Reconciling Unger's parser as a top-down parser for CF grammars for experimental purposes

Łukasz Kwiatkowski
lkwiatk@few.vu.nl

August 8, 2005

Thesis submitted on August 8, 2005
to the
Facuty of Science
Vrije Universiteit, Amsterdam

Supervisor: Dr Steven Klusener

Abstract

Current parsing techniques are not appropriate when it comes to large scale software analysis. The techniques that are used in industry (LR, LL, LALR) do not support full CF grammars and require severe changes to the grammar, moreover they are not compositional. The current replacement of generalized (scannerless) LR parsing being developed at CWI in Amsterdam is heavily used within research groups, such as the Free University Amsterdam. However also this technique has its limitations. The literature considers top-down parsing as a preferred method for deriving parse trees. In this study a neglected parsing technique, Unger's algorithm, is reconsidered as the point of departure for building a general top-down parser. The algorithm presented by Stephen H. Unger in 1968 requires exponential time in the worst case however incorporation of a known parsings table reduces the time requirement to polynomial space. Combination of the known parsings table along with a number of heuristic optimizations was applied to the Unger's method. Empirical results obtained from tests on the full context-free IBM VS Cobol II grammar present a significant performance improvements, still however not sufficient to use the technique in the industrial environment. Support for the full CF syntax and the transparent, top-down, approach in deriving parse-trees makes the technique suitable for experimenting with solutions to overcome the limitations of the current parsing techniques.

Contents

Chapter 1

Introduction

The art of software reengineering heavily depends on the parsing techniques. In order to reengineer software it is convenient to parse it - that's a commonly shared opinion in the reengineering community. The existing mechanisms used to generate parsers were developed with the focus on the fact that the grammar was known in advance and is not likely to change. This fact does not hold in the field of reengineering. When facing a problem of modifying legacy systems written in mature languages like COBOL, there arises a need of making modifications to the grammar. For, there is a number of dialects that emerged over the years or a number of extensions, like embedded CICS, SQL and/or DB2.

Most of the contemporary programming languages are defined within the subsets of context-free languages like LL(1) or LALR(1). These subsets pose certain limitations on how the grammar is expressed. The process of modifying a grammar to conform to, say LALR(1), is difficult and time-consuming for the programmer, and this transformation often destroys much of its original conceptual structure. Instead of describing the language to be parsed, the grammar describes the process used to parse it. This is very unfortunate, since the grammar is the most important piece of the parsing specification. A grammar can become easily maintainable by enabling parsing of a general context-free grammar. Context-free syntax does not impose severe constraints and allows to define languages in a clear and brief manner. As a result it enables rapid prototyping of programming languages, which could be a sought property when developing a parser for a particular dialect of a language.

The need for software reengineering is strong. Abandoning renovation of the legacy systems and migration to modern solutions might be seen as an alternative. This particularly could be done by replacement of existing software with new one, or by automated conversion of existing source codes into modern languages. The latter approach portrayed by some companies as simple delivers many problems[22] making such transition not feasible in most cases. Replacement of the existing software usually introduces problems related to cost, deployment time, and incompatibility. Therefore the need for development of software reengineering tools is well established but not well recognized. Furthermore, languages deemed to be *extinct*, like COBOL, do reflect the will for *survival*. Web-enabling COBOL, multi-threading support, interaction between COBOL and Java, .NET or WebSphere are nowadays core issues discussed on industrial congresses devoted to software reengineering.[1]

1.1 Research topic

Parsing techniques should take into consideration requirements that the grammar reengineering imposes. Suitable mechanisms for modifying languages are needed in situation when grammar is the object of constant change. The world in which we live is utterly imperfect. Languages that were designed long before formalization of the grammar classes still account for a huge amount of legacy source code with huge economic interest. Parsers are still needed for such languages but the current parsing technologies appear to be inadequate to provide them. [3] lists many of the shortcomings of the parsing tools being currently in use.

In a well-established LR-parsing technology the grammar should be free of conflicts.[2] In grammar reengineering the need of adding or removing a production rule is frequent. It is common that one such modification can introduce, say 50, conflicts that need to be resolved. Grammar specification that is subject to such modifications can become an unmaintainable business-critical application that needs to be reenginnered. For that reason the current parsing techniques can be considered *harmful* [4].

The fact of working with large sets of production rules makes user-friendliness of the tools a high priority. General context-free grammars are simple and of high power of expressiveness. Well established table-driven parsers like LR(1) limit its applicability to a narrow subset of context-free grammars, in which some complex programming languages cannot be defined, or their definition becomes far to blurred.

The parsing tools used for reengineering software at the Free University Amsterdam are based on the generalized LR algorithm, a bottom-up approach in deriving the parse-tree of the input sentence, capable of handling all CF grammars. Most of these tools face problems when parsing large software portfolios. An extensive report [10] lists many of the bottlenecks encountered with the existing SGLR[3] implementation which occurred when coordinating projects for large companies, like PinkRoccade and bank ABN AMRO. To list a few:

- case insensitivety of the tools
- problems with handling *continuation symbol* (COBOL)
- expansion of copybooks (include files for COBOL)
- priorities between productions (disambiguation)
- addition of disambiguation control information to BNF grammar descriptions

[1]COBOL Future is an annual Dutch congress (started in 2004) on the future of one of the oldest programming languages.

[2]The two conflicts encountered in LR parsing are shift/reduce, when a token can be shifted on the stack or a reduction can be made, and reduce/reduce, when there are two reductions possible

[3]SGLR stands for scannerless generalized LR, and is a part of ASF+SDF environment. The parser outputs parse trees or parse forests. Scannerless means that no scanner is used to tokenize the input stream. SGLR takes characters as tokens and uses a parse table that contains enough information for both the lexical and the context-free syntax.

- disambiguation filters for SGLR

Although GLR supports full CF, some practical problems remain. These could, probably, be resolved if the parsing mechanism was transparent, however, the current SGLR algorithm is too complex to be adapted. Hence, some problems are resolved in an ad-hoc manner by changing the grammar (which is highly undesired), and others are not resolved at all. Therefore a parsing method in which it is possible to experiment with solutions to the current problems is needed.

In opposed to bottom-up parsing top-down parsing, generally considered as far more simpler and natural method for constructing parse trees, is not widely used in practice, while many beliefs and opinions would claim it a good practice. Literature is inconsistent about top-down approach. Most often top-down parsing boils down to recursive descent, which does not support full CF. The top-down methods are not however limited to certain subsets of the context-free grammars. This thesis attempts to revisit a neglected, general top-down parsing technique, Unger's algorithm, and analyze its applicability in parsing real-life languages. The simplicity of the top-down approach delivers a transparent parsing mechanism and allows to constitute a solid *foundation* for experimenting with solutions to the existing problems with software analysis.

The Unger's parser performance is drawn as the primary issue since backtracking used in the parsing algorithm causes strong performance handicap. Certain improvements and enhancements to the algorithm are discussed. Implementation presented in [9] was used as the point of departure. The parser was designed and implemented in Java. A number of comparison experiments among various variants of the Unger's parser was conducted and results are presented.

1.2 Contributions

Primarily this work contributes an optimized variant of the Unger's algorithm along with its validation against a number of artificial grammars and the full IBM's VS COBOL II grammar. The series of conducted experiments explicitly show that even though the meaning of the applied optimizations on parsing time is significant still parsing source codes of reasonable length is not feasable. Issues of concern and avenues to be considered if attempting to enable practical general top-down parsing using Unger's method are presented.

Additionally this work contributes:

- A look-ahead mechanism based on the grammar derived knowledge and its incorporation into the implementation of the Unger's parser presented in [9].
- A number of auxiliary algorithms:
  - An algorithm that enables parsing iterative grammar constructs.
  - An algorithm for finding maximal lengths of derivable terminal strings from non-terminals.

1.3 Structure of this thesis

This thesis is structured as follows:

- *Chapter 2* introduces a formal framework used throughout the thesis related to programming languages and parsing. The chapter focuses on top-down parsing approach and provides an overview of top-down parsing methods. Key elements concerning the top-down approach are discussed.
- *Chapter 3* provides a detailed description of the Unger's general context-free parsing algorithm and its analysis. Also a method that enables parsing of extended context-free grammars is presented and discussed.
- *Chapter 4* lists and discusses bottlenecks of the Unger's algorithm that affect parsing performance. A method that enables parsing in polynomial time is discussed along with its efficient implementation. The chapter also describes the particular grammar derived knowledge and its use in the presented look-ahead mechanism.

How the discussed parsing algorithm and its optimized variant behave when applied to various grammars is covered in the following chapters:

- *Chapter 5* presents an overview of the implementation of the parser used for experiments and lists its core features.
- *Chapter 6* describes the experiments conducted on the selected artificial grammars and the full IBM VS COBOL II grammar with the optimized Unger's parser.
- *Chapter 7* presents the aspects related to the Unger's parser with respect to its real-life usability, based on the related work.
- *Chapter 8* concludes the presented work and proposes a list issues for future consideration.

Chapter 2

Programming languages and top-down parsing

Programming languages find their foundation in the theory of formal languages and automata. They establish a formal way of communication between human beings and computers. Essentially they define a set of syntactic and semantic rules for describing computer programs. There are many distinct classes of programming languages. Regardless of their nature, languages, as formal entities, are subject to definition of their syntax. This step is achieved by means of formal grammars. The automata theory provides a large number of methods for syntactic analysis, parsers. Parsers are used extensively in a number of disciplines: computer science, linguistics, in document preparation and conversion, in typesetting chemical formulas and in chromosome recognition and most likely in other, not listed here. In general two major approaches applied to parsing are distinguished: top-down and bottom-up. In the thesis top-down approach is exposed for deeper analysis.

This chapter presents the formalization of constructs used throughout the thesis. It provides definitions of a formal grammar and a parser, presents the classification of languages and shows where programming languages find their place. The two approaches to parsing are described with particular focus on top-down methods. An overview of existing top-down parsing algorithms along with their specification is provided.

2.1 Context-free languages

2.1.1 Definitions

The following are standard definitions as used in [7] and [9] for formal languages and grammars. Let an *alphabet* Σ be a finite set of symbols. A *language* over an alphabet Σ is a set of strings over Σ. A device used for specifying which strings are in this set is a *grammar*.

Definition 2.1 *Let* grammar G *be defined as a 4-tuple*

$$G = (V_N, V_T, P, S)$$

such that

1. *V_N and V_T are finite sets of symbols*
2. *$V_N \cap V_T = \emptyset$*
3. *P is a set of pairs (N, α) such that*
   (a) *$N \in (V_N \cup V_T)^+$*
   (b) *$\alpha \in (V_N \cup V_T)^*$*
4. *$S \in V_N$,*

Sets V_T and V_N contain *terminal* and *non-terminal* symbols respectively. The notation T^* denotes the set of all strings over T including the empty string, denoted by ϵ. The set T^+ is defined as $T^+ = T^* - \{\epsilon\}$. P defines the set of *production rules* of the grammar, where N is a left-hand side of a production rule and α is a right-hand side, a *sentential form*. The pair (N, α) will be denoted by $N \rightarrow \alpha$. The distinguished non-terminal S is the start symbol of the grammar G.

Grammar G defines a language:

Definition 2.2

$$\alpha \Rightarrow_G \beta \quad \mathit{iff} \quad \exists_{\gamma,\mu,\delta,N} \ : \ \alpha = \gamma N \delta \ \wedge \ \beta = \gamma \mu \delta \ \wedge \ N \rightarrow \mu \in P$$

Definition 2.3

$$\alpha \Rightarrow_G^* \beta \quad \mathit{iff} \quad \exists_{\alpha_0,\alpha_1,\ldots,\alpha_m} \ : \ \alpha \Rightarrow_G \alpha_0 \Rightarrow_G \alpha_1 \Rightarrow_G \ldots \Rightarrow_G \alpha_m \Rightarrow_G \beta$$

The sequence $\alpha_0, \alpha_1, ..., \alpha_m$ is a derivation of β from α

Definition 2.4 *For grammar $G = (V_N, V_T, P, S)$ let,*

$$L(G) = \{w \mid S \Rightarrow_G^* w \ \wedge \ w \in V_T^*\}$$

$L(G)$ is the language generated by the grammar G.

2.1.2 Classification of the grammars

The Chomsky hierarchy distinguishes four classes of grammars and their corresponding languages. The differences can be seen by examining the structure of the production rules of the grammars, or the nature of the algorithm(automaton) which can be used to identify them.
The four classes are:

Language class	**Grammar**	**Automaton**
3	Regular	NFA[1]or DFA[2]
2	Context-free	Push-down automaton
1	Context-sensitive	Linear-Bounded Automaton
0	Unrestricted	Turing machine

Table 2.1: The Chomsky hierarchy

[1]Non-deterministic final state automata
[2]Deterministic final state automata

In the context of this thesis the second class will be of the main focus. Based on the definition 2.1 let the context-free grammar be defined as follows:

Definition 2.5 *Grammar $G = (V_N, V_T, P, S)$ is context-free (CF) if and only if for every production rule $N \rightarrow \alpha$ in the set P, N and α are such that*

1. $N \in V_N$
2. $\alpha \in (V_N \cup V_T)^*$

Of the four grammar types, context-free grammars represent the best compromise between power of expression and ease of implementation. Unrestricted and context-sensitive grammars are hardly used mainly for two reasons: they are user-unfriendly, in a sense that it is hard to construct a clear and readable type 0 or type 1 grammar, and all known parsers for them have exponential time requirement. Regular grammars are used mainly to describe patterns that have to be found in surrounding text therefore they find its application in building lexical scanners. Context-free grammars define a large class of languages. It turns out that in practice this class is sufficient to describe all known structures encountered in the programming languages.

2.1.3 Compositionality of grammars

From the theory of context-free languages we know that if two context-free grammars are combined, the resulting one is also context-free. The combination means here the set-theoretic union of the grammar production rules. In general we can think of a subclass of context-free grammars that is compositional. If there exists an algorithm that is capable of parsing a certain subclass of context-free languages, then when any two grammars from that subclass are combined, the algorithm can also parse the resulting one.

Compositionality of grammars is an interesting property from the point of reengineering. Suppose one wishes to construct a COBOL/CICS grammar. This can be simply obtained by combining a COBOL grammar and a CICS grammar. This cannot be achieved when using the current parsing technology. The context-free grammar subclasses for which parsing algorithms exist are LL(k), LR(k), LALR(k), etc. None of these classes are compositional. As an example let's consider the following 2 grammars which are $LL(1)$.

$$\begin{array}{ll} E \rightarrow FE' & E \rightarrow FE' \\ E' \rightarrow +F \mid \epsilon & E' \rightarrow +FE' \mid \epsilon \\ F \rightarrow (E) \mid id & F \rightarrow (E) \mid id \end{array}$$

The combined grammar

$$\begin{array}{l} E \rightarrow FE' \\ E' \rightarrow +F \mid +FE' \mid \epsilon \\ F \rightarrow (E) \mid id \end{array}$$

is no longer LL(1) since the look-ahead of 1 is no longer sufficient to decide which alternative right-hand side for E' to choose. Similar examples exist for other subclasses.

2.2 Parsing

Parsing (syntactic analysis) enables to make practical use of a formally described language. Syntax analysis provides methods that enable to tell whether a sentence w belongs to the language generated by a grammar G, and further, how this sentence can be generated from that grammar. For this reason two types of algorithms could be distinguished:

- A *Recognizer* is an algorithm that determines whether the sentence w belongs to $L(G)$
- A *Parser* is an algorithm that determines whether the sentence w belongs to $L(G)$ and also provides derivation(s) of that sentence w in grammar G.

Formally there are two approaches used in parsing to derive a sentence's structure:

1. **Top-down** - start construction of the derivations from the start symbol of the grammar and by applying to it possible production rules try to reach the symbols of the input sentence.
2. **Bottom-up** - start construction of the derivations from the symbols of the input sentence, then systematically try to build up trees until they can be connected by a single node, the start symbol of the grammar.

2.2.1 Left-most and right-most derivations

At any stage of parsing a sentential form α is being analyzed by a parser. The parser has potentially two decisions to make:

1. which non-terminal in the α a production rule should be applied to
2. which production rule to apply

The sequence of those choices results in the end in a sequence of rules that tell how the sentence has been derived. In general there are two approaches that formulate the derivation structure. The first one requires that at each stage the left-most non-terminal is being substituted by one of the possible right-hand sides, the other one, the right-most non-terminal. In the first case a parser delivers *left-most* derivation(s), in the other *right-most* derivation(s).

2.2.2 Ambiguities

An arbitrary context-free grammar can construct sentences of a language in an unequivocal way. A grammar in which a sentence has more then one derivation is called ambiguous. Ambiguity of a grammar is usually an undesired effect of parsing since it introduces necessity of selecting the appropriate construction of the sentence from the potential ones. On the other hand an ambiguous grammar allows to expresses language constructs in a clear and simple way, which for reengineering purposes is a desired property. Let's consider a well known example of a so called *dangling else* problem:

stmt → **if** expr **then** stmt **else** stmt
| **if** expr **then** stmt

Figure 2.1: If-then-else grammar

This grammar is ambiguous, as can be illustrated by constructing potential parse trees of the following sentence:

if expr1 **then** **if** expr2 **then** stmtA **else** stmtB

Using the syntax given above, this statement can be interpreted either as:

```
if expr1 then
  if expr2 then
    stmtA
  else
    stmtB
```

or as:

```
if expr1 then
  if expr2 then
    stmtA
else
  stmtB
```

Both interpretations are consistent with the syntax definition given above, but they have very different outcomes. There are certain strategies on how to approach the *selection* of the one parse-tree from the obtained set of potential solutions. More on that is explained in chapter 7.

2.2.3 Directionality

The term directionality in parsing refers to the manner in which the parsing algorithm accesses tokens of the input sentence. Two ways are distinguished:

1. Directional - the input is accessed token after token, in most cases from left to right. Parsing progresses until the last element of the input sentence is processed.
2. Non-directional - the input is accessed in a random manner. This access method applies most likely only to the Unger's algorithm.

All of the discussed further top-down approaches, except for Unger's method, analyze input sentence from left to right. It is worth pointing out that directionality implies in general the amount of memory needed for the parser to consume. Sequential access to sentence tokens does not require the input to be stored in memory while in non-directional approach this is required.

2.2.4 Search strategy

At each point in the search process the parser has to rewrite the currently analyzed sentential form by substituting a non-terminal symbol with one of its right-hand sides. The existence of more then one possible substitutions causes the search process to bifurcate. The search for the appropriate alternative by trying all possibilities is called *backtracking*. In practice there are two approaches that can implement this mechanism:

1. Depth-first search (DFS) - at the decision point K one of the alternatives is chosen and the remaining are stored for later processing. The search continues with the selected one. After the selected alternative has been fully examined the search returns to the point K and selects one of the stored alternatives and continues with the new one.

2. Breath-first search (BFS) - in contrary to DFS, at each decision point K each alternative is examined before examining the descending alternatives.

The nature of DFS is to move as quickly as possible between root and leaves. A BFS approach explores all branches at each level before going down a level. Louise Ravelli in [19] notes that for top-down parsing, depth-first building is the preferred option, since "reaching the input as fast as possible will give the earliest indication of error in hypothesis."

2.2.5 Left-recursion

CF grammars deliver great freedom in defining production rules. A very particular construct which in general causes top-down parsers to fall into an infinite loop is a left-recursion. The following production rule is left-recursive:

$$N \rightarrow N\alpha$$

Left-recursion can occur in two forms:

- *immediate* left-recursion - a non-terminal generates a sentential form starting with itself, as presented above

- *indirect* left-recursion - there exist rules in the grammar such that the most left non-terminals substitution can create a loop, e.g. $\{N \rightarrow M\alpha, M \rightarrow N\beta\}$

The current top-down parsing techniques are not able to handle left-recursive constructs and therefore restrictions apply to the grammars.

2.3 A top-down approach

A top-down parser delivers parse tree(s) of a sentence by beginning its construction from the root of the grammar, start symbol. The parser builds a derivation down to the leaves of the tree, terminal symbols. Jacob Palme in [17] comments on this approach:

"The top-down algorithm is theoretically based on the idea of using a generative grammar to produce all possible sentences in a language until one is found which fits the input sentence. This would in most cases take too much time, but there are ways to restrict the method to the most fruitful possibilities."

2.3.1 Current practice in top-down parsing

When thinking of top-down parsing techniques that could be employed in reengineering most likely the class of $LL(k)$ methods should be listed. Foundations of these methods could be sought in a more generic predictive parsing, implemented using recursive descent.

Predictive parsing

Perhaps the most simple method to program, one that can be written by hand is predictive parsing. Predictive parsing is a top-down parsing without backtracking or lookahead. The method applies only to a subset of CF grammars for which it is always possible to determine, given the current input token a and the nonterminal N, which one of the production rules for N generates a string beginning with a. There must be at most one such production in order to avoid backtracking or lookahead. If there is no such production then no parse tree exists and an error is returned. Also it is important that the grammar is not left-recursive since this could rise infinite loops during parsing when an attempt to expand N at some stage could involve N being expanded again, without any input symbols having being consumed.

$$\begin{array}{l} E \rightarrow FE' \\ E' \rightarrow +FE' \mid -FE' \\ F \rightarrow a \end{array}$$

Figure 2.2: Grammar for simple arithmetic expressions, suitable for predictive parsing

The simplest way to implement a predictive parser is to use recursive descent. The parser in such case is written according to the following rules:

1. For each non-terminal in the grammar a procedure is associated.
2. The procedure implements the *behavior* of each alternative right-hand side for the non-terminal. The *behavior* is implemented as follows:
   (a) For each non-terminal symbol in the right-hand side of a rule, procedure for that non-terminal is called.
   (b) For each terminal symbol, the current token from the input is compared with the expected terminal in the right-hand side, and consumed in case of match. Otherwise an error is reported and handled.

The following figure presents a parser for the grammar 2.2 written using recursive descent:

```
PARSE($w$)
1  $i \leftarrow 0$
2  $t \leftarrow w[i]$
3  E
```

```
E
1  F E'

E'
1  if $t =$"+"
2      then $i \leftarrow i + 1$
3           $t \leftarrow w[i]$
4           F E'
5  elseif $t =$"-"
6      then $i \leftarrow i + 1$
7           $t \leftarrow w[i]$
8           F E'



F
1  if $t =$"a"
2      then $i \leftarrow i + 1$
3           $t \leftarrow w[i]$
4      else
5           print "error"
```

Figure 2.3: Predictive parser for grammar 2.2 written using recursive descent.

LL(k) methods

The presented predictive parser was simple in implementation however the number of grammars for which it can be deployed was rather small. Introduction of look-ahead enables more grammars to be parsed. A class of grammars known as the $LL(k)$ grammars, where k represents the number of lookahead symbols needed to eliminate any element of choice from a parse, enables parsing of more languages. Specifically $LL(k)$ stands for:

- L - scanning the input from **L**eft to right
- L - constructing a **L**eftmost derivation of the input sentence
- k - using **k** symbols of *look-ahead*

Let's suppose that the parser is at stage of trying to derive the remaining portion of the input: $\alpha a \gamma$ ($\alpha \in V_T^*$ being the recognized portion of input) from $\alpha N \beta$. The parser needs to make decision which right-hand side for N to use, in case of more then one choice. Let's suppose that $N \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_k$. For each α_j the following needs to be considered:

1. if α_j begins with a terminal then it must begin with a
2. if α_j begins with a non-terminal M ($\alpha_j = M\eta$) then
   (a) one of the derivations of M must begin with a

   (b) if M can derive ϵ then the η must derive a sentential form beginning with a

3. if N can derive ϵ then β must derive a sentential form beginning with a.

One of the above conditions needs to be satisfied in order for α_j to have any further use in parsing. In $LL(k)$ methods the mechanism of look-ahead is implemented by construction of parsing tables based on the knowledge derived from the grammar G, commonly referred in the literature as $First_k(\alpha)$ and $Follow_k(A)$ sets. The definitions of those sets the reader will find in [9].

The greater the k the more languages the parser is able to accept. It also means that the algorithm becomes more complex, especially at the parse table generation phase. In fact $LL(1)$ is the most commonly deployed top-down parsing technique. $LL(1)$ grammars define a reasonably large set of programming languages, to name a few[3]: C, Java 1.5, IDL, DTD, HTML, Python, PHP 5.0. What is however important to point out is grammars for these languages are far from *nice* looking. The production rules need to be built with a great care and obey severe restrictions. For instance, left-recursive constructs are beyond the scope of $LL(1)$ grammars, since they lead into conflicts in the parse table.

2.3.2 General context-free top-down parsing

The strong look-ahead mechanism used in deterministic $LL(k)$ parser is still not sufficient to handle all context-free grammars. Introduction of backtracking allows to obtain the effect of unbounded look-ahead. Such approach is used in the general context-free Unger's parser. The algorithm is discussed in details in chapter 3. Among the current parsing tools Unger's algorithm is not employed. The use of backtracking causes the parser to perform syntax analysis in the worst case in exponential time. This fact obviously eliminates it from being used in real-life. A method to avoid exponential time requirement is explained in [9] and also further in chapter 4. Incorporation of this method makes the parser to perform in time $O(n^{k+1})$ where n is the length of the input string and k is the maximum number of non-terminals in any right-hand side in the grammar. Considering the fact that general context-free parsing requires at least cubic time makes the modified Unger's parser an interesting option to consider.

2.4 Summary

When tracing scientific articles it becomes apparent that there is a widely spread *belief* that top-down parsing is preferred to bottom-up.

Aho and Ullman in [2] state that top-down parsing is better suited to syntax directed translation than is bottom-up parsing. "When we look at translation, however, the left parse appears more desirable."

Dick Grune and Ceriel Jacobs comment on top-down approach in [9], page 252: "If one is in a position to design the grammar along with the parser, there is little doubt that LL(1) is to be preferred: not only will parsing and performing

[3]The list obtained from the grammars repository maintained by *javacc* website https://javacc.dev.java.net, which is the home page for Java LL(1) parser generater.

semantic actions be easier, text that conforms to an LL(1) grammar is also clearer to the human reader."

In real-life applications, methods that perform in linear time are most commonly used. According to Aho and Ullman *linear algorithms suffice to parse essentially all languages that arise in practice.* This is indeed true however the following trade-off must be accepted:

- the grammar must be stable, that is modifications of it are not required
- the grammar must be a particular subset of context-free grammars
- the *designed* syntax of a language may differ much from the parser suitable grammar definition

All of the above conditions are the burden of the current parsing techniques. Enabling parsing of the full context-free syntax is a necessity. User-friendliness, freedom in language structure definitions and compositionality come for granted. Furthermore, the fact that top-down parsing is far more simpler to comprehend may help resolve problems encountered with the current parsing techniques.

Chapter 3

The Unger's algorithm

A non-directional top-down parsing method presented first by Stephen H. Unger[1] is capable of working with the entire class of context-free grammars. Even though the method, which is very intuitive, might have been invented by many people independently it was first formulated in 1968 [23]. The method expects as input a sentence and a context-free grammar. The parsing process explores all different ambiguities, resulting in a so called parse-forest.

This chapter discusses in detail the idea of the Unger's method. Pseudo-code of the algorithm listed in this chapter abstracts the implementation of the parser presented in [9]. This meta description is used further in chapter 4 for explanation of the optimizations designed and their incorporation into the parser. In addition a method that enables parsing iterative constructs is presented and explained.

3.1 Parsing approach

Let G denote a context-free grammar and w be an input sentence. Unger's method works based on the following principle: if the input sentence w belongs to the languange $L(G)$ it must be derivable from the start symbol S of the grammar G. Let S be defined as

$$S \rightarrow S_1 S_2 ... S_k$$

The input sentence w must be obtainable from the sequence of symbols $S_1 S_2 ... S_k$ in a way that S_1 must derive a first part of the input, S_2 a second part, and so on.

$$\overbrace{w_1 ... w_{p_1}}^{S_1} \overbrace{w_{p_1+1} ... w_{p_2}}^{S_2} ... \overbrace{w_{p_k-1} ... w_{p_k}}^{S_k}$$

[1]Stephen H. Unger is Professor of Computer Science and Electrical Engineering since 1961 at Columbia University. He is a graduate of the Polytechnic Institute of Brooklyn and received his doctorate at MIT. Prior to coming to Columbia, he was a member of the Technical Staff of Bell Telephone Laboratories, first doing research in digital systems, and later heading a development group working on the first electronic telephone switching system. He's research in parsing area is mostly reflected in publications from 1960's.

The assignment of the symbols of the currently considered sentential form to parts of the input sentence is called partitioning of the input sentence. During the parsing process the partitioning is always done with respect to some sentential form α, therefore the given input sentence, or a portion of it, is always split into as many parts as there are symbols in the given α. Input fragments of length 0 are also permitted.

The algorithm tries to find the partitioning of the input sentence such that each symbol in a right-hand side of the start symbol of the grammar can derive its corresponding portion of the input sentence. The potential partitions are enumerated and examined. The problem is obviously recursive: if symbol S_i must derive a certain part of the input, then there must exist a partition of this part that fits a right-hand side of S_i. Eventually the right-hand side must consist of terminal symbols only, so that it can be matched explicitly with the part of the input.

3.1.1 Finding a parse tree

In order to demonstrate the parsing process let's consider the following grammar[2]:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * a \mid a$$

Figure 3.1: Simple arithmetic expressions

Let's try to parse the sentence $a + a * a$, and for this moment let's limit the enumerated partitions to those that do not contain input fragments of length 0.

1. In the initial stage the algorithm tries to derive $a + a * a$ (the entire input sentence) from E (the start symbol). There exist two alternative right-hand sides for E that could derive the input. For each alternative all possible partitions of the input are generated.

a)

E	$+$	T
a	$+$	$a * a$
a	$+a$	$*a$
a	$+a*$	a
$a+$	a	$*a$
$a+$	$a*$	a
$a + a$	$*$	a

b)

T
$a + a * a$

2. As we look at the alternative (a) it is clear that only an attempt to match $E+T$ with $(a, +, a*a)$ partition has any chance to lead to a solution. It's the only partition that allows to explicitly match the terminal symbol $+$

[2]The grammar may appear very familiar to many people since it is the most common way for describing arithmetic expressions. This syntax is however far from the ideal one. When considering the full class of context-free grammars such arithmetic expressions could be described more briefly and clearly: $E \rightarrow E + E \mid E * E \mid a$

in the right-hand side with + in the corresponding input fragment. In the first place let's try to see if $E \Rightarrow_G^* a$.

(a) The first alternative for right-hand side of E poses a question whether $E + T \Rightarrow_G^* a$? Which obviously leads to a negative answer.

(b) The second alternative poses a question whether $T \Rightarrow_G^* a$? From the grammar definition we know that the second alternative for T derives terminal a.

It has been shown that $E \Rightarrow_G^* a$. Now let's move on to the last non-terminal in the right-hand side, T.

(a) The first alternative for T, $T * a$ poses a question whether $T * a \Rightarrow_G^* a * a$? The grammar contains production $T \rightarrow a$ and therefore the answer to the question is positive.

(b) The second alternative, a, cannot yield parsing for $a * a$.

At this point a solution to the problem has been found. It has turned out that the sentence $a + a * a$ can be derived from the first alternative of the start symbol when given the following partitioning of it : $(a, +, a * a)$. By tracing back the conducted analysis we can see that the found parse-tree of the input sentence is as follows:

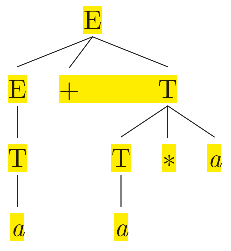


Figure 3.2: Parse tree for sentence $a + a * a$ in grammar 3.1

3. Going back to the first step, let's consider the alternative (b). This alternative poses a question if $T \Rightarrow_G^* a + a * a$? In order to answer this question two alternative right-hand sides for T need to be examined.

a)

T	$*$	a
a	$+$	$a * a$
a	$+a$	$*a$
a	$+a*$	a
$a+$	a	$*a$
$a+$	$a*$	a
$a + a$	$*$	a

b)

a
$a + a * a$

Only one partition: $(a + a, *, a)$ for alternative (a) appear to be interesting for further examination. The alternative (b) obviously fails.

4. Analyzing whether matching right-hand side $T * a$ with $(a + a, *, a)$ can deliver a solution proceeds as follows:

(a) In the first place let's try to answer whether T can derive $a + a$. Looking at the production rules we find that none of the potential alternatives will yield a solution. The answer is negative.

(b) The remaining terminal symbols $*$ and a match the corresponding input fragments explicitly.

Since not all symbols of the right-hand side could derive their corresponding input fragments no solution is delivered.

5. All possibilities have been examined. The parsing finishes resulting with one parse tree (as expected for the given unambiguous grammar).

The above presented parsing simulation has been limited to the partitions that do not contain input fragments of 0 length. Including those into the search process will enable parsing with grammars that contains ϵ rules. Additionally it will introduce a problem of the parser falling into an infinite loop. This situation is discussed in the next section.

3.1.2 All possible partitions

Let's consider the following ambiguous, left-recursive grammar for "even" simpler arithmetic expressions.

$$E \rightarrow E + E \mid a$$

Figure 3.3: "Even" simpler arithmetic expressions

Let's parse the sentence $a + a + a$ in the above grammar.

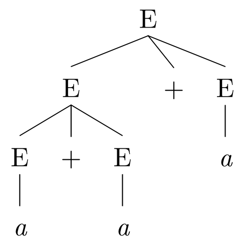
1. An attempt to derive the input from E requires two alternatives to be examined. The second alternative for right-hand side a cannot derive the input sentence. Let's concentrate on the first one. All potential partitions of the input sentence $a+a+a$ are generated. The following table lists only those that are of a potential interest[3]. Partitions for which $+$ in the right-hand side doesn't match corresponding portion of the input were rejected except for the *newly* included partitions containing input portions of 0 length.

E	$+$	E
a	$+$	$a+a$
	...	
$a+a$	$+$	a
	...	
$-$	$-$	$a+a+a$
	...	
$a+a+a$	$-$	$-$

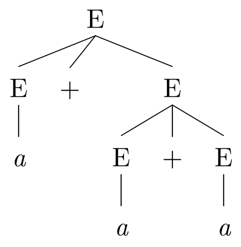
[3] Let's limit our considerations to the partitions that have any chance to deliver a solution. In a full/unoptimized parsing process all partitions need to be examined.

2. The first two partitions (a,+,a+a),(a+a,+,a) will be examined. From the set of production rules we see that E can derive $a + a$ by first substituting E with $E + E$ and further applying $E \rightarrow a$ production to all E's. The substitution can be done for either E and therefore two possible parse-trees for the input sentence can be constructed:

a)



b)



3. The last two partitions will cause some problems. An attempt to derive the entire input sentence $a+a+a$ from E will be the same as the initial parse problem. Furthermore an attempt to examine E against such partition will be called repeatedly in the parse process, as it progresses recursively, causing the algorithm to fall into an infinite loop. Such loop occurring in the parse process indicates that there exist infinitely many parsings of a sentence that can be found by following this particular path in the *search* tree. Since the only *interesting* derivations of the sentence are those that are without loops it is necessary to eliminate from the parsing process those examinations that cause falling into loops.

In order to define a solution to that problem let's introduce the concept of a *goal.*

Definition 3.1 *Let $L \rightarrow \alpha$ be a production rule, p and l be non-negative, integer numbers. A goal is a triplet,*

$$\mathbb{G} = (L \rightarrow \alpha, p, l)$$

where p and l describe the input fragment in some input sentence in a way that p denotes the starting position of the fragment in the input sentence and l denotes its length.

In the parsing process a goal is established at the moment when the selected alternative right-hand side[4] for the considered non-terminal is matched with a fragment of the input sentence. In order to avoid looping during parsing, the parser should maintain a list of goals that are under consideration in the currently followed path in the search tree. Each time before starting to examine a new goal it should be checked whether this goal hasn't been already pursued. If that is the case the goal should be ignored and parsing should progress to its next step.

[4]The goal is defined with the use of a rule instead of a right-hand side. For optimization purposes, that will be discussed further, the information about the *head* of the right-hand side is required.

3.2 The algorithm

The description of the algorithm as presented so far was supposed to give an idea of how the Unger's method approaches syntactic analysis of the input sentence. The pseudo-code presented in this section puts the aforementioned concepts into a formal scheme. The pseudo-code abstracts the Pascal implementation of Unger's parser presented in [9].

The following list explains the meaning of certain *shortcuts* used in the pseudo-code:

1. $S[G]$ - start symbol for grammar G
2. $RHS[P]$ - set of all alternative right-hand sides of the production rules for the non-terminal P
3. $rhs[i]$ - i-th symbol in the right-hand side denoted by rhs
4. PF - set of left-most derivations found during parsing (Parse-Forest)
5. D - stack for keeping rules of the constructed left-most derivation
6. Ω - operational stack of the parser
7. w - input sentence

Each element s placed on the stack Ω has the following structure:

$$s = (\mathbb{G}, \mathit{rhsPos}, \mathit{inRec}, \mathit{spans})$$

Meaning of the fields of s and the way in which they are referenced is as follows:

- $\mathbb{G}[s]$ - goal's description that corresponds to the definition 3.1, its fields are referenced as follows
  1. $\mathit{rule}[\mathbb{G}]$ - rule, it's components can be accessed directly:
     - $\mathit{lhs}[\mathbb{G}]$ - left-hand side of the rule
     - $\mathit{rhs}[\mathbb{G}]$ - right-hand side of the rule
  2. $\mathit{pos}[\mathbb{G}]$ - starting position on the input fragment
  3. $\mathit{len}[\mathbb{G}]$ - length of the input fragment
- $\mathit{rhsPos}[s]$ - cursor specifying the last processed symbol in the goal's right-hand side, starting from -1
- $\mathit{inRec}[s]$ - offset within the goals input fragment, relative to $\mathit{pos}[\mathbb{G}]$, specifying the number of terminal symbols successfully derived from the goal's right-hand side
- $\mathit{spans}[s]$ - array of maximal lengths of derivable terminal strings for each symbol in the goal's right-hand side (temporarily this array is not used, its use is explained further in chapter 4)

The tokens of the input sentence w are referenced by $w[i]$ which denotes i^{th} token of the input (i ranges from 1 to the $length[w]$. Each right-hand side of the rule $L \rightarrow rhs$ is delimited by an ϵ symbol. Such approach disables the need for check, during the analysis, whether the end of the rule's right-hand side has been reached.

Let $s = (\mathbb{G}, rhsPos, inRec, spans)$ represents an element of the parser's operational stack Ω. Let $L \rightarrow rhs$ represents the goals rule ($rule[\mathbb{G}]$), $p = pos[\mathbb{G}]$ and $l = len[\mathbb{G}]$. The following figure represents a visual interpretation of the element s:

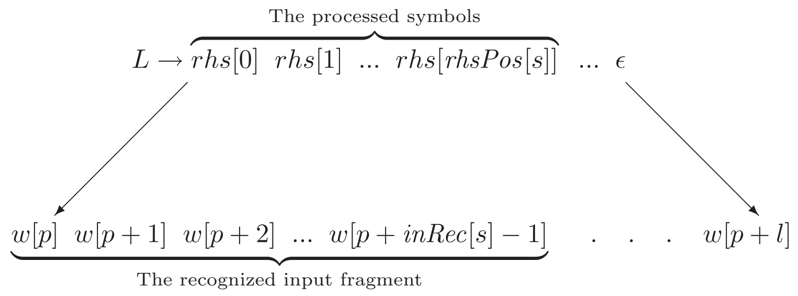

The processed symbols
$L \rightarrow rhs[0]\ \ rhs[1]\ \ ...\ \ rhs[rhsPos[s]]\ \ ...\ \ \epsilon$
$w[p]\ \ w[p+1]\ \ w[p+2]\ \ ...\ \ w[p+inRec[s]-1]\ \ .\ .\ .\ \ w[p+l]$
The recognized input fragment

Figure 3.4: A visual representation of the operational stack element, s.

Since the parser uses two stacks, D and Ω, the following operations have been defined for manipulating content of these stacks:

1. Push(S, e) - puts element e on top of the stack S
2. Pop(S) - removes the top most element from the stack S
3. Peek(S) - fetches the top most element from the stack S without removing it
4. Empty(S) - returns True if the stack S is empty or False otherwise

Additionally the following auxiliary operations are available to the parser:

1. Add(S, e) - add the element e to the set S.
2. Inc(n, i) - increment n by i and store the resulting value in n
3. Dec(n, i) - decrement n by i and store the resulting value in n

The objects G, PF, D, Ω and w are globally available. Let's now present the pseudo-code of the Unger's parser.

```
Unger-Parse($G, w$)
1 $PF \leftarrow \emptyset$
2 $D \leftarrow \emptyset$
3 $\Omega \leftarrow \emptyset$
4 TryAllRulesFor($S[G], 1, length[w]$)
5 return $PF$
```

```
TryAllRulesFor($(L, p, l)$)
1  for each $rhs \in RHS[L]$
2       do
3           TryRule($(L \to rhs, p, l)$)
```

```
TryRule($(L \to rhs, p, l)$)
 1  $g \leftarrow (L \to rhs, p, l)$            ▷ Establish a new goal
 2  if not IsToBeAvoided($(g)$)
 3     then
 4          if not KnownGoalSucceeds($(g)$)
 5             then
 6                  Push($(\Omega, (g, -1, 0, \text{Nil}))$)
 7                  StartNewKnownGoal($(g)$)
 8                  Push($(D, L \to rhs)$)
 9                  DoTopOfStack
10                  Pop(D)
11                  Pop($(\Omega)$)
```

```
DoTopOfStack
 1  $s \leftarrow$ Peek($(\Omega)$)                  ▷ Obtain the top-most element of the $\Omega$
 2  $g \leftarrow \mathbb{G}[s]$                      ▷ Extract information about the goal
 3  $C \leftarrow rhs[g][rhsPos[s] + 1]$         ▷ Obtain the next symbol to analyze
 4
 5  if $C = \epsilon$
 6     then if $inRec[s] = len[g]$
 7             then ▷ The goal $g$ has been successfully parsed
 8                  DoNextOnStack
 9  elseif $(inRec[s] < len[g])$ and $(C = w[pos[g] + inRec[s]])$
10     then
11          Inc($(rhsPos[s], 1)$)
12          Inc($(inRec[s], 1)$)
13          DoTopOfStack
14          Dec($(inRec[s], 1)$)
15          Dec($(rhsPos[s], 1)$)
16  elseif $C \in V_N$
17     then TryAllLengthsFor($(C, pos[g] + inRec[s], len[g] - inRec[s])$)
```

```
DoNextOnStack
 1  RecordKnownParsing
 2  $s \leftarrow$ Pop($(\Omega)$)
 3  if Empty($(\Omega)$)
 4     then     ▷ Solution has been found and is recorded on the $D$ stack
 5          Add($(PF, D)$)
 6     else
 7          Inc($(rhsPos[s], 1)$)
 8          Inc($(inRec[s], len[\mathbb{G}[s]])$)
 9          DoTopOfStack
10          Dec($(inRec[s], len[\mathbb{G}[s]])$)
11          Dec($(rhsPos[s], 1)$)
12  Push($(\Omega, s)$)
```

```
TryAllLengthsFor($N, p, len$)
1  for $i \leftarrow 0$ to $len$
2        do TryAllRulesFor($N, p, i$)
```

```
IsToBeAvoided($g$)
1  if $g$ is contained in any element maintained by the stack $\Omega$
2      then return True
3  return False
```

```
RecordKnownParsing
1  ▷ Do nothing (place holder for optimization explained later)
```

```
KnownGoalSucceeds($g$)
1  ▷ (place holder for optimization explained later)
2  return False
```

```
StartNewKnownGoal($g$)
1  ▷ Do nothing (place holder for optimization explained later)
```

3.2.1 Analysis

Unger-Parse is the parser's entry point procedure. The procedure expects as input

1. a context-free grammar G
2. an input[5] sentence $w = w_1 w_2 ... w_n$

The call to Unger-Parse starts the search process. An attempt to derive the entire input sentence w from the start symbol $S[G]$ of the grammar G is made by a call to TryAllRulesFor($S[G], 1, length[w]$).

TryAllRulesFor is responsible for traversing the search tree in a DFS manner. It does that by enumerating all alternative right-hand sides of the given non-terminal symbol and calling TryRule on the selected alternative (a full rule is passed as argument) and the given input fragment.

TryRule is the entry point for goals examination in the parser. First a new goal is established in g. Then it is checked if the goal hasn't been placed on the stack before in the currently analyzed path in the search tree. This check is done by a call to IsToBeAvoided and prevents the parser from looping in case of a left-recursive grammar. If the goal hasn't been examined yet, then its examination commences: the goal is placed on the operational stack along with additional values. The rule included in the goal is pushed onto the stack D. The analysis starts by calling DoTopOfStack.

DoTopOfStack is the core procedure in the parser. Itself it could be portrayed as directional top-down parser that re-derives the given fragment of input by

[5]The implemented parser expects the input to be split into lexical tokens.

making predictions. What is considered at any point in time is the sentential form (right-hand side of the goal's rule) and the remaining fragment of the input that can be determined by

- beginning position = *pos*[g]+*inRec*[s]
- length of the fragment = *len*[g]-*inRec*[s]

DoTopOfStack analyzes the input fragment, from left to right, with respect to the currently considered symbol, C, of the right-hand side being on top of operational stack. There are three possible cases that guide the parser's behavior:

1. $C = \epsilon$ - indicates that either the goal contains an ϵ-rule, or that the right-hand side has been exhausted[6]. In either case if all of the symbols of the input fragment have been matched the goal was achieved and can be removed from the operational stack by calling DoNextOnStack, in other case the goal failed, error can be reported, the procedure exits.
2. $C \in V_T$ - this implies that the symbol must match the first unrecognized symbol in the input fragment. In case of a successful match, the *inRec*[*s*] is increased by 1 to consume the recognized input token, the right-hand side symbol cursor *rhsPos*[*s*] is increased by 1, and the DoTopOfStack is called recursively to continue the analysis of the current right-hand side.
3. $C \in V_N$ - if the symbol C cannot be matched with the first unrecognized input token then it is assumed to be a non-terminal. A non-terminal encountered in the right-hand side causes the parser to make a decision as to which right-hand side from available alternatives to choose for *substitution*. At this stage the parser has to produce new potential goals from which one must succeed in order for the currently considered goal to be achieved. This is done by generating all possible assignments of the remaining portion of the input fragment to C, particularly by calling TryAllLengthsFor. The procedure enumerates all possible assignments and calls TryAllRulesFor for each.

As the parser progresses each fragment of the input is examined from left to right. Each examination of a new goal is preceded by placing on the D stack the rule under consideration, thus once all the goals on the operational stack are achieved, the D stack contains the left-most derivation of the input sentence. This situation is handled inside DoNextOnStack. The procedure takes *away* the goal from the top of the stack once it is achieved and remembers it in a local variable s. Removing from the operational stack the last goal signifies the point at which a derivative has been found. The derivative constructed on the D stack is then added to the PF set. The operational's stack content is then reconstructed, using goals recorded during recursive calls of DoNextOnStack, until a point is reached when TryAllLengths can continue parsing with a different alternative for the considered non-terminal. The parsing continues. The implicit stack maintained by DoNextOnStack allows to traverse the

[6]As it has been explained earlier the representation of the production rules requires that each right-hand side is delimited by an ϵ.

search tree in full which results in extraction of all possible parsings for the given input sentence.

The parsing process of the presented algorithm can be traced by analyzing the content of the operational stack. A sample stack trace for a simple grammar and an input sentence is presented in the appendix A.1.

3.3 Parsing iterative constructs

The usual grammar notation for production rules can be extended by introducing the usual additional descriptive operators: $+$, $*$ and ?. In general the operators describe repetition of the symbol they accompany. Presuming that N is a symbol, either terminal or non-terminal, the meaning of the operators is as follows:

1. $N+$ - "one or more N's"
2. $N*$ - "zero or more N's"
3. $N?$ - "zero or one N"

As a result of work on this thesis an algorithm that enables parsing of such constructs was designed.

3.3.1 Regular right-part grammars

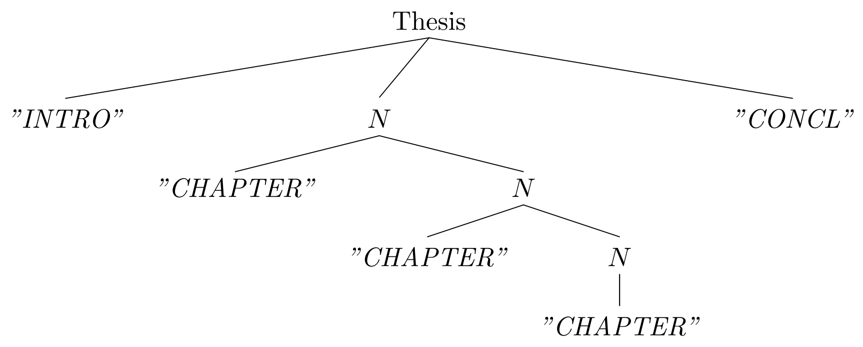
Context-free grammar with the extended notation as described above is called in literature ([9],[24]) *regular right-part* grammars. The name is derived from the way in which right-hand sides of the product rules are formed. The *Kleene star* *, + and ? operators are constructs of the regular expressions.

Regular right-part grammars do not have greater power of expressiveness then context-free grammars, since all iterative constructs have their counter parts in the *standard* notation. Their power lays in user-friendliness. When prototyping or redesigning a grammar such shortcuts come very handy. To portray the *beauty* of parse trees obtained from RRP grammars let's compare the following two grammars for describing thesis structure.

Thesis → "INTRO" N "CONCL"
N → "CHAPTER" | "CHAPTER" N

Figure 3.5: Context-free grammar

The sentence: *Intro Chapter Chapter Chapter Concl* has the following parse tree:

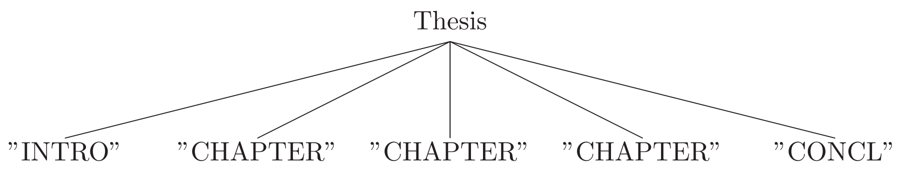


The equivalent grammar expressed in the extended notation,

$$\text{Thesis} \rightarrow \text{"INTRO" "CHAPTER"}^{+} \text{ "CONCL"}$$

Figure 3.6: Regular right-part grammar

constructs the parse tree in the following manner:



3.3.2 The algorithm

"Since the implementation of the iterative interpretation is far from trivial, most practical parser generators use the recursive interpretation in some form or another, whereas most research has been done on the iterative interpretation." from [9]

In order to enable the parser to accept RRP grammars and produce parse trees that result from iterative constructs the following approach has been taken:

1. Conversion of the grammar to its equivalent *standard* context-free notation (pre-processing)

2. Generation of the left-most derivations (parsing)

3. Reconstruction of the derivation into *RRP* left-most derivation. (post-processing)

Let G be a RRP grammar and let $A \rightarrow \alpha$ be one of it's production rules.

Definition 3.2
The production rule $A \to w$ is an instance of a production rule $A \to \alpha$, where α is a regular expression, if and only if $\alpha \Rightarrow_G^* w$.

Definition 3.3
An RRP left-most derivation is a left-most derivation of a sentence w in grammar G consisting only *of instances of the product rules of G.*

Referring back to the RRP grammar for thesis structure, the instance of the production

$$Thesis \to \text{"INTRO" "CHAPTER"}^{+} \text{ "CONCL"}$$

in the derivation of the considered sentence would be

$$Thesis \to \text{"INTRO" "CHAPTER" "CHAPTER" "CHAPTER" "CONCL"}$$

Conversion to *standard* form

As mentioned before, each product rule written using iterative constructs has its corresponding form in a standard notation without these constructs. The parser processes the rules given in the standard notation. Therefore in the pre-processing phase each rule that contains iterative operators needs to be modified. The following actions are taken:

1. For each rule of the form $A \to \alpha B{+}\beta$:
   (a) The rule $A \to \alpha B{+}\beta$ is transformed into $A \to \alpha\delta_B\beta$
   (b) Two new product rules are added:
   $$\delta_B \to B \mid B\delta_B$$
2. For each rule of the form $A \to \alpha B{*}\beta$:
   (a) The rule $A \to \alpha B{*}\beta$ is transformed into $A \to \alpha\delta_B\beta$
   (b) Two new product rules are added:
   $$\delta_B \to \epsilon \mid B\delta_B$$
3. For each rule of the form $A \to \alpha B?\beta$:
   (a) The rule $A \to \alpha B?\beta$ is transformed into $A \to \alpha\delta_B\beta$
   (b) Two new product rules are added:
   $$\delta_B \to \epsilon \mid B$$

Generation of the left-most derivations

The pre-processed grammar is provided to parser as input. The parsing process is coordinated and derivations (if exist) given on the output.

Convertion to RRP left-most derivations

The algorithm takes as input a left-most derivation of the parsed sentence. It produces the corresponding RRP derivation that corresponds to the original grammar. The non-terminal symbols added to the grammar in the preprocessing phase are distinguished as *special* symbols. This knowledge is required during rule conversion[7].
The algorithm processes the original derivation in 2 phases in which a *reconstruction table* is used to complete the conversion:

1. Scanning - each rule (read from left to right) from the input derivation is analyzed in terms of presence of *special* non-terminal symbols either in the left-hand side or the right-hand side of the product rule.
    (a) presence of the non-terminal symbol in the left-hand side causes an *update* in the *reconstruction table* of all entries that contain rules with this symbol in the right-hand side.
    (b) presence of the non-terminal symbol in the right-hand side causes insertion of a new record to the *reconstruction table.*
2. Synthesis - the information computed and collected in the *reconstruction table* is used to build a new derivation, free from *special* non-terminals. The original derivation is processed from left to right, those rules that contains *special* non-terminals are omitted, and in their place new rules are created, based on the *reconstruction table* data.

More specifically the process is explained below. The *reconstruction table* is an array in which every entry is a record consisting of 5 fields

$$(rule, symbPosition, position, active, repetition)$$

of the following meaning:

1. *rule* - rule (of the input derivation) containing a *special* non-terminal(s) in the right-hand side
2. *symbPosition* - position of the *special* non-terminal
3. *position* - position of the rule in the input derivation
4. *active* - flag set to *true* indicating that *instantiation* of the special non-terminal is not yet completed
5. *repetition* - number of symbols present in the instance

The following pseudo-code summarizes in a formal way the algorithm:
Let

1. S denotes a set of all *special* symbols present in the grammar after preprocessing phase
2. $lhs[rule]$ and $rhs[rule]$ be left-hand side and right-hand side of the $rule$, respectively

[7]In the implemented parser the *special* non-terminal symbols are denoted by $\&_n$, n being an automatically generated integer)

```
RRP-Derivation($D$)
 1  $r \leftarrow 0$
 2  for $i \leftarrow 1$ to $length[D]$
 3        do $inc[i] \leftarrow 1$
 4           $rule \leftarrow D[i]$
 5           if $lhs[rule]$ has special symbols
 6              then $k \leftarrow$ LastActive($lhs[rule]$)
 7                   if $rhs[rule] \neq \epsilon$
 8                      then $repetition[RT[k]] \leftarrow repetition[RT[k]] + 1$
 9                   if ($rhs[rule] = \epsilon$) or ($rhs[rule]$ has no special symbols)
10                      then $active[RT[k]] \leftarrow 0$
11                   $inc[i] \leftarrow 0$
12           elseif $rhs[rule]$ has special symbols
13              then    ▷ Special symbols are recorded in $RT$ in reverse order
14                   for $j \leftarrow length[rhs[rule]]$ downto 1
15                       do if $rhs[rule][j] \in S$
16                             then $RT[r] \leftarrow (rule, j, i, 1, 0)$
17                                  $r \leftarrow r + 1$
18  $j \leftarrow 1$
19  $RD \leftarrow \emptyset$
20  for $i \leftarrow 1$ to $length[D]$
21        do if $inc[i] = 1$
22      then $rule \leftarrow D[i]$
23           if $rhs[rule]$ has special symbols
24              then $RD[j] \leftarrow$ NewRule($i$)
25              else $RD[j] \leftarrow rule$
26                   $j \leftarrow j + 1$
27  return $RD$
```

```
LastActive($N$)
1  for $i \leftarrow length[RT] - 1$ downto 0
2        do $(rule, symPos, pos, act, rep) \leftarrow RT[i]$
3           if ($rhs[rule][symPos] = N$) and ($act = 1$)
4              then return $i$
5  return $-1$
```

```
NewRule($i$)
1  $r \leftarrow$ first $rule[RT]$ such that $position[RT] = i$
2  $it \leftarrow$ number of special symbols in $r$
3  $l \leftarrow$ sum of repetitions of special symbols
4  if $length[rhs[r]] - it + l > 0$
5     then $rp \leftarrow$ new rule with all special symbols in $r$ replaced their instances
6     else $rp \leftarrow lhs[r] \rightarrow \epsilon$
7  return $rp$
```

The above algorithm has been used along with the Unger's parser implementation. Sample parse trees that can be obtained with its use are presented in the appendix C.

3.4 Summary

The presented Unger's parsing algorithm is the only described top-down parser capable of accepting general context-free grammars. As a non-deterministic method the parser uses backtracking to test all possible alternatives. The following is a brief summary of the capabilities and add-on's of the parser:

1. Accepts all context-free grammars, particularly
   (a) handles left-recursive grammars
   (b) accepts ambiguous grammars
2. Delivers all possible parsings of the sentence (achieved by exhaustive search)
3. Equipped with mechanism for parsing iterative constructs is capable of accepting *user-friendly* grammars
4. Simpler to comprehend compared to bottom-up techniques

The use of backtracking and exhaustive search cause the parser to perform in exponential time. The following chapter discusses optimizations that have been applied to bring the time requirement to polynomial space by application of known parsings table mechanism as described in [9]. Specifically effective implementation of that mechanism, properly designed and incorporated look-ahead enables further performance boost which in described in chapter 6.

Chapter 4

Optimizations

Palme's opinion [17], referring to top-down parsing, in which he sees a possibility to limit the search space so that only fruitful possibilities (those that can derive the input sentence) are examined builds an instance of a problem for Unger's parser. This chapter focuses on methods that enable limiting the search space. In this particular problem it is desired to find a method that unquestionably allows to determine what established goals will *not* lead to finding a solution.

Unger's algorithm requires exponential time to deliver parsings. Such requirement is a serious obstacle for applying the parser in solving real-life problems. As claimed by Dick Grune and Ceriel Jacobs in [9], incorporation of the known parsings table into the algorithm causes it to yield parsings in a time proportional to n^{k+1}, where n is the number of tokens in the input sentence and k is the maximum number of non-terminal symbols standing in any right-hand side of the production rules in the grammar. An efficient implementation of such mechanism has been worked out and is presented. This chapter however concentrates mostly on pruning the search process by utilizing the grammar derived knowledge to design a look-ahead mechanism to be used in the presented Unger's parser. Performance improvement gained by applying the optimizations is presented in chapter 6.

4.1 Bottlenecks in Unger's parser

The presented scheme of the Unger's parser in chapter 3 is very generic. Let's use it to identify points that affect the parser's performance.

1. many of the goals established in the TryRule procedure are analyzed more then once
2. goals that have no chance of yielding parse-trees are analyzed
3. behavior of the TryAllLengthsFor procedure is inefficient

In the following sections the above issues will be addressed and solutions presented. The presented solutions involve both algorithmic and implementation optimizations.

4.2 Known parsings table

Let's recall the definition 3.1 of a goal pursued by the Unger's algorithm. The goal defines a subproblem to be solved. The solution to the problem is a partial parse tree. What can be observed when analyzing the parsing process is that many times the same goals are established and as a result the same path of finding solution to them is followed. This obviously results in unnecessary repetition of the work done before.

The partial solutions, partial parse trees, once found can be stored for further use during parsing. Such approach can be implemented in the presented Unger's parser. The idea here is to construct a known parsings table[1] that will associate goals with its partial solutions. A significant aspect of this approach is its implementation, particularly data structures used to store partial parse-trees and algorithms used to search, store and retrieve those information. An effective solution has been designed, implemented and is explained in the following sections. A sample known parsings table is presented in the appendix A.2.

4.2.1 Incorporation of the known-parsings table into the parser

The pseudo-code of the Unger's algorithm presented in chapter 3 provides three operations that enable incorporation of the mechanism for storing and retrieving known parsings. The known-parsings table in constructed *on-the-fly* as the partial parse trees become available which particularly happens after a goal is achieved. The following is a description of the operations:

1. KnownGoalSucceeds (on line number 4) in the TryRule procedure. The function is called after the established goal is accepted for analysis. At this point KnownGoalSucceeds retrieves from the known-parsings table information on the known parsings[2] for the given goal. If parsing(s) exist then the possible parse-trees are *tried* sequentially. Each partial derivative is appended to the D stack and the parsing process continues with the newly included partial parse-tree. In case when no known partial parse-trees exist the function returns false which allow for execution of the main block of the TryRule, hence searching for solution to the subproblem defined by the goal. The following pseudo-code presents the above description in a more formal way. KnownGoalSucceeds takes as input a considered goal g. The operations PushAll(S, d) and PopLastN(S, n) operate on the given stack S. The first one appends all of the rules from the derivative d to the top of the stack. The other one removes top n elements from the stack. The remaining abbreviations have the same meaning as in the pseudo-code for Unger's parser.

[1]Not to be confused with parse tables defined for LL(k) and LR(k) methods. The known parsings table differs in its structure and use during parsing.

[2]In an ambiguous grammar fragment of the input sentence can have more then one parsing. Let's suppose that in a grammar of a programming language arithmetic expressions are defined by an ambiguous syntax, e.g. $\{E \rightarrow E + E \mid E * E \mid id\}$

```
KnownGoalSucceeds($g$)
1  $T \leftarrow$ retrieve partial parse-trees for the given goal $g$
   from the known parsings table
2  if $T \neq \emptyset$
3     then for each $p \in T$
4              do PushAll($D, p$)
5                 if not Empty($\Omega$)
6                    then
7                         $s \leftarrow$ Peek($\Omega$)
8                         Inc($rhsPos[s], 1$)
9                         Inc($inRec[s], len[\mathbb{G}[s]]$)
10                        DoTopOfStack
11                        Dec($inRec[s], len[\mathbb{G}[s]]$)
12                        Dec($rhsPos[s], 1$)
13                   else
14                        ▷ $D$ contains a solutions, so add it the $PF$
15                        Add($PF, D$)
16                PopLastN($D, length[p]$)
17          return True
18    else
19          return False
```

Figure 4.1: Pseudo-code for KnownParsingSucceeds

2. StartNewKnownGoal (on line number 7) of the TryRule procedure. The procedure is called in the situation when no known partial solutions for the pursued goal exist in the known-parsings table. The procedure creates a new entry for the current goal and records the size of the D stack, to denote the starting position of a partial derivation being constructed for the goal.

3. RecordKnownGoal (on line number 1) of the DoNextOnStack procedure is called once a partial parse tree has been found for the goal residing on top of the operational stack S. The procedure adds to the known-parsings table the parse tree which is described by the last $size[D]-p$ rules on D stack, where p is the starting position recorded by StartNewKnownGoal. The added partial derivations is the solution to the achieved goal.

4.2.2 Implementation

The partial parse-trees for each combination of a rule and input fragment are very frequently accessed during parsing process. The structure of the known-parsings table and algorithms that operate on it need to be *fast* enough to

1. search whether partial parse-trees for any given goal exist
2. retrieve existing partial parse-trees to be applied in parsing
3. add new found partial parse-trees as they appear during parsing

Data structure for storing known-parsings table

The known-parsings table should allow for restoring information associated with goals. In order to organize an efficient structure let each rule in the grammar has an associated number r that enables to uniquely identify it. In such terms a goal is defined by three integer numbers (r, p, l), p and l describe input fragment as in the definition 3.1.

Let the structure of the known-parsings table be as follows. The entry level of the known-parsings table is an array indexed by rules' identifiers. Each entry of the array stores a tree structure. Each tree holds keys, represented as pairs (p, l) that determine input fragments. The keys are associated with sets that store partial left-most derivatives as sequences of product rules.

The Java API provides an implementation of Red-Black trees[3] as a $TreeMap$ class (java.util package). Instance of this class stores its elements in an ascending linear order. The TreeMap class allows for organizing structures comprising of any type, presuming that each stored element implements *Comparable* interface. The keys (pairs) are organized in a lexicographical order[4]. In the implemented prototype of the parser pairs have been implemented in a separate class that implements the *Comparable* interface. The relation that establishes lexicographical order for pairs was implemented inside $compareTo$ method of the *Comparable* interface. As reported by Java documentation, and as expected, the implementation of the Red-Black trees in Java API provides guaranteed $log(n)$ time cost for search, retrieve and insert operations. The algorithms are adaptations of those presented in [6]. The listing of the Java code for known-parsings table is presented in appendix B.1.

Cost and estimated performance gain

It can be shown [9] that the amount of work needed to construct the table cannot exceed $O(n^{k+1})$ where n is the length of the input sentence and k is the maximum number of non-terminals in any right-hand side of the product rules. The incorporated mechanism of the known-parsings table, created *on-the-fly* during parsing, takes exponential sting out of the depth-first search.

[3]A red-black tree is a binary search tree which has the following red-black properties: (a) Every node is either red or black, (b) Every leaf is black, (c) If a node is red, then both its children are black, (d) Every simple path from a node to a descendant leaf contains the same number of black nodes. By the fact that a red-black tree with n internal nodes has height at most $2log(n + 1)$ search operation can be accomplished in O(log n) time. The proof of this property can be found in [6].

[4]Let (p_1, p_2) and (r_1, r_2) be instances of two pairs. We say that $(p_1, p_2) < (r_1, r_2) \equiv (p_1 < r_1) \lor ((p_1 = r_1) \land (p_2 < r_2))$

4.3 Grammar derived knowledge

The search process establishes goals that are to be examined. These goals, when examined recursively, usually generate new goals. In reality many of the examined goals do not deliver solutions. Let's keep in mind Palme's opinion [17] and try to limit the search to the most *fruitful* possibilities. As presented in chapter 2 deterministic $LL(k)$ methods proceeded with parsing through the use of a *parse table*. Such table serves as an oracle that helps to determine, based on the considered non-terminal symbol in the current sentential-form and the current input token, what possible alternative for the right-hand side to choose. In other words the parse table helped to eliminate those alternatives that were fruitless. A similar approach can be adapted in the Unger's parser.

A grammar defines the syntax of a language. Besides that the grammar can also serve as a knowledge base and provide other valuable information that, when extracted and properly used, can aid the process of deriving parse tree(s) for a sentence. Each established goal in the Unger's algorithm can be verified prior to exploration. The verification would have to determine whether the goal has a chance to succeed. An established goal is in principle an attempt to derive from the given production rule the selected portion of the input sentence. Whether this attempt is reasonable, meaning *has a chance* to deliver a solution, can be determined after a number of checks. The checks find its grounds in the following:

- minimal and maximal lengths of the terminal sequences that the right-hand side can derive
- content of the portion of input sentence that is being considered with respect to content of the right-hand side

Foundation for building the proper checks is a particular knowledge derived from the grammar. The following sections formulate the required knowledge and describe how it is to be utilized within the Unger's parser to discard *fruitless* goals.

4.3.1 Minimum and maximum lengths of derivable N-strings

For any context-free grammar it is possible to compute the minimal and maximal lengths of derivable terminal strings, from each defined non-terminal symbol N. In the first place let's introduce the notion of an N-string.

Definition 4.1 *Let $N \in V_N$, $a_i \in V_T$ $(i = 1...n)$*

$$a_1a_2...a_n \textit{ is a derivable N-string} \quad \textit{iff} \quad N \Rightarrow_G^* a_1a_2...a_n$$

Let the minimal and maximal length of derivable N-strings be defined as follows:

Definition 4.2 *Let G be a context-free grammar, $N \in V_N$,*

$$\mathcal{D}(N) = \{\lambda \mid \lambda \textit{ is a derivable N-string in the grammar } G\}$$

Let $\mid \lambda \mid$ denotes the length (number of symbols) of N-string λ.

Definition 4.3

$$MIN(N) = \min_{\lambda \in \mathcal{D}(N)} \{k \mid k = \mid \lambda \mid\}$$

Definition 4.4

$$MAX(N) = \max_{\lambda \in \mathcal{D}(N)} \{k \mid k = \mid \lambda \mid\}$$

We define the minimum and maximum length of terminal symbols to be 1 and ϵ to be 0. In practice computation of the minimal and maximal lengths of the derivations for any non-terminal is based on the analysis of their right-hand sides. Unger presents an algorithm for determining the minimal lengths of derivable N-strings in [23]. An algorithm for computing maximal lengths of derivable N-strings was designed and is presented in the following section.

The process of computation of the minimal and maximal lengths of derivable N-strings will yield non-negative numbers for those lengths that exist, and ∞ in cases when a length cannot be determined. The latter will happen for instance in the case when a non-terminal is defined by a left-recursive production rule.

The following figure represents a sample grammar used to compute the minimal and maximal lengths of the derivable N-strings, the resulting values are presented in table 4.1

$$A \rightarrow a \mid aB$$

$$B \rightarrow A \mid aab$$

$$C \rightarrow \epsilon \mid C$$

$$D \rightarrow a \mid aa \mid aab$$

$$E \rightarrow E$$

Figure 4.2: $ABCDE$ grammar

N	$MIN(N)$	$MAX(N)$
A	1	∞
B	3	∞
C	0	∞
D	1	3
E	∞	∞

Table 4.1: Computed MIN and MAX lengths of derivable terminal strings for all non-terminals in the grammar presented in figure

Computing maximal lengths of derivable N-strings

Computation of the maximal lengths of derivable N-strings is done in 2 steps. First the knowledge on the non-terminals which can derive ϵ is obtained by calling the Find-Nullable procedure. In the second step the actual maximal values are computed by the Compute-MAX.

The Find-Nullable procedure takes as an argument a grammar G and refers to its non-terminal symbols set denoted by $NT[G]$ and set of production rules denoted by $P[G]$. The result returned by the procedure Find-Nullable is a mapping $E : NT[G] \longrightarrow \{0, 1\}$, where 1 is assigned to a non-terminal that can derive ϵ.

```
Find-Nullable($G$)
1  $u \leftarrow 1$
2  for each $n \in NT[G]$
3      do $E[n] \leftarrow 0$
4  while $u = 1$
5      do $u \leftarrow 0$
6         for each $(L \rightarrow R) \in P[G]$
7             do $AN \leftarrow 1$
8                for $j \leftarrow 1$ to $Length[R]$
9                    do if $(R[j] \in V_N)$
10                         then if $(E[R[j]] \neq 1)$
11                                 then $AN \leftarrow 0$
12                                      break
13                      elseif $(R[j] \in V_T)$
14                         then $AN \leftarrow 0$
15                              break
16               if $AN = 0$ and $E[L] \neq 1$
17                  then $u \leftarrow 1$
18                       $E[L] \leftarrow 1$
19 return $E$
```

```
COMPUTE-MAX($A, p, r$)
 1  $u \leftarrow 1$
 2  for each $n \in NT[G]$
 3      do $MAX[n] \leftarrow -1$
 4  while $u = 1$
 5      do $u \leftarrow 0$
 6         for each $(L \Rightarrow R) \in P[G]$
 7             do $M \leftarrow MAX[L]$
 8                $l \leftarrow 0$
 9                if $M = \infty$
10                   then $l \leftarrow M$
11                else
12                for $i = 0$ to $length[R] - 1$
13                    do $N \leftarrow R[i]$
14                       if $N \in V_N$
15                          then $m \leftarrow MAX[N]$
16                               if $(N = L)$ or $(m = \infty)$ or $(m = -1)$
17                                  then $l \leftarrow \infty$
18                                       break
19                                  else $l \leftarrow l + m$
20                       elseif $N \in V_T$
21                          then $l \leftarrow l + 1$
22                if $l > M$
23                   then $u \leftarrow 1$
24                        $MAX[L] = l$
```

4.3.2 Prefix, Suffix and Exclude sets

For each non-terminal in the grammar we can derive information on the content of derivable terminal strings. Let's introduce definitions that will serve as base for describing the extracted information.

Definition 4.5 *A string of terminal symbols $a_1a_2...a_n$ is an α-string if it appears as a consecutive string in some production rule of the grammar. Let*

$$\mathcal{ALPHA} = \{\lambda \mid \lambda \text{ is } \alpha\text{-string}\}$$

Let's consider the following set of production rules:

$$P \rightarrow abcAbBbceeeAAdf$$

$$R \rightarrow abcAbaaaBc$$

These production rules will contribute to $\mathcal{ALPHA}$ the strings *abc*, *b*, *bceee*, *df*, *baaa*, *c*.

The knowledge of the content of derivable N-strings can be classified into the following sets:

- $\mathcal{PRE}(N)$ - those members of $\mathcal{ALPHA}$ which can prefix N-strings
- $\mathcal{SUF}(N)$ - those members of $\mathcal{ALPHA}$ which can be suffixes of N-strings

- $\mathcal{EXC}(N)$ - those members of $\mathcal{ALPHA}$ which cannot appear in any of the N-strings

In the following sections we shall describe these sets more formally.

$\mathcal{PRE}(N)$ and $\mathcal{SUF}(N)$ sets

The aforementioned description of $\mathcal{PRE}(N)$ and $\mathcal{SUF}(N)$ sets may appear intuitive, however formal definition discloses certain details that need to be taken into consideration at construction time. Let's consider the following set of production rules for the given non-terminal N:

$$N \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$$

Let λ be an α-string, $\alpha \in (V_N \cup V_T)^*$ and $M \in V_N$. In order to define the $\mathcal{PRE}(N)$ set let's construct an auxiliary set $\mathcal{PRE}'(N)$ as follows:

1. $\mathcal{PRE}'_0(N) = \emptyset$
2. For all $i = 1..n$
   (a) if $\alpha_i = \epsilon$ then $\mathcal{PRE}'_i(N) = \mathcal{PRE}'_{i-1}(N)$
   (b) if $\alpha_i = \lambda\alpha$ then $\mathcal{PRE}'_i(N) = \mathcal{PRE}'_{i-1}(N) \cup \{\lambda\}$
   (c) if $\alpha_i = M\lambda\alpha$ and $MIN(M) = 0$ then $\mathcal{PRE}'_i(N) = \mathcal{PRE}'_{i-1}(N) \cup \{\lambda\}$
   (d) if $\alpha_i = M\alpha$ and $MIN(M) \neq 0$ then $\mathcal{PRE}'_i(N) = \mathcal{PRE}'_{i-1}(N) \cup \mathcal{PRE}'(M)$
3. $\mathcal{PRE}'(N) = \mathcal{PRE}'_n(N)$

Eventually the desired $\mathcal{PRE}(N)$ set is obtained from $\mathcal{PRE}'(N)$ by pruning all the elements that have prefixes in $\mathcal{PRE}'(N)$.

Similarly to the $\mathcal{PRE}$ sets we construct the family of $\mathcal{SUF}$ sets. In the first place we define an auxiliary set $\mathcal{SUF}'$ as follows:

1. $\mathcal{SUF}'_0(N) = \emptyset$
2. For all $i = 1..n$
   (a) if $\alpha_i = \epsilon$ then $\mathcal{SUF}'_i(N) = \mathcal{SUF}'_{i-1}(N)$
   (b) if $\alpha_i = \alpha\lambda$ then $\mathcal{SUF}'_i(N) = \mathcal{SUF}'_{i-1}(N) \cup \{\lambda\}$
   (c) if $\alpha_i = \alpha\lambda M$ and $MIN(M) = 0$ then $\mathcal{SUF}'_i(N) = \mathcal{SUF}'_{i-1}(N) \cup \{\lambda\}$
   (d) if $\alpha_i = \alpha M$ and $MIN(M) \neq 0$ then $\mathcal{SUF}'_i(N) = \mathcal{SUF}'_{i-1}(N) \cup \mathcal{SUF}'(M)$
3. $\mathcal{SUF}'(N) = \mathcal{SUF}'_n(N)$

In this case the final set $\mathcal{SUF}(N)$ is constructed from $\mathcal{SUF}'(N)$ by pruning all the elements from $\mathcal{SUF}'(N)$ that have suffixes in it.

The last step, pruning, done for both $\mathcal{PRE}$ and $\mathcal{SUF}$ sets removes redundancy of information. The sets will be used to check if prefix and suffix of the given terminal string is in either set. It is sufficient to check for shorter prefixes or suffixes.

$\mathcal{EXC}(N)$ sets

The family of $\mathcal{EXC}(N)$ sets is probably the most complex in construction compared to $\mathcal{PRE}(N)$ and $\mathcal{SUF}(N)$ sets. Essentially 3 steps can be distinguished:

For each $N \in V_N$:

1. Find subset $\Gamma(N) \subseteq \mathcal{ALPHA}$:

$$\Gamma(N) = \{\lambda \mid \lambda \in \mathcal{ALPHA} \text{ and } \lambda \text{ is used to form N-string } \}$$

2. Find subset $\mathcal{EXC}'(N) \subseteq \mathcal{ALPHA}$ of those elements that are *not* substrings of any concatenation of elements of $\Gamma(N)$.
3. Prune the $\mathcal{EXC}'(N)$ set by deleting all elements that have substrings in the set, assign the remaining elements to $\mathcal{EXC}(N)$.

The algorithms used to generate the families of $\mathcal{PRE}(N)$, $\mathcal{SUF}(N)$ and $\mathcal{EXC}(N)$ sets was presented by Unger in [23].

In order to present an instance of $\mathcal{PRE}(N)$, $\mathcal{SUF}(N)$ and $\mathcal{EXC}(N)$ sets, let's consider the following grammar:

$$A \to beCD \mid CeeD \mid Bfg$$
$$B \to eebC$$
$$C \to \epsilon$$
$$D \to aab$$

Figure 4.3: $ABCD$ grammar

The $\mathcal{ALPHA}$ set for the above production rules

$$\mathcal{ALPHA} = \{aab, be, ee, eeb, fg\}$$

provides the base for constituting $\mathcal{PRE}(N)$, $\mathcal{SUF}(N)$ and $\mathcal{EXC}(N)$ sets. The following table presents these sets as derived from the production rules of the grammar presented in figure 4.3.

N	$\mathcal{PRE}(N)$	$\mathcal{SUF}(N)$	$\mathcal{EXC}(N)$
A	$\{be,\ ee\}$	$\{aab,\ fg\}$	$\emptyset$
B	$\{eeb\}$	$\{eeb\}$	$\{fg,\ aab\}$
C	$\emptyset$	$\emptyset$	$\{be,\ ee,\ fg,\ aab\}$
D	$\{aab\}$	$\{aab\}$	$\{be,\ ee,\ fg\ \}$

Table 4.2: $\mathcal{PRE}(N)$, $\mathcal{SUF}(N)$ and $\mathcal{EXC}(N)$ sets for the $ABCD$ grammar.

4.4 Look-ahead

The knowledge derived from the grammar is the foundation for designing a proper *look-ahead* mechanism for the Unger's parser. As mentioned before the major bottleneck of the Unger's parser lays in unnecessary analysis of the goals that have no chance to deliver a solution. The look-ahead mechanism for the Unger's algorithm can be implemented by utilizing the grammar derived knowledge. Let's define the key elements of the proposed look-ahead.

1. Each time when a new goal is established in TRYRULE it is possible to verify whether the right-hand side and the input fragment satisfy some simple conditions. Particularly the following needs to be verified:
   (a) length of the input fragment with respect to minimal and maximal lengths of the derivable terminal strings from the given right-hand side. These information will allow to construct *the necessary condition.*
   (b) existence of an explicit match between terminal symbols of the right-hand side and the input fragment (A proper match).
2. Each right-hand side of the goal's rule and assigned input fragment can be verified with respect to $\mathcal{PRE}$, $\mathcal{SUF}$ and $\mathcal{EXC}$ sets for the non-terminal standing on the left-hand side of the production rule (*Quick* checks).

The following sections explain in detail the suggested verifications.

4.4.1 The necessary condition

Determining whether a particular right-hand side of a production rule has a chance to yield a parsing when tried on a specified portion of the input string can be done by checking the basic conditions that must be satisfied.

Let

$$N \rightarrow S_1 S_2 S_3 ... S_n$$

be a production rule in a grammar G. Let m and M be lengths of minimal and maximal derivable N-strings, respectively. Together along with an input fragment the rule constitutes a goal. Let's denote the length of the input fragment by l.

All of the following conditions must hold in order for the goal to have a chance to succeed.

1. $m \neq \infty$
2. $m \leq l$
3. $M \geq l$

Verifying the above conditions is simple once the required minimal and maximal lengths are computed for each non-terminal in the grammar. Presuming that the computation is done in the pre-processing phase and results stored execution of the check can be done in a constant time.

The following pseudo-code contributes a function that checks if the *necessary condition* is satisfied. This function will be further used in the description of the look-ahead mechanism.

```
IsNecessaryCond($g$)
1 if ($MIN(lhs[g]) \neq \infty$) and ($MIN(lhs[g]) \leq len[g]$) and ($MAX(L) \geq len[g]$)
2     then
3         return True
4     else
5         return False
```

4.4.2 A proper match

The *necessary condition* is obviously not a very precise check. There can be many goals that will satisfy it and yet trying to examine them will not deliver a solution. Having the *necessary condition* satisfied it makes sense to analyze the goal further. The right-hand side and the input fragment can be analyzed in terms of possible assignment of right-hand side's terminal symbols to terminal symbols of the input fragment. Let's consider figure 4.4 which represents a goal (top sequence being right-hand side of the rule and the bottom the assigned input fragment). The right-hand side's minimal lengths for non-terminal symbols are known and are as follows: $MIN(B) = 3$, $MIN(C) = 2$ and $MIN(E) = 1$.

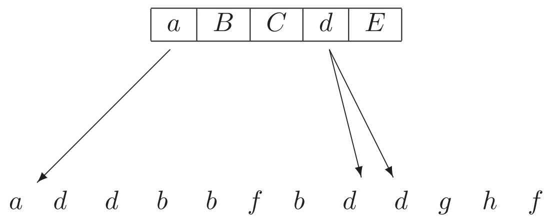


Figure 4.4: Assignment of the right-hand side's symbols to input fragment

Arrows represent potential explicit matches between terminal symbols. The first and second d's are not considered since the non-terminals B and C have to derive some portion of the input. From the computed values of $MIN(B)$ and $MIN(C)$ we see that their derivations must occupy at least 5 symbols. A derivation of E symbol can start at either last d or just after that. If there exists an assignment such that all terminal symbols of the right-hand side can be explicitly matched with the input fragment symbols then we shall assume that the goal has a chance to succeed.

A proper match verification

The verification whether a proper match existing for any given goal g is conducted as presented by the MatchExists function.

```
MatchExists($g$)
 1  if $rhs[g] = \epsilon$
 2     then if $l = 0$
 3             then return True
 4             else  return False
 5     else
 6           if not IsNecessaryCond($g$)
 7              then return False
 8           if prefix of $rhs[g]$ does not match the prefix of the fragment(p,l)
 9              then return False
10           if suffix of $rhs[g]$ does not match the suffix of the fragment(p,l)
11              then return False
12           $rl \leftarrow$ position of the first non-terminal symbol in $rhs[g]$
13           $ru \leftarrow$ position of the last non-terminal symbol in $rhs[g]$
14           $il \leftarrow rl$
15           $iu \leftarrow len[g] - ru$
16           $lastNT \leftarrow$ False
17           $R \leftarrow rhs[g]$
18           for $i = rl$ to $ul$
19                do if $R[i] \in V_T$
20                      then if $(lastNT)$
21                              then while $(il < len[g])$ and $(w[p + il] \neq R[i])$
22                                        do $q \leftarrow w[p + il] \neq R[i]$
23                                           $il \leftarrow il + 1$
24                                   if $il = iu$ and not $q$
25                                      then return False
26                                   $il \leftarrow il + 1$
27                              else
28                                   if $w[p + il] \neq R[i]$
29                                      then return False
30                                   $il \leftarrow il + 1$
31                           $lastNT \leftarrow$ False
32                   elseif $R[i] \in V_N$
33                      then $il \leftarrow il + MIN(R[i])$
34                           $lastNT \leftarrow$ True
35           if $il \leq up$
36              then return True
37              else  return False
```

MatchExists returns True if a proper match for the given goal exists or False otherwise. In lines 1-4 the case of an ϵ rule is handled separately - this is due to a higher performance gained by the simple check in line 2. All other rules are verified as follows:

1. The necessary condition is checked.
2. *Terminal* prefixes of the input fragment and the right-hand side are checked.
3. *Terminal* suffixes of the input fragment and the right-hand side are checked.
4. The *inside* portion of the right-hand side that contains non-terminal symbols is examined. The *lastNT* variable is set to False to indicate the the

previous considered symbol in the right-hand side was terminal. For each symbol in the right-hand side, starting with the left most non-terminal up to the right most non-terminal the following is done:

   (a) if the current symbol is a terminal then depending on the type of the previous symbol we try to find the direct match in the input portion. If the last symbol was a non-terminal we try to find the closest matching terminal, otherwise the next terminal in the input must match. If the latter is not satisfied then the function exits returning FALSE.

   (b) if the current symbol is a non-terminal then the left-input pointer *il* is increased by the minimal length of derivable string of that non-terminal.

5. After analyzing all the symbols the condition is $il \leq iu$ checked. When satisfied we know that there exists a proper match.

4.4.3 *Quick* checks

The knowledge obtained from the grammar in the instance of $\mathcal{PRE}$, $\mathcal{SUF}$ and $\mathcal{EXC}$ sets, allows for further elimination of goals. Taking into account the non-terminal standing on the left-hand side N of the rule and the input fragment it can be verified whether the input fragment

1. begins with one of the prefixes found in $\mathcal{PRE}(N)$
2. ends with one of the suffixes found in $\mathcal{SUF}(N)$
3. does not contain a substring found in $\mathcal{EXC}(N)$

If one of the above checks is not satisfied the goal has no chance to succeed. The following pseudo-code implements the above checks.

```
canNTDerive(g)
▷ The PREFIXES, SUFFIXES and SUBSTRINGS are sets of all
▷ prefixes, suffixes and substrings of the given input
▷ fragment (pos[g],len[g]), respectively
if ($len[g] >= MIN(lhs[g])$) and ($len[g] <= MAX(lhs[g])$)
   then if $l = 0$
           then return TRUE
        if $PREFIXES(pos[g], len[g]) \cap \mathcal{PRE}(lhs[g]) = \emptyset$
           then return FALSE
        if $SUFFIXES(pos[g], len[g]) \cap \mathcal{SUF}(lhs[g]) = \emptyset$
           then return FALSE
        if $SUBSTRINGS(pos[g], len[g]) \cap \mathcal{EXC}(lhs[g]) \neq \emptyset$
           then return FALSE
   else
        return FALSE
return TRUE
```

4.4.4 More efficient TryAllLengthsFor

Let's recall the behavior of TryAllLengthsFor. TryAllLengthsFor is invoked by DoTopOfStack when during the analysis of the current right-hand side a non-terminal N is encountered and needs to be *expanded*. The procedure attempts to derive from N all possible fragments of the remaining input portion. This is done by calling TryAllLengthsFor for all possible lengths, starting from 0 up to the length of the remaining input portion. Such approach obviously causes the procedure to be highly ineffective. It is sufficient to start generating possible lengths staring from $MIN(N)$ up to the minimum of either maximal *span* of a derivation for N or maximal length of the remaining input fragment. Taking the minimum does not permit to assign to N symbols of the input sentence that are beyond the given input fragment.

The maximal span for any given N non-terminal is computed for the given goal. Let $\mathbb{G} = (L \rightarrow \alpha, p, l)$ denote a goal where $\alpha = S_1 S_2 ... S_k$. Let m be the minimal length of the derivable string from α. Let's assume that m is a finite number[5]. Let O_i be the minimal length of the input fragment that must be assigned to the sequence of symbols $S_1 ... S_i$, in order to satisfy their minimal length requirement. For every terminal symbol S_j the $MIN(S_j) = 1$.

$$O_i = \sum_{k=1}^{i} MIN(S_k)$$

Let R_i denotes the maximal length of the input fragment that the symbol S_i may be able to derive, taking into account the minimal derivation length that can be obtained from symbols $S_{i+1} ... S_k$.

$$R_i = l - O_i$$

Based on the above formulas the maximal span for each symbol in α can be calculated as follows:

$$U_i(S_i) = \begin{cases} 1 & S_i \in V_T \\ \min\{R_i - (m - Q_i) + MIN(S_i), MAX(S_i)\} & S_i \in V_N \end{cases}$$

The following pseudo-code presents computation of the maximal spans for any given goal.

[5]In practice this requirement holds as long as the *necessary condition* is satisfied. Since TryAllLengthsFor is called after a goal has been accepted, in the optimized version of the parser it will also mean that is has satisfied the *necessary condition*.

```
ComputeMaxSpans($g$)
1  $R \leftarrow rhs[g]$
2  if $R = \epsilon$
3      then $l[0] \leftarrow 0$
4           return $l$
5  $Q_i \leftarrow 0$
6  $R_i \leftarrow l - Q_i$
7  $m \leftarrow \sum_{k=1}^{length[R]} MIN(R[k])$
8  for $i = 0$ to $length[R] - 1$
9       do
10          if $R[i] \in V_T$
11             then $Q_i \leftarrow Q_i + 1$
12                  $R_i \leftarrow R_i - 1$
13                  $l[i] \leftarrow 1$
14             else $Q_i \leftarrow Q_i + MIN(R[i])$
15                  $R_i \leftarrow R_i - MIN(R[i])$
16                  $l[i] \leftarrow \min(R_i - (M - Q_i) + MIN(R[i]), MAX(R[i]))$
17 return $l$
```

The old version of TryAllLengthsFor has now its optimized replacement, TrySelectedLengthsFor. The procedure makes use of the computed maximal spans. It is called from DoTopOfStack procedure and therefore the arguments passed are:

1. N - i^{th} symbol (non-terminal) of currently considered right-hand side
2. p - position in the input sentence
3. len - length of the input starting at p
4. $span$ - maximal span computed for the i^{th} symbol

```
TrySelectedLengthsFor($L, p, len, span$)
1  $m \leftarrow len$
2  if $span < m$
3     then $m \leftarrow span$
4  for $l \leftarrow MIN[N]$ to $m$
5       do TryAllRulesFor($N, p, l$)
```

4.4.5 Incorporation of the look-ahead mechanism into the parser

The look-ahead mechanism is implemented by modifying the following portions in the generic Unger's parser scheme presented in chapter 3.

1. TryRule shall be replaced by it's new version equipped with a goal analyzer which comprises of two functions:
   (a) MatchExists which conducts the *necessary condition* and which determines if the *proper match* exists, as discussed.
   (b) canNTDerive performs *quick* checks.

2. TryAllLengthsFor shall be replaced by its new version TrySelectedLengthsFor which utilizes the knowledge of the minimal length of derivable terminal strings and maximal span of N-strings.

The new TryRule

The new version of the TryRule differs from the original one at the entry point. Before the given goal can be examined the aforementioned checks are conducted.

```
TryRule($L \to R, p, l$)
1  $g \leftarrow (L \to rhs, p, l)$            ▷ Establish a new goal
2  if MatchExists($g$)
3     then if not canNTDerive($g$)
4             then return
5     else
6          return
7
8  if not IsToBeAvoided($g$)
9     then
10         if not KnownGoalSucceeds($g$)
11            then
12                 $spans \leftarrow$ ComputeMaxSpans($g$)
13                 Push($\Omega, (g, -1, 0, spans)$)
14                 StartNewKnownGoal($g$)
15                 Push($D, L \to rhs$)
16                 DoTopOfStack
17                 Pop(D)
18                 Pop($\Omega$)
```

Replacement of the TryAllLengthsFor

In the base-line version of the Unger's parser presented in chapter 3 TryAllLenghtsFor procedure was called in DoTopOfStack when a non-terminal symbol was encountered while analyzing the right-hand side of current goal. The new version, TrySelectedLengthsFor, requires an additional argument to be passed. The argument is supposed to be the value of the maximal span of the current non-terminal. This value is obtained from the element placed on the top of Ω stack. The unused before field *spans* is now assigned an array of maximal spans for the associated right-hand side. The array is computed in the new TryRule procedure in line 12, and placed on the parser's stack in the next line.

The modification to the DoTopOfStack is done by replacing the line 17 with the following lines:

```
$spanarr \leftarrow spans[s]$
$max \leftarrow spanarr[rhsPos[s] + 1]$
TrySelectedLengthsFor($C, pos[g] + inRec[s], len[g] - inRec[s], max$)
```

4.5 Summary

The effect of incorporating the aforementioned optimizations into Unger's parser is best seen through comparison tests. These are presented in chapter 6. It is necessary to point out that increase comes from a higher memory consumption needed to store $\mathcal{PRE}$, $\mathcal{SUF}$ and $\mathcal{EXC}$ sets, tables for minimal and maximal lengths of derivable strings and the known-parsings table. Introduction of the optimizations itself introduces certain performance overhead. Therefore much care and effort was put to come up with an efficient implementation.

Chapter 5

Implementation

The algorithms discussed in the previous chapters have been implemented. Since in this study the core requirement was the ability to prototype the presented ideas in a simple and fast manner Java was chosen as a programming languange. There is a number of reasons for which Java is suitable for experimenting:

1. Java is object oriented - this feature allows for a convenient design of the program's skeleton. Implementation of the selected components can be easily abstracted by the use of interfaces, allowing for having a number of test implementations easily portable. The mechanism of inheritance allows for simple building of extensions to existing classes.

2. The Java API offers a large number of data structures, ranging from essential ones (Stack, Vector, List) to more advanced once (Black-Red trees). Most of the structures are accompanied by implementations of various algorithms for manipulating these structures.

3. *Garbage collection* - reduces the possibility of memory leaks since memory is freed as needed, object that are being referenced cannot be deleted - this most often is an element of memory corruption in programs written in languages like C or C++.

4. Lack of pointers lowers the risk of manipulating unallocated memory blocks which usually causes difficult to detect errors.

5. Java is platform independent - enables simple migration from one operating system to another.

In order to provide a reliable experimental environment in which errors related to algorithmic flaws could be easily detected, each implemented component was subject to testing. For that purpose JUnit was employed. JUnit provides a simple to use and functional interface for coordinating class tests.

5.1 Overview

The implementation served primarily as a base for conducting experiments with variants of the Unger's algorithm on various grammars ranging from some artificial ones, created for simple tests and benchmarking, to a real-life grammar for Cobol.

The design of the platform enables simple prototyping of new variants of the Unger's parser. Many of the designed components can be easily extended for the future experimentation. The framework has been optimized for performance after many of the bottlenecks time critical aspects have been disclosed during the experiments.

5.1.1 Features of the framework

The core features of the framework enable simple preparation of tests and gathering of statistical information. Below is a list of the core features of the framework.

1. The ability to accept any context-free grammar provided in BNF format (BNF subset as discussed in section 5.2.1)
2. Generation of all derivations found during parsing for any given input sentence.
   (a) Parse trees obtained *correspond* to BNF syntax with iterative constructs
   (b) Parse trees are generated as *.dot* files (acceptable by Graphviz[1] tools)
3. Statistical information
   (a) Parsing time
   (b) Number of rules used for finding derivations (Unger's algorithm specific)
   (c) Number of derivations found
4. The ability to define and conduct performance tests in which variants of the Unger's parser can be compared in terms of
   - number of rules used to find derivations
   - parsing time

5.2 Parser's input/output

5.2.1 Input

Input files

The input sentences are provided in text files. The input is subject to lexical analysis. This portion of the pre-processing phase is done with the use of *javacc* lexical tokenizer.

[1]GraphViz (http://www.graphviz.org) A set of tools for representing structural information as diagrams of abstract graphs and networks.

Grammar definition file

BNF is a well known grammar formalism. The parser's input is restricted to the following variant of BNF:

- As atomic constructs there is a finite set of *terminals* (such as keywords), and a finite set of *non-terminals*
- As operators we have * , + and ? as usual, for the time being we allow these operators only on atomic constructs.
- An alternative is constructed from atomic-constructs, with or without operators, and sequential composition.
- A non-terminal N is defined by a finite collection of alternatives, this collection may be empty.

5.2.2 Output

- Parse-forest
- Statistical information on parsing
  - Number of parse-trees found
  - Time needed to find the parse-forest
  - Number of rules tried to find the parse-forest

Chapter 6

Experiments

The presented Unger's parser was implemented in three different versions:

1. (Generic) Unger's parser as presented in chapter 3. This variant of the Unger's parser requires exponential time for parsings. For that reason it was only used in a simple experiment that enabled to measure the actual exponential time parsing on a sample grammar.
2. (Basic) Unger's parser with incorporated the *known-parsings table.* Since this version enables parsing in polynomial time it is considered the base-line implementation for any further experiments.
3. (Optimized) Base-line Unger's parser with implemented optimizations described in chapter 4

A number of comparison experiments was performed with the above parsers. All timings were conducted on a 1,5Mhz Pentium M with 512MB of RAM and running Windows XP operating system. The Java virtual machine used was 1.5. The comparison results present two values as a performance measure: the time needed to complete the parsing and the number of rules tried during parsing. The latter refers to the number of rules examined during parsing as a result of accepted goals. Such measure value is considered to be objective when comparing performance of various version of the Unger's parser. All the presented results are the median of five trials.

6.1 Parsing in a polynomial time

6.1.1 Known-parsings table

As claimed in chapter 4 the incorporation of the known-parsings mechanism into the Unger's parser introduces a significant performance improvement by enabling parsing in polynomial time. The following experiment attempted to measure "real" difference between "exponential" parsing and "polynomial" parsings. The other interesting factor was the number of rules examined during the parsing process in both cases. In order to conduct the experiment in a reasonable amount of time, the following (small) grammar has been provided as the parser's input:

$$E \rightarrow E + F \mid F$$
$$F \rightarrow integer$$

Figure 6.1: Simple arithmetic expressions - unambiguous grammar

The parsing was coordinated on a serious of sentences: $2(+2)^n$. The results have been gathered in the following table:

	Basic		**Generic**	
Tokens	**Rules**	**Time (s)**	**Rules**	**Time (s)**
1	6	0.0000	12	0.0000
3	14	0.0000	57	0.0019
7	36	0.0019	975	0.01599
23	204	0.0240	64313648	75.5746
95	2544	0.06220	-	-
191	9696	0.4686	-	-
383	37824	3.8074	-	-
553	78114	11.094	-	-
819	170150	35.1726	-	-
999	252500	63.8036	-	-

Table 6.1: Basic vs. Generic versions of the Unger's parser

The parsing process conducted by the generic Unger's parser was stopped for the input sentences of number of tokens higher then 23. The time needed to complete the next given input sentence (95 tokens) was exceeding the *reasonable* expected time. The Unger's parser version equipped with the known-parsings table managed to deliver solutions in a time that allowed to coordinate the tests on a entire set of the input sentences.

6.1.2 Basic vs. Optimized

In this experiment the basic variant of the parser was compared with the optimized one. The goal was to observe how the added optimizations affect the number of rules tried during parsing and how it is reflected in time needed to complete parsing. These performance tests were coordinated on the input grammar presented in figure 6.1 and the input sentences of the form $2(+2)^n$.

	Basic		**Optimized**	
Tokens	**Rules**	**Time (s)**	**Rules**	**Time (s)**
1	6	0.0079	2	0.0000
3	14	0.0000	4	0.0019
7	36	0.0039	8	0.0019
23	204	0.0726	24	0.0159
95	2544	0.0782	96	0.0460
191	9696	0.4946	192	0.0459
383	37824	3.9878	384	0.2442
553	78114	11.4966	554	0.6527
819	170150	37.1714	820	1.9427
999	252500	70.6338	1000	3.6170

Table 6.2: Basic and Optimized variants of the Unger's parser

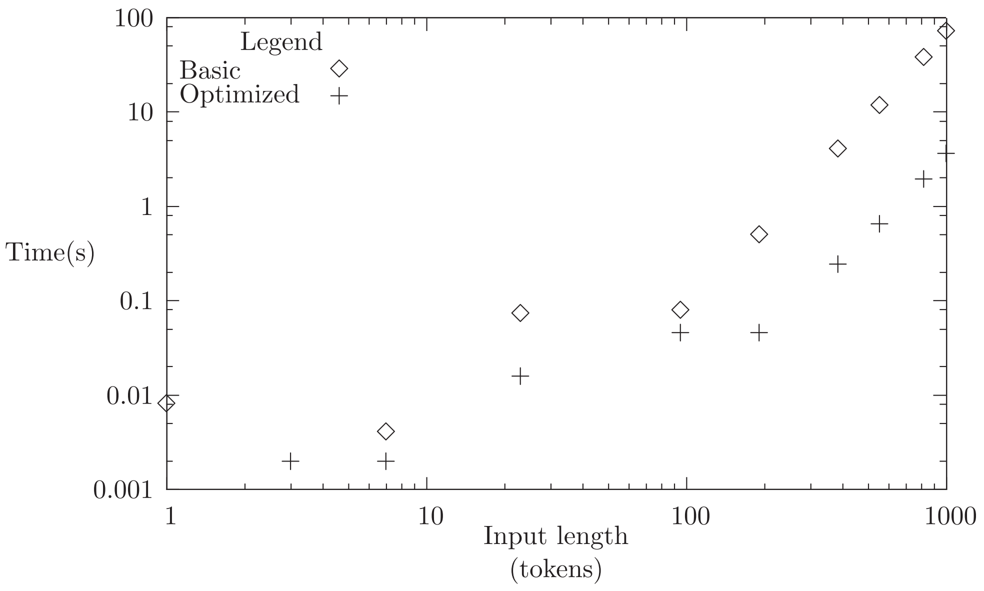


Figure 6.2: Basic and Optimized variants of the Unger's parser

6.2 Case study: COBOL

The computer systems developed years ago are still in use and are constantly subject of the evolution. The source code for legacy software was in most cases written in COBOL. COBOL unlike other programming languages has introduces many dialects and extensions over the years. All these modifications make COBOL difficult to parse.

6.2.1 COBOL's history

In the late 50's the need for a business oriented programming language was strong. In response to that COBOL (Common Business Oriented Language) was developed under supervision of the U.S. Department of Defense in cooperation with computer manufactures, users and universities. The language was supposed to address business oriented problems, be machine independent and ready for continuous changes and development. The initial specifications for COBOL were presented in a report of the executive committee of CODASYL[1] in April of 1960. Since 1960, COBOL has undergone considerable updates and improvements. It had emerged as the leading data processing language in the business world.

Throughout the history various implementations of COBOL appeared. Most deviations were intended to take advantage of hardware or environmental features which were not defined in the basic definition. There were attempts to establish COBOL standard. In 1968 American National Standards Institute (ANSI) as an attempt to overcome incompatibilities of different COBOL versions developed a standard form of a language known as (ANS) COBOL[2]. In 1974 and 1985 ANSI published revisions to the ANS COBOL. Currently the official standard for COBOL is being worked out by the ISO and ANSI committees. The latest standard was published in 2002.

6.2.2 VS COBOL II grammar

COBOL's initial definition was done through a grammar which was informal and descriptive, and was never intended to be generative [9]. The developed context-free parser was tested on a real-life VS COBOL II. As described by R. Lämmel and C. Verhoef the grammar was obtained in a semi-automatic process [13], from IBM's VS COBOL II Reference Summary, and tested for large test sets of COBOL code: 8 test sets from different countries with 1.819.765 LOC (before preprocessing: 1.631.825). The grammar was provided in a BNF format.

A short summary of the grammar used for experiments is presented in the following table:

Number of rules	538
Number of keywords	345
Number of lexical sorts	11

[1]CODASYL - Conference on Data System Languages. The organization was founded in 1957 by the U.S. Department of Defense with the mission to develop computer programming languages

[2](ANS) COBOL stands for American National Standard COBOL

6.2.3 Parsing Cobol

Two tests were coordinated. In the first one the provided grammar was applied to a set of small Cobol programs (number of lexical tokens did not exceed 100) and was tested on three different variants of the Unger's parser. The "basic", "optimized-nq" (variant with *Quick* checks optimization was disabled), and "optimized". The point of that experiment was to compare how added optimizations affect parsing time. The size of the input programs was chosen to be small to allow parsing in reasonable time by the "basic" version of the Unger's parser. Results are presented in figure 6.3 and 6.4.

The second series of tests involved only the optimized variant of the Unger's parser. The collection of input files was larger. The results are presented in figure 6.5 and 6.6.

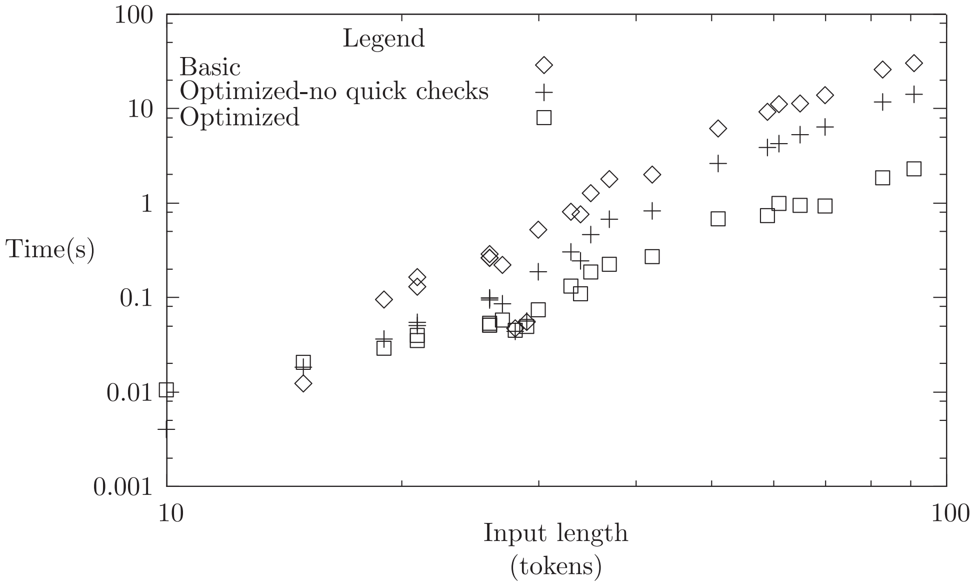


Figure 6.3: Comparison of Basic, Optimized-nq, and Optimized variants

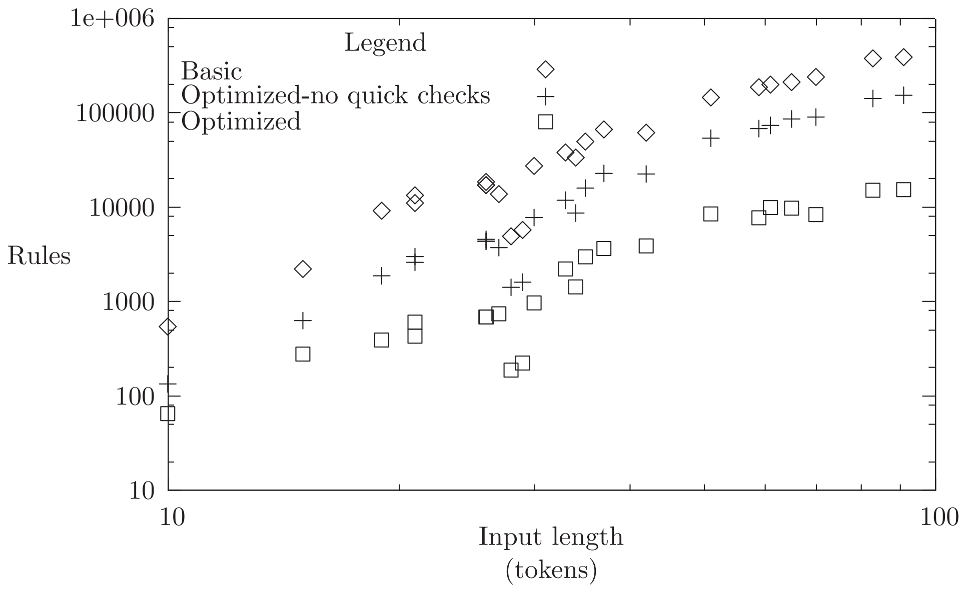


Figure 6.4: Comparison of Basic, Optimized-nq, and Optimized variants - rules used

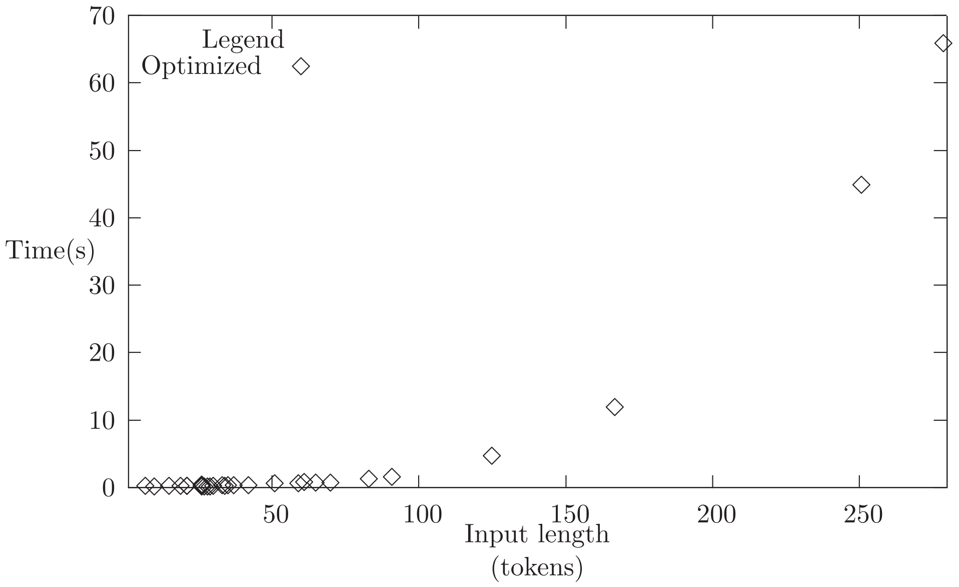


Figure 6.5: Optimized variant

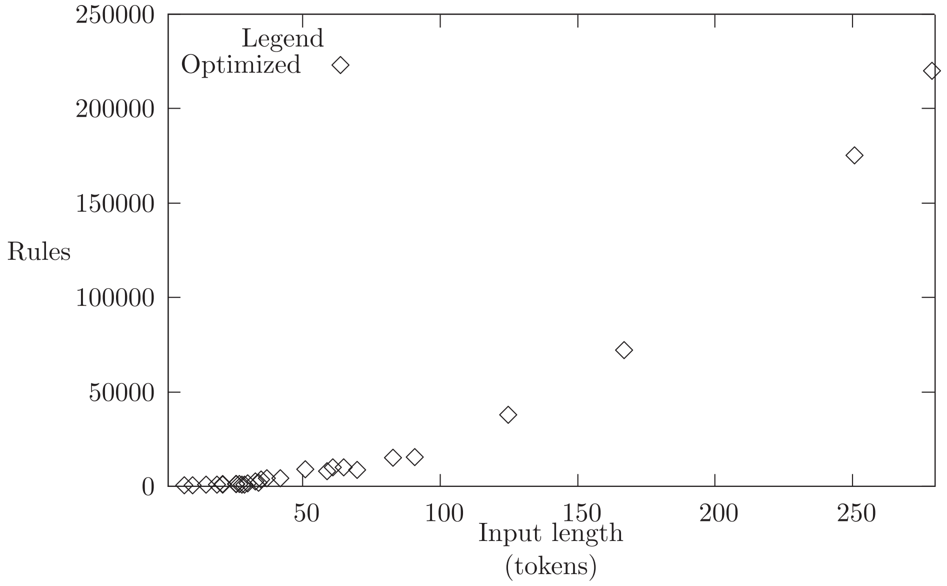


Figure 6.6: Optimized variant - rules used

Chapter 7

Related work

7.1 Performance

Unger's parser equipped with the known parsings table mechanism is capable of parsing in a polynomial [9] time, $O(n^{k+1})$, n being the length of the input sentence in tokens and k the maximum number of non-terminal symbols in any right-hand side of the production rules. In comparison to the general bottom-up technique, GLR, which in the worst case takes $O(n^{p+1})$ time [15], where p is the length of the longest right-hand side, Unger's method may appear more effective. GLR is however used in practice while this cannot be expected from the Unger's algorithm. GLR as an extension to the standard LR parsing is driven by a parse table. Much of the work needed to construct such table is done before the actual parsing process. The need for the use of multiple stacks when resolving reduce/reduce or shift/reduce conflicts introduces the non-linear time requirement. When considering the use of table for guiding parsing as well as complexity of the operations performed, GLR parsing performs better in practice then Unger's parser. Unger's algorithm coordinates a *pure* search process. Operations involved are more time consuming causing the algorithm to be less efficient in practice.

A suggested approach in seeking performance boost through conversion of the grammar[1] to Chomsky normal form, to limit the maximal length of the right-hand sides, should rather be a forbidden one since it affects the grammar.

Unger presents [23] observations from comparing his parser to a bottom-up parser implemented in RCA SNOBOL[2] as follows: "For certain sentences of certain grammars, the global[3] parser was faster then this other parser by factors as large as 50. In a few cases, the bottom-to-top parser was faster." No firm conclusion can be reached as to how the two parsers compare.

[1]A grammar can be converted to its equivalent Chomsky normal form presuming that no ϵ-rule exists in the productions set.

[2]SNOBOL language was developed by David J. Farber, Ralph E. Griswold and Ivan P. Polonsky at Bell Labs in 1962. The main language defect was the lack of built-in functions. A new implementation with built-ins produced SNOBOL2 (1964). Programmer defined functions were added to make SNOBOL3 (1965). None of these languages have anything to do COBOL.

[3]That's another name for the Unger's parser.

7.2 Disambiguation

Parsing all context-free grammars is closely paired with the issue of disambiguation. In many cases a list of potential derivations of a sentence is not of a practical use. The Unger's parser explores all different ambiguities and delivers a forest of parse trees (parse-forest).

In SDF and SGLR [5] the parsing process also explores all different ambiguities and in the post-processing phase the resulting parse-forest is subject to disambiguation filters. The intention is to obtain only one parse-tree. This approach, easily adaptable in the Unger's parser, may however lead to non-intended results as reported in [5] and [10].

There is a believe that filtering parse-forests complicates the parsing process far too much and that ambiguities have to be avoided as soon as possible. The term *disambiguation prevention* introduced in [11] stands for resolving ambiguities as they are detected. The two following features are claimed to be sufficient to resolve potential ambiguities for the real-life grammars:

1. innermost binding of nested constructs
2. ordering of alternatives for right-hand sides

The above two approach would need to be included into the design of the Unger's parser in order to filter out the generated parse-trees on-the-fly as they appear during parsing.

7.3 Error handling

Error handling is one of the most important aspects when building a parser, especially when the parser is to be applied for real-life programming languages. Syntax errors in programs are common and the need to locate them is important.

Recalling the behavior of the Unger's algorithm. The method tries to find such partitioning of the input sentence that can *match* one of the right-hand sides of the start symbol. An error will occur when such partitioning cannot be found. During the parsing process input fragments of the given partition are partitioned recursively. An error can occur in any considered sub-problem. It is possible to detect syntax errors but it is not possible to determine how many there are and where they occur. A.Grune and J.Ceriel in [9] comment: *"In a way, Unger's method is too well prepared for dealing with failures, because it expects any partition to fail"*. Unger's parser enables modifications in order to be able to detect syntax errors in a useful manner.

As a non-directional parser Unger's algorithm needs to consider a global error handling methods. These methods [9] take into consideration the entire input sentence (global context), they are very effective however also very time consuming. Since general context-free parsing requires at least cubic time these methods are deemed acceptable.

7.4 Parser generation

None of the general parsers [9] identifies with certainty a part of the parse tree before the whole parse tree is completed. This immediately implies that if semantic actions are connected to the grammar rules, then none of them can be performed until the whole parse if finished.

Chapter 8

Conclusions

The fact that the current parsing techniques used for a large scale software analysis expose certain limitations states a clear need for a parsing technique that is either able to overcome them or enable experiments with solutions to these limitations. In this study a top-down approach to parsing general context-free grammars using Unger's method was investigated.

8.1 Key points

The Unger's algorithm has been tested against a set of artificial grammars and the full (ambiguous) context-free VS COBOL II grammar. Unger's parser indeed supports full CF (with the iterative constructs from EBNF), however even with the incorporated optimizations presented in this thesis, it is not appropriate for parsing real-life grammars like COBOL, as the current implementation fails to parse programs of, for example, 1.000 lines of code.

It has been shown that elimination of certain *fruitless* goals from the search process by properly utilized grammar derived knowledge, in the form of the designed look-ahead mechanism, increased the speed of parsing on the average by a factor of 12, on the full COBOL grammar. This fact claims that even though the parser's $O(n^{k+1})$ dependency remains, the practical aspect of the optimizations is noticeable.

The optimization-specific experiments have shown that inclusion of the *quick* checks into the look-ahead results in a higher rejection of the rules examined by a factor of 8.5. It can be expected that further goal elimination heuristics decrease the number of rules examined in the search process.

The results obtained from the experiment conducted on the grammar presented in figure 6.1 show parsing can be coordinated with a linear dependency with respect to the number of rules examined during the search process. Such dependency does not hold however for time. What should be concluded is that process of elimination of the goals from the search process is time-consuming and therefore the optimizations design and implementation should be done with great care.

8.2 Future work

The fact that the Unger's top-down algorithm remains transparent delivers a number of avenues for experimentation with solutions to the limitations of the current parsing techniques as listed in [12]:

- case insensitivety of the tools - this issue can be resolved by a properly constructed lexical analyzer to be included with the Unger's parser. The current implementation was capable of case sensitive recognition of the input tokens.
- addition of disambiguation control information to BNF grammar description - the current implementation can be extended to include ordering for alternatives of the right-hand sides, priorities between production rules, or innermost-binding attributes. The nature of the Unger's parsing mechanism gives a large freedom in choosing production rules at parse-tree construction time. The disambiguation control information can be used to guide the rules selection process.
- disambiguation - the fact that filtering the obtained parse-forests in the post-processing phase of the parsing does not deliver desired results leans towards experimentation with resolving the ambiguities at the parsing level, as soon as they are detected. The fact that the ambiguities often arise on the higher levels in the parse-tree construction puts top-down approach to parsing a better suited for their earlier detection and reaction.

Enabling the Unger's parser for usage on the industrial level requires additional efforts to increase the parsing speed. Following the Elkhound parser idea of GLR-LR hybrid, the Unger's parser could be equipped with a build-in deterministic top-down parser, say LL(1). Such Unger-LL(1) hybrid could try to parse deterministic fragments of the input sentence using a linear top-down algorithm. Unger's method appears to be well suited for that due to its general idea of partitioning the input and trying to derive fragments independently. The resulting partial derivations could then be appended to the constructed derivation of the input in a similar manner as the known-parsings table mechanism tries to reuse the known partial parse-trees. Keeping in mind that in practice the grammars for programming languages are not "highly" ambiguous such approach could introduce a significant increase in parsing speed.

Appendix A

Parsing process

A.1 Unger's parser - operational stack

The following listing represents changes of the Unger's parser operational stack during the parsing process for the provided grammar and input sentence. Each element of the stack: (G[rule,pos,len],[rhsPos,inRec]), describes the goal *G* and the parser's operational variables, *rhsPos* and *inRec* (as described in section 3.2). The content of the stack (prefixed by 'Stack:') is captured after the given goal is accepted for processing and pushed on the operational stack (in TryRule procedure). The lines prefixed by 'Cut-off:' represent the goals that get rejected by the test IsToBeAvoided in TryRule (the test assures that the algorithm will not fall into an infinite loop). The lines prefixed by 'Found derivation:' represent the content of the derivations stack *D* after a parse tree is found.

```
Grammar:
S: L S D  |  ;
L: ;
D: "d" ;

Input:
 d d

Unger's parser: operational stack content
=======================================================================================================
Stack: (G[S->LSD ;,1,2],[-1,0])
Stack: (G[L-> eps ;,1,0],[-1,0])(G[S->LSD ;,1,2],[-1,0])
Stack: (G[S->LSD ;,1,0],[-1,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[L-> eps ;,1,0],[-1,0])(G[S->LSD ;,1,0],[-1,0])(G[S->LSD ;,1,2],[0,0])
Cut-off: G[S->LSD ;,1,0]
Stack: (G[S-> eps ;,1,0],[-1,0])(G[S->LSD ;,1,0],[0,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[D->"d" ;,1,0],[-1,0])(G[S->LSD ;,1,0],[1,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[S-> eps ;,1,0],[-1,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[D->"d" ;,1,0],[-1,0])(G[S->LSD ;,1,2],[1,0])
Stack: (G[D->"d" ;,1,1],[-1,0])(G[S->LSD ;,1,2],[1,0])
Stack: (G[D->"d" ;,1,2],[-1,0])(G[S->LSD ;,1,2],[1,0])
Stack: (G[S->LSD ;,1,1],[-1,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[L-> eps ;,1,0],[-1,0])(G[S->LSD ;,1,1],[-1,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[S->LSD ;,1,0],[-1,0])(G[S->LSD ;,1,1],[0,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[L-> eps ;,1,0],[-1,0])(G[S->LSD ;,1,0],[-1,0])(G[S->LSD ;,1,1],[0,0])(G[S->LSD ;,1,2],[0,0])
Cut-off: G[S->LSD ;,1,0]
Stack: (G[S-> eps ;,1,0],[-1,0])(G[S->LSD ;,1,0],[0,0])(G[S->LSD ;,1,1],[0,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[D->"d" ;,1,0],[-1,0])(G[S->LSD ;,1,0],[1,0])(G[S->LSD ;,1,1],[0,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[S-> eps ;,1,0],[-1,0])(G[S->LSD ;,1,1],[0,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[D->"d" ;,1,0],[-1,0])(G[S->LSD ;,1,1],[1,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[D->"d" ;,1,1],[-1,0])(G[S->LSD ;,1,1],[1,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[D->"d" ;,2,0],[-1,0])(G[S->LSD ;,1,2],[1,1])
Stack: (G[D->"d" ;,2,1],[-1,0])(G[S->LSD ;,1,2],[1,1])
```

```
Found derivation : LMD: S->LSD ;=>L-> eps ;=>S->LSD ;=>L-> eps ;=>S-> eps ;=>D->"d" ;=>D->"d" ;=>
Cut-off: G[S->LSD ;,1,1]
Stack: (G[S-> eps ;,1,1],[-1,0])(G[S->LSD ;,1,1],[0,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[L-> eps ;,1,1],[-1,0])(G[S->LSD ;,1,1],[-1,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[S-> eps ;,1,1],[-1,0])(G[S->LSD ;,1,2],[0,0])
Cut-off: G[S->LSD ;,1,2]
Stack: (G[S-> eps ;,1,2],[-1,0])(G[S->LSD ;,1,2],[0,0])
Stack: (G[L-> eps ;,1,1],[-1,0])(G[S->LSD ;,1,2],[-1,0])
Stack: (G[L-> eps ;,1,2],[-1,0])(G[S->LSD ;,1,2],[-1,0])
Stack: (G[S-> eps ;,1,2],[-1,0])
=================================================================================================
```

A.2 Unger's parser - known parsings table

The following listing represents changes of the Unger's parser operational stack during the parsing process for the provided grammar and input sentence. In this case the parser was equipped with the known parsings table, constructed on-the-fly. The content of the known parsings table was captured after the parsing process finished. Each rule r is associated with the input fragments determined by the pairs (p, l) (p being the absolute position of the input token in the input sentence, l being the length of the fragment) for which goals $G = (r, p, l)$ have been examined. Each pair (p, l) is assigned a set of parse trees ('Parsings=') found for the goal G. These *known parsings* are reused during the parsing process when the same goals are established.

```
Grammar:
E: E "+" E | int ;

Input:
2 + 2 + 2

Unger's parser: operational stack content
=====================================================================================================
Stack: (G[E->E"+"E ;,1,5],[-1,0])
Stack: (G[E->E"+"E ;,1,0],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
Cut-off: G[E->E"+"E ;,1,0]
Stack: (G[E->int ;,1,0],[-1,0])(G[E->E"+"E ;,1,0],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
Stack: (G[E->E"+"E ;,1,1],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
Cut-off: G[E->E"+"E ;,1,1]
Stack: (G[E->int ;,1,1],[-1,0])(G[E->E"+"E ;,1,1],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
Stack: (G[E->E"+"E ;,3,0],[-1,0])(G[E->E"+"E ;,1,5],[1,2])
Cut-off: G[E->E"+"E ;,3,0]
Stack: (G[E->int ;,3,0],[-1,0])(G[E->E"+"E ;,3,0],[-1,0])(G[E->E"+"E ;,1,5],[1,2])
Stack: (G[E->E"+"E ;,3,1],[-1,0])(G[E->E"+"E ;,1,5],[1,2])
Cut-off: G[E->E"+"E ;,3,1]
Stack: (G[E->int ;,3,1],[-1,0])(G[E->E"+"E ;,3,1],[-1,0])(G[E->E"+"E ;,1,5],[1,2])
Stack: (G[E->E"+"E ;,3,2],[-1,0])(G[E->E"+"E ;,1,5],[1,2])
Stack: (G[E->E"+"E ;,5,0],[-1,0])(G[E->E"+"E ;,3,2],[1,2])(G[E->E"+"E ;,1,5],[1,2])
Cut-off: G[E->E"+"E ;,5,0]
Stack: (G[E->int ;,5,0],[-1,0])(G[E->E"+"E ;,5,0],[-1,0])(G[E->E"+"E ;,3,2],[1,2])(G[E->E"+"E ;,1,5],[1,2])
Cut-off: G[E->E"+"E ;,3,2]
Stack: (G[E->int ;,3,2],[-1,0])(G[E->E"+"E ;,3,2],[-1,0])(G[E->E"+"E ;,1,5],[1,2])
Stack: (G[E->E"+"E ;,3,3],[-1,0])(G[E->E"+"E ;,1,5],[1,2])
Stack: (G[E->E"+"E ;,5,1],[-1,0])(G[E->E"+"E ;,3,3],[1,2])(G[E->E"+"E ;,1,5],[1,2])
Cut-off: G[E->E"+"E ;,5,1]
Stack: (G[E->int ;,5,1],[-1,0])(G[E->E"+"E ;,5,1],[-1,0])(G[E->E"+"E ;,3,3],[1,2])(G[E->E"+"E ;,1,5],[1,2])
Found derivation : LMD: E->E"+"E ;=>E->int ;=>E->E"+"E ;=>E->int ;=>E->int ;=>
Cut-off: G[E->E"+"E ;,3,3]
Stack: (G[E->int ;,3,3],[-1,0])(G[E->E"+"E ;,3,3],[-1,0])(G[E->E"+"E ;,1,5],[1,2])
Stack: (G[E->E"+"E ;,1,2],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
Cut-off: G[E->E"+"E ;,1,2]
Stack: (G[E->int ;,1,2],[-1,0])(G[E->E"+"E ;,1,2],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
Stack: (G[E->E"+"E ;,1,3],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
Found derivation : LMD: E->E"+"E ;=>E->E"+"E ;=>E->int ;=>E->int ;=>E->int ;=>
Cut-off: G[E->E"+"E ;,1,3]
Stack: (G[E->int ;,1,3],[-1,0])(G[E->E"+"E ;,1,3],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
Stack: (G[E->E"+"E ;,1,4],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
Cut-off: G[E->E"+"E ;,1,4]
Stack: (G[E->int ;,1,4],[-1,0])(G[E->E"+"E ;,1,4],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
```

```
Cut-off: G[E->E"+"E ;,1,5]
Stack: (G[E->int ;,1,5],[-1,0])(G[E->E"+"E ;,1,5],[-1,0])
==========================================================================================================


Unger's parser: Known Parsings Table
============================================================================
E: E "+" E
{ (5,1)=Parsings  NONE
, (5,0)=Parsings  NONE
, (3,3)=Parsings  [E->E"+"E ;, E->int ;, E->int ;]
, (3,2)=Parsings  NONE
, (3,1)=Parsings  NONE
, (3,0)=Parsings  NONE
, (1,5)=Parsings  [E->E"+"E ;, E->int ;, E->E"+"E ;, E->int ;, E->int ;]
                  [E->E"+"E ;, E->E"+"E ;, E->int ;, E->int ;, E->int ;]
, (1,4)=Parsings  NONE
, (1,3)=Parsings  [E->E"+"E ;, E->int ;, E->int ;]
, (1,2)=Parsings  NONE
, (1,1)=Parsings  NONE
, (1,0)=Parsings  NONE
}


E: int
{ (5,1)=Parsings  [E->int ;]
, (5,0)=Parsings  NONE
, (3,3)=Parsings  NONE
, (3,2)=Parsings  NONE
, (3,1)=Parsings  [E->int ;]
, (3,0)=Parsings  NONE
, (1,5)=Parsings  NONE
, (1,4)=Parsings  NONE
, (1,3)=Parsings  NONE
, (1,2)=Parsings  NONE
, (1,1)=Parsings  [E->int ;]
, (1,0)=Parsings  NONE
}
============================================================================
```

Appendix B

Implementation

B.1 Known parsings table Java implementation

```
/*
 * Created on 2005-07-12
 *
 */
package org.vu.cobol.parser.opts.parsetables;

/**
 * KPTable
 *
 * @author Lukasz Kwiatkowski (lkwiatk@few.vu.nl)
 *
 */
import java.util.Collection;
import java.util.Enumeration;
import java.util.SortedMap;
import java.util.TreeMap;
import java.util.Vector;

import org.vu.cobol.grammar.SingleRule;
import org.vu.cobol.parser.core.Derivation;
import org.vu.cobol.parser.core.StackElem;
import org.vu.cobol.parser.opts.KnownParsingsTable;
import org.vu.cobol.parser.opts.UngerParserCommon;

/**
 * KnownParsings - implementation of the
 * KnownParsingsTable interface, provides fast and
 * effective way of storing, retrieving and searching
 * through the recorded known parsings
 *
 * @author Lukasz Kwiatkowski (lkwiatk@few.vu.nl)
 */
public class KPTable implements KnownParsingsTable{

    /**
     * KnownGoal - represents collection of known
     * parsings for the given goal
     *
     * @author Lukasz Kwiatkowski (lkwiatk@few.vu.nl)
     */
    private class KnownGoal{
        public int              startParsing;
        public Vector           knownParsings;

        public KnownGoal(){
            startParsing = parser.getRulesStack().size();
            knownParsings = new Vector();
        }
    }

    private SortedMap[]          goals;
    private UngerParserCommon    parser;
    private Derivation           rulesStack;
    private StackElem            top;
    private KnownGoal            goal;
    private Partition            partitionInstance;

    public KPTable(UngerParserCommon parser){
        goals = new SortedMap[SingleRule.getLastId()];
        partitionInstance = new Partition(0,0);
        this.parser = parser;
        this.rulesStack = parser.getRulesStack();

        for(int i=0; i<goals.length; i++)
            goals[i] = new TreeMap();
    }

    /**
     * Returns the known goal from the "repository"
     * @param rule rule involved
     * @param pos beginning of the partition
     * @param len length of the partition
     * @return known goal from the list if exists
     * otherwise null
     */
    private KnownGoal KnownGoalNumber(SingleRule rule,
            int pos, int len){
        partitionInstance.pos = pos;
        partitionInstance.len = len;
        return (KnownGoal) goals[rule.getId()].
                get(partitionInstance);
    }


    public void StartNewKnownGoal(SingleRule rule,
            int pos, int len){
        goals[rule.getId()].put(
                new Partition(pos,len),new KnownGoal());
    }


    public void RecordKnownParsing(){
        top = (StackElem) parser.getStack().peek();
        goal = KnownGoalNumber(top.NmbField,
                top.PosField,top.LenField);
        goal.knownParsings.add(new Vector(rulesStack.
                getPart(goal.startParsing,
                        rulesStack.size())));
    }


    public boolean KnownGoalSucceeds(SingleRule rule,
            int pos, int len){
```

```
        KnownGoal goal = KnownGoalNumber(rule,pos,len);

        if (goal == null)
            return false;
        else{
            // If there are ANY parsings known
            // then try them
            for(Enumeration i=goal.
                    knownParsings.elements();
                i.hasMoreElements();){
                Collection known = (Collection)
                        i.nextElement();
                rulesStack.append(known);
                if (!parser.getStack().isEmpty()){
                    // Continue parsing after pasting the
                    // known parsing on the stack
                    parser.getStack().AdvanceTOS(len);
                    parser.DoTopOfStack();
                    parser.getStack().RetractTOS(len);
                }else
                    parser.parsingFound();

                rulesStack.removeLast(known.size());
            }
            return true;
        }
    }
}
```

Appendix C

Parse trees obtained from RRP grammars

C.1 Grammar: *Thesis*

C.1.1 Grammar definition file

```
#
# Grammar for describing thesis strucutre
#

Thesis
: "Intro" Chapter+ Bibliography Appendix*
;

Chapter
: "Par"+ "Sum"?
| "Sec"+
;

Bibliography
: "BibItem"+
;

Appendix
: "App"
;
```

Figure C.1: Simple grammar describing structure of a thesis

C.1.2 Parse tree

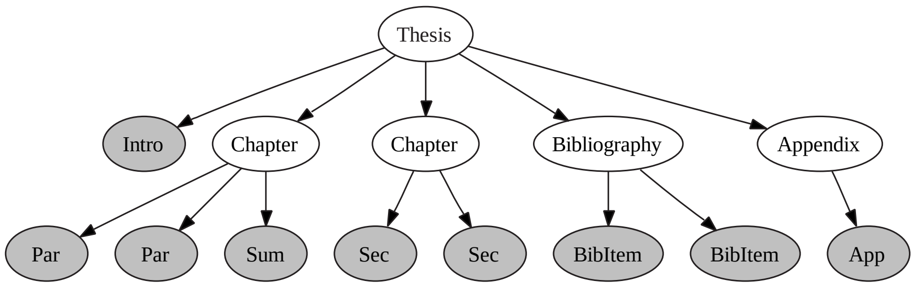


Figure C.2: Parse tree of a sentence: *Intro Par Par Sum Sec Sec BibItem BibItem App*

C.2 Grammar: *IBM VS Cobol II*

C.2.1 Cobol program: *id-div.commentr1.std*

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                     PROCDIV-COMMENTR1.
AUTHOR.
INSTALLATION.
DATE-WRITTEN.
DATE-COMPILED.
```

Figure C.3: Listing of a Cobol program used for experiments with the parser

C.2.2 Parse tree for *id-div.commentr1.std*

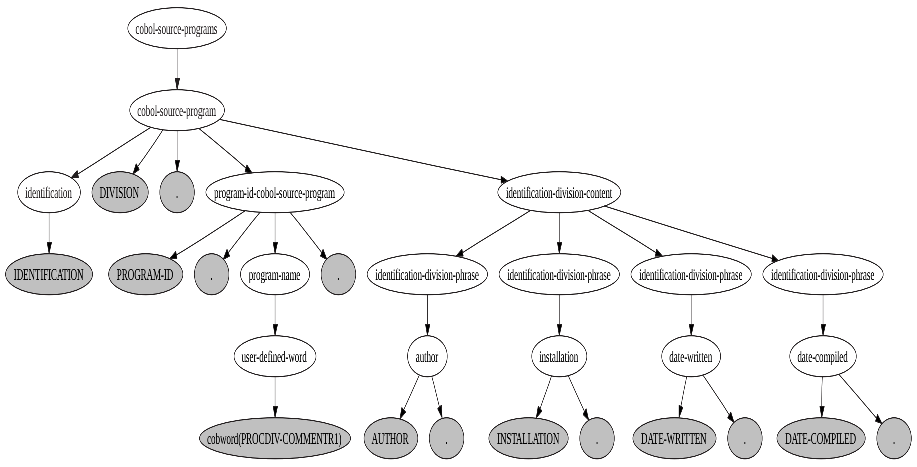


Figure C.4: Parse tree for Cobol program: *id-div.commentr1.std*

Appendix D

Grammar derived knowledge

D.1 A context-free grammar

```
#
# Master Project
# by Lukasz Kwiatkowski (lkwiatk@few.vu.nl)
# Vrije Universiteit Amsterdam, 2005
#
# Grammar for appendix C
#

A
: "b" "e" C D
| C "e" "e" D
| B "f" "g"
;

B
: "e" "e" "b" C
;

C :
;

D
: "a" "a" "b"
;
```

Figure D.1: Simple context-free grammar

D.2 Lengths of minimal and maximal derivable sentences

```
Lengths of MIN-MAX derivable string (min:max):
D = (3:3)
C = (0:0)
B = (3:3)
A = (5:5)
```

Figure D.2: Listing of the computed minimal and maximal lengths of derivable sentences from each non-terminal of the grammar D.1

D.3 Prefix, Suffix and Exclude sets

```
PREFIX sets:
D: ["a""a""b"]
C: []
B: ["e""e""b"]
A: ["b""e", "e""e"]

SUFFIX sets:
D: ["a""a""b"]
C: []
B: ["e""e""b"]
A: ["f""g", "a""a""b"]

EXCLUDE sets:
D: ["b""e", "f""g", "e""e"]
C: ["b""e", "f""g", "e""e", "a""a""b"]
B: ["f""g", "a""a""b"]
A: []
```

Figure D.3: Listing of the computed PREFIX, SUFFIX and EXCLUDE sets for each non-terminal of the grammar D.1

List of Figures

List of Tables

Bibliography

[1] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, *Compilars - Priniciples, Techniques, and Tools,* Addison-Wesley, Reading, MA, USA, 1986.

[2] Alfred V. Aho and Jeffrey D. Ullman *Principles of Compiler Design,* Addison-Wesley, 1972.

[3] Darius Blasband, *Parsing in a hostile world,* Preceedings of the Eighth Working Converence on Reverse Engineering, p. 291-301, IEEE Computer Society, October 2001

[4] Mark van den Brand, Alex Sellink and Chris Verhoef, *Current Parsing Techniques in Software Renovation Considered Harmful* Proceedings of the International Workshop on Program Comprehension, June 24-26, 1998 Ischia, Italy.

[5] M. van den Brand, J. Scheerder, J.J. Vinju, and E. Visser. *Disambiguation filters for scannerless generalized LR parsers,* In Compiler Construction (CC) '02, volume 2304 of LNCS.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms,* The MIT Press Cambridge, Massachusetts London, England, 1990.

[7] Jay Earley, *An Efficient Context-Free Parsing Algorithm,* Communications of the ACM, vol. 13, no. 2, p. 94-102, February 1970.

[8] John K. Gough *Syntax Analysis and Software Tools,* Addison Wesley: Sydney, 1987.

[9] Dick Grune and Ceriel Jacobs, *Parsing techniques - a practical guide,* Ellis Horwood, Chichester, England, 1990.

[10] Steven Klusener, *Experience report Generic Language Technology,* Internal note, Free University Amsterdam, February 2004.

[11] Steven Klusener, *Parsing software portfolios* Free Universiteit Amsterdam, January 2004.

[12] Steven Klusener and Ralph Lämmel, *Deriving tolerant grammars from a base-line grammar,* International Conference on Software Maintenance, ICSM 2003.

[13] Ralf Lämmel and Chris Verhoef, *Semi-Automatic Grammar Recovery,* SoftwarePractice and Experience, 31(15):13951438, December 2001.

[14] Ralf Lämmel and Chris Verhoef, *VS COBOL II grammar Version 1.0.4,* URL: http://www.cs.vu.nl/grammarware/browsable/vs-cobol-ii/, Vrije Universiteit, April 29, 2003.

[15] Scott McPeak and G C. Necula, *Elkhound: A fast, practical GLR parser generator,* University of California Postprints, 2004.

[16] Michael O'Donnel, *Sentence Analysis and Generation – a Systemic Perspective,* URL: http://www.wagsoft.com/Papers/Thesis/, Ph.D. Dissertation, Linguistics Department, University of Sydney.

[17] Jacob Palme, *Making Computers Understand Natural Language,* Artificial Intelligence and Heuristic Search, Edinburgh University Press 1971.

[18] James Power, *Notes on Formal Language Theory and Parsing,* URL: http://www.cs.may.ie/∼jpower/Courses/parsing/new-main.html, National University of Ireland, Maynooth, Co. Kildare, Ireland, November 2002.

[19] Louise Ravelli *More than a matter of form: a systemic functional approach to the automatic parsing of natural language,* M.Ph. Thesis, Dept. Of English, University of Birmingham.

[20] Jan Rosek, *Metody translacji - lecture notes,* Jagiellonian University, Kraków, Poland, 2002.

[21] Alex Sellink and Chris Verhoef, *Reflections on the evolution of COBOL* Technical Report P9721, University of Amsterdam, 1997. Available at http://www.cs.vu.nl/∼x/lib/lib.html.

[22] Andrey A. Terekhov and Chris Verhoef, *The realities of language conversions,* IEEE Software, p. 111-124, November/December 2000.

[23] Stephen H. Unger, *A Global Parser for Context-Free Phrase Structure Grammars,* Communications of the ACM, vol. 11, no. 4, p. 240-247, April 1968.

[24] Reinhard Wilhelm and Dieter Maurer, *Compilar Design,* Addison Wesley Reading, MA, USA, 1995.

[25] Niels P. Veerman, *Revitalizing modifiability of legacy assets,* In Proc. Conference on Software Maintenance and Reengineering (CSMR03), pages 19−29. IEEE Press, 2003.