Adaptive Multi-Pass Parsing

Adam Carmi
Adaptive Multi-Pass Parsing

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science

Adam Carmi

Submitted to the Senate of
the Technion—Israel Institute of Technology

ADAR 5770    HAIFA    February 2010
The research was done under the supervision of Prof. Michael Kaminski and Assoc. Prof. Ron Pinter in the department of Computer Science.

The generous financial help of the Technion is gratefully acknowledged.
# Contents

Abstract .......................... 1  
List of Symbols ........................ 2  

1 Introduction .......................... 4  
1.1 Adaptive Grammars .................. 4  
1.2 Syntax Macros ........................ 5  
1.3 Motivation .......................... 6  
1.4 Summary of Contributions ......... 8  
1.5 Related Work ........................ 10  
1.6 Organization of the Thesis ........ 17  

2 Preliminaries ........................ 18  
2.1 Relations .......................... 18  
2.2 Strings and Languages ............ 19  
2.3 Context-Free Grammars .......... 19  
2.4 Lists, Sets and Maps ............. 20  
2.5 LR(k) Parsing ..................... 20  

3 LR-based Grammar Adaptivity .... 25  
3.1 Unrestricted LR-based Adaptivity . 25  
3.2 Root-Driven LR-based Adaptivity . 27  
3.3 Lookahead LR-based Adaptivity ... 28  

4 Adaptive Multi-pass Grammars .... 30  
4.1 Overview .......................... 30  
4.2 Grammars, Derivations and Languages 33  
4.3 Computational Power .............. 37  
4.4 AMGs vs. CFGs ........................ 39  

5 Adaptive Multi-pass Parsers ....... 40  
5.1 Overview .......................... 40  
5.2 AMG Parsers (AMPs) ............... 42  
5.3 AMP Computations ................. 45  
5.4 Correctness ........................ 46  

6 Deterministic AMGs (DAMGs) ....... 57  
6.1 Computational Power ............... 57  
6.2 CFG-Like Derivations .............. 59  
6.3 Testing for Determinism .......... 62
7 DAMG Derivation Complexity
   7.1 CFG Rightmost Derivation Complexity ........................................... 65
   7.2 DAMG Derivation Complexity .......................................................... 78

8 Efficient DAMG(1) Parsing
   8.1 Vocabularies, Rules and Item Sets .................................................. 84
   8.2 Closures and Functions ...................................................................... 86
   8.3 Efficient LR(1) item set construction ................................................. 87
   8.4 Deterministic AMG(1) parsing complexity ........................................... 97

9 Conclusions and Future Work ................................................................ 102

Bibliography ......................................................................................... 104
List of Figures

2.1 LR(k) parsing routine ........................................... 21
2.2 LR(k) parser generation routines ................................... 22

4.1 AMG $G_{DU}$ for the language $L_{DU}$ .......................... 31
4.2 AMG $G_{UD}$ (with a multi-pass rule) for the language $L_{UD}$ ........ 33
4.3 AMG derivation of $abcd$ in $G_{UD}$ ................................. 38
4.4 AMG $G_{L,k}$ defined for the Turing machine $M_L$ ............... 38

5.1 AMP grammar adaptation routines .................................. 43
5.2 AMP parsing routines ............................................. 44

6.1 DAMG testing routines ............................................. 64

7.1 An $\epsilon$-free CFG with worst-case rightmost derivation complexity of $w \in T^+$ 68
7.2 A CFG with worst-case rightmost derivation complexity of $\epsilon$ ........ 69
7.3 A CFG with worst-case rightmost derivation complexity of $w \in T^+$ ..... 76

8.1 LR(1) item set computation routines ................................ 89
Abstract

An adaptive grammar is a formal grammar that explicitly provides mechanisms within the formalism to allow self-induced manipulation of the production rules. In the context of programming languages, adaptive grammars are useful for specifying (and parsing) syntax macros, and common context-sensitive syntactical restrictions, e.g., type correctness and scoping rules, traditionally referred to as the static-semantics aspects of programming languages. Although several adaptive grammar formalisms have been proposed since the early 70s, most fail to provide the mechanisms required for specifying syntactical constructs, where entities may appear before their point of declaration (e.g., goto statements, Java class members, etc.); and all but one fail to provide the mechanisms required for fully specifying syntax macro expansions. Furthermore, the over-permissive adaptive nature of these formalisms prevents the development of efficient, practical parsers. In this thesis we propose an adaptive formalism which is more restrictive with respect to grammatical manipulation yet powerful enough to handle constructs commonly found in extensible programming languages. It suggests a multi-pass approach that goes hand in hand with the adaptive paradigm, and elegantly solves the two aforementioned issues. Furthermore, by taking advantage of the restrictions, we are able to provide an LR(1)-based parsing algorithm that is amenable to practical implementation and handles both incremental and decremental changes in the grammar efficiently. We formally prove the correctness of the parser and analyze its complexity. To accommodate frequent grammar modifications, the parser lazily constructs its parse-table as parsing progresses. An efficient algorithm for computing LR(1) item sets is presented, that significantly improves upon previously known methods.
# List of Symbols

## Integers and Strings

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathbb{N} )</td>
<td>The set of all nonnegative integers</td>
</tr>
<tr>
<td>(</td>
<td>x</td>
</tr>
<tr>
<td>( j : x )</td>
<td>The ( j )-length prefix of string ( x )</td>
</tr>
<tr>
<td>( x : j )</td>
<td>The ( j )-length suffix of string ( x )</td>
</tr>
</tbody>
</table>

## Sets, Lists and Maps

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ e_1, e_2, \ldots }</td>
<td>A set containing elements ( e_1, e_2, \ldots )</td>
</tr>
<tr>
<td>\langle e_1, e_2, \ldots \rangle</td>
<td>A list containing elements ( e_1, e_2, \ldots )</td>
</tr>
<tr>
<td>\langle e_1, e_2, \ldots \rangle[i]</td>
<td>The element ( e_i ) at position ( i )</td>
</tr>
<tr>
<td>{ k_1 \mapsto v_1, k_2 \mapsto v_2, \ldots }</td>
<td>A mapping of keys ( k_1, k_2, \ldots ) to values ( v_1, v_2, \ldots )</td>
</tr>
<tr>
<td>{ k_1 \mapsto v_1, k_2 \mapsto v_2, \ldots }[k_i]</td>
<td>The value ( v_i ) mapped to the key ( k_i )</td>
</tr>
<tr>
<td>{ }</td>
<td>An empty mapping</td>
</tr>
</tbody>
</table>

## Context-Free Grammars

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a, b, c )</td>
<td>A terminal symbol</td>
</tr>
<tr>
<td>( S, A, B, C )</td>
<td>A nonterminal symbol</td>
</tr>
<tr>
<td>( X, Y, Z )</td>
<td>A symbol</td>
</tr>
<tr>
<td>( u, v, w, x, y, z )</td>
<td>A terminal string</td>
</tr>
<tr>
<td>( \alpha, \beta, \gamma, \delta, \omega )</td>
<td>A symbol string</td>
</tr>
<tr>
<td>( A \rightarrow \alpha )</td>
<td>A production rule</td>
</tr>
<tr>
<td>( G(N, T, P, S) )</td>
<td>A CFG ( G )</td>
</tr>
<tr>
<td>( D_G )</td>
<td>The nullability depth of ( G )</td>
</tr>
<tr>
<td>( W_G )</td>
<td>The nullability width of ( G )</td>
</tr>
<tr>
<td>( H_G )</td>
<td>The length of the longest loopless path in ( G )</td>
</tr>
<tr>
<td>( \Rightarrow )</td>
<td>The CFG derivation relation</td>
</tr>
<tr>
<td>( \Rightarrow )</td>
<td>The CFG rightmost derivation relation</td>
</tr>
<tr>
<td>( (\alpha_0, \alpha_1, \ldots, \alpha_n) )</td>
<td>A CFG rightmost derivation of length ( n )</td>
</tr>
<tr>
<td>( L(G) )</td>
<td>The language of ( G )</td>
</tr>
</tbody>
</table>

## LR(\( k \)) parsing

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ \</td>
<td>The endmarker symbol</td>
</tr>
<tr>
<td>( k )</td>
<td>Lookahead length</td>
</tr>
<tr>
<td>( [A \rightarrow \alpha \bullet \beta, y] )</td>
<td>An LR(( k )) item</td>
</tr>
<tr>
<td>( [\gamma]_G^k )</td>
<td>All LR(( k )) items valid for the symbol string ( \gamma ) in ( G )</td>
</tr>
<tr>
<td>( K_\gamma )</td>
<td>The kernel of ( [\gamma]_G^k )</td>
</tr>
</tbody>
</table>
The kernel nonterminals of $[\gamma]^k_G$

The non-kernel nonterminals of $[\gamma]^k_G$

The non-kernel rules of $[\gamma]^k_G$

An item set (i.e., state)

The set of all item sets in $G$

A state string

The state string implied by the symbol string $\gamma$ in $G$

**Adaptive Multi-Pass Grammars**

$\langle X, w \rangle$ An annotated symbol

$A_T$ The set of all annotated terminal symbols

$A_{N,T}$ The set of all annotated symbols

$a, b, c$ An annotated terminal symbol

$u, v, w, x, y, z$ An annotated terminal string

$\alpha, \beta, \gamma, \delta, \omega, \rho$ An annotated symbol string

$\bar{\rho}$ The core of the annotated symbol string $\rho$

$\tau_1, \tau_2, \tau_3, \ldots$ A variable symbol

$r$ A production rule

$\xi(r)$ The CFG skeleton of $r$

$f_r(\tau_1, \tau_2, \ldots, \tau_n)$ An annotation function

$\langle A, f_r \rangle \rightarrow X_1X_2\cdots X_n$ A single-pass production rule

$\langle A, f_r \rangle \leftarrow X_1X_2\cdots X_n$ A multi-pass production rule

$R_{N,T}$ The set of all single-pass production rules

$R_{N,T}$ The set of all multi-pass production rules

$R$ The set of all production rules

$\Delta(R, r, \rho)$ An adaptation function

$G(N, T, R_0, S, \Delta, k)$ An AMG $G$

$G_R^\xi$ The skeletal CFG underlying $G$ with respect to $R$

$(\gamma, R)$ An AMG configuration

$C_G$ The set of all AMG configurations of $G$

$\Rightarrow$ The AMG derivation relation

$\Rightarrow$ The AMG derivation by single-pass rule relation

$\Leftrightarrow$ The AMG derivation by multi-pass rule relation

$((\gamma_0, R_0), \ldots, (\gamma_n, R_n))$ An AMG derivation of length $n$

$L(G)$ The language of $G$

**Adaptive Multi-Pass Parsing**

$M$ AMP

$M_R^\xi$ The skeletal CFG underlying $M$ with respect to $R$

$(\phi, \gamma|x, R)$ An AMP configuration

$(S, \gamma|x, R)$ A valid AMP configuration

$\Rightarrow$ An AMP computation progression (i.e., parse action)

$L(M)$ The language of $M$
Chapter 1

Introduction

1.1 Adaptive Grammars

In the 1950s, Chomsky proposed a generative approach to grammar, in which sentences of a language are generated by an abstract engine [13]. Chomsky proposed a single abstract engine for all languages; each individual language is then defined by a set of rules that tell the engine how to generate all sentences of that language, and no others. He further distinguished several classes of grammars, depending on the generality of form of the grammar rules. The most general forms (types 0 and 1) are powerful, but opaque and difficult to work with. The more restricted (types 2 and 3), especially the context-free grammars or CFGs (type 2), are more lucid and easier to use, but not powerful enough to describe the syntax of most programming languages, which abound with context-dependent constructs such as lexical scoping rules and static typing.

In order to reconcile the desire to use CFGs with the need to rigorously define context-dependent programming language features, various extensions of the CFG model have been developed. Most of these extensions retain a CFG kernel, and augment it with a distinct facility that handles context-dependence. The most popular of these extensions are Attribute Grammars [26] introduced by Knuth in the late 1960s.

An attribute grammar consists of a CFG and an attribute system. Terminal strings are processed in two stages. First, the CFG is used to parse the terminal string. Then, the nodes in the parse tree are decorated with attribute name / value pairs. The attribute system associates with each CFG rule a semantic rule, which specifies the dependencies between the attributes of a parent node and the attributes of its children.

While originally devised as a technique for specifying the semantics of languages defined by CFGs, semantic rules are traditionally used for rejecting invalid sentences of superset languages defined by overly permissive CFGs, based on context-dependent criteria. This treatment of context-dependent syntactic constructs in the context of semantic analysis gives rise to the false notion that programming language syntax is merely that part of the syntax which can be described by a CFG, and everything else is semantics [32]. Although having a distinct advantage over context-dependent Chomsky grammars as a descriptive formalism, attribute grammars can easily obscure their own descriptions: the more context-dependent a programming language is, the looser its context-free superset will be, and the more heavily it will rely on its semantic rules. For example, [15] cites the two-and-a-half pages of semantic conditions specifying the context-dependent aspects of general procedure and function calls in the ADA programming language [46].

In recent years, a number of adaptive grammar formalisms [40] have been proposed for describing the full syntax of programming languages, in an attempt to elude the shortcomings of the traditional approach detailed above. Based on the observation that the set...
of permissible sentences within a small region in a particular program can be described by a CFG [17], these formalisms allow their grammars to "adapt" their (CFG-like) production rules to the various contexts created by individual programs. The basic principle underlying this approach, is that declarations (e.g., variable declarations) effectively modify or "extend" the grammar of the language. In effect, adaptive grammars offer a way to describe context-dependent language features in terms of context-free structure.

Example 1.1. Consider the following toy programming language, in which programs consist only of integer variable declarations and assignments of one variable to another. The context-free syntax of the language is described by the following CFG:

\[
\begin{align*}
\langle \text{program} \rangle & \rightarrow \langle \text{decl-list} \rangle \langle \text{stmt-list} \rangle \\
\langle \text{decl-list} \rangle & \rightarrow \langle \text{decl-list} \rangle \langle \text{decl} \rangle | \varepsilon \\
\langle \text{decl} \rangle & \rightarrow \text{int} \langle \text{id} \rangle ; \\
\langle \text{stmt-list} \rangle & \rightarrow \langle \text{stmt-list} \rangle \langle \text{stmt} \rangle | \varepsilon \\
\langle \text{stmt} \rangle & \rightarrow \langle \text{id} \rangle = \langle \text{id} \rangle ; \\
\langle \text{id} \rangle & \rightarrow \text{a} | \text{b} | \text{c} | \cdots | \text{z}
\end{align*}
\]

The only context-dependent restriction on valid programs is that an identifier cannot be used in a statement unless it has been declared.

Consider the following two programs:

\begin{align*}
(1) \quad \text{int x; int y; x = y;} & \quad (2) \quad \text{int x; x = y;}
\end{align*}

While both programs are syntactically correct with respect to the CFG, only the first program is valid. Using an adaptive grammar, the above context-dependent restriction can be easily enforced, if we replace the rule \(\langle \text{stmt} \rangle \rightarrow \langle \text{id} \rangle = \langle \text{id} \rangle ;\) with the rule

\[
\langle \text{stmt} \rangle \rightarrow \langle \text{declared-id} \rangle = \langle \text{declared-id} \rangle ;
\]

in the initial CFG and add a new rule \(\langle \text{declared-id} \rangle \rightarrow \text{a}\) whenever a declaration \text{int a;} is encountered during the expansion of the nonterminal \(\langle \text{decl-list} \rangle .\) As a result, the second program is no longer syntactically correct with respect to the (adaptive) CFG, since \(\langle \text{declared-id} \rangle \) does not derive the terminal y.

1.2 Syntax Macros

Independently of adaptive grammars, syntax macros have long been advocated as a means for extending the syntax of programming languages [29, 12, 11, 28, 49, 6]. A syntax macro definition has three components: a result type which is a nonterminal of the host grammar, a pattern specifying the invocation syntax, and a body that must comply with the result type. The result type declares the type of the body and thereby the syntactic context (e.g., statement, expression, etc.) in which invocations of the macro are permitted. The invocation pattern consists of a nonempty string of metavariables and tokens, the first of which is a token identifying the macro. The macro identifier serves as a delimiter simplifying the recognition of macro invocations. Each metavariable is associated with a syntactic context (i.e., a nonterminal) of the host language and constitutes a formal parameter of the macro. The macro body prescribes the replacement text which is to be substituted for a macro call. If the invocation pattern contains formal parameters then
the actual parameters supplied in a macro call are copied into the replacing text where indicated by corresponding formal parameters in the macro body.

For example, consider the following macro definition which syntax is borrowed from [6].

\[
\text{syntax } \langle \text{decl} \rangle \text{ declare } \langle \text{id} \rangle \text{ as integer } ; \quad ::= \{ \text{int } \langle \text{x} \rangle ; \}
\]

The keyword syntax identifies a macro definition. The result type is the nonterminal decl, implying that the macro can be invoked only where a declaration is syntactically expected, and also dictates the syntactic type of the macro body. The invocation pattern is

\[
declare \langle \text{id} \rangle \text{ as integer } ;
\]

where declare is the macro identifier, and \text{x} is a metavariable of syntactic type id. Finally, the macro body, consisting of the metavariable \text{x} is the declaration \text{int } \langle \text{x} \rangle ;.

The syntax of macro definitions is defined along with the base syntax of the hosting programming language. Macro definitions in turn, allow a programmer to explicitly extend the syntax of the host language by adding new grammar rules dictated by the macro’s invocation pattern. The meaning (i.e., semantics) of the new syntactic construct is defined by means of other syntactic constructs which meaning is already defined in the (extended) host language as dictated by the macro body. Using syntax macros, a programmer can substantially modify the syntax of a host language during the course of writing a program in that language. In effect, the grammar itself becomes subject to dynamic variation, and the actual syntax of the language becomes dependent on the program being processed.

For example, a compiler encountering the above macro definition (during syntactic analysis phase) would effectively extend the syntax of the toy programming language of Example 1.1 with the new rule

\[
\langle \text{decl} \rangle \rightarrow \text{declare } \langle \text{id} \rangle \text{ as integer } ;
\]

Subsequent declarations matching this rule, would be substituted with the macro body after replacing the metavariable \text{x} with the actual identifier provided in the macro call. As a result, the following program

\[
\text{syntax } \langle \text{decl} \rangle \text{ declare } \langle \text{id} x \rangle \text{ as integer } ; \quad ::= \{ \text{int } \langle \text{x} \rangle ; \}
\]

\[
declare x \text{ as integer } ;
\]

\[
declare y \text{ as integer } ;
\]

\[
x = y;
\]

\[(3)\]

is valid, and semantically equivalent (via macro expansion) to program (1) above.

Programming languages that permit the programmer to introduce explicit modifications to their syntax are said to possess the property of syntactic extensibility [38]. Early motivation for the development of such languages was their ability to reduce the "conceptual distance" between a base general-purpose language and the application domain, thereby simplifying the creation of concise and clear programs free from contamination with low-level details [44]. Recent work in this area [6, 4] is further motivated by the ability to leverage syntax extensibility as a vehicle for planned growth of a language [45].

1.3 Motivation

Despite the fact that adaptive grammars and syntactical extensibility share the same underlying concept of extensibility through grammar modification, research in these areas
followed distinct, independent paths. In fact, existing adaptive grammar formalisms are ill-suited for fully describing the syntax of syntactically extensible programming languages.

In his survey of adaptive grammars [15], Christiansen identified six context-dependent programming language restrictions that could not be properly modeled using adaptive grammars. One of these restrictions, namely forward references (e.g., goto statements that precede label declarations, recursive declarations, etc.), has not submitted to elegant description by most existing formalisms.

Furthermore, all existing grammar formalisms but [40] lack the descriptive machinery required for modeling the syntactic substitutions implied by syntactic macros and similar extension mechanisms. This cripples their ability to enforce context-dependent restrictions in syntactically extensible languages, since crucial contextual information may be hidden in (parameterized) macro bodies. As an example, consider program (3) of the extensible toy programming language of Example 1.1. Unless an adaptive grammar can properly interpret the user-defined declaration form declare x as integer; as the equivalent declaration int x; , it will fail to add the rule ⟨declared-id⟩ → x and thus consider the program as syntactically invalid.

Last, from a practical perspective, describing the syntax of an extensible programming language using an adaptive grammar is of little merit unless that grammar is accompanied by an efficient parser that can readily be incorporated into a compiler for that language. Such a parser (hereafter referred to as an adaptive parser) must be able to accept changes to its underlying grammar during the process of parsing a program. The applicability of existing adaptive parsers for this task is questionable, as their parse time complexity is exponential in either the length of the input sentence, or in the size of the (dynamic) grammar (even when adding or removing a single production rule), or plainly inconsistent with their respective formalisms.

The goal of this thesis is to develop an adaptive grammar formalism (and a corresponding parser) that can practically be used to fully specify the syntax of syntactically extensible programming languages. Utilizing such a formalism to define the syntax of extensible programming languages has the following advantages over the traditional syntax macro based approach:

1. Facilitate complete, uniform and concise syntax definitions covering both context-dependent constructs and extension facilities of programming languages.

2. Enhance the expressiveness of language extensions by potentially allowing any non-terminal of the language grammar to be used as a macro result type, and by allowing user defined CFG fragments as macro invocation syntax.

3. Limit the scope of macro declarations in accordance with the scoping rules of the language (encoded in the adaptive grammar).

4. Facilitate syntax restrictions by means of grammar rules removal. Syntax restrictions are useful for specializing a language to a specific application domain, adjusting a host-language grammar to better assimilate an embedded language, and for restricting the usage of deprecated language features (e.g., goto statements in structured programming languages).

5. Avoid the development of dedicated macro processing parsers by leveraging the grammar’s adaptive parser.
1.4 Summary of Contributions

The main contribution of this work is the development of an adaptive grammar formalism and a corresponding LR(\(k\))-based parser that can be practically used to fully specify the syntax of syntactically extensible programming languages. Through the development of the formalism, two additional results were obtained, pertaining to the complexity of CFG derivations and traditional LR parsing techniques.

Adaptive Multi-pass Grammars (AMGs)

We propose an adaptive grammar formalism that is more restrictive with respect to grammatical manipulation but powerful enough to concisely specify context-dependent restrictions commonly found in programming languages. Adaptivity restrictions are due to a \(k\)-length \((k > 0)\) lookahead string employed by the formalism’s derivation relation, which facilitates the efficient implementation of a corresponding LR(\(k\))-based adaptive parser.

We further define a sub-class of Deterministic AMGs (DAMGs) for which deterministic LR(\(k\))-based adaptive parsers can be constructed. We prove that this sub-class covers all languages covered by general AMGs and that DAMGs define the class of recursively enumerable sets even with a 1-length lookahead string. Last, we formally analyze the time and space complexities of DAMG derivations and prove that they are linear in the length of the derived terminal strings (in all passes) given a fixed bound on the size of the grammar and the number of grammar modifications.

In our AMG formalism, a global set of production rules is maintained throughout the derivation process, which determines the set of rules applicable for rewriting at each derivation step. On each such rewrite, a computable adaptation function is invoked which outputs sets of rules to be added and removed.

The problems of forward references and syntactic substitutions are resolved by means of a novel multi-pass derivation step, which allows certain terminal strings (as opposed to a single nonterminal in traditional CFG derivations) to be rewritten by distinguished multi-pass production rules. Associated with each production rule \(r\), is an annotation function \(f_r\) that synthesizes for its LHS nonterminal an annotation which is a terminal string that describes the fringe of the parse-tree rooted by that nonterminal. From the view point of an LR parser, when a multi-pass rule is reduced, its LHS nonterminal is discarded from the parse stack and its annotation is pushed back to the input stream, thus allowing it to be subsequently reprocessed by the parser.

For example, when reducing the rule representing the macro definition in program (3) of the toy language of Example 1.1, its annotation function synthesizes the terminal string ”\texttt{int(\{x\});}” which corresponds to the body of the macro. In addition, the new multi-pass rule

\[
\begin{align*}
\tau = \text{decl} & \leftrightarrow \text{declare id as integer} \\
\end{align*}
\]

is added to the grammar (note the \(\leftrightarrow\) denotation) along with the annotation function

\[
\begin{align*}
f_{\tau}(\tau_1, \tau_2, \tau_3, \tau_4, \tau_5) = "\texttt{int(\{x\});}".\text{Replace}("\texttt{\{x\}}", \tau_2)
\end{align*}
\]

Next, when the new rule reduces the declaration of the integer variable \(x\), the variable \(\tau_2\) binds with the annotation ”\texttt{x}” of the nonterminal \texttt{id} (the second symbol at the RHS of the rule), and the annotation function \(f_{\tau}\) consequentially synthesizes the annotation ”\texttt{int x;}”, which is then pushed back to the parser’s input stream. As a result, the declaration \texttt{int x;} is subsequently processed by the parser, ensuring that the rule

\[
\langle \text{declared-id} \rangle \rightarrow x
\]
is added as expected. Similarly, forward references can be properly validated by performing two passes over the program, the first with a looser grammar that identifies all declarations, and the second with a stricter grammar that allows only declared entities to appear.

Adaptive Multi-pass Parsers (AMPs)

We present an efficient LR(k)-based parsing algorithm for AMGs, that is amenable to practical implementation and handles both incremental and decremental changes to the grammar gracefully. Our parser utilizes a traditional LR(k) state stack to guide its parse actions and state transitions. We define a Deterministic AMP (DAMP) as an AMP that aborts if more than a single parse action is applicable. We formally prove the correctness of AMPs with respect to AMGs, and analyze the time and space complexities of DAMPs with a 1-length lookahead string. We prove that given a fixed upper bound on the number of grammar modifications and grammar size, a DAMP with a 1-length lookahead string parses a terminal string \( x \) in time \( O(n(O_t(\Delta) + O_t(f))) \) and space \( O(n \cdot O_s(f) + O_s(\Delta)) \), where all invocations of the parser’s adaptation function execute in time \( O_t(\Delta) \) and space \( O_s(\Delta) \), all invocations of annotation functions execute in time \( O_t(f) \) and space \( O_s(f) \), and \( n \geq |x| \) is the number of terminals shifted by the parser, including the terminals of \( x \) and all terminals pushed back to the input stream due to multi-pass reduce actions.

In order to handle frequent grammar modifications efficiently, our parser employs a lazy approach to parse-table generation. As parsing progresses, only those states that are pushed to the state stack are generated. Moreover, our parser avoids the parse-table representation entirely, by maintaining raw prefix-automaton states and inferring parse actions and state transitions directly from the state at the top of the state stack. Two additional stacks are maintained in parallel to the state stack: a symbol stack that maintains the prefix-automaton path defined by the contents of the state stack, and an annotation stack that maintains the annotations associated with symbols of the symbol stack. Following a grammatic modification, the entire state stack is discarded, and a new stack is built according to the path indicated by the symbol stack (with respect to the prefix-automaton of the new grammar). If the path does not fully exist in the new prefix-automaton, the parser terminates with an adaptation error. During rule reduction, the annotation for the rule’s LHS nonterminal is computed by applying the rule’s annotation function to the top of the annotation stack (which contains the annotations of all RHS symbols). When reducing a multi-pass rule, the traditional reduce action is further extended by discarding the representations of the LHS nonterminal from all stacks and pushing its annotation back to the parser’s input stream. Furthermore, the representations of all terminal symbols uncovered at the top of the parser’s symbol stack are also discarded and their annotations are pushed back to the parser’s stream, to ensure compliance with the formalism’s derivation relation.

Efficient LR(1) item set construction

We present an efficient algorithm for computing LR(1) states which improves upon the traditional algorithm by constructing a dependency graph representing the contribution of lookahead strings between non-kernel items. Lookahead sets are computed by traversing this graph in an SCC-aware fashion, thus avoiding the recurring redundant computation steps inherent in the traditional algorithm. Our algorithm is further optimized by exploiting the following property of LR(k) states: for any given LR(k) state \( q \), production rule \( A \rightarrow \alpha \in P \) and terminal string \( y \) we have

\[
[A \rightarrow \bullet \alpha, y] \in q \implies \{[A \rightarrow \bullet \alpha', y] | A \rightarrow \alpha' \in P \} \subseteq q
\]
It allows us to reduce the size of the dependency graph by representing sets of non-kernel items sharing the same LHS nonterminal by a single (nonterminal) node.

We formally prove the correctness of the algorithm, and analyze its time and space complexities. We prove that for a given CFG $G(N,T,P,S)$ where $m \geq 0$ is the maximal number of occurrences of any symbol $X$ in the RHS of some rule in $P$, our algorithm computes an LR(1) item set in time and space $O(\min\{|G|, m|P|\}|T|)$.

**CFG rightmost derivation complexity**

We set bounds on the time and space complexities of rightmost derivations of sentences in a free-form context-free grammar and formally prove their validity. We identify three grammar dependent properties, namely nullability depth, nullability width and longest loopless path by which the bounds are stated with respect to the length of the first and last sentential forms of a derivation. For the special case where the first sentential form is a nonterminal and the last sentential form is a terminal string, we prove that our bounds are tight in the sense that there exists a sequence of grammars that actually attain them.

The granularity of our bounds validates the practical applicability of CFG-based adaptive grammars to the task of fully specifying the syntax of extensible programming languages, as they are independent of the size of the grammar. It can be shown (although beyond the scope of this thesis), that the grammar modifications required for specifying context-dependent constructs of programming languages have a minor impact on the above mentioned grammar properties and thus on the overall worst-case complexity of derivations.

### 1.5 Related Work

**Attribute Grammars**

Attribute grammars [26] are the most used tool for the full description of the syntax of traditional programming languages. The model provides for a precise and readable notation, and at the same time the attributes and their dependencies represent an abstraction over the data structures and data flow in a compiler, and thus serve as an implementation guideline. However, the modeling of context-dependent syntactic constructs by means of semantic rules, blurs the distinction between language syntax and its semantics (i.e., intended computations), and often leads to overly complex semantic constructs and dependencies, accompanied by a superficial CFG specification.

The approach proposed by the AMG model and many of the other adaptive formalisms, is to retain the conceptual clarity and simplicity of the CFG model as a descriptive tool, by employing a semantic rules-like mechanism (namely, the adaptation function), solely for the purpose of specifying modifications to the underlying set of CFG-like production rules during the derivation process. As demonstrated in the literature [14, 8, 9, 15, 40, 5, 35], this organization results with concise modification specifications for many context-dependent syntactic constructs common in modern programming languages.

The shortcomings of attribute grammars, are manifested in their extremity when considering syntactically extensible languages. Since the CFG component of an attribute grammar is fixed, a pure attribute grammar specification for such languages would consist of a CFG that accepts arbitrary terminal strings, and fully rely on the attribute system to validate the syntactic correctness of programs. To accomplish this, the attribute system must model the syntactic backbone of the language, the extension facility (e.g., syntax macros) and the syntax modifications resulting from its application. Further-
more, language developers would be required to write custom parsers for such languages, since automatic parser generation tools such as YACC [24] generate parsers according to a fixed set of CFG production rules.

AMGs, on the other hand, allow for straightforward modeling of syntactically extensible languages by careful utilization of rule annotation functions, adaptation functions, and multi-pass rules (as described in Section 1.4). By modeling an extensible language using an AMG, compiler developers are spared the effort of developing a custom parser, due to the availability of a powerful, efficient, adaptive multi-pass parser for their grammars.

The AMG formalism can be considered as an extension of a restricted form of attribute grammars, where symbol annotations are represented as a single synthesized attribute associated with each grammar symbol, and rule annotation functions are semantic rules that evaluate the values of these attributes during the bottom-up, left-to-right construction of a parse-tree. It should be noted, that although the AMG model defines the class of recursively enumerable sets, it is not intended for describing the semantics of programming languages. For this purpose, attribute grammars are clearly a superior model.

Adaptive Grammars

We now compare our AMG formalism with other adaptive grammar models. Since a comprehensive survey of many adaptive grammar models is already given in the literature [14, 15, 40], we avoid the full specification of these models, and only highlight the areas where they differ from AMGs.

Extensible Context-Free Grammars (ECFGs) [48] were developed for describing the extensible programming system ECL [47]. ECFGs consist of a context-free-grammar and an associated deterministic finite state transducer. The finite state transducer analyzes the source text in parallel with normal parsing (left to right, single pass), and whenever a pattern has been recognized as a grammar rule (enclosed in special brackets), it can either be added or deleted from the grammar directing the parser. The adaptation function of ECFGs is a finite state transducer which is significantly weaker than the computable adaptation function of AMGs. As a result, the model is incapable of describing context-sensitive aspects of programming languages that rely on scoping rules, nor can it handle forward references. While intended to describe syntax modifications due to syntax macros, ECFGs can only model the invocation syntax of defined macros; macro expansion is handled at later stages of compilation. Lastly, ECL employs an LR(k) parser guided by a traditional parse table. On every grammar modification, parse tables are regenerated from scratch and parsing commences using the "new" parser. The worst-case time and space complexities of each grammatical modification are thus exponential in the size of the grammar.

Dynamic Syntax [20] is a generative device based on λ-calculus, originally developed for generating syntactically correct programs for testing compiler front-ends [19, 18]. CFG vocabulary symbols and production rules are represented by so called syntactic functions ranging over zero or more variables, which generate program phrases. The LHSs and RHSs of production rules represent function names and definitions, respectively. The body of a rule function concatenates the program phrases generated by RHS symbol functions. These phrases are produced by applying symbol functions to zero or more arguments which may also be syntactic functions producing program phrases. A side effect of such function applications is the introduction of new rules (i.e., functions generating program phrases) which are subsequently provided as arguments to other syntactic functions. Forward references are handled by partial binding of variables. That is, it is possible to provide a partial set of arguments to a syntactic function, in which case, it results with a function
that expects the remaining arguments in order to produce a program phrase. This step corresponds to a first pass of a compiler. As a side effect of this "first pass", new functions (i.e., production rules) are introduced, that can generate program entities that correspond to generated declarations. The new functions are then passed as arguments to the partially evaluated functions produced by the "first pass" resulting with a complete program phrase. This step corresponds to a second pass of a compiler. Using similar techniques, dynamic syntax devices can model other common context-dependent syntactic constructs. The application of the model to extensible languages was not considered by its inventors but seems feasible. In the referenced paper, the description of dynamic syntax is not properly formalized, but rather given as a series of partial specification examples. A major difficulty that arises from the representation of grammar rules as λ-calculus expressions, is that multiple CFG rules may share the same LHS while function names cannot be associated with multiple function bodies. Moreover, no parsing algorithm is described for accepting languages described by means of dynamic syntax.

Dynamic Template Translators (DTTs) [31] are an extension of syntax-directed translation schemes: the application of a production rule triggers the execution of an action sequence that can optionally add new production rules, delete existing production rules, or add actions to the action list of an existing production rule. A production rule consists of a single LHS element, and zero or more RHS elements. Each element is a string of symbols over a unified alphabet of symbols and templates. Terminal elements are distinguished from nonterminal elements by their prefix. Only nonterminal elements may appear at the LHS of production rules. A sequence of template symbols appearing in a production rule is called a template name. Multiple occurrences of the same template name may appear in a production rule and in production rules of its action sequence. At each step of a DTT derivation, a sentential form consisting of template-free elements is rewritten using a production rule. Before a rule is applied, its template names are substituted with symbol strings, such that all occurrences of a template name are substituted with the same string. The resulting template-free production rule, can then be used to rewrite a single nonterminal element of the sentential form, which matches its LHS nonterminal element. The model is shown to define the class of recursively enumerable sets. However, due to the lack of a multi-pass mechanism, the grammar cannot gracefully model forward references and syntax macro expansions.

Generative Grammars [14] are a generalization of extended attribute grammars [30] in which the entire structure of the parse tree is directly dependent on attribute evaluation. A distinguished (inherited) attribute is used for passing grammar rules. The derivation relation is defined in terms of this attribute, such that a nonterminal may only be rewritten by rules included in its grammar attribute. Semantic rules can synthesize new rules and remove rules from sets applied to subsequent derivation steps. The model was later reformulated in terms of definite clause grammars under the name Generative Clause Grammars (GCGs) [15]. Unlike AMGs and many of the other adaptive formalisms, the set of production rules applicable in each derivation step, is not a global state of the derivation but rather associated locally with each nonterminal of the sentential form. As a result, the order in which nonterminals can be rewritten is not imposed by the model. In order to handle forward references or model macro expansions, GCGs must employ traditional attribute grammar techniques, which are not considered satisfactory by the model's inventor. A parser for GCGs was implemented as a Prolog program, which (according to the author) suffers from a large amount of backtracking and a potentiality for infinite loops.

Modifiable Grammars [8, 7, 9], consist of a CFG and a recursive transducer that halts on all inputs. An LR(1)-based parser for a bottom-up variant of the model (BUMG) is
described in [7]. Upon rule reduction, the full sequence of rules previously reduced by the parser is provided to the recursive transducer, which outputs a list of rules to be added and removed from the grammar. Most of the examples provided in the referenced papers, heavily rely on the existence of an attribute system, which is completely absent in the formal definition of the model. Moreover, the proposed LR(\(k\))-based parser is clearly inconsistent with the formalism. Its actions are guided by a traditional LR(0) prefix-automaton, that restricts the set of reduceable rules at each parse-step to those which may lead to a successful parse according to the current set of grammar rules. However, this restriction is not reflected in the formalism. The parser further attempts to resolve LR conflicts by means of lazily computed lookahead terminals, which further deviates from the adaptive framework implied by the formalism. Similarly to our adaptive multi-pass parser, a BUMG parser lazily computes prefix-automaton states as parsing progresses. However, unlike our parser it first computes a complete NFA representing the grammar and lazily turns it into a deterministic automaton. In face of a grammatical modification, the parser traverses all the edges and states of the automaton and updates it to reflect the new grammar. Similarly to the ECFG parser, the time and space complexities of this operation are exponential in the size of the grammar. Furthermore, following the update of the automaton, the parser state stack may contain empty states. While our parser reports an error in such cases (in consistency with the AMG model), the BUMG parser would fail only if the parser attempts to pop such states from the state stack. As part of its error recovery scheme, the BUMG parser may enter a ”multi-pass” parsing mode, where it attempts to perform reductions not from the top of the parse stack, but rather from its deeper states. This form of multi-pass behavior may be used as an ad-hoc technique for capturing forward references. However, by no means is it satisfactory as a modeling technique for such constructs. Furthermore, since repeating passes only occur on the state stack, this method cannot be used to specify syntax macro expansions, where the expanded text differs from the macro invocation string.

**Recursive Adaptable Grammars (RAGs)** [40] is a formalism designed with the stated goal of incorporating grammar adaptivity without compromising the conceptual elegance of CFGs. The alphabets of terminals, nonterminals and semantic values are all identical, such that any value in the combined alphabet is equally capable of playing any of these roles. Symbols of this combined alphabet are called answers. A rule function maps each answer with a (possibly empty) set of unbound rules that can rewrite it. Unbound rules consist of variables which are bound to answers when a rule is applied. Through this binding process, semantic values synthesized during derivation can be incorporated in the RHS of applied rules, where they play the role of terminal, nonterminal, or semantic symbols. A binary query operator, written \(a : b\), denotes the semantic value synthesized when deriving the answer \(b\) from the answer \(a\). Expressions involving this operator may appear as ”symbols” of unbound rules. When applying such rules, query occurrences are evaluated, resulting with the ”recursive” invocation of the derivation process. The stated role of this operator is to facilitate arbitrary computations expressed by means of successions of trivially computable derivation steps. Similarly to Generative Grammars [14] (and CFGs), the order in which answers of a sentential form can be rewritten is not imposed by the model. Expressions appearing in RAG unbound rules, can be used to shape synthesized semantic values similarly to how AMG annotation functions synthesize annotation strings. Moreover, utilizing the query operator, recursive derivations can simulate the effect of AMG multi-pass rules. On the other hand, the very ”elegance” of the model can result with overly complex specifications that could be greatly simplified utilizing the brute force of an AMG adaptation function. Furthermore, the permissive nature of the model’s derivation relation, significantly complicates the development of a
practical RAG parser. In fact, the first and only published attempt at this task [37], resulted with a top-down breadth-first parser for a restricted subclass of RAGs, that does not allow left-recursive RAG rules, and which time complexity is exponential in the length of its input string even for grammars not involving the query operator.

Evolving Grammars [10] consist of a CFG and a recursive transducer. Upon rule reduction, the transducer is provided with the reduced rule, and results with a set of nonterminals and rules to be added to the grammar. A later presentation of the capabilities and applications of a corresponding Dynamic Parser [35], reveals that its grammar specifications further consist of so called nonterminal parameters and semantic actions that play a similar role to that of AMG symbol annotations and annotation functions. The values of these parameters can be used by the transducer when adding rules to the grammar. Moreover, rule additions can be specified in the context of grammatical scopes that can be added and subsequently removed from the grammar, thus facilitating rule deletion. Due to the limited input provided to the transducer and lack of a multi-pass mechanism, the grammar cannot gracefully model forward references and macro expansions. An LALR(1)-based dynamic parser implementation called Zz is publicly available at [34]. Unfortunately, similarly to the parser described in [7] for Modifiable Grammars, it is inconsistent with its formalism. The parser may thus avoid reducing rules that are applicable for derivation by its formalism. Furthermore, following grammar modifications, the LR(0) states stored in the parser’s state stack are not updated according to the new grammar. This implies that subsequent GOTO actions (and newly generated target states) inferred from these stale states, will be computed with respect to the old grammar rather than the new one, as would be expected.

Dynamic Grammars [5] is a formalism consisting of a CFG and a recursive transducer. Its derivation relation is defined by means of a dynamic configuration which corresponds to the state of a bottom-up CFG parser. Two types of derivation steps are allowed: scan, which corresponds to a shift action, and emit, which corresponds to a reduce action. Upon rule reduction (i.e., emit derivation step), the transducer is provided with the current configuration which consists of a CFG, a sentential form, the full sequence of previously reduced rules, and the entire remaining input stream. In addition, the transducer is provided with the reduced rule, which is of the configuration’s CFG, and which RHS is an LR-handle of the configuration’s sentential form. As output, the transducer produces a new CFG to be included in the resulting dynamic configuration. Unlike Modifiable Grammars and Evolving Grammars, Dynamic Grammars are more restrictive with respect to rule reduction, such that only rules which can lead to a successful parse according to the current grammar can be reduced (see Root-driven LR-based adaptive grammars in Chapter 3). This, in turn, facilitates the appropriate construction of an LR(0)-based dynamic parser. The parser lazily generates an LR(0) prefix-automaton as parsing progresses. It attempts to resolve LR-conflicts by simulating all subsequent parse steps involving at most $k > 0$ shifted terminals. Throughout this process, grammar modifications are applied and reverted as the different computation paths are explored. Paths are discarded when syntax errors are encountered. If all successful paths correspond to a single (conflicting) action, the conflict is resolved by choosing that action. If no successful path is found, a syntax error is reported. Upon grammar modification, affected LR(0) states are located and disconnected from the prefix-automaton. The time complexity of this process is exponential in the size of the grammar. Similarly to our AMG parser,
following grammar modifications, states contained in the parser’s stack are updated to reflect the new grammar. Lastly, due to the lack of a multi-pass mechanism, Dynamic Grammars cannot gracefully model forward references and macro expansions.

Meta-S Calculus [21, 22, 23] is a CFG-like formalism developed to describe the formal properties of the commercial adaptive parser Meta-S. The vocabularies of terminals and nonterminals are finite and immutable. Each nonterminal symbol is implicitly associated with a set of terminal strings that can rewrite it during derivation. A terminal string derived from any nonterminal appearing at the RHS of a production rule can be added or removed from these sets by means of so called binding expressions. Production rules may also consist of predicates of the form $\{\alpha\}^\theta$ where $\alpha$ is a string of symbols, and $\theta$ is a nonterminal. A production rule can be applied in a derivation only if the terminal string derived by $\alpha$ is also derivable by $\theta$. Evaluation of predicates has the effect of recursively initiating the derivation process, and implies the usual evaluation and side-effects of binding expressions. Using the predicate mechanism, a multi-pass effect can be easily specified, thus facilitating proper handling of forward references. However, since grammar modifications are limited to the addition of rules consisting only of terminal RHSs (through binding expressions), parameterized syntax macros cannot be properly specified by the model. The Meta-S parser, is a backtracking LL(k), recursive decent parser. Consequentially, it is unable to contain left-recursive production rules, and its worst-case time complexity is exponential in the length of the parsed input even for grammars not involving grammar modifications and recursive predicate evaluations.

Syntax macros

Syntax macros were first introduced in 1966 in two separate papers [12, 29]. Since then, many systems have been proposed, which greatly vary in their characteristics and properties.

In most implementations of syntax macros, a macro invocation must start with a macro delimiter, which is usually an identifier consisting of letters. This restriction prohibits macro systems from introducing new infix operators. In camlp4 [36], macro invocation syntax may start with a formal parameter. Macro systems such as the Java Syntactic Extender (JSE) [4] only allow for a fixed set of formal macro parameters following the macro identifier, while camlp4, Dylan [39] and Metamorphic Syntax Macros (MSMs) [6] allow argument syntax to be described by means of user defined grammar. Some macro systems, such as the Meta Syntactic Macro System ($MS^2$) [49] require that macro delimiters be unique, while others (e.g., Dylan, MSMs, JSE) can detect the most suitable macro definition according to the full invocation syntax.

There are two notions of type in conjunction with syntax macros, namely result types and argument types, both ranging over the nonterminals of the host language grammar. Some syntax macro systems verify that macro bodies conform to their result type, while others don’t. Different systems also differ in the amount of nonterminals from the host grammar that can be used as macro types.

A macro body may contain further macro invocations. Some macro systems expand macro bodies eagerly at their point of definition (e.g., MSM, $MS^2$), or lazily at each invocation (e.g., Scheme [16]). An eager strategy will find all errors in the macro body at definition time, even if the macro is never invoked. Some systems use automatic $\alpha$-conversion of identifiers in macro bodies to obtain hygienic macros [27].

Most macro systems have one-pass scope rules for macro definitions, meaning that a macro is visible from its lexical point of definition and onward. Only in $MS^2$, macro definitions are available even before their lexical point of definition. Some systems allow
macros to be undefined or redefined, and can limit the scope of macro definitions (e.g., MSMs).

Using AMGs, the full spectrum of the above mentioned syntax macro characteristics can be gracefully modeled. Syntax macros allowing arbitrary user-defined macro invocation syntax can be easily specified, and the prevailing macro-delimiter requirement can be lifted. Providing that macro invocations are unambiguous, correct macro definitions are automatically chosen during normal AMG derivation. Using the multi-pass facility of the formalism, forward references to macros can be easily modeled, and due to the formalism’s support for decremental grammar modifications, local macro definitions, macro redefinition and macro removal can be easily specified. Depending on the rules modeling macro definitions, macro bodies can either be eagerly or lazily expanded. Moreover, careful combination of multi-pass rules and annotation functions can be used to implement α-conversion of identifiers in macro bodies.

Most syntax macro systems utilize hand-written LL(1)-based parsers to process macro definitions and activations. Such parsers cannot handle left-recursions and heavily rely on macro-delimiters for detecting macro invocations. As an exception, the macro system camlp4 is built upon the extensible Generalized LR parser dypgen [33]. Unfortunately, in face of syntax modifications, the parser fully reconstructs its parse tables, a procedure which time and space complexities are exponential in the size of the grammar. A further source of complication is due to the fact that macro expansions are often implemented as transformations of Abstract Syntax Trees (ASTs) produced by macro processing parsers. These systems are therefore intimately tied to the internal data structures and APIs of the hosting compiler.

By modeling a syntax macro-system using an AMG, the difficult task of developing a corresponding parser and macro expansion processor is spared, due to the availability of an efficient adaptive multi-pass parser for the given AMG specification. Furthermore, since the parser is capable of handling both syntax modifications and macro expansions, the entire processing of syntax macros is contained in the syntax-analysis phase of compilation, resulting with an improved compiler organization.

**LR(1) item set construction**

A traditional algorithm for computing LR(\(k\)) states is due to Knuth [25]. This algorithm appears in most parsing and compiler construction textbooks (e.g., [3, 2]). Given an input set of kernel items, the algorithm naively computes a complete LR(\(k\)) state for a CFG \(G(N,T,P,S)\) by iteratively traversing all existing items and production rules, and adding newly discovered items not already included in the item set. This procedure is repeated until no more items can be added. Assuming \textsc{First} sets are precomputed for all nonterminals and production rule suffixes, the algorithm computes a complete LR(1) item set in time \(O_t(|G||P|^3|T|^3 \cdot \log(|G||T|))\) and space \(O_s(|G||T|)\). The algorithm listed in Figure 2.2, is a slight variation of the traditional algorithm, where a set of LR(\(k\)) items

\[
\{[A \rightarrow \alpha \bullet \beta, x_1], [A \rightarrow \alpha \bullet \beta, x_2], \ldots, [A \rightarrow \alpha \bullet \beta, x_n]\}
\]

is represented as the data structure \([A \rightarrow \alpha \bullet \beta, \{x_1, x_2, \ldots, x_n\}]\) which is internally implemented as a reference to a rule \(A \rightarrow \alpha \beta\), an integer \(\alpha\) identifying the location of the \(\bullet\) symbol in the RHS of the rule, and a set of terminal strings \(\{x_1, x_2, \ldots, x_n\}\).

In [43] (page 26) a faster algorithm is given for constructing a canonical LR(\(k\)) prefix-automaton for a CFG. The algorithm initially computes a dependency graph, which states are LR(\(k\)) items, and edges connect each item \([A \rightarrow \alpha \bullet B \beta, y]\) with an item \([B \rightarrow \bullet \omega, z]\) if \(z \in \text{FIRST}_k(\beta y)\). For the case \(k = 1\), the graph consists of at most \(|G||T|\) states and
\(|G| |P| |T|^2\) edges. Given a set of kernel LR(1) items, all non-kernel items of an item set can be computed by performing a DFS over the dependency graph and returning all items reachable from items of the kernel set, in time and space \(O(|G| |P| |T|^2)\).

As described in Section 1.4, our algorithm computes a complete LR(1) item set in time and space \(O(\min\{ |G| , m |P| |T| )\), where \(m \geq 0\) is the maximal number of occurrences of any symbol \(X\) in the RHS of some rule in \(P\).

**CFG derivation complexity**

Bounds on the time and space complexities of derivations in free-form context-free grammars are stated and rigorously proved in [42] (and in [41]). A sequence of context-free grammars that attain these bounds are also shown to exist, thus proving the *tightness* of the bounds. The bounds apply to all derivations of terminal strings from a single nonterminal symbol of a grammar. They are stated in terms of three CFG properties: (1) the number \(n\) of nonterminals that derive a nonempty terminal string, (2) the number \(n'\) of nullable nonterminals, and the length \(m\) of the right-hand side of the longest rule of the grammar. For example, it is shown that if \(A\) is a nonterminal, \(w \in L(A) \setminus \{\varepsilon\}, n \geq 1,\) and \(m \geq 2,\) then \(A\) derives \(w\) simultaneously in time

\[
(2nm' - (m'n' - 1)/(m - 1))|w| - nm'n' + (m'n' - 1)/(m - 1)
\]

and space

\[
|w| + n'(m - 1)
\]

Our work improves upon the above results, by identifying more granular properties of context-free grammar, which more accurately determine the complexity of derivations. The correctness of previous results is not undermined, since in some pathological cases, the values of our grammar properties coincide with those of the above (as in the case of the grammar sequence shown in [42]). Furthermore, our bounds apply to rightmost derivations of general symbol strings from general symbol strings, and not limited to derivations of terminal strings from a single nonterminal.

### 1.6 Organization of the Thesis

The remainder of this thesis is organized as follows. In Chapter 2 we review the mathematical and computer science background on which the presentation in this work is based. In Chapter 3 we examine various notions of LR-based grammar adaptivity, and position our formalism and other LR-based formalisms in an hierarchy characterized by a progression of restrictions on derivations. In Chapter 4 we introduce and formally define our AMG formalism, and prove that AMGs define the class of recursively enumerable sets. In Chapter 5 we present a nondeterministic LR(\(k\))-based Adaptive Multi-pass Parser (AMP) for AMG languages and formally prove its correctness. In Chapter 6 we define a subclass of AMG grammars called Deterministic AMGs (DAMGs) for which AMPs operate deterministically. We study the properties of DAMGs and show that they define the class of recursively enumerable sets. We further prove that CFG-like derivations in DAMGs correspond to shortest CFG rightmost derivations. In Chapter 7 we consider the time and space complexities of AMG derivations. As a preliminary step, we establish bounds on the time and space complexities of rightmost derivations in context-free-grammars. In Chapter 8 we state bounds on the time and space complexities of deterministic AMPs utilizing a single lookahead symbol. We also present an efficient algorithm for computing LR(1) item sets, formally prove its correctness and determine its time and space complexities. Lastly, in Chapter 9 we conclude this work and discuss areas for future research.
Chapter 2

Preliminaries

In this chapter we review the mathematical and computer science background on which the presentation in this work is based.

2.1 Relations

Let \( A \) and \( B \) be sets. A relation \( R \) from \( A \) to \( B \), denoted by \( R : A \to B \), is any subset of the Cartesian product of \( A \) and \( B \). That is, \( R \subseteq A \times B \). \( A \) and \( B \) are called the \textit{domain} and \textit{range} of \( R \), respectively. \( R \) is a \textit{relation on} \( A \) if \( A = B \). \((a,b) \in R\) can be denoted by \( aRb \). Let \( A' \) be a subset of \( A \). The \textit{image of} \( A' \) \textit{under} \( R \), denoted by \( R(A') \), is defined by

\[
R(A') = \{ b \in B | aRb \text{ for some } a \in A' \}
\]

In the case of the singleton set \( \{a\} \) we may write \( R(a) \) for \( R(\{a\}) \). The relation \( R^{-1} \) from \( B \) to \( A \) defined by

\[
R^{-1} = \{ (b,a) \in B \times A | aRb \}
\]

is called the \textit{inverse} of \( R \). The product of relations \( R_1 : A \to B \) and \( R_2 : B \to C \), denoted by \( R_1 R_2 \), is the relation from \( A \) to \( C \) defined by

\[
R_1 R_2 = \{ (a,c) | aR_1 b \text{ and } bR_2 c \text{ for some } b \in B \}
\]

The \textit{identity relation} on \( A \) denoted by \( id_A \), is defined by

\[
id_A = \{ (a,a) | a \in A \}
\]

Let \( R \) be a relation on \( A \) and \( n \) be a natural number. The \textit{n}th \textit{power of} \( R \), denoted by \( R^n \), is defined recursively by

\[
R^n = \begin{cases} 
  id_A & n = 0 \\
  RR^{n-1} & n > 0 
\end{cases}
\]

The \textit{transitive closure of} \( R \), denoted by \( R^+ \), is the relation on \( A \) defined by

\[
R^+ = \bigcup_{n=1}^{\infty} R^n
\]

The \textit{reflexive transitive closure of} \( R \), denoted by \( R^* \), is the relation on \( A \) defined by \( R^* = id_A \cup R^+ \). A relation \( f \) from a set \( A \) to a set \( B \) is a \textit{partial function} if for all \( a \in A \), \( f(a) \) contains at most one element. If in addition, \( f(a) \) is nonempty for all \( a \in A \), then \( f \) is called a \textit{(total) function}. 

Technion - Computer Science Department - M.Sc. Thesis
MSC-2010-11 - 2010
2.2 Strings and Languages

The notions of symbol and string are assumed here. The empty string is denoted by \( \varepsilon \). An alphabet is a set of symbols. \( V^* \) denotes the set of all strings of symbols from \( V \). The length of a string \( \alpha \) is denoted by \( |\alpha| \). The \( k \)-length prefix and suffix string of a string \( \alpha \) is denoted by \( k : \alpha \) and \( \alpha : k \), respectively. \( k > |\alpha| \) implies that \( k : \alpha = \alpha : k = \alpha \). A language is a set of strings.

2.3 Context-Free Grammars

A context-free grammar (CFG) is a quadruple \( G = (N, T, P, S) \) where \( T \) is a finite set of terminal symbols, \( N \) is a finite set of nonterminal symbols disjoint from \( T \), \( S \in N \) is the start symbol, and \( P \) is a finite subset of \( N \times (N \cup T)^* \) where each member \((A, \omega)\) is called a production rule, written \( A \rightarrow \omega \). A production rule which right-hand-side (RHS) is \( \varepsilon \) is called an \( \varepsilon \)-rule. A grammar which has no \( \varepsilon \)-rules is said to be \( \varepsilon \)-free. The size of a production rule \( r = A \rightarrow \omega \), denoted by \( |r| \), is \( |r| = |\omega| + 1 \). The size of \( G \), denoted by \( |G| \), is defined as the sum of sizes of all rules in \( P \) or the size of the set \( N \cup T \), whichever is larger.

The derivation by rule \( r \) relation, denoted by \( \overset{r}{\Rightarrow} \) is defined on \((N \cup T)^*\) such that
\[
\alpha A \beta \overset{r}{\Rightarrow} \alpha \omega \beta
\]
for all \( \alpha, \beta \in (N \cup T)^* \), and \( r = A \rightarrow \omega \in P \). If no ambiguity arises, the relation may be denoted by \( \overset{r}{\Rightarrow} \). The derivation by rule string \( \pi \) relation on \((N \cup T)^*\), denoted \( \overset{\pi}{\Rightarrow} \), is defined recursively as follows:
\[
\overset{\pi}{\Rightarrow} = \left\{ \begin{array}{ll}
\{ (\alpha, \alpha) | \alpha \in (N \cup T)^* \} & \pi = \epsilon \\
\overset{r}{\Rightarrow} \overset{\pi}{\Rightarrow} & \pi = r\pi'
\end{array} \right.
\]
If no ambiguity arises, the relation may be denoted by \( \overset{\pi}{\Rightarrow} \). A loop in \( G \) is a non-empty rule string \( \pi \) such that for some nonterminal \( A \in N \), \( A \overset{\pi}{\Rightarrow} A \). A rule string that does not contain a loop is said to be loopless. The derivation relation, denoted by \( \Rightarrow \), consists of all pairs in \( \cup_{r \in P} \overset{r}{\rightarrow} \). Both \( \Rightarrow * \) and \( \Rightarrow + \) are pronounced “derives”. A sequence of strings over \((N \cup T)^*\) \( (\gamma_0, \ldots, \gamma_n) \), \( n \in \mathbb{N} \), is a derivation of length \( n \) of \( \gamma_n \) from \( \gamma_0 \) in \( G \) if \( \gamma_i \overset{\pi}{\Rightarrow} \gamma_{i+1} \) for \( i = 0 \ldots n - 1 \). A nonterminal is said to be nullable if it derives the empty string. A production rule is said to be nullable if its RHS derives the empty string. The language generated by \( \alpha \in (N \cup T)^* \), denoted by \( L(\alpha) \), consists of all terminal strings \( x \in T^* \) such that \( \alpha \overset{*}{\Rightarrow} x \). The language generated by a grammar \( G \) is \( L(G) = L(S) \).

The rightmost derivation by rule \( r \) relation, denoted by \( \overset{\pi}{\Rightarrow} \) is defined on \((N \cup T)^*\) such that
\[
\alpha A x \overset{r}{\Rightarrow} \alpha \omega x
\]
for all \( \alpha \in (N \cup T)^* \), \( x \in T^* \) and \( r = A \rightarrow \omega \in P \). If no ambiguity arises, the relation may be denoted by \( \overset{\pi}{\Rightarrow} \). The notions of rightmost derivation by rule string \( \pi \) relation, rightmost loop, rightmost derivation relation, and rightmost derivation are defined similarly to their non-rightmost counterparts. A CFG \( G \) is said to be ambiguous if there exist rule strings \( \pi_1 \neq \pi_2 \) such that for some terminal string \( x \)
\[
S \overset{\pi_1}{\Rightarrow} x \text{ and } S \overset{\pi_2}{\Rightarrow} x
\]
A string of rules $\pi \in P^+$ is a right parse of $w \in T^*$ in $G$ if $S^{m_1^N}$. FIRST$_k(\alpha)$ denotes the set $\{k : x \mid x \in L(\alpha)\}$. Let $G(N, T, P, S)$ be a CFG. A symbol $X \in N \cup T$ is said to be useful if $S \Rightarrow^* \alpha X \beta \Rightarrow^* w$ for some $w \in T^*$ and $\alpha, \beta \in (N \cup T)^*$. Otherwise $X$ is useless. $G$ is said to be reduced if it contains no useless symbols. Unless explicitly specified otherwise, all CFGs considered hereafter are reduced.

### 2.4 Lists, Sets and Maps

The following data structures and notation are used in algorithm specifications. An ordered list is denoted by a sequence of comma-separated elements enclosed by angle brackets (e.g., $\langle e_1, e_2, e_3 \rangle$). An empty list is denoted by $\langle \rangle$. In the following let $l, l_1, l_2, \ldots, l_m$ denote lists and let $n \in \mathbb{N}$. The length of list $l$ is denoted by $|l|$. PUSH$(l_1, l_2)$ adds all the elements of $l_2$ to the end of $l_1$. POP$(l_1, l_2, \ldots, l_m, n)$ removes the last $n$ elements from each of the lists $l_1, l_2, \ldots, l_m$. TOP$(l_1, n)$ returns a new list which consists of the last $n$ elements of $l_1$. SHIFT$(l_1, l_2)$ adds all the elements of $l_2$ to the beginning of $l_1$. UNSHIFT$(l, n)$ removes the first $n$ elements from the list $l$. The $n^{th}$ element of $l$ is denoted by $l[n]$ (e.g., $\langle e_1, e_2, e_3 \rangle[2] = e_2$). If $|l_1| = |l_2| = n$, the Hadamard product of $l_1$ and $l_2$, denoted by $l_1 \times l_2$ is defined as the list $\langle \langle l_1[1], l_2[1] \rangle, \langle l_1[2], l_2[2] \rangle, \ldots, \langle l_1[n], l_2[n] \rangle \rangle$.

A set is denoted by a sequence of comma-separated elements enclosed by curly brackets (e.g., $\{e_1, e_2\}$). An empty set is denoted by $\{\}$ or $\emptyset$. A map is denoted by a set of comma-separated key-value mappings (e.g., $\{k_1 \mapsto v_1, k_2 \mapsto v_2, \ldots, k_n \mapsto v_n\}$). The value mapped to the key $k$ of map $m$ is denoted by $m[k]$. The empty map is denoted by $\{\mapsto\}$.

### 2.5 LR($k$) Parsing

The majority of work presented in this thesis is based on traditional LR parsing, due to Knuth [25]. Familiarity of the reader with LR concepts is therefore required before the details of the AMG model can be discussed. To this end, we provide the following overview of LR parsing, along with some definitions and denotation which will be used later in this paper. We concentrate only on techniques that relate to LR($k$) parsing where $k > 0$. For a more complete and thorough coverage of the subject see [1], [2], and [43].

An LR parser examines its input string from left to right, one symbol at a time. The context of the parser within the parsing process is maintained using a stack of states, the state at the top of which is called the current state. An LR parser is driven by its parse table, which has an ACTION and a GOTO components. The ACTION table determines, on the basis of the current state of the parser and the current input symbol, the action for the parser to perform. An action can be either a Shift, Reduce or Accept. The GOTO table determines the new state which the parser should assume following the execution of an action. The action $\langle$Shift$\rangle$ means that the parser has advanced one step in recognizing the RHS of rules where the next input symbol is expected. The parser obtains its new state from the GOTO table according to its current state and input symbol, and pushes it on the state stack. The action $\langle$Reduce, $r$$\rangle$ means that the parser has recognized the entire RHS of the production rule $r$ and has advanced one step in recognizing the RHS of rules where the LHS nonterminal of $r$ is expected. It obtains its new state by first discarding all the states which correspond to the recognized RHS from the top of the parse stack and then pushing the state encoded in the GOTO table under the state uncovered at the
top of the stack (after discarding RHS states) and the LHS nonterminal symbol of the recognized rule. The action \( \langle \text{Accept} \rangle \) means that the whole input has been recognized as a sentence of the language of the parser. The sequence of rules reduced by an LR parser until it encounters an Accept action is a right parse of the accepted sentence with respect to the grammar for which the parser’s parse table was constructed. If at some point of the parsing, none of the above parse actions can be applied, the input sentence is not in the parser’s language.

\[
\text{LrParse}(k, \text{startState}, x)
\]

1. \( \text{StateStack} \leftarrow (\text{startState}) \)
2. \( \text{while True do} \)
3. \( (q) \leftarrow \text{Top}(\text{StateStack}, 1) \)
4. \( \text{if exists action in ACTION}(q, k : x) \text{ then} \)
5. \( \text{if action} = (\text{Shift}) \text{ then} \)
6. \( \text{Push}(\text{StateStack}, (\text{Goto}(q, 1 : x))) \)
7. \( x \leftarrow x : ([x] - 1) \)
8. \( \text{else if action} = (\text{Reduce}, A \rightarrow \omega) \text{ then} \)
9. \( \text{Pop}(\text{StateStack}, |\omega|) \)
10. \( (q) \leftarrow \text{Top}(\text{StateStack}, 1) \)
11. \( \text{Push}(\text{StateStack}, (\text{Goto}(q, A))) \)
12. \( \text{else if action} = (\text{Accept}) \text{ then} \)
13. \( \text{exit "Success"} \)
14. \( \text{else} \)
15. \( \text{exit "Syntax error"} \)

Figure 2.1: LR\((k)\) parsing routine

Figure 2.1 lists an LR\((k)\) parsing algorithm, \textit{LrParse}. The algorithm consists of a single routine which takes the following arguments. \( k \) is positive integer stating the number of input symbols to be used when consulting the ACTION table, \textit{startState} is the state used to determine the first action of the parser, and \( x \) is the string to be parsed. On each iteration of the algorithm the parser consults the ACTION table and executes a single parse action. When encountered with an Accept action, the parser terminates reporting “Success”. If at some point of parsing no parse action is applicable, the parser terminates reporting a “Syntax error”. The parser is said to operate deterministically if every entry of its ACTION table consists of at most one action. Otherwise, whenever more than a single action is possible, the parser nondeterministically executes one of the possible actions. An input sentence is said to be in the language of the parser if there exists a computational path of the algorithm where the sentence is accepted.

The table generated by an LR parse table generator is a concise representation of an intermediate structure built by the generator called a \textit{prefix-automaton}. The prefix-automaton is a directed single rooted graph of \textit{item sets}, each representing a stage in the parsing process from which applicable parse actions can be deduced. Graph edges, annotated by vocabulary symbols, represent valid progressions of the parser from one parsing stage to another such that all item sets are reachable from the root of the graph. Each item set corresponds to a parser state which uniquely identifies it. In the sequel we shall use the term \textit{state} when referring to the state of a parser as well as for the item set which corresponds to it.

Each item of an LR\((k)\) state (for \( k > 0 \)) is a construct of the form \([A \rightarrow \alpha \cdot \beta, y]\) where \( A \rightarrow \alpha \beta \) is a production rule and \( y \) is a terminal string called the \textit{lookahead} of the item. Intuitively, it indicates that an input string derivable from \( \alpha \) has been read, and if the parser further reads a string derivable from \( \beta \), the rule \( A \rightarrow \alpha \beta \) could be reduced, providing that at that point the next \( k \) input symbols of the parser form the string \( y \).
Definition 2.1 (LR(k)-validity). Let \( G(N, T, P, S) \) be a CFG. An item \([A \rightarrow \alpha \bullet \beta, y]\) is said to be LR(k)-valid for a string \( \gamma \in (N \cup T)^* \) if there exist strings \( \delta \in (N \cup T)^* \) and \( z \in T^* \) such that
\[
S \Rightarrow^* \delta A z \Rightarrow^* \delta \alpha \beta z = \gamma \beta z, \quad \text{and} \quad y = k : z
\]

We denote the set of all LR(k)-valid items for a sentential form \( \gamma \in (N \cup T)^* \) in \( G \) by \([\gamma]_G^k\). We denote the set of all LR(k)-valid item sets in \( G \) by \([G]^k\). A string of item sets is called a state string. The state string that corresponds to \( \gamma \in (N \cup T)^* \) in \( G \), denoted by \( S_\gamma^G \), is defined as
\[
S_\gamma^G = [0 : [\gamma]^k_G][1 : [\gamma]^k_G] \cdots [|\gamma| : [\gamma]^k_G]
\]

The state at the root of the prefix-automaton of a grammar \( G \) (aka start state) is \([\varepsilon]^k_G\). All other states consist of all items that are LR(k)-valid for the string obtained by concatenating the symbols annotating the edges of the paths leading to them from the start state.

```plaintext
Figure 2.2: LR(k) parser generation routines

A traditional algorithm for generating a prefix-automaton for a CFG \( G(N, T, P, S) \) is listed in Figure 2.2. An LR(k) state \( q \) is represented as a structure where \( q.items \) is a
```

```plaintext
GeneraterParser_k()
1 \( q_0 \leftarrow \text{CreateState}_k([\{S' \rightarrow \bullet S$, $\varepsilon]\}) \)
2 \( \text{work} \leftarrow \{q_0\} \)
3 \( Q \leftarrow \{q_0\} \)
4 \( \text{while exists} \ q \ \text{in} \ \text{work} \ \text{do} \)
5 \( \text{work} \leftarrow \text{work} \setminus \{q\} \)
6 \( \text{for each} \ X \in N \cup T \ \text{do} \)
7 \( q' \leftarrow \text{CreateState}_k(\text{Kernel}(q, X)) \)
8 \( \text{if exists} \ q'' \ \text{in} \ Q \ \text{where} \ q''.items = q'.items \ \text{then} \)
9 \( q.edges[X] \leftarrow q'' \)
10 \( \text{else if} \ q'.items \neq \emptyset \ \text{then} \)
11 \( \text{work} \leftarrow \text{work} \cup \{q'\} \)
12 \( Q \leftarrow Q \cup \{q'\} \)
13 \( q.edges[X] \leftarrow q' \)
14 \( \text{return} \ q_0 \)

Kernel(state, X)
1 \( \text{kernel} \leftarrow \{\} \)
2 \( \text{for each} \ [A \rightarrow \alpha \bullet X\beta, \Theta] \ \text{in} \ state.items \ \text{do} \)
3 \( \text{kernel} \leftarrow \text{kernel} \cup \{[A \rightarrow \alpha X \bullet \beta, \Theta]\} \)
4 \( \text{return} \ \text{kernel} \)

CreateState_k(kernel)
1 \( \text{state} \leftarrow \text{new} \)
2 \( \text{state.items} \leftarrow \text{kernel} \)
3 \( \text{state.edges} \leftarrow \{\Rightarrow\} \)
4 \( \text{do} \)
5 \( \text{for each} \ [A \rightarrow \alpha \bullet X\beta, \Theta] \ \text{in} \ state.items \ \text{do} \)
6 \( \text{if} \ X \in N \ \text{then} \)
7 \( \Theta' \leftarrow \bigcup_{y \in \Theta} \text{First}_k(\beta y) \)
8 \( \text{for each} \ X \rightarrow \omega \ \text{in} \ P \ \text{do} \)
9 \( \text{if exists} \ [X \rightarrow \bullet \omega, \Theta''] \ \text{in} \ state.items \ \text{then} \)
10 \( \Theta'' \leftarrow \Theta'' \cup \Theta' \)
11 \( \text{else} \)
12 \( \text{state.items} \leftarrow \text{state.items} \cup \{[X \rightarrow \bullet \omega, \Theta']\} \)
13 \( \text{while} \ \text{state.items} \ \text{is modified} \)
14 \( \text{return} \ \text{state} \)
```

Technion - Computer Science Department - M.Sc. Thesis MSC-2010-11 - 2010
set of items of the form $[A \to \alpha \bullet \beta, \Theta]$ representing the set $\{[A \to \alpha \bullet \beta, y] | y \in \Theta\}$ of LR($k$)-items, and $q$.edges is a map which represents annotated edges connecting the state to other states of the automaton.

The routine `CREATESTATE_k` takes a set of items (kernel) that are LR($k$)-valid for some string. It then performs a transitive closure operation which results with additional items that are LR($k$)-valid for the same string, and returns a new state which consists of all these items.

The routine `KERNEL` takes a state consisting of all items that are LR($k$)-valid for some string $\gamma$, and a vocabulary symbol $X$. It returns a set of (kernel) items that are LR($k$)-valid for the string $\gamma X$ such that if provided as input to the `CREATESTATE_k` routine, the resulting state would consist of all LR($k$)-valid items for the string $\gamma X$.

Given a context-free grammar $G(N,T,P,S)$, the routine `GENERATEPARSER_k` generates a complete prefix-automaton of $k$-length lookahead strings for a modified grammar $G'$ called the $\$$-augmented grammar for $G$. Formally, this grammar is defined as

$$G'(N \cup \{S'\}, T \cup \{$, P \cup \{S' \rightarrow S$$\}, S')$$

where the symbols $\$$ and $S'$ are assumed not to appear in $N \cup T$. By doing so, the start symbol is guaranteed not to derive itself non trivially ($S' \Rightarrow^+ S'$) — a fact which prevents the creation of deterministic LR($k$) parsers for some ambiguous grammars (e.g., $S \rightarrow a | S$). The symbol $\$$ following $S$ at the RHS of the rule $S' \rightarrow S\$$ is called the right endmarker. It marks the end of the parser’s input stream and must be the last symbol in the input sentence ($x$) provided to the LRPARSE routine. Choosing a symbol $\$$ that is not of the grammar’s vocabulary guarantees that the right endmarker does not appear anywhere in the input string. The routine begins by creating the start state and proceeds by generating all states reachable from it until no new state can be created. Each state $q$ is connected to the states reachable from it by means of directed edges represented as the mapping $q$.edges[$X$] = $q'$ where $X$ is the vocabulary symbol annotating the edge and $q'$ is the target state. The start state of the automaton is created using the single item $[S' \rightarrow \bullet S\$$, \varepsilon]$. The set $Q$ contains all the nonempty states created by the routine. On each iteration of the main loop of the algorithm (lines 4-13), new states and edges are computed from a single existing state. This is accomplished by attempting to extract a new (kernel) set of items for every possible symbol of the ($\$$-augmented) grammar’s vocabulary, and augmenting it to a complete state. Empty states are discarded, while others are either new states (lines 11-13), or existing states to which edges are added (lines 8-9). The work set is used to contain all new states that have not yet been processed by the main loop. Every new state, starting with the start state, is added to the work set upon creation, and removed once processed. When no more states are left in the work set, the routine terminates.

ACTION and GOTO table components can be computed directly from the states of the prefix-automaton as follows. An item

$$[A \to \alpha \bullet t\beta, y] \in q$.items$$

where $t \in T$ implies that $\langle$SHIFT$\rangle \in$ ACTION($q,tx$) for every $x \in$ FIRST$_{k-1}(\beta y)$. An item

$$[A \to \alpha \bullet, y] \in q$.items$$

where $A \neq S'$ implies that $\langle$REDUCE, $A \rightarrow \alpha \rangle \in$ ACTION($q, y$). An item

$$[S' \rightarrow S\$$\bullet, $\varepsilon] \in q$.items
implies that \( \langle \text{Accept} \rangle \in \text{ACTION}(q, \varepsilon) \). Finally, an item

\[
A \rightarrow \alpha \bullet X \beta, y \in q.\text{items}
\]

implies that \( \text{GOTO}(q, X) = q.\text{edges}[X] \)

The more lookahead symbols are used to construct a prefix-automaton, the more CFGs there are for which deterministic LR parsers can be constructed. For example, if for a given grammar a prefix-automaton with a lookahead of \( k = 2 \) symbols contains a state \( q \) for which

\[
\langle \text{Reduce}, A \rightarrow \alpha \rangle \in \text{ACTION}(q, ab)
\]

and

\[
\langle \text{Reduce}, B \rightarrow \beta \rangle \in \text{ACTION}(q, ac)
\]

then a prefix-automaton constructed with a single lookahead symbol for the same grammar would contain a state \( q' \) where the two actions

\[
\langle \text{Reduce}, A \rightarrow \alpha \rangle, \langle \text{Reduce}, B \rightarrow \beta \rangle
\]

are in \( \text{ACTION}(q', a) \). A CFG is said to be LR\((k)\) if a deterministic LR\((k)\) parser can be generated for it.
Chapter 3

LR-based Grammar Adaptivity

Adaptive grammars are grammatical formalisms that allow changes to their set of production rules during derivation. At each derivation step, a new set of rules applicable for rewriting in following derivation steps is determined, according to specifications provided by the grammar writer.

An important property of CFG derivations is that they are insensitive to the order in which production rules are applied. However, this is not necessarily the case with CFG based adaptive grammars, as the application of each rule may trigger the removal of other rules, thereby preventing them from being applied in subsequent derivation steps. Consequently, most CFG based adaptive grammars impose an order on the application of production rules during derivation.

Naturally, CFG-based adaptive grammars that impose an order on rule application also restrict the type of parsers that can be used to produce their derivations (e.g., bottom-up, top-down, the order in which the input string is processed, etc.). We chose an LR (the input is read from left to right, the derivation is rightmost, i.e., bottom-up) based formalism, which dictates a trivial order restriction (rightmost derivation) and an efficient, well-known (LR) parsing algorithm. Other LR-based adaptive formalisms previously proposed are Modifiable Grammars [8, 7, 9], Evolving Grammars [10], and Dynamic Grammars [5].

Informally, LR-based adaptive formalisms can be generalized as consisting of a possibly infinite vocabulary partitioned into a terminal alphabet $T$ and a nonterminal alphabet $N$, a start symbol $S \in N$, a base set of CFG production rules $R_0 \subseteq N \times (T \cup N)^*$, and an adaptation function $\Delta$ which determines the set of production rules available at each derivation step. Various LR-based adaptive formalisms differ in the way in which production rules can be applied, and in the arguments provided to the adaptation function at each derivation step.

We proceed by examining various notions of LR-based grammar adaptivity and characterize them as a progression of restrictions on derivations, starting with unrestricted LR-based adaptivity.

3.1 Unrestricted LR-based Adaptivity

Definition 3.1 (Unrestricted LR-based adaptive grammar). A terminal string $w$ is in the language of an unrestricted LR-based adaptive grammar $G(N, T, R_0, S, \Delta)$ iff there
exists a (rightmost) derivation 

\[(S, R_n) = (\alpha_n A_n x_n, R_n) \Rightarrow \]
\[(\alpha_n \omega_n x_n, R_{n-1}) = (\alpha_{n-1} A_{n-1} x_{n-1}, R_{n-1}) \Rightarrow \]
\[(\alpha_{n-1} \omega_{n-1} x_{n-1}, R_{n-2}) = (\alpha_{n-2} A_{n-2} x_{n-2}, R_{n-2}) \Rightarrow \]
\[\vdots \]
\[(\alpha_2 \omega_2 x_2, R_1) = (\alpha_1 A_1 x_1, R_1) \Rightarrow \]
\[(\alpha_1 \omega_1 x_1, R_0) = (w, R_0) \]

such that \( \alpha_i \in (T \cup N)^* \), \( x_i \in T^* \), \( R_i \subseteq N \times (T \cup N)^* \), \( r_i = A_i \rightarrow \omega_i \in R_{i-1} \), and \( R_i = \Delta(R_{i-1}, r_i) \) for all \( i = 1, \ldots, n \). \( \square \)

From a generative point of view, the derivation begins with the start symbol \( S \) and an arbitrary set of production rules \( (R_n) \), and ends with a terminal string \( (w) \) and the base set of production rules \( R_0 \). Reading the derivation in reverse order provides a right parser’s perspective of the derivation. Parsing begins with an empty parse stack, the base set of production rules and the terminal string as input, and ends with the start symbol at the parser’s stack and an empty input string. Note that the production rule used in each derivation step is taken from the set of production rules to the right of the \( \Rightarrow \) symbol.

**Example 3.1.** Consider the adaptive grammar

\[ G_1(N, T, R_0, S, \Delta) \]
\[ N = \{ S, A, B, C \} \]
\[ T = \{ a, b, c \} \]
\[ R_0 = \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c \} \]
\[ \Delta(R, r) = \begin{cases} 
R \cup \{ S \rightarrow C \} & r = C \rightarrow c \\
R \cup \{ B \rightarrow b \} & r = A \rightarrow a \\
R & \text{otherwise} 
\end{cases} \]

According to the derivation of Definition 3.1, the language of \( G_1 \) is \( \{ a, c, ab \} \) as implied by the following derivations

(a) 

\[(S, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c, S \rightarrow C \}) \Rightarrow^{B} C \]
\[(C, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c, S \rightarrow C \}) \Rightarrow^{C} c \]
\[(c, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c \}) \]

(b) 

\[(S, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c, B \rightarrow b \}) \Rightarrow^{S} AB \]
\[(AB, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c, B \rightarrow b \}) \Rightarrow^{B} b \]
\[(Ab, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c, B \rightarrow b \}) \Rightarrow^{A} a \]
\[(ab, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c \}) \]

(c) 

\[(S, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c, B \rightarrow b \}) \Rightarrow^{S} AB \]
\[(AB, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c, B \rightarrow b \}) \Rightarrow^{B} \varepsilon \]
\[(A, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c, B \rightarrow b \}) \Rightarrow^{A} a \]
\[(a, \{ S \rightarrow AB, B \rightarrow \varepsilon, A \rightarrow a, C \rightarrow c \}) \]
As implied by the above example, an unrestricted adaptive parser must “blindly” attempt to reduce each available rule until the start symbol is reached. Clearly, such a parser can hardly be put to practical use. Both Modifiable Grammars [8, 7, 9] and Evolving Grammars [10] reflect this notion of LR-based adaptivity. Modifiable Grammars consist of a CFG and a recursive transducer that halts on all inputs. A bottom-up variant of the model (BUMG) is described in [7]. Upon rule reduction, the full sequence of rules previously reduced by the parser is provided to the recursive transducer, which outputs a list of rules to be added and removed from the grammar. Evolving Grammars consist of a CFG and a recursive transducer. Upon rule reduction, the transducer is provided with the reduced rule, and results with a set of nonterminals and rules to be added to the grammar. In the original publication of the formalism, only rule additions were supported. In a later presentation of the capabilities and applications of a corresponding Dynamic Parser [35], examples reveal that rules may be specified in the context of grammatical scopes that can be added and subsequently removed from the grammar, thus facilitating rule deletion.

3.2 Root-Driven LR-based Adaptivity

Next, we define a more restricted form of adaptive grammars for which more efficient parsers can be constructed. The restriction will allow a parser to “reduce” at each parse step only those rules which, according to its current set of production rules, may lead to a successful parse (that is, reduction of the entire input string to the start symbol).

**Definition 3.2** (Root-driven LR-based adaptive grammar). A terminal string $w$ is in the language of a root-driven LR-based adaptive grammar $G(N, T, R_0, S, \Delta)$ iff there exists a (rightmost) derivation

$$
(S, R_n) = (\alpha_nA_n x_n, R_n) \xrightarrow{R_n} \\
(\alpha_n \omega_n x_n, R_{n-1}) = (\alpha_{n-1} A_{n-1} x_{n-1}, R_{n-1}) \xrightarrow{R_{n-1}} \\
(\alpha_{n-1} \omega_{n-1} x_{n-1}, R_{n-2}) = (\alpha_{n-2} A_{n-2} x_{n-2}, R_{n-2}) \xrightarrow{R_{n-2}} \\
\vdots \\
(\alpha_2 \omega_2 x_2, R_1) = (\alpha_1 A_1 x_1, R_1) \xrightarrow{R_1} \\
(\alpha_1 \omega_1 x_1, R_0) = (w, R_0)
$$

where for all $i = 1, \ldots, n$ there exists a terminal string $y_i$ such that

$$S \xrightarrow{R_{i-1}} \alpha_i A_i y_i \xrightarrow{R_{i-1}} \alpha_i \omega_i y_i$$

An important property of the derivation of Definition 3.2 (and that of Definition 3.1) is that at each derivation step, the terminal string $x_i$ appearing at the suffix of the rewritten string (which corresponds to the remaining input of the parser) need not necessarily be considered valid with respect to the rule set associated with the rewritten string ($R_{i-1}$). It may even consist of terminal symbols that do not appear in any of the production rules in $R_{i-1}$. This is reflected by the terminal string $y_i$ that is completely unrelated to $x_i$ for all $i = 1, \ldots, n$. As a result, root-driven adaptive parsers cannot rely on lookahead techniques for resolving nondeterminism, and must follow each and every parse action that potentially leads to acceptance of $w$ with respect to its current set of applicable production rules.
As a root-driven LR-based adaptive grammar, the language of the grammar $G_1$ from Example 3.1 is narrowed to \{a, ab\}. The terminal string c is no longer in the language of $G_1$ since in the last derivation step of derivation (a) above, there exists no terminal string $y \in T^*$ such that
$$S \xrightarrow{R_0} Cy \xrightarrow{R_0} cy$$
and there exists no other rightmost derivation of c from the start symbol S. Consequently, a root-driven adaptive parser will never attempt to reduce the rule $C \rightarrow c$ when encountered with the input terminal c, because according to its current set of production rules, this reduction will not lead to a successful parse.

The Dynamic Grammars devised by Boulier [5] reflect this notion of grammar adaptivity. A corresponding Dynamic Parser relies on an LR(0) prefix-automaton for determining parse actions. In order to resolve LR-conflicts, the parser simulates all subsequent parse steps involving at most $k > 0$ shifted terminals. Throughout this process, grammar modifications are applied and reverted as the different computation paths are explored.

### 3.3 Lookahead LR-based Adaptivity

We now present a novel, even more restricted notion of grammar adaptivity called Lookahead LR-based adaptivity which allows corresponding parsers to rely on lookahead symbols computed from their current set of production rules for resolving nondeterminism at each parse step.

**Definition 3.3** (Lookahead LR-based adaptive grammar). A terminal string $w$ is in the language of a Lookahead LR-based adaptive grammar $G(N, T, R_0, S, \Delta, k)$ iff there exists a (rightmost) derivation

$$(S, R_n) = (\alpha_n A_n x_n, R_n) \xrightarrow{R_n}$$

$$(\alpha_n \omega_n x_n, R_{n-1}) = (\alpha_{n-1} A_{n-1} x_{n-1}, R_{n-1}) \xrightarrow{R_{n-1}}$$

$$(\alpha_{n-1} \omega_{n-1} x_{n-1}, R_{n-2}) = (\alpha_{n-2} A_{n-2} x_{n-2}, R_{n-2}) \xrightarrow{R_{n-2}}$$

$$\vdots$$

$$(\alpha_2 \omega_2 x_2, R_1) = (\alpha_1 A_1 x_1, R_1) \xrightarrow{R_1}$$

$$(\alpha_1 \omega_1 x_1, R_0) = (w, R_0)$$

where $k > 0$ and for all $i = 1, \ldots, n$ there exists a terminal string $y_i$ such that

$$S \xrightarrow{R_{i-1}} \alpha_i A_i y_i \xrightarrow{R_{i-1}} \alpha_i \omega_i y_i \text{ and } k : y_i = k : x_i$$

As a lookahead LR-based adaptive grammar, the language of the grammar $G_1$ from Example 3.1 is further narrowed to \{a\} for all $k > 0$. The terminal string ab is no longer in the language of $G_1$ since in the last derivation step of derivation (b) above, there exists no string $y \in T^*$ such that

$$S \xrightarrow{R_0} Ay \xrightarrow{R_0} ay \text{ and } k : y = k : b$$

and there exists no other rightmost derivation of ab from the start symbol S. Consequently, an adaptive parser with $k > 0$ lookahead symbols will report a syntax error when encountered with the input ab, because according to its current set of production rules, a reduction of the input terminal a by the rule $A \rightarrow a$ cannot lead to a successful parse, given that the next input terminal is b.
It is evident from the derivation relation of Definition 3.3 that there exists a trade off between the amount of lookahead symbols used and the adaptivity of the grammar: the larger $k$ is, the easier it is to construct a grammar for which the corresponding parser is deterministic, but the adaptivity of the grammar is reduced, as less derivations are permissible and therefore less opportunities for triggering adaptive changes can be exploited. Since the same grammar may specify different languages with respect to different lookahead amounts, the lookahead amount of a grammar should be included as part of its specification.

Adaptive Multi-pass Grammars (AMGs) is the first LR-based adaptive grammar formalism that reflects this notion of grammar adaptivity. Consequently, we are able to develop an efficient LR($k$)-based parsing algorithm for AMG languages. Note that the parsers suggested in [7, 10] for Modifiable Grammars and Evolving Grammars also rely on lookahead-based techniques for resolving parser nondeterminism (i.e., LR conflicts) and are thus inconsistent with their unrestricted adaptive formalisms.
Chapter 4

Adaptive Multi-pass Grammars

In this chapter, Adaptive Multi-pass Grammars (AMGs) are introduced and formally defined. We first give an informal overview of the formalism, providing a conceptual framework into which the formal elements of the model, as defined in Section 4.2, can be placed. We then prove that the AMG model defines the class of recursively enumerable sets, and conclude the chapter by considering the relationship between AMGs and CFGs, and showing that CFGs are a special case of AMGs.

4.1 Overview

An adaptive multi-pass grammar (AMG) is an adaptive formalism similar to Lookahead LR-based Adaptive Grammars of Section 3.3. It is defined over a finite terminal alphabet $T$ and a possibly infinite nonterminal alphabet $N$. Sentential forms are strings of annotated symbols of the form $\langle X, w \rangle$ where $X \in T \cup N$ is a symbol and $w \in T^*$ is its annotation, and if $X$ is a terminal $w = X$. An annotated terminal $\langle a, a \rangle$ can be denoted by the terminal symbol $a$.

Similarly to many other grammatical formalisms, the AMG model defines a derivation relation involving production rules, as the basis for its language definition. An AMG production rule $r$ is a construct of the form $\langle A, f_r \rangle \rightarrow X_1X_2 \cdots X_n$ where $A \in N$, $X_i \in T \cup N$ for all $i = 1, \ldots, n$, and $f_r : (T^*)^n \rightarrow T^*$ is a total function called the annotation function associated with $r$. The skeleton of $r$, denoted $\xi(r)$, is defined as the CFG production rule

$$A \rightarrow X_1X_2 \cdots X_n$$

A production rule $r$ can be used to rewrite an annotated nonterminal $\langle A, w \rangle$ as an annotated symbol string $\langle X_1, w_1 \rangle \langle X_2, w_2 \rangle \cdots \langle X_n, w_n \rangle$ provided that

$$\xi(r) = A \rightarrow X_1X_2 \cdots X_n \text{ and } w = f_r(w_1, w_2, \ldots, w_n)$$

Example 4.1. The AMG production rule

$$r = \langle A, \tau_1 \cdot \tau_2 \cdot \tau_3 \rangle \rightarrow ABc$$

can be used to rewrite $\langle A, abbc \rangle$ as $\langle A, a \rangle \langle B, bb \rangle \langle c, c \rangle$ since $\xi(r) = A \rightarrow ABc$ and

$$f_r(a, bb, c) = a \cdot bb \cdot c = abbc$$
An AMG production rule \( r \) can be denoted by its underlying skeleton, in which case, the annotation function \( f_r(\tau_1, \tau_2, \ldots, \tau_n) = \tau_1 \cdot \tau_2 \cdots \tau_n \) is assumed. For example, the AMG production rule of Example 4.1 can be denoted by

\[
A \rightarrow ABc
\]

**Example 4.2.** Figure 4.1 lists an AMG for the context-sensitive language

\[
L_{DU} = \{a^n b^m c^n d^m | n, m > 0\}
\]

The language is reminiscent of the “declare before use” pattern common in many programming languages. In a given “program” \( a^k b^j c^k d^j \), the string \( a^k \) can be considered as the declaration of an identifier and the string \( c^k \) a reference to that identifier occurring later in the program. Similarly, the string \( d^j \) can be considered as a reference to the previously declared identifier \( b^j \).

\[
G_{DU}(N, T, R_0, S, \Delta, 1) :
\]

\[
N = \{ S, D_1, D_2, U_1, U_2, A, B, C, D \}
\]

\[
T = \{ a, b, c, d \}
\]

\[
R_0 = \{
\begin{align*}
  r_1 &= S \rightarrow D_1 D_2 U_1 U_2, \\
  r_2 &= D_1 \rightarrow A, \\
  r_3 &= D_2 \rightarrow B, \\
  r_4 &= U_1 \rightarrow C, \\
  r_5 &= U_2 \rightarrow D, \\
  r_{6,7} &= A \rightarrow Aa | a, \\
  r_{8,9} &= B \rightarrow Bb | b, \\
  r_{10,11} &= C \rightarrow Cc | c, \\
  r_{12,13} &= D \rightarrow Dd | d
\end{align*}
\]

\[
\Delta(R, r, \alpha) = \begin{cases} 
  \{ \{ r_{14} = U_1 \rightarrow c^{|w|} \}, \{ r_4 \} \} & r = r_2, \alpha = \langle A, w \rangle \\
  \{ \{ r_{15} = U_2 \rightarrow d^{|w|} \}, \{ r_5 \} \} & r = r_3, \alpha = \langle B, w \rangle \\
  (\emptyset, \emptyset) & \text{otherwise}
\end{cases}
\]

Figure 4.1: AMG \( G_{DU} \) for the language \( L_{DU} \)

The grammar \( G_{DU} \) consists of a nonterminal alphabet \( N \), a terminal alphabet \( T \), a base set of AMG production rules \( R_0 \), a start symbol \( S \), an adaptation function \( \Delta \), and a lookahead amount \( (k = 1) \). The derivation process of an AMG is best understood when considered from an LR parser’s perspective, starting with a string of annotated terminals and applying derivation steps in reverse order until the start symbol is reached. The parser starts with the base rule set \( R_0 \) which skeletons derive (by traditional CFG rightmost derivation) the language \( a^+ b^+ c^+ d^+ \). Using rules \( r_6 \) and \( r_7 \) it reduces the string \( a^n \) to the annotated nonterminal \( \langle A, a^n \rangle \) which is then reduced to \( \langle D_1, a^n \rangle \) using rule \( r_2 \). Upon each reduction, the parser checks its adaptation function \( \Delta(R, r, \alpha) \) for required modifications to its set of production rules. The function takes the current set of production rules \( (R) \), the reduced production rule \( (r) \), and the annotated symbols reduced from the top of parser’s stack which correspond to the symbol string written in the corresponding derivation step \( (\alpha) \). It returns a set of production rules to be added to the set of rules applicable for subsequent reductions, and a set of production rules to be removed. Consequently, Following the reduction by rule \( r_2 \), the parser replaces rule \( r_4 \) with the rule

\[
r_{14} = U_1 \rightarrow c^n
\]
since $\Delta(R_0, r_2, \langle A, a^n \rangle) = (\{r_{14}\}, \{r_4\})$. Next, the parser reduces the terminal string $b^m$ to $\langle D_2, b^m \rangle$ using rules $r_8, r_9$ and $r_3$, and replaces rule $r_5$ with the rule

$$r_{15} = U_2 \rightarrow d^m$$

As a result, the parser will terminate successfully only if its remaining input is the terminal string $c^n d^m$. Given such input, the parser continues by reducing the string $c^n$ to $\langle U_1, c^n \rangle$ using rule $r_{14}$, and reducing the string $d^m$ to $\langle U_2, d^m \rangle$ using rule $r_{15}$. Finally, it reduces the stack string $\langle D_1, a^n \rangle \langle D_2, b^m \rangle \langle U_1, c^n \rangle \langle U_2, d^m \rangle$ to the annotated nonterminal $\langle S, a^n b^m c^n d^m \rangle$ using rule $r_1$ and terminates successfully.

The parser that corresponds to the AMG $G_{UD}$ of Example 4.2 starts with an over-permissive set of production rules and continually adjusts it according to the contents of the parsed string. Since LR-parsers consume their input from left to right, this strategy falls short in situations where the entities that trigger grammatical changes occur after the point where these modifications should be applied. These situations, generally referred to in the literature as forward referencing, are quite common in programming languages. Well known examples include goto statements that can refer to labels that occur later in the program, and Java class members (fields, methods, etc.) that can be referenced within the class before their point of declaration. In order to handle these situations, the AMG model provides multi-pass production rules which, when viewed from an LR-parser’s perspective, trigger multiple left to right passes over the input string. These rules are distinguished from their “normal” (a.k.a single-pass) counterparts (see page 30) by the $\leftrightarrow$ symbol separating their LHS and RHS components, as in the multi-pass rule

$$r = \langle A, f_r \rangle \leftrightarrow X_1 X_2 \cdots X_n$$

Similarly to single-pass production rules, the multi-pass rule $r$ can be used to reduce a symbol string $\langle X_1, w_1 \rangle \langle X_2, w_2 \rangle \cdots \langle X_n, w_n \rangle$ to the annotated nonterminal $\langle A, w \rangle$ where $w = f_r(w_1, w_2, \ldots, w_n)$. However, reduction by a multi-pass rule triggers a side effect which causes the parser to remove the annotated nonterminal $\langle A, w \rangle$ from the top of its stack, and push its annotation $w$ to the beginning of its input stream. Thus, annotated symbols, annotation functions and multi-pass production rules, together constitute a powerful multi-pass mechanism: nonterminal annotations encode the terminal strings to be reparsed following reduction by a multi-pass production rule, annotation functions synthesize new nonterminal annotations from existing ones, and multi-pass production rules trigger the insertion of annotation strings to the beginning of the input stream, to be subsequently reparsed.

**Example 4.3.** The grammar $G_{UD}$ in Figure 4.2 uses a single multi-pass rule ($r_1$) to define the language $L_{UD} = L_{DU}$.

The language $L_{UD}$ captures the essence of the forward referencing problem, in the sense that the string $a^n b^m$ represents references to identifiers that are declared by the string $c^n d^m$ later in the program. Given the input string $a^j b^k c^n d^m$ a corresponding parser first reduces the string $a^j b^k c^n$ to the stack string $\langle U_1, a^j \rangle \langle U_2, b^k \rangle \langle D_1, c^n \rangle$ using its base set of production rules, and replaces the rule $r_2$ with the rule

$$r_{14} = U_1 \rightarrow a^n$$

It then reduces the string $d^m$ to $\langle D_2, d^m \rangle$ resulting with the stack string

$$\langle U_1, a^j \rangle \langle U_2, b^k \rangle \langle D_1, c^n \rangle \langle D_2, d^m \rangle$$
$G_{UD}(N, T, R_0, S, \Delta, 1)$:

$N = \{S, D_1, D_2, U_1, U_2, A, B, C, D\}$

$T = \{a, b, c, d\}$

$R_0 = \{\$

$\begin{align*}
    r_1 &= \langle S, \tau_1 \cdot \tau_2 \rangle \leftrightarrow U_1U_2D_1D_2, \\
    r_2 &= U_1 \rightarrow A, \\
    r_3 &= U_2 \rightarrow B, \\
    r_4 &= D_1 \rightarrow C, \\
    r_5 &= D_2 \rightarrow D, \\
    r_{6,7} &= A \rightarrow Aa|a, \\
    r_{8,9} &= B \rightarrow Bb|b, \\
    r_{10,11} &= C \rightarrow Cc|c, \\
    r_{12,13} &= D \rightarrow Dd|d \}
\end{align*}$

$\Delta(\Delta, r, \alpha) = \left\{ \begin{array}{ll}
    (\{r_{14} = U_1 \rightarrow a^{|w|}\}, \{r_2\}) & r = r_4, \alpha = \langle C, w \rangle \\
    (\{r_{15} = U_2 \rightarrow b^{|w|}\}, \{r_3\}) & r = r_5, \alpha = \langle D, w \rangle \\
    (\{r_{16} = S \rightarrow U_1U_2\}, \{r_1\}) & r = r_1 \\
    (\emptyset, \emptyset) & \text{otherwise}
\end{array} \right.$

Figure 4.2: AMG $G_{UD}$ (with a multi-pass rule) for the language $L_{UD}$

and replaces the rule $r_3$ with the rule

$r_{15} = U_2 \rightarrow b^m$

Then, using the multi-pass rule

$r_1 = \langle S, \tau_1 \cdot \tau_2 \rangle \leftrightarrow U_1U_2D_1D_2$

the parser reduces its stack string to the annotated nonterminal $\langle S, a^ib^k \rangle$ and replaces rule $r_1$ with the rule

$r_{16} = S \rightarrow U_1U_2$

Since $r_1$ is a multi-pass rule, the parser further removes the annotated nonterminal $\langle S, a^ib^k \rangle$ from its stack, and pushes the terminal string $a^ib^k$ back to the beginning of the input stream. Consequently, following the reduction by $r_1$, the parser is left with an empty stack and the string $a^ib^k$ as its input. However, its adapted set of production rules (namely rules $r_{14}$, $r_{15}$ and $r_{16}$) guarantees that it will successfully re-parse the input string if and only if $j = n$ and $k = m$.

4.2 Grammars, Derivations and Languages

In this section, formal definitions of the elements constituting AMGs are given, concluding with the definition of the language of an AMG. Throughout the following definitions $T$ denotes a terminal alphabet and $N$ denotes a nonterminal alphabet.

Definition 4.1 (Annotated symbol). An annotated symbol over $N \cup T$ is a construct of the form $\langle X, w \rangle$ where $X \in N \cup T$ is a symbol, $w \in T^*$ is its annotation, and $X \in T$ implies that $w = X$.

The denotations of a terminal $a$ and an annotated terminal $\langle a, a \rangle$ are interchangeable. That is, a terminal $a$ can be used to denote the annotated terminal $\langle a, a \rangle$ and vice versa.
A string of annotated symbols is called a *sentential form*. The set of all annotated symbols over \( N \cup T \) is denoted by \( A_{N,T} \). The set of all annotated terminals in \( A_{N,T} \) is denoted by \( A_T \).

**Definition 4.2** (Core). The core of an annotated symbol string
\[ \rho = \langle X_1, w_1 \rangle \langle X_2, w_2 \rangle \cdots \langle X_n, w_n \rangle \]
denoted \( \bar{\rho} \), is the symbol string \( \bar{\rho} = X_1X_2 \cdots X_n \).

**Definition 4.3** (Annotation function). An annotation function is a total function
\[ f : (T^*)^n \rightarrow T^*, n \geq 0 \]

**Definition 4.4** (Production rule). A production rule \( r \) over \( N \cup T \) is a construct of the form
\[ \langle A, f_r \rangle \oplus X_1X_2 \cdots X_n \]
where \( \oplus \in \{ \rightarrow, \leftrightarrow \} \), \( A \in N \), \( n \geq 0 \), \( f_r \) is an annotation function, and for all \( i = 1, \ldots, n \) \( X_i \in N \cup T \). If \( \oplus = \leftrightarrow \), \( r \) is called a multi-pass rule; otherwise it is called a single-pass rule.

The set of all multi-pass rules over \( N \cup T \) is denoted by \( R_{N,T}^{\rightarrow\rightarrow} \). The set of all single-pass rules over \( N \cup T \) is denoted by \( R_{N,T}^{\rightarrow\rightarrow} \). The set of all production rules is denoted by \( R_{N,T} = R_{N,T}^{\rightarrow\rightarrow} \cup R_{N,T} \).

**Definition 4.5** (Rule skeleton). Let \( r = \langle A, f_r \rangle \oplus X_1X_2 \cdots X_n \) be a rule. The skeleton of \( r \), denoted by \( \xi(r) \), is the CFG rule
\[ A \rightarrow X_1X_2 \cdots X_n \]
The skeleton of a rule set \( R \subseteq R_{N,T} \), denoted by \( \xi(R) \), is defined by
\[ \xi(R) = \{ \xi(r) | r \in R \} \]
and the skeleton of a rule string \( \pi \in R_{N,T}^* \), denoted by \( \xi(\pi) \), is defined recursively by
\[ \xi(\pi) = \begin{cases} \varepsilon & \pi = \varepsilon \\ \xi(r)\xi(\pi') & \pi = r\pi' \end{cases} \]

**Definition 4.6** (Adaptation function). An adaptation function over \( N \cup T \) is a computable function
\[ \Delta : \mathcal{P}(R_{N,T}) \times R_{N,T} \times A_{N,T}^* \rightarrow \mathcal{P}(R_{N,T})^2 \]
where
\[ \Delta(R, r, \alpha) = (R_a, R_r) \]
implies that \( R, R_a \), and \( R_r \) are finite, \( R_r \subseteq R \), \( r \in R \), and \( r = \langle A, f_r \rangle \oplus \bar{\alpha} \) for some nonterminal \( A \in N \).

The set \( R_a \) contains production rules to be added, and the set \( R_r \) contains production rules to be removed.

**Definition 4.7** (AMG). An adaptive multi-pass grammar (AMG) is a sextuple
\[ G(N, T, R_0, S, \Delta, k) \]
where \( N \) is a possibly infinite nonterminal alphabet, \( T \) is a finite terminal alphabet, \( R_0 \subseteq R_{N,T} \) is a finite set of rules called the base rule set, \( S \in N \) is a start symbol, \( \Delta \) is an adaptation function and \( k > 0 \) is a lookahead length specifier.
An AMG $G$ with lookahead length $k$ is said to be $AMG(k)$.

The AMG model defines a derivation (by rule) relation which reflects the derivation constraints of lookahead LR-based adaptive grammars (see Definition 3.3 in page 28) as well as the novel multi-pass mechanism that is unique to this model. The derivation relation is defined over $AMG$ configurations which represent both the rewritten sentential form and set of available rewrite (i.e., production) rules.

**Definition 4.8 (AMG Configuration).** Let $G(N, T, R_0, S, \Delta, k)$ be an AMG. An AMG configuration of $G$ is a pair

$$\langle \gamma, R \rangle$$

where $\gamma \in A^*_N,T$ is a sentential form and $R \subseteq R_{N,T}$ is a finite rule set.

The set of all AMG configurations of an AMG $G$ is denoted by $C_G$.

**Definition 4.9 (AMG derivation by rule relation).** Let $G(N, T, R_0, S, \Delta, k)$ be an AMG and let

$$r = \langle A, f_r \rangle \oplus X_1X_2 \cdots X_n$$

be a production rule. The adaptive multi-pass derivation by rule $r$ relation on $C_G$ in $G$, denoted by $\Rightarrow_G^r$, consists of all pairs

$$\langle (\rho(A, w)x, R'), (\rho(X_1, w_1) \cdots (X_n, w_n)x, R) \rangle \oplus = \rightarrow$$

$$\langle (pwx, R'), (\rho(X_1, w_1) \cdots (X_n, w_n)x, R) \rangle \oplus = \leftarrow$$

such that

$$\tag{4.1} r \in R, \rho \in A^*_N,T, x \in A^*_T, w = f_r(w_1, \ldots, w_n),$$

$$\tag{4.2} (R_a, R_r) = \Delta(R, r, \langle X_1, w_1 \rangle \cdots \langle X_n, w_n \rangle), R' = (R \cup R_a) \setminus R_r,$$

$$\tag{4.3} S \Rightarrow pA_{\xi(r)} \Rightarrow pX_1X_2 \cdots X_nz \text{ where } \pi \in \xi(R)^* \text{ and } k : z = k : \bar{x}$$

The rightmost derivation at the basis of the derivation relation (4.3) facilitates the construction of LR$(k)$ based parsers for recognizing AMG languages. From a parser’s perspective (i.e., considering AMG configuration pairs from right to left), $R$ represents the set of rules available for reduction, $x$ represents the remaining input, and $\rho(X_1, w_1) \cdots (X_n, w_n)$ is the parse stack. The parser then reduces the stack string $\langle X_1, w_1 \rangle \cdots \langle X_n, w_n \rangle$ using rule $r \in R$, to the annotated nonterminal $\langle A, w \rangle$ where $w = f_r(w_1, \ldots, w_n)$ (4.1). If $r$ is a multi-pass rule ($\oplus = \rightarrow$), the annotated nonterminal $\langle A, w \rangle$ is removed from the top of the parse stack, and its annotation $w$ is pushed back to the input stream. $R'$ represents the set of rules that are applicable for subsequent reductions, as determined by the adaptation function (4.2). The lookahead amount specifier $k$ specifies the amount of lookahead symbols used by the parser to determine the set of rules applicable for reduction (4.3) of which $r$ is a member.

An adaptive multi-pass derivation by a multi-pass rule $r \in R_{N,T}^k$ is denoted by $\Rightarrow_G^r$. An adaptive multi-pass derivation by a single-pass rule $r \in R_{N,T}^k$ is denoted by $\Rightarrow_G^r$.

**Definition 4.10 (AMG derivation by rule string relation).** Let $G(N, T, R_0, S, \Delta, k)$ be an AMG. The adaptive multi-pass derivation by rule string $\pi$ relation on $C_G$ in $G$, denoted by $\Rightarrow_G^\pi$, is defined recursively by

$$\Rightarrow_G^\pi = \left\{ \begin{array}{ll}
\{(\gamma, R), (\gamma, R) \} & \pi = \varepsilon \\
\Rightarrow_G^r & \pi = r\pi'
\end{array} \right.$$
Definition 4.11 (AMG derivation relation). Let $G(N,T,R_0,S,\Delta,k)$ be an AMG. The adaptive multi-pass derivation relation (or AMG derivation for short) on $C_G$ in $G$, denoted by $\Rightarrow_G$, is defined as

$$\Rightarrow_G = \bigcup_{r \in R_{N,T}} \frac{r}{G}$$

The reflexive transitive closure of $\Rightarrow_G$ is denoted by $\Rightarrow^*_G$. We say that $\gamma$ derives $\gamma'$ in $G$ if there exist rule sets $R,R' \subseteq R_{N,T}$ such that $(\gamma,R) \Rightarrow^*_G (\gamma',R')$.

Definition 4.12 (AMG derivation). Let $G(N,T,R_0,S,\Delta,k)$ be an AMG, and let $n \in \mathbb{N}$. A sequence of configurations

$$((\gamma_0,R_0), (\gamma_1,R_1), \ldots, (\gamma_n,R_n))$$

is an AMG derivation of length $n$ of $\gamma_n$ from $\gamma_0$ in $G$ if for all $i = 0, \ldots, n-1$

$$(\gamma_i,R_i) \Rightarrow_G (\gamma_{i+1},R_{i+1})$$

If $d = ((\gamma_0,R_0), (\gamma_1,R_1), \ldots, (\gamma_n,R_n))$ is an AMG derivation and for all $i = 0, \ldots, n-1$

$$(\gamma_i,R_i) \frac{r_i}{G} (\gamma_{i+1},R_{i+1})$$

we say that $\pi = r_1r_2\cdots r_n$ is the rule string underlying $d$. Furthermore, we denote the derivation $d$ by rule string $\pi$ as

$$(\gamma_0,R_0) \frac{r_1}{G} (\gamma_1,R_1) \frac{r_2}{G} (\gamma_2,R_2) \cdots \frac{r_n}{G} (\gamma_n,R_n)$$

A derivation by rule string $\pi \in R^*_{N,T}$ is called a single-pass derivation.

Definition 4.13 (Language of an AMG). Let $G(N,T,R_0,S,\Delta,k)$ be an AMG. The language of the AMG $G$, denoted by $L(G)$, consists of all annotated terminal strings $x \in \mathcal{A}_T^*$ for which there exist $R \subseteq R_{N,T}$ and $w \in T^*$ such that

$$(\langle S,w \rangle,R) \frac{w}{G} (x,R_0)$$

A rule string $\pi \in R^*_{N,T}$ is said to be an adaptive multi-pass parse of an annotated terminal string $x \in L(G)$ if there exists a rule set $R$ and annotation string $w$ such that

$$(\langle S,w \rangle,R) \frac{\pi}{G} (x,R_0)$$

A derivation of $x \in L(G)$ is said to be complete if the rule string underlying it is an adaptive multi-pass parse of $x$. A terminal string $t_1t_2\cdots t_n$ is said to be in the language of an AMG $G$ if

$$\langle t_1,t_1 \rangle \langle t_2,t_2 \rangle \cdots \langle t_n,t_n \rangle \in L(G)$$

Example 4.4. Figure 4.3 contains a complete adaptive multi-pass derivation of the annotated terminal string $abcd$ in $G_{UD}$.

Since there exists a rule set $R_3$ and a terminal string $ab$ such that

$$(\langle S,ab \rangle,R_3) \frac{ab}{G_{UD}} (abcd,R_0)$$

we conclude that $abcd \in L(G_{UD})$. 

Technion - Computer Science Department - M.Sc. Thesis ASC-2010-11 - 2010
4.3 Computational Power

In this section we prove that the AMG model defines the class of recursively enumerable sets. Given a recursively enumerable language $L$, there exists a Turing machine $\mathcal{M}_L$ with input alphabet $T = \{t_1, \ldots, t_n\}$ which recognizes $L$. That is, $\mathcal{M}_L$ halts if and only if its input is a member of $L$. We define the AMG $G_{L,k}$ in terms of $\mathcal{M}_L$ as presented in Figure 4.4.

**Lemma 4.1.** Let the following be a derivation in $G_{L,k}$ for some $n \geq 0$.

$$(\rho_n, R_n) \Rightarrow (\rho_{n-1}, R_{n-1}) \Rightarrow \cdots \Rightarrow (\rho_0, R_0)_{G_{L,k}}$$

Then $R_i = R_0$ for $i = 1, \ldots, n$.

*Proof.* Immediate from the definition of $\Delta$. \qed

**Lemma 4.2.** Let $w, x, y \in T^*$ be terminal strings. $(\langle I, w \rangle x, R_0) \Rightarrow (y, R_0)_{G_{L,k}}$ implies that $wx = y$.

*Proof.* The proof is by induction on $l = |w|$. According to Lemma 4.1 all the configurations of the derivation consist of the rule set $R_0$. If $l = 0$, according to the rules in $R_0$, the derivation must be of the form

$$(\langle I, \varepsilon \rangle x, R_0) \Rightarrow (x, R_0)_{G_{L,k}}$$

and the lemma holds since $\varepsilon x = x$. Assume then that $l > 0$ and, as the induction hypothesis, that the lemma holds for derivations where $|w| < l$. Let $w = zt_j$ where $z \in T^{l-1}$ and $t_j \in T$. According to the rules in $R_0$, the derivation must be of the form

$$(\langle I, zt_j \rangle x, R_0) \Rightarrow (\langle I, z \rangle t_j x, R_0) \Rightarrow (y, R_0)_{G_{L,k}}$$

Thus, according to the induction hypothesis $wx = zt_j x = y$. \qed

**Theorem 4.3.** Let $L$ be a recursively enumerable language. For all $k > 0$ there exists an AMG $G(N, T, R_0, S, \Delta, k)$ such that $L(G) = L$.

*Proof.* First, we prove that $L(G_{L,k}) \subseteq L$ for $k > 0$. Let $x$ be a terminal string in $L(G_{L,k})$. According to Definition 4.13, there exists a terminal string $w$ and a rule set $R$ such that

$$(\langle S, w \rangle, R) \Rightarrow (x, R_0)_{G_{L,k}}$$

Lemma 4.1 implies that all the configurations of this derivation consist of the rule set $R_0$. According to the rules in $R_0$, and $\Delta$, the derivation starts with

$$(\langle S, w \rangle, R_0) \Rightarrow (\langle M, w \rangle, R_0) \Rightarrow (\langle I, w \rangle, R_0) \cdots$$

and $\mathcal{M}_L$ accepts $w$. Lemma 4.2 implies that $w = x$ which leads us to conclude that $x \in L$.

Next we prove that $L \subseteq L(G_{L,k})$ for $k > 0$. Let $x = t_{a_1}t_{a_2} \cdots t_{a_m} \in L$ where $m \geq 0$, and $1 \leq a_i \leq n$ for $i = 1, \ldots, m$. The following derivation of $x$ is valid in $G_{L,k}$ for all $k > 0$.

$$(\langle S, t_{a_1}t_{a_2} \cdots t_{a_m} \rangle, R_0) \Rightarrow$$

$$(\langle M, t_{a_1}t_{a_2} \cdots t_{a_m} \rangle, R_0) \Rightarrow$$

$$(\langle I, t_{a_1}t_{a_2} \cdots t_{a_m} \rangle, R_0) \Rightarrow$$

$$(\langle I, t_{a_1}t_{a_2} \cdots t_{a_m-1} \rangle t_{a_m}, R_0) \Rightarrow$$

$$\vdots$$

$$(\langle I, t_{a_1} \rangle t_{a_2} \cdots t_{a_m}, R_0) \Rightarrow$$

$$(\langle I, \varepsilon \rangle t_{a_1}t_{a_2} \cdots t_{a_m}, R_0) \Rightarrow$$

$$(t_{a_1}t_{a_2} \cdots t_{a_m}, R_0)$$
Figure 4.3: AMG derivation of $abcd$ in $G_{UD}$

$$G_{L,k}(N,T,R_0,S,\Delta,k):$$

- $T = \{t_1, \ldots, t_n\}$
- $N = \{S,M,I\}$
- $R_0 = \{$
  - $r_1 = S \rightarrow M$,
  - $r_2 = M \rightarrow I$,
  - $r_3 = I \rightarrow \varepsilon$,
  - $r_4 = I \rightarrow I t_1$,
  - $r_5 = I \rightarrow I t_2$,
    
  $\vdots$
  - $r_{n+3} = I \rightarrow I t_n$\}$

- $\Delta(R,r,\alpha):$
  - if $r = r_2$ and $\alpha = \langle I, w \rangle$ then
    - Run $\mathcal{M}_L$ on $w$
  - return $(\emptyset, \emptyset)$

Figure 4.4: AMG $G_{L,k}$ defined for the Turing machine $\mathcal{M}_L$
Definition 4.13 implies that \( x \in L(G_{L,k}) \) for all \( k > 0 \) and therefore, for all \( k > 0 \), \( L \subseteq L(G_{L,k}) \). Thus, by mutual inclusion we conclude that for every recursively enumerable language \( L \) and \( k > 0 \) there exists an AMG \( G(N, T, R_0, S, \Delta, k) \) (\( = G_{L,k} \)) such that \( L = L(G) \).

4.4 AMGs vs. CFGs

The AMG model can be viewed as an extension of CFGs. From a structural standpoint, AMGs differ from CFGs in the following ways:

- The nonterminal alphabet \( N \) can be infinite.
- Production rules are associated with annotation functions.
- A distinction exists between single-pass and multi-pass production rules.
- AMGs consist of an adaptation function \( \Delta \) and a lookahead specifier \( k > 0 \).

The AMG derivation relation and language definition further differ from their CFG counterparts as follows:

- Sentential forms consist of annotated symbols. Nonterminal annotations encode the terminal strings rewritten by multi-pass production rules. From an LR-parser perspective, they encode the terminal strings to be pushed back to the input stream following reduction by such rules.
- Annotation functions compute the annotations of rewritten annotated nonterminals from the annotations of written annotated strings.
- The set of production rules applicable for rewriting sentential forms may vary between derivation steps. The model’s adaptation function dictates these changes.

Given these differences, it is easy to see that an AMG \( G(N, T, S, R_0, \Delta, k) \) whose adaptation function never varies the set the production rules, and whose base rule set does not consist of multi-pass rules, will operate exactly like a CFG. All complete derivations in \( G \) are of the form

\[
(S, w), R_0) = (\gamma_0, R_0) \xrightarrow{\xi_1} (\gamma_1, R_0) \xrightarrow{\xi_2} (\gamma_2, R_0) \cdots \xrightarrow{\xi_n} (\gamma_n, R_0) = (x, R_0)
\]

and exist if and only if

\[
S = \bar{\gamma}_0 \Rightarrow \bar{\gamma}_1 \Rightarrow \bar{\gamma}_2 \cdots \Rightarrow \bar{\gamma}_n = \bar{x}
\]

Thus, we conclude that CFGs are a special case of AMGs, as formally stated in the following theorem.

**Theorem 4.4.** Let \( G_1(N, T, R_0, S, \Delta, k) \) be an AMG where \( N \) is finite, \( R_0 \subseteq R_{N,T}^- \), and \( \Delta(R, r, \alpha) = (\emptyset, \emptyset) \) for all \( R, r \) and \( \alpha \). Further, let \( G_2(N, T, \xi(R_0), S) \) be a CFG, and let \( x \in A_T^* \). Then

\[
x \in L(G_1) \iff \bar{x} \in L(G_2)
\]

**Proof.** Immediate from Definitions 4.9 and 4.13. \( \square \)
Chapter 5

Adaptive Multi-pass Parsers

In this chapter we present an LR($k$)-based Adaptive Multi-pass Parser (AMP) for AMG languages. We begin with an overview of the LR($k$) parser extensions employed by our AMG parser to support parse-time grammar modifications and reduction by multi-pass rules. Next, we present a complete, implementation friendly, nondeterministic parsing algorithm for AMG languages. We further define a construct called an *AMP configuration* that allows us to succinctly describe the state of an AMP and track the progress of its computations while parsing an input sentence. We conclude this chapter by formally proving the correctness of our parser with respect to the AMG formalism.

5.1 Overview

As described in Section 2.5, an LR($k$) parser is driven by a parse table that encodes all possible parse actions and state progressions needed for recognizing a sentence in the language of a given CFG. The context of the parser within the parsing process is dynamically maintained using a stack of states. As implied by the LRPARSE algorithm presented in Figure 2.1, all other aspects of the LR($k$) parsing algorithm are stateless and grammar independent. Therefore, in the event of grammar modification, an adaptive LR($k$) parser must be able to reconstruct its parse table to reflect the state progressions and parse actions of the new grammar, and restore its parsing context by repopulating its state stack with states of the new parse table.

An important property of deterministic LR($k$) parsers is that their parse time is linear in the length of the parsed input sentence. However, the problem of generating a complete LR($k$) prefix-automaton (and thus, a complete parse table) for a given CFG is exponential in the number of production rules of the grammar [43]. Rebuilding the entire parse table per grammatical change is clearly inefficient; especially in face of frequent modifications, where very few states of the generated table are actually traversed by the parser before it is discarded. To avoid this deficiency, our parser employs a lazy approach to parse-table generation. As parsing progresses, only those states that are pushed to the state stack are generated. In fact, our parser avoids the construction of the parse table representation entirely, by maintaining the raw prefix-automaton states it generates and inferring parse actions and state transitions directly from these states, as described in Section 2.5. This technique is feasible due to the locality property of the LR($k$) prefix-automaton. It follows from the KERNEL method of Figure 2.2, that the kernel of a state can be fully generated given a preceding state and a symbol. The resulting set of kernel items can then be augmented to include all LR($k$)-valid items simply by considering the properties of the underlying CFG (see CREATESTATE$_k$). After initializing the state stack with the start state, every other state required by the parser can be generated directly from the state
at the top of the parse stack.

The contents of an LR parser’s state stack, define a path in the grammar’s prefix-automaton leading from the start state to the parser’s current state. The symbols annotating the edges of this path, constitute a sentential form that derives the terminal string that was read by the parser up to that point. Following a grammar modification, the state stack must be rebuilt to represent this sentential form using states of the new prefix-automaton. Our parser accomplishes this task by maintaining a stack of symbols alongside the state stack that consists of the symbols annotating the prefix-automaton edges along the path defined by the state stack. Whenever a state is (lazily) created using a symbol X and pushed to the state stack, the symbol X is pushed onto the symbol stack. Whenever a state is removed from the top of the state stack so does the edge symbol that lead to it. Following a grammatical change, our parser discards the existing state stack, and attempts to reconstruct it by creating and pushing all the states of the new prefix-automaton that appear along the path specified by the contents of the symbol stack. If the path does not fully exist in the new prefix-automaton, the parser terminates with an adaptation error to indicate that the input sentence is not in the language recognized by the parser.

Another aspect of the AMG model that dictates extensions to the traditional LR\((k)\) parsing algorithm is multi-pass derivations. As implied by Definition 4.9, these involve computation of symbol annotations and rewriting of annotated terminal strings with the RHS of AMG rules. Computation of symbol annotations is accomplished by maintaining yet another stack of symbol annotations. Whenever a terminal \(a\) is shifted from the input sentence, its annotation \((a)\) is pushed onto the annotation stack. When performing a Reduce action, the annotation associated with the LHS nonterminal symbol of the reduced rule is computed using its annotation function directly from the annotations stored in the annotations stack which correspond to the RHS symbols of the reduced rule. Following this computation, all RHS annotations are discarded and replaced by the new LHS annotation. When reducing a multi-pass rule, the traditional Reduce action is further extended by discarding the representations (i.e., state, symbol, and annotation) of the rule’s LHS nonterminal from all parser stacks and pushing its annotation back to the beginning of the input sentence. Furthermore, the representations of all terminal symbols uncovered at the top of the parser’s symbol stack are also discarded and their annotations are pushed back to the input sentence. Though not as intuitive, the latter step is required to properly simulate derivations by multi-pass rules. The discarded terminals represent a series of Shift actions that immediately preceded the current Reduce action. They are not necessarily valid following the reduce action because (1) the \(k\)-length prefix of the restored input sentence may differ from the one available when the actions were inferred, and (2) the underlying set of rules following the reduce action may have changed. For example, consider the AMG

\[
G(N, T, R_0, S, \Delta, 1) : \\
N = \{S, A, C\} \\
T = \{a, b, c, d\} \\
R_0 = \{ \\
    r_1 = S \to abC \\
    r_2 = S \to Abd \\
    r_3 = A \to a \\
    r_4 = (C, d) \leftrightarrow c \} \\
\Delta(R, r, \rho) = (\emptyset, \emptyset)
\]
and the following derivation of \( abc \) from \( \langle S, abd \rangle \) in \( G \):

\[
\begin{align*}
(\langle S, abd \rangle, R_0) & \xrightarrow{r_2} (\langle A, a \rangle bd, R_0) \xrightarrow{r_3} (abd, R_0) \xrightarrow{r_4} (abc, R_0)
\end{align*}
\]

In order to produce this derivation (in reverse order), an AMG parser would first shift the terminals \( a, b, \) and \( c \), then reduce the terminal \( c \) using the multi-pass rule \( r_4 \), resulting with an empty symbol stack and the string \( abd \) as its input sentence. Next, it would shift \( a \), reduce it to the nonterminal \( \langle A, a \rangle \) using rule \( r_3 \), shift the remaining terminals \( b, d \) and conclude by reducing its entire stack \( \langle A, a \rangle bd \) to \( \langle S, abd \rangle \) using rule \( r_2 \). Clearly, if the terminals \( a \) and \( b \) would not have been discarded from the parser’s stack following the reduction of rule \( r_4 \), the parser would not have been able to produce the above derivation.

## 5.2 AMG Parsers (AMPs)

We now present a nondeterministic LR(\( k \))-based Adaptive Multi-pass Parser (AMP) for AMG languages. The parser is a variation of the LR(\( k \)) parser for CFGs described in Section 2.5. It begins parsing with an initial set of AMG rules which corresponds to the base rule set of the grammar \( (R_0) \). Upon each rule reduction, the set of rules applicable for subsequent parsing steps is determined by consulting the grammar’s adaptation function \( (\Delta) \). The set of applicable rules is represented by the set \( Rules \). For simplicity, we assume that no two distinct rules in \( Rules \) share the same skeleton. By doing so, we can avoid the distinction between CFG rules and AMG rules altogether; that is, we may apply an AMG rule where its respective skeleton is expected and vice versa. Furthermore, the \( Rules \) set implicitly defines the CFG

\[
G_{\xi}(\{A | A \rightarrow \omega \in \xi(Rules)\}, T, \xi(Rules), S')
\]

whose prefix-automaton is used to drive the parser’s state transitions and applicable parse actions, as reflected by the rightmost derivation at the base of the model’s derivation relation (see clause 4.3 of Definition 4.9).

In addition to the traditional state stack, \( StateStack \), our parser maintains a stack of symbols, \( SymStack \), that is used to reconstruct the state stack following a change to the grammar’s underlying set of CFG rule skeletons. For all \( i = 1, \ldots, |StateStack| \) the state \( StateStack[i] \) contains all items that are LR(\( k \))-valid for the string

\[
SymStack[1] \cdot SymStack[2] \cdots SymStack[i - 1]
\]

with respect to \( G_{\xi} \). A third stack, \( AnnStack \), is used for storing symbol annotations, such that for all \( i = 1, \ldots, |AnnStack| \)

\[
\langle SymStack[i], AnnStack[i] \rangle \in A_{N,T}
\]

represents an annotated symbol. Together with the parser’s input string \( x \), the symbol and annotation stacks, and the set of available AMG rules, simulate an AMG configuration

\[
((SymStack \times AnnStack) \cdot x, Rules) \in C_{\xi}
\]

Figure 5.1 presents the two routines that facilitate parse-time adaptation to grammatical modifications. The UpdateRules method updates the \( Rules \) set with rule modifications indicated by the grammar’s adaptation function. It takes two sets of rules as input: one indicating rules to be added \( (R_a) \), and another indicating rules to remove \( (R_r) \). The routine first adds the AMG rules from the input set \( R_a \) and then removes
\begin{verbatim}
UpdateRules(R_a, R_r)
1   Rules ← (Rules ∪ R_a) \ R_r
2   return True if and only if Rules was modified

Adapt()
1   q ← CreateState_{k}(\{[S' → \bullet S$, $]},
2   StateStack ← \langle q \rangle
3   for each X in SymStack do
4       q ← Goto(q, X)
5       if q = \emptyset then
6           exit “Adaptation error”
7       Push(StateStack, \langle q \rangle)

Goto(q, X)
1   return CreateState_{k}(Kernel(q, X))
\end{verbatim}

Figure 5.1: AMP grammar adaptation routines

those from the $R_r$ set (line 1). It returns True if and only if the Rules set has changed (line 2). The Adapt routine updates the parser’s state stack with respect to the (modified) CFG $G$. It reconstructs the state stack by first discarding the current stack, and recreating it along with a fresh start state (lines 1-2). It then continues to create and push all subsequent states that appear in the new grammar’s prefix-automaton along the path indicated by the parser’s symbol stack (lines 3-7). If during this process, a state with no items is encountered, the routine exits with an adaptation error, thereby failing the parse process. Replacing the traditional GOTO table, the Goto routine is invoked (with the same arguments) to compute state transitions. Instead of consulting a parse table, a new LR($k$) state is created directly from the current state, resulting with the lazy prefix-automaton generation described in Section 5.1.

The main routine of the parser – AMGParse – is presented in Figure 5.2. The routine consist of two main parts: parser initialization (lines 1-4) and the main parsing loop (lines 5-33). During parser initialization, the Rules set and parser stacks are created (lines 1-2), and the endmarker $\$ is added to the end of the input sentence $x$ (line 3). Next, the Rules set is updated with the initial set of AMG rules applicable for reduction, as indicated by the grammar $G$. This final initialization step is accomplished by invoking the UpdateRules routine with the set of rules to be added to the parser’s (initially empty) rule set (line 4). As commonly practiced with traditional LR parsers, the grammar’s initial set of AMG rules is augmented with the rule $S' \rightarrow S\$. The adapt variable is used to store the routine’s return value, which is True if and only if the underlying set of CFG rules has changed. Since the set of added rules is not empty, the routine is bound to return True.

The main parsing loop also consists of two parts: parser adaptation (lines 6-8) and parse action execution (lines 9-35). In the first part, the parser adapts to grammar modifications that may have occurred in the previous iteration of the parsing loop, by invoking the Adapt routine. The routine is invoked only if the adapt variable is set to True. In the first iteration of the parsing loop, immediately after the parser is initialized, the Adapt routine initializes the state stack with the parser’s start state. The parse action execution part of the algorithm is very similar to that of a traditional LR parser. It starts by obtaining the current state from the top of the state stack and uses it to infer one of the three LR parse actions: Shift, Reduce or Accept (lines 9-10).

Shift actions are handled in lines 11-15 of the parsing loop. In addition to pushing a new state onto the state stack and removing the shifted terminal $1 : x$ from the input
AMGParse(x)
1 Rules ← {};
2 StateStack, SymStack, AnnStack ← ⟨⟩, ⟨⟩, ⟨⟩;
3 x ← x · $;
4 adapt ← UpdateRules(R_0 ∪ \{ S' \to S$\}, {});
5 while True do
6 if adapt then
7 Adapt();
8 adapt ← False;
9 ⟨q⟩ ← Top(StateStack, 1);
10 if exists action in Action(q, k : x) then
11 if action = ⟨Shift⟩ then
12 Push(StateStack, ⟨Goto(q, 1 : x)⟩);
13 Push(SymStack, ⟨1 : x⟩);
14 Push(AnnStack, ⟨1 : x⟩);
15 x ← x : (|x| − 1);
16 else if action = ⟨Reduce, r = ⟨A, f_r⟩ ⊕ X_1X_2 \cdots X_n⟩ then
17 ρ ← Top(SymStack × AnnStack, n);
18 R_a, R_r ← Δ(Rules \{ S' \to S$\}, r, ρ);
19 rhs ← Top(AnnStack, n);
20 w ← f_r(rhs[1], ..., rhs[n]);
21 Pop(StateStack, SymStack, AnnStack, n);
22 ⟨q⟩ ← Top(StateStack, 1);
23 Push(StateStack, ⟨Goto(q, A)⟩);
24 Push(SymStack, ⟨A⟩);
25 Push(AnnStack, ⟨w⟩);
26 if r in R_{\mathcal{N}} then
27 do
28 x ← AnnStack[AnnStack] · x;
29 Pop(StateStack, SymStack, AnnStack, 1);
30 while |SymStack| > 0 and SymStack[SymStack]| in T
31 adapt ← UpdateRules(R_a, R_r);
32 else if action = ⟨Accept⟩ then
33 exit “Success”;
34 else
35 exit “Syntax error”;

Figure 5.2: AMP parsing routines
sentence, the shifted terminal is also pushed onto the symbol and annotation stacks, effectively simulating the shift of the annotated terminal $\langle 1 : x, 1 : x \rangle$.

Upon rule reduction (lines 16-30), the grammar's adaptation function $\Delta$ is invoked, resulting with the sets of AMG rules to be added ($R_a$) and removed ($R_r$) from the parser's set of available rules (lines 17-18). The rule $S' \rightarrow SS$ is not included in the input to $\Delta$ since it is artificially added by the parser and is not part of grammar $G$. Next, the rule's LHS nonterminal annotation is computed from the annotation stack by applying the rules' annotation function, denoted by $f_r$ (lines 19-20). Then, the representations of the reduced rule's RHS are removed from all stacks, and the corresponding representations of its LHS nonterminal are pushed (lines 21-25). If the reduced rule is a multi-pass rule, the representations of its LHS nonterminal are removed from all parse stacks, and its annotation is shifted back to the input stream to be subsequently re-parsed. Furthermore, all terminal symbols uncovered at the top of the parser's symbol stack (indicating previous Shift actions), are also removed from the parser stacks and pushed back to the input sentence, to ensure that all Reduce actions that are now applicable with respect to the possibly modified input sentence or rule set can be executed (lines 26-30).

The iteration concludes with a call to $\text{UPDATERULES}$ which updates the set of rules available for future reduction according to the output of the grammar's adaptation function (line 31). If the $\text{Rules}$ set was indeed modified, the adapt variable is set to True, ensuring that the parser's state stack will be updated by the $\text{ADAPT}$ routine at the beginning of the next iteration.

The remaining parts of the algorithm are identical to the operation of a traditional LR parser. An Accept action triggers the successful termination of the parse process, indicating that the input sentence is in the language of the underlying AMG (lines 32-33). On the other hand, if at some stage of the parsing process no parse action is applicable, the parser terminates with a syntax error to indicate the input sentence is not of the grammar's language (lines 34-35).

### 5.3 AMP Computations

We proceed by defining a construct called an $\text{AMP configuration}$ that allows us to succinctly describe the state of an AMP and track the progression of its computations while parsing an input sentence.

**Definition 5.1 (AMP configuration).**

Let $G(N,T,R_0,S,\Delta,k)$ be an AMG and $M$ its AMP. We denote the state of $M$ during execution of the main parsing loop by means of an AMP configuration of the form

$$(q_1,q_2,\ldots,q_m,\gamma|x,R)$$

where $m \geq 0$, $R \subseteq R_{N,T} \cup \{S' \rightarrow SS\}$ represents $M$'s (finite) set of applicable AMG rules (i.e., its Rules set), $x \in T^*\{$$\}$ denotes $M$'s input stream, $q_1,\ldots,q_m \in [\xi(R')]^k*$, where $R' \subseteq R_{N,T} \cup \{S' \rightarrow SS\}$, is a state string denoting the contents of $M$'s state stack

$$\text{StateStack} = \langle q_1,q_2,\ldots,q_m \rangle$$

and $\gamma = \langle X_1,w_1 \rangle \langle X_2,w_2 \rangle \cdots \langle X_{m-1},w_{m-1} \rangle \in \mathcal{A}_{N,T}^\ast\{$$\}$\} is the annotated string represented by $M$'s symbol and annotation stacks

$$\text{SymStack} = \langle X_1,X_2,\ldots,X_{m-1} \rangle, \text{AnnStack} = \langle w_1,w_2,\ldots,w_{m-1} \rangle$$
Unless explicitly specified otherwise, an AMP configuration represents the state of $M$ at the beginning of a parsing loop (line 6). An AMP configuration $(q_1 \cdots q_m, \gamma|x, R)$ is said to be \textit{valid} if

$$S_{\xi(R)}^q = q_1 \cdots q_m,$$

and $q_i \neq \emptyset$ for all $i = 1, \ldots, m$

A valid AMP configuration can be denoted by

$$(S, \gamma|x, R) = (S_{\xi(R)}^q, \gamma|x, R)$$

On each non-aborting iteration of the main parsing loop, an AMP executes exactly one AMG parse action. This computation progression from (the beginning of) one parsing loop iteration to the next is denoted by

$$(\phi_1, \gamma_1|x_1, R_1) \xrightarrow{r'} (\phi_2, \gamma_2|x_2, R_2)$$

where $r'$ is either a Shift action, or a Reduce action ($\text{REDUCE}, r$) where $r$ is an AMG rule (based on the assumption that no two AMG rules in $R$ share the same rule skeleton). The computation progression by action string $\pi$ is recursively defined as

$$\pi = \varepsilon : (\phi_1, \gamma_1|x_1, R_1) \xrightarrow{\varepsilon} (\phi_1, \gamma_1|x_1, R_1)$$
$$\pi = r'\pi' : (\phi_1, \gamma_1|x_1, R_1) \xrightarrow{r'} (\phi_3, \gamma_3|x_3, R_3) \xrightarrow{\pi'} (\phi_2, \gamma_2|x_2, R_2)$$

We can now formally define the language of an AMP as a computation progression from its initial configuration to its accepting configuration.

\textbf{Definition 5.2 (Language of an AMP).}

\textit{Let $G(N, T, R_0, S, \Delta, k)$ be an AMG and $M$ its AMP. The language of the AMP $M$, denoted by $L(M)$, consists of all annotated nonterminals $x \in T^*$ for which there exists a rule set $R \subseteq R_{N,T}$, an annotation string $w \in T^*$ and an action string $\pi'$ such that}

$$(\varepsilon, \varepsilon|x$$,$$R_0 \cup \{S' \rightarrow S\}) \xrightarrow{\pi'} (S, (S, w)(\$$, $$\varepsilon, R \cup \{S' \rightarrow S\})$$

\textit{It is easy to see that the (invalid) AMP configuration on the left corresponds to the state of the parser at the beginning of the first iteration, immediately following its initialization. The AMP configuration on the right represents the beginning of an accepting iteration. Having no terminals left in the input stream, and}

$$[S' \rightarrow S\$$,$$\varepsilon] \in [S\$$,$$k]_{\xi(R_0 \cup \{S' \rightarrow S\})}$$

\textit{imply that the parser is about to obtain an Accept parse action and terminate successfully. The action string $\pi'$ which leads to the acceptance of the string $x$ by the AMP $M$ is called an adaptive multi-pass parse of $x$ in $M$. An adaptive multi-pass parse consisting of $m > 0$ multi-pass Reduce actions is called an adaptive multi-pass parse with $m + 1$ passes.}

\section{5.4 Correctness}

In this section we formally prove the correctness of our AMG parsing algorithm. The correctness proof is organized as follows. We give two lemmas (5.5, 5.8) that relate adaptive multi-pass derivations of an AMG with iterations of its parser’s main parsing loop. Using these lemmas we prove that not only the language accepted by an AMP is identical to that of its AMG, but that for each sentence $x$ in the language of the AMG, the parser produces exactly all adaptive multi-pass parses of $x$ in $G$ (Theorem 5.11).

The following two lemmas state some basic properties of CFG rightmost derivations and LR($k$)-valid item sets that will be of use to us in subsequent proofs.
Lemma 5.1. Let \((N, T, P, S')\) be a CFG, \(\pi \in P^*, \gamma, \eta, \delta \in (N \cup T)^*, A \in N\) and \(y \in T^*\) a terminal string such that

(a) \[ S \xrightarrow{\delta} \gamma \eta y = \delta Ay, \text{ and } \pi \neq \varepsilon \]

Then there are strings \(\delta' \in (N \cup T)^*\) and \(y' \in T^*\), rule strings \(\pi', \pi'' \in P^*\) and a rule \(r = A' \rightarrow \alpha' \beta' \in P\) such that \(\pi = \pi' r \pi''\), \(\alpha' : 1 = \gamma : 1\),

(b) \[ S \xrightarrow{\delta'} \delta' A' y' \xrightarrow{r} \delta' \alpha' \beta' y' = \gamma \beta' y', \text{ and } \beta' y' \xrightarrow{\pi''} \eta y \]

Proof. The proof is by induction on the length of rule string \(\pi\). If \(|\pi| = 1\), (a) implies that \(\pi = S \rightarrow \gamma \eta y\) is a rule in \(P\). Statement (b) then holds if we choose \(\delta' = y' = \varepsilon\), \(\pi' = \pi'' = \varepsilon\), \(r = \pi\), \(\alpha' = \gamma\), and \(\beta' = \eta y\). Assume then that \(|\pi| > 1\) and, as the induction hypothesis, that the lemma holds for rule strings shorter than \(\pi\). Statement (a) then implies the existence of strings \(\delta_1 \in (N \cup T)^*\) and \(y_1 \in T^*\), a rule string \(\pi_1\), and a rule \(r_1 = A_1 \rightarrow \omega_1\) such that \(\pi = \pi_1 r_1\) and

\[
S \xrightarrow{\delta_1} \delta_1 A_1 y_1 \xrightarrow{r_1} \delta_1 \omega_1 y_1 = \gamma \eta y = \delta Ay
\]

Here \(y_1\) must be a suffix of \(y\), that is, \(y = xy_1\) for some \(x\). Moreover, either \(\gamma = \delta_1 \alpha'\) for some \(\alpha' \neq \varepsilon\) or \(\delta_1 = \gamma \alpha\) for some \(\alpha\). In the former case, \(\omega_1 = \alpha' \eta x\) and statement (b) holds if we choose \(\delta' = \delta_1\), \(y' = y_1\), \(\pi' = \pi_1\), \(\pi'' = \varepsilon\), \(r = r_1\), and \(\beta' = \eta x\). In the latter case, we may write the first derivation segment in (a) as

\[
S \xrightarrow{\delta_1} \gamma \eta_1 y_1 = \delta_1 A_1 y_1
\]

where \(\eta_1 = \alpha A_1\). Since \(|\pi| > 0\) implies that \(\pi_1 \neq \varepsilon\), we can apply the induction hypothesis to (5.2) and conclude that there are strings \(\delta' \in (N \cup T)^*\) and \(y' \in T^*\), rule strings \(\pi', \pi_2\), and a rule \(r = A' \rightarrow \alpha' \beta'\), such that \(\pi_1 = \pi' r \pi_2\), \(\alpha' : 1 = \gamma : 1\),

\[
S \xrightarrow{\delta'} \delta' A' y' \xrightarrow{r} \delta' \alpha' \beta' y' = \gamma \beta' y', \text{ and } \beta' y' \xrightarrow{\pi''} \eta_1 y_1
\]

Since \(\delta_1 \omega_1 y_1 = \gamma \alpha \omega_1 y_1, y = xy_1,\) and \(\alpha \omega_1 y_1 = \eta y\), we have \(\alpha \omega_1 = \eta x\), and

\[
\eta_1 y_1 = \alpha A_1 y_1 \xrightarrow{r_1} \alpha \omega_1 y_1 = \eta xy_1 = \eta y
\]

By combining (5.3) and (5.4) and choosing \(\pi'' = \pi_2 r_1\) we conclude that statement (b) holds.

Lemma 5.2. Let \(G'(N', T', P', S')\) be a $ augmented CFG, \(r = A \rightarrow \alpha \beta \in P'\) and \(\delta \in (N' \cup T')^*\). Then

(a) \[ [A \rightarrow \alpha \cdot \beta, z] \in [\delta \alpha]_{G'}^k \]

implies that

(b) \[ \alpha' \beta' = \alpha \beta \implies [A \rightarrow \alpha' \cdot \beta', z] \in [\delta \alpha']_{G'}^k \]

(c) \[ A \neq S' \implies [\delta A]_{G'}^k \neq \emptyset, \text{ and} \]

(d) \[ [i : \delta \alpha \beta]_{G'}^k \neq \emptyset \text{ for all } i = 0, \ldots, |\delta \alpha \beta| \]
We conclude that statement (d) holds.

By choosing $A = A'$, we can write

$$S' \xrightarrow{\delta} \delta A y \xrightarrow{\delta} \delta A' y , \text{ and } z = k : y$$

which implies that

$$[A \rightarrow \alpha' \bullet \beta', z] \in [\delta \alpha' \beta'_G]$$

Thus, we conclude that statement (b) holds, and

$$[i : \delta \alpha \beta]_{c_G} \neq \emptyset \text{ for all } i = |\delta|, \ldots, |\delta \alpha \beta|$$

If $\pi = \varepsilon$, we have in (5.5) $\delta = x = \varepsilon$, $A = S$ and $r = S' \rightarrow S\$$. $A = S$ implies that (c) trivially holds, and by choosing $\alpha \beta = S\$ we conclude from (5.6) that statement (d) also holds. If $\pi \neq \varepsilon$ we can apply Lemma 5.1 to the first derivation segment of (5.5), and conclude that for all $i = 0, \ldots, |\delta A|$, given the string $\gamma_i = i : \delta A$, there are strings $\delta_i \in (N' \cup T')^*$ and $y_i \in T'^*$, a rule string $\pi_i$ and a rule $r_i = A_i \rightarrow \alpha_i \beta_i \in P'$ such that $\alpha_i : 1 = \gamma_i : 1$ and

$$S \xrightarrow{\delta} \delta A_i y_i \xrightarrow{\delta \alpha_i \beta_i} \gamma_i \beta_i y_i$$

Thus, we have for all $i = 0, \ldots, |\delta A|$ (5.7)

$$[A_i \rightarrow \alpha_i \bullet \beta_i, k : y_i] \in [\gamma_i^{k_G}, [i : \delta A]^{k_G}]$$

By choosing $i = |\delta A|$ we conclude that statement (c) holds, and by combining (5.6) and (5.7) we conclude that statement (d) holds.  

Next, we formally define the skeletal CFG underlying an AMP while parsing an input sentence with respect to its current set of applicable AMG rules.

**Definition 5.3 (Skeletal CFG).**

Let $G(N, T, R_0, S, \Delta, k)$ be an AMG, $M$ its AMP, $R \subseteq R_{N,T}$ a finite set of AMG rules, and $R' = R \cup \{S' \rightarrow S\}$. The skeletal CFG underlying $M$ with respect to $R'$ is the CFG

$$M^{R'}_{\xi} = \{\xi' | A \rightarrow \omega \in \xi(R)\}, T \cup \{\$\}, \xi(R), S\}$$

The skeletal CFG underlying $G$ with respect to $R$ is the CFG

$$G^{R}_{\xi} = \{\xi' | A \rightarrow \omega \in \xi(R)\}, T, \xi(R), S\}$$

Wherever a CFG is expected, $\xi(R)$ can be used to denote the skeletal CFG $M^{R'}_{\xi}$.

The following fact states the correctness of the traditional prefix-automaton construction algorithms (listed in Figure 2.2) that are put to use by our parser.

**Fact 5.3.** Let $G'(N', T', P', S')$ be a $\$ augmented context-free grammar. Further, let $k > 0$, $\gamma \in (N' \cup T')^*$, $X \in N' \cup T'$, and $S' \rightarrow S\$ $\in P'$. Then

$$[\varepsilon]^{k_G} = \text{createState}^k(\{[S' \rightarrow \bullet S\$, $\varepsilon]\})$$

$$[\gamma X]^{k_G} = \text{createState}^k(\text{Kernel}([\gamma]^{k_G}, X))$$

The following lemma states the conditions for the parser’s successful adaptation to parse-time grammar modifications, taking into account the use of the adapt variable which prevents the unnecessary invocation of the ADAPT routine on every iteration.
Lemma 5.4. Let $G(N,T,R_0,S,\Delta,k)$ be an AMG and $M$ its AMP. Furthermore, let $(\phi,\gamma|y\$, $R)$ be an AMP configuration of $M$. Then $M$ results with the valid configuration $(S,\gamma|y\$, $R)$ upon execution of lines 6-7 of the main parsing loop if and only if $[\gamma]_{\xi(R)}^k \neq \emptyset$.

Proof. The proof is by induction on the iteration count $i \geq 1$ of the main parsing loop. If $i = 1$, the AMP configuration

$$(\phi,\gamma|y\$, $R) = (\epsilon,\epsilon|x\$, $R_0')$$

where $R_0' = R_0 \cup \{S' \rightarrow S\}$, represents the state of $M$ at the beginning of the first iteration, immediately following its initialization (lines 1-4). On line 1 the set of AMG rules Rules is initialized to an empty set, and the invocation of UPDATERULES on line 4 adds at least one rule $S' \rightarrow S\$. Thus, accept is initialized to TRUE on line 4 and the ADAPT routine is called on line 7. Since the symbol stack is empty, only lines 1-2 of the ADAPT routine are executed, resulting (by Fact 5.3) with the AMP configuration $([\epsilon]_{\xi(R_0')}^k, \epsilon|x\$, $R_0)$.

Thus we conclude that the lemma holds. On the other hand, if $r \neq \emptyset$ is not empty, then at line 10 of the previous iteration $M$ has a parse action $r'$, the item set $[i : \gamma]_{\xi(R)}^k \neq \emptyset$. As we have shown in the base case, in lines 1-2 of the ADAPT method, the state stack is initialized with the item set $[\epsilon]_{\xi(R_0')}^k \neq \emptyset$. Further, by Fact 5.3, upon iteration $1 \leq j \leq |\gamma|$ of the loop of lines 3-7, the item set $[j : \gamma]_{\xi(R)}^k$ is created, and (since it is not empty) pushed onto the state stack. Thus we conclude that the ADAPT routine does not exit with an error and results with the valid AMP configuration $(S,\gamma|y\$, $R)$ as stated by the lemma. On the other hand, lines 5-6 of ADAPT, ensure that no empty item sets are created, including the item set $[\gamma]_{\xi(R)}^k$ of the final iteration. Thus we conclude that the lemma holds in this case as well.

Next, consider the case where $adapt = False$. If $(S,\gamma|y\$, $R)$ is valid, $[\gamma]_{\xi(R)}^k \neq \emptyset$ by definition. Assume then, that $[\gamma]_{\xi(R)}^k \neq \emptyset$. Since at line 10 of the previous iteration $M$ has a parse action $r'$, the item set $q$ at the top of the state stack is not empty. By applying the induction hypothesis, we conclude that the AMP configuration $(S,\gamma|y\$, $R)$ resulting from the execution of lines 6-7 of that iteration is valid. If $r'$ is a Shift action, the iteration completes after executing lines 12-15 of the parsing loop, which results with the AMP configuration

$$(S'y\$, $\gamma'(t,t)|y\$, $R) = (\phi,\gamma|y\$, $R)$$

where $y' = ty$. Since $[\gamma]_{\xi(R)}^k \neq \emptyset$ we conclude that $(\phi,\gamma|y\$, $R)$ is valid as stated by the lemma. On the other hand, if $r'$ is a Reduce action, popping of all stacks in line 22 results with a valid configuration, and so does (by Fact 5.3 and Lemma 5.2) the pushing of the reduced rule’s LHS representation in lines 23-25. Further, if the reduced rule is a multi-pass rule, a valid AMP configuration also results following the popping of the representations of its LHS nonterminal and subsequent terminals from all stacks in lines 26-30. Finally, since $adapt = False$, the call to UPDATERULES in line 30, does not modify the skeletal CFG underlying $M$, which implies that the AMP configuration remains valid. As the iteration completes at this point, we conclude that the AMP configuration $(\phi,\gamma|y\$, $R)$ at the beginning of iteration $i$ is valid as stated by the lemma. Since we have covered all possible cases, we conclude that the lemma holds. \QED
In the next lemma we show that every computation progression of a parser yields a corresponding derivation in its AMG. For that aim, we define the homomorphism $\tau$ from the set of AMP action strings to the set of AMG rule strings, which maps every (REDUCE, $r$) action to the AMG rule $r$, and every (SHIFT) action to the empty string $\epsilon$.

**Lemma 5.5.** Let $G(N,T,R_0,S,\Delta,k)$ be an AMG and $M$ its AMP. Further, let $\phi_1, \phi_2$ be state strings, $\gamma_1, \gamma_2 \in A_{N,T}^*$, $y_1, y_2 \in T^*$, $R_1', R_2' \subseteq R_{N,T} \cup \{S' \rightarrow S\}$, and let $\pi'$ be an action string such that

(a) $$(\phi_1, \gamma_1|y_1S, R_1') \xrightarrow{\pi'}(\phi_2, \gamma_2|y_2S, R_2')$$

Then we have in $G$

(b) $$(\gamma_2y_2, R_2) \xrightarrow{\tau(\pi')R}(\gamma_1y_1, R_1)$$

where $R_i = R'_i \setminus \{S' \rightarrow S\}$ for $i \in \{1, 2\}$.

**Proof.** The proof is by induction on the length of $\pi'$. If $|\pi'| = 0$, statement (b) trivially holds if we choose $\gamma_2 = \gamma_1, y_2 = y_1, \phi_2 = \phi_1$, and $R_2' = R_1'$. Assume then that $|\pi'| > 0$ and, as the induction hypothesis, that the lemma holds for action strings shorter than $\pi'$. Let $\pi' = r'\pi'_1$ where $r'$ is a single action. Thus, we have for some state string $\phi_3$ and symbol strings $\gamma_3 \in A_{N,T}^*$, $y_3 \in A_T^*$

$$\begin{align*}
(\phi_1, \gamma_1|y_1S, R_1') & \xrightarrow{r'}(\phi_3, \gamma_3|y_3S, R_3') \xrightarrow{\pi'_1}(\phi_2, \gamma_2|y_2S, R_2') \\
(\gamma_2y_2, R_2) & \xrightarrow{\tau(\pi'_1)R}(\gamma_3y_3, R_3)
\end{align*}$$

By applying the induction hypothesis on the second derivation segment, we conclude that

$$\begin{align*}
\phi_1, \gamma_1|y_1S, R_1' & \xrightarrow{r'}(\phi_3, \gamma_3|y_3S, R_3') \xrightarrow{\pi'_1}(\phi_2, \gamma_2|y_2S, R_2') \\
(\gamma_2y_2, R_2) & \xrightarrow{\tau(\pi'_1)R}(\gamma_3y_3, R_3)
\end{align*}$$

On the other hand, since the parser successfully executes the action $r'$ in the first derivation segment, it does not exit upon execution of lines 6-7 of the parsing loop. Thus, we conclude (by Lemma 5.4) that the AMP configuration $(S, \gamma_1|y_1S, R'_1)$ resulting from the call to $\text{ADAPT}$ is valid. We first handle the case where $r'$ is a Reduce action of the form $r' = (\text{Reduce}, r)$ where $r$ is an AMG rule. Let $\delta, \omega \in A_{N,T}^*$ such that $\gamma_1 = \delta\omega$, $\xi(r) = A \rightarrow \bar{\omega}$, and

$$\omega = (X_1, w_1)(X_2, w_2) \cdots (X_{|\omega|}, w_{|\omega|})$$

Then, the successful execution of line 10 of the main parsing loop implies that

$$\braket{\text{Reduce, } \xi(r)} \in \text{ACTION}([\gamma_{1|\xi(R'_1)}]^k, k : y_1S)$$

From (5.13), we have $[A \rightarrow X_1 \cdots X_{\bullet}, k : y_1S] \in [\gamma_{1|\xi(R'_1)}]^k$ and $A \neq S'$ which implies that

$$S' \Rightarrow S\bar{s} \Rightarrow \delta A\bar{s} \Rightarrow \delta \omega \bar{s} = \gamma_1z\bar{s}$$

where $k : z\bar{s} = k : y_1S$. Since $S'$ does not appear in the RHS of any rule in $R'_1$ we conclude that

$$S \Rightarrow \delta A \Rightarrow \delta \omega z = \gamma_1z \text{ in } G_{\xi}^{R'_1}$$
and that $k : z = k : y_1$. From lines 17-25 of the parsing loop, and by Lemma 5.2, we have

\begin{equation}
(R_n, R_r) = \Delta(R_1, \delta \omega, r) \quad \text{(lines 17-18)}
\end{equation}

\begin{equation}
w = f_r(w_1, \ldots, w_n) \quad \text{(lines 19-20)}
\end{equation}

\begin{equation}
(S, \delta \langle A, w \rangle | y_1 \$, R') \text{ is valid \quad (lines 21-25)}
\end{equation}

We now have two cases to consider: (1) $r \in R_{\overline{N}}^-$; and (2) $r \in R_{\overline{N}T}^-$. In the first case, we skip lines 27-30 of the parsing loop, and upon invocation of the \texttt{UpdateRules} routine on line 31, we get

\begin{equation}R'_3 = \text{Rules} = (R'_1 \cup R_n) \setminus R_r\end{equation}

and the iteration completes with the possibly invalid AMP configuration

\begin{equation}(S^\delta_{\xi(R'_1)}, \delta \langle A, w \rangle | y_1 \$, R'_3)\end{equation}

Given (5.12), (5.14), (5.15), (5.16), (5.17), and by Definition 4.9, we conclude that

\begin{equation}(\delta \langle A, w \rangle | y_1, R_3) \xrightarrow{r} (\delta \omega y_1, R_1) = (\gamma_1 y_1, R_1)
\end{equation}

By choosing $\gamma_3 = \delta \langle A, w \rangle$, and $y_3 = y_1$, and combining (5.11) and (5.18) we have

\begin{equation}(\gamma_2 y_2, R_2) \xrightarrow{\tau(\pi'_1)^R} (\gamma_3 y_3, R_3) \xrightarrow{r} (\gamma_1 y_1, R_1)
\end{equation}

\begin{equation}\tau(\pi'_1)^R = \tau(r \pi'_1)^R = \tau(\pi')^R\end{equation}

and thus conclude that statement (b) holds. In case (2), let $\delta = \rho v$ where $v \in A_T^\uparrow$ and $\rho : 1 \notin A_T$. Following the execution of lines 27-30 of the parsing loop, we obtain the valid AMP configuration $(S, \rho | v w y_1 \$, R'_1)$, and following the call to \texttt{UpdateRules}, the iteration completes with the possibly invalid AMP configuration

\begin{equation}(S^\delta_{\xi(R'_1)}, \rho | v w y_1 \$, R'_3)\end{equation}

Given (5.12), (5.14), (5.15), (5.16), (5.19), and by Definition 4.9, we conclude that

\begin{equation}(\rho v w y_1, R_3) \xrightarrow{r} (\delta \omega y_1, R_1) = (\gamma_1 y_1, R_1)
\end{equation}

Similarly to case (1), by choosing $\gamma_3 = \rho$, and $y_3 = v w y_1$, and combining (5.11) and (5.20) we conclude that statement (b) holds. Finally, we handle the case where $r' = \langle \text{Shift} \rangle$. Due to the fact that in the target configuration of the first derivation segment of (5.10) $M$’s input is $y_3 \$, we conclude that the shifted symbol is not $\$, and $y_1 = a y'_1$ for some terminal $a$. From lines 12-15 and Fact 5.3 we have

\begin{equation}(\phi, \gamma_1 | y_1 \$, R'_1) \xrightarrow{r'} (S, \gamma_1 t | y'_1 \$, R'_1)
\end{equation}

and since $\tau(r') = \varepsilon$, we also have

\begin{equation}(\gamma_1 t y'_1, R_1) = (\gamma_1 y_1, R_1) \xrightarrow{\tau(\pi'_1)^R} (\gamma_1 y_1, R_1)
\end{equation}

By choosing $\gamma_3 = \gamma_1 t$, $y_3 = y'_1$, $R_3 = R_1$ and combining (5.11) and (5.22), we have

\begin{equation}(\gamma_2 y_2, R_2) \xrightarrow{\tau(\pi'_1)^R} (\gamma_3 y_3, R_3) = (\gamma_1 y_1, R_1)
\end{equation}

\begin{equation}\tau(\pi'_1)^R = \tau(r' \pi'_1)^R = \tau(\pi')^R\end{equation}

and thus conclude that statement (b) holds.
Lemma 5.6. Let $G(N,T,R_0,S,\Delta,k)$ be an AMG and $M$ its AMP. Then $L(M) \subseteq L(G)$, and if $\pi'$ is an adaptive multi-pass parse of $x$ in $M$, $\tau(\pi')$ is an adaptive multi-pass parse of $x$ in $G$.

Proof. Let $x \in L(M)$. By Definition 5.2, there exists a rule set $R$, annotation string $w$, and action string $\pi'$ such that

$$((\epsilon,\epsilon|x\$),R_0\cup\{S' \rightarrow S\}) \xrightarrow{\pi'}((S,(S,w)\$,\$)\epsilon,R \cup \{S' \rightarrow S\})$$

Because, the top most symbol in $\text{SymStack}$ and action string $\pi$ of $\pi'$ and $\pi$ Let $\pi$ action in $\epsilon,\epsilon$ and if $\pi$ action in $\epsilon,\epsilon$

Lemma 5.7. Let $G(N,T,R_0,S,\Delta,k)$ be an AMG and $M$ its AMP. Further, let $R \subseteq R_{N,T}$, $R' = R \cup \{S' \rightarrow S\}$, $\gamma \in A_{N,T}^k$, and $x,y \in T^*$ such that

(a) $$[A \rightarrow a \bullet \beta,z] \in [\gamma x]^k_{\xi(R')}, \text{ and}$$
$$k : y\$ \in \text{FIRST}_k(\beta z)$$

Then there is an $n$-length action string $\pi'$ of Shift actions of $M$, and state string $\phi_2$, such that for any state string $\phi_1$,

(b) $$((\phi_1,\gamma|xy\$,R') \xrightarrow{\pi'}(\phi_2,\gamma|x\$,R')$$

Proof. We prove the lemma by showing that for iterations $i = 1, \ldots, |x|$ following the configuration $(\phi_1,\gamma|xy\$,R'), $M$ has a Shift action $r'_i$ at line 10 of the parsing loop

$$r'_i = \langle \text{Shift} \rangle \in \text{ACTION}([\gamma x]^k_{\xi(R')},z_i)$$

where $x,x'_i = x$, $|x_i| = i - 1$, and $z_i = k : x'_i y\$$. Observe that if that is the case, then on the first iteration, Lemma 5.4 guarantees that the execution of lines 6-7 completes successfully, and results with the valid AMP configuration $(S,\gamma|xy\$,R'). This is because (due to (a)) $[\gamma x]^k_{R'} \neq \emptyset$, which implies (by Lemma 5.2) that $[\gamma]^k_{R'} \neq \emptyset$. Furthermore, since $\text{adapt}$ is set to $\text{FALSE}$ on line 8, and when handling Shift actions (lines 11-15) the $\text{UPDATERULES}$ routine is not invoked, the $\text{ADAPT}$ routine is not invoked in all subsequent iterations. Thus, from lines 12-15 of the parsing loop and by statement (5.9) of Fact 5.3, we get

$$((\phi_1,\gamma|xy\$,R') = (\phi_1,\gamma|a_1 \cdots a_n y\$,R')$$
$$\xrightarrow{r'_1}(S,\gamma a_1 \cdots a_n y\$,R')$$
$$\vdots$$
$$\xrightarrow{r'_{n-1}}(S,\gamma a_1 \cdots a_{n-1} |a_n y\$,R')$$
$$\xrightarrow{r'_n}(S,\gamma a_1 \cdots a_n |y\$,R') = (\phi_2,\gamma x|y\$,R')$$
Lemma 5.8.

Statement (b) then holds if we choose $\pi' = r'_1 \cdots r'_n$. We now show that $M$ indeed has the Shift actions $r_1, \ldots, r_n$. Let $1 \leq i \leq n$ be an integer. Because of (a) we have for some $\delta$ and $z'$,

$$S' \xrightarrow{\delta} \delta A z' \Rightarrow \delta a \alpha z' = \gamma a_1 \cdots a_n \beta z' \text{ in } M_{\xi}'$$

We have two cases to consider: (1) $a_i$ is contained in $\alpha$, that is, $\alpha = \alpha' a_i \cdots a_n$; (2) $a_i$ is not contained in $\alpha$, that is $\delta = \gamma a_1 \cdots a_{j-1}$ and $\alpha = a_j \cdots a_n$ for some $j > i$. In case (1) we have (by Lemma 5.2)

$$[A \rightarrow \alpha' \bullet a_i \cdots a_n \beta, z] \in [\gamma \alpha']^k_{\xi(R)}$$

Moreover, the condition $k : \gamma y \in \text{FIRST}_k(\beta z)$ implies that

$$k : a_i \cdots a_n y \gamma \in \text{FIRST}_k(a_i \cdots a_n \beta z)$$

Since $[\gamma \alpha']^k_{\xi(R)}$ is the item set at the top of the state stack, that is provided as input to ACTION (lines 9-10), and the first $k$ symbols of $M$’s input stack, also provided as input, are $k : a_i \cdots a_n y \gamma$, we conclude that $M$ has the Shift action $r'$. In case (2), since $\delta \neq \varepsilon$, we can apply Lemma 5.1 to the derivation

$$S' \xrightarrow{\delta} \delta A z' = \gamma a_1 \cdots a_i \cdots a_{j-1} A z' \text{ in } M_{\xi}'$$

and conclude that for some strings $\delta'$ and $y'$ and rule $A' \rightarrow \alpha'' a_i \beta', \gamma y'$

$$S' \xrightarrow{\delta'} \delta A' y' \Rightarrow \delta' \alpha'' a_i \beta' y' = \gamma a_1 \cdots a_i \beta' y' \text{ and } \beta' y' \Rightarrow a_i \cdots a_{j-1} A z' \text{ in } M_{\xi}'$$

Thus we have

$$[A' \rightarrow \alpha'' \bullet a_i \beta', k : y'] \in [\gamma a_1 \cdots a_i \beta]_{\xi(R)}$$

Since $k : z' = z$, $y \gamma \in \text{FIRST}_k(\beta z)$, and

$$a_i \beta' y' \Rightarrow a_i \cdots a_{j-1} A z' \Rightarrow a_i \cdots a_n \beta z' \text{ in } M_{\xi}'$$

we also have

$$k : a_i \cdots a_n y \gamma \in \text{FIRST}_k(a_i \beta'(k : y'))$$

and thus conclude that in this case too $M$ has the Shift action $r'$.

In the next lemma we show that for every derivation of an AMG there exists a corresponding computation progression of its AMP.

Lemma 5.8. Let $G(N,T,R_0,S,\Delta,k)$ be an AMG and $M$ its AMP. Further, let $\gamma_1, \gamma_2 \in A_{N,T}^*$, $x_1, y_1, y_2 \in A_T$, $R_1, R_2 \subseteq R_{N,T}$, and $\pi$ a rule string in $R_{N,T}^*$ such that

(a) $\gamma_1 : 1, \gamma_2 : 1 \notin A_T$ and $(\gamma_2 y_2, R_2) \xrightarrow{\pi} (\gamma_1 x_1 y_1, R_1)$

and $|\pi| > 0$ implies that

$$A \rightarrow \omega = \xi(1 : \pi)$$

(b) $[A \rightarrow \omega \bullet, z] \in [\gamma_1 x_1]\xi(R_1')$, and

$k : \gamma_1 y \gamma = z$

Then there exists an action string $\pi'$, and state string $\phi_2$, such that for any state string $\phi_1$

(c) $(\phi_1, \gamma_1 | \bar{x}_1 \bar{y}_1 \gamma \gamma, R_1') \xrightarrow{\pi'} (\phi_2, \gamma_2 | \bar{y}_2 z \gamma, R_2')$

and $\tau(\pi') = \pi$.

where $R'_i = R_i \cup \{S' \rightarrow S\gamma\}$ for $i \in \{1, 2\}$. 

Proof. The proof is by induction on the length of the rule string $\pi$. If $|\pi| = 0$, statement (c) trivially holds if we choose $\phi_1 = \phi_2$, $\gamma_1 = \gamma_2$, $x_1 = \varepsilon$, $y_1 = y_2$, $R_1 = R_2$, and $\pi' = \varepsilon$. Assume then that $|\pi| > 0$ and that the induction hypothesis holds for strings shorter than $\pi$. Let $\pi = \pi_1 r$, and $\xi(r) = A \rightarrow \bar{w}$ where

$$\omega = \langle X_1, w_1 \rangle \cdots \langle X_n, w_n \rangle \in \mathcal{A}_{N,T}^*$$

Then we have

$$\omega \leftarrow \pi_1$$

where $x_3, y_3 \in \mathcal{A}_T^*$, $\delta, \gamma_3 \in \mathcal{A}_{N,T}^*$, $\gamma_3 : 1 \notin \mathcal{A}_T$, and $\delta \omega = \gamma_3 x_3$. By applying the induction hypothesis on the second derivation segment we conclude that there exists an action string $\pi'_1$, and state strings $\phi_1, \phi_3$ such that

$$\pi_1 = \pi_1' r_1$$

On the other hand, from the first derivation segment in (5.25), Definition 4.9, and (5.24) we have

$$S \xrightarrow{\delta} A z \xrightarrow{\xi(r)} \delta \omega z' \text{ in } C_{\xi}^{R_3} \text{ and } k : z' = k : y_3,$$

where $R_2 = (R_3 \cup R_a) \setminus R_r$ where $(R_a, R_r) = \Delta(R_3, \delta \omega, r), w = f_r(w_1, \ldots, w_n)$, and

$$\gamma_2 y_2 = \begin{cases} \delta(A, w) y_3 & r \in \mathcal{R}_{N,T}^- \\ \delta w y_3 & r \in \mathcal{R}_{N,T}^+ \end{cases}$$

Thus, we conclude that

$$[A \rightarrow \bar{w} \bar{\cdot}, \bar{y}_3 \bar{\cdot}] \in [\bar{y}_3 \bar{x}_3]_{\xi}^{k(R_3)}$$

which implies, by Lemma 5.7, that there exists an $|x_3|$-length action string $\pi'_2$ of Shift actions of $M$, and a state strings $\phi'_3$ such that

$$\pi_2 = \pi'_2 r_2$$

By Lemma 5.4 and (5.28) we conclude that given the configuration $(\phi'_3, \gamma_3 x_3 | \bar{y}_3 \bar{\cdot}, R'_3)$, the execution of lines 6-7 of the parsing loop completes successfully, and results with the valid configuration

$$\pi = \pi_1 r_1 \pi'_2 r_2$$

Furthermore, by (5.28) $M$ has a Reduce action

$$r' = \text{Reduce, } r \in \text{ACTION}([\bar{\delta} \bar{\omega}]_{\xi}^{k(R_3)}, k : \bar{y}_3 \bar{\cdot})$$

in line 10 of the parsing loop. Also, from lines 17-25 of the parsing loop, and by Lemma 5.2, we have

$$(R_a, R_r) = \Delta(R_3, \delta \omega, r) \quad \text{(lines 17-18)}$$

$w = f_r(w_1, \ldots, w_n) \quad \text{(lines 19-20)}$$

$$(\mathcal{S}, \delta(A, w) | \bar{y}_3 \bar{\cdot}, R'_3) \text{ is valid} \quad \text{(lines 21-25)}$$
We now have two cases to consider: (1) \( r \in R_{N,T} \); and (2) \( r \in R_{N,T}^{*} \). In the first case, we skip lines 27-30 of the parsing loop, and by the output of \( \Delta(R_{3}, r, \omega) \) given in (5.27), we conclude that following the call to \textsc{UpdateRules} on line 30, the iteration completes with the possibly invalid AMP configuration

\[(5.32) \quad \langle S_{\xi(R_{3})}^{\delta A}, \delta(A, w) | \bar{y}_{3} S, R'_{2} \rangle \]

By combining (5.26), (5.29), (5.30), (5.31), (5.32), and choosing \( \gamma_{2} = \delta(A, w) \) and \( y_{2} = y_{3} \) (in accordance with (5.27)), \( \phi_{2} = S_{\xi(R_{3})}^{\delta A}, \) and \( \pi' = \pi_{1}' \pi_{2}' r' \), we get

\[
(\phi_{1}, \gamma_{1}| \bar{x}_{1} \bar{y}_{1} S, R'_{1}) \xrightarrow{\pi'} (\phi_{2}, \gamma_{2}| \bar{y}_{2} S, R'_{2}) , \text{ and} \]

\[
\tau(\pi') = \tau(\pi_{1}' \pi_{2}' r') = \tau(\pi_{1}') \tau(\pi_{2}') \tau(r') = \pi_{1} r = \pi
\]

and thus conclude that statement (c) holds. In case (2), following the execution of lines 27-30 of the parsing loop, \( M \)'s state (in accordance with (5.27)) is the valid AMP configuration \( \langle S, \gamma_{2} \bar{y}_{2} S, R'_{3} \rangle \), and following the call to \textsc{UpdateRules}, the iteration completes with the possibly invalid AMP configuration

\[(5.33) \quad \langle S_{\xi(R_{3})}^{\gamma_{2}}, \gamma_{2}| \bar{y}_{2} S, R'_{2} \rangle \]

By combining (5.26), (5.29), (5.30), (5.31), (5.33), and choosing \( \phi_{2} = S_{\xi(R_{3})}^{\gamma_{2}}, \) and \( \pi' = \pi_{1}' \pi_{2}' r' \), we get

\[
(\phi_{1}, \gamma_{1}| \bar{x}_{1} \bar{y}_{1} S, R'_{1}) \xrightarrow{\pi'} (\phi_{2}, \gamma_{2}| \bar{y}_{2} S, R'_{2}) , \text{ and} \]

\[
\tau(\pi') = \tau(\pi_{1}' \pi_{2}' r') = \tau(\pi_{1}') \tau(\pi_{2}') \tau(r') = \pi_{1} r = \pi
\]

and thus conclude that statement (c) holds. \( \square \)

**Lemma 5.9.** Let \( G(N, T, R_{0}, S, \Delta, k) \) be an AMG and \( M \) its AMP. Further, let \( \gamma \in A_{N,T}^{*}, x, y \in A_{T}^{*}, R \subseteq R_{N,T}, \) and \( \pi \) a rule string in \( R_{N,T}^{*} \) such that \( \gamma : 1 \notin A_{T} \) and

\[
(\gamma y, R) \xrightarrow{\pi} (x, R_{0})
\]

Then there exists an action string \( \pi' \), and state string \( \phi \) such that \( \tau(\pi') = \pi \) and

\[
(\varepsilon, \varepsilon|x S, R'_{0}) \xrightarrow{\pi'} (\phi, \gamma|y S, R')
\]

where \( R' = R \cup \{ S' \rightarrow S \} \), and \( R'_{0} = R_{0} \cup \{ S' \rightarrow S \} \).

**Proof.** If \( \pi = \varepsilon, \gamma = \varepsilon, y = x \) and \( R = R_{0} \), and by choosing \( \phi = \varepsilon \) and \( \pi' = \varepsilon \) we have \( (\varepsilon, \varepsilon|x S, R'_{0}) \xrightarrow{\pi'} (\gamma|y S, R') \) and \( \tau(\pi') = \pi \). Otherwise, \( |\pi| > 0 \), and there exists a rule \( r \), rule string \( \pi_{1} \), rule set \( R_{1} \), and an annotated symbol string \( \alpha \) such that \( \pi = r \pi_{1} \), and

\[
(\gamma y, R) \xrightarrow{\pi_{1} R} (\alpha, R_{1}) \xrightarrow{r} (x, R_{0})
\]

By Definition 4.9 we have

\[
S \xrightarrow{\delta A z} \delta w z \text{ in } G_{\xi}^{R}
\]

where \( \xi(r) = A \rightarrow \omega, x = \delta w v \) and \( k : z = k : v \), which implies that

\[
S' \xrightarrow{S} \delta A z \xrightarrow{\delta \omega z} \delta w z \text{ in } M_{\xi}^{R_{0}}
\]
and $k : zS = k : vS$. Thus, we conclude that

$$[A \to \omega \cdot, k : vS] \in [\delta \omega]_{\xi(R_0^\prime)}^k$$

and by choosing $\gamma_2 = \gamma, y_2 = y$, $R_2 = R$, $\gamma_1 = \varepsilon, x_1 = \delta \omega, y_1 = v$, and $R_1 = R_0$, Lemma 5.8 implies that there exist a state string $\phi$, and action string $\pi$ such that

$$\langle \varepsilon, \varepsilon | \delta \omega \bar{v}, R_0 \rangle = \langle \varepsilon, \bar{x} | \bar{x}, R_0 \rangle \xrightarrow{\pi, (\phi, \gamma)} \langle \phi, \gamma | \bar{y}, R_0 \rangle$$

where $\tau(\pi) = \tau, R_0 = R_0 \cup \{S' \to S\}$ and $R = R \cup \{S' \to S\}$. \hfill \Box

**Lemma 5.10.** Let $G(N, T, R_0, S, \Delta, k)$ be an AMG and $M$ its AMP. Then $L(G) \subseteq L(M)$, and for any adaptive multi-pass parse of $x$ in $G$, there exists an adaptive multi-pass parse $\pi'$ of $x$ in $M$ such that $\tau(\pi') = \pi$.

**Proof.** Let $x \in L(G)$. By Definition 4.13 there exist a rule set $R$, and an annotation string $w$ such that

$$(\langle S, w \rangle, R) \xrightarrow{s}(x, R_0)$$

Lemma 5.9 implies (by choosing $\gamma = (S, w), y = \varepsilon, x = x$) that there exist an action string $\pi_1'$ and a state string $\phi$ such that $\tau(\pi_1') = \pi$ and

$$(\varepsilon, \varepsilon | \bar{x}, R_0) \xrightarrow{\pi_1'} (\phi, (S, w)|S, R')$$

where $R_0' = R_0 \cup \{S' \to S\}$ and $R' = R \cup \{S' \to S\}$. On the other hand, $S' \Rightarrow S' \Rightarrow S$ in $M_{R'}$ implies that

$$[S' \to S \cdot S, \varepsilon] \in [S]_{\xi(R')}^k$$

Since $[S]_{\xi(R')}^k \neq \emptyset$, Lemma 5.4 implies that given the AMP configuration $(\phi, (S, w)|S, R')$, the execution of lines 6-7 of the parsing loop will succeed and result with the valid configuration $(\phi, (S, w)|S, R')$. Furthermore, $M$ has a Shift action $\tau'$ at line 10 of the parsing loop, which implies that

$$(\phi, (S, w)|S, R') \xrightarrow{\tau'} (\phi, (S, w)|\varepsilon, R_0')$$

By combining (5.34) and (5.35), we have an action string $\pi' = \pi_1' \tau'$ such that

$$(\varepsilon, \varepsilon | x, R_0') \xrightarrow{\pi'} (\phi, (S, w)|\varepsilon, R')$$

Since $\tau(\pi') = \tau(\pi_1') \tau(\tau') = \tau(\pi_1') = \pi$, we conclude that the lemma holds. \hfill \Box

**Theorem 5.11.** Let $G(N, T, R_0, S, \Delta, k)$ be an AMG and $M$ its AMP. Then, $L(G) = L(M)$, and for each sentence $x \in L(G)$, $M$ produces exactly all adaptive multi-pass parses of $x$ in $G$. 
Chapter 6

Deterministic AMGs (DAMGs)

The nondeterministic AMG parser described in Chapter 5 operates deterministically as long as on each iteration of the main paring loop, the invocation of the ACTION routine (line 10) returns at most one parse action. A parser modified to abort erroneously if more than a single parse action is applicable, would accept the languages of only a proper subset of AMG grammars. We call such a parser a Deterministic AMP (or DAMP for short) and the subclass of AMGs which language can be properly recognized by such parsers deterministic AMGs (DAMGs).

The importance of DAMGs is that they facilitate the implementation of practical deterministic LR-based parsers for their languages. In this chapter we study their properties. We show that although the class of DAMGs is a proper subclass of AMGs (Theorem 6.1), for any AMG $G$ there exists a DAMG(k) $G'$ such that $L(G) = L(G')$ (Theorem 6.2), and that DAMG(k)s, like AMGs, define the class of recursively enumerable sets for all $k \geq 1$ (Theorem 6.3). We further show that derivations in DAMGs that behave like CFG rightmost derivations (i.e., involve no grammar modifications and no multi-pass rules) always correspond to shortest rightmost derivations in their underlying CFGs. We conclude the chapter by showing that in the general case, it is impossible to test whether an AMG is a DAMG. We further present an approximation algorithm for testing this property given bounds on the lengths of symbol annotations and on the number of distinct production rule sets to be tested.

6.1 Computational Power

Theorem 6.1. There exists an AMG that is nondeterministic.

Proof. Let $G(N, T, R_0, S, \Delta, k)$ be an AMG where $R_0 = \{ S \to a | S \}$ and $\Delta(R, r, \rho) = (\emptyset, \emptyset)$ for all $R, \rho,$ and $r$. By definition of $\Delta$, $G$ is not adaptive and performs all derivations according to the initial set of rules $R_0$. Let $q$ be the state residing at path $S$ from the start state of the prefix-automaton for the $\$$ augmented CFG $G'$: $S' \to S \$, S \to a | S$. The items

$[S' \to S \$, $, \varepsilon]$ and $[S \to S \$, $, \{\$$]}$

are both LR($k$) valid for the string $S$ with respect to $G'$ and are therefore contained in $q$. This in turn implies that

$(\text{Shift}), (\text{Reduce}, S \to S) \in \text{ACTION}(q, \$$)$

Thus we conclude that the AMG $G$ is nondeterministic. \qed
Deterministic LR(\(k\)) parsers define a proper subset of context-free languages that can be generated by LR(\(k\)) grammars. However, this is not the case with deterministic AMGs. The following theorem states that for every nondeterministic AMG \(G\) there exists a deterministic AMG \(G'\) of lookahead length \(k \geq 1\) such that \(L(G') = L(G)\).

**Theorem 6.2.** Let \(G\) be a non-deterministic AMG. For all \(k \geq 1\) there exists a DAMG(\(k\)) \(G'\) such that \(L(G') = L(G)\).

**Proof.** Let \(G\) be a nondeterministic AMG and let \(M_G\) be the Turing machine that runs the AMGParse algorithm for the grammar \(G\) and accepts or rejects its input according to the exit status of the parsing algorithm. Then according to Theorem 5.11, \(L(M_G) = L(G)\). Let \(G'\) be the AMG of Figure 4.4 constructed for \(M_G\). Theorem 4.3 implies that \(L(G') = L(M_G) = L(G)\) for \(k > 0\). According to Lemma 4.1 the initial rule set \(R_0\) of \(G'\) is used in every configuration appearing in the derivation of a terminal string in \(L(G')\). Since the CFG skeleton underlying \(G'\) with respect to \(R_0\) is LR(1), that is, its LR(1) prefix-automaton is deterministic, we conclude that \(G'\) with lookahead \(k \geq 1\) is deterministic. \(\Box\)

Immediate from Theorem 4.3 and Theorem 6.2 we have the following important property of deterministic AMGs.

**Theorem 6.3.** Let \(L\) be a recursively enumerable language. For all \(k \geq 1\) there exists a DAMG(\(k\)) \(G\) such that \(L(G) = L\).

Although AMGs and DAMGs possess the same computational power, nondeterministic AMGs allow for more concise language descriptions. Similarly to CFGs, often a “natural” grammar for describing a language is ambiguous, and thus nondeterministic. A common example is the “dangling-else” description of conditional statements in programming languages as in the nondeterministic grammar:

\[
S \rightarrow \text{if } B \text{ then } S \\
S \rightarrow \text{if } B \text{ then } S \text{ else } S \\
S \rightarrow a \\
B \rightarrow b
\]

In the case of the sentence

\[
\text{if } b \text{ then if } b \text{ then } a \text{ else } a
\]

the keyword else may be associated either with the first or the second then. This ambiguity can be removed if we arbitrarily decide that an else should be attached to the last preceding then. We can now formulate an equivalent, yet bigger and less intuitive deterministic grammar for conditional statements:

\[
S \rightarrow \text{if } B \text{ then } S \\
S \rightarrow \text{if } B \text{ then } S' \text{ else } S \\
S \rightarrow a \\
S' \rightarrow \text{if } B \text{ then } S' \text{ else } S' \\
S' \rightarrow a \\
B \rightarrow b
\]

In practice, this disadvantage of DAMGs can be easily diminished by employing syntactic disambiguation rules [2], which allow for deterministic parsing of ambiguous languages by selecting a unique, or “canonical” parse for each sentence of the language. The design and implementation of a disambiguation system suitable for adaptive grammars is an area for future research as described in Chapter 9.
6.2 CFG-Like Derivations

In the remainder of this chapter we formally define CFG-like AMG derivations and show that in deterministic AMGs they correspond to shortest CFG rightmost derivations.

**Definition 6.1** (CFG-like AMG derivation). Let $G(N, T, R'_0, S, \Delta, k)$ be an AMG. An AMG derivation $d = ((\alpha_0, R_0), (\alpha_1, R_1), \ldots, (\alpha_n, R_n))$ by rule string $\pi$ in $G$ where $n \geq 0$ is said to be CFG-like with respect to $R_n$ if $\pi \in R_{\Delta, k}^{\ast}$ and $R_1 = R_2 = R_3 \cdots = R_n$.

**Lemma 6.4.** Let $G(N, T, R_0, S, \Delta, k)$ be an AMG, and $d = ((\alpha_0, R_0'), (\alpha_1, R), \ldots, (\alpha_{\mid \pi \mid}, R))$ a CFG-like derivation with respect to $R$ by rule string $\pi$ in $G$. Then $\vec{d} = (\vec{\alpha}_0, \vec{\alpha}_1, \ldots, \vec{\alpha}_{\mid \pi \mid})$ is a rightmost derivation of $\vec{\alpha}_{\mid \pi \mid}$ from $\vec{\alpha}_0$ by rule string $\vec{\pi}(\pi)$ in $G^R_{\xi}$.

**Proof.** The proof is by induction on the length of $\pi$. If $\mid \pi \mid = 0$, $\alpha_0 = \alpha_{\mid \pi \mid}$ and $(\vec{\alpha}_0)$ is the required rightmost derivation by rule string $\vec{\pi}(\pi) = \varepsilon$ in $G^R_{\xi}$. Assume then, that $\mid \pi \mid > 0$ and, as the induction hypothesis, that the lemma holds for rule strings shorter than $\pi$. Since $\mid \pi \mid > 0$ we can write $\pi = r\pi'$ and $(\alpha_0, R') \xrightarrow{r}(\alpha_1, R) \xrightarrow{\pi'}(\alpha_{\mid \pi \mid}, R)$.

Since $\mid \pi' \mid < \mid \pi \mid$ we can apply the induction hypothesis to the second derivation segment and conclude that $(\vec{\alpha}_1, \ldots, \vec{\alpha}_{\mid \pi \mid})$ is a rightmost derivation by rule string $\vec{\pi}(\pi')$ in $G^R_{\xi}$. On the other hand, since $r \in R_{\Delta, k}^{\ast}$, Definition 4.9 implies that $\alpha_0 = \rho(A, w)x$, $\alpha_1 = \rho\omega x$, $r \in R$, and $\xi(r) = A \rightarrow \omega$ where $\omega, \rho \in A^*_N, x \in A^*_T$, and $w \in T^*$. Since $\bar{\alpha}_0 = \bar{\rho}A\bar{x}(\xi(r))\bar{\rho}\bar{\omega}x = \bar{\alpha}_1$ in $G^R_{\xi}$, we conclude that $(\bar{\alpha}_0, \ldots, \bar{\alpha}_{\mid \pi \mid})$ is a rightmost derivation by rule string $\vec{\pi}(\pi)$ in $G^R_{\xi}$. □

**Lemma 6.5.** Let $G'(N', T', P', S')$ be a $\Sigma$-augmented CFG, and $I_1 = [A \rightarrow \omega \bullet z] \in [\gamma]^k_{G'} (k > 0)$. Then $[A' \rightarrow \alpha \bullet \beta, y] \in [\gamma]^k_{G'}$ such that $\beta \Rightarrow x_\mu \beta'$, $x \neq \varepsilon$, $\mu \notin T^*$, and $z \in \text{First}_k(x_\mu y)$ implies that $[\gamma]^k_{G'}$ contains an item that exhibits a shift-reduce conflict with $I_1$.

**Proof.** We first note that $\beta \Rightarrow x_\mu \beta'$ implies that $z \in \text{First}_k(\beta y)$. Let $I_2 = [A' \rightarrow \alpha \bullet \beta, y]$. If $1 : \beta = t \in T'$, we can write $\beta = t\beta'$ and since $z \in \text{First}_k(\beta y)$, we also have $z \in t\text{First}_{k-1}(\beta y)$ and thus conclude that $I_2$ exhibits a shift-reduce conflict with $I_1$. Otherwise, $1 : \beta \in N'$ and there exists a nonempty rule string $\pi$ such that $\beta \Rightarrow x_\mu \beta'$. By Definition 2.1, $I_2 \in [\gamma]^k_{G'}$ implies that $S' \Rightarrow \gamma v \beta v$ where $v \in T^*$ and $y = k : v$. Let $\pi_1$ be the shortest prefix of $\pi$ such that $\beta \Rightarrow x_\mu \rho$ where $u$ is a nonempty prefix of $x$. 1 : $\beta \in N'$ and 1 : $u \in T'$ implies that $\mid \pi_1 \mid > 0$ and we can write $\pi_1 = \pi_2 r$. Furthermore, since $\mu$ consists of at least one nonterminal, the RHS of $r$ is not $\varepsilon$, and we can write $r = B \rightarrow u\rho'$ such that $\rho = \rho'\rho''$ and $S' \Rightarrow \gamma v \beta v \Rightarrow \gamma B\rho'' v \Rightarrow \gamma u\rho' \rho'' v = \gamma u\rho v$. Thus, By Definition 2.1 we conclude that $I_3 = [B \rightarrow \bullet u\rho', k : \rho'' v] \in [\gamma]^k_{G'}$. Since $\text{First}_k(u\rho'(k : \rho'' v)) = \text{First}_k(u\rho'(k : \rho v)) = \text{First}_k(u\rho'(k : v)) = \text{First}_k(u\rho y)$ and $u\rho \Rightarrow x_\mu$, we have $\text{First}_k(x_\mu y) \subseteq \text{First}_k(u\rho y)$ and $z \in \text{First}_k(u\rho'(k : \rho'' v))$. By writing $u = tu'$ where $t \in T'$ we have $z \in \text{First}_k(u\rho'(k : \rho'' v)) = t\text{First}_{k-1}(u'\rho'(k : \rho'' v))$ and thus conclude that $I_3 = [B \rightarrow \bullet t u'\rho', k : \rho'' v]$ exhibits a shift-reduce conflict with $I_1$. □
Lemma 6.6. Let \( G'(N', T', P', S') \) be a \$\$\$-augmented CFG, \( k > 0 \), \( \delta_1, \delta_2 \) symbol strings, \( x_1, x_2 \) terminal strings, \( r_1 = A_1 \rightarrow \omega_1 \), \( r_2 = A_2 \rightarrow \omega_2 \) rules, and \( \pi_1, \pi_2 \) rule strings such that \( S' \rightarrow \delta_1 A_1 x_1 \), \( \delta_1 \omega_1 x_1 \), \( S' \rightarrow \delta_2 A_2 x_2 \), \( \delta_2 \omega_2 x_2 \), \( \delta_1 A_1 \neq \delta_2 A_2 \), and \( \delta_1 \omega_1 (k : x_1) \) is a prefix of \( \delta_2 \omega_2 x_2 \). Then if \( |\delta_1 \omega_1| \leq |\delta_2 \omega_2| \), \( [\delta_1 \omega_1]_{G'}^k \) contains a pair of items exhibiting a reduce-reduce or shift-reduce conflict with respect to the lookahead string \( k : x_1 \), and if \( |\delta_2 \omega_2| < |\delta_1 \omega_1| \), \( [\delta_2 \omega_2]_{G'}^k \) contains a pair of items exhibiting a shift-reduce conflict with respect to the lookahead string \( k : y_1 x_1 \) where \( y_1 \in T^+ \), and \( \delta_1 \omega_1 = \delta_2 \omega_2 y_1 \).

Proof. If \( |\delta_1| = |\delta_2| \) we have \( A_1 \neq A_2 \), \( \delta_1 = \delta_2 \) and \( \omega_1 (k : x_1) \) is a prefix of \( \omega_2 x_2 \). If \( |\omega_1| \leq |\omega_2| \) there exists a terminal string \( y_2 \) such that \( \omega_2 = \omega_1 y_2 \) and \( k : x_1 = k : y_2 x_2 \). Thus, we have

\[
[A_1 \rightarrow \omega_1 \bullet k : y_2 x_2], [A_2 \rightarrow \omega_1 \bullet y_2, k : x_2] \in [\delta_1 \omega_1]_{G'}^k
\]

and since \( k : y_2 x_2 = k : y_2 (k : x_2) \in \text{FIRST}_k(y_2 (k : x_2)) \), we conclude that \( y_2 = \varepsilon \) implies that these items exhibit a reduce-reduce conflict, and \( y_2 \neq \varepsilon \) implies that they exhibit a shift-reduce conflict. If \( |\omega_1| > |\omega_2| \) there exists a nonempty terminal string \( y_1 \) such that \( \omega_1 = \omega_2 y_1 \) and \( k : x_2 = k : y_1 x_1 \). Thus, we have

\[
[I = A_1 \rightarrow \omega_1 \bullet k : y_2 \omega_2 x_2] \in [\delta_1 \omega_1]_{G'}^k
\]

Furthermore, \( \delta_2 \neq \varepsilon \) implies that \( \delta_2 A_2 x_2 \neq S' \) and thus \( |\pi_2| > 0 \); by choosing \( \gamma = \delta_1 \omega_1 \), \( \eta = y_2 A_2 \) and \( y = x_2 \), Lemma 5.1 implies that there exist a symbol string \( \delta' \), terminal string \( y' \), and rule \( A' \rightarrow \alpha' \beta' \) such that

\[
S' \rightarrow \delta' A' y' \rightarrow \delta' \alpha' \beta' y' = \gamma \beta' y' = \delta_1 \omega_1 \beta' y' \text{ and } \beta' y' \rightarrow \eta y = y_2 A_2 x_2
\]

Thus, by writing \( x_2 = x_2' y' \) we have \( [A' \rightarrow \alpha' \bullet \beta', k : y'] \in [\delta_1 \omega_1]_{G'}^k, \beta' \rightarrow \ast y_2 A_2 x_2' \),

\[
k : y_2 \omega_2 x_2 \in \text{FIRST}_k(y_2 A_2 x_2) = \text{FIRST}_k(y_2 A_2 x_2' y') = \text{FIRST}_k(y_2 A_2 x_2' (k : y'))
\]

and Lemma 6.5 implies that there exists an item in \( [\delta_1 \omega_1]_{G'}^k \) that exhibits a shift-reduce conflict with \( I \). If \( |\delta_2| \leq |\delta_1 \omega_1| \leq |\delta_2 \omega_2| \), there exist symbol strings \( \omega'_1, \omega'_2 \) and a terminal string \( y_2 \) such that \( \omega_1 = \omega'_1 \omega'_2 \), \( \omega'_1 \neq \varepsilon \), \( k : x_1 = k : y_2 x_2 \), and \( \omega_2 = \omega'_2 y_2 \). Thus, we have

\[
[A_1 \rightarrow \omega'_1 \omega_2 \bullet k : y_2 x_2], [A_2 \rightarrow \omega'_2 \bullet y_2, k : x_2] \in [\delta_1 \omega_1]_{G'}^k
\]

and since \( k : y_2 x_2 = k : y_2 (k : x_2) \in \text{FIRST}_k(y_2 (k : x_2)) \), we conclude that \( y_2 = \varepsilon \) implies that these items exhibit a reduce-reduce conflict, and \( y_2 \neq \varepsilon \) implies that they exhibit a shift-reduce conflict. If \( |\delta_2 \omega_2| < |\delta_1 \omega_1| \), there exist a nonempty symbol string \( \omega'_1 \) and a nonempty terminal string \( y_1 \), such that \( \delta_1 \omega'_1 = \delta_2, \omega_1 = \omega'_1 \omega_2 y_1 \) and \( k : x_2 = k : y_1 x_1 \). Thus, we have

\[
[A_1 \rightarrow \omega'_1 \omega_2 \bullet y_1, k : x_1], [A_2 \rightarrow \omega'_2 \bullet y_1 x_1] \in [\delta_2 \omega_2]_{G'}^k
\]

and since \( k : y_1 x_1 = k : y_1 (k : x_1) \in \text{FIRST}_k(y_1 (k : x_1)) \) we conclude that these items exhibit a shift-reduce conflict. Lastly, we handle the case \( |\delta_1| > |\delta_2| \). If \( |\delta_2 \omega_2| < |\delta_1| \), there
exists a nonempty terminal string $y_2$ such that $k : x_2 = k : y_2\omega_1 x_1$ and $\delta_1 = \delta_2\omega_2 y_2$. Thus, $y_1 = y_2\omega_1 \in T^*$ such that $\delta_1\omega_1 = \delta_2\omega_2 y_1$, and we have

$$I = [A_2 \to \omega_2 \cdot, k : y_1 x_1] \in [\delta_2\omega_2]_{G'}$$

Furthermore, $\delta_1 \neq \varepsilon$ implies that $\delta_1 A_1 x_1 \neq S'$ and thus $|\pi_1| > 0$; by choosing $\gamma = \delta_2\omega_2$, $\eta = y_2 A_1$ and $y = x_1$, Lemma 5.1 implies that there exist a symbol string $\delta'$, terminal string $y'$, and rule $A' \to \alpha' \beta'$ such that

$$S' \Rightarrow^{*}\delta' A' y' \Rightarrow^{*}\delta' \alpha' \beta' y' = \gamma \beta' y' = \delta_2\omega_2 \beta' y' \text{ and } \beta' y' \Rightarrow^{*}\eta y = y_2 A_1 x_1$$

Thus, by writing $x_1 = x'_1 y'$ we have $[A' \to \alpha' \cdot \beta', k : y'] \in [\delta_2\omega_2]_{G'}$ and Lemma 6.5 implies that there exists an item in $[\delta_2\omega_2]_{G'}$ that exhibits a shift-reduce conflict with $I$. If $|\delta_1| \leq |\delta_2\omega_2| < |\delta_1\omega_1|$, there exist symbol strings $\omega_1', \omega_2'$ and a nonempty terminal string $y_1$ such that $\omega_2 = \omega_2' \omega_1'$, $\omega_2' \neq \varepsilon$, $k : x_2 = k : y_1 x_1$, and $\omega_1 = \omega_1' y_1$. Thus, we have

$$[A_1 \to \omega_1' \cdot y_1, k : x_1], [A_2 \to \omega_2' \cdot, k : y_1 x_1] \in [\delta_2\omega_2]_{G'}$$

and since $k : y_1 x_1 = k : y_1(k : x_1) \in \text{FIRST}_k(y_1(k : x_1))$, we conclude that these items exhibit a shift-reduce conflict. If $|\delta_1\omega_1| \leq |\delta_2\omega_2|$, there exist a nonempty symbol string $\omega_2'$ and a terminal string $y_2$, such that $\delta_2\omega_2' = \delta_1$, $\omega_2 = \omega_2' y_2$ and $k : x_1 = k : y_2 x_2$. Thus, we have

$$[A_1 \to \omega_1' \cdot y_1, k : x_1], [A_2 \to \omega_\omega_1 y_2 \cdot, k : y_2 x_2] \in [\delta_1\omega_1]_{G'}$$

and since $k : x_1 = k : y_2 x_2 = k : y_2(k : x_2) \in \text{FIRST}_k(y_2(k : x_2))$ we conclude that $y_2 = \varepsilon$ implies that these items exhibit a reduce-reduce conflict, and $y_2 \neq \varepsilon$ implies that they exhibit a shift-reduce conflict.

\[\Box\]

**Lemma 6.7.** Let $G(N, T, R_0, S, \Delta, k)$ be a deterministic AMG, $\pi$ an adaptive multi-pass parse of $u$ in $G$, $d$ the derivation implied by $\pi^R$, $\alpha, \beta \in A^*_N$, and $d_1$ a CFG-like subderivation of $d$ with respect to $R \subseteq R_{N, T}$ of $\beta$ from $\alpha$ by rule string $\pi_1$ in $G$. Then $d_1$ is the shortest rightmost derivation of $\beta$ from $\tilde{\alpha}$ in $G^R_\tilde{\alpha}$.

**Proof.** Let $M$ be the AMP of $G$, $R' = r \cup \{S' \to S$\} and let $M^R_\tilde{\alpha} = (N', T', P', S')$ be $\$-$augmented CFG of $G^R_\tilde{\alpha}$. If $|\pi_1| = 0$, $\tilde{\alpha} = \beta$ and $\tilde{d} = (\tilde{\alpha})$ is the shortest derivation of $\beta$ from $\tilde{\alpha}$ in $G^R_\tilde{\alpha}$. If $|\pi_1| > 0$, $d_1$ contains at least one derivation step. Definition 4.9 then implies with respect to the first derivation step in $d_1$, that we can write $\alpha = \alpha_1 x$ where $x \in A_T$, $\tilde{\alpha}_1 : 1 \in N$, and $S \Rightarrow^{*} \tilde{\alpha}_1 v$ in $G^R_\tilde{\alpha}$ where $k : v = k : \tilde{\alpha}_1$, which further implies that $S' \Rightarrow^{*} \tilde{\alpha}_1 v$ in $M^R_\tilde{\alpha}$. Since $d_1$ is CFG-like, we can also write $\beta = \beta_1 x$. Furthermore, for each derivation step (by rule $r_1$) in $d_1$ there exists rule sets $R_1, R_2$, rule strings $\pi_2, \pi_3$, annotated symbol strings $\delta_1, \omega_1$, an annotated terminal string $z_1$, a terminal string $w$, and a nonterminal $A_1$ such that $\pi_1 = \pi_2 r_1 \pi_3$, $\tilde{r}_1 = \xi(r_1) = A_1 \to \tilde{\omega}_1$, and

$$(\alpha, R_1) = (\alpha_1 x, R_1) \xrightarrow{\pi_2}(\delta_1 \langle A_1, w \rangle z_1 x, R_2) \xrightarrow{r_1}(\delta_1 \omega_1 z_1 x, R) \xrightarrow{\pi_3}(\beta_1 x, R) = (\beta, R)$$

where $\pi_2 = \varepsilon$ implies that $R_2 = R_1$ and $\pi_2 \neq \varepsilon$ implies that $R_2 = R$. Definition 4.9 then implies that there exists a terminal string $x_1 \in T^*$ such that $k : x_1 = k : \tilde{z}_1 \tilde{x}$ and $S \Rightarrow^{*} \delta_1 A_1 x_1 \xrightarrow{\tilde{r}_1} \tilde{\delta}_1 \tilde{\omega}_1 x_1$ in $G^R_\tilde{\alpha}$, which further implies that $S' \Rightarrow^{*} \delta_1 A_1 x_1 \tilde{s}_1 \xrightarrow{\tilde{r}_1} \tilde{\delta}_1 \tilde{\omega}_1 x_1 \$ in $M^R_\tilde{\alpha}$ and $[A_1 \to \tilde{\omega}_1 \cdot, k : z_1 \tilde{x}] \in [\tilde{\delta}_1 \tilde{\omega}_1]_{(\tilde{\alpha}')}$. Let $\gamma$ be the longest prefix of $\delta_1 \omega_1$ such that $\gamma = 1 \notin A_T$, and let $\delta_1 \omega_1 = \gamma y$. Then, Lemma 5.5 implies (by choosing $\gamma_1 = \gamma$, $x_1 = \tilde{y}$, $y_1 = y_2 = \tilde{z}_1 \tilde{x}$,
and $\gamma_2 = \delta_1(A_1, w)$) that there exist an action string $\pi'_1$ and a state string $\phi_2$ such that for any state string $\phi_1$

$$(\phi_1, \gamma | y \tilde{z}_1 x \$$, $R') \xrightarrow{\pi'_1} (\phi_2, \delta_1(A_1, w) | \tilde{z}_1 x \$$, $R'_2)$$

in $M$ where $R' = R \cup \{S' \rightarrow SS\}$ and $R'_2 = R_2 \cup \{S' \rightarrow SS\}$, and $\tau(\pi'_1) = r$. Let $\pi_6, \pi_7$ be rule strings such that $\pi^R = \pi_6^R R_1 \pi_7^R$. Since

$$(\delta_1 \omega_1 z_1 x, R) = (\gamma y, R) \xrightarrow{\pi^R}(u, R_0)$$

Lemma 5.9 implies that there exist an action string $\pi'_7$ and a state string $\phi_1$ such that $\tau(\pi'_7) = \pi_7$ and

$$(\epsilon, \epsilon | u \$$, $R_0') \xrightarrow{\pi'_7}(\phi_1, \gamma | y \tilde{z}_1 x \$$, $R')$$

where $R'_0 = R_0 \cup \{S' \rightarrow SS\}$. Theorem 5.11 implies that $M$ produces $\pi$. Since $G$ is deterministic, $M$ has only one computation path for producing $\pi$ and thus we conclude that $M$ executes $\pi'_1$ when producing $\pi$. Assume towards a contradiction that there exists a rightmost derivation of $\beta$ from $\alpha$ by rule string $\tau_5$ in $G^R$ that is shorter than $\bar{d}_1$. Let $\tau$ be the shortest rule string such that $S' \Rightarrow \bar{d}_1 v$ in $M^R$. If $\tau_5$ is a suffix of $\xi(\pi_1)$, there exists rule strings $\pi_4, \pi_5$ such that $\tau_5 = \xi(\pi_5), \bar{\pi}_1 = \pi_4 \pi_5, \pi_4 \neq \epsilon$, and we can write

$$(\alpha, R_1) = (\alpha_1 x, R_1) \xrightarrow{\pi_4}(\alpha_1 x, R) = (\alpha, R) \xrightarrow{\pi_5}(\beta, R)$$

The shortness of $\tau$ implies that $\xi(\pi_4)$ is not its suffix, and since $S'$ does not appear at the RHS of any rule in $P'$, neither $\tau$ is a suffix of $\xi(\pi_4)$. Since $\pi_2 R_1 = \pi_4$ implies that $\bar{\delta}_1 \omega_1 = \alpha_1, z = \epsilon$, and $\bar{\alpha}_1(k : x_1) = \bar{\alpha}_1(k : \tilde{x}) = \bar{\alpha}_1(k : v)$, and since $S' \Rightarrow \bar{d}_1 v$ in $M^R$, we conclude that there exists a derivation step by rule $r_1$ in $\pi_4$, a rule $r_2$ in $\pi$, symbol strings $\bar{\delta}_2, \omega_2$, a terminal string $x_2$, and a nonterminal $A_2$ such that $\tau_2 R_2$ is a prefix of $\tau, S' \Rightarrow \bar{\delta}_2 A_2 x_2 \Rightarrow \bar{\delta}_2 \omega_2 x_2, \bar{\delta}_1 A_1 \neq \bar{\delta}_2 A_2$ and $\bar{\delta}_1 \omega_1(k : x_1)$ is a prefix of $\delta_2 \omega_2 x_2$. If $|\bar{\delta}_1 \omega_1| \leq |\delta_2 \omega_2|$, Lemma 6.6 implies that the state $[\bar{\delta}_1 \omega_1]_{\xi(\pi_4)}^k$ exhibits an LR conflict for lookahead $x : \tilde{x}_1$. Otherwise, $\delta_2 \omega_2$ is a proper prefix of $\delta_1 \omega_1$, and Lemma 6.6 implies that the item set $[\delta_2 \omega_2]_{\xi(\pi_4)}^k$ consists of conflicting items with respect to the lookahead string $k : y_1 \tilde{x}_1$ where $y_1 \neq \epsilon$ and $\bar{\delta}_1 \omega_1 = \delta_2 \omega_2 y_1$. Since $\gamma$ is a prefix of both $\bar{\delta}_1 \omega_1$ and $\delta_2 \omega_2$, we conclude that while producing $\pi_4$, $M$ will pass through state $[\delta_2 \omega_2]_{\xi(\pi_4)}^k$ with lookahead $k : y_1 \tilde{x}_1$, and state $[\bar{\delta}_1 \omega_1]_{\xi(\pi_4)}^k$ with lookahead $k : \tilde{x}_1$. This implies that $M$ is nondeterministic, in contradiction to $G$ being deterministic. We next handle the case where $\tau_5$ is not a suffix of $\xi(\pi_1)$. $|\tau_5| < |\xi(\pi_1)|$ implies that neither is $\xi(\pi_1)$ a suffix of $\tau_5$. Since $\pi_3 = \epsilon$ implies that there exists a terminal string $x'_1$ such that $\tilde{x}_1 = x'_1 x, \bar{\delta}_1 \omega_1 x'_1 = \beta_1$, and $\bar{\delta}_1 \omega_1(k : x)$ is a prefix of $\beta_1(k : v)$, and since $S' \Rightarrow \bar{\alpha}_1 v \Rightarrow \beta_1 v$, we conclude that there exists a derivation step by rule $r_1$ in $\pi_1$, a rule $r_2$ in $\tau_5$, symbol strings $\bar{\delta}_2, \omega_2$, terminal strings $x_2, x'_2$, and a nonterminal $A_2$ such that $\tau_2 R_2$ is a prefix of $\tau_5, S' \Rightarrow \bar{\alpha}_1 v \Rightarrow \bar{\delta}_2 A_2 x'_2 \Rightarrow \bar{\delta}_2 \omega_2 x'_2, \bar{\delta}_1 A_1 \neq \bar{\delta}_2 A_2$ and $\bar{\delta}_1 \omega_1(k : x_1)$ is a prefix of $\delta_2 \omega_2 x_2$. By the same reasoning applied for the case where $\tau_5$ is a suffix of $\xi(\pi_1)$ we conclude that $M$ is nondeterministic, in contradiction to $G$ being deterministic. Thus, we conclude that the existence of a shorter derivation of $\beta$ from $\alpha$ in $G^R$ contradicts the determinism of $G$ which implies that $\bar{d}_1$ is the shortest.

\section{6.3 Testing for Determinism}

A problem that arises from the definition of DAMGs, is that given an AMG, it is impossible, in the general case, to test whether or not it is deterministic. This is due to two
reasons: (1) the number of distinct production rule sets that may result due to grammatical modifications is potentially infinite and depends on the derived (or parsed) sentence, and (2) the grammar’s adaptation function may loop forever on some inputs. While for most practical applications, an adaptation function that halts on all inputs can be easily specified, formally proving the LR($k$) property of all possible production rule sets, is not practical.

For some applications of AMGs, where the user of a language is explicitly allowed to manipulate its syntax (e.g., modeling of syntax macro systems in page 5), lack of testing (beyond the base rule set) is acceptable. In face of nondeterminism, the user can reformulate the syntax modifications such that they would allow for deterministic parsing. However, in cases where production rules modifications are implicitly implied by the grammar, a nondeterminism-related failure for what should otherwise be a valid sentence of the language would be considered by the user as an error in the definition of the language. Therefore, tools for approximate DAMG testing are required for practical application of AMGs.

Figure 6.1 presents an approximation algorithm TestDAMG that tests whether a given AMG is a DAMG by exhaustively testing the LR($k$) property of each production rule set that can result during parsing, given bounds on the lengths of symbol annotations ($MAX_A$), and the number of distinct production rule sets to test ($MAX_\Delta$). The algorithm is guaranteed to halt if the input grammar’s adaptation function halts on all inputs. The routine TestDAMG takes an AMG $G$ and returns True if it can determine that $G$ is a DAMG (with respect to $MAX_A$ and $MAX_\Delta$). It starts by initializing a history set $H$ that contains all production rule sets that have been found to be LR($k$), and a work set $W$ that contains production rule sets to be tested (lines 1-2). Associated with each vocabulary symbol $X$, is an initially empty set of possible annotations $A(X)$. In lines 3-4, the annotations set of each terminal $a$ is set to contain the terminal $a$. The main loop of the algorithm (lines 5-17) is executed as long as the work set is not empty, and the number of tested rule sets did not exceed $MAX_\Delta$. On each iteration, a single rule set $R$ is chosen from the work set and its LR($k$) property is tested (line 7). The test can be performed by constructing the full LR($k$) prefix-automaton for the grammar $\xi(R)$ and verifying that none of its states contains an LR($k$) conflict. A more efficient, polynomial-time technique for testing the LR($k$) property of a CFG is described in [43]. If the grammar $\xi(R)$ is not LR($k$), the algorithm returns False to indicate that it cannot determine that the grammar $G$ is DAMG (line 8). Note that this result does not imply that $G$ is not a DAMG, since it is possible that the tested production rule set $R$ will not be generated by the parser for any input sentence, or that rogue states detected will not be generated by the parser (which lazily generates only states that are pushed onto the parse stack) for any input sentence. If the tested production rule set $R$ is found to be LR($k$), the sets of all possible annotations that can be associated with nonterminals of $R$ are computed, by invoking the routine ComputeAnnotations (line 9). The routine repeatedly synthesizes new nonterminal annotations from existing ones by simulating reduce actions until no more annotations (of length shorter than $MAX_A$) can be synthesized. Next, all possible grammar modifications resulting from $R$ are computed by simulating all possible adaptation function invocations due to reductions by rules in $R$ (lines 10-15). Resulting rule sets that have not been previously tested, are added to the work set $W$ to be tested in subsequent iterations of the main loop (lines 14-15). The loop concludes by removing the rule set $R$ from the work set and adding it to the history set (lines 16-17). When no more rule sets are available in the work set, or the maximal number of LR($k$) tests have been (successfully) performed, the main loop of the algorithm is exited, and the algorithm returns True to indicate that $G$ is a DAMG.
The algorithm TestDamg is useful for testing the DAMG property of AMGs whose derivations consist of annotations of bounded length, or for which a bound on the length of derivations results with a sufficiently comprehensive representative set of annotations. Since the algorithm is executed only by language developers during the development of an AMG based languages, its efficiency is of secondary importance. The efficiency and accuracy of the algorithm can be improved in many ways. However, these fall outside the scope of this thesis and are considered an area for future work.
Chapter 7

DAMG Derivation Complexity

In this chapter, we consider the time and space complexities of AMG derivations. We show that given a fixed bound on the size of rule sets and number of grammar modifications, the complexity of complete AMG derivations of deterministic AMGs is linear in the length of the derived terminal strings (see Section 7.2). As a preliminary step, we establish bounds on the time and space complexity of rightmost derivations in context-free-grammars.

7.1 CFG Rightmost Derivation Complexity

In this section, we set bounds on the time and space complexities of rightmost derivations of sentences in a free-form context-free grammar and formally prove their validity. We identify three grammar dependent properties, namely nullability depth, nullability width and longest loopless path by which the bounds are stated with respect to the length of the first and last sentential forms of a derivation. For the special case where the first sentential form is a nonterminal and the last sentential form is a terminal string, we prove that our bounds are tight in the sense that there exists a sequence of grammars that actually attain them.

The proofs of this section are organized as follows. First we determine bounds on derivations of nonempty symbol strings in $\varepsilon$-free grammars (Lemma 7.2), and show that these bounds are tight when deriving nonempty terminal strings (Theorem 7.3). Then, we consider grammars in which the empty string can be derived from some nonterminal, and determine tight bounds on such derivations (Theorem 7.4). Next, we determine bounds on derivations of nonempty symbol strings in a free-form grammar $G$. Initially, these bounds are formulated in terms of the properties of derivations of an $\varepsilon$-free grammar $\hat{G}$ constructed from $G$ (Lemma 7.5). Then, using a series of helping lemmas we formulate these bounds in terms of properties of the grammar $G$ (Lemma 7.8), and conclude by showing that these bounds are tight when deriving nonempty terminal strings (Theorem 7.9). We start by defining the time and space complexity of CFG rightmost derivations.

Definition 7.1 (Rightmost derivation complexity).

Let $G(N,T,P,S)$ be a context-free grammar, and $d = (\gamma_0, \ldots, \gamma_n)$ be a rightmost derivation of length $n$ in $G$. The time complexity of $d$, denoted $O_t(d)$, is defined as its length $n$, and the space complexity of $d$, denoted $O_s(d)$, is defined as the length of the longest of the strings $\gamma_0, \ldots, \gamma_n$. Let $\gamma_1 \Rightarrow^* \gamma_2$ in $G$, and let $D$ be the set of all rightmost derivations of $\gamma_2$ from $\gamma_1$. The time complexity of rightmost deriving $\gamma_2$ from $\gamma_1$ in $G$, denoted by $O_t(\gamma_1, \gamma_2)$, is defined by $O_t(\gamma_1, \gamma_2) = \min\{O_t(d) | d \in D\}$. The space complexity of rightmost deriving $\gamma_2$ from $\gamma_1$ in $G$, denoted by $O_s(\gamma_1, \gamma_2)$, is defined by $O_s(\gamma_1, \gamma_2) = \min\{O_s(d) | d \in D\}$. We say that $\gamma_1$ rightmost derives $\gamma_2$ in time $t$ if $O_t(\gamma_1, \gamma_2) \leq t$, in space $s$ if $O_s(\gamma_1, \gamma_2) \leq s$.
and simultaneously in time \( t \) and in space \( s \) if for some derivation \( d \in D \), \( O_t(d) \leq t \) and \( O_s(d) \leq s \).

The following CFG properties bound the time and space complexities of CFG derivations.

**Definition 7.2** (Nullability depth).
Let \( G(N, T, P, S) \) be a context-free grammar. We inductively define the set of nullable nonterminals with nullability depth \( k \), denoted \( D_k \), for all \( k > 0 \) by

\[
D_1 = \{ A \mid A \to \epsilon \in P \}
\]

\[
D_k = \left\{ A \mid A \notin \bigcup_{i=1}^{k-1} D_i, A \to A_1 \cdots A_m \in P, A_1, \ldots, A_m \in \bigcup_{i=1}^{k-1} D_i \right\}
\]

We denote the set of nonterminals that are not nullable in \( N \) by \( D_0 \). The nullability depth of a nonterminal \( A \) in \( G \) is defined as \( D_G(A) = k \) such that \( A \in D_k \). The nullability depth of \( G \), denoted by \( D_G \), is defined as \( D_G = \max\{D_G(A) \mid A \in N\} \).

**Definition 7.3** (Nullability width).
Let \( G(N, T, P, S) \) be a CFG. The nullability width of \( G \), denoted by \( W_G \), is defined as the maximal number of nullable nonterminals appearing in some rule in \( P \).

**Definition 7.4** (Loopless path).
Let \( G(N, T, P, S) \) be a CFG. A path in \( G \) is a string of rules \( \pi = r_1r_2 \cdots r_h \in P^* \) such that \( h \geq 0 \), \( A_h \in N \), \( r_i = A_{i-1} \to \alpha_i\beta_i \), and \( \alpha_i\beta_i \) is nullable for all \( i = 1, \ldots, h \). A path \( \pi \) is said to be a loopless if \( A_i \neq A_j \) for all \( 0 \leq i < j \leq h \). We denote the length of the longest loopless path in \( G \) by \( h_G \).

The following lemma states a basic property of rightmost derivations that will be of use to us in subsequent proofs.

**Lemma 7.1.** Let \( G = (N, T, P, S) \) be a context-free grammar, and let \( \alpha_1 \cdots \alpha_m \Rightarrow \beta \) where \( m \geq 1 \). Then there exist rule strings \( \pi_1, \ldots, \pi_m \) and symbol strings \( \beta_1, \ldots, \beta_m \) such that \( \pi_1 \cdots \pi_m = \pi, \alpha_i \pi_i \Rightarrow \beta_i \) for \( i = 1, \ldots, m \) and \( \beta_1 \cdots \beta_m = \beta \).

**Proof.** The proof is by induction on the length of \( \pi \). If \( |\pi| = 0 \) then \( \alpha_1 \cdots \alpha_m = \beta \), and the lemma holds if we choose \( \pi_i = \epsilon \) and \( \beta_i = \alpha_i \) for \( i = 1, \ldots, m \). Assume then that \( |\pi| > 0 \) and, as the induction hypothesis, that the lemma holds for rule strings shorter than \( \pi \). Since \( |\pi| > 0 \), there are strings \( \gamma, \delta \in (N \cup T)^*, w \in T^*, \pi' \in P^* \) and a rule \( A \to \omega \in P \) such that \( \pi = r \pi' \), and

\[
\alpha_1 \cdots \alpha_m = \delta Aw \Rightarrow \delta w = \gamma \pi'
\]

Let \( 1 \leq j \leq m \) such that \( A \) is contained in \( \alpha_j \). We choose \( \gamma_1 \cdots \gamma_m = \gamma \) and \( \pi'_m \cdots \pi'_1 = r \) such that \( \alpha_i = \gamma_i \) and \( \pi'_i = \epsilon \) for every \( i \neq j \), \( \pi'_j = r \), and \( \alpha_j \Rightarrow \gamma_j \). Thus, we get

\[
\alpha_1 \cdots \alpha_m \Rightarrow \gamma_1 \cdots \gamma_m \pi'_1
\]

Since \( |\pi'| < |\pi| \), the induction hypothesis implies that there exist rule strings \( \pi''_1, \ldots, \pi''_m \) and terminal strings \( \beta_1, \ldots, \beta_m \) such that \( \pi''_m \cdots \pi''_1 = \pi', \gamma_i \Rightarrow \beta_i \) for \( i = 1, \ldots, m \) and
\[ \beta_1 \ldots \beta_m = \beta. \] Since \( \gamma_{j+1}, \ldots, \gamma_m \in T^* \), \( \pi''_i = \varepsilon \) for \( i = 1, \ldots, m \), and \( \pi'' \cdots \pi''_1 = \pi' \). By choosing \( \pi_i = \pi'_i \pi''_i \) we get \( \alpha_i \Rightarrow \gamma_i \Rightarrow \beta_i \) for \( i = 1, \ldots, m \), and
\[
\pi_m \cdots \pi_1 = \pi''_m \pi''_{m-1} \cdots \pi''_1 \pi'_1 \\
= \pi''_{j+1} \cdots \pi''_{j+1} \pi''_{j+1} \cdots \pi''_{j+1} \cdots \pi''_1 \\
= \pi'_j \pi'_j \cdots \pi'_j = r \pi' = \pi 
\]

The next lemma states bounds on the time and space complexity of rightmost derivations by loopless rule strings in \( \varepsilon \)-free grammars.

**Lemma 7.2.** Let \( G(N, T, P, S) \) be an \( \varepsilon \)-free CFG. If \( \pi \) is a loopless rule string and \( d \) is a rightmost derivation of \( \beta \neq \varepsilon \) from \( \alpha \) by rule string \( \pi \), then \( O_t(d) \leq (2H_G+2)|\beta|-(H_G+1) \) and \( O_s(d) \leq |\beta| \).

**Proof.** First we note that the space complexity of any derivation of \( \beta \) in \( G \) is \( |\beta| \), because in an \( \varepsilon \)-free grammar no derivation step can decrease the length of the sentential form. We prove the time complexity bound by induction on the length of \( \beta \). Let \( u = H_G \). If \( |\beta| = 1 \) and \( \alpha \Rightarrow \beta \) then the fact that \( G \) is \( \varepsilon \)-free implies that \( |\alpha| = 1 \) and \( \pi \) can contain only unit rules. Since \( \pi \) is loopless and since \( \beta \) may be a terminal, we conclude that
\[
O_t(d) \leq u + 1 = (2u + 2)|\beta| - (u + 1) 
\]

Assume then that \( |\beta| > 1 \) and, as the induction hypothesis, that the theorem holds for derivation of strings shorter than \( \beta \). If \( |\alpha| = |\beta| \), \( \pi \) can contain only unit rules. Since \( \pi \) is loopless, \( 2|\beta| > |\beta| + 1 \) and \( \beta \) may be a terminal string, we conclude that
\[
O_t(d) \leq (u + 1)|\beta| = (u + 1)(|\beta| + 1) - (u + 1) < (2u + 2)|\beta| - (u + 1) 
\]

If \( |\alpha| < |\beta| \), then there exists a rule \( r = A \rightarrow X_1 \cdots X_m \) where \( m \geq 2 \), symbols \( Y_1, \ldots, Y_k \) where \( k = |\alpha| + 1 + m \), and rule strings \( \pi', \pi'' \) such that \( \pi = \pi' \pi'' \) and
\[
\alpha \Rightarrow \delta A \xrightarrow{r} \delta X_1 \cdots X_m x = Y_1Y_2 \cdots Y_k \Rightarrow \beta 
\]
for some strings \( \delta \in (N \cup T)^* \) and \( x \in T^* \) such that \( |\delta A x| = |\alpha| \). By Lemma 7.1, there are rule strings \( \pi_1, \ldots, \pi_k \) and strings \( \beta_1, \ldots, \beta_k \) such that \( Y_i \Rightarrow \beta_i \) for \( i = 1, \ldots, k \), \( \pi'' = \pi_k \cdots \pi_1 \) and \( \beta_1 \cdots \beta_k = \beta \). Furthermore, since \( \pi \) is loopless so are \( \pi_1, \ldots, \pi_k \). \( |\delta A x| = |\alpha| \) and the nonterminal \( A \) imply that \( \pi' \) contains only unit rules of which at most \( |\alpha| - 1 \) have terminals at their right-hand side. Since \( \pi' \) is loopless, we conclude that
\[
|\pi'| \leq |\alpha| \cdot u + |\alpha| - 1 = (u + 1)|\alpha| - 1 
\]

In addition, since \( m \geq 2 \) and \( G \) is \( \varepsilon \)-free, \( 0 < |\beta_i| < |\beta| \) for \( i = 1, \ldots, k \). Thus, by the induction hypothesis, \( \alpha_i \) rightmost derives \( \beta_i \) in time \( (2u + 2)|\beta_i| - (u + 1) \) for \( i = 1, \ldots, k \), and we have
\[
O_t(d) = |\pi| = |\pi' r \pi''| = |\pi'| + 1 + |\pi_k \cdots \pi_1| \\
\leq (u + 1)|\alpha| - 1 + 1 + \sum_{i=1}^{k} |\pi_i| \\
\leq (u + 1)|\alpha| + \sum_{i=1}^{k} (2u + 2)|\beta_i| - (u + 1) \\
= (u + 1)|\alpha| + (2u + 2)|\beta| - k(u + 1) \\
= (2u + 2)|\beta| + (1 - m)(u + 1) \\
\leq (2u + 2)|\beta| - (u + 1) 
\]
The following theorem states tight bounds on the derivation of nonempty terminal strings in \( \varepsilon \)-free grammars. The bounds are derived directly from Lemma 7.2. They are shown to be tight by proving that the sequence of \( \varepsilon \)-free grammars \( G_u \) \((u \geq 0)\) listed in Figure 7.1 attain them.

**Theorem 7.3.** Let \( G(N, T, P, S) \) be an \( \varepsilon \)-free CFG. If \( X \in (N \cup T)^* \) and \( w \in L(X) \), then \( X \) rightmost derives \( w \) simultaneously in time \((2H_G + 2)|w| - (H_G + 1)\) and space \(|w|\). Moreover, these bounds are tight.

**Proof.** Lemma 7.2 proves the validity of the bounds. We prove that the bounds are tight by showing that a grammar exists which attains them. Let \( G = G_u \) \((u \geq 0)\) be the grammar listed in Figure 7.1 where \( u = H_G \). Clearly, \( L(G) = L(A_0) = \{a^k | k \geq 1\} \). We show by induction on \( k \) that the rightmost derivation of \( a^k \) from \( A_0 \) attains the bounds of the theorem. If \( k = 1 \), the shortest rightmost derivation of \( a^1 \) from \( A_0 \) is obtained from the rewrite

\[
A_0 \Rightarrow A_1 \Rightarrow \cdots \Rightarrow A_u \Rightarrow a
\]

The time complexity of this derivation is

\[
u + 1 = (2u + 2)|w| - (u + 1)
\]

and the space complexity is \( 1 = |w| \). Assume then, that \( k > 1 \) and, as the induction hypothesis, that the derivation of terminal strings shorter than \( k \) attain the specified bounds. Since \( |a^k| > 1 \), \( A_0 \) must derive the sentential form \( A_0 A_0 \) before deriving \( a^k \), as follows

\[
A_0 \Rightarrow A_1 \Rightarrow \cdots \Rightarrow A_u \Rightarrow A_0 A_0 \Rightarrow *A_0 a^j \Rightarrow *a^i a^j = a^k
\]

where \( 1 \leq i, j < k \) and \( i + j = k \). Thus, according to the induction hypothesis, the time complexity of deriving \( a^k \) from \( A_0 \) is

\[
t = u + 1 + (2u + 2)j - (u + 1) + (2u + 2)i - (u + 1)
\]

\[
t = (2u + 2)(j + i) - (u + 1) = (2u + 2)k - (u + 1)
\]

and the longest sentential form of the derivation, obtained while rightmost deriving \( a^k \) from \( A_0 a^j \), is of length

\[
s = 1 + j - 1 + i = j + i = k = |w|
\]
\[ G_{n,p} : A_p \rightarrow A_{p-1}^n \]
\[ A_{p-1} \rightarrow A_{p-2}^n \]
\[ \vdots \]
\[ A_2 \rightarrow A_1^n \]
\[ A_1 \rightarrow \varepsilon \]

Figure 7.2: A CFG with worst-case rightmost derivation complexity of \( \varepsilon \)

The next theorem states tight bounds on the time and space complexity of deriving the empty string in grammars consisting of \( \varepsilon \)-rules. The bounds are shown to be tight by proving that the sequence of context-free grammars \( G_{n,p} (n \geq 0, p \geq 1) \) listed in Figure 7.2 attain them.

**Theorem 7.4.** Let \( G(N, T, P, S) \) be a CFG. For every nullable nonterminal \( A \in N \), if \( W_G \leq 1 \) then \( A \) rightmost derives \( \varepsilon \) simultaneously in time \( D_G(A) \) and space 1; otherwise, \( A \) rightmost derives \( \varepsilon \) simultaneously in time \( (W_G^{D_G(A)} - 1)/(W_G - 1) \) and space \( (D_G(A) - 1)(W_G - 1) + 1 \). Moreover, these bounds are tight.

**Proof.** We begin by proving the validity of the bounds. Let \( n = W_G \). If \( n = 0 \) and \( A \) is nullable then \( A \rightarrow \varepsilon \in P \). Thus, \( A \) rightmost derives \( \varepsilon \) simultaneously in time \( 1 = D_G(A) \) and space \( 1 = |A| \). For \( n \geq 1 \) the proof is by induction on \( D_G(A) \). If \( D_G(A) = 1 \) then \( A \rightarrow \varepsilon \in P \). Thus, \( A \) rightmost derives \( \varepsilon \) simultaneously in time

\[
1 = \begin{cases} 
D_G(A) & n = 1 \\
(n^{D_G(A)} - 1)/(n - 1) & n \geq 2 
\end{cases}
\]

and space

\[
1 = \begin{cases} 
|A| & n = 1 \\
(D_G(A) - 1)(n - 1) + 1 & n \geq 2 
\end{cases}
\]

Assume then that \( D_G(A) > 1 \) and, as the induction hypothesis, that for any nullable nonterminal \( B \) such that \( D_G(B) < D_G(A) \) the bounds on the rightmost derivation of \( \varepsilon \) hold. If \( D_G(A) > 1 \) then by Definition 7.2, \( G \) has a rule \( r = A \rightarrow A_1 \cdots A_k \) such that \( A_i \) is nullable and \( D_G(A_i) < D_G(A) \) for \( i = 1, \ldots, k \). \( n = 1 \) implies that \( k = 1 \), in which case the induction hypothesis implies that \( A_1 \) rightmost derives \( \varepsilon \) simultaneously in time \( D_G(A_1) \) and space 1. Thus, \( A \) rightmost derives \( \varepsilon \) simultaneously in time \( D_G(A) = 1 + D_G(A_1) \) and space \( 1 = \max(|A|, 1) \). For \( n > 1 \) the induction hypothesis implies that \( A_i \) rightmost derives \( \varepsilon \) simultaneously in time \( (n^{D_G(A_i)} - 1)/(n - 1) \) and space \( (D_G(A_i) - 1)(n - 1) + 1 \). Thus, \( A \) rightmost derives \( \varepsilon \) simultaneously in time

\[
t = 1 + \sum_{i=1}^{k} (n^{D_G(A_i)} - 1)/(n - 1) \\
\leq 1 + k(n^{D_G(A) - 1} - 1)/(n - 1) \\
\leq 1 + n(n^{D_G(A) - 1} - 1)/(n - 1) \\
= (n^{D_G(A)} - 1)/(n - 1)
\]
and space
\[ s = \max\{k, \max\{(D_G(A_i) - 1)(n - 1) + 1|i = 1, \ldots, k\}\} \]
\[ \leq \max\{k, (D_G(A) - 1)(n - 1) + 1\} \]
\[ \leq \max\{n, (D_G(A) - 1)(n - 1) + 1\} \]
\[ = (D_G(A) - 1)(n - 1) + 1 \]

We now prove that the bounds are tight by showing that the grammar \( G = G_{n,p} \) \((n \geq 0, p \geq 1)\) where \( p = D_G \) listed in Figure 7.2 attains them. The proof is by induction on \( i = 1, \ldots, p \). Clearly, for \( i = 1 \), the time and space complexity of rightmost deriving \( \varepsilon \) from \( A_1 \) is 1 regardless of the value of \( n \). Assume then, that \( i > 1 \) and, as the induction hypothesis, that the derivations of \( \varepsilon \) from \( A_1, \ldots, A_{i-1} \) attain the specified bounds. If \( n = 0 \), then \( A_i \) directly derives \( \varepsilon \) in time and space 1; otherwise, \( A_i \Rightarrow^* \varepsilon \) implies that \( A_i \Rightarrow A_{i-1}^n \Rightarrow^* \varepsilon \). According to the induction hypothesis, and since \( D_G(A_i) = D_G(A_{i-1}) + 1 \), if \( n = 1 \), the time complexity of rightmost deriving \( \varepsilon \) from \( A_i \) is
\[ 1 + D_G(A_{i-1}) = D_G(A_i) \]
and the space complexity is 1. Otherwise, the time complexity of rightmost deriving \( \varepsilon \) from \( A_i \) is
\[ t = 1 + n[(n^{D_G(A_{i-1}) - 1})/(n - 1)] \]
\[ = (n - 1 + n^{D_G(A_{i-1})+1} - n)/(n - 1) \]
\[ = (n^{D_G(A_i) - 1})/(n - 1) \]
and since \( 1 < i = D_G(A_i) \), the space complexity is
\[ s = \max\{n, n - 1 + (D_G(A_{i-1}) - 1)(n - 1) + 1\} \]
\[ = \max\{n, D_G(A_{i-1})(n - 1) + 1\} \]
\[ = \max\{n, (D_G(A_i) - 1)(n - 1) + 1\} \]
\[ = (D_G(A_i) - 1)(n - 1) + 1 \]

The following lemma states bounds on the time and space complexity of rightmost derivations of nonempty strings in free-form grammars. Given a rightmost derivation of a nonempty sentential form by rule string \( \pi \) in a free-form context-free grammar \( G \), the bounds are specified with respect to the length of a corresponding derivation in an \( \varepsilon \)-free grammar \( \hat{G} \) constructed from \( G \).

**Lemma 7.5.** Let \( G(N,T,P,S) \) be a CFG and let \( \hat{G}(N,T,\hat{P},S) \) be the \( \varepsilon \)-free CFG constructed from \( G \) by replacing its set of production rules \( P \) with the set
\[ \hat{P} = \{A \Rightarrow \alpha_1 \cdots \alpha_{l+1} | l \geq 0, \alpha_1 \cdots \alpha_{l+1} \notin \varepsilon, A \Rightarrow \alpha_1 B_1 \alpha_2 \cdots \alpha_l B_l \alpha_{l+1} \in P, B_1 \cdots B_l \Rightarrow \varepsilon_G \} \]
If \( \alpha \) and \( \beta \) are nonempty symbol strings, and \( d \) is a derivation of \( \beta \) from \( \alpha \) by the rule string \( \pi \) in \( G \) where \( l \geq 0 \) is the maximal integer such that
\[ \alpha = \alpha_1 B_1 \alpha_2 B_2 \cdots \alpha_l B_l \alpha_{l+1}, \]
\[ \pi = \pi_{l+1} \pi_{B_1} \pi_{B_{l-1}} \cdots \pi_2 \pi_{B_1} \pi_1 \text{ where } B_i \Rightarrow \varepsilon_G \text{ in } G \text{ for all } i = 1, \ldots, l, \]
\[ \alpha_1 \Rightarrow \beta_1 \text{ in } G \text{ for all } i = 1, \ldots, l + 1 \text{ where } \beta_1 \cdots \beta_{l+1} = \beta, \]
and any rewrite of a nullable nonterminal to \( \varepsilon \) in \( d \) occurs simultaneously in time \( t \) > 0 and space \( s > 0 \),
then there exists a rule string $\hat{\pi} \in \hat{P}^*$ such that

$$
\alpha_1 \alpha_2 \cdots \alpha_{l+1} \xrightarrow{\hat{\pi}} \beta \text{ in } \hat{G},
$$

where $0 \leq k \leq |\hat{\pi}_u|$ is the maximal integer such that $k : \hat{\pi} \in (N \times N)^*$, and

$$
|\pi_1| + |\pi_2| + \cdots + |\pi_{l+1}| \leq (1 + W_G t) |\hat{\pi}|,
$$

$$(b) \quad |\pi_B_1| + |\pi_B_2| + \cdots + |\pi_B_l| \leq lt, \text{ and}
$$

$$
O_s(d) = |\beta| + l + s - 1 + W_G(|\hat{\pi}_u| + 1)(|\beta| - |\alpha_1 \alpha_2 \cdots \alpha_{l+1}|) + W_G(k + 1)
$$

where $\hat{\pi}_u$ is the longest substring of $\hat{\pi}$ such that $\hat{\pi}_u \in (N \times N)^*$, and

as stated in (b). Assume then that $|\pi| > 0$ and, as the induction hypothesis, that the specified derivation exists in $G$ for rule strings shorter than $\pi$. Since $\pi$ is a rightmost derivation, $|\pi| > 0$ and $B_l$ is a nonterminal (if $l > 0$), either (1) $\pi_{l+1} \neq \varepsilon$ and $\pi_{l+1} = r \pi_{l+1}'$ or (2) $l > 0$, $\alpha_{l+1} \in T^*$, $\pi = \pi_B \pi'$, and $\alpha_{l+1} \neq \varepsilon$. In case (1) we can write

$$
\alpha = \gamma \alpha_{l+1} = \gamma \alpha_{l+1}' \beta_1 \beta_2 \cdots \beta_{l+1}
$$

where $0 \leq m \leq n$ is the maximal integer such that

$$
\gamma = \alpha_1 B_1 \alpha_2 B_2 \cdots \alpha_l B_l, \alpha_0' \in (N \cup T)^*, \alpha_{m+2}' \in T^*,
$$

$$
r = A \Rightarrow B_1' \alpha_2' B_2' \cdots \alpha_m' B_m' \alpha_{m+1}' \alpha_{m+2}' \in G,
$$

$$
\pi''_{l+1} = \pi_{m+2}' \pi_{m+1}' \pi_{m}' \pi''_{m+1} \pi''_{m} \cdots \pi''_1 \pi''_0
$$

where $B_1' \xrightarrow{\pi''} \varepsilon \in G$ for all $i = 1, \ldots, m$, and

$$
\alpha_i' \xrightarrow{\pi''} \beta_i' \text{ in } G \text{ for all } i = 0, \ldots, m + 2 \text{ where } \beta_0' \beta_1' \cdots \beta_{m+1}' \beta_{m+2}' = \beta_{l+1}
$$

We therefore conclude that $d' = d : |\pi'|$ is a rightmost derivation of $\beta$ from $\alpha'$ by rule string $\pi'$ in $G$ where

$$
\alpha_1 B_1 \alpha_2 B_2 \cdots \alpha_l B_l \alpha_0' \alpha_1' B_1' \alpha_2' B_2' \cdots \alpha_m' B_m' \alpha_{m+1}' \alpha_{m+2}'
$$

$$
\pi' = \pi_{m+2}' \pi_{m+1}' \pi_{m}' \cdots \pi'_2 \pi'_1 \pi_0' \pi_{m+2}' \pi_{m+1}' \pi_m'
$$

where $B_i \xrightarrow{\pi''} \varepsilon \text{ in } G$ for all $i = 1, \ldots, l$ and $B_i \xrightarrow{\pi''} \varepsilon \text{ in } G$ for all $i = 1, \ldots, m$,

$$
\alpha_i \xrightarrow{\pi''} \beta_i \text{ in } G \text{ for all } i = 1, \ldots, l \text{ and } \alpha_i' \xrightarrow{\pi''} \beta_i' \text{ in } G \text{ for all } i = 0, \ldots, m + 2
$$

where $\beta_0' \beta_1' \cdots \beta_{m+1}' \beta_{m+2}'$ and any rewrite of a nullable nonterminal to $\varepsilon$ in $d'$ occurs simultaneously in time $t > 0$ and space $s > 0$.

Since $|\pi'| < |\pi|$, we can apply the induction hypothesis to $d'$, and conclude that there exists a rule string $\hat{\pi}' \in \hat{P}^*$ such that

$$
\alpha_1 \alpha_0' \alpha_1' \alpha_2' \cdots \alpha_{m+2}' \xrightarrow{\hat{\pi}'} \beta \text{ in } \hat{G},
$$

$$
|\pi_1| + \cdots + |\pi_i| + |\pi_0'| + \cdots + |\pi_{m+2}'| \leq (1 + nt) |\hat{\pi}'|,
$$

$$
|\pi_B_1| + \cdots + |\pi_B_i| + |\pi_B'_i| + \cdots + |\pi_B_m'| \leq (l + m)t, \text{ and}
$$

$$
O_s(d') \leq |\beta| + l + m + s - 1 + n(|\hat{\pi}_u'| + 1)(|\beta| - |\alpha_1 \cdots \alpha_0' \cdots \alpha_{m+2}'|) + n(k' + 1)
$$

where $\hat{\pi}'_u$ is the longest substring of $\hat{\pi}'$ such that $\hat{\pi}'_u \in (N \times N)^*$, and

$0 \leq k' \leq |\hat{\pi}'_u|$ is the maximal integer such that $k' : \hat{\pi}' \in (N \times N)^*$.
Further, since $\alpha'_1 \alpha'_2 \cdots \alpha'_{m+1} = \varepsilon$ contradicts the maximality of $l$, we have $\alpha'_1 \alpha'_2 \cdots \alpha'_{m+1} \neq \varepsilon$. Thus, by construction of $G$, there exists a rule $\hat{r} = A \to \alpha'_1 \alpha'_2 \cdots \alpha'_{m+1} \in P$ such that

$$\alpha_1 \alpha_2 \cdots \alpha_l \alpha_{l+1} = \alpha_1 \alpha_2 \cdots \alpha_l \alpha_0' \pi_{m+2}^\beta \cdots \alpha_l \alpha_0' \alpha'_1 \cdots \alpha'_{m+2}^\beta \pi_G^k$$

Since the rule string $\pi_{B_i}'$ for all $i = 1, \ldots, m$ is part of the derivation of $\beta_{l+1}$ from $\alpha_{l+1}$ we have

$$|\pi_1| + \cdots + |\pi_l| + |\pi_{l+1}| \leq (1 + nt)|\hat{\pi}| + |r| + |\pi_{B_1}| + \cdots + |\pi_{B_m}|$$

$$\leq (1 + nt)|\hat{\pi}| + |r| + nt \leq (1 + nt)|\hat{\pi}| + (1 + nt)$$

$$= (1 + nt)|\hat{\pi}|,$$

and

$$|\pi_{B_1}| + \cdots + |\pi_{B_l}| \leq (l + m)t - (|\pi_{B_1}| + \cdots + |\pi_{B_m}|) \leq lt$$

Furthermore, since $\alpha'_1 \alpha'_2 \cdots \alpha'_{m+1} \neq \varepsilon$ implies that $|\alpha| \leq |\alpha'|$, we have $O_s(d) \leq O_s(d')$. Let $\hat{\pi}_u$ be the longest substring of $\hat{\pi}'$ such that $\hat{\pi}_u \in (N \times N)^*$, and let $k$ be the maximal integer such that $k : \hat{\pi}_u' \in (N \times N)^*$. If $|\alpha'_1 \alpha'_2 \cdots \alpha'_{m+1}| = 1$, $|\alpha_1 \cdots \alpha_l \alpha_0' \cdots \alpha'_{m+2}| = |\alpha_1 \cdots \alpha_{l+1}|$, $k = k' + 1$, $|\hat{\pi}_u'| \leq |\hat{\pi}_u|$, and we have

$$O_s(d) \leq |\beta| + l + m + s - 1 + n(|\hat{\pi}_u' + 1)|(|\beta| - |\alpha_1 \cdots \alpha_l \alpha_0' \cdots \alpha'_{m+2}|) + n(k' + 1)$$

$$= |\beta| + l + m + s - 1 + n(|\hat{\pi}_u' + 1)|(|\beta| - |\alpha_1 \cdots \alpha_{l+1}|) + n(k' + 1)$$

$$\leq |\beta| + l + m + s - 1 + n(|\hat{\pi}_u + 1)|(|\beta| - |\alpha_1 \cdots \alpha_{l+1}|) + n(k' + 1)$$

$$= |\beta| + l + s - 1 + n(|\hat{\pi}_u + 1)|(|\beta| - |\alpha_1 \cdots \alpha_{l+1}|) + n(k + 1)$$

$$\leq |\beta| + l + s - 1 + n(|\hat{\pi}_u + 1)|(|\beta| - |\alpha_1 \cdots \alpha_{l+1}|) + n(k + 1)$$

Otherwise, $|\alpha'_1 \alpha'_2 \cdots \alpha'_{m+1}| > 1$, $|\alpha_1 \cdots \alpha_l \alpha_0' \cdots \alpha'_{m+2}| \geq |\alpha_1 \cdots \alpha_{l+1}| + 1$, $k = 0$, $|\hat{\pi}_u'| = |\hat{\pi}_u|$, and we have

$$O_s(d) \leq |\beta| + l + m + s - 1 + n(|\hat{\pi}_u' + 1)|(|\beta| - |\alpha_1 \cdots \alpha_l \alpha_0' \cdots \alpha'_{m+2}|) + n(k' + 1)$$

$$\leq |\beta| + l + m + s - 1 + n(|\hat{\pi}_u' + 1)|(|\beta| - |\alpha_1 \cdots \alpha_l \alpha_0' \cdots \alpha'_{m+2}|) + n(|\hat{\pi}_u' + 1)$$

$$= |\beta| + l + m + s - 1 + n(|\hat{\pi}_u + 1)|(|\beta| - |\alpha_1 \cdots \alpha_{l+1}|) + n(|\hat{\pi}_u + 1)$$

$$\leq |\beta| + l + s - 1 + n(|\hat{\pi}_u + 1)|(|\beta| - |\alpha_1 \cdots \alpha_{l+1}|) + n(k + 1)$$

Thus, by choosing $\hat{\pi} = \hat{\pi}'$, we satisfy statement (b). In case (2), $l > 0$, $\alpha_{l+1} \in T^*$, $\pi = \pi_{B_1} \pi'$, $\pi_{B_1} \neq \varepsilon$, and we can write

$$\alpha = \alpha_1 B_1 \alpha_2 B_2 \cdots B_{l-1} \alpha_l B_l \alpha_{l+1} \pi_{B_l}^G \alpha_1 B_1 \alpha_2 B_2 \cdots B_{l-1} \alpha_l \alpha_{l+1} = \alpha' \pi_G^l \beta_1 \cdots \beta_{l+1}$$

Note that the maximality of $l$ implies that $|\alpha_i| \leq |\beta_i|$ for all $i = 1, \ldots, l + 1$. Since $|\pi'| < |\pi|$ and since $\alpha'$ clearly satisfies the conditions stated in (a) we can apply the induction hypothesis to the derivation $d'$ of $\beta$ from $\alpha'$ by $\pi'$ in $G$, and conclude that there
exists a rule string $\pi' \in \hat{P}^*$ such that
\[\alpha_1\alpha_2 \ldots \alpha_{t+1} \Rightarrow \beta \text{ in } \hat{G},\]
\[|\pi| + |\pi_2| + \cdots + |\pi_{t+1}| \leq (1 + nt)|\pi'|,\]
\[|\pi_{B_1}| + |\pi_{B_2}| + \cdots + |\pi_{B_{t-1}}| \leq (t - 1)t, \text{ and}\]
\[O_s(d) = |\beta| + l - 1 + s - 1 + n(|\pi_u| + 1)(|\beta| - |\alpha_1\alpha_2 \cdots \alpha_{t+1}|) + n(k + 1)\]
where $\pi_u$ is the longest substring of $\pi'$ such that $\pi_u \in (N \times N)^*$, and $0 \leq k \leq |\pi_u|$ is the maximal integer such that $k : \pi' \in (N \times N)^*$.

On the other hand, the time complexity of the derivation $d''$ of $\alpha'$ from $\alpha$ by rule string $\pi_{B_t}$ in $G$ is $t$, and its space complexity is
\[O_s(d'') = |\alpha| - 1 + s = |\alpha_1| + |\alpha_2| + \cdots + |\alpha_{t+1}| + l + s - 1\]
\[\leq |\beta_1| + |\beta_2| + \cdots + |\beta_{t+1}| + l + s - 1 = |\beta| + l + s - 1,\]
and by choosing $\hat{\pi} = \hat{\pi}'$, we have
\[|\pi_1| + |\pi_2| + \cdots + |\pi_{t+1}| \leq (1 + nt)|\hat{\pi}|, |\pi_{B_1}| + |\pi_{B_2}| + \cdots + |\pi_{B_{t-1}}| \leq lt, \text{ and}\]
\[O_s(d) = \max\{d', d''\} \leq |\beta| + l + s - 1 + n(|\pi_u| + 1)(|\beta| - |\alpha_1\alpha_2 \cdots \alpha_{t+1}|) + n(k + 1)\]
which satisfies statement (b).

Our next goal is to reformulate the bounds of Lemma 7.5 in terms of the attributes of the grammar $G$ rather than those of derivations in $\hat{G}$. Since $\hat{G}$ is an $\varepsilon$-free grammar, we can do so by applying the bounds of Lemma 7.2 to the derivation by rule string $\hat{\pi}$, providing that $\hat{\pi}$ is loopless. The following lemma implies that choosing the shortest derivation in $G$ guarantees this property of $\hat{\pi}$, and Lemma 7.7 allows us to bound the length of loopless nonterminal unit derivations $(A_0, A_1, \ldots)$ in $G$ by the length of certain rule strings in $G$.

**Lemma 7.6.** Let $G(N, T, P, S)$ be a CFG and $\hat{G}$ the $\varepsilon$-free grammar of Lemma 7.5 constructed from $G$. Furthermore, let $\pi$ be a rule string such that $A \xrightarrow{\hat{\pi}} \beta$ in $G$, and $\hat{\pi}$ the rule string constructed from $\pi$ as in the proof of Lemma 7.5 such that $A \xrightarrow{\pi} \beta$ in $\hat{G}$. If $A \xrightarrow{\hat{\pi}} A$ where $\hat{\pi}_1 \neq \varepsilon$ is a nonempty prefix of $\hat{\pi}$ (i.e., $\hat{\pi}_1$ is a rightmost loop), then there exists a rule string shorter than $\hat{\pi}$ by which $A$ derives $\beta$ in $G$.

**Proof.** We first prove by induction on $0 \leq k \leq |\hat{\pi}|$ that
\[(a) \quad A \xrightarrow{\hat{\pi}_1} A' \xrightarrow{\hat{\pi}_2} \beta \text{ in } \hat{G} \text{ where } \hat{\pi} = \hat{\pi}_1 \hat{\pi}_2 \text{ and } \hat{\pi}_1 = k : \hat{\pi}\]
implies that
\[(b) \quad A \xrightarrow{\pi_1} A' \xrightarrow{\pi_2} \beta \text{ in } G \text{ where } \pi = \pi_1 \pi_2, |\pi_1| \geq |\hat{\pi}_1|, \text{ and } \pi'_2 \text{ is a prefix of } \pi_2\]
If $k = 0$ we have $\hat{\pi}_1 = \varepsilon$, $\hat{\pi}_2 = \hat{\pi}$ and $A \xrightarrow{\hat{\pi}} A \xrightarrow{\hat{\pi}} \beta$ in $\hat{G}$. By choosing $\alpha = \varepsilon$, $\pi_1 = \varepsilon$ and $\pi'_2 = \pi_2 = \pi$ we have $A \xrightarrow{\pi} A \xrightarrow{\pi} \beta$ in $G$, and $|\pi_1| \geq |\hat{\pi}_1|$. Assume then, that $k > 0$ and, as the induction hypothesis, that statement (b) holds for prefixes of $\hat{\pi}$ shorter than $\hat{\pi}_1$. Since $k > 0$ we can write $A \xrightarrow{\hat{\pi}_1} A' \xrightarrow{\hat{\pi}_2} A' \xrightarrow{\hat{\pi}} \beta$ in $\hat{G}$ where $\hat{\pi}_1 = \hat{\pi}_2 \hat{\pi}$ and $\hat{\pi} = A' \rightarrow A'$. Since $|\hat{\pi}_1| < |\hat{\pi}_1|$, we can apply the induction hypothesis, and conclude that there are rule strings $\pi'_1, \pi'_2, \pi'_2$, and a nonterminal string $\alpha_0$ such that $A \xrightarrow{\pi'_1} \alpha_0 A' \xrightarrow{\pi'_2} \beta$ and $A' \xrightarrow{\pi'_2} \beta$ in $G$.
where \( \pi = \pi'_1\pi'_2 \), \( |\pi'_1| \geq |\hat{\pi}'_1| \) and \( \pi''_2 \) is a prefix of \( \pi'_2 \). Furthermore, by construction of \( \hat{\pi} \), we have \( r = A' \to \alpha_1A'\alpha_2 \in P \) where \( \alpha_1, \alpha_2 \in N^* \), and \( \pi''_2 = r\pi_{\alpha_3}A'\pi_{\alpha_3}A_{\alpha_0} \) where \( \alpha_i \not\Rightarrow^* \varepsilon \) in \( G \) for \( i = 0, 1, 2 \) and \( A'\pi_{\alpha_3}A_{\alpha_0} \) in \( G \). By choosing \( \alpha = \alpha_0\alpha_1, \pi_1 = \pi'_1r\pi_{\alpha_3}, \pi_2 = A'\pi_{\alpha_3}A_{\alpha_0} \) and \( \pi''_2 = \pi_{\alpha_3}A_{\alpha_0} \) we have \( A'\pi_{\alpha_3}A_{\alpha_0} \) and \( A'\pi_{\alpha_3}A_{\alpha_0} \) in \( G \) where \( \pi = \pi_1\pi_2, |\pi_1| \geq |\hat{\pi}'_1| \), and \( \pi''_2 \) is a prefix of \( \pi_2 \). Thus we conclude that exists a rule string shorter than \( \pi \) by which \( A \) derives \( \beta \) in \( G \).

\( \square \)

**Lemma 7.7.** Let \( G(N,T,P,S) \) be a CFG and \( \hat{G} \) the \( \varepsilon \)-free grammar of Lemma 7.5 constructed from \( G \). Furthermore, let \( d_u = (A_0, A_1, \ldots, A_u) \) the longest rightmost derivation in \( \hat{G} \) such that \( u \geq 0, A_i \neq A_j \) for all \( 0 \leq i < j \leq u \), and \( A_i \in N \) for \( i = 0, \ldots, u \). Then there exists a rule string \( \pi = r_1r_2 \cdots r_u \) in \( G \) such that \( r_i = A_{i-1} \to \alpha_iA_i\beta_i \), and \( \alpha_i\beta_i \not\Rightarrow^* \varepsilon \) in \( G \) for \( i = 1, \ldots, u \).

Proof. The proof is by induction on \( u \). The case \( u = 0 \) trivially holds. Assume then that \( u > 0 \) and, as the induction hypothesis, that the lemma holds for derivations shorter than \( \hat{d} \). If \( \hat{d} = (A_0, A_1, \ldots, A_u) \) is a derivation in \( \hat{G} \) that meets the terms of the lemma, so does the derivation \( d = (A_0, A_1, \ldots, A_{u-1}) \), and \( G \) has a rule \( \hat{r} = A_{u-1} \to A_u \) such that \( A_u \neq A_0, \ldots, A_{u-1} \). The induction hypothesis implies that there exists a rule string \( \pi' = r_1r_2 \cdots r_{u-1} \) in \( G \), such that \( r_i = A_{i-1} \to \alpha_iA_i\beta_i \), and \( \alpha_i\beta_i \not\Rightarrow^* \varepsilon \) in \( G \) for \( i = 1, \ldots, u-1 \). By construction of \( \hat{G} \), \( \hat{r} \in \hat{P} \) implies that \( G \) has a rule \( r = A_{u-1} \to A_uA_u\beta_u \) where \( A_u\beta_u \not\Rightarrow^* \varepsilon \) in \( G \). Thus, the required rule string \( \pi = \pi'\hat{r} \) exists in \( G \).

The next lemma combines the results of Theorem 7.4 and Lemmas 7.2, 7.6, and 7.7 to formulate the bounds stated in Lemma 7.5 in terms of the properties of the CFG \( G \).

**Lemma 7.8.** Let \( G(N,T,P,S) \) be a CFG, \( \alpha \) and \( \beta \) nonempty symbol strings, \( d \) a derivation of \( \beta \) from \( \alpha \) by the rule string \( \pi \) in \( G \) where \( \pi \) is the shortest rule string by which \( \beta \) can be derived from \( \alpha \), and \( l \geq 0 \) is the maximal integer such that \( \alpha = \alpha_1B_1\alpha_2B_2 \cdots \alpha_lB_l\alpha_{l+1} \), \( \pi = \pi_{i=1,2} \cdots \pi_{i=l+1} \), and \( B_i \not\Rightarrow^* \varepsilon \) in \( G \) for all \( i = 1, \ldots, l \). Then \( W_G \leq 1 \) implies that

\[
\begin{align*}
O_t(d) &\leq (1 + W_GD_G)(2H_G + 2)|\beta| - (H_G + 1) + lD_G \\
O_s(d) &\leq |\beta| + l + W_G(H_G + 1)(|\beta| - |\alpha_1 \cdots \alpha_{l+1}| + 1)
\end{align*}
\]

and if \( W_G > 1 \)

\[
\begin{align*}
O_t(d) &\leq (1 + (W_GD_G + 1 - W_G)/(W_G - 1))(2H_G + 2)|\beta| - (H_G + 1) + l(W_GD_G - 1)/(W_G - 1) \\
O_s(d) &\leq |\beta| + l + W_G(H_G + 1)(|\beta| - |\alpha_1 \cdots \alpha_{l+1}| + 1) + (D_G - 1)(W_G - 1) - 1
\end{align*}
\]

Proof. Let \( u = H_G \), and \( n = W_G \). Lemma 7.5 implies that there exists a rule string \( \hat{\pi} \in \hat{P}^* \) such that \( \alpha_1\alpha_2 \cdots \alpha_{l+1} \not\Rightarrow^* \varepsilon \) in \( \hat{G} \), \( O_s(d) \leq |\beta| + l + n(|\hat{\pi}_0| + 1)(|\beta| - n|\alpha_1 \cdots \alpha_{l+1}| + 1) + n(k + 1) + s - 1 \), and \( O_t(d) \leq (1 + nt)|\hat{\pi}| + lt \), where any rewrite of a nullable nonterminal to \( \varepsilon \) in \( d \) occurs simultaneously in time \( t > 0 \) and space \( s > 0 \), \( \hat{\pi}_0 \) is the longest substring of \( \hat{\pi} \) such that \( \hat{\pi}_0 \in (N \times N)^* \), and \( 0 \leq k \leq |\hat{\pi}_0| \) is the maximal integer such that \( k : \hat{\pi} \in (N \times N)^* \). Since \( d \) is the shortest derivation of \( \beta \) from \( \alpha \), Lemma 7.6 implies that \( \hat{\pi} \) does not contain a rightmost loop. Since \( \hat{\pi}_0 \) is substring of \( \hat{\pi} \), the same applies to \( \hat{\pi}_0 \).
If \((A_0, A_1, \ldots, A_u)\) is the longest rightmost derivation in \(\hat{G}\) such that \(u' \geq 0\), \(A_i \neq A_j\) for all \(0 \leq i < j \leq u'\), and \(A_i \in N\) for \(i = 0, \ldots, u'\), we conclude that \(|\pi| \leq u'\), and by Lemma 7.2, that \(|\pi| \leq (2u' + 2)|\beta| - (u' + 1)\). Furthermore, Lemma 7.7 implies that \(u' \leq u\) and thus \(k \leq |\pi| \leq u\), and \(|\pi| \leq (2u + 2)|\beta| - (u + 1)\). Combining these results we have

\[
O_t(d) \leq (1 + nt)[(2u + 2)|\beta| - (u + 1)] + lt
\]

\[
O_s(d) \leq |\beta| + l + n(u + 1)(|\beta| - |\alpha_1 \cdots \alpha_{l+1}| + 1) + s - 1
\]

Lastly, Theorem 7.4 implies that every nullable nonterminal in \(G\) rightmost derives \(\varepsilon\) simultaneously in time

\[
t = \begin{cases} 
  D_G & n \leq 1 \\
  (nD_G - 1)/(n - 1) & n > 1
\end{cases}
\]

and space

\[
s = \begin{cases} 
  1 & n \leq 1 \\
  (D_G - 1)(n - 1) + 1 & n > 1
\end{cases}
\]

Thus, we conclude that \(n \leq 1\) implies that

\[
O_t(d) \leq (1 + nD_G)[(2u + 2)|\beta| - (u + 1)] + lD_G
\]

\[
O_s(d) \leq |\beta| + l + n(u + 1)(|\beta| - |\alpha_1 \cdots \alpha_{l+1}| + 1)
\]

and if \(n > 1\)

\[
O_t(d) \leq (1 + (nD_G - 1 - n)/n - 1))[(2u + 2)|\beta| - (u + 1)] + ln(nD_G - 1)/n - 1
\]

\[
O_s(d) \leq |\beta| + l + n(u + 1)(|\beta| - |\alpha_1 \cdots \alpha_{l+1}| + 1) + (D_G - 1)(n - 1) - 1
\]

as stated by the lemma.

The following theorem states tight bounds on the time and space complexity of deriving nonempty terminal strings in free-form context-free grammars. The bounds are shown to be tight by proving that the sequence of context-free grammars \(G_{u,n,p}\) \((u \geq 0, n \geq 0, p \geq 1)\) listed in Figure 7.3 attain them.

**Theorem 7.9.** Let \(G(N, T, P, S)\) be a CFG. If \(A\) is a nonterminal, and \(w \in L(A) \setminus \{\varepsilon\}\), then \(A\) rightmost derives \(w\) simultaneously in time

\[
t = \begin{cases} 
  (1 + W_G D_G)[(2H_G + 2)|w| - (H_G + 1)] & W_G \leq 1 \\
  [1 + (W_G D_G + 1 - W_G)/(W_G - 1)][(2H_G + 2)|w| - (H_G + 1)] & W_G > 1
\end{cases}
\]

and space

\[
s = \begin{cases} 
  (1 + W_G(H_G + 1))|w| & W_G \leq 1 \\
  (1 + W_G(H_G + 1))|w| + (D_G - 1)(W_G - 1) & W_G > 1
\end{cases}
\]

Moreover, these bounds are minimal.
Figure 7.3: A CFG with worst-case rightmost derivation complexity of $w \in T^+$

Proof. Lemma 7.8 proves the validity of the bounds if we choose $\beta = w$, $\alpha_1 \ldots \alpha_{l+1} = A$ and $l = 0$. We prove that the bounds are tight by showing that a grammar exists which attains them. Let $G = G_{u,n,p}$ ($u \geq 0, n \geq 0, p \geq 1$) where $u = H_G$, $n = W_G$, and $p = D_G$ be the grammar listed in Figure 7.3. Clearly, $L(G) = L(A_0) = \{a^k | k \geq 1\}$. Let $d_\varepsilon$ denote the rightmost derivation of $\varepsilon$ from $B_p$ in $G$. Since the subset of rules which LHS is $B_i$ for $1, \ldots, p$ constitute the grammar $G_{n,p}$ of Theorem 7.4, $d_\varepsilon$ attains the bounds stated in the theorem. We show by induction on $k \geq 1$ that the rightmost derivation of $a^k$ from $A_0$ attains the bounds of the theorem. If $k = 1$, the shortest rightmost derivation of $a$ from $A_0$ is obtained from the rewrite

$$A_0 \Rightarrow B^a_p A_1 \Rightarrow B^2_p A_2 \cdots \Rightarrow B^{an}_p A_u \Rightarrow B^{(u+1)n}_p a \Rightarrow a$$

The time complexity of this derivation is

$$t = u + 1 + (u + 1)n \cdot O_t(d_\varepsilon) = (1 + n \cdot O_t(d_\varepsilon))(u + 1)$$

$$= (1 + n \cdot O_t(d_\varepsilon))[(2u + 2)|w| - (u + 1)]$$

and the space complexity is

$$s = |a| + |B^{(u+1)n}_p| - |B_p| + O_s(d_\varepsilon) = 1 + n(u + 1) - 1 + O_s(d_\varepsilon)$$

$$= (1 + n(u + 1)|w| + O_s(d_\varepsilon) - 1$$

By applying the bounds of Theorem 7.4 we get

$$t = \begin{cases} 
(1 + nD_G)[(2u + 2)|w| - (u + 1)] & n \leq 1 \\
[1 + (nD_G - n)/(n - 1)][(2u + 2)|w| - (u + 1)] & n > 1
\end{cases}$$
and space
\[
s = \begin{cases} 
(1 + n(u + 1)|w| & n \leq 1 \\
(1 + n(u + 1)|w| + (D_G - 1)(n - 1) & n > 1 
\end{cases}
\]
as stated in the theorem. Assume then, that \( k > 1 \) and, as the induction hypothesis, that the derivation of terminal strings shorter than \( k \) attain the specified bounds. Since \( |a^k| > 1 \), in order to derive \( a^k \), \( A_0 \) must first derive the sentential form \( B_{p}^{(u+1)n} A_0' A_0 \), where \( A_0' \) derives \( a \) and \( A_0 \) derives \( a^{k-1} \).

\[
A_0 \Rightarrow B_{p}^{u} A_u \Rightarrow B_{p}^{(u+1)n} A_0' A_0 \Rightarrow B_{p}^{(u+1)n} A_0' a^{k-1} \Rightarrow B_{p}^{(u+1)n} a^{k-1} \Rightarrow a^k
\]
The induction hypothesis implies that \( A_0 \) derives \( a^{k-1} \) in time
\[
t = \begin{cases} 
(1 + nD_G)(2u + 2)(k - 1) - (u + 1) & n \leq 1 \\
[1 + (nD_G - 1)(n - 1)](2u + 2)(k - 1) - (u + 1) & n > 1 
\end{cases}
\]
and space
\[
s = \begin{cases} 
(1 + (u + 1)n)(k - 1) & n \leq 1 \\
(1 + (u + 1)n)(k - 1) + (D_G - 1)(n - 1) & n > 1 
\end{cases}
\]
Thus, if \( n \leq 1 \), \( A_0 \) derives \( a^k \) simultaneously in time
\[
t = 2(u + 1) + (u + 1)nD_G + (1 + nD_G)(2u + 2)(k - 1) - (u + 1) + (u + 1)nD_G
\]
\[
= (2u + 2) + (2u + 2)nD_G + (1 + nD_G)(2u + 2)(k - 1) - (u + 1)
\]
\[
= (1 + nD_G)(2u + 2) + (1 + nD_G)(2u + 2)(k - 1) - (u + 1)
\]
\[
= (1 + nD_G)(2u + 2)k - (u + 1)
\]
and space
\[
s = (u + 1)n + 1 + (1 + n(u + 1))(k - 1) = 1 + n(u + 1) + (1 + n(u + 1))(k - 1)
\]
\[
= (1 + n(u + 1))k = (1 + n(u + 1))|w|
\]
and if \( n > 1 \), \( A_0 \) derives \( a^k \) simultaneously in time
\[
t = 2(u + 1) + (u + 1)n[(nD_G - 1)/(n - 1)] + \\
[1 + (nD_G - 1)/(n - 1)](2u + 2)(k - 1) - (u + 1) + \\
(u + 1)n[(nD_G - 1)/(n - 1)]
\]
\[
= (2u + 2) + (2u + 2)[(nD_G - 1)/(n - 1)] + \\
[1 + (nD_G - 1)/(n - 1)](2u + 2)(k - 1) - (u + 1)
\]
\[
= (2u + 2)[1 + (nD_G - 1)/(n - 1)] + \\
[1 + (nD_G - 1)/(n - 1)](2u + 2)(k - 1) - (u + 1)
\]
\[
= [1 + (nD_G - 1)/(n - 1)](2u + 2)k - (u + 1)
\]
\[
= [1 + (nD_G - 1)/(n - 1)](2u + 2)|w| - (u + 1)
\]
and space

\[ s = (u + 1)n + 1 + (1 + n(u + 1))(k - 1) + (D_G - 1)(n - 1) \]
\[ = 1 + n(u + 1) + (1 + n(u + 1))(k - 1) + (D_G - 1)(n - 1) \]
\[ = (1 + n(u + 1))k + (D_G - 1)(n - 1) \]
\[ = (1 + n(u + 1))|w| + (D_G - 1)(n - 1) \]

\[ \square \]

## 7.2 DAMG Derivation Complexity

As we have shown in the previous section, the time and space complexities of CFG rightmost derivations have polynomial and exponential dependencies in certain properties of their grammars. However, since the underlying set of production rules is fixed for all derivations of a given grammar, these constructs can be dismissed as "constants" leaving only a linear dependency in the length of the derived string. Since the time and space complexities of CFG derivations directly bound the number of parse-loop iterations and parse stack size of LR(\(k\)) parsers, this reasoning is also applied when considering the time and space complexities of such parsers as being linear in the length of the input string.

When considering the time and space complexities of AMG derivations, these inherent non-linear dependencies can no longer be dismissed as constants. Throughout the derivation these values change along with the underlying rule sets. The extent and number of the modifications is directly related to the rules applied in the derivation and thus to the length and content of the derived string. Furthermore, since the AMG model consists of multi-pass derivation steps, the complexity of derivations must be expressed in the length of all the strings derived in the various "passes" and not just that of the final configuration.

In this section, we formally define the complexity of AMG derivations, and prove that given a fixed bound on the size of rule sets and number of grammar modifications, the complexity of complete AMG derivations of deterministic AMGs remain linear in the length of the derived terminal strings. An AMG derivation can be considered as a sequence of CFG-like derivations separated by derivation steps consisting of rule set modifications or multi-pass rules. We first establish bounds on the time and space complexities of individual of CFG-like derivations (Lemma 7.11). We then proceed by setting bounds on the time and space complexities of single-pass derivations (Lemma 7.13) and complete AMG derivations in deterministic AMGs (Theorem 7.15).

The following lemma states complexity bounds which may be uniformly applied to all CFG rightmost derivations.

**Lemma 7.10.** Let \( G(N, T, P, S) \) be a CFG, \( \alpha \) a nonempty symbol strings, \( \beta \) a symbol string, and \( d \) a derivation of \( \beta \) from \( \alpha \) by the rule string \( \pi \) in \( G \) where \( \pi \) is the shortest rule string by which \( \beta \) can be derived from \( \alpha \). Then

\[ O_t(d) \leq (1 + D_G + W_G^{D_G + 1})(2H_G + 2)|\beta| + |\alpha|(W_G^{D_G} + D_G) \]

and

\[ O_s(d) \leq |\alpha| + (1 + W_G(H_G + 1))|\beta| + W_GD_G \]
Proof. Let $u = H_G$ and $n = W_G$. If $β = ε$, let $α = X_1X_2⋯X_{|α|}$ where $X_i ∈ N$ for all $i = 1, \ldots, |α|$. Theorem 7.4 implies that if $n ≤ 1$,

$$O_t(d) ≤ \sum_{i=1}^{|α|} D_G(X_i) ≤ |α|D_G$$

$$≤ (1 + D_G + n^{D_G+1})(2u + 2)|β| + |α|(n^{D_G} + D_G)$$

and

$$O_s(d) ≤ |α| ≤ |α| + (1 + n(u + 1))|β| + nD_G$$

and, if $n > 1$

$$O_t(d) ≤ \sum_{i=1}^{|α|} (n^{D_G(A)} - 1)/(n - 1) ≤ |α|n^{D_G}$$

$$≤ (1 + D_G + n^{D_G+1})(2u + 2)|β| + |α|(n^{D_G} + D_G)$$

and

$$O_s(d) ≤ |α| - 1 + (D_G(A) - 1)(n - 1) + 1 ≤ |α| + nD_G$$

$$= |α| + (1 + n(u + 1))|β| + nD_G$$

On the other hand, if $β ≠ ε$, Theorem 7.8 implies (since $l < |α|$ and $|α_1 ⋯ α_{t+1}| ≥ 1$) that if $n ≤ 1$

$$O_t(d) ≤ (1 + nD_G)[(2u + 2)|β| - (u + 1)] + |α|D_G$$

$$≤ (1 + nD_G)(2u + 2)|β| + |α|D_G$$

$$≤ (1 + D_G + n^{D_G+1})(2u + 2)|β| + |α|(n^{D_G} + D_G)$$

and

$$O_s(d) ≤ |β| + |α| + n(u + 1)(|β| - |α_1 ⋯ α_{t+1}| + 1)$$

$$≤ |β| + |α| + n(u + 1)|β|$$

$$= |α| + (1 + n(u + 1))|β| + nD_G$$

and, if $n > 1$

$$O_t(d) ≤ (1 + (n^{D_G+1} - n)/(n - 1))[(2u + 2)|β| - (u + 1)] + |α|(n^{D_G} - 1)/(n - 1)$$

$$≤ (1 + n^{D_G+1})(2u + 2)|β| + |α|n^{D_G}$$

$$≤ (1 + D_G + n^{D_G+1})(2u + 2)|β| + |α|(n^{D_G} + D_G)$$

and

$$O_s(d) ≤ |β| + |α| + n(u + 1)(|β| - |α_1 ⋯ α_{t+1}| + 1) + (D_G - 1)(n - 1) - 1$$

$$≤ |β| + |α| + n(u + 1)|β| + nD_G$$

$$= |α| + (1 + n(u + 1))|β| + nD_G$$

□
We define the complexity of AMG derivation similarly to that of CFG rightmost derivations.

**Definition 7.5 (AMG derivation complexity).**
Let $G(N,T,R_0,S,\Delta,k)$ be an AMG, and $d = ((\gamma_0,R_0), (\gamma_1,R_1), \ldots, (\gamma_n,R_n))$ an AMG derivation of length $n$ in $G$. The time complexity of $d$, denoted $O_t(d)$, is defined as its length $n$, and the space complexity of $d$, denoted $O_s(d)$, is defined as the length of the longest of the strings $\bar{\gamma_0}, \ldots, \bar{\gamma_n}$.

The following lemma proves that the space complexity of CFG-like sub-derivations of complete AMG derivations in deterministic AMGs which rule sets’ sizes are bound, is linear in the size of the derived string.

**Lemma 7.11.** Let $G(N,T,R_0,S,\Delta,k)$ be a deterministic AMG, and $s \in \mathbb{N}$ a constant. If $d$ is a complete derivation in $G$ where $|G\xi| \leq s$ for every rule set $R \subseteq R_{N,T}$ appearing in $d$, then there exists a constant $c$ such that the space complexity of every CFG-like sub-derivation of $d$ of some annotated symbol string $\beta$ in $G$ is bound by $c|\beta| + c$.

**Proof.** Let $\pi$ be the rule string underlying $d$, and $d'$ an arbitrary CFG-like sub-derivation of $d$ with respect to $R \subseteq R_{N,T}$ of $\beta$ from $\alpha$ by rule string $\pi'$ in $G$, $M$ the AMP of $G$, and $R' = R \cup \{S' \rightarrow S\}$. By Lemma 5.10 there exists an adaptive multi-pass parse $\pi''$ of $x$ in $M$ such that $\tau(\pi'') = \pi R$. Since $G$ is deterministic, $\pi''$ is the only parse of $x$ in $M$. Let $\alpha_2$ be the longest substring of $\alpha$ such that $\alpha = \alpha_1\alpha_2\alpha_3$, $\pi' = \pi_3\pi_2\pi_1$, $\alpha_1$ derives $\beta_1$ by $\pi_1$, $\alpha_2$ derives $\varepsilon$ by $\pi_2$, $\alpha_3$ derives $\beta_3$ by $\pi_3$, and $\beta = \beta_1\beta_3$. Since no rule in $\pi_2$ contains a terminal symbol at its RHS, $M$ performs no Shift actions while reducing the empty string to $\alpha_2$. Further, $|G\xi| < s$ implies that there exists a constant $c_2$ such that $|M\xi| \leq c_2$. As there are at most $|M\xi|(|T| + k)^k$ distinct LR($k$)-items with respect to $M\xi$, we conclude that while producing $\pi_2$, $M$’s prefix-automaton consists of at most $c_3 = 2^{s_2(|T|+1)^k}$ states and $c_4 = c_3^2$ edges. If $\pi_2 = \varepsilon$, $\alpha_2 = \varepsilon$, $|\alpha| \leq |\beta|$ and for any constant $c \geq 0$ we have $|\alpha| \leq c|\beta| + c$. Otherwise, $|\pi_2| > 0$ and

$$(\alpha_1\alpha_2\beta_3, R_1) \xrightarrow{\pi_2}(\alpha_1\beta_3, R)$$

implies (by Lemma 5.5) that there exist an action string $\pi'_2$, a state string $\phi_2$ and a rule set $R_1$ such that for any state string $\phi_1$

$$(\phi_1, \alpha_1|\beta_3S, R') \xrightarrow{\pi'_2}(\phi_2, \alpha_1\alpha_2|\beta_3S, R'_1)$$

in $M$ where $R'_1 = R_1 \cup \{S' \rightarrow S\}$ and $\tau(\pi'_2) = \pi_2 R$. Since $M$ is deterministic, and its input stream remains the same throughout the reduction sequence $\pi'_2$ (as there are no Shift actions), we conclude that given a state $s$, $M$ will execute exactly the same Reduce action whenever $s$ becomes its current state. Assume towards a contradiction that $|\alpha_2| > c_4$. Since each stack symbol $A$ in $\alpha_2$ represents a transition between two states (i.e., an edge) following a Reduce action by a rule which LHS symbol is $A$, we conclude that during the reduce sequence $\pi'_2$, at least one edge of the prefix-automaton is represented twice in the parse stack. The determinism of $M$ then implies that $M$ entered an infinite loop which contradicts $\pi_2$ being of finite length, which further contradicts $\pi$ being a parse of $x$ in $G$. Thus, the size of $\alpha_2$ and every other sequence of nonterminals deriving the empty string in $d'$ is bound by $c_4$. Since there can be at most $|\beta| + 1$ such maximal sequences in $\alpha$ (separated by at most $|\beta|$ symbols that do not derive the empty string in $d'$), we conclude that $|\alpha| \leq (|\beta| + 1)c_4$. Thus, there exist a constant $c = c_4$ such that $|\alpha| \leq c|\beta| + c$ also for the case $\pi_2 \neq \varepsilon$. Finally, since the same bound applies on the length of the first annotated symbol string of every suffix of $d'$ we conclude that the space complexity of $d'$ is $c|\beta| + c$ as stated by the lemma.
The following lemma extends the results of the previous lemma and proves that both the time and space complexities of CFG-like sub-derivations of complete AMG derivations in deterministic AMGs which rule sets’ sizes are bound, are linear in the size of the derived string.

**Lemma 7.12.** Let $G(N, T, R_0, S, Δ, k)$ be a deterministic AMG, and $s ∈ \mathbb{N}$ a constant. If $d$ is a complete derivation in $G$ where $|G^R_\xi| ≤ s$ for every rule set $R ≤ R_{N,T}$ appearing in $d$, then there exists a constant $c$ such that the time and space complexities of every CFG-like sub-derivation of $d$ of some annotated symbol string $β$ in $G$ are bound by $c|β| + c$.

**Proof.** Let $d'$ be an arbitrary CFG-like sub-derivation of $d$ with respect to $R ≤ R_{N,T}$ of $β$ from $α$ by rule string $π'$ in $G$, and let $G^R_ξ = G'$. Lemma 6.4 implies that $d'$ is a rightmost derivation of $β$ from $α$ in $G^R_ξ$, and since $G$ is deterministic, Lemma 6.7 implies that it is also the shortest. Thus, we conclude (by Lemma 7.10) that

$$O_t(d') = O_t(d') ≤ (1 + D_{G'} + W_{G'}D_{G'}+1)(2H_{G'} + 2)|β| + |α|(W_{G'}D_{G'} + D_{G'})$$

Since $D_{G'} ≤ |G'|$, $W_{G'} ≤ |G'|$, $H_{G'} ≤ |G'|$, and $|G'| ≤ s$ we have

$$O_t(d') ≤ (1 + s + s^{s+1})(2s + 2)|β| + |α|(s^s + s)$$

Thus, we can write $O_t(d') ≤ c_2|β| + c_4|α|$ where $c_2 = (1+s+s^{s+1})(2s+2)$, and $c_4 = s^s + s$. Lemma 7.11 implies that there exists a constant $c_3$ such that $|α| ≤ O_s(d') ≤ c_4|β| + c_4$, which further implies that

$$O_t(d') ≤ c_2|β| + c_3(c_4|β| + c_4) = (c_2 + c_3c_4)|β| + c_3c_4$$

Thus, we conclude that there exists a constant $c = c_2 + c_3c_4$ such that $O_t(d') ≤ c|β| + c$ and $O_s(d') ≤ c|β| + c$. \(\square\)

We now formally define AMG derivations with multiple passes. Given an AMG derivation, each "pass" corresponds to a sub-derivation of maximal length that does not consist of multi-pass derivation steps. Except for the first pass of the derivation, each pass (by rule string $π_i$ where $1 ≤ i ≤ m$) is preceded by a multi-pass derivation step, where the annotated terminal string $w_i$ is rewritten with the RHIS ($ω_i$) of a multi-pass rule ($r_i$).

**Definition 7.6 (AMG derivation with $m$ passes).**

Let $G(N, T, R_0, S, Δ, k)$ be an AMG. An AMG derivation with $m$ passes by rule string $π$ in $G$ is a derivation of the form

$$(γ(1,0), R(1,0)) \xrightarrow{x} (γ(1,|π_1|), R(1,|π_1|)) = (ρ_2ω_2x_2, R(1,|π_1|)) \xrightarrow{x} (ρ_2ω_2x_2, R(2,0)) = (γ(2,0), R(2,0)) \xrightarrow{x} (γ(2,|π_2|), R(2,|π_2|)) = (ρ_3ω_3x_3, R(2,|π_2|)) \xrightarrow{x} (ρ_3ω_3x_3, R(3,0)) = (γ(3,0), R(3,0)) \xrightarrow{x} (γ(3,|π_3|), R(3,|π_3|)) = \cdots$$

$$(γ(m-1,0), R(m-1,0)) \xrightarrow{x} (γ(m-1,|π_{m-1}|), R(m-1,|π_{m-1}|)) = (ρ_mω_mx_m, R(m-1,|π_{m-1}|)) \xrightarrow{x} (ρ_mω_mx_m, R(m,0)) = (γ(m,0), R(m,0)) \xrightarrow{x} (γ(m,|π_m|), R(m,|π_m|))$$

where $m > 0$, $ξ(r_j) = A_j → ω_j$, $r_j ∈ R_{N,T}^*$, $π_j ∈ R_{N,T}^*$, $ω_j, ρ_j, γ(j,0), γ(j,1), \ldots, γ(j,|π_j|) ∈ A_{N,T}^*$, $ω_j, x_j ∈ A_{T}^*$, and $R(j,0), R(j,1), \ldots, R(j,|π_j|) ≤ R_{N,T}$ for all $j = 1, \ldots, m$. 
We call an AMG derivation with \( m = 1 \) passes a single-pass derivation. Next, we prove that the time and space complexities of single-pass sub-derivations of complete AMG derivations in deterministic AMGs with a bounded number of grammar modifications and which rule sets’ sizes are bound, are linear in the size of the derived string.

**Lemma 7.13.** Let \( G(N,T,R_0,S,\Delta,k) \) be a deterministic AMG, and \( s,g \in \mathbb{N} \) constants. If \( d \) is a complete derivation in \( G \) consisting of at most \( g \) grammar modifications and \( |G^R| \leq s \) for every rule set \( R \subseteq R_{N,T} \) appearing in \( d \), then there exists a constant \( c \) such that the time and space complexities of every sub-derivation of \( d \) by rule string \( \pi' \in R^*_{N,T} \) of some annotated symbol string \( \beta \) in \( G \) are bound by \( c|\beta| + c \).

**Proof.** Since \( \pi' \) does not consist of multi-pass rules, we can regard \( d' \) as a sequence of consecutive CFG-like sub-derivations. Let \( p \geq 1 \) be the minimal integer such that \( \pi' = \pi_1 \pi_2 \cdots \pi_p \) and for all \( i = 1, \ldots, p \), each sub-derivation \( d_i \) of \( d' \) of \( \gamma_i \) from \( \gamma_{i-1} \) by rule string \( \pi_i \) in CFG-like with respect to \( R_i \), and \( \gamma_p = \beta \). The minimality of \( p \) implies that each two adjacent derivations are CFG-like with respect to different rule sets, which further implies that \( p \leq g + 1 \). By Lemma 7.13 we conclude that there exist a constant \( c_1 \) such that the time and space complexities of every derivation \( d_i \) are bound by \( c_1|\gamma_i| + c_1 \) for \( i = 1, \ldots, p \), which further implies that

\[
|\gamma_i| \leq c_1|\gamma_i| + c_1 \leq c_1^2|\gamma_{i+1}| + c_1^2 + c_1 \leq c_1^3|\gamma_{i+2}| + c_1^3 + c_1^2 + c_1 \leq \cdots \\
\leq c_1^{p+i-1}|\gamma_p| + c_1^{p+i-1} + c_1^{p+i} + c_1^{p+i-1} + \cdots + c_1^2 + c_1
\]

Thus, there exists a constant \( c_2 = c_1^p + c_1^{p-1} + \cdots + c_1^2 + c_1 \) such that

\[
|\gamma_{i-1}| \leq c_2|\gamma_p| + c_2 = c_2|\beta| + c_2
\]

Furthermore, we can write

\[
O_t(d') = O_t(d_1) + O_t(d_2) + \cdots + O_t(d_p) \\
\leq (c_1|\gamma_1| + c_1) + (c_1|\gamma_2| + c_1) + \cdots + (c_1|\gamma_p| + c_1) \\
= c_1(|\gamma_1| + |\gamma_2| + \cdots + |\gamma_p|) + c_1p \\
\leq c_1p(c_2|\beta| + c_2) + c_1p \\
= c_1c_2p|\beta| + (c_1c_2 + c_1)p \\
\leq c_1c_2(g + 1)|\beta| + (c_1c_2 + c_1)(g + 1)
\]

and

\[
O_s(d') = \max\{O_s(d_1), O_s(d_2), \ldots, O_s(d_p)\} \\
\leq \max\{c_1|\gamma_1| + c_1, c_1|\gamma_2| + c_1, \ldots, c_1|\gamma_p| + c_1\} \\
= c_1 \max\{|\gamma_1|, |\gamma_2|, \ldots, |\gamma_p|\} + c_1 \\
\leq c_1(c_2|\beta| + c_2) + c_1 \\
= c_1c_2|\beta| + c_1c_2 + c_1
\]

and we conclude that there exists a constant \( c = (c_1c_2 + c_1)(g + 1) \) such that the time and space complexities of \( d' \) are bound by \( c|\beta| + c \). \( \square \)

The following lemma proves that the time and space complexities of suffixes of complete AMG derivations in deterministic AMGs with a bounded number of grammar modifications and which rule sets’ sizes are bound, are linear in the sum of lengths of the derived annotated terminal string (\( w_1 \)) and all strings rewritten due to multi-pass derivation steps (\( w_2, w_3, \ldots, w_m \)).
Lemma 7.14. Let $G(N,T,R_0,S,\Delta,k)$ be a deterministic AMG, and $s,g \in \mathbb{N}$ constants. If $d$ is a complete derivation of $w_1 \in \mathcal{A}^*_T$ in $G$ consisting of at most $g$ grammar modifications and $|G_R^R| \leq s$ for every rule set $R \subseteq \mathcal{R}_{N,T}$ appearing in $d$, then there exists a constant $c$ such that the time and space complexities of every suffix $d'$ of $d$ with $m > 0$ passes by rule string $\pi'$ are bound by $c(|w_1| + |w_2| + |w_3| + \cdots + |w_m|) + c$.

Proof. The proof is by induction on $m > 0$. If $m = 1$, $d'$ is an AMG derivation by rule string $\pi' \in \mathcal{R}_{N,T}^*$ in $G$, and by Lemma 7.13, there exists a constant $c$ such that its time and space complexities are bound by $c|w_1| + c$. Assume then that $m > 1$ and that the lemma holds for suffixes of $d$ with fewer than $m$ passes. $m > 1$ implies that we can write $\pi' = \pi_1r_2\pi_2$ such that $d_1$ is the AMG derivation of $\gamma(1,|\pi_1|),R_{(1,|\pi_1|)}$ from $\gamma(1,0),R_{(1,0)}$ by rule string $\pi_1 \in \mathcal{R}_{N,T}^*$ in $G$, and $d_2$ is the AMG derivation of $\gamma(\pi_1,|\pi_2|),R_{(\pi_1,|\pi_2|)}$ from $\gamma(2,0),R_{(2,0)}$ with $m - 1$ passes by rule string $\pi_2$ in $G$. Since $d_2$ consists of at most $g$ grammar modifications, we conclude (by applying the induction hypothesis) that there exists a constant $c_1$ such that its time and space complexities are bound by $c_1(|w_3| + |w_4| + \cdots + |w_m| + |w_1|) + c_1$. On the other hand, $|\gamma(1,|\pi_1|)| = |\rho_2w_2x_2| = |\rho_2\omega_2x_2| - |\omega_2| + |w_2| \leq |\rho_2\omega_2x_2| + |w_2|$. Since $\rho_2\omega_2x_2 = \gamma(2,0)$ appears in $d_2$, we conclude that

$$|\gamma(1,|\pi_1|)| \leq |w_2| + c_1(|w_3| + |w_4| + \cdots + |w_m| + |w_1|) + c_1$$

$$\leq c_1(|w_1| + |w_2| + |w_3| + \cdots + |w_m|) + c_1$$

Lastly, by Lemma 7.13 there exists a constant $c_2$ such that the time and space complexities of $d_1$ are bound by

$$c_2|\gamma(1,|\pi_1|)| + c_2 \leq c_2c_1(|w_1| + |w_2| + |w_3| + \cdots + |w_m|) + c_2c_1 + c_2$$

Thus, there exists a constant $c = 2(c_2c_1 + c_2)$ such that

$$O_t(d') = O_t(d_1) + O_t(d_2) \leq c(|w_1| + |w_2| + |w_3| + \cdots + |w_m|) + c$$

and

$$O_s(d') = \max\{O_s(d_1),O_s(d_2)\} \leq c(|w_1| + |w_2| + |w_3| + \cdots + |w_m|) + c$$

as stated by the lemma. \qed

The following theorem is implied directly from Lemma 7.14.

Theorem 7.15. Let $G(N,T,R_0,S,\Delta,k)$ be a deterministic AMG, and $s,g \in \mathbb{N}$ constants. If $d$ is a complete derivation of $w_1 \in \mathcal{A}^*_T$ with $m > 0$ passes in $G$ consisting of at most $g$ grammar modifications and $|G_R^R| \leq s$ for every rule set $R \subseteq \mathcal{R}_{N,T}$ appearing in $d$, then the time and space complexities of $d$ are linear in $|w_1| + |w_2| + |w_3| + \cdots + |w_m|$. 

Technion - Computer Science Department - M.Sc. Thesis MSC-2010-11 - 2010
Chapter 8

Efficient DAMG(1) Parsing

In this chapter we state bounds on the time and space complexities of adaptive multi-pass parses by deterministic adaptive multi-pass parsers utilizing a single lookahead symbol. These parsers can recognize the languages and produce the derivations of deterministic AMGs with a single lookahead symbol. As shown in Chapter 6, this class of grammars is of great practical importance as its grammars maintain full computational power (i.e., Turing powerful) while posing minimal restrictions on grammar adaptivity. In Sections 8.1 and 8.2 we describe the various constructs and data structures employed by our parser and analyze the time and space complexities of performing certain operations on sets and graphs containing these elements. In Section 8.3 we present an efficient algorithm for computing LR(1) item sets, prove its correctness and determine its time and space complexities. Since our adaptive multi-pass parser lazily generates its prefix-automaton at parse time, the efficiency of this task is crucial to the overall performance of the parsing process. In Section 8.4 we analyze the time and space complexities of the various subroutines of the parser and of its main parsing loop. We conclude the chapter by stating bounds on the complexity of complete adaptive multi-pass parses.

8.1 Vocabularies, Rules and Item Sets

In this section we describe the various constructs and data structures employed by our parser. We start with the representation of grammar symbols (e.g., terminals) and analyze the time and space complexity of performing certain operations on sets containing these symbols (Lemmas 8.1 and 8.2). Next, we describe the representation of AMG rules, their underlying skeletons, LR(1) items and item sets.

Let $G(N, T, R_0, S, \Delta, k)$ be an AMG, $M$ its AMP, $\pi'$ an action string, $(\phi, \gamma|y, R)$ an AMP configuration of $M$ such that $(\epsilon, \epsilon|x\$, $R_0 \cup \{S' \rightarrow S\}) \xrightarrow{\pi'} (\phi, \gamma|y, R)$ and let $M^R = (N', T', P', S')$. Each element $e$ of the set $N' \cup T' \cup \{\epsilon\}$ is represented by a single object $e$ of size $O_s(1)$ with reference $r(e)$ such that $e, e'$ represent the same element if and only if $r(e) = r(e')$. Furthermore, $\epsilon$ is associated with the integer $n(\epsilon) = 0$, and each terminal $t \in T$ is associated with a unique integer in the range $1, \ldots, |T'|$ denoted by $n(t)$. Only the nonterminals which appear in $N'$ (that is, the rules of the set $Rules$) are explicitly represented by $M$. Since this set is finite, so is the set of represented nonterminals. Unless specified otherwise, we use $N'$ to denote the finite set of nonterminals appearing in rules of the set $Rules$. Each nonterminal $A \in N'$ is associated with a unique integer in the range $|T'| + 1, \ldots, |T'| + |N'|$ denoted by $n(A)$. This unique integer is recomputed whenever the
skeletal CFG underlying $M$ changes. Given a symbol $X$, the test $X \in N'$ is performed by computing $n(X) > |T'|$ in constant time.

Subsets of the set $N' \cup T' \cup \{\varepsilon\}$ are organized as linked lists. Set assignment $s_1 \leftarrow s_2$ is performed by assigning the value of the reference $s_2$ to $s_1$. Set union $s_1 \cup s_2$ is performed using an auxiliary array

$$\tau : \{0, \ldots, |T'| + |N'|\} \rightarrow \{0, 1\}$$

which values are initially set to 0. First, a new reference $s_3 \leftarrow s_1$ is initialized to hold the set resulting from the union. Then, $\tau[n(e)] \leftarrow 1$ is set for every element $e \in s_1$. Next, each element $e' \in s_2$ such that $\tau[n(e')] = 0$ is linked before the link referenced by $s_3$ and $s_3$ is set to reference the new link. Finally, $\tau[n(e)] \leftarrow 0$ is set for every element $e \in s_1$. By adjusting the size and indices of the auxiliary array for operations over subsets of terminals ($\tau : \{0, \ldots, |T'|\} \rightarrow \{0, 1\}$) and nonterminals ($\{|T'| + 1, \ldots, |N'| + |T'|\} \rightarrow \{0, 1\}$) we obtain the following lemma.

**Lemma 8.1.** Let $s_1, s_2$ be subsets of $N' \cup T' \cup \{\varepsilon\}$. Then a set assignment operation $s_1 \leftarrow s_2$ is performed in constant time and space, and a set union operation $s_1 \cup s_2$ using an initialized auxiliary array is executed in time $O_i(|s_1| + |s_2|)$ and space

$$O_s = \begin{cases} 
O(|T'|) & s_1, s_2 \subseteq T' \cup \{\varepsilon\} \\
O(|N'|) & s_1, s_2 \subseteq N' \\
O(|N'| + |T'|) & s_1, s_2 \subseteq N' \cup T'
\end{cases}$$

A sequence of union operations

$$s_1 \leftarrow s_1 \cup s_2, s_1 \leftarrow s_1 \cup s_2, \ldots, s_1 \leftarrow s_1 \cup s_n$$

where $s_i \subseteq N' \cup T' \cup \{\varepsilon\}$ for all $i = 1, \ldots, n$ is performed using an auxiliary array dedicated to $s_1$

$$\tau : \{0, \ldots, |T'| + |N'|\} \rightarrow \{0, 1\}$$

which values are all initially set to reflect the contents of $s_1$

$$\tau[n(e)] = \begin{cases} 
1 & e \in s_1 \\
0 & e \notin s_1
\end{cases}$$

For all $1 < i \leq n$, each element $e \in s_i$ for which $\tau[n(e)] = 0$ is linked before the link referenced by $s_1$, $\tau[n(e)] \leftarrow 1$, and $s_1$ is set to reference the new link. Again, by adjusting the indices of the auxiliary array we obtain the following lemma.

**Lemma 8.2.** A sequence of set unifications

$$s_1 \leftarrow s_1 \cup s_2, s_1 \leftarrow s_1 \cup s_2, \ldots, s_1 \leftarrow s_1 \cup s_n$$

can be executed using an initialized auxiliary array dedicated to $s_1$ in time

$$O_i(|s_2| + |s_3| + \cdots + |s_n|)$$

and space

$$O_s = \begin{cases} 
O(|T'|) & s_1, \ldots, s_n \subseteq T' \cup \{\varepsilon\} \\
O(|N'|) & s_1, \ldots, s_n \subseteq N' \\
O(|N'| + |T'|) & s_1, \ldots, s_n \subseteq N' \cup T' \cup \{\varepsilon\}
\end{cases}$$
An AMG rule skeleton \( r = A \rightarrow X_1 \cdots X_n \) \((n \geq 0)\) is a pair \((r_l, r_r)\) of size \(O_s(n)\), where \(r_l\) is a reference to the LHS nonterminal \(A\) and \(r_r\) is an array of size \(n\) representing \(r\)'s RHS, such that \(r_r[i]\) is a reference to the symbol \(X_i \in N' \cup T'\) for all \(1 \leq i \leq n\). Each CFG rule \(r\) is represented as a single object with reference \(r(r)\) such that \(r, r'\) represent the same rule if and only if \(r(r) = r(r')\). Each nonterminal \(A \in N'\) is associated with a double-linked list \(\xi(A)\) of AMG rule skeletons in which LHS it appears. The "next" and "back" links which link rules together are embedded within the objects representing the rules so that they can be added and removed from the list in constant time. The representation of an AMG rule is based on that of its skeleton. Associated with each skeleton rule object is a flag denoting whether its AMG rule is a multi-pass rule, and a reference to the rule’s annotation function. The \(Rules\) set is also implemented as a double-linked list and a second set of "next" / "back" links is included in rule objects to allow addition and removal of rules in constant time and space. An additional flag \(R(r)\) associated with each rule is set if and only if \(r \in Rules\) thus facilitates testing of rule membership in constant time and space.

An LR(1) items set \([A \rightarrow \alpha \bullet \beta, \Theta]\) is represented as a triplet \((r, |\alpha|, \Theta)\) where \(r\) is a reference to the rule skeleton \(A \rightarrow \alpha \beta\), \(|\alpha|\) indicates the number of symbols preceding the \(\bullet\) symbol, and \(\Theta\) is a reference to a subset of \(T' \cup \{\varepsilon\}\). An LR(1) state (i.e., item set) \(q\) is represented as a list which elements are items sets of the form \([A \rightarrow \alpha \bullet \beta, \Theta]\). The number of items sets contained in \(q\) is denoted by \(|q|\), and its size is denoted by \(O_s(q)\) (see Theorem 8.15).

### 8.2 Closures and Functions

Next, we state bounds on the time and space complexities of computing closures and restricted types of functions defined over relations.

**Lemma 8.3.** Let \(R\) be a relation on a finite set \(A\) where \(R\) is a multiset. Given a subset \(B\) of \(A\), \(R^*(B)\) can be computed in time and space \(O(max\{|R|, |A|\})\).

**Proof.** An algorithm for computing \(R^*(B)\) is given in [42] along with a formal correctness and time complexity proof. The algorithm treats the relation \(R\) as a graph which nodes are members of \(A\), and edges are members of \(R\). Each node is augmented with a flag that is used to mark visited nodes. It begins by clearing the flags of all nodes. Then initiates a DFS from each node in \(B\) using a recursive routine, marking all reachable nodes and skipping nodes that were already marked. Upon completion, the resulting set of marked nodes is \(R^*(B)\). If the relation \(R\) is represented as a multiset, there may be more than one edge connecting two nodes in the graph. As this does not affect the workings of a DFS and since such multiple edges are taken into account in the stated complexity bounds we conclude that the algorithm and its stated time complexity bounds are correct when \(R\) is represented as a multiset. It should be clear that augmenting each node of the graph with a boolean flag requires additional space \(O_s(|A|)\) and that the recursion stack of the DFS routine does not exceed the size \(O_s(|R|)\). Thus, we conclude that \(R^*(B)\) can be computed in space \(O_s(max\{|R|, |A|\})\).

**Lemma 8.4.** Let \(A, B\) be finite sets, \(R\) a relation on \(A\) where \(R\) is a multiset, and \(F_0 : A \rightarrow 2^B\) a function (whose values are known). Then the function \(F : A \rightarrow 2^B\) defined by

\[
F(a) = \bigcup_{aR^*b} F_0(b)
\]
can be computed for all \( a \in A \) in time and space
\[
O_t\left(\max\{t_1 \cdot |A|, t_2 \cdot |R|\}\right), O_s\left(\max\{|R|, |A||B|\}\right)
\]
where \( t_1 \) is the time required for one set assignment \((x \leftarrow F_0(b))\) operation, and \( t_2 \) is the time required for one set union \((x \leftarrow x \cup F_0(b))\) operation on subsets of \( F(A) \).

**Proof.** An algorithm for computing the function \( F \) above is given in [42] along with a formal correctness and time complexity proof. The algorithm treats the relation \( R \) as a graph which nodes are members of \( A \), and edges are members of \( R \). Each node is augmented with a flag that is used to mark visited nodes. In addition, associated with each node \( a \), is a set \( F_0 \) representing the values of \( F_0(a) \), and another, initially empty, set \( F \) that would contain the values of \( F(a) \) upon completion of the algorithm. The algorithm begins by clearing the flags of all nodes. Then initiates an extended form of Trajan’s algorithm for finding strongly connected components (SCC), which computes the image of each node \( a \) with respect to the function \( F \) by performing a single set unification per reachable node (i.e., per edge), and a single set assignment per SCC member (i.e., per node), while traversing the graph. Upon completion, the set \( F \) associated with each node \( a \) contains the elements of the image \( F(a) \). If the relation \( R \) is represented as a multiset, there may be more than one edge connecting two nodes in the graph. As this does not affect the workings of the algorithm and since such multiple edges are taken into account in the stated complexity bounds we conclude that the algorithm and its stated time complexity bounds are correct when \( R \) is represented as a multiset. It should be clear that augmenting each node of the graph with a boolean flag requires additional space \( O_s(|A|) \), that the recursion stack of Trajan’s algorithm does not exceed the size \( O_s(|R|) \), and that the SCC stack maintained by the algorithm does not exceed the size \( O_s(|A|) \). Moreover, the sets \( F \) computed by the algorithm are limited by the size of the function’s domain \( B \) and thus do not exceed the size \( O_s(|A||B|) \). Thus, we conclude that the function \( F \) can be computed in space \( O_s(\max\{|R|, |A||B|\}) \).

8.3 Efficient LR(1) item set construction

Traditionally, LR(1) prefix-automaton generation occurs only during parser construction, where it is used to derive a parse table. At parse-time this pre-built parse table guides the actions of the LR(1) parser. Since the size of the automaton is known to be exponential in the number of production rules (see [43]) and its generation does not occur at parse-time, little attention is given in the literature to the efficient construction of LR(1) prefix-automata. In fact, most texts describe numerous techniques for reducing parse-table size while presenting the most trivial prefix-automata generation algorithms, such as the traditional algorithm of Figure 2.2. Since our adaptive multi-pass parser lazily generates the prefix-automaton at parse time, the efficiency of this task is crucial to the overall performance of the parser.

We next present an efficient algorithm for computing LR(1) item sets. It improves upon the traditional algorithm by constructing a dependency graph representing the contribution of lookahead strings between non-kernel items. Lookahead sets are computed by traversing this graph in an SCC-aware fashion, thus avoiding the recurring redundant computation steps inherent in the traditional algorithm. Our algorithm is further optimized by taking advantage of the following property of LR(\(k\)) states.

\[
[A \rightarrow \bullet \alpha, y] \in [\gamma]_G^1 \implies \{[A \rightarrow \bullet \alpha', y]|A \to \alpha' \in P\} \subseteq [\gamma]_G^1
\]
It allows us to reduce the size of the dependency graph by representing sets of non-kernel items sharing the same LHS nonterminal by a single (nonterminal) node.

Let $G(N, T, P, S)$ be a CFG, $\gamma \in (N \cup T)^*$, and let $[\gamma]_G$ be an item set. Further, let

\[(a) \quad K_\gamma = \begin{cases} {[S \to \omega, \varepsilon] \in [\varepsilon]_G} & \gamma = \varepsilon \\ \text{KERNEL}([\delta]_G, X) & \gamma = \delta X \end{cases} \]

for some strings $\omega, \delta \in (N \cup T)^*$ and symbol $X \in N \cup T$. We call $K_\gamma$ the kernel of the item set $[\gamma]_G$, its items kernel items, and all items in $[\gamma]_G \setminus K_\gamma$ non-kernel items. Furthermore, we define the set of kernel non-terminals of $[\gamma]_G$ by

\[(b) \quad N^K_\gamma = \{ A|B \to \alpha \cdot A\beta, y \in K_\gamma \} \]

If non-empty, $K_\gamma$ represents a valid input argument provided by an LR(1) parse-table generator to the CREATESTATE routine. As we shall demonstrate, given this input set of kernel items, the set of all non-kernel items that are LR(1)-valid for the string $\gamma$ can be computed. Together with the set of kernel items, they represent the complete item set $[\gamma]_G$. The relation $\text{beginsRHS}$ on $N$ is defined by

\[(c) \quad \text{beginsRHS} = \{(A, B)|B \to A\alpha \in P\} \]

Using the inverse of this relation and the set of kernel-nonterminals $N^K_\gamma$, we can derive the set of LHS nonterminals of rules appearing in non-kernel items of $[\gamma]_G$. We formally define this set of non-kernel nonterminals by

\[(d) \quad N_\gamma = (\text{beginsRHS}^{-1})^*(N^K_\gamma) \]

We define the set of non-kernel rules of $[\gamma]_G$ by

\[(e) \quad P_\gamma = \{ A \to \omega \in P|A \in N_\gamma \} \]

As we shall soon prove, this set consists of exactly those rules appearing in non-kernel items of the item set $[\gamma]_G$. In order to determine proper lookahead components of non-kernel items, we next define the relation $\text{containsLA}$, and functions $F_a$, $F_b$ and $F_0$. The relation $\text{containsLA}$ on $N$ is defined by

\[(f) \quad \text{containsLA} = \{(A, B)|B \to A\alpha \in P_\gamma, \alpha \Rightarrow \varepsilon\} \]

The functions $F_a$, $F_b$, and $F_0$, with domain $N$ and range $2^T$, are defined by

\[(g) \quad F_a(A) = \{x|[B \to \alpha \cdot A\beta, y] \in K_\gamma, x \in \text{FIRST}_1(\beta y)\} \]

\[F_b(A) = \{x|[B \to A\alpha, x \in \text{FIRST}_1(\alpha) \setminus \{\varepsilon\}\} \]

\[F_0(A) = F_a(A) \cup F_b(A) \]

Finally, we define the function $F : N \to 2^T$ by

\[(h) \quad F(A) = \cup_{A \text{containsLA}} F_0(B) \]

As we shall prove, for each rule $A \to \omega \in P_\gamma$, we have

\[y \in F(A) \iff [A \to \omega, y] \in [\gamma]_G \setminus K_\gamma \]

This somewhat fragmented definition of $F$ facilitates the efficient computation of its values, as stated by Lemma 8.4.
INIT()
1 compute Nullable : \( N \rightarrow \{0, 1\} \)
2 compute First\(_1\) : \( N \rightarrow 2^T \)
3 for each \( A \) in \( N \) do
4 \( \tau(N_A)[n(A)] \leftarrow 0 \)
5 \( F_a(A) \leftarrow F_b(A) \leftarrow B(A) \leftarrow C(A) \leftarrow \{\} \)
6 compute beginsRHS\(^{-1}\)
7 for each \( r = A \rightarrow X_1 \cdots X_n \) in \( P \) do
8 \( N(r, n + 1) \leftarrow 1 \)
9 \( F(r, n + 1) \leftarrow \{\varepsilon\} \)
10 for \( i = n, \ldots, 1 \) do
11 \( F(r, i) \leftarrow \text{First\(_1\)}(X_i) \setminus \{\varepsilon\} \)
12 if Nullable(X_i) then
13 \( F(r, i) \leftarrow F(r, i) \cup F(r, i + 1) \)
14 if \( N(r, i + 1) = 1 \) then
15 \( N(r, i) \leftarrow 1 \)

CREATE\(_1\)(K\(_\gamma\))
1 compute \( K^R_\gamma, F_a \)
2 compute \( K^L_\gamma \)
3 compute \( \text{containsLA}, F_b \)
4 compute \( F_0 \)
5 compute \( F \)
6 \( q \leftarrow K_\gamma \)
7 for each \( A \) in \( N_\gamma \) do
8 for each \( A \rightarrow \omega \) in \( \xi(A) \) do
9 \( q \leftarrow q \cup \{[A \rightarrow \omega, F(A)]\} \)
10 return state

Figure 8.1: LR(1) item set computation routines

An algorithm for computing an LR(1) item set \( [\gamma]_G^1 \) given an input set of kernel items \( K_\gamma \) is listed in Figure 8.1. It consists of two routines. The INIT routine performs preliminary, state independent computations, and initializes data-structures used by the CREATE\(_1\) routine. It must be invoked once, before any LR(1) states are created. In the case of our adaptive parser, it should be called after every modification of the skeletal CFG underlying \( M \). In lines 1-2, the routine computes the function Nullable that determines for each nonterminal whether or not it is nullable, and the function First\(_1\) such that for each nonterminal \( A \)

\[
\text{First}_1(A) = 1 : L(A)
\]

In lines 3-5 it initializes the auxiliary array \( \tau(N_A) \), and the sets \( F_a(A), F_b(A), B(A), \) and \( C(A) \) associated with each nonterminal \( A \). The auxiliary array \( \tau(N_A) \) will be used for performing fast union operations over subsets of nonterminals while computing \( N_\gamma \). During the computation of LR(1) states, the sets \( F_a(A), F_b(A) \) and \( C(A) \) will be used to store the image of \( A \) under the relations \( F_a, F_b \) and \( \text{containsLA} \), respectively. \( B(A) \) sets are used to store the image of \( A \) under the relation beginsRHS\(^{-1}\) that is computed in line 6 of the routine. In the remaining lines (7-15), the routine computes for each rule \( r = A \rightarrow X_1 \cdots X_n \) and \( 0 \leq i \leq n + 1 \) the nullability of the RHS suffix \( X_i \cdots X_n \), denoted by \( N(r, i) \), and the image

\[
\text{First}_1(X_i \cdots X_n) = 1 : L(X_i \cdots X_n)
\]

denoted by \( F(r, i) \).
By utilizing the functions pre-computed by the INIT method, CREATESTATE1 efficiently computes the $LR(1)$ item set $[\gamma]_G$ given an input set of kernel items $K_\gamma$. In lines 1-5 it computes the function $F$ and the set of kernel nonterminals $N_\gamma$ as defined in (b), (d), (f), (g), and (h). The rule set $P_\gamma$, required for computing $F_0$ and containsLA, is implicitly computed by traversing the set $\xi(A)$ for each nonterminal $A$ in $N_\gamma$, as done in lines 7-8 of the routine. The routine completes by initializing a new item set $q$ with the input set of kernel items (line 6), adding to it all non-kernel items represented by $F$ and $N_\gamma$ (lines 7-9) and returning it. As we prove in Theorem 8.15 the resulting set of items $q$ is the item set $[\gamma]_G$.

We now prove the correctness of the algorithm and determine its time and space complexity. We begin with the INIT routine.

**Lemma 8.5.** Let $G(N,T,P,S)$ be a CFG, and $r = A \rightarrow X_1 \cdots X_n \in P$. Given the pre-computed functions $\text{NULLABLE}$ and $\text{FIRST}_1$, the INIT routine computes the nullability and $\text{FIRST}_1$ sets of all suffixes of $X_1 \cdots X_n$ in time and space $O(1 + n \cdot |T|)$.

**Proof.** We first prove the correctness of the algorithm by induction on the length of the suffix $X_1 \cdots X_n$ where $1 \leq i \leq n + 1$. Note that the algorithm computes $N(r,i)$ and $F(r,i)$ for the suffix $X_1 \cdots X_n$ only after computing these values for all suffixes $X_j \cdots X_n$ where $i < j \leq n + 1$. If $|X_1 \cdots X_n| = 0$, we have $X_1 \cdots X_n = \varepsilon$, $i = n + 1$ and in lines 8-9 the algorithm correctly computes $N(r,i) = 1$ and $F(r,i) = \{\varepsilon\}$. Assume then, that $|X_1 \cdots X_n| > 0$ and, as the induction hypothesis, that the algorithm correctly computed $N(r,j)$ and $F(r,j)$ per suffix $X_j \cdots X_n$ for all $i < j \leq n + 1$. $X_1 \cdots X_n \Rightarrow \varepsilon$ implies that $X_i$ and $X_{i+1} \cdots X_n$ are nullable. Thus, $\text{NULLABLE}(X_i) = 1$, by the induction hypothesis $N(r,i+1) = 1$, and from lines 10, 12, 14-15 we conclude that the algorithm correctly computes $N(r,i) = 1$. Similarly, $a \neq \varepsilon \in \text{FIRST}_1(X_1 \cdots X_n)$ implies that $a \in \text{FIRST}_1(X_i)$ or that $X_i$ is nullable and $a \in \text{FIRST}_1(X_{i+1} \cdots X_n)$. In the former case, by line 11 we have $a \in F(r,i)$. In the latter case, the induction hypothesis implies that $a \in F(r,i+1)$ and by lines 12-13 we conclude that $a \in F(r,i)$. Furthermore, $\varepsilon \in \text{FIRST}_1(X_1 \cdots X_n)$ implies that $X_i$ is nullable and $\varepsilon \in \text{FIRST}_1(X_{i+1} \cdots X_n)$ and, again by lines 12-13, we have $\varepsilon \in F(r,i)$. On the other hand, it is clear from lines 11-15 that $N(r,i) = 1$ implies that $X_i$ is nullable and $N(r,i+1) = 1$, and that $a \in F(r,i)$ implies that either $a \neq \varepsilon \in \text{FIRST}_1(X_i)$ or $X_i$ is nullable and $a \in F(r,i+1)$. By applying the induction hypothesis to these results we conclude that $X_i \cdots X_n$ is nullable, and that $a \in \text{FIRST}_1(X_1 \cdots X_n)$. Thus, we conclude that $N(r,i) = 1$ if and only if $X_i \cdots X_n$ is nullable, and $a \in F(r,i)$ if and only if $a \in \text{FIRST}_1(X_1 \cdots X_n)$.

Next, we prove the time and space bounds on the complexity of the computation. Associated with the rule $r$ are $n + 1$ flags $N(r,i) \ (1 \leq i \leq n + 1)$ each requiring constant space and can be set in constant time. During execution of lines 8-15, each flag is set exactly once totaling with $n + 1$ assignments. Thus we conclude that the nullability of all suffixes of $X_1 \cdots X_n$ can be computed in time and space $O(n)$. Also associated with the rule $r$ are $n + 1$ subsets $F(r,i) \ (1 \leq i \leq n + 1)$ of $T \cup \{\varepsilon\}$, each requiring $O_s(|T|)$ space. During execution of lines 8-15, we perform a single unification operation and a single assignment operation per set. By Lemma 8.1 these can be performed in time and space $O(|T|)$ and $O(1)$, respectively. Thus, we conclude that the sets $F(r,i), 1 \leq i \leq n + 1$ can be computed for all suffixes of $X_1 \cdots X_n$ in time and space $O(n|T|)$. Combining these results we conclude that the lemma holds.

**Theorem 8.6.** The INIT method executes in time and space $O(|G||T|)$ with respect to the CFG $G(N,T,P,S)$.

**Proof.** Algorithms for computing the $\text{NULLABLE}$ function of line 1 in time and space $O(|G|)$, and the $\text{FIRST}_1$ function of line 2 in time and space $O(|G||T|)$ are given in [42].
along with formal correctness proofs and complexity analysis. Since each assignment
operation in line 4-5 is performed in constant time, we conclude that traversing all non-
terminals in lines 3-5 is performed in time $O_i(|N|)$. Computation of $\text{beginsRHS}^{-1}$ of line 6 can be done in a single pass over the set of rules $P$. Associated with each nonterminal $A$ is a nonterminal list $B(A)$, that is used to encode the image of $A$ (as a multiset) with respect to the relation $\text{beginsRHS}^{-1}$. Given a rule $r = A \to A\alpha \in P$, we add the nonterminal $A$ to the set $B(A')$ in constant time and space. Since we traverse all the rules in $P$ and thus add at most $|P|$ nonterminals, we conclude that line 6 can be executed in time and space $O(|P|)$. Finally, in lines 7-15 the nullability and $\text{FIRST}_1$ function values are computed for each rule in $P$. By Lemma 8.5 we conclude that lines 7-15 and the entire routine can be executed in time and space $O(|G||T|)$.

Next, we consider the $\text{CREATESTATE}_1$ routine, prove its correctness and determine its complexity.

**Lemma 8.7.** Let $G(N,T,P,S)$ be a CFG, $\gamma \in (N \cup T)^*$, and $[\gamma]_G^1$, an item set. Then

$$I = [A \to \alpha \bullet \beta, y] \in [\gamma]_G^1 \setminus K_\gamma \implies \alpha = \varepsilon$$

**Proof.** Assume towards a contradiction that $\alpha \neq \varepsilon$. By Definition 2.1 we have

$$S \xrightarrow{\delta} \delta Ax \Rightarrow \delta \alpha \beta x = \gamma \beta x, \text{ and } y = 1 : x$$

for some strings $\delta, x$. $\alpha \neq \varepsilon$ implies that we can write $\gamma = \delta \alpha' X$ where $X$ is a symbol, and thus

$$[A \to \alpha' \bullet X \beta, y] \in [\delta \alpha']^1_G$$

According to (a) we have

$$I = [A \to \alpha' X \bullet \beta, y] \in \text{KERNEL}([\delta \alpha']^1_G, X) = K_\gamma$$

which contradicts the assumption that $I$ is a non-kernel item. \hfill \Box

**Lemma 8.8.** Let $G(N,T,P,S)$ be a CFG and let $\gamma \in (N \cup T)^*$ be a symbol string. Then

$$[A \to \bullet \omega, y] \in [\gamma]_G^1 \setminus K_\gamma$$

implies that

$$y \in F(A) \text{ and } A \in N_\gamma$$

**Proof.** $[A \to \bullet \omega, y] \in [\gamma]_G^1$ implies that there exists a rule string $\pi$, and strings $\delta, x$ such that

$$S \xrightarrow{\pi} \delta Ax \xrightarrow{\varepsilon} \delta \omega x = \gamma \omega x \text{ and } y = 1 : x$$

The proof is by induction on the length of $\pi$. If $\pi = \varepsilon$, we have $\delta = \gamma = x = y = \varepsilon$ and $A = S$. From (a) we have $[A \to \bullet \omega, y] \in K_\gamma$ and the lemma trivially holds. Assume then, that $|\pi| > 0$ and, as the induction hypothesis, that the lemma holds for rule string shorter than $\pi$. Since $|\pi| > 0$, Lemma 5.1 implies that there are strings $\delta' \in (N \cup T)^*$ and $x' \in T^*$, rule strings $\pi', \pi'' \in P^*$, and a rule $r = A' \to \alpha' A\beta' \in P$ such that $\pi = \pi' r \pi''$,

$$S \xrightarrow{\pi} \delta' A' x' \xrightarrow{\varepsilon} \delta' A' \beta' x' = \delta A \beta' x', \text{ and } \beta' x' \xrightarrow{\pi''}$$

Thus, we conclude that

$$[A' \to \alpha' \bullet A\beta', 1 : x] \in [\gamma]_G^1, \text{ and } 1 : x \in \text{FIRST}_1(\beta' x')$$


If \([A' \to \alpha' \cdot A\beta', 1 : x']\) is a kernel item, we have

\[(8.3)\]

\[
A \in N^K_\gamma, \quad \text{and} \quad \text{FIRST}_1(\beta'x') \subseteq F_\alpha(A) \subseteq F_0(A) \subseteq F(A)
\]

From (8.1), (8.2) and (8.3) we have \(y = 1 : x \in F(A)\) and, since \(A \in N^K_\gamma\) implies that \(A \in N_\gamma\), we conclude that the lemma holds. On the other hand, if \([A' \to \alpha' \cdot A\beta', 1 : x']\)

is a non-kernel item, Lemma 8.7 implies that \(\alpha' = \varepsilon\). Since \(|\pi'| < |\pi|\) we can apply the induction hypothesis and obtain

\[
1 : x' \in F(A') \quad \text{and} \quad A' \in N_\gamma
\]

Since \(A' \to A\beta' \in P\) we conclude that \((A', A) \in \text{beginsRHS}^{-1}\) and thus \(A \in N_\gamma\) and \(A \to A\beta' \in P_\gamma\). Furthermore, if \(1 : x \in \text{FIRST}_1(\beta') \setminus \{\varepsilon\}\) we have

\[
1 : x \in F_\alpha(A) \subseteq F_0(A) \subseteq F(A)
\]

and the lemma holds. Otherwise, (8.2) implies that \(\beta' \Rightarrow^* \varepsilon\) and \(1 : x = 1 : x'\), which further implies that

\[(8.4)\]

\((A, A') \in \text{containsLA}\)

Lastly, \(1 : x' \in F(A')\) implies that there exists a nonterminal \(A''\) such that \((A', A'') \in \text{containsLA}^*\) and \(1 : x' \in F_0(A'')\). Combining this with (8.4) we get

\[
y = 1 : x' \in F_0(A'') \subseteq F(A)
\]

and thus conclude that the lemma holds. \(\square\)

**Lemma 8.9.** Let \(G(N, T, P, S)\) be a reduced CFG, and let \(\gamma \in (N \cup T)^*\) by a symbol string. Then

\[A \to \omega \in P_\gamma \implies [A \to \cdot \omega, y] \in [\gamma]^1_G\]

for some terminal string \(y\).

**Proof.** By (e), \(A \to \omega \in P_\gamma\) implies that \(A \in N_\gamma\). The proof is by induction on \(n \geq 0\) such that

\[A \in (\text{beginsRHS}^{-1})^n(N^K_\gamma)\]

If \(n = 0\), we have \(A \in N^K_\gamma\) which implies that there exists a kernel item

\[[B \to \alpha \cdot A\beta, z] \in K_\gamma\]

Thus, we have for some symbol string \(\delta\) and terminal string \(x\)

\[S \Rightarrow \delta Bx \Rightarrow \delta\alpha A\beta x = \gamma A\beta x \quad \text{and} \quad z = 1 : x\]

Let \(u\) be a terminal string such that \(\beta \Rightarrow^* u\). Then we have

\[S \Rightarrow \delta\alpha A\beta x \Rightarrow \delta\alpha \omega ux = \gamma \omega ux\]

which implies that

\[[A \to \cdot \omega, 1 : ux] \in [\gamma]^1_G\]

Assume then, that \(n > 0\) and, as the induction hypothesis, that the lemma holds for all \(0 \leq j < n\). Then there exists a nonterminal

\[(8.5)\]

\[A' \in (\text{beginsRHS}^{-1})^{n-1}(N^K_\gamma)\]
such that \((A', A) \in \text{beginsRHS}^{-1}\) which implies that there exists a rule \(A' \rightarrow A\alpha \in P_\gamma\). By applying the induction hypothesis on (8.5) we conclude that for some terminal string \(z\) we have

\[
[A' \rightarrow \bullet A\alpha, z] \in [\gamma]_G^1
\]

which implies that for some symbol string \(\delta\) and terminal string \(x\) we have

\[
S \xrightarrow{\delta} \delta A'x \Rightarrow \delta A\alpha x = \gamma A\alpha x \quad \text{and} \quad z = 1 : x
\]

Let \(u\) be a terminal string such that \(\alpha \Rightarrow u\). Then we have

\[
S \xrightarrow{\delta} \delta Aux \Rightarrow \delta \omega ux = \gamma \omega ux
\]

which implies that

\[
[A \rightarrow \bullet \omega, 1 : ux] \in [\gamma]_G^1
\]

Thus we conclude that the lemma holds.

\[
\square
\]

**Lemma 8.10.** Let \(G(N, T, P, S)\) be a reduced CFG, and let \(\gamma \in (N \cup T)^*\) be a symbol string. Then \(A \in N_\gamma\) implies that for every rule \(r = A \rightarrow \omega \in P\) and terminal string \(y \in F(A)\)

\[
[A \rightarrow \bullet \omega, y] \in [\gamma]_G^1
\]

**Proof.** By (h), \(y \in F(A)\) implies that there exists a nonterminal \(B\) such that

- \(A \text{ contains } L A^* B\) and \(y \in F_0(B)\)

The proof is by induction on \(n \geq 0\) such that

- \(A \text{ contains } L A^n B\) and \(y \in F_0(B)\)

If \(n = 0\), we have \(A = B\) and \(y \in F_0(A)\). If \(y \in F_0(A)\), there exists a kernel item

\[
[B \rightarrow \alpha \bullet A\beta, z] \in K_\gamma, \quad \text{and} \quad y \in \text{First}_1(\beta z)
\]

which implies that there exist a symbol string \(\delta\) and terminal strings \(x, u\) such that

\[
S \xrightarrow{\delta} \delta Bx \Rightarrow \delta A\beta x = \gamma A\beta x, \quad z = 1 : x,
\]

\[
\beta \Rightarrow u, \quad \text{and} \quad y = 1 : uz = 1 : ux
\]

Thus, we have for every rule \(r = A \rightarrow \omega \in P\)

\[
S \xrightarrow{\delta} \delta Aux \Rightarrow \delta \omega ux = \gamma \omega ux, \quad \text{and} \quad y = 1 : ux
\]

which implies that

\[
[A \rightarrow \bullet \omega, y] \in [\gamma]_G^1
\]

Otherwise, \(y \in F_0(A)\), and there exists a rule \(B \rightarrow A\alpha \in P_\gamma\) such that \(y \in \text{First}_1(\alpha) \setminus \{\varepsilon\}\). By Lemma 8.9 there exists a terminal string \(z\) such that

\[
[B \rightarrow \bullet A\alpha, z] \in [\gamma]_G^1
\]

which implies that there exist a symbol string \(\delta\) and terminal strings \(x, u\) such that

\[
S \xrightarrow{\delta} \delta Bx \Rightarrow \delta A\alpha x = \gamma A\alpha x, \quad z = 1 : x,
\]

\[
\alpha \Rightarrow u \neq \varepsilon, \quad \text{and} \quad y = 1 : u
\]
Thus, we have for every rule \( r = A \rightarrow \omega \in P \)

\[
S \Rightarrow^* \delta A u x \Rightarrow^* \delta \omega u x = \gamma \omega u x, \text{ and } y = 1 : u x
\]

which implies that

\[
[A \rightarrow \omega, y] \in [\gamma]_G^n
\]

Assume then, that \( n > 0 \) and, as the induction hypothesis, that the lemma holds for all \( 0 \leq j < n \). Then there exists a nonterminal \( A' \) such that

\[
A \text{ contains } A' \text{ contains } A^n B
\]

which implies (by (h)) that

\[
(8.6) \quad A' \rightarrow A \alpha \in P, \alpha \Rightarrow \varepsilon, \text{ and } y \in F(A')
\]

Since \( A' \in N_\gamma \) we can apply the induction hypothesis, and conclude that

\[
[A' \rightarrow \bullet A \alpha, y] \in [\gamma]_G^n
\]

which implies that there exist a symbol string \( \delta \) and a terminal string \( x \) such that

\[
S \Rightarrow^* \delta A' x \Rightarrow^* \delta A \alpha x = \gamma A \alpha x \Rightarrow^* \gamma A x, \text{ and } y = 1 : x
\]

Thus, we have for every rule \( r = A \rightarrow \omega \in P \)

\[
S \Rightarrow^* \gamma A x \Rightarrow^* \gamma u x, \text{ and } y = 1 : x
\]

which implies that

\[
[A \rightarrow \bullet \omega, y] \in [\gamma]_G^n
\]

Thus we conclude that the lemma holds.

\[ \square \]

**Lemma 8.11.** Let \( G(N, T, P, S) \) be a reduced CFG, and let \( \gamma \in (N \cup T)^* \) be a symbol string. Then

\[
\text{CREATESTATE}_1(K_\gamma) \subseteq [\gamma]_G^n
\]

**Proof.** The proof is by induction on the length of \( \gamma \). We will show that given

\[
q \leftarrow \text{CREATESTATE}_1(K_\gamma)
\]

we have

\[
I = [A \rightarrow \alpha \bullet \beta, y] \in q \implies I \in [\varepsilon]_G^n
\]

If \( \gamma = \varepsilon \), by (a) we have \( K_\gamma = \{[S \rightarrow \bullet \omega, \varepsilon] \in [\varepsilon]_G^n\} \). Thus, \( I \in K_\gamma \) implies that \( I \in [\varepsilon]_G^n \). Otherwise, \( I \notin K_\gamma \). Lines 7-9 of \( \text{CREATESTATE}_1 \) imply that \( A \in N_\gamma \) and \( y \in F(A) \) which further implies, by Lemma 8.10, that \( I \in [\gamma]_G^n \). Assume then, that \( \gamma = \delta X \) for some string \( \delta \) and symbol \( X \) and, as the induction hypothesis, that the lemma holds for all symbol strings shorter than \( \gamma \). If \( I \in K_\gamma \), (a) implies that we can write \( \alpha = \alpha' X \) for some string \( \alpha' \) and

\[
[A \rightarrow \alpha' \bullet X \beta, y] \in [\delta]_G^n
\]

Thus we have for some symbol string \( \delta' \) and terminal string \( x \)

\[
S \Rightarrow^* \delta' A x \Rightarrow^* \delta' \alpha' X \beta x = \delta' \beta x = \gamma \beta x, \text{ and } y = 1 : x
\]

which implies that \( I \in [\gamma]_G^n \). On the other hand, if \( I \notin K_\gamma \), Lines 7-9 of \( \text{CREATESTATE}_1 \) imply that \( A \in N_\gamma \) and \( y \in F(A) \) which further implies, by Lemma 8.10, that \( I \in [\gamma]_G^n \). \( \square \)
Lemma 8.12. Let \( G(N,T,P,S) \) be a reduced CFG, and let \( \gamma \in (N \cup T)^* \) be a symbol string. Then

\[
[\gamma]_G^1 \subseteq \text{createState}_1(K_\gamma)
\]

Proof. Let \( I = [A \rightarrow \alpha \bullet \beta, y] \in [\gamma]_G^1 \). If \( I \in K_\gamma \), by line 6 of \( \text{createState}_1 \) we have

\[
I \in \text{createState}_1(K_\gamma)
\]

Assume then, that \( I \notin K_\gamma \). Lemma 8.7 implies that \( \alpha = \varepsilon \). Thus, by Lemma 8.8 we conclude that \( A \in N_\gamma \) and \( y \in F(A) \). Lines 7-9 of \( \text{createState}_1 \) then imply that

\[
I \in \text{createState}_1(K_\gamma)
\]

and we conclude that the lemma holds. \( \Box \)

Lemma 8.13. Let \( G(N,T,P,S) \) be a reduced CFG, and let \( \gamma \in (N \cup T)^* \) be a symbol string. Following execution of the Init routine, the call

\[
q \leftarrow \text{createState}_1(K_\gamma)
\]

computes \( q = [\gamma]_G^1 \) in time and space \( O(|K_\gamma||T| + |P||T|) \) and \( |q| \leq |K_\gamma| + |P| \).

Proof. \( [\gamma]_G^1 = \text{createState}_1(K_\gamma) \) follows immediately from Lemmas 8.11 and 8.12. We proceed by proving the bounds on the time and space complexity of the computation.

First, recall that LR(1) item sets are represented as linked lists so that addition of a single item set, as well as set initialization (of line 6) occur in constant time and space.

Computation of \( N^K \gamma \) and \( F_a \) of line 1 can be done in a single pass over the item sets contained in \( K_\gamma \). Associated with each nonterminal \( A \) is a set \( F_a(A) \) that will be used to store the image of \( A \) with respect to the function \( F_a \). We represent the set \( N^K \gamma \) as a linked list of nonterminals with a dedicated auxiliary array of size \( O_s(|N|) \) which values are already initialized to 0. The set is initialized with an empty list in constant time. Given a item set \( I = [A \rightarrow \alpha \bullet \beta, \Theta] \), we can test if a nonterminal immediately follows \( \bullet \) and add it to \( N^K \gamma \) if it is not already contained there, in constant time. Furthermore, we can compute the values contributed to \( F_a(A) \) by \( I \) by obtaining \( \Phi = F(r,|\alpha|+1) \) in constant time and space, determining if \( \beta \) is nullable in constant time by consulting \( N(r,|\alpha|+1) \), computing \( \Phi \leftarrow \Phi \cup \Theta \) if indeed \( \beta \) is nullable in time and space \( O(|T|) \), and computing \( F_a(A) \leftarrow F_a(A) \cup \Phi \) in time and space \( O(|T|) \). After traversing all item sets in \( K_\gamma \), the auxiliary array of \( N^K \gamma \) is reset (in favor of future executions of the routine) by assigning 0 to every nonterminal in \( N^K \gamma \) in time \( O_t(|N^K \gamma|) \). Since \( G \) is reduced we have \( |N| \leq |P| \). As we also have \( |N^K | < |K_\gamma| \) and, in the worst case, for each nonterminal \( A \in N^K \gamma \), \( O(|F_a(A)|) = O(|T|) \), we conclude that line 1 can be executed in time \( O_t(|K_\gamma||T|) \) and space \( O_s(|P| + |K_\gamma||T|) \).

Since \( |\text{beginsRHS}^{-1}| \leq |P| \) and \( N^K \gamma \subseteq N \), Lemma 8.3 implies that \( N_\gamma \) of line 2 can be computed in time and space \( O(max\{|P|,|N|\}) \). Due to the fact the \( G \) is reduced, we have \( |N| \leq |P| \) and thus conclude that line 2 executes in time and space \( O(|P|) \).

Computation of \( \text{containsLA} \) and \( F_b \) of line 3 can be done in a single pass over the set of rules \( P_\gamma \). This is accomplished by traversing the rules in which LHS nonterminals of \( N_\gamma \) appear, as done in lines 7-8 of the algorithm. Associated with each nonterminal \( A \) is an empty nonterminal list \( C(A) \), that is used to encode the image of \( A \) (as a multiset) with respect to the relation \( \text{containsLA} \). Also associated with each nonterminal is the set \( F_b(A) \) that is used to store the image of \( A \) with respect to the function \( F_b \). Given a rule \( r = B \rightarrow A\alpha \in \xi(B) \), we can compute the nonterminals contributed by \( r \) to the set \( \text{containsLA}(A) \) by determining if \( \alpha \) is nullable in constant time by consulting
Furthermore, we can compute the terminals contributed by \( r \) to the set \( F_b(A) \) by computing \( F_b(A) \leftarrow F_b(A) \cup \{F(r,2) \setminus \{\} \} \) in time and space \( O(|T|) \). Since, in the worst case, we traverse all the rules in \( P \) and add \(|T| + 1\) symbols to \(|P|\) distinct \( F_b \) sets, and at most \(|P|\) nonterminals are added to \( C \) sets, we conclude that line 3 can be executed in time \( O(|P||T|) \) and space \( O_s(|P||T|) \).

The computation of \( F_0 \) in line 4 is performed by traversing all the nonterminals in \( N_r \) and for each nonterminal \( A \) computing \( F_0(A) \leftarrow F_0(A) \cup F_b(A) \) in time and space \( O(|T|) \). As before, the set \( F_0(A) \) is associated with the nonterminal \( A \) and stores the image of \( A \) with respect to \( F_0 \). Once \( F_0(A) \) is computed, the sets \( F_0(A) \) and \( F_b(A) \) are no longer needed, and can be cleared by computing \( F_n(A) \leftarrow F_b(A) \leftarrow \{\} \) in constant time and space. Since \( G \) is reduced we have \(|N| \leq |P|\), and thus conclude that line 4 can be executed in time and space \( O(|P||T|) \).

Since \(|\text{containsLA}| \leq |P|\), and for subsets of \( T \cup \{\varepsilon\} \) assignment operations require constant time and space, and the time and space required by union operations (with auxiliary array) are bounded by \( O(|T|) \), Lemma 8.4 implies that function \( F \) of line 5 can be computed in time \( O_t(|N| + |P||T|) \) and space \( O_s(|P| + |N||T|) \). Once function \( F \) is computed, values of \( C \) sets associated with nonterminals are no longer required and can be reset. This is done by traversing each nonterminal \( A \in N_r \), and assigning \( C(A) \) with an empty list. Since \( G \) is reduced, \(|N| < |P|\), and we conclude that line 5 is executed in time \( O_t(|P||T|) \) and space \( O_s(|P||T|) \).

Finally, in lines 7-9, at most \(|P|\) item sets are created and added to \( q \). This is done in time \( O_t(|P|) \) and space \( O_s(|K_r| + |P|) \).

Combining these results, we conclude that \(|q| \leq |K_r| + |P|\) and that \( q \) can be computed in time and space \( O(|K_r||T| + |P||T|) \) as stated by the lemma. \( \square \)

**Lemma 8.14.** Let \( G(N,T,P,S) \) be a reduced CFG, \( \gamma \in (N \cup T)^* \) a symbol string,

\[
K'_\gamma = \begin{cases}
\{[S \rightarrow \bullet; \{\varepsilon\}][S \rightarrow \omega \in P]\} & \gamma = \varepsilon \\
\mathrm{KERNEL}(\text{CreateState}_1(K'_\delta), X) & \gamma = \delta X
\end{cases}
\]

and let \( m \geq 0 \) denote the maximal number of occurrences of any symbol \( X \in N \cup T \) in the RHS of some rule in \( P \). Then \(|K'_\gamma| \leq \min\{|G|, (1 + m)|P|\} \).

**Proof.** Let the core of an item set \( [A \rightarrow \alpha \bullet \beta, \Theta] \) be the component \( A \rightarrow \alpha \bullet \beta \), and let \( q_\gamma = \text{CreateState}_1(K'_\gamma) \)

We first show by induction on the length of \( \gamma \) that at most \(|P|\) distinct cores can appear in two distinct item sets in \( q_\gamma \) and none can appear in more than two distinct item sets of \( q_\gamma \). If \( \gamma = \varepsilon \) we have \(|K'_\varepsilon| \leq |P|\) and each core appears only in one item set of \( K'_\varepsilon \subseteq q_\varepsilon \).

By lines 7-9 of \( \text{CreateState}_1 \), we conclude that at most \(|P|\) non-kernel item sets with distinct cores are added to \( q_\varepsilon \) and thus we conclude that the induction’s statement holds. Assume then, that \( \gamma = \delta X \) for some symbol \( X \in N \cup T \), and that the induction hypothesis holds for strings shorter that \( \gamma \). Since the induction hypothesis applies to \( q_\delta \), we conclude that at most \(|P|\) distinct cores appear in two distinct item sets in \( \mathrm{KERNEL}(q_\delta, X) \) and none appear in more than two distinct item sets. Furthermore, in the core of each item set, the \( \bullet \) is preceded by the symbol \( X \). By lines 7-9 of \( \text{CreateState}_1 \), we conclude that at most \(|P|\) non-kernel item sets with distinct cores are added to \( q_\gamma \). Since no symbol precedes the \( \bullet \) in any of these item sets, we conclude that the induction statement holds.

We now show that \(|K'_\gamma| \leq \min\{|G|, (1 + m)|P|\} \). If \( \gamma = \varepsilon \) we have

\[
|K'_\varepsilon| \leq |P| \leq \min\{|G|, (1 + m)|P|\}
\]
as stated by the lemma. Otherwise, \( \gamma \neq \varepsilon \), and we can write \( \gamma = \delta X \) for some symbol \( X \in N \cup T \). Assume towards a contradiction that \( |K'_\gamma| = l > (1 + m)|P| \). This implies that \( q_\delta \) contains exactly \( l \) item sets in which the \( \bullet \) precedes the symbol \( X \). Since at most \( |P| \) cores appear in two distinct item sets we conclude that there are at least \( l - |P| > m|P| \) item sets in \( q_\delta \) with distinct cores where the \( \bullet \) precedes the symbol \( X \). This implies that there is at least one rule appearing in more than \( m \) item sets of distinct cores, in which the \( \bullet \) precedes the symbol \( X \). Since this contradicts the assumption that no symbol in \( N \cup T \) appears more than \( m \) times in the RHS of a rule, we conclude that \( |K'_\gamma| \leq (1 + m)|P| \). On the other hand \( \gamma \neq \varepsilon \) implies that in all kernel items, \( X \) precedes the \( \bullet \). Thus, there are at most \( |G| - |P| \) distinct cores among item sets of \( K'_\gamma \). In the worst case, \( |P| \) distinct item sets in \( K'_\gamma \) share their core with other \( |P| \) item sets in \( K'_\gamma \), which implies that \( |K'_\gamma| = |G| \). Thus we conclude that

\[
|K'_\gamma| \leq \min\{|G|, (1 + m)|P|\}
\]

as stated by the lemma.

**Theorem 8.15.** Let \( G(N, T, P, S) \) be a reduced CFG, \( \gamma \in (N \cup T)^* \) a symbol string,

\[
K'_\gamma = \begin{cases} 
\{[S \rightarrow \bullet \omega, \{\varepsilon\}] | S \rightarrow \omega \in P\} & \gamma = \varepsilon \\
\text{KERNEL(CreateState}_1(K'_\delta), X) & \gamma = \delta X
\end{cases}
\]

\( m \geq 0 \) the maximal number of occurrences of any symbol \( X \in N \cup T \) in the RHS of some rule in \( P \), \( n > 0 \), and \( l = \text{min}\{|G|, (1 + m)|P|\} \). Then following execution of the \text{INIT} routine, the call

\[
q \leftarrow \text{CreateState}_1(K'_\gamma)
\]

(a) computes \( q = [\gamma]^l_G \)
(b) in time and space \( O(l|T|) \),
(c) \( |q| \leq l \), and
(d) \( n \cdot O_s(q) = O(n \cdot l|T| + |G|) \)

**Proof.** Statements (a), (b) and (c) follow immediately from Lemmas 8.13 and 8.14. From (c) we have \( |q| \leq l \). Each item set is of size \( O_s(1) \), and contains references to a lookahead set of size \( O_s(|T|) \) and a rule skeleton. Since each rule skeleton is represented as a single object, item sets \( q \) can reference at most \( |P| \) rules which combined size does not exceed \( |G| \). Thus, we conclude that \( O_s(q) = O(l|T| + |G|) \), and that \( n \cdot O_s(q) = O(n \cdot l|T| + |G|) \) as stated in (d).

### 8.4 Deterministic AMG(1) parsing complexity

In this section we state bounds on the time and space complexities of adaptive multi-pass parses by deterministic adaptive multi-pass parsers utilizing a single lookahead symbol. We first state complexity bounds based on the number of Shift and Reduce actions performed during the parse with respect to the number of grammar modifications, maximal symbol stack size, maximal skeletal grammar size, and the complexity of the grammar’s adaptation and annotation functions (Lemma 8.20). We then combine these bounds with the complexity bounds stated in Chapter 7 on complete derivations of deterministic AMGs. We show that given a fixed bound on the number of grammar modifications and skeletal CFG size, a deterministic parser utilizing a single lookahead symbol parses a terminal string \( x \) in time

\[
O(n(O_1(\Delta) + O_1(f)))
\]
and space

\[ O(n \cdot O_s(f) + O_s(\Delta)) \]

where all invocations of the parser’s adaptation function \( \Delta \) execute in time \( O_t(\Delta) \) and space \( O_s(\Delta) \), all invocations of annotation functions execute in time \( O_t(f) \) and space \( O_s(f) \), and \( n \geq |x| \) is the number of terminals shifted by the parser including the terminals of \( x \) and all terminals pushed back to the input stream due to Reduce actions by multi-pass rules (Theorem 8.21 and Corollary 8.22).

We begin by stating bounds on the time and space complexities of the parser’s subroutines: \textsc{UpdateRules}, \textsc{Adapt}, \textsc{Action}, and \textsc{Goto}.

**Lemma 8.16.** Let \( G(N, T, R_0, S, \Delta, k) \) be an AMG, and \( M \) its AMP. The call

\[ \text{UpdateRules}(R_a, R_r) \]

executes in time and space \( O(|R| + |R_a| + |R_r|) \) where \( R \) denotes the parser’s Rules set prior to the call.

**Proof.** The modification of the Rules set clearly executes in \( O(|R_a| + |R_r|) \) time and space since it supports constant time and space membership tests (utilizing the \( R(r) \) flag) as well as constant time and space rule addition and removal operations. Following the modification of the Rules set, a unique integer is assigned to each nonterminal appearing in Rules. Clearly, this can be accomplished in \( O(|R| + |R_a|) \) time and space. \( \square \)

**Lemma 8.17.** Let \( G(N, T, R_0, S, \Delta, 1) \) be an AMG, \( M \) its AMP, and \((\phi, \gamma)(y, R)\) a configuration of \( M \). If \textsc{Adapt} is successfully invoked in line 7 of the main parsing loop, it executes in time and space \( O(|M^R_\xi||T||\gamma|) \).

**Proof.** According to Theorem 8.6, the implicit call to the \textsc{Init} method required after the modification of the Rules set, executes in time and space \( O(|M^R| |T|) \). In line 1 of the \textsc{Adapt} method, the kernel \( \{[S' \rightarrow \bullet S$, $\varepsilon]\} \) can be constructed in constant time and space, and by Theorem 8.15, the call to \textsc{CreateState} \( k \) executes in time and space \( O(|M^R_\xi||T|) \). The initialization of the state stack in line 2, also executes in constant time and space. The loop in lines 3-6 performs exactly \( |\gamma| \) iterations. In each iteration, a state is created in time and space \( O(|M^R_\xi||T|) \), and pushed onto the parse stack in constant time. Theorem 8.15 also implies that the space complexity of each state is bounded by \( O(|M^R_\xi||T|) \). Thus, we conclude that the routine executes in time and space \( O(|M^R_\xi||T||\gamma|) \) as stated by the lemma. \( \square \)

**Lemma 8.18.** Let \( G(N, T, R_0, S, \Delta, 1) \) be an AMG, \( M \) its AMP, \((S, \gamma)(y, R)\) a configuration of \( M \), \( q = [\gamma]^{\xi(R)}_\xi \), and \( t \in T \cup \{\$\} \). Then \textsc{Action}(q, t) executes in time \( O(|M^R_\xi| + |R||T|) \) and space \( O(|R|) \).

**Proof.** As previously described, the state \( q \) is organized as a list of items sets of the form \([A \rightarrow \alpha \bullet \beta, \Theta] \), each represented as a triplet \((r, |\alpha|, \Theta) \) where \( \Theta \subseteq T \cup \{\varepsilon, \$\} \) and \( r \) is a reference to the rule skeleton \( A \rightarrow \alpha \beta \). Let \( A \rightarrow \bullet \beta \) be the core of \([A \rightarrow \alpha \bullet \beta, \Theta] \). We first show by induction on the length of \( \gamma \) that no two item sets in \( q \) share the same core.

If \( \gamma = \varepsilon \), \( K_\varepsilon = \{[S' \rightarrow \bullet SS, \{\varepsilon]\}\} \) and by lines 7-9 of \textsc{CreateState} \( k \) each non-kernel item set consists of a distinct rule. Furthermore, since \( S' \) does not appear in the RHS of any rule in \( \xi(R) \), we conclude that \( S' \notin N_\varepsilon \) and therefore, the rule \( S' \rightarrow SS \) does not appear in any non-kernel item set. Thus, we conclude that no two item sets in \( q \) share the same core. Assume then, that \( \gamma = \delta X \) for some symbol \( X \in N \cup T \cup \{\$\} \), and that the induction hypothesis holds for strings shorter that \( \gamma \). Since every kernel item set in
q is obtained from an item set of $[\delta]^k_{\xi(R)}$ by advancing the $\bullet$ symbol one symbol to the right, we conclude that no two kernel item sets in q share the same core. By lines 7-9 of CREATESTATE1 each non-kernel item set of q consists of a distinct rule and thus a distinct core. Since in all kernel item sets the symbol X precedes the $\bullet$ symbol, and in all non-kernel item sets no symbol precedes the $\bullet$ symbol, we conclude that no two item sets in q share the same core.

We proceed by describing an implementation of the ACTION routine which complexity is bound as stated in the lemma. The ACTION method initializes a result set in constant time and space and then proceeds to infer actions by traversing each item set in q and performing the following tests (as described in Section 2.5). If $t = 1 : \beta$ a Shift action is added to the result set, and a flag is set to prevent further Shift actions from being added (both test and insertion execute in constant time and space). Reduce and Accept actions are added to the result set if $\beta = \varepsilon$, $t \in \Theta$, and $A \neq S'$ or $A = S'$, respectively. Except for the test $t \in \Theta$ which executes in time $O(|\Theta|) \leq O(|T|)$, all other tests can clearly be executed in constant time and space. Theorem 8.15 implies that $|q| \leq |M^R_\xi|$. However, since no two item sets in q share the same core, there are at most $|R|$ item sets in q where $\beta = \varepsilon$. As only for these item sets the test $t \in \Theta$ should be performed, we conclude that the entire routine executes in time $O(|M^R_\xi| + |R||T|)$. Since an LR action can be encoded in $O(1)$ space, and in the worst case, at most one Shift action and |R| Reduce actions are added to the result set, we conclude that the routine executes in space $O(|R|)$. \hfill $\Box$

Note that the implementation described in the above proof, guarantees that no LR action appears twice in the set of actions returned by the ACCEPT routine.

**Lemma 8.19.** Let $G(N, T, R_0, S, \Delta, 1)$ be an AMG, M its AMP, $(S, \gamma|y, R)$ a configuration of M, $q = [\gamma]^k_{\xi(R)}$, and $X \in N \cup T \cup \{\$\}$. Then GOTO$(q, X)$ executes in time and space $O(|M^R_\xi||T|)$.

**Proof.** As listed in Figure 5.2, the GOTO routine first obtains the set of kernel item sets

$$K = Kernel(q, X)$$

and then computes the complete state $[\gamma X]^k_{\xi(R)}$ by calling

CREATESTATE1$(K)$

The Kernel routine first initializes a result set in constant time and space. It then traverses each items set in q, and for every items set of the form $[A \rightarrow \alpha \bullet X \beta, \Theta]$ it adds to the result set the items set $[A \rightarrow \alpha X \bullet \beta, \Theta]$. Since $|q| \leq |M^R_\xi|$ (by Theorem 8.15) and since each items set can be created in constant time and space (by reusing the reference to the lookahead set of the source items set), we conclude that the routine executes in time and space $O(|M^R_\xi|)$. Theorem 8.15 further implies that the call CREATESTATE1$(K)$ executes in time and space $O(|M^R_\xi||T|)$. Thus, we conclude that GOTO$(q, X)$ executes in time and space $O(|M^R_\xi||T|)$ as stated by the lemma. \hfill $\Box$

The following lemma states complexity bounds on adaptive multi-pass parses based on the number of Shift and Reduce actions performed with respect to the number of grammar modifications, maximal symbol stack size, maximal skeletal grammar size, and the complexity of the grammar’s adaptation and annotation functions.

**Lemma 8.20.** Let $G(N, T, R_0, S, \Delta, 1)$ be an AMG, M its AMP, and $\pi'$ an adaptive multi-pass parse of x in M involving $s \geq 1$ Shift actions, $r \geq 1$ Reduce actions, and $q \geq 0$ rule set modifications, such that $l \geq 2$ is the maximal symbol stack length during the parse,
$p \geq 0$ is the maximal skeletal CFG size underlying Rules during the parse, all invocations of $\Delta$ execute in time $O_{t}(\Delta)$ and space $O_s(\Delta)$, and all invocations of annotation functions execute in time $O_{t}(f)$ and space $O_s(f)$. Then the parse of $x$ in $M$ executes in time

$$O(sp + r(p + O_{t}(\Delta) + O_{t}(f)) + glp)$$

and space

$$O(l(p + O_{s}(f)) + O_{s}(\Delta))$$

Proof. During the parse of $x$ in $M$, the parser initializes once (lines 1-4), and executes the main parse loop exactly $|\pi'| + 1 = r + s + 1$ times. Throughout these iterations, lines 6,9-10 are executed $|\pi'| + 1$ times, lines 7-8 are executed $q$ times, lines 11-15 are executed $s$ times, lines 16-26 and 31 are executed $r$ times, lines 27-30 are executed at most $s + r$ times (since every terminal pushed back to input stream is later shifted in the parse and the reduced rule’s LHS nonterminal may be associated with a null annotation string), and lines 32-33 are executed once. Since lines 1-3 execute in constant time and space, and (by Lemma 8.16) line 4 executes in time and space $O(|R_0|) \leq O(p)$, we conclude that the parser initializes in time and space $O(p)$. Let $R = Rules$. In the main parse loop, lines 6,8 and 9 execute in constant time and space, and by Lemma 8.17 line 7 executes in time and space $O(lp|T|)$. Furthermore, since $|R| \leq |M_{P}^{R}| \leq p$, Lemma 8.18 implies that line 10 executes in time $O(p|T|)$ and space $O(p)$. In case of a Shift action, all the statements in lines 11-15 except for the invocation of the GOTO routine, execute in constant time and space. By Lemma 8.19 the latter routine executes in time and space $O(p|T|)$. In case of a Reduce action, lines 16-18 are executed in time $O_{t}(\Delta)$ and space $O_s(\Delta)$. Note that $\Delta$ can be implemented to process the symbol and annotation stacks directly such that the explicit construction of $\rho$ in line 17 can be skipped. Similarly, lines 19-20 are executed in time $O_{t}(f)$ and space $O_s(f)$. The removal of $n$ entries from all stacks in line 21 executes in time $O(n) \leq O(|M_{P}^{R}|) \leq O(p)$, and all statements in lines 22-26 except for the invocation of the GOTO routine in line 23 (which executes in time and space $O(p|T|)$) execute in constant time and space. Since $|R_a| + |R_c| \leq |R_a| + |R| \leq 2p$, Lemma 8.16 implies that the invocation of UPDATERULES in line 31 executes in time and space $O(p)$. If the reduced rule is a multi-pass rule, each iteration of the loop in lines 27-30 executes in constant time and space. Lastly, in case of an Accept action, lines 32-33 execute in constant time and space. Combining the above results (and since $|T|$ is a constant in $M$) we conclude that the parse of $x$ in $M$ executes in time

$$O(|\pi'|p|T| + glp|T| + sp|T| + r(O_{t}(\Delta) + p + O_{t}(f) + p|T|)) =$$

$$O(sp + r(p + O_{t}(\Delta) + O_{t}(f)) + glp)$$

and since the only space retained between iterations is stored in the Rules set and the parse stacks, we conclude that the parse of $x$ in $M$ executes in space

$$O(l(p|T| + O_s(f)) + O_s(\Delta)) = O(l(p + O_s(f)) + O_s(\Delta))$$

By combining the bounds of Lemma 8.20 with the complexity bounds stated by Theorem 7.15 on complete derivations of deterministic AMGs we obtain the following theorem.

**Theorem 8.21.** Let $G(N,T,R_0,S,\Delta,1)$ be a deterministic AMG, $M$ its AMP, and $p, g \in \mathbb{N}$ constants. If $\pi'$ is an adaptive multi-pass parse of $x_1$ in $M$ with $m > 0$ passes and at
most $g$ grammar modifications, and throughout the parse the size of the skeletal CFG underlying Rules is bound by $p$, then $M$ parses $x_1$ in time 

$$O((|x_1| + |x_2| + |x_3| + \cdots + |x_m|)(O_t(\Delta) + O_t(f)))$$

and space

$$O((|x_1| + |x_2| + |x_3| + \cdots + |x_m|) \cdot O_s(f) + O_s(\Delta))$$

where for all $2 \leq i \leq m$, $x_i$ denotes the terminal string shifted back to the input stream by the $i$-th multi-pass Reduce action, all invocations of $\Delta$ execute in time $O_t(\Delta)$ and space $O_s(\Delta)$, and all invocations of annotation functions execute in time $O_t(f)$ and space $O_s(f)$.

**Proof.** By Lemmas 5.6 and 5.5 there exists a complete derivation $d$ of $x$ in $G$ by rule string $\pi$ such that $\tau(\pi') = \pi^R$ and the maximal length of the symbol stack throughout the parse is bound by the space complexity of $d$. Furthermore, by considering $d$ as an AMG derivation with $m$ passes where $w_2, w_3, \ldots, w_m$ are the rewritten annotated strings, $w_i$ is a suffix of $x_{m+2-i}$ for all $2 \leq i \leq m$. Let $n = |x_1| + |x_2| + \cdots + |x_m|$. By Theorem 7.15, the time and space complexities of $d$ are linear in $|x_1| + |w_2| + |w_3| + \cdots + |w_m|$ and are thus linear in $n$. This in turn implies that there exist a constant $c$ such that $O_t(d) \leq cn + c$ and $O_s(d) \leq cn + c$. Since $M$ executes exactly $|\pi| = O_t(d)$ Reduce actions, and $n$ Shift actions, Lemma 8.20 implies that the parse of $x_1$ in $M$ executes in time

$$O(np + (cn + c)(p + O_t(\Delta) + O_t(f)) + (cn + c)gp) = O(n(O_t(\Delta) + O_t(f)))$$

and space

$$O((cn + c)(p + O_s(f)) + O_s(\Delta)) = O(n \cdot O_s(f) + O_s(\Delta))$$

Since every successful parse ends with an empty input string we obtain the following conclusion.

**Corollary 8.22.** Let $G(N, T, R_0, S, \Delta, 1)$ be a deterministic AMG, $M$ its AMP, and $p, g, s \in \mathbb{N}$ constants. If $\pi'$ is an adaptive multi-pass parse of $x$ in $M$ with at most $g$ grammar modifications, and throughout the parse the size of the skeletal CFG underlying Rules is bound by $p$, then $M$ parses $x$ in time

$$O(n(O_t(\Delta) + O_t(f)))$$

and space

$$O(n \cdot O_s(f) + O_s(\Delta))$$

where all invocations of the parser’s adaptation function $\Delta$ execute in time $O_t(\Delta)$ and space $O_s(\Delta)$, all invocations of annotation functions execute in time $O_t(f)$ and space $O_s(f)$, and $n \geq |x|$ is the number of terminals shifted by the parser.
Chapter 9

Conclusions and Future Work

Although several adaptive grammar formalisms have been proposed since the early 1970s, most fail to provide the mechanisms required for modeling forward references and syntax macro expansions. They are therefore ill-suited for fully describing the syntax of syntactically extensible programming languages. Furthermore, the over-permissive adaptive nature of these formalisms prevents the development of efficient, practical parsers.

In this thesis we have proposed an adaptive formalism which is more restrictive yet powerful enough to handle constructs commonly found in extensible programming languages. It suggests a multi-pass approach that goes hand in hand with the adaptive paradigm, and elegantly solves the two aforementioned issues. Furthermore, taking advantage of the restrictions, we have developed an efficient LR(1)-based parsing algorithm that is amenable to practical implementation and handles both incremental and decremental changes in the grammar gracefully.

In accordance with the stated goal of this thesis, our work concentrated on ensuring that the suggested formalism and parser would indeed be applicable for practical use. First, we have formulated tight bounds on the time and space complexities of CFG rightmost derivations. Since our bounds are independent of the size of the grammar, we are able to show that the very concept of encoding context-sensitive constructs by growing and adapting a CFG is computationally practical. The increase in grammar size is not likely to result with the exponential growth of derivation (and parsing) complexity as implied by the previously known bounds. Second, our adaptive formalism was designed to facilitate the development of an efficient LR(k)-based parser by incorporating k-length lookahead strings in its derivation relation. Third, our parsing algorithm was designed to accommodate frequent grammar modifications by employing a lazy approach to parse table generation. Fourth, we have developed an efficient algorithm for computing LR(1) item sets, which significantly improves upon previously known algorithms.

Future work on the AMG model can proceed in both theoretical and empirical tracks. On the theoretical track, further study of the formal properties of AMGs is in order. An interesting direction would be to investigate special cases of AMGs and their relation to other grammar formalisms. For instance, it can be shown that the model retains its computational strength even if its rule sets are limited to regular grammars and multi-pass rules are not allowed (see proof of Theorem 4.3). It can also be shown that its computational strength is retained even if the adaptation function is total. Furthermore, certain non-context-free languages, such as \( \{ a^n b^n c^n | n \geq 0 \} \), can be specified without any grammar modifications. Other directions include the investigation of closure operations on AMGs and the existence of a pumping lemma. Finally, it should be possible to derive tighter bounds on the time and space complexities of AMG derivations, especially with regards to less restricted forms of AMG derivations.
On the empirical track, an immediate priority for continued research on AMGs is the implementation of an adaptive parser based on the AMGPARSE algorithm. Such an implementation would enable empirical study of the parser’s performance when parsing real-world extensible programming languages. It would further facilitate the development and evaluation of various optimization schemes:

- A lazy and incremental algorithm for computing nullable nonterminals and $\text{FIRST}_1$ sets.

- A cache persisting complete LR(1) item sets based on their kernel signature and grammar revision. As an alternative to generating a new item set from a given kernel, an existing item set can be looked up in the cache and verified against the aggregation of grammatical modifications that have occurred since its creation.

- Marking (as part of language specification) areas of the grammar that are immutable, and allowing the parser to retain and reuse certain (partial) item sets accordingly.

- Transparent, parser-initiated grammar modifications aimed to minimize the number of reduce actions needed for completing a parse (in accordance with the grammar properties underlying our bounds on derivational complexity).

Another area for further research is related to the design and implementation of a system of disambiguation rules suitable for the adaptive paradigm. Many of the prevailing techniques (such as those provided by YACC [24]) rely on a fixed grammar and would not necessarily scale well in face of a mutable set of production rules. Similarly, the applicability of existing error-recovery schemes to adaptive parsing should be studied and may lead to the development of more suitable techniques.
Bibliography


We prove the correctness of the algorithm and that, when given the syntax-free context-free language $G(N,T,P,S)$ and $0 \leq m \leq |P|$, the algorithm computes the LR(1) state in time and space complexity $O(\min\{|G|, m|P|, |T|\})$. A further result is a bound on the time and space complexity of the rightmost derivations in syntax-free context-free languages.

We identify three properties of context-free languages on which the bounds depend on the length of the symbols at the beginning and at the end of the derivation.

For derivations of terminal strings of a single variable, we prove that the bound is tight by presenting a series of context-free languages in which the rightmost derivations exactly match the bound.

The fact that the properties of the context-free languages on which the bound is based do not depend on the size of the context-free language (number of rules, total derivation length, or length) is a significant advantage of the context-free parser-adaptive automaton for the definition of context-dependent and context-independent languages.
In this thesis, we rely on forward reference to significantly reduce the number of terminal groups. We build a dependency graph representing the content of the repository and use it.

Traditional deterministic parsers handle high efficiency with these steps. They are equivalent to a large number of backtracking for parsing strings.

In this thesis, we define a class of adaptive parsers that can be analyzed by a dependency parser. By placing a fixed bound on the number of backtracking steps, we can analyze parsers that are not adapted to the full syntax of the language, which is sure to be too slow for backtracking dependency parsing.

It is known that deterministic parsers are equivalent to a single parsing step and do not use the power of backtracking. However, in general, they are not enough to handle the full power of the language.

In this thesis, we propose a model where the dependency parser is combined with a general backtracking and can be used to analyze adaptive parsers in the context of the language.

We prove formally the correctness of these claims and can be used to analyze adaptive parsers in the context of the language.
הקשר

למען להראות את כל התחבירים של קוד דיפקстер במהלך הפקה, מן הרחבת התוכן, המתחבטים בקשר לשתי הדיקוקה של כתבי-ה cerco-ה halkהוורפ. {}

למשל, אם岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐岐lige...
המחקר נעשה בהנחיית פרופ' מיכאל קמינסקי ופרופ'/ה' רון פינטר.

אני מודה לטכניון על העזרה הכספית והדידית בהשתלמותי.

אני מודה לוכנינו על הועדה הנוכשת המדורתית בשיתולים.
ניתוח תחבירי אדפטיבי רב-מעברי

חיבור על מחקר

לשמ מילוי חלקי של הדרישות לקבלי התוחור
מגייסר لمдушם במועד המחשב

אדם כרמי

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

פברואר 2010
ניתוח תחבירי אדפטיבי רב-מעבר

אדם קרמי