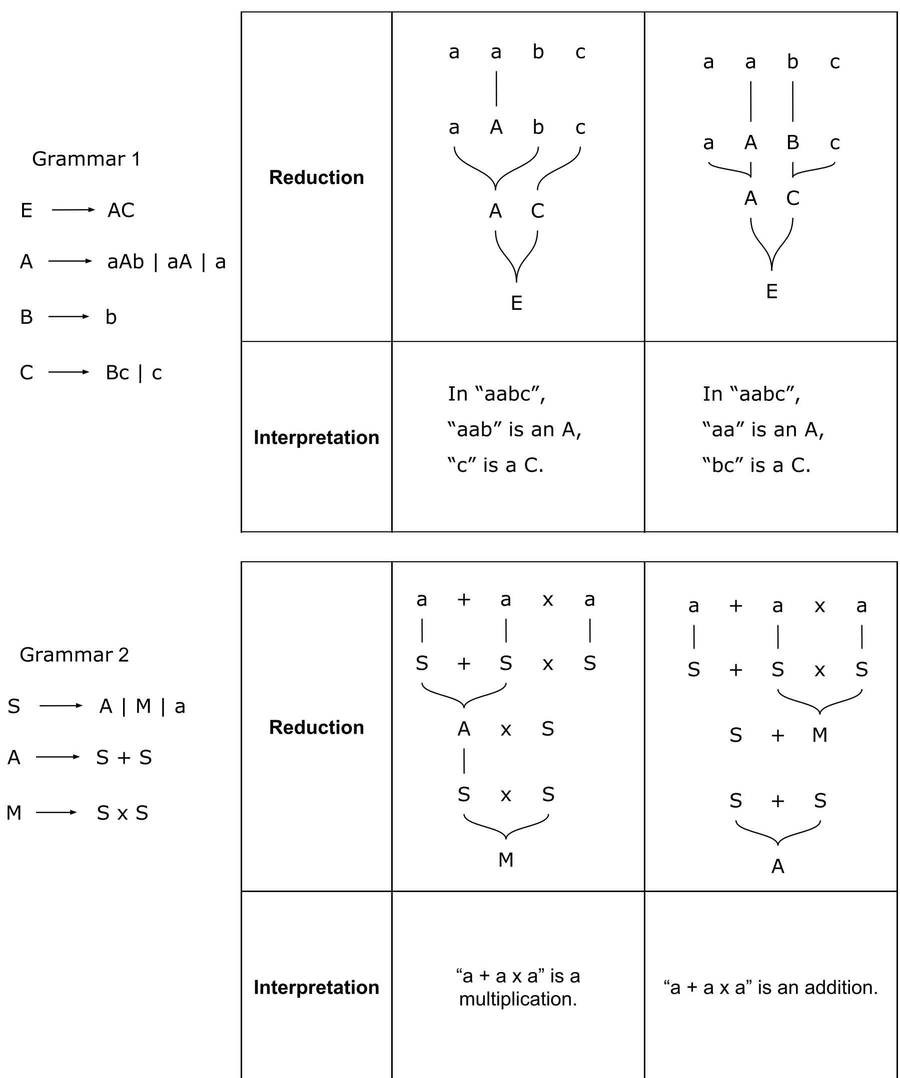

Figure 8. Two examples showing that interpretation of a string may vary depending how it is reduced

Some context-free grammars allow ambiguity in parsing. This means the information obtained from a string can vary based on how it is reduced to the language variable. In some cases, interpretation of portions of the string may change (figure 8, top). In some other cases, the meaning of the string may change entirely (figure 8, bottom).

Choosing to commit to a certain reduction scheme simplifies parsing context-free language. For this reason, this thesis deals with only one type of reduction: left-most reduction. This does not mean that parsing can only be done with one type of reduction. This only means that this thesis commits to one type of reduction for the sake of producing an example parsing algorithm. For comparison, figure 9 illustrates both left-most and

right-most reductions. Left-most reduction is shown on the left of figure 9 and right-most reduction on the right of figure 9.

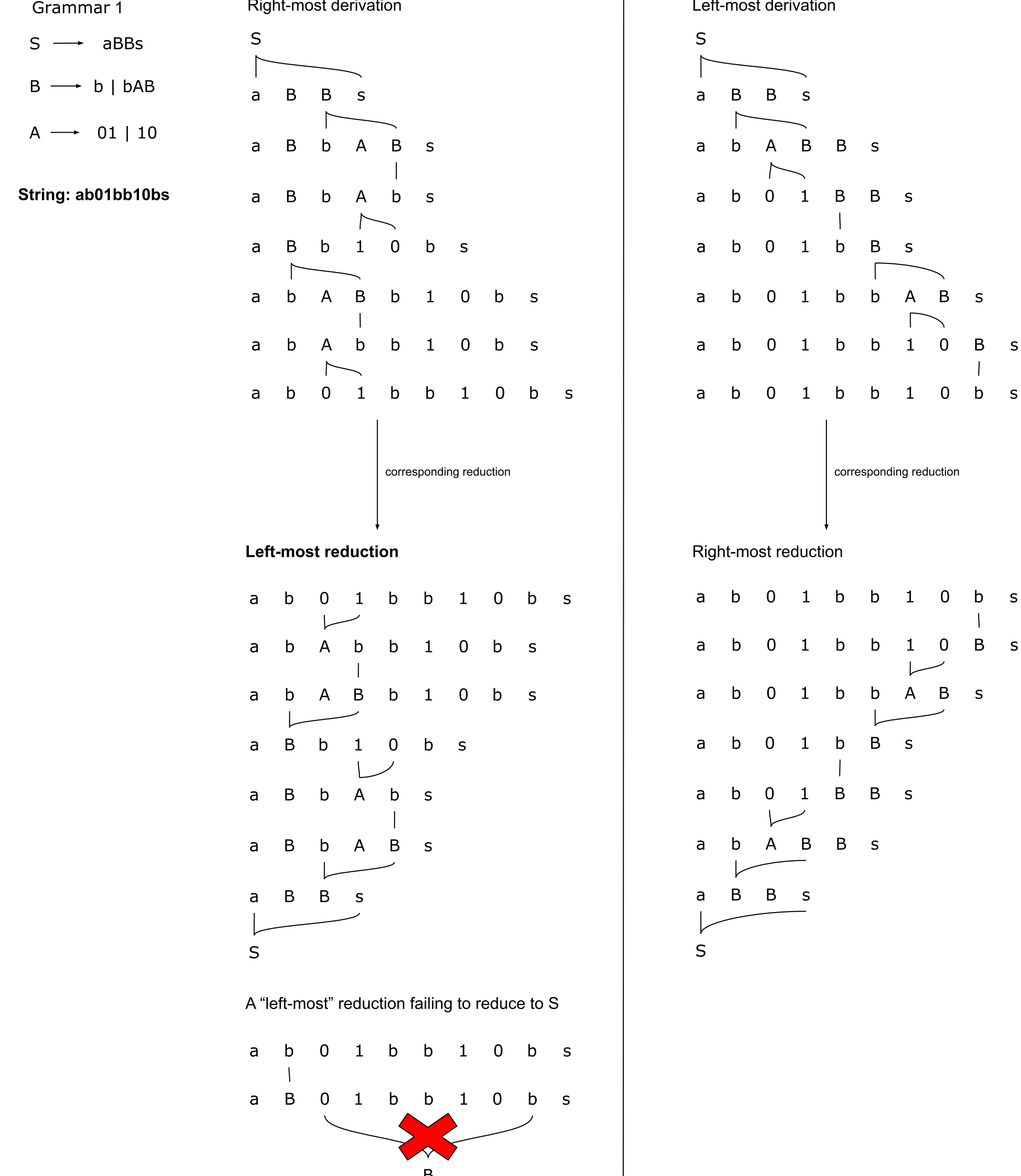

Figure 9. Left-most reduction of a string (left) and right-most reduction of the same string for comparison (right)

A left-most reduction is a right-most derivation in reverse. Figure 9 demonstrates this relationship. It also shows that reducing the left-most reduceable substring does not necessarily succeed in producing a correct reduction. In short, from this point forward, left-most reduction refers only to the reduction corresponding to a right-most derivation (figure 9).

Left-most reduction bears similar pattern to scanning a string from left to right and reducing the first reduceable sub-string. Therefore, we choose it as a simulation target to produce an example parsing algorithm. This pattern of reduction also suggests relevance to certain practical problems such as parsing from a stream where parsing must occur upon receiving input symbols without knowing when the input stream ends. [6, p.135-146].

5 Parsing context-free language by simulating left-most reduction

Parsing a string against a context-free grammar is to achieve two goals: determining if the string reduces to the language variable and determining what variables the areas of the string reduce to. The first goal answers whether the string belongs to the language or, in another word, is grammatically correct. Examples in practice include verifying that a piece of source code in a programming language is not malformed (without syntax error). The second goal extracts the meaning of the string by reducing the areas of the string to variables, effectively assigning them meaning. An example of this is shown in figure 10 through the parsing of a piece of pseudo code “int_add(int_a,_int_b)_{_return_a_+_b_}”.

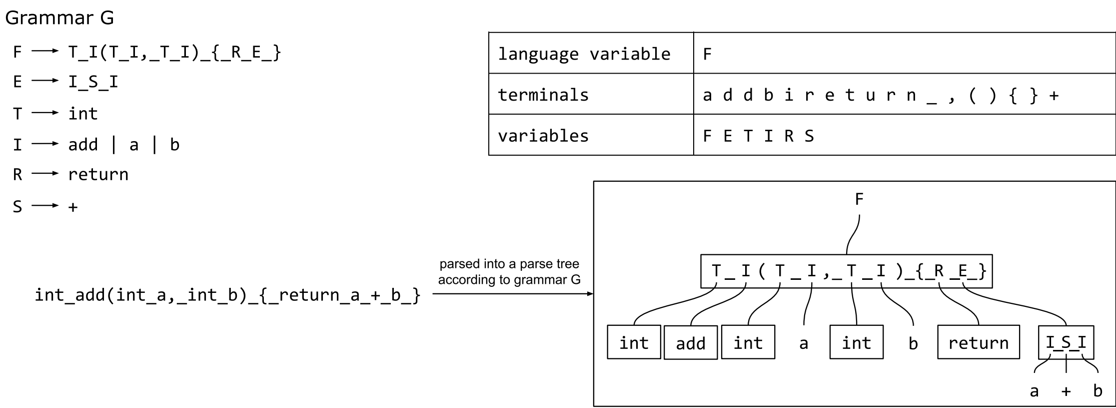

Figure 10. An example of “parsing is to produce a parse tree”

Figure 10 shows two things: reduction of the areas of “int_add(int_a,_int_b)_{_return_a_+_b_}” to variables, and reduction of the whole string “int_add(int_a,_int_b)_{_return_a_+_b_}” to F. If interpreted as a process, this process interprets the meaning of the substrings “int”s, “add”, “return”, “a”s, “b”s, “+” to be T’s, I, R, I’s, I’s, S respectively, and determines that “int_add(int_a,_int_b)_{_return_a_+_b_}” conforms to the grammar (by successfully reducing it to the language variable F). A counter-example of this is the

string “int_add(int_a,_int_b)__return_a_+_b_}” which cannot be reduced to F in any way due to the missing left brace ‘{‘.

While figure 10 demonstrates a process of parsing, it does not show how parsing can be done in detail when receiving the symbols one-by-one, left-to-right, instead of having the entire string available at all time during parsing and human-level pattern recognition to reduce the sub-strings. For learning purposes, this thesis presents an algorithm to parse a stream of input symbols by simulating left-most reduction. The input of the algorithm is a context-free grammar and a string of terminal symbols to be parsed, the output of the algorithm is zero or more parse trees of that string in the given context-free grammar. This algorithm is based on a procedure called the DK-test meant for determining if a context-free grammar is a deterministic context-free grammar [6, p.139-146].

5.1 Illustration of the parsing algorithm

The workings of the algorithm will be shown through an illustration in figure 11, 12, 13 before the algorithm is defined explicitly. Figure 11, 12, 13 illustrate parsing of the string “int_add(int_a,_int_b)_{_return_a_+_b_}” against the grammar G from figure 10. The functioning principle of the parsing algorithm is simple. The algorithm starts with a set of theories. When presented with a new piece of information, it discards the theories that are wrong and pursues the theories that are supported. The total outcome of these pursuits is the outcome of the algorithm.

In more details, the illustration is a diagram starting in figure 11 with a box containing a set of theories and the input string. The theories are denoted by what are called tracking rules (or dotted rules as they are called in [6, p.139-146]). Since the purpose of parsing is to determine the meaning of the string and its substrings, the purpose of a tracking rule is to theorize about the meaning of the string portion being read. If the string portion being read supports any theory in the box, it advances the tracking rule of that theory to pursue the theory further. Sometimes, multiple theories are supported at the same time, in which case, the diagram branches to pursue all of them (figure 12). Once a theory on the meaning of the string portion is confirmed, that string portion is assigned that meaning, that is, the string portion is reduced to a variable. The process goes on until all areas of the string have been assigned meaning or until no more theory is supported (figure 13).

Theories
1. F→>T_I(T_I,_T_I)_{_R_E_}
2. T→>int

1. The input string about to be received reduces to F, so variable T is expected.
2. Variable T is expected so the content of T, “int”, is expected.

>|int_add(int_a,_int_b)_{_return_a_+_b_} ← Input string

Reads i

To the right of the | caret is the stream of input symbols to be read. The | caret moves to consume a symbol from the input string when it is read. Anything on the right of the | caret is not yet read/consumed.

The symbol detected is the letter i, so the second theory is supported and the first theory is discarded because the second theory expects the terminal i, while the first theory expects variable T.

Since the second theory, T→>int, is pursued further, it is advanced to T→i>nt, which theorizes that the area of the string being encountered reduces to T.

T→i>nt
i>|nt_add(int_a,_int_b)_{_return_a_+_b_}

Reads n and advances the theory because n supports T→i>nt.

T→in>t
in>|t_add(int_a,_int_b)_{_return_a_+_b_}

Reads t and advances the theory because n supports T→in>t.

T→int>
int>|_add(int_a,_int_b)_{_return_a_+_b_}

T→int> indicates that the theory is confirmed, the string “int” read reduces to T.

Reduces “int” to T

The “int” being reduced to T is immediately on the left of the > caret in int>|_add(int_a,.... This is one of the purposes of the > caret.

After reduction, T replaces “int”, making >T|_add(int_a,_int_b)_{_return_a_+_b_} an intermediate string

int int

holding two things: the unread stream of input terminals on the right of the | caret and the processed elements and results on the left.

Finally, since the intermediate string is just a version of the input string where “int” is replaced by T, it is treated with the same theories as the input string in the first step (top rectangular box).

F→>T_I(T_I,_T_I)_{_R_E_}
T→>int
>T|_add(int_a,_int_b)_{_return_a_+_b_}
int

In a way, this bottom box is the same as the box at the top.

Figure 11. Illustration of parsing “int_add(int_a,_int_b)_{_return_a_+_b_}” up to the first reduction, “int” to T

Figure 11 walks through the steps of parsing “int_add(int_a,_int_b)_{_return_a_+_b_}” up to the first reduction (“int” to T). Each box in figure 11 (also in figure 12, 13) is a self-contained packet of information that can determine the next box in the diagram. Specifically, each box, with its tracking rules and bookmarked string, completely determines the next box (using information from grammar G in figure 10). There are three types of box in figure 11 (determined by their border): bold box, dotted box and dashed box. The bold boxes hold those situations where the entire string in question is suspected to correspond to the language variable. The dashed boxes hold those situations when one or more theories have been confirmed and a reduction is identified. The dotted boxes hold in-progress analyses where a theory is being investigate but has not been confirmed. These types of boxes are classifiable by the box content alone (by the set of tracking

rules and bookmarked string). The rendering – bold, dotted, dashed – are only for ease of reference.

Parsing of “int_add(int_a,_int_b)_{_return_a_+_b_}” against grammar G continues beyond figure 11. Figure 12 shows branching when multiple tracking rules are advanceable and failure when no tracking rule is advanceable.

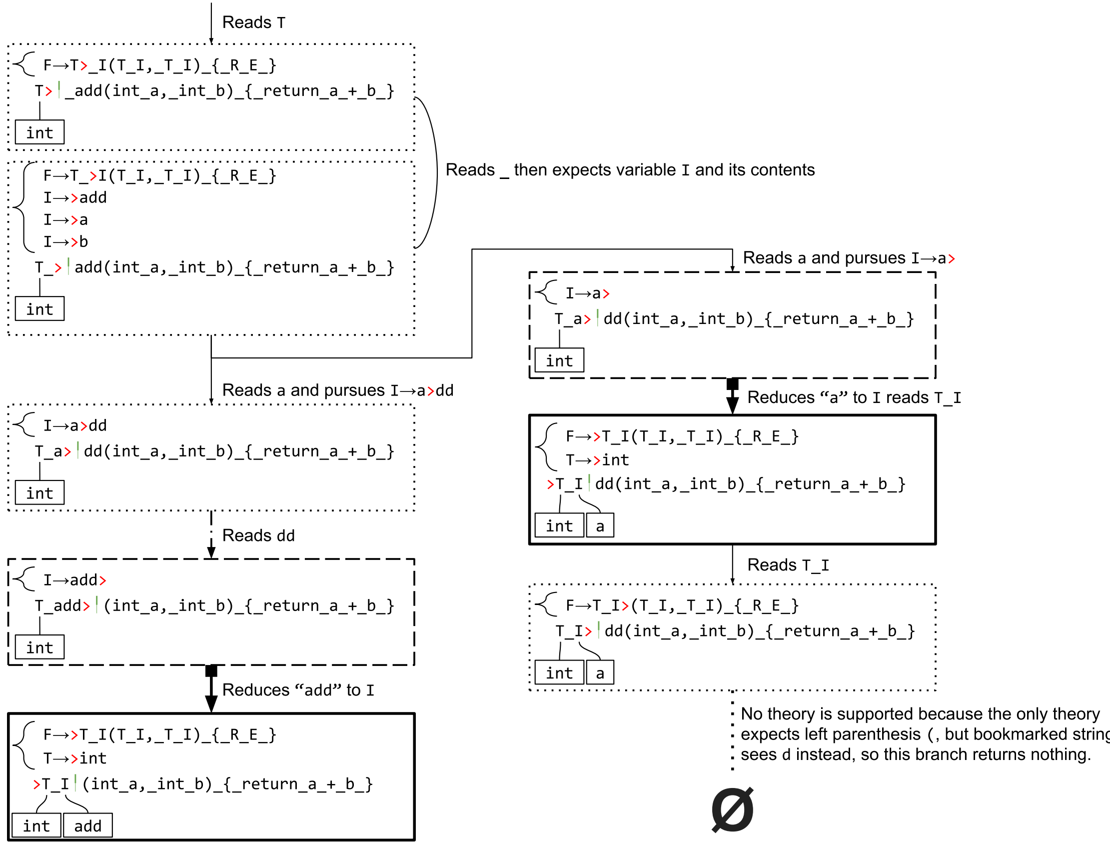

Figure 12. Parsing of “int_add(int_a,_int_b)_{_return_a_+_b_}” after the first reduction up to the second reduction, showing branching and failure

Figure 12 continues from figure 11. It shows the situation where multiple branches of computation can exist because a box can pursue multiple theories. The output of a branch of computation is a set of parse trees (because that branch may branch into more branches). When a branch fails due to unsupported theories, it returns an empty set. The union output of all branches forms the output of the algorithm. Figure 12 shows a failing branching returning the empty set but does not show any successful branches. Figure 13 continues from figure 12 but skipping ahead to the end of the evaluation to show a successful branch (the only successful branch in the parsing of

“int_add(int_a,_int_b)_{_return_a_+_b_}”). In addition, figure 13 also demonstrates the usage of the > caret in the bookmarked string more clearly than figure 11 and 12.

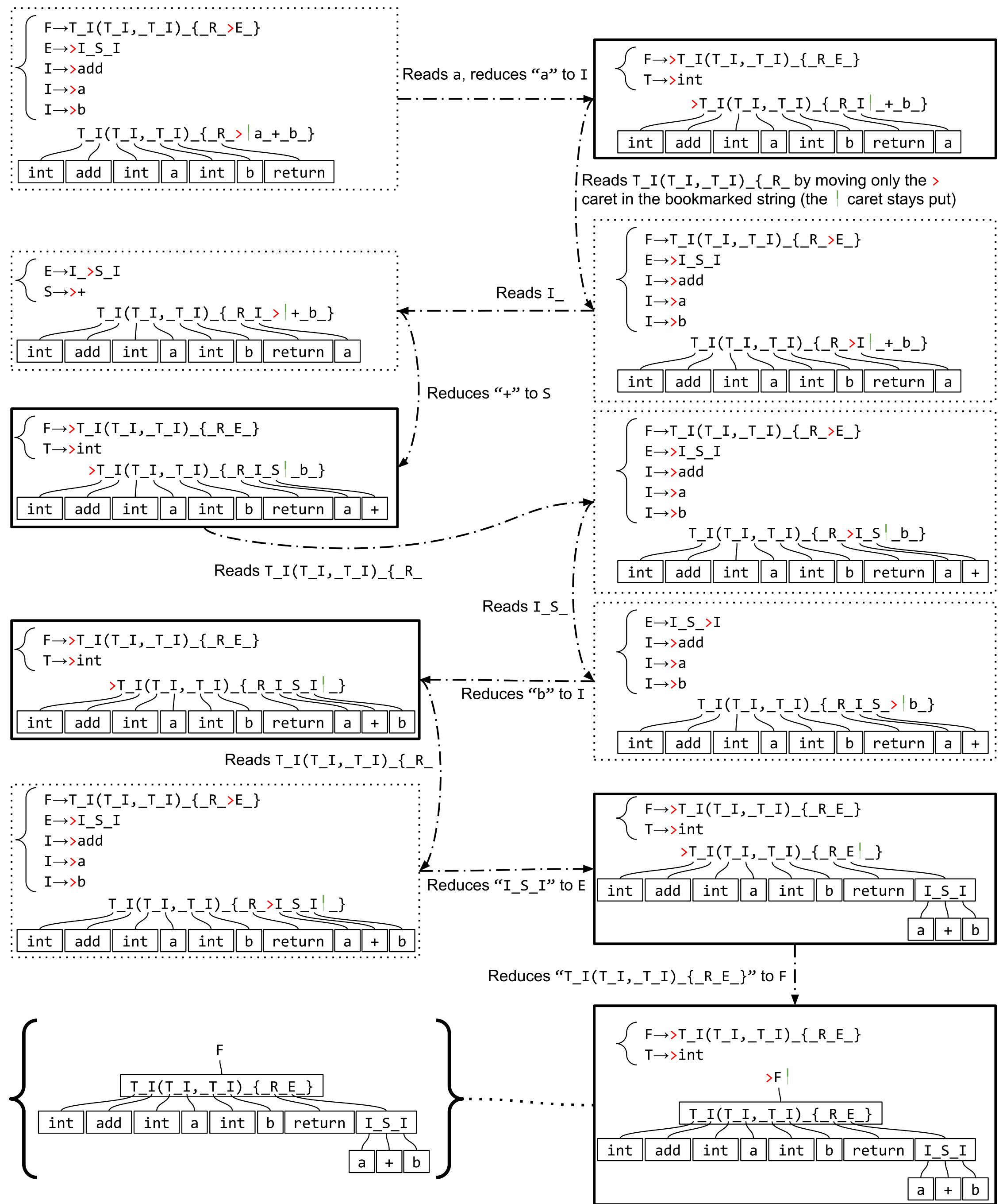

Figure 13. Parsing of ““int_add(int_a,_int_b)_{_return_a_+_b_}” producing a parse tree

Figure 13 highlights two things: that the output of the algorithm is a set of parse trees and that the algorithm constructs the parse tree incrementally. The output is a set of parse trees because sometimes parsing fails due to invalid input, and sometimes parsing

produces more than one parse tree if the grammar permits left-most reduction to do so. The algorithm is a chain of reasoning where deduction works to produce a parse tree. Each link in the chain consumes a piece of information (a variable or terminal from the bookmarked strings) to make a deduction, contributing a piece to the final image of the parse tree.

The parsing algorithm is defined by capturing the patterns in figure 11, 12, 13 and will be stated by two functions. However, in order to capture the algorithm, two key elements must first be discussed. The first element is a method to denote parse trees. The second element is a method to denote the tracking rules and their properties. Although both elements are already illustrated in figure 11, 12 and 13, to state the parsing algorithm, they require concrete notational methods.

5.2 The parsing algorithm

A tracking rule of a rule $V \to x_1 x_2 \dots x_n$ is any in the sequence ($V \to > x_1 x_2 \dots x_n,\ V \to x_1 > x_2 \dots x_n,\ V \to x_1 x_2 > x_3 \dots x_n, \dots ,\ V \to x_1 x_2 \dots x_{n-1} > x_n,\ V \to x_1 x_2 \dots x_n >$). The purpose of a tracking rule is to manage suspected possible reduction and the expected symbol that will reinforce and advance the suspicion (e.g. x_2 immediate on the right of $>$ in $V \to x_1 > x_2 \dots x_n$ is the expected symbol that will advance this tracking rule to the next form, $V \to x_1 x_2 > x_3 \dots x_n$). A tracking rule of $V \to x_1 x_2 \dots x_n$ is said to be completed or completes when it takes on the last form in the sequence, $V \to x_1 x_2 \dots x_n >$, where the caret $>$ has advanced to the end of the right-hand side. This indicates that a reduction from $x_1 x_2 \dots x_n$ to V is available (see figure 11, 12, 13, the dashed boxes). Lastly, it is important to note that when a tracking rule expects a variable (as opposed to expecting a terminal), it also makes the algorithm expect that variable's content. That is, if a grammar has three rules $H \to Abc$ and $A \to a \mid xy$ ($A \to a \mid xy$ is counted as two rules $A \to a$ and $A \to xy$), when the algorithm expects $H \to > Abc$, it must also expect $A \to > a$ and $A \to > xy$ (see figure 11, 12, 13). The tracking rule set for the expectation of variable H is $\{H \to > Abc,\ A \to > a,\ A \to > xy\}$ and is denoted as $Expect(H)$. If x is a terminal, $Expect(x) = \emptyset$. This notation will be used in the statement of the parsing algorithm later, so it is defined here.

$$Expect(x) = \begin{cases} \emptyset, if\ x\ is\ a\ terminal \\ \{V \rightarrow> p_1, \dots, V \rightarrow> p_n\} \cup \left(\bigcup_{p=p_1,\dots,p_n} Expect(first\ symbol\ of\ p) \right), \\ if\ x\ is\ a\ variable\ V\ with\ the\ rules\ V \rightarrow p_1, \dots, V \rightarrow p_n, \\ where\ p_1, \dots, p_n\ are\ strings\ of\ terminals\ and\ variables. \end{cases}$$

The parsing algorithm must return some result. The result of parsing is captured in a parse tree. A parse tree is a tree and is not to be confused with the graphical variant of parse trees used to demonstrate derivation and reduction in figure 7, 8, 9. These graphical variants are not trees. Moreover, in figure 11, 12, 13, even though the trees and tree elements depicted are meant to represent trees, they are subjected to visual interpretation and creativity, thus, unreliable as a source of tree definition.

The definition of trees used here is specialized for the purpose of expressing parse trees. Given a context-free grammar $G = (V, A, R, S)$, where V is the set of variables and A is the set of terminals (see section 4.1), trees are members of the set denoted by $t(V, A)$ which consists of two parts: leaves of the trees and nodes (branches) of the trees. The leaves are formed by terminals in the alphabet A and a node is a variable in V linked with a string of other leaves and nodes. By this construction, a node captures a reduction, where a string of terminals and variables reduces to a variable.

$$t(V, A) = A \cup \{ (v,\ t_1 t_2 \dots t_n) \mid v \in V\ and\ t_1, t_2, \dots, t_n \in t(V, A) \}$$

As an example, the tree (E, I S I, a + b) from figure 10 is concretely denoted $(E,\ (I,\ a)_(S, +)_(I,\ b))$. As a larger, more complex example, the entire parse tree from figure 10 is denoted $(F,\ (T,\ int)_(I,\ add)'('(T,\ int)_(I,\ a),_(T,\ int)_(I,\ b)')'_\{_(R,\ return)_(E,\ (I,\ a)_(S, +)_(I,\ b))_\})$ where the parentheses that are terminal symbols are surrounded by apostrophes $'$ to be distinguished from parentheses used for the tree notation.

The statement of the parsing algorithm will sacrifice rigorousness for simplicity in some places. Some notational convenience will be used, such as notation of a set of strings formed by an alphabet (i.e. a set of all strings where each character in a string comes from another set) or considering a tree node such as $t = (T,\ int)$ as the variable symbol

T in phrases such as "t is not expected by tracking rule $S \rightarrow a > bc$" or "t is expected by $S \rightarrow > Tbx$", even though t is a tree instead of the variable it wraps.

The concept of lists, sequences and strings is useful for describing the inputs of the two functions describing the algorithm. Given a set X, X^* denotes the set of all strings $x_1 \dots x_n$ where $x_1, \dots, x_n \in X$ including the empty string denoted by ε, that is, $X^* = \{x_1 \dots x_n | x_1, \dots, x_n \in X\} \cup \{\varepsilon\}$. Since strings, lists and sequences are isomorphic, the same notation is used to refer to set of lists and set of sequences from a given set. This will be useful when stating the algorithm where the boxes in figure 11, 12, 13 are characterized.

The parsing algorithm is defined by two functions. The first function, $STEP$, captures the box evolution in figure 11, 12, 13 including the branching mechanism and the return mechanism (in figure 12 and 13). The second function, $PARSE$, uses the first function to satisfy the desired input and output requirement of the parsing algorithm. The $STEP$ function does the bulk of the work.

The first step to constructing $STEP$ is to characterize a box in figure 11, 12, 13. Each box contains two elements: a set of tracking rules and a bookmarked string. While the parse tree elements and unread terminal stream have been provided enough notational devices such as $t(V, A)$ for trees and X^* for strings, lists or sequences from set X, nothing has been provided for the group of tracking rules.

To represent any group of tracking rules in a context-free grammar K, which is, for example, defined by two rules $S \rightarrow abc$, $U \rightarrow uk$, $Tr(K)$ is used. This means $Tr(K)$ denotes any subset of all the tracking rules of K which is the set $\{S \rightarrow > abc,\ S \rightarrow a > bc,\ S \rightarrow ab > c,\ S \rightarrow abc >,\ U \rightarrow > uk,\ U \rightarrow u > k,\ U \rightarrow uk >\}$. Since $Tr(K)$ is used to represent any group of K's tracking rules, it is also used to denote the set containing all groups of K's tracking rules. Expanding this to any context-free language G, $Tr(G)$ represents any group of tracking rules formed from the rules of G. Another way to define $Tr(G)$ is with the concept of power set – the set containing all subsets of a set. If $\mathcal{P}(A)$ denotes the set containing all subsets of the set A, $Tr(G)$ can be defined as $\mathcal{P}(all\ tracking\ rules\ of\ grammar\ G)$.

Now that all the necessary pieces to represent the elements of a box in figure 11, 12, 13 are available, a box can be represented by $Tr(G) \times t(V, A)^* \times t(V, A)^* \times A^*$ where $G = (V, A, R, S)$ is the context-free grammar and $\times$ denotes the Cartesian product. To simplify,

in the expression $Tr(G) \times t(V,A)^* \times t(V,A)^* \times A^*$, the $Tr(G)$ portion stands for the group of tracking rules, the first $t(V,A)^*$ portion stands for the partially constructed parse tree elements left of the > caret, the second $t(V,A)^*$ portion stands for the tree elements between the > caret and the ¦ caret (figure 11, 12, 13). The combined $t(V,A)^* \times t(V,A)^* \times A^*$ portion represents the bookmarked string in each box. This is how $Tr(G) \times t(V,A)^* \times t(V,A)^* \times A^*$ represents a box in figure 11, 12, 13.

The example parsing algorithm is presented in listing 5 and 6. The $PARSE$ function is listed first in listing 5 and the $STEP$ function is listed in listing 6. For simplicity, the algorithm assumes the context-free grammar being used, so that the functions need not include the grammar in their inputs.

$$Given\ a\ context - free\ grammar\ G = (V, A, R, S)$$

$PARSE(l)$:

$Input: a\ string\ l \in A^*$

$Output: a\ set\ of\ parse\ trees\ reducing\ l\ to\ the\ language\ variable\ S\ or\ the\ empty\ set$
$indicating\ that\ l\ is\ not\ in\ the\ language\ of\ G$

$PARSE(l) = STEP(Expect(S), \varepsilon, \varepsilon, l)$

Listing 5. The interface of the example parsing algorithm

The $PARSE$ function in listing 5 admits a string (and a context-free grammar) as input. It parses the input string by initiating the box transition with the $STEP$ function from a box expecting the language variable and the input string bookmarked at the beginning (figure 11, the top bold box). From then on, the $STEP$ function goes through the necessary steps to produce an output (listing 6).

$STEP(T, r_1, r_2, r_3)$:

$Input: a\ step\ represented\ by\ Tr(G) \times t(V, A)^* \times t(V, A)^* \times A^*$

$T \in Tr(G)\ is\ a\ group\ of\ tracking\ rules\ in\ G.$

$r_1\ is\ the\ string\ of\ tree\ elements\ left\ of\ the\ >\ caret.$

$r_2\ is\ the\ string\ of\ tree\ elements\ right\ of\ the\ >\ caret\ and\ left\ of\ the\ |\ caret.$

$r_3\ is\ the\ string\ of\ terminals\ right\ of\ the\ |\ caret.$

$Output: a\ set\ of\ parse\ trees\ or\ the\ empty\ set$

$STEP(T = \{T_1, \dots, T_m\}, \varepsilon, \varepsilon, \varepsilon) = \emptyset$

$STEP(T = \{T_1, \dots, T_m\}, r_1 = t_1 \dots t_k, \varepsilon, r_3 = s_1 s_2 \dots s_x)$ (illustration. $t_1 \dots t_k > | s_1 s_2 \dots s_x$)

$$\begin{cases} 1. fail\ case.\ if\ no\ tracking\ rule\ in\ T\ expects\ s_1, returns\ \emptyset \\ 2. advancing\ tracking\ rules\ and/or\ reduction\ (possible\ branching). \\ \quad if\ there\ are\ T_a, \dots, T_j\ in\ T\ which\ all\ expect\ s_1, then \\ \quad\quad A = \bigcup_{t = T_a, \dots, T_j} STEP(t^+, t_1 \dots t_k s_1, \varepsilon, s_2 \dots s_x), else\ A = \emptyset; \\ \quad if\ there\ are\ T_r, \dots, T_z\ in\ T\ which\ all\ complete\ and\ agree\ with\ r_1 \\ \quad\quad R = \bigcup_{(X \to p >) = T_r, \dots, T_z} STEP(Expect(S), \varepsilon, [r_1\ before\ p](X, p), r_3)\ where\ (X, p) \in t(V, A), \\ \quad\quad else\ R = \emptyset; \\ returns\ A \cup R \end{cases}$$

$STEP(T = \{T_1, \dots, T_m\}, \varepsilon, r_2 = t_1 t_2 \dots t_k, r_3)$ (illustration. $> t_1 t_2 \dots t_k | r_3$)

$$\begin{cases} 1. fail\ case.\ if\ no\ tracking\ rule\ in\ T\ expects\ t_1, returns\ \emptyset \\ 2. advancing\ tracking\ rules\ (possible\ branching). \\ if\ there\ are\ T_a, \dots, T_j\ in\ T\ which\ all\ expect\ t_1, returns\ \bigcup_{t = T_a, \dots, T_j} STEP(t^+, t_1, t_2 \dots t_k, r_3) \\ 3. successful\ parsing.\ if\ r_2\ contains\ a\ single\ node\ t_1 = (S, \dots), the\ parse\ tree\ is\ complete, \\ returns\ \{(S, \dots)\} \end{cases}$$

$STEP\big(T = \{T_1, \dots, T_m\}, r_1 = t_1 \dots t_i, r_2 = t_h t_k \dots t_q, r_3\big)$ (illustration. $t_1 \dots t_i > t_h t_k \dots t_q | r_3$)

$$\begin{cases} 1. fail\ case.\ if\ no\ tracking\ rule\ in\ T\ expects\ t_h, returns\ \emptyset \\ 2. advancing\ tracking\ rules\ and/or\ reduction\ (possible\ branching). \\ \quad if\ there\ are\ T_a, \dots, T_j\ in\ T\ which\ all\ expect\ t_h, then \\ \quad\quad A = \bigcup_{t = T_a, \dots, T_j} STEP(t^+, t_1 \dots t_i t_h, t_k \dots t_q, r_3), else\ A = \emptyset; \\ \quad if\ there\ are\ T_r, \dots, T_z\ in\ T\ which\ all\ complete\ and\ agree\ with\ r_1 \\ \quad\quad R = \bigcup_{(X \to p >) = T_r, \dots, T_z} STEP\big(Expect(S), \varepsilon, [r_1\ before\ p](X, p) t_h t_k \dots t_q, r_3\big)\ where\ (X, p) \in t(V, A), \\ \quad\quad else\ R = \emptyset; \\ returns\ A \cup R \end{cases}$$

Listing 6. The core function ($STEP$) of the example parsing algorithm (the grammar $G = (V, A, R, S)$ is assumed)

The $STEP$ function (listing 6) is listed with the aid of some minor notations. For advancements of a tracking rule $t = V \rightarrow x_1 \dots > x_k x_{k+1} \dots x_n$, the set of tracking rules obtained by advancing t is denoted by $t^+ = \{V \rightarrow x_1 \dots x_k > x_{k+1} \dots x_n\} \cup Expect(x_{k+1})$. For clarifying the condition of a reduction, listing 6 uses the phrase “a tracking rule agrees with a string”. A tracking rule is said to agree with a string r if that tracking rule takes the form $V \rightarrow x_1 \dots x_n >$ and r ends with $x_1 \dots x_n$. Finally, to aid the string replacement in a reduction, $[r\ before\ l]$ denotes the portion of the string r that precedes the string l if r ends with l. That is, given $r = x_1 \dots x_i x_j \dots x_n$, $l = x_j \dots x_n$, $[r\ before\ l]$ is the prefix $x_1 \dots x_i$.

Comparing function $STEP$ to the illustration in figure 11, 12, 13 can explain its statement in listing 6 with better context. The input of the $STEP$ function is a box in the illustration of figure 11, 12, 13. The function brings one box to the next until a box can no longer be advanced due to unmet expectations or until a box can return a parse tree. Whenever the flow of the box transition branches, $STEP$ collects the outputs from all branches at that junction into a set. Branching may occur due to multiple supported theories and/or multiple available reductions. Should a branch fail to proceed further, it returns the empty set $\emptyset$, otherwise it returns a set of parse trees. All this work is done by the $STEP$ function while the $PARSE$ function merely wraps the $STEP$ function to present a proper interface for the parsing algorithm.

The parsing algorithm presented in this thesis can be classified as a naïve parsing algorithm. One example of the limitations of the algorithm is parsing against context-free grammars which accept empty strings. That is, the algorithm is not equipped to parse context-free languages whose grammars have rules such as $V \rightarrow \varepsilon$ where ε stands for an empty string (see the second example in section 2 where the age field in an application can be empty). If forced to parse against this kind of grammar, the algorithm in listing 5 and 6 may fail to evaluate or fail to terminate. The subject of parsing these empty-string-accepting languages is complex and deserves its own treatment. Therefore, this thesis chooses not to explore it. However, for the purpose of understanding how context-free language can be parsed in a simple and constructive way, the algorithm presented in this section suffices.

6 Conclusion

Context-free grammar is a useful tool which is present in almost all places in software technologies, from important foundational tools such as programming languages to communication and interoperability tools such as data transmission formats. Everyday tools such as editors, browsers, compilers all utilize parsing in one way or another. Context-free grammar is a concise tool for describing many structured ideas. However, it is not enough to just describe an idea into language. The language must be translatable to the idea as well. Studying how to parse context-free languages is one of the first steps to studying parsing in general.

Context-free grammars use substitution to describe languages. Therefore, parsing a context-free language is just using its grammar to undo the substitutions. Organizing this into a concrete thought process forms a parsing algorithm. The example algorithm provided in this thesis, despite being rudimentary, incorporates ideas that can be expanded upon to form practical and advanced parsing algorithms. The example algorithm along with detailed illustrations of the parsing-related concepts completes this thesis as an introductory material to parsing context-free languages.

References

[1] Dennis M. Ritchie. C Reference Manual. Bell Laboratories, Murray Hill, New Jersey. April 1977.

[2] James Gosling et al. The Java® Language Specification, Java SE 11 Edition. August 2018.

[3] Haskell Community. Haskell 2010 Language Report. 2010 [Internet]. Accessed March 4, 2020. URL: https://www.haskell.org/definition/haskell2010.pdf

[4] Ecma International. The JSON Data Interchange Syntax. ECMA-404, 2nd edition. December 2017.

[5] Khronos Group. glTF 2.0 Specification, June 9, 2017 [Internet]. Accessed March 4, 2020. URL: https://github.com/KhronosGroup/glTF/blob/master/specification/2.0/README.md

[6] Michael Sipser. Introduction to the Theory of Computation, 3rd edition. Cengage Learning. 2018

[7] Paul Graham. On Lisp: Advanced Techniques for Common Lisp. Prentice Hall, 1993. [Internet] Accessed March 4, 2020. URL: http://www.paulgraham.com/onlisp.html