

**ADAPTABLE PARSING EXPRESSION
GRAMMARS**

LEONARDO VIEIRA DOS SANTOS REIS

ADAPTABLE PARSING EXPRESSION
GRAMMARS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

ORIENTADOR: ROBERTO DA SILVA BIGONHA
COORIENTADOR: VLADIMIR OLIVEIRA DI IORIO

Belo Horizonte
Novembro de 2014

LEONARDO VIEIRA DOS SANTOS REIS

**ADAPTABLE PARSING EXPRESSION
GRAMMARS**

Thesis presented to the Graduate Program
in Computer Science of the Universidade
Federal de Minas Gerais in partial fulfill-
ment of the requirements for the degree of
Doctor in Computer Science.

ADVISOR: ROBERTO DA SILVA BIGONHA
CO-ADVISOR: VLADIMIR OLIVEIRA DI IORIO

Belo Horizonte
November 2014

© 2014, Leonardo Vieira dos Santos Reis.
Todos os direitos reservados.

Reis, Leonardo Vieira dos Santos

R375a Adaptable Parsing Expression Grammars / Leonardo
Vieira dos Santos Reis. — Belo Horizonte, 2014

xxvi, 90 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas
Gerais — Departamento de Ciência da Computação

Orientador: Roberto da Silva Bigonha
Coorientador: Vladimir Oliveira Di Iorio

1. Computação – Teses. 2. Linguagens de
Programação (Computadores) - Sintaxe – Teses.
3. Compiladores (Computadores) – Teses.
I. Orientador. II. Coorientador. III. Título.

CDU 519.6*33(043)



UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

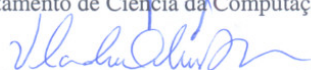
FOLHA DE APROVAÇÃO


Adaptable parsing expression grammars

LEONARDO VIEIRA DOS SANTOS REIS

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:


PROF. ROBERTO DA SILVA BIGONHA - Orientador
Departamento de Ciência da Computação - UFMG


PROF. VLADIMIR OLIVEIRA DI IORIO - Coorientador
Departamento de Informática - UFV


PROF. FERNANDO MAGNO QUINTÃO PEREIRA
Departamento de Ciência da Computação - UFMG


PROFA. MARIZA ANDRADE DA SILVA BIGONHA
Departamento de Ciência da Computação - UFMG


PROF. MARTIN ALEJANDRO MUSICANTE
Departamento de Informática e Matemática Aplicada - UFRN


PROF. ROBERTO IERUSALIMSCHY
Departamento de Informática – PUC/RJ

Belo Horizonte, 17 de novembro de 2014.

to my wife Helen and my son Emmanuel

Acknowledgments

I dedicate a special thanks to my parents, Maria Cristina and Ednaldo, who have given me love and did everything possible to give me the best education. My thanks to my grandmother Anizia, my uncles Nilton, M6ises, Pedro, J6se, Rosival, my aunts N6ia, Idma, Marlene, Argentina, Vera, my brothers Diego e Milton and my sister Maiza.

Thank you all my friends. My roommates and friends, Henrique (Ratinho), Hussin, Fillipe and Diorgenes, for the good relationship. All my work friends, Thiago Augusto, Wagner, M6nica, Elton, Rodrigo, Alex, Filipe, Welbert and Guilherme Knop, for they make the work more enjoyable. Thank you my doctorate friends, S6rgio, Eliseu, Isabel and Glauber, that allow me a better environment to study.

Thank you to all people who made part of my life in Diamantina city. I specially thank Mariana and my Taekwondo friends: Noel, Gui, Dayvson, Nagila, Camila, Vito and Alexandre. You are a second family for me.

My sincere thanks to my advisors: Prof. Roberto Bigonha and Prof. Vladimir Di Iorio. Thanks for the patience and the learning, I really learned a lot with you two.

I am very grateful to my wife Helen Cristina and my son Emmanuel. Without your love, patience and support I never would go so far. I love you!

*“Why program by hand in five days what you can spend five years of your life
automating?”*

(Terrence Parr)

Resumo

Geradores automáticos de analisadores sintáticos têm sido usados por mais de 50 anos. Ferramentas tais como o YACC automaticamente geram um analisador sintático a partir de uma definição formal da sintaxe da linguagem, que usualmente é baseada em uma gramática livre do contexto. A principal motivação para geradores automáticos de analisadores sintáticos é garantir que o compilador está correto e que reconhece todas as sentenças da linguagem que se pretende especificar, visto que com uma implementação manual é muito difícil de garantir que todos os programas de uma linguagem serão corretamente analisados. Apesar das vantagens mencionadas acima, geradores automáticos de analisadores sintáticos ainda não dão suporte a linguagens que permitem modificar o seu próprio conjunto de regras dinamicamente. Faltam modelos apropriados para descrever tais linguagens, assim como geradores automáticos de analisadores sintáticos eficientes. Portanto, os analisadores sintáticos dessas linguagens são manualmente implementados. Nesta tese, é apresentado o projeto e modelo formal de *Adaptable Parsing Expression Grammars (APEG)*, uma extensão de *Parsing Expression Grammar (PEG)* que permite a manipulação do conjunto de regras sintáticas durante a análise do programa de entrada. Mostramos, também, que APEG é poderoso o suficiente para definir linguagens que exigem a modificação de seu conjunto de regras dinamicamente e analisadores sintáticos gerados a partir do modelo são eficientes para serem usados na prática.

Palavras-chave: APEG, PEG, gramáticas adaptáveis, análise sintática, sintaxe

Abstract

Parser generators have been used for more than 50 years. Tools like YACC can automatically build a parser from a formal definition of the syntax of a language, usually based on context-free grammars (CFG). The main motivation for automatic parser generation is compiler correctness and recognition completeness, since with manual implementation it is very difficult to guarantee that all programs in a given language will be correctly analysed. Despite the advantages mentioned above, the technology of automatic parser generation is still not available for languages that allow on-the-fly modifications on their own set of grammar rules. There is a lack of appropriate formal models for describing the syntax of these languages, therefore efficient parsers may not be automatically generated, requiring handwritten code. In this thesis, we present the design and formal definition of *Adaptable Parsing Expression Grammars (APEG)*, an extension to the *Parsing Expression Grammar (PEG)* model that allows the modification of production rules during the analysis of an input string. We also show that APEG is capable to define languages that require on-the-fly modifications and allows automatic generation of parsers that are reasonably efficient to be used in practice.

Keywords: APEG, PEG, adaptable grammars, parsing, syntax

List of Figures

1.1	An example of a program in the language Foo.	3
1.2	Grammar describing the Foo language.	3
1.3	Examples of programs in extensible Foo	5
2.1	<i>Context Free Grammar</i> (CFG) for the toy language	9
2.2	Illegal program.	9
2.3	A W-grammar for generating the language $\{a^n b^n c^n\}$	10
2.4	Affix grammar for the language $\{a^n b^n c^n \mid n > 0\}$	12
2.5	CFG for generating binary numbers	13
2.6	Derivation tree of the string 1001	14
2.7	Decorated derivation tree of the string 1001	14
2.8	<i>Attribute Grammar</i> (AG) for binary numbers	14
2.9	AG for the language $\{a^n b^n c^n \mid n \geq 0\}$	15
2.10	<i>Extended Attribute Grammar</i> (EAG) for binary numbers	16
2.11	EAG for the language $\{a^n b^n c^n \mid n \geq 0\}$	16
2.12	<i>Parsing Expression Grammar</i> (PEG) for the language $\{a^n b^n c^n \mid n \geq 0\}$. .	22
3.1	Attribute PEG for binary numbers	30
3.2	Semantics of Adaptable PEG - Part I.	35
3.3	Semantics of Adaptable PEG - Part II.	36
3.4	An Attribute PEG for a data dependent language.	38
3.5	Using adaptability in the PEG of Figure 3.4.	38
3.6	Syntax of block with declaration and use of variables (simplified).	39
3.7	Adaptable PEG for declaration and use of variables.	39
4.1	Concrete syntax for the example of Figure 3.4.	47
4.2	APEG with user-defined functions and character classes.	48
4.3	Example with the set of production rules changed at parse time.	48

4.4	Data structures representing a copy of an APEG with a modified production rule.	52
4.5	Example with the set of production rules changed at parse time.	56
4.6	Example of code generated by a PEG.	57
4.7	Generated code for the block language.	59
5.1	Processing of a SugarJ top-level declaration (borrowed from [Erdweg et al., 2011]).	64
5.2	A definition of sugar library for Pairs in SugarJ.	65
5.3	Use of the pair syntax.	66
5.4	Syntax definition of sugar libraries.	68
5.5	Syntax definition of compilation units.	68
5.6	Composition of more than one sugar library.	69
5.7	Definition of the closure syntax.	69
5.8	Definition of a for loop in Fortress.	71
5.9	APEG formalization of Fortress language.	72
5.10	Combining grammars.	73
A.1	Simplified version of the APEG grammar	83

List of Tables

1.1	Parse table for language Foo.	4
2.1	Rules generated by metarules and hyperrules	11
2.2	Rules generated by affixes and rules schema for generating the string <i>aaabbbccc</i>	12
5.1	Time in milliseconds for parsing programs written in the SugarJ language. The performance of the original SugarJ compiler and the APEG version are compared.	74

List of Acronyms

AMG *Adaptive Multi-Pass Grammar*

APEG *Adaptable Parsing Expression Grammar*

AG *Attribute Grammar*

AST *Abstract Syntax Tree*

CFG *Context Free Grammar*

DFA *Deterministic Finite Automaton*

DSL *Domain-Specific Language*

EAG *Extended Attribute Grammar*

LGI *Language Generator by Instil*

PEG *Parsing Expression Grammar*

RAG *Recursive Adaptable Grammar*

TLG *Two-Level Grammar*

Contents

Acknowledgments	xi
Resumo	xv
Abstract	xvii
List of Figures	xix
List of Tables	xxi
List of Acronyms	xxiii
1 On-the-Fly Grammar Modification	1
1.1 Parsing Adaptability	2
1.2 Thesis Objective	5
1.3 Contributions	5
1.4 Thesis Organization	6
2 Toward new Models to Describe Syntax	7
2.1 From Context-Free to Extended Attribute Grammars	8
2.1.1 W-Grammars	10
2.1.2 Affix Grammars	11
2.1.3 Attribute Grammar and Extended Attribute Grammar	13
2.1.4 Discussion	16
2.2 Adaptable Models	17
2.2.1 Imperative Adaptable Models	17
2.2.2 Declarative Adaptable Models	19
2.2.3 Discussion	21
2.3 Parsing Expression Grammar	22
2.4 Data-Dependent Grammar	25

2.5	Conclusion	27
3	Adaptable Parsing Expression Grammar	29
3.1	Attribute Parsing Expression Grammar	30
3.2	Adaptable Parsing Expression Grammar	34
3.3	APEG in Action	37
3.3.1	Data Dependent Languages	37
3.3.2	Static Semantics	38
3.4	Conclusion	40
4	Implementation of Adaptable Parsing Expression Grammars	43
4.1	PEG-related implementations	44
4.2	Implementing an Interpreter for APEG	46
4.2.1	Examples Showing the Concrete Syntax	46
4.2.2	Implementing PEG with Attributes	49
4.2.3	Implementing Adaptability	51
4.3	APEG Properties Associated with Memoization	52
4.4	Mixing Code Generation and Interpretation – An Initial Approach . . .	56
4.5	Conclusion	60
5	Evaluation and Validation	63
5.1	An APEG Implementation of an Extensible Language	64
5.2	The Syntax of SugarJ	65
5.3	The Syntax of Fortress	70
5.4	Performance Evaluation	74
5.5	Conclusion	77
6	Conclusion and Future Work	79
6.1	Adaptability at a Low Complexity Cost	79
6.2	A Reasonably Efficient Implementation	81
6.3	Future Work	81
6.4	Publications	82
A	Adaptable Parsing Expression Grammar	83
	Bibliography	85

Chapter 1

On-the-Fly Grammar Modification

Program source code readability is far more important than writability: every code is written once, and read many times, mainly during the debugging phase of software development. Clarity and legibility are more important than brevity [Rahien, 2010]. Readability can be achieved by many ways, e.g., better software organization, conscious use of modularity, careful coding. The use of *Domain-Specific Languages* (DSLs) has been considered a good way to improve readability [Kosar et al., 2012; Barišić et al., 2011; Kieburtz et al., 1996].

Fowler [2010] describes DSLs as computer languages of limited expressiveness focused on a particular domain. In this definition, it is clear that a DSL is a language to be used by humans to instruct a computer to do something. However, this language, differently from a general-purpose language as Java, only has a minimum set of features needed to support its domain. Therefore, a DSL is not a language to be used for building an entire system; rather, a DSL may be useful for only a single aspect of a system. There is an increasing use of DSLs in software development, because they bring about a lot of benefits, such as improving development productivity [Kosar et al., 2012, 2010] and maintainability [Kosar et al., 2010; van Deursen and Klint, 1998].

Among the various methods for implementing DSLs, *extensible languages*, which are languages that have features that allow adding new constructions to themselves [Wilson, 2004], seem to have several advantages over other approaches [Erdweg et al., 2011; Tobin-Hochstadt et al., 2011; Kosar et al., 2008]. One of the advantages is the possibility of implementing DSLs in a modular way. In this respect, Erdweg et al. [2011] show how DSLs can be implemented using the extensible language SugarJ, by means of syntax units designated as *sugar* libraries, which specify a new construction for a domain concept. Tobin-Hochstadt et al. [2011] also discuss the advantages of implementing DSLs by means of libraries.

The implementation of extensible languages requires adapting the parser every time the language is extended with a new construction. Unfortunately, most popular tools designed for automatic generation of syntactic analyzers do not address resources for the specification and implementation of extensible languages. Moreover, important advances in parsing theory, such as *LL(*)* [Parr and Fisher, 2011], *Adaptive LL(*)* [Parr et al., 2014], *Indentation-Sensitive Grammars* [Adams, 2013] and *Parsing Expression Grammars* (PEGs) [Ford, 2004] do not include features for allowing on-the-fly modification on the rules of the grammar language, changing the parser dynamically. Therefore, extensible languages have been implemented in an ad-hoc way by regenerating a static grammar which accomplishes the new changes, compiling this grammar and using it for parsing the program [Erdweg et al., 2011; Ryu, 2009; Reis et al., 2009]. The development of techniques to enable parser generation to cope with on-the-fly modifications of the parsing grammar seems very important to make the use of DSLs more popular, since it may allow an easier definition of DSLs using extensible languages.

In this thesis, we focus on models and parsing algorithms which allow modifications on the grammar rules on the fly. Our goal is to develop a model that is powerful enough to specify the syntax of extensible languages, as SugarJ [Erdweg et al., 2011], Fortress [Allen et al., 2008, 2009; Ryu, 2009] and XAJ [Reis et al., 2009]. The model must also allow automatic generation of parsers for the defined languages. As a result of the work, we developed *Adaptable Parsing Expression Grammar* (APEG), which is an adaptable model based on PEG.

The organization of the remaining of this chapter is as follows: Section 1.1 discusses the problem we are addressing in more details. Section 1.2 presents the thesis objective and sections 1.3 and 1.4 present the contributions and the organization of this thesis, respectively.

1.1 Parsing Adaptability

One of the most important tasks in a compiler is checking if and how the text representing a program (source code) is a valid sentence of the language, i.e. parsing the program. Generally, a language designer implements this task based on a *Context Free Grammar* (CFG) for defining the syntax of the language, and generates a parse table from this grammar to drive the parsing process. For example, suppose a language designer is creating a language Foo, which is a simple language with expressions on prefixed notation and statements. Figure 1.1 shows an example of a program in this

Figure 1.1 An example of a program in the language Foo.

```

program:
  variables x, y;
  x := 1;
  y := + x 2;
end

```

Figure 1.2 Grammar describing the Foo language.

$start \rightarrow$ “program” “.” $var\ stm$ “end”	r_1
$var \rightarrow$ “variables” $id\ vars$	r_2
$vars \rightarrow$ “,” $id\ vars$	r_3
$vars \rightarrow$ “,”	r_4
$stm \rightarrow id$ “:=” $expr$ “,” stm	r_5
$stm \rightarrow \lambda$	r_6
$expr \rightarrow num$	r_7
$expr \rightarrow$ “+” $expr\ expr$	r_8
$expr \rightarrow$ “*” $expr\ expr$	r_9
$id \rightarrow$ [“a”–“z”] id	r_{10}
$id \rightarrow \lambda$	r_{11}
$num \rightarrow$ [“0”–“9”] num	r_{12}
$num \rightarrow \lambda$	r_{13}

language.

To implement a parser for the Foo language, first the language designer may specify the formal syntax of the language using a CFG and, next, uses this grammar for generating a parser. Ignoring details, a parser algorithm generated is, basically, a table that guides the derivation relation of CFGs. This table tells what rule must be used for the derivation of the leftmost nonterminal considering a lookahead of $k \geq 1$ symbols of the current input. Figure 1.2 shows a grammar which describes the syntax of the Foo language, and Table 1.1 presents a parse table generated from this grammar, using a top-down approach with $k = 1$. Every rule of the grammar presented in Figure 1.2 has a label, r_i , which is used for referencing it on Table 1.1.

The parse table presented in Table 1.1 is used for verifying the program in Figure 1.1 as follows: the recognition process begins with the nonterminal *start* and if the first symbol on input is the token **program**, the table defines that the rule to be used is $start \rightarrow$ “program” “.” $var\ stm$ “end”. As the two first terminals of this rule, “program” and “.”, match the two input tokens, they are consumed. Now, the next token on input is the token **variables** and the leftmost nonterminal is *var*. Again, looking at line *var* and column **variables** on the table, the rule of *var* that is used is

	$[a - z]$	$[0 - 9]$	“+”	“*”	“,”	“;”	“:=”	“program”	“variables”	“end”
<i>start</i>								r_1		
<i>var</i>									r_2	
<i>vars</i>					r_4	r_3				
<i>stm</i>	r_5									r_6
<i>expr</i>		r_7	r_8	r_9						
<i>id</i>	r_{10}				r_{11}	r_{11}	r_{11}			
<i>num</i>		r_{12}	r_{13}	r_{13}	r_{13}					

Table 1.1: Parse table for language Foo.

$var \rightarrow \text{“variables” } id \text{ vars}$. This process progresses until either an error is found (blank cell in table) or the input ends.

CFGs and their parsing algorithms can handle a large class of languages which do not perform changes in their set of rules during the parsing. In fact, CFGs are not suitable for languages which allows on-the-fly modifications in their rules. For illustrating this situation, suppose an extensible version of the Foo language, which has a statement to add new rules to its own grammar. Such a statement can be represented by the following rule, added to the CFG of Figure 1.2:

$$stm \rightarrow \text{“\#” } id \text{ “} \Rightarrow \text{” } rhs \text{ “;”}$$

A statement that extends the language begins with the symbol $\#$ followed by the name of a nonterminal, the symbol \Rightarrow , the right side of a rule (defined by nonterminal rhs whose rules are not presented) and ends with a semicolon. When this rule is applied in a derivation, its semantics is to extend the grammar, adding a rule $id \rightarrow rhs$, such as the program of Figure 1.3a, which adds a rule for division expressions. However, if the parsing table is not updated immediately, the valid program in Figure 1.3b is not correctly parsed, because this program uses the extension just added.

This problem happens because the Foo language defines how to parse itself, but traditional parsing algorithms use a static parser table and do not allow to modify it during the input analyses. The same problem arises when a bottom-up approach is taken. The example above is a toy language, but it illustrates a feature required in extensible languages such as SugarJ, Fortress, Racket [Tobin-Hochstadt et al., 2011] and Lisp.

The challenge is to design a parsing algorithm that performs on-the-fly changes on the parse table (or an equivalent structure used to drive the parser) at parsing time and add an information to use new rules, such as $expr \rightarrow \text{“/” } expr \text{ expr}$.

Figure 1.3 Examples of programs in extensible Foo

<pre> program: variables x, y; x := 4; # expr => “/” expr expr; y := + x 2; end </pre> <p>(a) A program which does not use new syntax</p>	<pre> program: variables x, y; x := 4; # expr => “/” expr expr; y := / x 2; end </pre> <p>(b) A program which uses a new syntax.</p>
--	---

1.2 Thesis Objective

Our goal is the development of an adaptable model which must satisfy the following requirements:

- to offer facilities for adapting the grammar during the parsing process, without adding too much complexity to the base model;
- to assure that grammar extensions take effect immediately;
- to allow an implementation of an automatic parser generator with reasonable efficiency.

The model developed in this thesis is based on PEG and is called *Adaptable Parsing Expression Grammar* (APEG). The proposed model has focus on the syntactic issues only, so we only discuss in details how to define the syntax of extensible languages using APEG. However, the model does not impose any restriction on how the semantics issues may be dealt with. Similarly to other models, for example, an *Abstract Syntax Tree* (AST) can be built after parsing the program, and the semantics can be defined by operations over this AST.

1.3 Contributions

The contributions of this work are in implementations of programming languages. The specific contributions are:

- the design of an adaptable model based on PEG;
- a careful formalization of the model;

- on-the-fly grammar and parser adaptability;
- an automatic parsing generation for APEG;
- evaluation of the using of APEG in the implementation of real extensible languages.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 presents a revision of formalisms for the specification of the syntax of languages. Our definition model and formalization are presented in Chapter 3, together with examples of use and a comparative discussion with other models. Chapter 4 explains the implementation of APEG, discussing how we combine PEG and attributes and how adaptability is achieved in the model. Chapter 5 shows how the model can be used for describing the syntax of real extensible languages and an experimental evaluation of our implementation with respect to execution time. Finally, Chapter 6 concludes this work and discusses the future work.

Chapter 2

Toward new Models to Describe Syntax

A language is a set of sentences over an alphabet. To define the syntax of languages, Chomsky [1956, 1959] proposed grammars, which is a system that generates every sentence of the language that is intended to describe.

Formally, a grammar is a 4-tuple (V_N, V_T, R, S) , where $V = V_N \cup V_T$ is the vocabulary, R is a finite set of rewrite rules and $S \in V_N$ is the initial sentential form. V_N and V_T are disjoint set of nonterminals and terminals, respectively. A rule has the form $\varphi \rightarrow \psi$, where φ and ψ are strings of symbols of the vocabulary. A sentence of the language is generated from the initial sentential form S and applying the rewrite rules until reaching a string with only terminal symbols.

For example, suppose a grammar containing the rules: $S \rightarrow AB$, $A \rightarrow C$, $CB \rightarrow Cb$ and $C \rightarrow a$. A derivation of the sentence ab is showed below

$$S \xRightarrow{S \rightarrow AB} AB \xRightarrow{A \rightarrow C} CB \xRightarrow{CB \rightarrow Cb} Cb \xRightarrow{C \rightarrow a} ab$$

where the derivation begins with nonterminal S , then the rule $S \rightarrow AB$ is applied to get the sentence AB . Next, the rule $A \rightarrow C$ is applied to the sentence, resulting CB . The symbol \Rightarrow is used to represent the derivation relation. In order to show all details of the process, the rule used in every derivation step is showed on top of the derivation symbol \Rightarrow .

After his seminal paper in 1956 [Chomsky, 1956], Chomsky made restrictions on the form of the rules of a grammar to obtain better devices for reasoning about the structure of natural languages [Chomsky, 1959]. The most important restriction is the one that allows only a nonterminal symbol on the left hand side of a rule, which creates

the *Context Free Grammars* (CFGs) that had demonstrated to be of great importance for describing the syntax of programming languages since its first use in the definition of the syntax of ALGOL. The restrictions made by Chomsky is what we now know as Hierarchy of Chomsky with four types of grammars: regular grammars (type 3), context-free grammars (type 2), context-sensitive grammars (type 1) and unrestricted grammars (type 0).

Although the main purpose of grammars was to study the structure of the English language, CFGs (type 2 grammars) have been used for describing the syntax of programming languages, because of the simplicity of the generative mechanism and the existence of efficient programs to check if a string is generated by a particular CFG. However, there are some programming language features, such as the relation between declaration and use of variables or on-the-fly grammar modification, that cannot be formalized by a CFG. Therefore, various augmentations of CFG or new models have been proposed for describing the full syntax of programming languages, such as *Extended Attribute Grammar* (EAG) [Watt and Madsen, 1983], Christiansen' Adaptable Grammar [Christiansen, 1990] and *Recursive Adaptable Grammar* (RAG) [Shutt, 1998].

After this brief historical introduction on grammars, this chapter presents a review of the various models that have been proposed for describing the syntax of programming languages. First, Section 2.1 discusses several augmentations to CFGs, which are not adaptable, i.e., it is impossible to explicitly manipulate the set of the grammar rules. Next, Section 2.2 discusses adaptable models. In Sections 2.3 and 2.4, we discuss two other models: *Parsing Expression Grammar* (PEG) and *Data-Dependent Grammars*. We have decided to discuss them in separate sections because they are important advances in theory of programming languages and our model presented in Chapter 3 is strongly inspired by PEG and has features of data-dependent grammars. Finally, Section 2.5 concludes this chapter.

2.1 From Context-Free to Extended Attribute Grammars

Consider a toy language in which a program begins with a list of declarations of variables followed by a list of update commands. The type of the variables can be integer or boolean, and an update command is composed by a variable on the left hand side and an expression on the right hand side. Moreover, this language has the following constraints: 1) a variable cannot be declared more than once; 2) a variable cannot be

Figure 2.1 CFG for the toy language

Prog	→	program Decl Stmt end
Decl	→	Type Vars ; Type Vars ; Decl
Type	→	integer boolean
Vars	→	Identifier Vars , Identifier
Stmt	→	Identifier = Expr ; Identifier = Expr ; Stmt
Expr	→	Identifier Number Boolean
Identifier	→	Char Char Identifier
Char	→	a b ... z
Number	→	Digit Digit Number
Digit	→	0 1 ... 9
Boolean	→	true false

Figure 2.2 Illegal program.

```

program
  boolean c;
  c = 5;
end

```

used if it was not declared; and 3) the expression on the right hand side of an update command must have the same type of the variable on the left hand side.

Figure 2.1 describes the syntax of the language using a CFG, except for the constraints, because they cannot be specified with a CFG, since these constraints are not context-free. Therefore, illegal programs can be derived from this grammar, such as the program in Figure 2.2. According to Slonneger and Kurtz [1995], language implementers say that infractions, like that in Figure 2.2, which involves the context, belongs to the **static semantics** of a language. This name, static semantics, is because detecting an error involves the meaning of symbols and it can be statically determined from the text of the program. However, Slonneger and Kurtz [1995] argue that static errors belong to the syntax, not the semantics of a language.

Although Slonneger and Kurtz [1995] claim that “static semantics” are related to syntax, CFG cannot describe them, therefore, “static semantics” must be specified by other way, possibly, using other formalisms more powerful than CFG. Context-sensitive grammars are powerful enough to describe all the aspects of programming languages, but these grammars are unsuitable for several reasons: 1) the expansion of a node in the derivation tree may depend on sibling nodes; 2) the direct hierarchical relationships between nonterminals that furnish a basis for semantics descriptions is lost; 3) formal context-sensitive grammars are difficult to construct and understand; and 4) parsing algorithms for context-sensitive grammars are, in general, not efficient, thus, are not

Figure 2.3 A W -grammar for generating the language $\{a^n b^n c^n\}$

$\begin{array}{l} N \vdash u \mid uN \\ L \vdash a \mid b \mid c \end{array}$ <p>(a) metarules.</p>	$\begin{array}{l} N \mapsto Na, Nb, Nc \\ NuL \mapsto NL, L \\ uL \mapsto L \end{array}$ <p>(b) hyperrules.</p>
---	---

suitable for compiler implementation [Slonneger and Kurtz, 1995]. The remaining of this section discusses augmentations of CFGs which are capable to describe these aspects of the syntax of programming languages and, at the end of this section, we discuss the strength and weakness of them focusing in our main problem: adapting the grammar on the fly.

2.1.1 W -Grammars

The ALGOL 68 report [Wijngaarden, 1969] presents a formalism called *W-Grammars*, formerly known as *Two-Level Grammar* (TLG), for defining the syntax and semantics of this language [de Chastellier and Colmerauer, 1969]. TLGs are a formalism which uses two levels of rules in a way that the first rules generate a scheme of nonterminals which are used in the second group of rules to create a possible infinite number of rules. For example, to generate the language $\{a^n b^n c^n \mid n \geq 0\}$ the metarules (as called the first level of rules) in Figure 2.3a act generating a set of strings from every nonterminal which is used to rewrite this nonterminal in the hyperrules (the second level of rules) in Figure 2.3b. The metarules are similar to CFG, but the hyperrules use a comma as a separate symbol.

In this example, for deriving the string **a,a,b,b,c,c**, first we derive the string uu from the start symbol N as follows (we use the symbol \models for the derivation relation on the metarules to distinguish of the symbol \Rightarrow used for deriving strings)

$$N \models uN \models uu$$

which combined with the hyperrule $N \mapsto Na, Nb, Nc$ generates the rule $uu \rightarrow uua, uub, uuc$. So, this rule is used in a derivation process from the axiom uu (an axiom is every string that can be derived from the start symbol using only the metarules) as

$$uu \Rightarrow uua, uub, uuc$$

In a similar way, the rules used in the derivation are built by the combination of the metarules and hyperrules. Following, we show all steps in the derivation of the

number rule	metarules	hyperrule	rule
1	$N \stackrel{*}{\models} uu$	$N \mapsto Na, Nb, Nc$	$uu \rightarrow uua, uub, uuc$
2	$N \vdash u$ and $L \vdash a$	$NuL \mapsto NL, L$	$uua \rightarrow ua, a$
3	$L \vdash a$	$uL \mapsto L$	$ua \rightarrow a$
4	$N \vdash u$ and $L \vdash b$	$NuL \mapsto NL, L$	$uub \rightarrow ub, b$
5	$L \vdash b$	$uL \mapsto L$	$ub \rightarrow b$
6	$N \vdash u$ and $L \vdash c$	$NuL \mapsto NL, L$	$uuc \rightarrow uc, c$
7	$L \vdash c$	$uL \mapsto L$	$uc \rightarrow c$

Table 2.1: Rules generated by metarules and hyperrules

string $\mathbf{a,a,b,b,c,c}$. The number of the rules used in every step of the derivation is on top of the derivation symbol \Rightarrow and Table 2.1 shows how the metarules and hyperrules are used to generate them with their respective number rules.

$$\begin{aligned}
uu &\stackrel{1}{\Rightarrow} uua, uub, uuc \stackrel{2}{\Rightarrow} ua, a, uub, uuc \stackrel{3}{\Rightarrow} a, a, uub, uuc \stackrel{4}{\Rightarrow} a, a, ub, b, uuc \\
&\stackrel{5}{\Rightarrow} a, a, b, b, uuc \stackrel{6}{\Rightarrow} a, a, b, b, uc, c \stackrel{7}{\Rightarrow} a, a, b, b, c, c
\end{aligned}$$

As discussed, the idea of TLG is to combine two grammars to generate an infinite set of rules, in which strings generated by a nonterminal (represented by capital letter) of metarules can be replaced in hyperrules to form rules. TLGs can generate any recursively enumerable language [Sintzoff, 1967], but there are not efficient automatic parser generators based on TLGs. Also, although the model allows generating an infinite set of rules, it does not have features to easily adapt the set of rules during the parsing of a program. However, the idea introduced by TLGs of generating rules by replacing some symbols is used in other models.

2.1.2 Affix Grammars

Affix Grammars is an augmentation of CFG proposed by Koster [1970]. In this model, nonterminals have affixes which can assume values over a certain domain and act like parameters of the nonterminals. Affix Grammars are very similar to W-Grammars and have a two-level grammar; the first level generates the domain of affixes and the second is the rule scheme with nonterminals and its affixes. As an example, the language, $\{a^n b^n c^n \mid n \geq 1\}$, generated in Section 2.1.1 using a W-Grammar, can be specified using Affix Grammar as in Figure 2.4a and 2.4b.

The affix rules presented in Figure 2.4a generate the domain of affix and, in this example, define every natural number in a notation of successor. A string over the domain of an affix (generated by a CFG) is used for forming a rule by substituting it

Figure 2.4 Affix grammar for the language $\{a^n b^n c^n \mid n > 0\}$.

$N \vdash 1 \mid s(N)$ (a) Rules for generating the domain of affixes.	$S \mapsto A(N) B(N) C(N)$ $A(1) \mapsto a$ $A(s(N)) \mapsto a A(N)$ $B(1) \mapsto b$ $B(s(N)) \mapsto b B(N)$ $C(1) \mapsto c$ $C(s(N)) \mapsto c C(N)$ (b) Rule schema.
--	---

in the rule schema. For example, substituting the string $s(s(s(1)))$, which is generated from N , in the rule schema $S \mapsto A(N) B(N) C(N)$, we obtain the rule $S \rightarrow A(s(s(s(1)))) B(s(s(s(1)))) C(s(s(s(1))))$. Using the affix and rule schema for generating new rules, it is possible to generate every string of the language $\{a^n b^n c^n \mid n > 0\}$. Table 2.2 shows the rules used for generating the string $aaaabbbbcccc$.

The affixes can be of two types: *inherited* or *derived*. The inherited affixes are the input parameters of the nonterminals and the derived ones are the output parameters. The values of the derived affixes are calculated by functions called in productions rules. In the example above, the affixes of the nonterminals are inherited.

Affix Grammars are very similar to W-Grammars, but they are more suitable for parsing than W-Grammars and have been used for describing natural languages [Koster, 1991a,b]. As W-Grammars, Affix Grammars do not have features to modify the set of rules during parsing.

Affix rule	Rule schema	rule
$N \stackrel{*}{\models} s(s(s(1)))$	$S \mapsto A(N) B(N) C(N)$	$S \rightarrow A(s(s(s(1)))) B(s(s(s(1)))) C(s(s(s(1))))$
$N \stackrel{*}{\models} s(s(s(1)))$	$A(s(N)) \mapsto a A(N)$ $B(s(N)) \mapsto b B(N)$ $C(s(N)) \mapsto c C(N)$	$A(s(s(s(1)))) \rightarrow a A(s(s(1)))$ $B(s(s(s(1)))) \rightarrow b B(s(s(1)))$ $C(s(s(s(1)))) \rightarrow c C(s(s(1)))$
$N \stackrel{*}{\models} s(s(1))$	$A(s(N)) \mapsto a A(N)$ $B(s(N)) \mapsto b B(N)$ $C(s(N)) \mapsto c C(N)$	$A(s(s(1))) \rightarrow a A(s(1))$ $B(s(s(1))) \rightarrow b B(s(1))$ $C(s(s(1))) \rightarrow c C(s(1))$
$N \stackrel{*}{\models} s(1)$	$A(s(N)) \mapsto a A(N)$ $B(s(N)) \mapsto b B(N)$ $C(s(N)) \mapsto c C(N)$	$A(s(1)) \rightarrow a A(1)$ $B(s(1)) \rightarrow b B(1)$ $C(s(1)) \rightarrow c C(1)$

Table 2.2: Rules generated by affixes and rules schema for generating the string $aaaabbbbcccc$

Figure 2.5 CFG for generating binary numbers

$$\begin{aligned}
S &\rightarrow T \\
T &\rightarrow B T \\
T &\rightarrow B \\
B &\rightarrow 0 \\
B &\rightarrow 1
\end{aligned}$$

2.1.3 Attribute Grammar and Extended Attribute Grammar

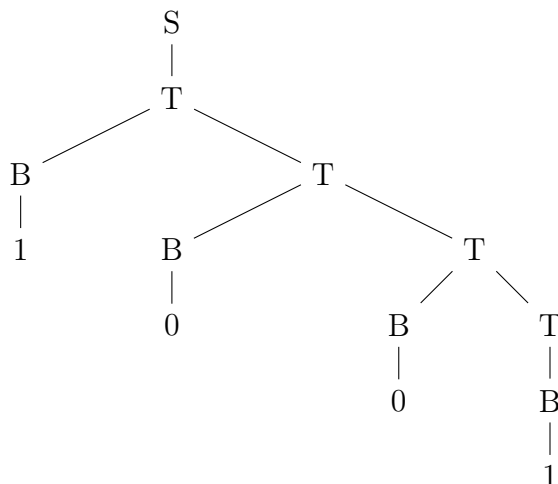
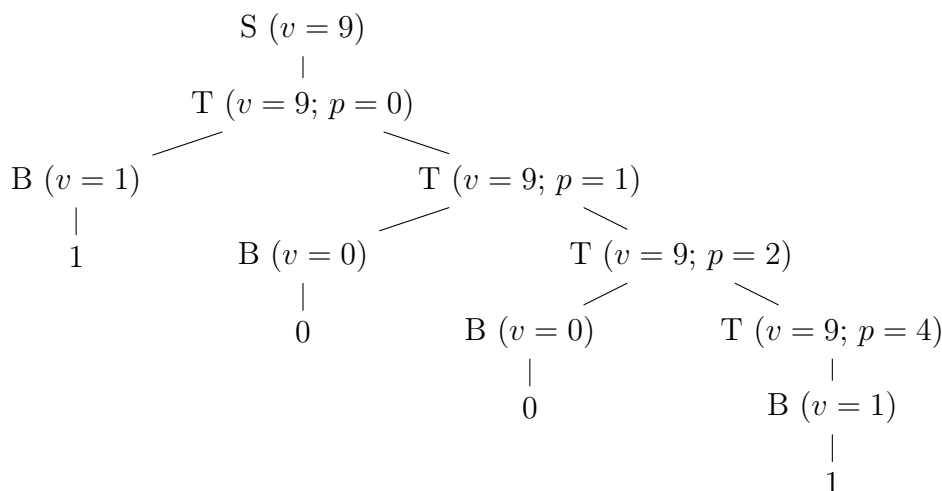
Differently from the other formalisms presented here, which were designed for describing the full syntax of a programming language, *Attribute Grammars* (AGs) was designed by Knuth [1968] to specify the meaning (semantics) of strings generated by CFGs. An AG is a CFG where every symbol of the vocabulary has a set of attributes and there are a set of functions for every rule in the grammar that compute the semantic values of the attributes.

In an AG, the semantics of a string is given from its derivation tree by computing the value of the attributes of every node in the derivation tree using the corresponding *semantic equations*. There are two types of attributes: *inherited* and *synthesized*. Inherited attributes are those whose values are based on the attributes of the ancestor or sibling symbols and synthesized attributes are the ones based on attributes of the descendants.

To illustrate how AG works, let's consider the grammar of Figure 2.5, which generates binary numbers and the derivation tree for the string 1001, in Figure 2.6. In order to give the semantic of a binary number generated by this grammar, we add the following attributes to the nonterminals:

- S has a synthesized attribute named $v(S)$. This attribute, over the domain of integer numbers, represents the semantic value of the binary number;
- T has two attributes, a synthesized attribute $v(T)$ and an inherited attribute $p(T)$, both over the domain of integer numbers. The attribute $v(T)$ represents the semantic value of the binary number and $p(T)$ will represent the value of the string generated before T in the derivation tree;
- B has only one synthesized attribute, $v(B)$. This attribute, over the domain of integer numbers, represents the semantic value of the string generated from B .

The attribute $v(T)$ is used to flow the semantic value of the expression to the root, and the inherited attribute $p(T)$ is used to pass the intermediate value of the expression. Figure 2.8 shows the AG for binary numbers with the semantic equations

Figure 2.6 Derivation tree of the string 1001**Figure 2.7** Decorated derivation tree of the string 1001**Figure 2.8** AG for binary numbers

$S \rightarrow T$	$v(S) \leftarrow v(T)$	$p(T) \leftarrow 0$
$T \rightarrow B T_1$	$v(T) \leftarrow v(T_1)$	$p(T_1) \leftarrow 2 * p(T) + v(B)$
$T \rightarrow B$	$v(T) \leftarrow 2 * p(T) + v(B)$	
$B \rightarrow 0$	$v(B) \leftarrow 0$	
$B \rightarrow 1$	$v(B) \leftarrow 1$	

for each rule of the grammar. In this grammar, we use a subscript in a nonterminal to avoid ambiguities when an attribute is referenced. Figure 2.7 shows the derivation tree for the string 1001 decorated with the values of the attributes.

An AG can also be used to specify the context-sensitive aspects of the syntax of a programming language. The idea is to generate the language without checking the context-sensitive aspects with the CFG facilities and use the attributes, semantic

Figure 2.9 AG for the language $\{a^n b^n c^n \mid n \geq 0\}$.

$S \rightarrow A B$	condition: $n(A) = n(B)$
$A \rightarrow a A_1 b$	$n(A) \leftarrow n(A_1) + 1$
$A \rightarrow \lambda$	$n(A) \leftarrow 0$
$B \rightarrow c B_1$	$n(B) \leftarrow n(B_1) + 1$
$B \rightarrow \lambda$	$n(B) \leftarrow 0$

equations and conditions to check the context-sensitive aspects on the derivation tree. For example, the language $\{a^n b^n c^n \mid n \geq 0\}$ can be recognized by an AG by generating the context-free language $\{a^n b^n c^m \mid n, m \geq 0\}$ and using attributes for checking whether n is equal to m . Figure 2.9 shows the AG that generates this language. The strings $aabbcc$ and $abcc$ are generated by the CFG presented in Figure 2.9, but only the former satisfies the attribute conditions. So, the second string is not in the language defined by the AG presented in Figure 2.9.

An *Extended Attribute Grammar* (EAG) is a model for formalizing context-sensitive features of programming languages based on AG and Affix Grammar and was designed to be more elegant, readable and generative in nature [Watt and Madsen, 1983]. Essentially, an EAG is a AG where the conditions and semantic equations are embedded in the rules. EAG, for improving the readability, also uses the up arrow and down arrow symbols for synthesized and inherited attributes, respectively.

Inherited attributes on the left hand side and synthesized attributes on the right hand side of a rule are said to be in *defining positions*. Synthesized attributes on the left hand side and inherited attributes on the right hand side of a rule are said to be in *applied positions*. An improvement of EAG over AG is the use of an expression to calculate the values of the attributes in the applied positions, allowing a more concise specification of the semantic equations. For example, Figure 2.10 shows the EAG deriving from the AG presented in Figure 2.8 for binary numbers. In the first rule, there is an implicit semantic equation which copy the value of the first synthesized attribute of the nonterminal T , on the right hand side of the rule, to the synthesized attribute of S . This is done by using the same variable name for both attributes. There is another semantic equation in the first rule that defines the value of the inherited attribute of the nonterminal T as 0. In the second rule, the value of the inherited attribute of the nonterminal T , on the right hand side, is calculated using the equation $2 * p + v_1$, which uses the values of the inherited attribute p of the nonterminal T of the left hand side of the rule and the synthesized attribute v_1 of the nonterminal B . The other semantic equations are defined in a similar way.

In general, defining positions have a variable name. Constraints may be specified

Figure 2.10 EAG for binary numbers

$\langle S \uparrow v \rangle$	\rightarrow	$\langle T \uparrow v \downarrow 0 \rangle$
$\langle T \uparrow v \downarrow p \rangle$	\rightarrow	$\langle B \uparrow v_1 \rangle \langle T \uparrow v \downarrow 2 * p + v_1 \rangle$
$\langle T \uparrow 2 * p + v \downarrow p \rangle$	\rightarrow	$\langle B \uparrow v \rangle$
$\langle B \uparrow 0 \rangle$	\rightarrow	0
$\langle B \uparrow 1 \rangle$	\rightarrow	1

Figure 2.11 EAG for the language $\{a^n b^n c^n \mid n \geq 0\}$.

$\langle S \rangle$	\rightarrow	$\langle A \uparrow n \rangle \langle B \uparrow n \rangle$
$\langle A \uparrow n + 1 \rangle$	\rightarrow	$a \langle A \uparrow n \rangle b$
$\langle A \uparrow 0 \rangle$	\rightarrow	λ
$\langle B \uparrow n + 1 \rangle$	\rightarrow	$c \langle B \uparrow n \rangle$
$\langle B \uparrow 0 \rangle$	\rightarrow	λ

using the same name of variable or expressions in defining positions of a rule. For example, Figure 2.11 shows an EAG for the language $\{a^n b^n c^n \mid n \geq 0\}$ where there is a constraint defining that the value of the synthesized attribute of the nonterminals A and B must be the same. This constraint is specified using the same name, n , in the defining position.

None of EAGs or AGs allow changing the set of rules during the parsing, but there are many parser generators based on AGs and efficient parsing algorithms. Our model developed in this work, *Adaptable Parsing Expression Grammar* (APEG), uses the idea of attributes and also has a notation similar to EAGs.

2.1.4 Discussion

So far, we presented some formalisms to specify context-sensitive aspects of programming languages. As CFG has demonstrated to be elegant, readable, intuitive and simple, all these formalisms augment the CFG kernel with distinct facilities. Besides the formalisms presented here, there are dozen of others which augment the CFG kernel [Aho, 1968; Salomaa, 1972; Mayer, 1972; Rozenberg and Wood, 1980]. AG and EAG have proven to be the most successful formalisms to describe the context-sensitive aspects of programming languages, because of their simplicity and suitability for automatic compiler construction.

However, AG and EAG, and also the other formalisms presented in this section, are not suitable for describing languages whose syntax can be changed during parsing, because their model is based on an immutable grammar during the derivation process. As these languages may change their own set of rules during the derivation process, other formalisms which allow modify their grammar may be more appropriate. The

next section presents this type of formalism.

2.2 Adaptable Models

Some authors [Burshteyn, 1990b; Christiansen, 1990; Shutt, 1998] argue that a way of interpreting context-sensitive features of programming languages, such as variable declarations, is to see them as adding new grammatical constructs to the language. The developed models based on this idea, called *adaptable grammars*, explicitly provide mechanisms to manipulate the set of rules.

In an adaptable grammar, the task of checking whether a variable used in an expression has been previously defined may be performed as follows. Instead of having a general rule like `variable -> identifier`, each variable declaration may add a new rule to the grammar. For example, the declaration of a variable with name `x` adds the following production rule:

```
variable -> "x".
```

The nonterminal *variable* will then generate only the declared variables, and not a general identifier.

The main purpose of adaptable grammars was initially to give a syntactic treatment for the definition of context-sensitive features of programming languages, but they are powerful enough even for the definition of advanced extensibility mechanisms of programming languages, like the one presented in Figure 1.3b in Section 1.1. The purpose of this section is to review adaptable models. For this, we use the classification of adaptable models adopted by Shutt [1998]. Based on the way the set of rules is manipulated, Shutt classifies adaptable models as *imperative* or *declarative*, which we discuss in the sequel.

2.2.1 Imperative Adaptable Models

Imperative models are inherently dependent on the parsing algorithm. The set of rules is treated as a global entity that is modified while derivations are processed. Thus, the grammar designer must know exactly the order of decisions made by the parser.

To the best of our knowledge, it seems that Wegbreit [1970] was the first to formalize the idea of grammars that allow the manipulation of their own set of rules, so the idea has been around for at least 40 years. Wegbreit proposed *Extensible Context Free Grammars*, consisting of a CFG together with a finite state transducer. The finite state transducer is used to modify the set of rules as follows: in every step of

a derivation, $xA\alpha$, the transducer gets the terminal string x and outputs the rules to change the grammar; the new set of productions is used in the remaining derivation process to replace the nonterminal A . The derivation must be leftmost.

In [Burshteyn, 1990b], the author presents the idea of *modifiable grammar*, which is an *Attribute Grammar* where the semantic equations add new rules to the grammar. The inherited and synthesized attributes are used to carry values that are used for compounding the new rules. The rules are inserted in the grammar when a reduction is applied on parsing. In a following work, Burshteyn [1990a] formalized this idea and presented two versions of modifiable grammar: *top-down modifiable grammar* and *bottom-up modifiable grammar*. In both formalizations, a modifiable grammar consists of a CFG and a Turing transducer, with instructions that may define a list of rules to be added, and another to be deleted. Burshteyn presents these two versions because of the dependency on the parser algorithm. The former is for a top-down and the later for a bottom-up parsing.

Cabasino et al. [1992] propose *evolving grammars*, which are grammars that evolve the set of rules and nonterminals over time. In their definition, the grammar evolution is specified by a sequence of grammars, G_k, G_{k+1}, \dots, G_n , where every G_{i+1} grammar differs from previous one, G_i , by an addition of new productions and nonterminals. Every production of the CFG may have an associated rule which defines the set of new nonterminals and production that are added to the current grammar to create the next grammar in the evolving sequence. The derivation must be rightmost and the associated parser bottom-up. Although evolving grammars are capable to define context-sensitive conditions like declare-use of variables, Cabasino et al. [1992] reported problems to deal with syntactic scope.

Boullier [1994] defines *Dynamic Grammars*, a formalism with a recursively enumerated set of grammars which is very similar to Evolving Grammar of Cabasino et al. [1992]. A Dynamic Grammar is a tuple (\mathcal{M}, G_0) , where \mathcal{M} is a Turing machine and G_0 is the initial grammar. The Turing machine generates the next grammar of the recursive set from the current configuration on the derivation and the current grammar. Differently from Evolving Grammar, the Turing machine can also generate grammars by deleting rules. The derivation must be rightmost and the associated parser must be bottom-up, such as Evolving Grammar.

Carmi [2010] argues that existing adaptable formalisms do not handle forward references well, such as goto statements that precede label declarations, and extensible languages with features like macro syntax and its expansion. Thus, he proposes a new model, called *Adaptive Multi-Pass Grammar* (AMG). As in Dynamic Grammars and Evolving Grammars, the formalism proposed by Carmi [2010] is driven by the parsing

algorithm and the derivation must be rightmost. Differently from these models, the nonterminal symbols of AMGs may have annotation strings and a special type of rule, a multi-pass rule. A multi-pass rule is similar to a simple rule, however, when the parser reduces using this type of rule, the annotated string of the rule is put as a prefix of the input to be parsed. The multi-pass rules together with nonterminal's annotations allow parsing a prefix of the input string and putting it back in the input to be parsed again, possible with other set of rules, allowing handling forward references and macros definition and expansion.

2.2.2 Declarative Adaptable Models

Imperative models have the disadvantage of the inherent dependency of the parsing algorithm, thus the grammar designer must know exactly the order of decisions made by the parser. Declarative models, on the other hand, are relatively independent of the parsing algorithm and do not impose restrictions on the derivation order. Below we discuss the declarative models.

Christiansen [1987] proposes *Generative Grammars*, that is essentially an EAG in which the first attribute of every nonterminal symbol is inherited and represents the *language attribute*. The language attribute contains the set of rules allowed in each derivation. The initial grammar works as the language attribute for the root node of the parse tree, and new language attributes may be built and used in different nodes. Each grammar adaptation is restricted to a specific branch of the parse tree. One advantage of this approach is that it is easy to define statically scope dependent relations, such as the block structure declarations of several programming languages. In his survey of approaches for adaptable grammar formalisms, Christiansen [1990] introduces a new syntax for *Generative Grammar*, called *Generative Clause Grammars*, whose syntax is more related with the implementation of the model in Prolog. Later, he proposed an equivalent approach, using definite clause grammars [Christiansen, 2009]. Here we refer to the first formalization presented by Christiansen.

Shutt [1998] argues that, in AG and other extensions for CFGs, the clarity of the original base CFG model is undermined by the power of the extending facilities. As an example, Shutt cites that Christiansen gives an example of an AG for ADA, in which a single rule representing function calls needs two and a half pages to describe the context conditions associated to it. Shutt claims that adaptable grammars should be a natural way of describe context-sensitive elements of a programming language. However, he observes that even the generative grammars of Christiansen inherit the non orthogonality of AGs, with two different models competing. The CFG kernel is

simple, generative, but computationally weak. The augmenting facility is obscure and computationally strong. Therefore, he proposes a declarative model called RAG.

Shutt defines RAG as a one-sorted algebra, where a single domain combines the syntactic elements (terminals), meta-syntactic (nonterminals and the language attribute) and semantic values (all other attributes). A RAG has a rule function which maps nonterminals to a set of *unbound rules*, where an unbound rule is a rule of the form

$$\langle v_0, e_0 \rangle \rightarrow t_0 \langle e_1, v_1 \rangle t_1 \dots \langle e_{n-1}, v_{n-1} \rangle t_{n-1} \langle e_n, t_n \rangle t_n$$

and every t_i are terminals, v_i are distinct variables and e_i are elements of a term algebra extended with the variables set $\{v_0, v_1, \dots, v_n\}$. *Bound rules* are formed by substituting the variables by elements of the algebra. The bound rules are used in the derivation process. For clarity, consider the following example taken from [Shutt, 1998]. The RAG with the rule function present below generates the language $\{a^n b^n \mid n \geq 0\}$

$$\rho(S) = \left\{ \begin{array}{l} \langle v_0, \lambda \rangle \rightarrow \lambda \\ \langle v_0, \lambda \rangle \rightarrow a \langle v_0, v_1 \rangle b \end{array} \right\}$$

where S is the start symbol.

RAG imposes the restriction that the leftmost variable v_0 of a rule r must be bound only to some element, a , such that $r \in \rho(a)$. Therefore, the only possible assignment to v_0 is S . For deriving any string, we must begin using a bound rule of the start symbol S . For example, bounding v_0 to S and v_1 to λ we can start the derivation with the rule $\langle S, \lambda \rangle \rightarrow a \langle S, \lambda \rangle b$, thus

$$\langle S, \lambda \rangle \Rightarrow a \langle S, \lambda \rangle b$$

Following this idea, the derivation of the string $aaabbb$ is

$$\langle S, \lambda \rangle \Rightarrow a \langle S, \lambda \rangle b \Rightarrow aa \langle S, \lambda \rangle bb \Rightarrow aaa \langle S, \lambda \rangle bbb \Rightarrow aaabbb$$

Note that the only bindings to the variables which yields into a useful rule are v_0 to S and v_1 to λ , in this example. The only possible value to v_0 is S because the restriction imposed by RAG, as discussed above, and any value bounded to v_1 different from λ results in a nonterminal, in the rule $\langle S, \lambda \rangle \leftarrow a \langle S, v_1 \rangle b$, that is impossible to rewrite in the derivation process.

For defining more complex languages, a RAG designer must understand well the algebraic machinery defined for RAG and uses nonterminals as n -arity operators and several pre-defined operators. Although the main idea of RAG is the simple substitu-

tion of variables to make rules, the operators that must be defined to create complex languages can become a hard task to create RAGs.

Stansifer and Wand [2011] define *Parsing Reflective Grammars*, whose purpose is to give a formal definition of the syntax of extensible languages. New rules are added to a parsing reflective grammar through a special meta-symbol, \mathbb{R} , which indicates the point that begins an extension of the grammar. The designer of the language specifies the points of extension using this special symbol on the right hand side of any rule. The set of the rules is manipulated in a declarative way, since there is no order imposed on derivation. However, the model is very limited and imposes a special symbol before the insertion of new rules. Moreover, a power model capable of defining the syntax of extensible languages is also capable to define static semantics by changing the grammar, and Parsing Reflective Grammars are not powerful enough for this task.

2.2.3 Discussion

Christiansen [1990] enumerates several difficulties of adaptable models for describing static semantics of programming languages: removing rules at block exit; delayed or indirect declarations; preventing multiple declarations. These concerns are also relevant for extensible languages, because they, in general, deal with scope of the new rules and it is important to describe languages as libraries, such as in SugarJ [Erdweg et al., 2011], Fortress [Allen et al., 2008] and Racket [Tobin-Hochstadt et al., 2011].

Imperative adaptable models have problems to deal with scope, because the grammar is global. Some imperative models, such as Wegbreit’s model or Evolving Grammar of Cabasino, do not allow rule removals, which makes it impossible to work with scope. The strategy of the declarative model of Christiansen, where each grammar adaptation is restricted to a specific branch of the parse tree, makes easy to define scope dependent relations. RAG of Shutt also has these same properties of Christiansen Adaptable Model and can check for multiple declarations, which is very difficult in Christiansen Adaptable Model.

As Christiansen and Shutt declarative models deal well with scope, they seem better than the others presented here for describing extensible languages. The Shutt model, however, is difficult to learn and to use it well, it is necessary a high level of mathematics background. When working with languages that allow the user to add new rules on the language grammar, it also important to deal with lexical extensions. All adaptable models studied do not treat lexical extensions. Also, we do not have any evidence indicating that parsers based on the adaptable models studied are reasonable efficient to define the syntax of real languages. Indeed, Erdweg et al. [2011] suggest

Figure 2.12 PEG for the language $\{a^n b^n c^n \mid n \geq 0\}$

$$\begin{aligned} S &\leftarrow !X a^* B !. \\ A &\leftarrow a A b / \lambda \\ B &\leftarrow b B c / \lambda \\ X &\leftarrow !(A !b) \end{aligned}$$

adaptable models as an alternative to implement the SugarJ language, but questioned their efficiency. The model developed in this work incorporates several ideas of Christiansen's Adaptable Grammars.

2.3 Parsing Expression Grammar

In a generative system, the syntax of languages is defined by a set of rules that are applied recursively to generate strings. In contrast, a recognition system has rules and predicates that decide if the string belongs to the language. To illustrate it, Ford [2004] gives an example of a language of a 's whose length is even. In a generative system style, this language is defined as $\{(aa)^n \mid n \geq 0\}$ and in a recognition system style as $\{s \in a^* \mid (|s| \bmod 2 = 0)\}$. The former definition *generates* strings by concatenation of two a 's, while the latter *tests* if a string of a 's is in the language by checking whether the rest of the division of its length by two is zero.

Ford [2004] argues that the most practical language applications in computer science involve recognition and structural decomposition of strings, however, the formal description of languages uses generative models instead of recognition. Ford developed *Parsing Expression Grammar* (PEG) to fill this gap.

PEG is a new model for describing the syntax of programming languages, which uses a recognition-based approach instead of a generative system. Similarly to CFG, formally a PEG is a 4-tuple (V_N, V_T, R, S) where V_N , V_T and S are as in a CFG and R is a rule function which maps nonterminals to *parsing expressions*.

A parsing expression is very similar to the right hand side of a CFG in the extended Backus-Naur form with the addition of new operators, the not-predicate (!) and the prioritized choice (/). The not-predicate operator checks if the string matches some syntax without consuming the input and the prioritized choice lists alternative patterns to be tested in order. As an example, Figure 2.12 shows a PEG that recognizes the language $\{a^n b^n c^n \mid n \geq 0\}$.

In order to understand how PEG works, let's see the recognition of the string $aabbcc$. The recognition process starts with the initial parsing expression S and the

entire input $aabbcc$. As S is a nonterminal, it is replaced by the corresponding parsing expression using the function R . So we have the pair $(!X a^* B !., aabbcc)$, representing the current parsing expression and the current input to be recognized. We use the notation $\frac{B}{A}$ from proof trees to express that, to satisfy A , one must first satisfy B . Then, our first goal is

$$\frac{(!X a^* B !., aabbcc)}{(S, aabbcc)}$$

The parsing expression $!X a^* B !.$ is a sequence of simpler parsing expressions, therefore we try to recognize the input following each smaller parsing expression, in order. So, first, we try to parse the input $aabbcc$ with the parsing expression $!X$, and, next, the remaining input with the other parsing expression, $a^* B !.$. We increment our notation to $(e, i) \Rightarrow o$ which says that the parsing expression e matches or consumes the prefix o of the input string i in the recognition process, then we can represent our goal as

$$\frac{(!X, aabbcc) \Rightarrow o_1 \quad (a^* B !., x) \Rightarrow o}{(!X a^* B !., aabbcc) \Rightarrow o_1 o}$$

Note that the input $aabbcc$ must be the concatenation of the prefix consumed by the parsing expression $!X$, o_1 , with it remaining, x .

The symbol $!$ is the not-predicate operator and the parsing expression $!X$ will succeed in this input if the parsing expression X fails in the same input. The not-predicate only tests if the input matches the syntax and does not consume input symbols, so the output of the not-predicate is the empty string λ . Our proof tree representing the recognition process improves to

$$\frac{\frac{(X, aabbcc) \Rightarrow f}{(!X, aabbcc) \Rightarrow \lambda} \quad (a^* B !., aabbcc) \Rightarrow o}{(!X a^* B !., aabbcc) \Rightarrow o}$$

where the symbol f represents a failure.

Next, to reach the goal $(X, aabbcc) \Rightarrow f$, we replace X by its corresponding parsing expression $!(A !b)$. Again, we have the not-predicate operator, then the parsing expression $A !b$ must succeed and consume some prefix of the input, then the overall parsing expression with the not-predicator operator will fail. The following proof tree shows this process

$$\frac{\frac{(A !b, aabbcc) \Rightarrow o_2}{(!A !b), aabbcc) \Rightarrow f}}{(X, aabbcc) \Rightarrow f}$$

The goal $(A !b, aabbcc) \Rightarrow o_2$ has a sequence of simpler parsing expressions, then it is divided into A and $!b$. Next, we try to match A and replace it by its parsing expression $a A b / \lambda$, resulting in the following proof tree

$$\frac{\frac{(a A b / \lambda, aabbcc) \Rightarrow o_1}{(A, aabbcc) \Rightarrow o_1} \quad (!b, i_1) \Rightarrow o}{(A !b, aabbcc) \Rightarrow o_2}$$

The parsing expression $a A b / \lambda$ uses the prioritized choice operator, $/$. Then, we try to match the input with the first choice $a A b$ and the second is tried only if the first reaches a failure. As the first parsing expression is a sequence, the proof tree for this goal is

$$\frac{\frac{(a, aabbcc) \Rightarrow a \quad (A b, abbcc) \Rightarrow o_1}{(a A b, aabbcc) \Rightarrow ao_1}}{(a A b / \lambda, aabbcc) \Rightarrow ao_1}$$

Note that when the parsing expression is a terminal, it succeeds only if the first symbol of the input is the same terminal and consumes it, otherwise fails. The interpretation $(a, aabbcc) \Rightarrow a$ shows this. The same idea explained above is used to parse the branch $(A b, abbcc) \Rightarrow o_1$, resulting in the following proof tree

$$\frac{\frac{\frac{\frac{(a, bbcc) \Rightarrow f}{(a A b, bbcc) \Rightarrow f} \quad (\lambda, bbcc) \Rightarrow \lambda}{(a A b / \lambda, bbcc) \Rightarrow \lambda}}{(A, bbcc) \Rightarrow \lambda} \quad (b, bbcc) \Rightarrow b}{(a, abbcc) \Rightarrow a \quad (A b, bbcc) \Rightarrow b} \quad \frac{(a A b, abbcc) \Rightarrow ab}{(a A b / \lambda, abbcc) \Rightarrow ab} \quad (b, bcc) \Rightarrow b}{(A, abbcc) \Rightarrow ab} \quad (A b, abbcc) \Rightarrow abb$$

Observe that, as the first symbol of the input is b in $(a, bbcc) \Rightarrow f$, the parsing expression fails and then the parsing expression $a A b$, which is a sequence, also fails. Therefore, the second alternative of $a A b / \lambda$ is tested and it succeeds, because the parsing expression λ always succeeds and does not consume any symbol of the input. As a result, the parsing expression $A b$ does not fail and consumes the prefix abb of the input.

Now, we can finish the proof tree of the parsing expression $a A b / \lambda$ on input $aabbcc$, which succeeds and consumes the prefix $aabb$ of the input. Below, we show the proof tree.

$$\frac{\frac{(a, aabbcc) \Rightarrow a \quad (A b, abbcc) \Rightarrow abb}{(a A b, aabbcc) \Rightarrow aabb}}{(a A b / \lambda, aabbcc) \Rightarrow aabb}$$

So, the proof tree of $(A !b, aabbcc) \Rightarrow o_2$ is

$$\frac{(a A b / \lambda, aabbcc) \Rightarrow aabb}{\frac{(A, aabbcc) \Rightarrow aabb \quad (!b, cc) \Rightarrow \lambda}{(A !b, aabbcc) \Rightarrow aabb}}$$

The $(!b, cc) \Rightarrow \lambda$ succeeds because the first symbol of the input is c and the expression b fails, so the not-predicate succeeds and does not consume any symbol of the input. Therefore, the proof tree of the start expression is showed below

$$\frac{\frac{\frac{(A !b, aabbcc) \Rightarrow aabb}{(!A !b, aabbcc) \Rightarrow f}}{(X, aabbcc) \Rightarrow f}}{(!X, aabbcc) \Rightarrow \lambda} \quad \frac{\frac{(a^*, aabbcc) \Rightarrow aa}{(a^* B !., aabbcc) \Rightarrow aabbcc}}{(!X a^* B !., aabbcc) \Rightarrow aabbcc} \quad \frac{\frac{\frac{\vdots}{(B, bbcc) \Rightarrow bbcc} \quad (!., \lambda) \Rightarrow \lambda}{(B !., bbcc) \Rightarrow bbcc}}{(S, aabbcc) \Rightarrow aabbcc}$$

The branch $(a^*, aabbcc) \Rightarrow aa$ matches aa because the operator star is greedy and matches all a 's until reaching the first b . Similar to the nonterminal A , B recognizes a sequence of b 's followed by c 's of the same length, so $(B !., bbcc) \Rightarrow bbcc$. PEG does not require that the entire input is consumed in order to recognize it. It is possible to succeed matching only a prefix of the input. So the parsing expression $!$ explicitly tests if there are additional symbols or if it is the end of the input (the dot symbol matches any input symbol).

In the example above, the language recognized, $\{a^n b^n c^n \mid n \geq 0\}$, is not context-free and, therefore, there is not a CFG that generates this language. Although PEG can recognize languages that are not context-free, left recursion is a problem in PEG, producing expressions that recognize no string at all. For example, the parsing expression $A a / \lambda$ does not recognize any string. This happens because the alternative is prioritized.

2.4 Data-Dependent Grammar

Different from almost all formalisms presented here, which have been designed to formally describe the syntax of languages, YAKKER was developed to be a better tool for parser generation [Jim et al., 2010]. Jim et al. [2010] argue that there are several applications that require parser generated by hand, because parser generators are not

suitable for them. Jim et al. [2010] give as example the SSL message data format and HTML 5.0 where the designers tried to use a parser generator, but they were forced to abandon this idea and built a parser by hand.

In their paper, Jim et al. [2010] present some languages that require special features usually not implemented by parser generators. These examples involve data formats, networking and web systems. The authors describe what are the required features and present the formalization of YAKKER. YAKKER is an extension of CFG with the addition of the following features:

- *Regular right hand side.* The right hand side of a rule is a regular expression. So, the rules of the YAKKER are a function that maps nonterminals to regular expressions;
- *binding.* YAKKER allows the storage of portions of the generated string in variables and uses this information in the derivation process;
- *semantic predicates and constraints.* The grammar also allows the definition of constraints which it is possible to test properties on the variables;
- *parameterized nonterminals.* The nonterminals may have one parameter, which acts like an inherited attribute. This parameter can be combined with predicates and variables to implement context-sensitive aspects.

We show these features by an example. Jim et al. [2010] give an example of a data language where the length of the data comes before the data. Below, we show a simplified version

$$literal8 \rightarrow \{ x := number (+ | \lambda) \} \{ n := string2int(x); \} \\ ([n > 0] OCTET \{ n := n - 1 \})^* [n = 0]$$

The terminal symbols are the braces in bold and the plus symbol. The names *number* and *OCTET* are nonterminals that generate a decimal number and a number in octal, respectively. This example illustrates several features of YAKKER. First, the variable x binds to the string derived by the nonterminal *number*, then the semantic predicate uses the string value of x to convert it into a number (function *string2int*) and binds the result to the variable n . The semantic predicates are in braces. The value of

the variable n is used in the constraint (within square brackets) to generate an octal number with n digits.

Another feature of YAKKER is parameterized nonterminals, allowing nonterminals to have one parameter. This parameter is similar to an inherited attribute and Jim et al. [2010] claim that it can be used to support modular reuse. For example, a fixed-width string may be defined as

$$\text{stringFW}(n) \rightarrow ([n > 0] \text{CHAR} \{n := n - 1\})^* [n = 0]$$

2.5 Conclusion

In this chapter, we have presented a survey on formalisms for describing the syntax of programming languages. The main motivation of these formalisms are the description of the full syntax of programming languages, including context-sensitive features. The models presented in Section 2.1 extend CFGs with facilities for expressing context-sensitive aspects of programming languages. The most successful model is AG and its extension EAG. They have been used for specifying the syntax of programming languages and in automatic compiler generation.

However, AG, EAG and the other models presented in Section 2.1 require a global and immutable grammar. Therefore, they are not suitable for describing the syntax of languages which require changes on their own set of rules during parsing, let alone for automatic parser generation for these languages. Section 2.2 surveys adaptable grammars, which are models with explicit manipulation of the grammar set of rules and that can handle these kinds of languages.

We have seen no evidence that the adaptable grammars studied are good for efficient parser generator. As an example, Erdweg et al. [2011] cite adaptable grammars as an alternative to implement the parser of the language SugarJ, but they question their efficiency. Adaptable grammars were initially proposed to give only a syntax treatment for context-sensitive dependencies of programming languages. The idea came from the observation that context-sensitive features of programming languages add new syntactic constructions to the language. However, adaptable grammars have been around at least 30 years and they have not been used in the definition of real programming languages. There are two reasons for that. First, there are some static semantic which is difficult to define with adaptable grammars, as checking for multiple declaration of variables, although this task can be easily accomplished using AGs and a symbol table. A second reason regards their unfitness for automatic parser generation and the low efficiency of parsers based on adaptable grammars.

Sections 2.3 and 2.4 presented PEG and YAKKER, which are new models for describing programming languages. Although PEG and YAKKER are not suitable for languages which modify themselves during parsing, they have been designed for parsing programming languages and have added features to deal with data, web and networking languages. The parsing algorithm for PEG and YAKKER are scannerless and have other features related to new requirements for parsing. Thus, we based our model, APEG, on PEG and Christiansen's Adaptable Grammar. We also borrow some features of YAKKER. Our decision to choose PEG as the base model is based on some standard features of PEG that are helpful for the description of context dependency, and for the efficiency of the implementation:

- PEG defines operators for checking an arbitrarily long prefix of the input, without consuming it. We show in Chapter 5 that this feature may allow a simple solution for specifying forward reference, which is important to define the Fortress language;
- The choice operator of PEG is ordered, giving more control of which alternative will be used. Our implementation relies on this feature to provide an implementation that does not need to build complex parsing tables whenever a rule is updated, favouring efficiency.

Chapter 3

Adaptable Parsing Expression Grammar

In this chapter, we present the formalization of our proposed model, *Adaptable Parsing Expression Grammar* (APEG). The design goals of the model are:

- it must be expressive enough for describing the syntax of any extensible language;
- it must favour legibility and simplicity; and
- it must be suitable for automatic generation of efficient syntactic analyzers.

APEG is an adaptable model based on PEG. It also uses the idea of attributes of AGs [Watt and Madsen, 1983] to guide parsing. APEG are L-attributed and their purposes are syntactic and not semantic as in AGs. The adaptability of APEG is provided in a way similar to Christiansen Adaptable Grammar. So, APEG also possesses a special attribute called *language attribute*, which represents the set of rules that are currently used. Language attribute values can be defined by means of embedded semantic actions and can be passed to different branches of the parse tree. This allows a formal representation of a set of syntactic rules that can be modified on-the-fly, i.e., during the parsing of an input string.

We have decided to define APEG according to these formalisms based on some of the reasons explained below. PEG has several features that facilitate parsing programming languages, such as unlimited lookahead, it is a recognition-based approach and it is scannerless. PEG also has a prioritized operator which allows free manipulation of the production rules without the insertion of undesirable ambiguities. Attribute systems have been used in compiler construction and language design, and they have shown to be a simple and elegant system with well known solutions for several context-sensitive

Figure 3.1 Attribute PEG for binary numbers

$$\begin{aligned} \langle S \uparrow x_0 \rangle &\leftarrow \langle T \uparrow x_0 \rangle \\ \langle T \uparrow x_0 \rangle &\leftarrow \langle B \uparrow x_0 \rangle (\langle B \uparrow x_1 \rangle [x_0 = 2 * x_0 + x_1])^* \\ \langle B \uparrow x_1 \rangle &\leftarrow (0 [x_1 = 0]) / (1 [x_1 = 1]) \end{aligned}$$

problems of programming languages. So, the adaptability of the model can be combined with the attribute system to define real extensible programming languages.

The remaining of this chapter explains APEG. In order to understand the formal definition of the model, it is necessary to know how attributes are evaluated and how constraints over them can be defined. First, we discuss our design decisions on how to combine PEG and attributes, called *Attribute Parsing Expression Grammar* or *Attribute PEG* (Section 3.1) and, next, we present how adaptability is achieved in the model (Section 3.2). In Section 3.3, we illustrate how the model works defining interesting features of programming languages using APEG. We end the chapter with the conclusions, in Section 3.4.

3.1 Attribute Parsing Expression Grammar

Attribute Parsing Expression Grammar (Attribute PEG) is an extension of PEG with attributes. The Attribute PEG uses the same notation of EAG and also includes explicit evaluation rules. Attribute PEG includes explicit rules for manipulating the attributes because the attribute expressions embedded in the rules are not powerful enough to replace all uses of explicit evaluation rules in PEGs. In PEGs, the use of recursion is frequently replaced by the use of the *repetition* operator “*”, giving definitions more closer to an imperative model. Therefore, we allow that evaluation rules in Attribute PEGs to update the values of the attribute variables, treating them as variables of an imperative language.

As an example for an informal introduction, Figure 3.1 shows an Attribute PEG which recognizes binary numbers and calculates their value. Expressions in brackets are explicit evaluation rules. In the third line, each of the options of the ordered choice has its own evaluation rule, defining that the value of the variable x_1 is either 0 (if the input is “0”) or 1 (if the input is “1”). It is not possible to replace these evaluation rules with attribute expressions, in the same way of EAGs, because the options are defined in a single parsing expression and it uses ordered choices. In the second line, the value of variable x_0 is initially defined on the first use of the nonterminal B . Then it is cumulatively updated by the evaluation rule $[x_0 = 2 * x_0 + x_1]$.

In Attribute PEGs, we also allow the use of constraints, as predicates defined in

any position on the right hand side of a rule. If a predicate fails, the evaluation of the parsing expression also fails. The use of attributes as variables of an imperative language and predicate evaluation are similar to the approach adopted for the formal definition of YAKKER. We also include the “bind expression”, featured defined by YAKKER, in Attribute PEG, that allows binding a prefix of the input string to a variable. This allows concise specification of languages.

Another improvement provided by EAG is the possibility of using the same attribute variable in more than one defining position in a rule. Repeated occurrence of the variable name of an attribute defines an implicit constraint, requiring the variable to have the same value in all instances. In our proposition for Attribute PEG, we do not adopt this last improvement of EAG, because it would not be consistent with our design decision of allowing attributes to be updated as variables of an imperative language. In the following, we formally define the syntax of Attribute PEG.

Definition 1 (Attribute System). *An attribute system is a 4-tuple (V_N, A, V_A, F) , where V_N is a set of nonterminal names, A is a function that maps a symbol of V_N to $\mathbb{N} \times \mathbb{N}$, V_I is a set of variable names disjoint from V_N , and F is a set of functions.*

An attribute system defines a specification of the attributes of every nonterminal, defining the numbers of inherited and synthesized attributes each nonterminal has, in a definition of an Attribute PEG. For example, the definition of the function A for the Attribute PEG in the example of binary numbers presented in Figure 3.1 is

$$\begin{aligned} A(S) &= (0, 1) \\ A(T) &= (0, 1) \\ A(B) &= (0, 1) \end{aligned}$$

which says that every nonterminal has only one synthesized attribute. If the nonterminal S had two attributes where one is inherited and another synthesized, then the definition of the function A for S would be $A(S) = (1, 1)$. We also define $arity(N)$ as the number of attributes of a nonterminal N , i.e., the sum of the two numbers of the tuple $A(N)$.

From an attribute system, we also define a simple, untyped language of *attribute expressions*, which are expressions over the attribute domains, using functions from the set F and variables. The attribute expression language includes variables from V_A , boolean, integer and string literals. If $f \in F$ is a function of arity n and $\varepsilon_1, \dots, \varepsilon_n$ are attribute expressions, then $f(\varepsilon_1, \dots, \varepsilon_n)$ is also an attribute expression.

The representation of a nonterminal in Attribute PEG is done in angular brackets using the name of the nonterminal followed by a list of attribute expressions with a symbol to identify if the attribute is inherited (\downarrow) or synthesized (\uparrow). As an example, the nonterminal T , which has only one synthesized attribute, is used at the rule of the nonterminal S in Figure 3.1, represented by $\langle T \uparrow x_0 \rangle$ (the nonterminal name followed by the up arrow and one attribute expression, x_0).

Definition 2 (Parsing Expression). *Suppose an attribute system (V_N, A, V_A, F) , and a set of terminals, V_T , disjoint from $V_N \cup V_A$. Let e , e_1 and e_2 be parsing expressions, ε_i attribute expressions, \dagger an element of the set $\{\downarrow, \uparrow\}$ and $\vartheta \in V_A$ a variable name. Then, the set of valid parsing expressions (\mathcal{P}_e) is recursively defined as:*

$$\begin{aligned}
& \lambda \in \mathcal{P}_e && \text{(empty expression)} \\
& a \in \mathcal{P}_e, \text{ for every } a \in V_T && \text{(terminal expression)} \\
& \langle A \dagger \varepsilon_1 \dots \dagger \varepsilon_p \rangle \in \mathcal{P}_e, \text{ for every } A \in V_N && \text{(nonterminal expression)} \\
& \text{and } p = \text{arity}(A) && \\
& e_1 e_2 \in \mathcal{P}_e && \text{(sequence expression)} \\
& e_1 / e_2 \in \mathcal{P}_e && \text{(ordered choice expression)} \\
& e^* \in \mathcal{P}_e && \text{(zero-or-more repetition expression)} \\
& !e \in \mathcal{P}_e && \text{(not-predicate expression)} \\
& [\vartheta = \varepsilon] \in \mathcal{P}_e && \text{(update expression)} \\
& [\varepsilon] \in \mathcal{P}_e && \text{(constraint expression)} \\
& \vartheta = e \in \mathcal{P}_e && \text{(bind expression)}
\end{aligned}$$

The first seven types of parsing expressions are standard to PEG. Only *nonterminal expressions* are different because of the use of attributes. To this set of *parsing expressions*, we add three new types of expressions. *Update expressions* have the format $[\vartheta = \varepsilon]$, where ϑ is a variable name and ε is an attribute expression. They are used to update the value of variables in an environment. *Constraint expressions* with the format $[\varepsilon]$, where ε is an attribute expression that evaluates to a boolean value, are used to test for predicates over the attributes. *Bind expressions* have the format $\vartheta = e$, where ϑ is a variable and e is a parsing expression. It assigns to ϑ the part of the input that the parsing expression e matches. If there is no match, the variable ϑ is unbounded.

Definition 3 (Attribute PEG). *Let H be the set of all n -tuples of attribute expressions. An Attribute PEG is a 7-tuple $(V_N, V_T, A, R, S, V_A, F)$, where V_N and V_T are finite sets of nonterminals and terminals, respectively. (V_N, A, V_A, F) forms an attribute system*

and $R : V_N \rightarrow \mathcal{P}_e \times H$ is a total rule function which maps every nonterminal name to a pair of a parsing expression and the attribute expressions of the nonterminal. S is the initial parsing expression.

Figure 3.1 shows the rule function in a sugared syntactic notation. In this notation, we put together the nonterminal and its attributes on the left side and the parsing expression on the right side. We will use this notation throughout the text. The example of Figure 3.1 can be expressed formally as $G = (\{S, T, B\}, \{\mathbf{0}, \mathbf{1}\}, \{(S, (0, 1)), (T, (0, 1)), (B, (0, 1))\}, R, S, \{x_0, x_1\}, \{+, *\})$, where R is formally defined as

$$\begin{aligned} R(S) &= (\langle T \uparrow x_0 \rangle, x_0) \\ R(T) &= (\langle B \uparrow x_0 \rangle \langle B \uparrow x_1 \rangle [x_0 = 2 * x_0 + x_1]^*, x_0) \\ R(B) &= ((0 [x_1 = 0]) / (1 [x_1 = 1]), x_1) \end{aligned}$$

We make the restriction that the attribute expressions in defining positions (inherited attributes on the left side of a rule and synthesized attributes on the right side of a rule) are always represented by a single variable. Without loss of generality, we will also assume that all inherited attributes are represented in a nonterminal before its synthesized attributes. So, suppose that e is the parsing expression associated with nonterminal A , p is its number of inherited attributes and q the number of synthesized attributes. Then $\langle A \downarrow \vartheta_1 \downarrow \vartheta_2 \dots \downarrow \vartheta_p \uparrow \varepsilon_1 \uparrow \dots \uparrow \varepsilon_q \rangle \leftarrow e$ represents the rule for A and its attributes.

In this thesis, we use the following convention for attribute expressions, parsing expressions, strings and values:

- the Greek letters ε , ϑ and κ represent attribute expressions. The letters ϑ and κ are used only in defining positions, which must be a variable name;
- the letter e represents parsing expressions;
- the letters x , y and w represent strings; and
- letters v and u are used for the semantic domain of attribute expressions, i.e., representing the value of the evaluation of an attribute expression.

All symbols above may be decorated with prime and/or an integer index. We also use the simplified notation \vec{v}_n to denote a sequence v of n expressions. As defining positions of a rule are represented by the letters ϑ and κ and applying positions by the letters ε , the $\langle A \downarrow \vartheta_1 \downarrow \dots \downarrow \vartheta_p \uparrow \varepsilon_1 \uparrow \dots \uparrow \varepsilon_q \rangle \leftarrow e$ may be replaced by $\langle A \downarrow \vec{\vartheta}_p \uparrow \vec{\varepsilon}_q \rangle \leftarrow e$. On the other hand, if the same symbol is used in the right side of a rule, then the notation

for a nonterminal expression $\langle A \downarrow \varepsilon'_1 \downarrow \dots \downarrow \varepsilon'_p \uparrow \kappa_1 \uparrow \dots \uparrow \kappa_q \rangle$ may be replaced by $\langle A \downarrow \vec{\varepsilon}'_p \uparrow \vec{\kappa}_q \rangle$.

3.2 Adaptable Parsing Expression Grammar

The adaptability of APEGs is achieved by means of an attribute associated with every nonterminal, representing the current grammar. Therefore, syntactically, APEG is an Attribute PEG in which the first attribute of all nonterminals is inherited and represents the *language attribute* (the set of parsing expression rules that may be used). In this section, we define the semantics of APEG. For this purpose, first we define the notion of an environment.

Definition 4 (Environment). *An environment is a function that maps variables to values. The following notation is used for constructing environments:*

- $.$ (a dot) represents an empty environment, i.e., all variables map to the unbound value;
- $[\vartheta_1/v_1, \dots, \vartheta_n/v_n]$ maps ϑ_i to v_i , $1 \leq i \leq n$, and other variables to unbound;
- $E[\vartheta_1/v_1, \dots, \vartheta_n/v_n]$ is an environment which is equal to E , except for the values of ϑ_i that map to v_i , $1 \leq i \leq n$.

We write $E[[\varepsilon]]$ to indicate the value of the attribute expression ε evaluated in the environment E . For example, suppose an environment $E = [x/5; y/7; z/2]$, then $E[[x]]$ is equal to 5 and $E[[x * z + y]]$ is equal to 17. If we define the environment $Y = E[x/3; w/6]$, then $Y[[x]]$ is equal to 3 and $Y[[w * x + z]]$ is 20. Note that the evaluation of a variable that is not defined results in the *unbound* value, for example $E[[w]] = \text{unbound}$.

Figures 3.2 and 3.3 present the semantics of an APEG. Almost all the formalization is related to Attribute PEG. Only the last equation in Figure 3.3 defines adaptability. Figures 3.2 and 3.3 define the judgement $E \vdash (e, x) \Rightarrow o \vdash E'$, which says that the interpretation of the parsing expression e , for the input string x , in an environment E , results in o , and produces a new environment E' . The result $o \in V_T^*$ indicates the prefix of x that is consumed, if the expression succeeds, or $f \notin V_T^*$, if it fails.

The equations in Figure 3.2 describe the semantics of regular PEG operations, and their relation to environments. Note that changes in an environment are discarded when an expression fails. For example, in a sequence expression, a new environment

Figure 3.2 Semantics of Adaptable PEG - Part I.

$$\boxed{E \vdash (e, x) \Rightarrow o \vdash E'}$$

$$\mathbf{Empty} \frac{}{E \vdash (\lambda, x) \Rightarrow \lambda \vdash E}$$

$$\mathbf{Term} \frac{a \in V_T}{E \vdash (a, ax) \Rightarrow a \vdash E}$$

$$\mathbf{-Term}_1 \frac{a, b \in V_T \quad a \neq b}{E \vdash (a, bx) \Rightarrow f \vdash E}$$

$$\mathbf{-Term}_2 \frac{a \in V_T}{E \vdash (a, \lambda) \Rightarrow f \vdash E}$$

$$\mathbf{Seq} \frac{E_1 \vdash (e_1, x_1x_2y) \Rightarrow x_1 \vdash E_2 \quad E_2 \vdash (e_2, x_2y) \Rightarrow x_2 \vdash E_3}{E_1 \vdash (e_1e_2, x_1x_2y) \Rightarrow x_1x_2 \vdash E_3}$$

$$\mathbf{-Seq}_1 \frac{E_1 \vdash (e_1, xy) \Rightarrow x \vdash E_2 \quad E_2 \vdash (e_2, y) \Rightarrow f \vdash E_3}{E_1 \vdash (e_1e_2, xy) \Rightarrow f \vdash E_1}$$

$$\mathbf{-Seq}_2 \frac{E \vdash (e_1, x) \Rightarrow f \vdash E'}{E \vdash (e_1e_2, x) \Rightarrow f \vdash E}$$

$$\mathbf{Choice}_1 \frac{E \vdash (e_1, xy) \Rightarrow x \vdash E'}{E \vdash (e_1/e_2, xy) \Rightarrow x \vdash E'}$$

$$\mathbf{Choice}_2 \frac{E \vdash (e_1, xy) \Rightarrow f \vdash E_1 \quad E \vdash (e_2, xy) \Rightarrow x \vdash E_2}{E \vdash (e_1/e_2, xy) \Rightarrow x \vdash E_2}$$

$$\mathbf{-Choice} \frac{E \vdash (e_1, x) \Rightarrow f \vdash E_1 \quad E \vdash (e_2, x) \Rightarrow f \vdash E_2}{E \vdash (e_1/e_2, x) \Rightarrow f \vdash E}$$

$$\mathbf{Rep} \frac{E_1 \vdash (e, x_1x_2y) \Rightarrow x_1 \vdash E_2 \quad E_2 \vdash (e^*, x_2y) \Rightarrow x_2 \vdash E_3}{E_1 \vdash (e^*, x_1x_2y) \Rightarrow x_1x_2 \vdash E_3}$$

$$\mathbf{-Rep} \frac{E_1 \vdash (e, x) \Rightarrow f \vdash E_2}{E_1 \vdash (e^*, x) \Rightarrow \lambda \vdash E_1}$$

$$\mathbf{Neg} \frac{E \vdash (e, xy) \Rightarrow x \vdash E'}{E \vdash (!e, xy) \Rightarrow f \vdash E'}$$

$$\mathbf{-Neg} \frac{E \vdash (e, x) \Rightarrow f \vdash E'}{E \vdash (!e, x) \Rightarrow \lambda \vdash E'}$$

Figure 3.3 Semantics of Adaptable PEG - Part II.

$$\boxed{E \vdash (e, x) \Rightarrow o \vdash E'}$$

$$\begin{array}{c}
\mathbf{Atrib} \frac{v = E[\varepsilon]}{E \vdash ([\kappa = \varepsilon], x) \Rightarrow \lambda \vdash E[\kappa/v]} \qquad \neg\mathbf{Atrib} \frac{unbound = E[\varepsilon]}{E \vdash ([\kappa = \varepsilon], x) \Rightarrow f \vdash E} \\
\mathbf{Bind} \frac{E \vdash (e, xy) \Rightarrow x \vdash E'}{E \vdash (\kappa = e, xy) \Rightarrow x \vdash E'[\kappa/x]} \qquad \neg\mathbf{Bind} \frac{E \vdash (e, x) \Rightarrow f \vdash E'}{E \vdash (\kappa = e, x) \Rightarrow f \vdash E} \\
\mathbf{True} \frac{true = E[\varepsilon]}{E \vdash ([\varepsilon], x) \Rightarrow \lambda \vdash E} \qquad \mathbf{False} \frac{false = E[\varepsilon]}{E \vdash ([\varepsilon], x) \Rightarrow f \vdash E} \\
\langle A \downarrow \vartheta_1 \downarrow \dots \downarrow \vartheta_p \uparrow \varepsilon'_1 \uparrow \dots \uparrow \varepsilon'_q \rangle \leftarrow e \in E[\varepsilon_1], \text{ where } E[\varepsilon_1] \equiv \text{language attribute} \\
v_i = E[\varepsilon_i], 1 \leq i \leq p \quad u_j = E'[\varepsilon'_j], 1 \leq j \leq q \\
\mathbf{Adapt} \frac{[\vartheta_1/v_1, \dots, \vartheta_p/v_p] \vdash (e, x) \Rightarrow o \vdash E'}{E \vdash (\langle A \downarrow \varepsilon_1 \downarrow \dots \downarrow \varepsilon_p \uparrow \kappa_1 \uparrow \dots \uparrow \kappa_q \rangle, x) \Rightarrow o \vdash E[\kappa_1/u_1, \dots, \kappa_q/u_q]}
\end{array}$$

is computed when it succeeds, a situation represented by rule **Seq**. If the first or the second subexpression of a sequence expression fails, the changes are discarded and the environment used is the one before the sequence expression. These situations are represented by rules $\neg\mathbf{Seq}_1$ and $\neg\mathbf{Seq}_2$. A similar behaviour is defined for $\neg\mathbf{Term}_1$ and $\neg\mathbf{Term}_2$, when a terminal expression fails, for $\neg\mathbf{Rep}$, when a repetition fails, and for **Choice**₂, when the first alternative of a choice fails. Rules **Neg** and $\neg\mathbf{Neg}$ are the only one that change the environment in the outer level, even in a fail situation. The change in the **Neg** rule allows arbitrary lookahead, calculating some information about the prefix and using it later. To allow simulating positive predicate which changes in the environment (similar to the semantics of the not-predicate), we allow the $\neg\mathbf{Neg}$ rule to change the environment. Although this decision enables a collateral effect in the outer level of a not-predicate, this effect is not propagated outside a sequence parsing expression because the other rules do not allow changing in the environment in a fail situation.

The equations in Figure 3.3 describe the semantics of the operations that our model added to the PEG model, namely *update*, *constraint* and *bind* expressions, and also the semantics of the evaluation of *nonterminal expressions*. Rules **Atrib** and $\neg\mathbf{Atrib}$ define the behaviour for update expression, and rules **True** and **False** represent predicate evaluation in constraint expressions. Rules **Bind** and $\neg\mathbf{Bind}$ match the prefix of the input with the given parsing expression and store this prefix in a variable, if the match succeeds. The most interesting rule is **Adapt**. It defines how nonterminal expressions are evaluated, considering attributes and also the current set of production

rules, represented by the language attribute. Attribute values are associated with variables using an approach similar to EAG, but in a way more operational; it is also similar to *parametrized nonterminals* of YAKKER described in [Jim et al., 2010], but allowing several return values instead of just one. When a nonterminal is processed, the value of its inherited attributes are calculated considering the current environment. The corresponding parsing expression is fetched from the current set of production rules, defined by the language attribute, which is always the first attribute of the symbol. In rule **Adapt**, the language attribute is represented by expression e_1 , and R_{e_1} represents the map from nonterminals to parsing expressions from the adaptable PEG defined by e_1 . Rule **Adapt** is the only point in all the rules of Figures 3.2 and 3.3 that is associated with the property of adaptability.

We define the language accepted by an APEG as follows. Let $G = (V_N, V_T, A, R, S, V_A, F)$ be an APEG. Then

$$L(G) = \{w \in V_T^* \mid \cdot \vdash (\langle S \downarrow G \rangle, w) \Rightarrow w' \vdash E' \wedge w' \neq f\}$$

The derivation process begins using an empty environment, with the starting parsing expression S matching the input string w . The original grammar G is used as the value for the inherited language attribute of S . If the process succeeds, w' is the prefix of w matched and E' is the resulting environment. The language $L(G)$ is the set of words w that do not produce f (failure).

3.3 APEG in Action

In this section, we present two examples of use of APEG. The first example is a definition of context-dependent constraints commonly required in binary data specification. The second illustrates a specification of static semantics of programming languages.

3.3.1 Data Dependent Languages

As a motivating example of a context-sensitive language specification, Jim et al. [2010] present a data format language in which an integer number is used to define the length of the expression that follows it, between brackets. For instance, a valid sentential form is “6[abcdef]”.

Figure 3.4 shows how a similar language may be defined in an Attribute (non adaptable) PEG. The nonterminal *number* has a synthesized attribute, whose value is used in the constraint expression that controls the length of text to be parsed in the

Figure 3.4 An Attribute PEG for a data dependent language.

$$\begin{aligned}
\langle literal \rangle &\leftarrow \langle number \uparrow n \rangle \text{ "[" } \langle strN \downarrow n \rangle \text{ "]" } \\
\langle strN \downarrow n \rangle &\leftarrow ([n > 0] \text{ CHAR } [n = n - 1])^* [n == 0] \\
\langle number \uparrow x_2 \rangle &\leftarrow \langle digit \uparrow x_2 \rangle (\langle digit \uparrow x_1 \rangle [x_2 = x_2 * 10 + x_1])^* \\
\langle digit \uparrow x_1 \rangle &\leftarrow \mathbf{0} [x_1 = 0] / \mathbf{1} [x_1 = 1] / \dots / \mathbf{9} [x_1 = 9]
\end{aligned}$$

Figure 3.5 Using adaptability in the PEG of Figure 3.4.

$$\begin{aligned}
\langle literal \downarrow g \rangle &\leftarrow \langle number \downarrow g \uparrow n \rangle \text{ "[" } \\
&\quad [g_1 = g \oplus \text{ rule}(\langle strN \downarrow g \rangle \leftarrow + \text{rep}(\text{"CHAR"}, n))] \\
&\quad \langle strN \downarrow g_1 \rangle \text{ "]" }
\end{aligned}$$

sequel. In line 2, the terminal *CHAR* represents any single character. The value of the inherited attribute n is used in the update expression $[n = n - 1]$, inside a repetition expression. Each time a character is read, the value of n is decreased. There are also two constraint expressions. The first one checks if the value of n is non negative, inside the repetition expression, ensuring that the loop will end when n is zero. The second constraint expression, after the repetition expression, ensures that the number of characters read is exactly the value initially set to n , when $\langle strN \downarrow n \rangle$ was evaluated.

Using features from APEG in the same language, we could replace the first two rules of Figure 3.4 by the rule in Figure 3.5. In an APEG, every nonterminal has the language attribute as its first inherited attribute. The attribute g of the start symbol is initialized with the original APEG, but when nonterminal *strN* is used, a new grammar g_1 is considered. The symbol " \oplus " represents an operator for adding rules to a grammar and function *rep* produces a string repeatedly concatenated. The formalization of these functions, specially how to change an existing grammar, is discussed in Chapter 4 which treats about the implementation of the APEG model. Using these functions, g_1 will be equal to g together with a new rule that indicates that *strN* can generate a string with length n . For example, if n is 3, then the following rule will be added: $\langle strN \downarrow g \rangle \leftarrow \text{CHAR CHAR CHAR}$.

3.3.2 Static Semantics

Figure 3.6 presents a PEG definition of a toy block structured language in which a block consists of a list of declarations of integer variables, followed by a list of assignment statements. An assignment statement consists of a variable on the left side and a variable on the right side. For simplicity, the whitespaces are not considered.

Suppose that the context dependent constraints are: a variable cannot be used

Figure 3.6 Syntax of block with declaration and use of variables (simplified).

$\begin{aligned} \textit{block} &\leftarrow \{ \textit{dlist} \textit{slist} \} \\ \textit{dlist} &\leftarrow \textit{decl} \textit{decl}^* \\ \textit{slist} &\leftarrow \textit{stmt} \textit{stmt}^* \end{aligned}$		$\begin{aligned} \textit{decl} &\leftarrow \mathbf{int} \textit{id} ; \\ \textit{stmt} &\leftarrow \textit{id} = \textit{id} ; \\ \textit{id} &\leftarrow \textit{alpha} \textit{alpha}^* \end{aligned}$
--	--	--

Figure 3.7 Adaptable PEG for declaration and use of variables.

$\langle \textit{block} \downarrow g \rangle$	\leftarrow	$\{ \langle \textit{dlist} \downarrow g \uparrow g_1 \rangle \langle \textit{slist} \downarrow g_1 \rangle \}$
$\langle \textit{dlist} \downarrow g \uparrow g_1 \rangle$	\leftarrow	$\langle \textit{decl} \downarrow g \uparrow g_1 \rangle [g = g_1] (\langle \textit{decl} \downarrow g \uparrow g_1 \rangle [g = g_1])^*$
$\langle \textit{decl} \downarrow g \uparrow g_1 \rangle$	\leftarrow	$!(\mathbf{int} \langle \textit{var} \downarrow g \rangle) \mathbf{int} \langle \textit{id} \downarrow g \uparrow n \rangle ;$ $[g_1 = g \oplus \textit{rule}(\langle \textit{var} \downarrow g \rangle \leftarrow \#n \textit{!}(\textit{alpha} \downarrow g))]$
$\langle \textit{slist} \downarrow g \rangle$	\leftarrow	$\langle \textit{stmt} \downarrow g \rangle \langle \textit{stmt} \downarrow g \rangle^*$
$\langle \textit{stmt} \downarrow g \rangle$	\leftarrow	$\langle \textit{var} \downarrow g \rangle = \langle \textit{var} \downarrow g \rangle ;$
$\langle \textit{id} \downarrow g \uparrow s \rangle$	\leftarrow	$s = (\langle \textit{alpha} \downarrow g \rangle \langle \textit{alpha} \downarrow g \rangle^*)$
$\langle \textit{alpha} \downarrow g \rangle$	\leftarrow	$a / \dots / Z / 0 / \dots / 9 / _$

if it has not been declared before, and a variable cannot be declared more than once. The APEG in Figure 3.7 implements these context-dependent constraints.

In this example, the idea of implementing the context-dependent constraints is to adapt the nonterminal *var* on the fly in order to allow only declared variables to be recognized. Note that, in the beginning, the nonterminal *var* does not recognize any symbols (there is no definition to it at the beginning). However, when a variable is declared (nonterminal *decl*), a new grammar rule is produced by the addition of a new choice in the definition of nonterminal *var*, which allows the recognition of the new variable name. The resulting new grammar is passed as the language attribute, in the definition of the nonterminal *block*, to the nonterminal *slist*, and, in the sequel, to *stmt*. As a result, the nonterminal *stmt* now can recognize the declared variable.

For example, suppose the input string $\{\textit{int} \textit{a}; \textit{int} \textit{b}; \textit{a}=\textit{b}; \textit{b}=\textit{a};\}$. The recognition of this string starts with the nonterminal *block* and its language attribute is the grammar in Figure 3.7. After recognizing the first symbol, $\{$, the parser proceeds to recognize a list of declarations (nonterminal *dlist*), passing down the same grammar as the language attribute to the nonterminal *dlist*. During the recognition of the nonterminal *dlist*, it first tries to match a variable declaration through the nonterminal *decl*, passing to it the same language attribute. The nonterminal *decl* first checks if the variable is already declared using the parsing expression “ $!(\mathbf{int} \langle \textit{var} \downarrow g \rangle)$ ”. The not-lookahead operator, $!$, succeeds if the expression enclosed in parentheses fails, and it does not consume any symbol from the input. In order to check whether the variable **a** has already been declared, the parsing expression enclosed in parentheses matches the **int** string, but

the nonterminal *var* does not recognize the variable **a**, because it does not have any rule for it yet. In the sequel, the parsing expression “**int** $\langle id \downarrow g \uparrow n \rangle$;” recognizes the declaration of variable **a**. Note the use of the nonterminal *id* instead of the nonterminal *var*. The nonterminal *id* is used here to recognize any valid variable name, since it is a new one. The symbol “#” indicates that the identifier following it (*n*, in this case) must be treated as a variable inside a string, then, a new grammar is built from the current grammar by the addition of a new choice, $\langle var \downarrow g \rangle \leftarrow \mathbf{a} ! \langle alpha \downarrow g \rangle$, on the definition of nonterminal *var*. This new grammar becomes the value of the synthesized attribute *g1*.

The grammar synthesized by the nonterminal *decl* is used in the nonterminal *dlist* as language attribute of other calls of the nonterminal *decl*. Proceeding, the next variable declaration will be recognized, and the nonterminal *dlist* synthesizes a new grammar with these two choices, in this order, “**a** ! $\langle alpha \downarrow g \rangle$ ” and “**b** ! $\langle alpha \downarrow g \rangle$ ”, for the nonterminal *var*. This new grammar is used by the nonterminal *block* to pass it as the language attribute of the nonterminal *slist*. As a result, the two statements, $a = b$ and $b = a$, can be recognized, because the nonterminal *var* in the language attribute passed to the nonterminal *stmt* has rules to recognize the variables **a** and **b**.

The use of the operator “!” on the rule defining *decl* prevents multiple declarations, a problem reported as very difficult to solve when using adaptable models based on CFG [Christiansen, 1990; Carmi, 2010]. The new rule added to the current APEG ensures that a variable may be used only if it was previously declared. The symbol *block* may be part of a larger APEG, with the declarations restricted to the static scope defined by the block.

3.4 Conclusion

In order to define APEG, an adaptable model based on PEG, we specified how to combine PEG and attributes, which we called Attribute PEG. When we have developed our formalization for Attribute PEG, we were unaware of the Mercer [2008] work which also gave a formalization to combine attributes and PEG. The semantics of both formalization are equivalent, although when we published our first formalization of APEG [Reis et al., 2012], the semantics of the not-predicate was different and did not allow to propagate changes to the outer level environment. Defining the syntax of the extensible language Fortress, we realize it is important to allow the not-predicate to propagate changes. So, we have modified the semantics of the not-predicate, resulting in a semantics equivalent to the one on Mercer [2008]’work.

We defined APEG on top of Attribute PEG and achieved adaptability using a special inherited attribute present at every nonterminal that represents the current set of rules which can be used in every step of the recognition of the input.

APEG keeps some of the most important advantages of declarative adaptable models, such as an easy definition of context-dependent aspects associated to static scope and nested blocks. We specified two examples using APEG in order to illustrate how it works. We showed that the use of PEG as the basis for the model allowed a very simple solution for the problem of checking for multiple declarations of an identifier. This problem is reported as very difficult to solve with adaptable models based on CFG [Christiansen, 1990; Carmi, 2010]. Although we do not intend to define static semantics of programming languages using APEG, the example in Section 3.3.2 shows that the lookahead operator could be useful when forward analysis of the input is required. Indeed, we take advantages of this feature to simulate a multi-pass approach to parse the Fortress language.

In order to know exactly the adaptations performed by an APEG, a developer must be aware that it works as a top down parser. It could be considered as a disadvantage when compared to declarative models, but any PEG developer is already prepared to deal with this feature, since PEG is, by definition, a description of a top down parser.

Chapter 4

Implementation of Adaptable Parsing Expression Grammars

In this chapter, we discuss issues related to the difficulties of implementing the APEG model. Compared to the standard PEG model, the main new features of APEG are:

- introduction of inherited and synthesized attributes, creating “PEG with attributes”;
- introduction of update and constraint parsing expressions to compute and test attribute values;
- adaptability of the model, allowing changes in the grammar while the input is processed.

Ford developed *packrat parsers* [Ford, 2002b], which can be used for efficient implementation of PEGs. Packrat parsers allow unlimited lookahead with linear time processing, at the cost of additional storage for memoization. We have developed an implementation for the APEG model that is also based on the features of packrat parsers, but we had to deal with challenges not faced by a standard PEG. For instance, the use of attributes requires a more sophisticated memoization mechanism, since a nonterminal symbol may be called with different values for inherited attributes, resulting in different behaviours. Adaptability is another challenge, because changes in the set of rules may invalidate some memoization results.

In the following, we present a discussion on tools implementing ideas related to the PEG model (Section 4.1). We describe details of our own implementation of an interpreter for APEG model, analysing the impacts of the new proposed features (Section 4.2). We prove that the mechanism of memoization can be adapted to the

APEG model (Section 4.3). We discuss a mixed approach of code generation and interpretation for APEG grammars (Section 4.4). Then, we conclude this chapter (Section 4.5).

4.1 PEG-related implementations

As stated before, Ford developed *packrat parsing*, a method for implementing linear-time top-down parsers [Ford, 2002b]. Packrat parsing provides better composition properties than LL/LR parsing, making it more suitable for dynamic or extensible languages. The main disadvantage of the method is the storage cost, which may be a product of the number of nonterminal symbols by the total input size, rather than being proportional to the height of a syntax tree built during the parsing process. Ford implemented a packrat parser generator named Pappy [Ford, 2002a], which takes declarative parser specifications and generates packrat parsers in Haskell. The specification language accepted by Pappy allows computing semantic values with the help of embedded code fragments written in Haskell. It also allows the definition of *semantic predicates*, with parsing decisions depending on semantic values and not just on syntactic rules. An important disadvantage of this implementation is that the generated parsers need to have the entire input available up-front, thus making them unusable for interactive applications. No features for conveniently extending the set of rules are provided.

Redziejowski [2009] developed Mouse, a parser generator based on PEG. Using a PEG, Mouse generates Java code for a recursive descent parser. It offers limited resources for memoization, hence the name Mouse instead of Packrat. Mouse does not allow the explicit use of attributes. Semantic actions are represented by tags that can only be inserted at the end of each alternative of a rule. The code for the semantic actions is written as a method in a separate Java class. Methods are bound to semantic actions using the same name as the tags. Inside these semantic actions, it is possible to associate values with the symbols of a rule. This scheme allows a peculiar implementation of synthesized attributes, but not inherited attributes.

In *Language Generator by Instil* (LGI) [nez Guzmán, 2009], instead of generating a descent recursive parser, a PEG is represented as an *Abstract Syntax Tree* (AST) that is interpreted. Interpretation allows more flexibility, but it is less efficient than generated code. LGI implements full memoization, but again the memory management is very inefficient. The input must be completely read before processing and stored in memory. Memoization is implemented using a two-dimensional array, with a line for

each nonterminal, and a column for each input symbol. There is no way to define values for attributes, nor semantic actions. The result of processing an input is a syntax tree created to represent this input.

Instead of generating a descent recursive parser, LPEG [Ierusalimsky, 2009] translates parsing expressions to a virtual parsing machine which interprets them. LPEG can capture text and execute semantic actions, but does not allow explicit use of attributes. LPEG also does not use memoization of intermediate results, so it can have an exponential behavior. However, as LPEG is intended to be used in pattern matching applications, the exponential behavior is not expected to be common in practice. Moreover, the simplicity of the parsing machine allows the implementation of several optimizations.

ANTLR [Parr and Quong, 1994] is a parser generator that introduced a novel parsing approach, called LL(*) [Parr and Fisher, 2011]. It is not exactly PEG-based, but it implements some PEG and packrat parser features. For instance, it offers a lookahead operator and backtracking with memoization, which can be controlled by the programmer. ANTLR allows the use of inherited and synthesized attributes and semantic actions as embedded code. It generates efficient code, representing each nonterminal symbol as a function, in a recursive descent style. Inherited attributes are implemented as function parameters, and synthesized attributes are implemented as function return values. When combining backtracking and memoization with arbitrary semantic actions, an important issue arises: a memoized result is no longer valid if the semantic actions have collateral effects. ANTLR deals with this problem separating the process of defining which rule will be used in two steps. First, semantic actions and memoization are turned off and it uses lookahead to define the correct rule to be used. Then semantic actions are turned on and the parsing rules are processed again.

Rats! [Grimm, 2006] is another PEG based tool for parser generation, building packrat parsers with full memoization. The input is treated as a stream of characters, so it does not suffer from the important limitations imposed by Pappy. All optimizations introduced by Pappy and several others are implemented in Rats!, producing efficient parsers. A global state object can be managed by semantic actions without violating memoization, thus preserving linear time performance. In order to accomplish this, Rats! requires that all state modifications be performed within possibly nested, lightweight transactions, with some additional restrictions. Rats! allows the construction of extensible and modular grammars that may offer a great level of reuse. To provide reuse of grammars, it is possible to import modules and extend their production rules. Extensibility is carried out only statically. As all other tools analysed in this section, it is not possible to modify the set of production rules at parse time.

Katahdin [Seaton, 2007] is a language that allows its own syntax and semantics to be modified at runtime. It is not exactly a parser generator, but it is mentioned here because it has two features closely related to our work: the specification of new constructs is based on PEGs, and the syntax of these constructs may be dynamically defined and immediately used, at runtime, applying techniques of just-in-time compilation in the implementation. Some changes on the PEG model make the Katahdin approach differ from the other tools described here. For example, the PEG *ordered choice* operator “/” is replaced by an operator “|”, which gives priority to longest match when choosing among alternatives, instead of applying the order of alternatives given. According to the authors, this decision may allow better composition of grammars.

The first version of our implementation, described in the next sections, was developed as an interpreter for the APEG model. An abstract representation of an APEG is interpreted, similarly to the LGI tool. The interpreter allows the use of any number of synthesized and inherited attributes, with a concrete syntax inspired by ANTLR. The input is treated as a stream of characters, following the approach of Rats! and ANTLR, and avoiding the restrictions imposed by Pappy and LGI. Similarly to Pappy and Rats!, full memoization is provided. The use of embedded code, formally defined by the model as conditional and assignment expressions, is completely implemented in the interpreter. The most important novelty, when compared with the tools described in this section, is obviously the possibility of manipulating the set of production rules at parse time.

4.2 Implementing an Interpreter for APEG

The implementation of an interpreter for APEG was developed using the ANTLR parser generator [Parr and Quong, 1994], with embedded code written in Java. The syntax analysis is combined with the static semantics analysis in one single pass. It is possible to use a nonterminal before declaring its production rule, so all uses of nonterminals are collected during the analysis single pass and verified later using Java code and data structures. An AST tree is generated using the operators provided by ANTLR for AST construction. The generated AST is then interpreted, implementing the semantics of the model.

4.2.1 Examples Showing the Concrete Syntax

A grammar for the concrete syntax adopted is presented in Appendix A, inspired on the syntax of ANTLR. Inherited attributes are defined as parameters for nonterminal

Figure 4.1 Concrete syntax for the example of Figure 3.4.

```

1 apeg datadependent;
2
3 literal locals[int n]: number<n> '[' strN<n> ']' !. ;
4
5 strN[int n]: ( {? n > 0 } CHAR { n = n - 1; } )* {? n == 0 } ;
6
7 number returns[int x2] locals[int x1]:
8   digit<x2> ( digit<x1> { x2 = x2 * 10 + x1; } )* ;
9
10 digit returns [int x1]:
11   '0' { x1 = 0; }
12   / '1' { x1 = 1; }
13   / '2' { x1 = 2; }
14   / '3' { x1 = 3; }
15   / '4' { x1 = 4; }
16   / '5' { x1 = 5; }
17   / '6' { x1 = 6; }
18   / '7' { x1 = 7; }
19   / '8' { x1 = 8; }
20   / '9' { x1 = 9; } ;
21
22 CHAR : . . ;

```

symbols. Synthesized attributes are defined as a list of return values. We also defined local attributes, which are synthesized attributes whose values are manipulated only locally. All attributes are typed, anticipating our desire for an efficient code generation. Update expressions are defined between `{...}` and constraint expressions are defined between `{?...}`. Desugaring syntax for optional, and-predicate, positive Kleene closure and character classes are also available.

In Figure 4.1, the APEG of Figure 3.4 is written using the concrete syntax proposed. The specification starts with the name of the grammar followed by a list of production rules. Rule `strN` shows an example of use of an inherited attribute `n`, in line 5. Rule `digit` defines a synthesized attribute `x1` after the keyword “returns”, in line 9. Rule `literal` defines a local variable `n` after the keyword “locals”, in line 3. When a nonterminal is referenced inside a PEG expression, the values of the inherited and synthesized attributes are listed between `<...>`, with all inherited attributes coming first. Character sequences are defined inside single quotes (e.g. `'['` and `']'`, in line 3). The dot operator (“.”) matches any single character. All other elements in Figure 4.1 follow the definitions of the APEG model and may be easily understood.

There is no fixed start symbol for the APEG. When a user applies an APEG to an input file, the name of the chosen start symbol must be provided, together with values for its inherited attributes. So any of the nonterminal symbols of Figure 4.1

Figure 4.2 APEG with user-defined functions and character classes.

```

1 apeg datadependent2;
2 functions StringFunctions;
3
4 literal locals[int n]: number<n> '[' strN<n> ']' !. ;
5
6 strN[int n]: ( {? n > 0 } CHAR { n = n - 1; } )* {? n == 0 } ;
7
8 number returns[int x2] locals[String t]: t=[0-9]+ {x2 = strToInt(t)};
9
10 CHAR: . ;

```

Figure 4.3 Example with the set of production rules changed at parse time.

```

1 apeg adapdatadependent3;
2 options { isAdaptable = true; }
3 functions AdaptableFunctions, StringFunctions;
4
5 literal[Grammar g] locals[Grammar g1]:
6   number<n>
7   {g1 = addRule(copyGrammar(g), concat(
8     concat('strN□:□', concatN('CHAR□', n)), ',');} '[' strN<g1> ']'
9   ;
10
11 strN[Grammar g]: {? false } ;
12
13 number returns[int r] locals[String t] : t=[0-9]+ r = strToInt(t)};
14
15 CHAR : . ;

```

could, in principle, be used as a start symbol, each one describing a different language.

Figure 4.2 presents a simplified version of the same APEG discussed above, showing other features of the implementation. In line 2, the “functions” directive is used, allowing the definition of one or more files with the code for user-defined functions. As we implemented the interpreter in Java, the files are associated to Java classes, and the functions available for the APEG are all static functions defined inside these classes. One example is the function “strToInt”, used in line 8, which converts a string into its integer representation. Also in line 8, there is a use of a *character class*, defining the symbols ‘0’ ... ‘9’, and a use of the feature for binding a variable to the input matched by an APEG expression (“t=...”).

Figure 4.3 shows an example in which the set of production rules is changed at parse time. It is the concrete syntax equivalent to the APEG described in Figure 3.5. The option *isAdaptable* is set to *true*, indicating for the interpreter that the APEG itself must be provided as a value for the first inherited attribute of the start symbol.

The model defined in Chapter 3 requires that every nonterminal must have an APEG (type *Grammar*) as its first inherited attribute. Some nonterminal symbols do not need to mention the APEG attribute, like `number` and `CHAR` in Figure 4.3, so the user does not need to declare it

Consider the APEG expression in lines 6 to 8, in Figure 4.3, and suppose that the input file contains “3[abc]”. The attribute `n` will be assigned the integer value 3. Then a new APEG `g1` will be constructed, adding to the current APEG `g` a rule defined by the string “`strN : CHAR CHAR CHAR;`”. This string is built using the functions `concat` and `concatN`, defined in file `StringFunctions`. Then the nonterminal symbol `strN` will be processed having `g1` as its set of valid production rules. Note that this nonterminal is referenced in line 8, even before the actual rule is created. The interpreter requires that all nonterminal symbols used must be statically declared, otherwise an error is detected at analysis time, so a *stub* rule was added for `strN` in line 11. The function `addRule`, used in line 7, adds a new set of rules to the given APEG, following the steps below:

- The textual representation of the new rules is analysed. If a syntax or static semantics error is detected, an exception is raised.
- If the symbol on the left side of a rule to be added is a new nonterminal, then a new rule is added to the APEG.
- If the symbol on the left side of a rule to be added is already defined, the APEG expressions are combined using the *ordered choice* operator. For example, the rule for the nonterminal symbol `strN` in `g1` will have the following format: “`strN[Grammar g] : {? false} / CHAR CHAR CHAR;`”. It is not possible to change the list of declared attributes of an existing rule.

4.2.2 Implementing PEG with Attributes

In this section, we discuss how attributes are implemented by the interpreter, and how they affect the memoization mechanism of the parser.

In a standard packrat parser, the application of nonterminals is memoized as a mapping from pairs $\langle \textit{nonterminal}, \textit{position} \rangle$ to results, where a result is either *fail* or an integer *newPos*. Because of backtracking, a nonterminal may be applied to the same position of the input file more than once. On the second application and on the following ones, the memoized result is used, allowing linear time processing. The memoized result indicates that the application will either fail again (*fail*), or may

succeed and so it provides the new position (*newPos*) of the input file that must be considered.

As discussed in Section 4.1, tools like LGI implement memoization in a naive and expensive way, in a form of a two-dimensional table with one line for each nonterminal symbol, and one column for each position of the input file. This representation is expensive because the table is frequently very sparse. On the tool Rats!, on the other hand, the columns of this table are stored as several chunks, representing only the cases that really needed memoization.

When inherited attributes are taken into account, the naive approach is even infeasible. A nonterminal symbol may be used with different values for the inherited attributes. A resulting memoization associated with a pair $\langle \textit{nonterminal}, \textit{position} \rangle$ is not enough, because the behaviour may be different for different inherited attributes, even for a same position of the input file. We have implemented memoization as a mapping from a 3-tuple $\langle \textit{nonterminal}, \textit{inhAttr}, \textit{position} \rangle$ to either *fail* or $\langle \textit{synAttr}, \textit{newPos} \rangle$. In this case, *inhAttr* is a list of values for the inherited attributes and *synAttr* is a list of synthesized attributes calculated on the first application of the given 3-tuple. If the application succeeds, the list of synthesized attributes is reused, and the input file is repositioned on *newPos*. This memoization mapping is currently implemented using hash tables, distributed on the nonterminal symbols defined by the APEG. Each nonterminal symbol stores a hash table mapping elements $\langle \textit{inhAttr}, \textit{position} \rangle$ to results of the form *fail* or $\langle \textit{synAttr}, \textit{newPos} \rangle$.

It is important to note that we consider that a set of values for a 3-tuple $\langle \textit{nonterminal}, \textit{inhAttr}, \textit{position} \rangle$ will always produce the same result during parsing, even though the model offers assignment expressions that may change the environment. This is true because we consider that expressions use only values defined locally, no global state is available. When user-defined functions are used, we assume that they also produce the same results when given the same list of arguments. If this property is not valid, our mechanism for memoization would not work. In Section 4.3, we formally define and prove this property.

The stack of environments may be used also during the application of several standard PEG operators. As discussed in Section 3.2, some equations of Figure 3.2 define that changes in an environment are discarded when an expression fails. So our interpreter is supposed to store a copy of the environment in several cases, not only when a new instance of a nonterminal symbol is used. We believe that this semantics is easy to understand, but it may be also very expensive and we are not sure if the costs of these benefits are worthwhile. Currently, the interpreter offers a directive that allows the user to control this semantics. The default behaviour is the one defined

in Figure 3.2, but it is also possible to adopt a semantics in which the changes on the environment produced during the evaluation of the expressions are not discarded, resulting in a more efficient processing.

4.2.3 Implementing Adaptability

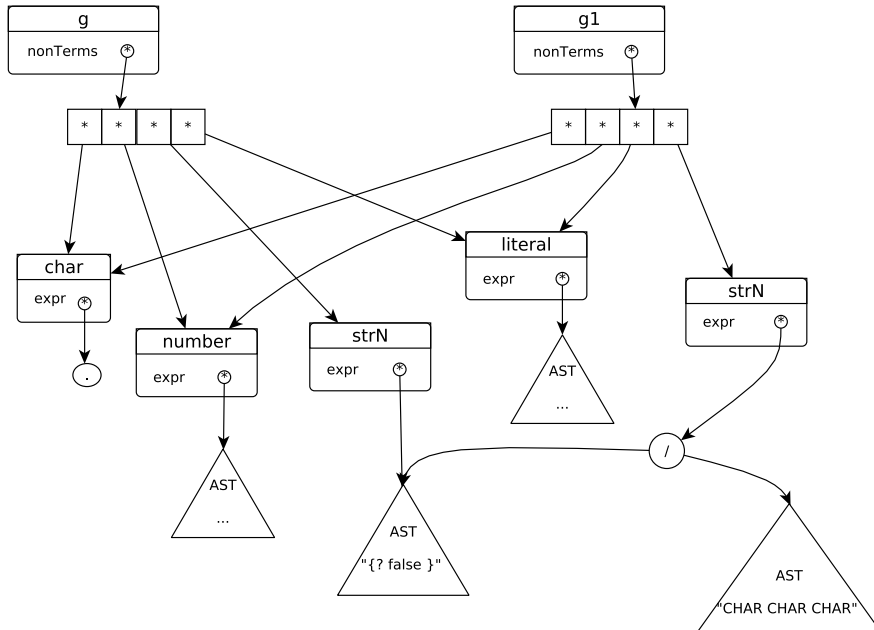
In the APEG model, the set of productions may change at parsing time. This is the feature that presents the most important challenges for an implementation. The decision of building an interpreter instead of generating code to simulate the production rules was directly affected by this feature. Code generation may produce a more efficient parser, but if production rules are changed at parse time, parts of the generated code may be invalidated. So we decided that the first implementation for the model would be an interpreter.

Two other important implementation issues are directly associated with the adaptability of the model. The first one is the cost of storing the set of rules as an inherited attribute at every nonterminal symbol. The second one is the impact on the memoization caused by changes on the production rules. We discuss these two issues in the following.

In Section 4.2.1, the APEG of Figure 4.3 shows an example in which a copy of the current APEG was produced, and a production rule of this copy was modified. We have designed data structures that implement these potentially very expensive operations in an efficient way. An APEG is represented internally by the interpreter as a class named *Grammar*, which contains a set of *NonTerminal* objects. Each nonterminal contains information about its attributes and stores an AST representing the APEG expression associated to it, built during the analysis phase.

When a copy g_1 of an APEG g is generated, g_1 shares with g the set of nonterminal symbols, so the copy operation is very efficient. When a production rule is added to g_1 and the nonterminal symbol is new, only the new *NonTerminal* object is added to g_1 . A more interesting situation happens when a production rule of g_1 is modified, involving a nonterminal symbol that is already defined, and its definition may be shared with g . An example is the addition of a rule for the nonterminal symbol `strN` in the APEG of Figure 4.3. A copy of the *NonTerminal* object associated with `strN` is generated for g_1 and a new AST is created for it. Our approach for modifying existing rules, combining expressions with the *ordered choice* operator, allows a very efficient implementation. The objects shared after the creation of the rule for `strN` are represented in Figure 4.4. Note that it was necessary to create only a copy of the modified *NonTerminal* object and a node to represent the *ordered choice* operator. All other structures are shared

Figure 4.4 Data structures representing a copy of an APEG with a modified production rule.



by the two Grammar objects.

In Section 4.2.2, we described our approach for memoization, but we did not consider situations with changes on the set of production rules. If a production rule changes at parse time, some of the memoization results stored may be invalidated. Currently, our implementation creates copies of the memoization table, discarding memoized values associated to symbols whose rules have changed, and also to other symbols depending on the symbols whose rules have changed. We are studying mechanisms for optimizing this process.

4.3 APEG Properties Associated with Memoization

The following property is a requirement for the correct operation of the memoization scheme proposed in Section 4.2.2: the parser must have always the same behaviour when evaluating a nonterminal symbol for a given position of the input file, if it is given the same values for the set of inherited attributes. When backtrack occurs and a nonterminal symbol must be evaluated a second time using these same parameters, the memoized value can be used instead. In the following, we formally define and prove the desired property. First, an auxiliary lemma is presented.

Lemma 1. *In any APEG, if there is an interpretation of a parsing expression e , for an input string x , in an environment E , it is unique.*

Proof. Suppose there is an interpretation $E \vdash (e, x) \Rightarrow o \vdash E'$. We prove by induction on the heights of the proof trees (see Figures 3.2 and 3.3) for the antecedents that this interpretation is unique, i.e., there is only one result for o and E' . It is easy to see that when the height of the proof tree is 1, the interpretation is unique. We divide the proof of the inductive step in the following cases:

- *Case e is a sequence $e_1 e_2$:* By hypothesis, the interpretation $E \vdash (e, x) \Rightarrow o \vdash E'$ exists, so it must have at least a proof tree, which has one of the forms **Seq**, \neg **Seq**₁ or \neg **Seq**₂. Using the induction hypothesis, the interpretation $E \vdash (e_1, x) \Rightarrow o_1 \vdash E_1$ is unique. Therefore, if $o_1 = f$, then the interpretation $E \vdash (e, x) \Rightarrow o \vdash E'$ must have the form of \neg **Seq**₂ and o is f and E_1 is equal to E . Otherwise, the parsing expression e_1 succeeds for input x in the environment E and the proof tree is **Seq** or \neg **Seq**₁. Using the induction hypothesis on the interpretation of the parsing expression e_2 for the remaining input in the resulting environment, we have that it is unique, $E_1 \vdash (e_2, y) \Rightarrow o_2 \vdash E_2$ with $x = o_1 y$. So, if o_2 is a failure, f , then the proof tree must be \neg **Seq**₁ and o is f and E' is equal to E . Otherwise, the interpretation succeeds, having the form of **Seq**, and o is $o_1 o_2$ and E' is equal to E_2 .
- *Case e is a choice e_1/e_2 :* Using the induction hypothesis, the interpretation of the parsing expression e_1 for the input x in the environment E is unique. If it is not a failure, then the proof tree of $E \vdash (e, x) \Rightarrow o \vdash E'$ has the form **Choice**₁ and the prefix consumed and the resulted environment are the same as the interpretation of e_1 for the input x in the environment E . Otherwise, using the induction hypothesis we have that the interpretation $E \vdash (e_2, x) \Rightarrow o_1 \vdash E_1$ is unique. If it is a failure, then the proof tree must have the form \neg **Choice** and the values of o and E' are f and E , respectively. Otherwise, the form of the proof tree is **Choice**₂ and o is equal to o_1 and E' is E_1 .
- *Case e is a repetition e_1^* :* Using the induction hypothesis, the interpretation $E \vdash (e_1, x) \Rightarrow o_1 \vdash E_1$ is unique. If o_1 is a failure, then the form of the proof tree is \neg **Rep** and the results o is λ and E' is E . Otherwise, the form of the proof tree must be **Rep**. Using the induction hypothesis, we guarantee that the interpretation $E_1 \vdash (e_1^*, y) \Rightarrow o_2 \vdash E_2$, with $x = o_1 y$ and $y \neq \lambda$, is unique. Therefore the values of o and E' is $o_1 o_2$ and E_2 , respectively.

- *Case e is a not-predicate $!e_1$:* Using the induction hypothesis, the interpretation $E \vdash (e_1, x) \Rightarrow o_1 \vdash E_1$ is unique. If the o_1 is f , then the form of the proof tree is $\neg\mathbf{Neg}$ and o is λ and E' is E_1 . If o_1 is not a failure, then the proof tree has the form of \mathbf{Neg} and the values of o and E' are f and E_1 , respectively.
- *Case e is a bind expression $\kappa = e_1$:* Using the induction hypothesis, the interpretation $E \vdash (\kappa = e_1, x) \Rightarrow o_1 \vdash E_1$ is unique. If the o_1 is f , then the form of the proof tree is $\neg\mathbf{Bind}$ and o is f and E' is E . If o_1 is not a failure, then the proof tree has the form of \mathbf{Bind} and the values of o and E' are o_1 and $E_1[\kappa/o_1]$, respectively.
- *Case e is a nonterminal expression $\langle A \downarrow \varepsilon_1 \downarrow \dots \downarrow \varepsilon_p \uparrow \kappa_1 \uparrow \dots \uparrow \kappa_q \rangle$:* As the only possibility for the proof tree has the form of \mathbf{Adapt} and, Using the induction hypothesis, the interpretation $[\vartheta_1/v_1, \dots, \vartheta_p/v_p] \vdash (e_1, x) \Rightarrow o_1 \vdash E_1$ with $v_i = E[\varepsilon_i]$, $1 \leq i \leq p$ and $\langle A \downarrow \vartheta_1 \downarrow \dots \downarrow \vartheta_p \uparrow \varepsilon'_1 \uparrow \dots \uparrow \varepsilon'_q \rangle \leftarrow e_1 \in E[\varepsilon_1]$ is unique, then the only possible values to o is o_1 and to E' is $E[\kappa_1/u_1, \dots, \kappa_q/u_q]$, in which $u_j = E_1[\varepsilon'_j]$, $1 \leq j \leq q$

□

This lemma states that the model is deterministic. Suppose an APEG (V_N, V_T, A, R, S, F) and the judgement $E \vdash (e, x) \Rightarrow o \vdash E'$. Given values for e , x and E , if it is possible to calculate the values for o and E' , they are uniquely defined. Note that the evaluation of an expression can fall into infinite loop and have no interpretation. For example, if the evaluation of the expression e fails for input x in some environment, the evaluation of the expression $(!e)^*$ will fall into infinite loop for the same input.

In order to guarantee that Lemma 1 is valid, the implementation of the interpreter developed assumes that the user-defined functions, representing the set F of an APEG (V_N, V_T, A, R, S, F) , are referentially transparent. This lemma will be used to help proving the desired property, discussed in the beginning of this section.

In the proof of the next theorem, we use the simplified notation \vec{v}_n to denote a sequence v of n expressions. We adopt the following convention: letters ϑ , κ and κ' are used for defining positions of a rule and ε , ε' and ε'' are used for applying positions of a rule. For example, if a nonterminal symbol A with p inherited attributes and q synthesized attributes is used in the left side of a rule, then $\langle A \downarrow \vartheta_1 \downarrow \dots \downarrow \vartheta_p \uparrow \varepsilon_1 \uparrow \dots \uparrow \varepsilon_q \rangle$ may be replaced by $\langle A \downarrow \vec{\vartheta}_p \uparrow \vec{\varepsilon}_q \rangle$. On the other hand, if the same symbol is used in the right side of a rule, then the notation for the nonterminal expression $\langle A \downarrow \varepsilon'_1 \downarrow \dots \downarrow \varepsilon'_p \uparrow \kappa_1 \uparrow \dots \uparrow \kappa_q \rangle$ may be replaced by $\langle A \downarrow \vec{\varepsilon}'_p \uparrow \vec{\kappa}'_q \rangle$. Defining positions are always represented by a single variable.

Theorem 1. *In any APEG, if there is an interpretation of a nonterminal expression for an input string in a given environment, there is also another interpretation of the same nonterminal expression in any other environment, provided that the same values are used for the set of inherited attributes of the nonterminal symbol. The input consumed by both interpretations will be the same, and also the values calculated for the synthesized attributes in both interpretations will be the same.*

Proof. Consider two environments E and E' , an integer number $p \geq 1$ and expressions $\varepsilon_1, \dots, \varepsilon_p, \varepsilon'_1, \dots, \varepsilon'_p$. Suppose that the value of the expression ε_i evaluated in the environment E is the same of the expression ε'_i evaluated in the environment E' for every $1 \leq i \leq p$, i.e., $E[\varepsilon_i] = E'[\varepsilon'_i]$.

Let A be a nonterminal symbol of the APEG and consider the evaluation of a nonterminal expression $\langle A \downarrow \vec{\varepsilon}_p \uparrow \vec{\vartheta}_q \rangle$ in the environment E or $\langle A \downarrow \vec{\varepsilon}'_p \uparrow \vec{\vartheta}'_q \rangle$ in E' . The language attribute (attribute that represents the current set of rules) is the same in environments E and E' , because $E[\varepsilon_1] = E'[\varepsilon'_1]$. So the rule for nonterminal A is the same for both environments: $\langle A \downarrow \vec{\kappa}_p \uparrow \vec{\varepsilon}'_q \rangle \leftarrow e$. Let v_i be the value of the attribute expression ε_i evaluated in environment E or the attribute expression ε'_i evaluated in environment E' , i.e., $v_i = E[\varepsilon_i] = E'[\varepsilon'_i]$ for every $1 \leq i \leq p$. Consider that the interpretation, if exists, of the parsing expression e in the environment $[\kappa_1/v_1, \dots, \kappa_p/v_p]$ for a input string x is given by $[\kappa_1/v_1, \dots, \kappa_p/v_p] \vdash (e, x) \Rightarrow o \vdash E''$. Using Lemma 1, we may guarantee that this interpretation is unique.

Let the u_j be the value of the expression ε''_j , for every $1 \leq j \leq q$, evaluated in environment E'' . We can build the following proof trees to the interpretation of $\langle A \downarrow \vec{\varepsilon}_p \uparrow \vec{\vartheta}_q \rangle$ in the environment E and $\langle A \downarrow \vec{\varepsilon}'_p \uparrow \vec{\vartheta}'_q \rangle$ in the environment E' , for the same input x :

$$\begin{array}{l} \langle A \downarrow \vec{\kappa}_p \uparrow \vec{\varepsilon}''_q \rangle \leftarrow e \in E[\varepsilon_1], \text{ where } E[\varepsilon_1] \equiv \text{language attribute} \\ v_i = E[\varepsilon_i], 1 \leq i \leq p \quad u_j = E''[\varepsilon''_j], 1 \leq j \leq q \\ \text{Proof Tree 1} \frac{[\kappa_1/v_1, \dots, \kappa_p/v_p] \vdash (e, x) \Rightarrow o \vdash E''}{E \vdash (\langle A \downarrow \vec{\varepsilon}_p \uparrow \vec{\vartheta}_q \rangle, x) \Rightarrow o \vdash E[\vartheta_1/u_1, \dots, \vartheta_q/u_q]} \end{array}$$

$$\begin{array}{l} \langle A \downarrow \vec{\kappa}_p \uparrow \vec{\varepsilon}'_q \rangle \leftarrow e \in E'[\varepsilon'_1], \text{ where } E'[\varepsilon'_1] \equiv \text{language attribute} \\ v_i = E'[\varepsilon'_i], 1 \leq i \leq p \quad u_j = E''[\varepsilon''_j], 1 \leq j \leq q \\ \text{Proof Tree 2} \frac{[\kappa_1/v_1, \dots, \kappa_p/v_p] \vdash (e, x) \Rightarrow o \vdash E''}{E' \vdash (\langle A \downarrow \vec{\varepsilon}'_p \uparrow \vec{\vartheta}'_q \rangle, x) \Rightarrow o \vdash E'[\vartheta'_1/u_1, \dots, \vartheta'_q/u_q]} \end{array}$$

The interpretations above consume the same prefix o of the input x . The resulting environments may be different, but the same values u_j are produced for the synthesized

attributes. If there is no interpretation of the expression e for input string x in the environment $[\kappa_1/v_1, \dots, \kappa_p/v_p]$, there will be no interpretation for $(\langle A \downarrow \vec{\varepsilon}_p \uparrow \vec{\vartheta}_q \rangle, x)$ in the environment E , or for $(\langle A \downarrow \vec{\varepsilon}_p \uparrow \vec{\vartheta}_q \rangle, x)$ in E' . \square

In order to understand the results presented above, it is important to note the subtle differences between Lemma 1 and Theorem 1. Lemma 1 states the uniqueness of the interpretation of a parsing expression, when an environment and an input string are defined. This result is used to prove Theorem 1, which says that, for a given input string, it is only necessary to have the same set of values for the inherited attributes of a nonterminal symbol, in order to guarantee that the interpretation of this nonterminal will consume the same input and produce the same values for the synthesized attributes.

The consequence of Theorem 1 is that the memoization algorithm used in packrat parsers can be adapted to the APEG model, using the approach described in Section 4.2.2.

4.4 Mixing Code Generation and Interpretation – An Initial Approach

Figure 4.5 Example with the set of production rules changed at parse time.

```

1 block[Grammar g]:
2   '{' dlist<g, g1> slist<g1> '}' !.;
3
4 dlist[Grammar g] returns[Grammar g1]:
5   decl<g, g1> {g = g1;} (decl<g, g1> {g = g1;})*;
6
7 decl[Grammar g] returns[Grammar g1]:
8   !('int_' var) 'int_' id<s> ',';
9   {g1 = g + 'var:_' + s + '\ ' !alpha<g, ch>;};
10
11 var[Grammar g]:
12   {? false };
13
14 slist[Grammar g]:
15   stmt<g> stmt<g>*;
16
17 stmt[Grammar g]:
18   var<g> '=' var<g> ',';
19
20 id[Grammar g] returns[String s]:
21   alpha<g, ch1> {s = ch1;} (alpha<g, ch2> {s = s + ch2;})*;
22
23 alpha[Grammar g] returns[String ch]:
24   ch=[a-zA-Z0-9_];

```

Usually, parser generators for PEG produce a top-down recursive descent parser. Every nonterminal is implemented by a function whose body is a code mapped from its parsing expression. The return value of each function is an integer value representing the position on the input that it has got moved on or the value -1 if it fails. It is straightforward to extend this idea to include attributes: the inherited attributes are implemented as parameters of functions and synthesized attributes as return values. For example, Figure 4.5 shows a concrete syntax of the example of the block languages presented in Section 3.3 and Figure 4.6¹ shows the code generated for the nonterminal *var* of Figure 4.5. The function *var* has one parameter, the language attribute, and returns an object of type *Result*, which must contain fields representing the portion of the input consumed and the values of the synthesized attributes, when specified.

Complications with this scheme arise when the base grammar is dynamically extended during parsing. When new choices are added to the *var* nonterminal, the function of Figure 4.6 does not represent anymore the correct code for this nonterminal, then this function must be updated. However, it is cumbersome regenerating all the parser code on the fly to reflect these small changes. In cases where the grammar changes several times, as in extensible languages, the on-the-fly regeneration of all the parser is very expensive (we discuss this in Chapter 5). An alternative solution is to interpret the whole grammar directly, as we discussed in Section 4.2. However, this may decrease parsing efficiency. Following, we discuss an approach to efficiently adapt the grammar, generating the code from the base grammar and include hooks to jump from the generated code to interpret the parts that have been added dynamically. Our strategy is based on the assumption that the code for the base grammar is expected to be large and used many times.

Since APEG only allows changes in the definition of nonterminal symbols by insertion of new choices at the end of the rules, we generate a recursive descent parser from an APEG grammar, so there is a function for each nonterminal and, whenever necessary, we place at the end of the body of these functions a call to the interpreter.

¹For simplification, we omit the local attribute declarations

Figure 4.6 Example of code generated by a PEG.

```
1 Result var(Grammar g) {
2   if(false) {
3     // do nothing
4   } else
5     return new Result(-1); // a fail result
6 }
```

Figure 4.7 shows a scratch of the code generated for the grammar of Figure 4.5. As shown, we generate a Java class which has a function to each nonterminal definition on the grammar. The generated class extends the predefined class *Grammar* that has the implementation of standard functions, such as the function *interpretChoice* to interpret an AST and functions to add rules to the grammar or to clone the grammar itself.

The vector *adapt* (line 3 in Figure 4.7) stores a possible new choice for each nonterminal. Notice the hook at the end of the function body of each nonterminal (lines 29 and 40) to call the interpreter with its possible choice. This hook will be reached only if its preceding code fails, indicating that we must interpret the new choice. For example, if the code representing the parsing expression `'{dlist<g, g1> slist<g1> '}` on lines 6 to 24 fails, then we call the interpreter passing its new choice. So, the action of adapting a grammar is just an action of including a new choice rule on the vector *adapt*.

Using this strategy, all the base code for the grammar is generated and compiled, and only the choices that are added dynamically must be interpreted. Our strategy is based on the assumption that the code for the base grammar is expected to be large and used many times. Therefore, the expected result shall be a faster parser than the interpreter we have implemented and discussed in Section 4.2, and will still allow an efficient method for managing syntactic extensions.

In the APEG model, a parsing expression of a nonterminal is fetched from its language attribute. Using different language attributes, it is possible to get different parsing expressions for the same nonterminal, thus effectively adapting the grammar. To have this behaviour, each function generated from a nonterminal has the language attribute as a parameter. The type of this parameter is the type of the grammar generated. In our example, Figure 4.7 shows the language attribute, whose type is *BlockLanguage*, of the functions *block* (line 5) and *var* (line 33).

We use the *dot* notation to call a nonterminal function associated with its correct language attribute. For example, the nonterminals *dlist* and *slist* on the definition of the function *block* are called as *g.dlist(g)* (line 11) and *g1.slist(g1)* (line 15). Note that, as the language attribute passed to each nonterminal is different, we call each nonterminal function from a different language attribute. We must call *slist* from the object *g1* instead of *g* because the vector *adapt* of *g1* has a different value of choices for the function *var*. So, the interpreter is called, the new choice is passed, allowing the use of the variables that have been declared.

A restriction to this approach is that as we use the generated class as the language attribute type, e.g. the *BlockLanguage* type in Figure 4.7, it is not possible to

Figure 4.7 Generated code for the block language.

```

1 public class BlockLanguage extends Grammar {
2   // vector of new choices
3   private CommonTree[] adapt = new CommonTree[8];
4   ...
5   public Result block(BlockLanguage g) {
6     BlockLanguage g1; // local attribute
7     Result result;
8     int position = g.match("{", currentPos);
9     if(position > 0) {
10      g.currentPos = position;
11      result = g.dlist(g);
12      if(!result.isFail()) {
13        g1 = (BlockLanguage) result.getAttribute(0);
14        g1.currentPos = result.getNext_pos();
15        result = g1.slist(g1);
16        if(!result.isFail()) {
17          position = g.match("}", result.getNext_pos());
18          if(position > 0) {
19            char ch = g.read(position);
20            if(APEGInputStream.isEOF(ch))
21              return new Result(position);
22          }
23        }
24      }
25    }
26    Environment env;
27    ... // set the environment to start the interpreter
28
29    // interpreter the choice of block (index 0)
30    return g.interpretChoice(adapt[0], env);
31  }
32
33  public Result var(BlockLanguage g) {
34    if(false) {
35      // do nothing
36    }
37    Environment env;
38    // set the environment to start the interpreter
39
40    // interpreter the choice of var (index 3)
41    return g.interpretChoice(adapt[3], env);
42  }
43  ... // other functions
44 }

```

pass a different grammar which is not subtype of the generated class, as the language attribute. For example, suppose a grammar with other definitions for the same nonterminals presented in Figure 4.7. One may want to pass as the language attribute this grammar in a specific context on the definition of the block language of Figure 4.7. However, as this grammar is not a subclass of *BlockLanguage*, there will be a type error. Instead of using the generated class as the language attribute, we could use the base type, *Grammar*, as the language attribute and use reflection on runtime to invoke the nonterminal functions. However, as the use of reflection may result in a slower program than the use of the dot notation to call functions, we avoid this solution.

During the interpretation process of a parsing expression, it is possible to encounter a reference to a predefined nonterminal. In this case, the interpreter must execute the function code of this nonterminal. For example, suppose an input to the block language example of Figure 4.5 which adds the choice *var* : 'a' !*alpha*(*ch*); to the definition of the nonterminal *var*. The nonterminal referenced in this choice, *alpha*, is the one defined in Figure 4.5 and has a code generated for it. So, when the interpreter reaches this nonterminal, it must stop interpreting and invoke the function of this nonterminal. We implemented this feature using the reflection mechanism of the Java language. Whenever interpreting a nonterminal, the interpreter checks whether the nonterminal is a method of the language attribute object, and if so, the interpreter invokes the method code by reflection. Otherwise, it continues the interpretation.

The code presented in Figure 4.7 was not automatically generated. In order to test our approach, we first produced handwritten code for some APEG specifications, such as the one presented in Figure 4.5 and another for the example of the data-dependent presented in Section 3.3. For the interpretation, we modified the prototype interpreter we had developed and discussed in Section 4.2. One of the main modifications was the code for calling, from the interpreter, functions on the generated code. This feature was implemented using reflection in the Java language, as previously discussed. After our mixed approach proposal be proved useful, we will write the code generator to automatically produce a recursive descent parser from an APEG specification.

4.5 Conclusion

In this chapter, we discussed our design decisions to implement the APEG model. We showed an implementation of PEG with attributes, as an LL-attributed approach, and how the adaptability can be efficiently managed. We also proved that the memoization mechanism can be adapted to work with APEG. As a proof of concept, we implemented

an interpreter with all the features discussed here.

Automatic generation code from an APEG grammar is difficult because changes on the nonterminal rules may invalidate the generated parser code. However, code generation is known to be faster than interpretation. In order to ameliorate this problem, we also propose a novel mixed approach to generate an extensible parser from an APEG grammar, which combines compilation with interpretation. The greatest virtue of this proposal is its simplicity, which comes from the APEG model.

Chapter 5

Evaluation and Validation

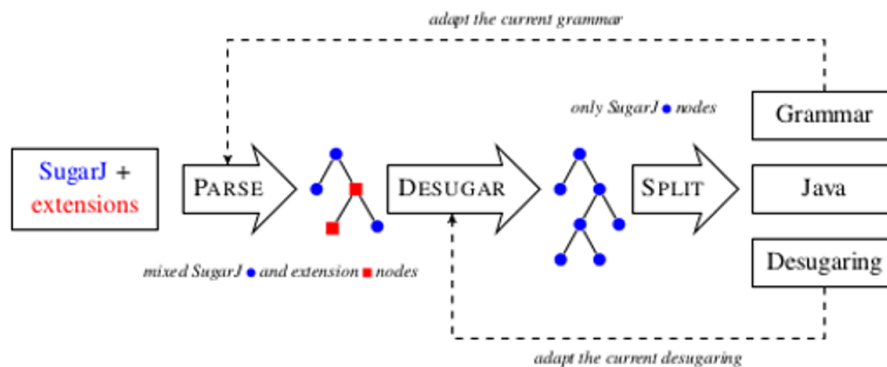
Extensible languages are languages which have features that allow extending its own set of constructs [Wilson, 2004; Tobin-Hochstadt et al., 2011]. Extensible languages seem to have several advantages to implement *Domain-Specific Languages* (DSLs) over other approaches. One of the advantages is the possibility of implementing DSLs in a modular way. For example, Erdweg et al. [2011] show how DSLs can be implemented using the extensible language SugarJ, by means of syntax units designated as *sugar* libraries, which specify a new construct for a domain concept. Tobin-Hochstadt et al. [2011] also discuss the advantages of implementing DSLs by means of libraries.

An interesting application of adaptable models, such as APEG, is to provide a formal mechanism to specify the syntax of extensible languages, including their extensibility mechanism. Using formal specifications, it may be possible to automatically parse programs from these languages. In this chapter, we use extensible languages to evaluate the APEG model because they require dynamic modifications on their own set of rules. In our evaluation, we are trying to answer:

- is APEG able to define the syntax of extensible languages and automatically parse them?
- is an APEG parser efficient enough to be used in practice for defining extensible languages?

To answer the first question, we used APEG to specify the syntax of the languages SugarJ and Fortress and we compared the results with the original implementations. We have chosen these languages because they do not impose restrictions on the syntax of new constructions and they also require the parser to be dynamically modified during parsing every time a new construction is used. Before we discuss the implementation

Figure 5.1 Processing of a SugarJ top-level declaration (borrowed from [Erdweg et al., 2011]).



of these language in Sections 5.2 and 5.3, we discuss, in Section 5.1, the complete cycle of the implementation of an extensible language and how APEG can be used to make the implementation easier.

To find out if APEG is feasible to be used in practice, we compared the execution time to parse programs in the SugarJ language using the original parser implementation and the one using APEG. Section 5.4 shows this evaluation. Finally, Section 5.5 summarizes this chapter conclusions.

5.1 An APEG Implementation of an Extensible Language

It is common to divide the implementation of a programming language in phases: lexical analysis; parsing; semantic analysis; code generation and optimization. An implementation of an extensible language uses the same division of phases, but it has to deal with extensions to the language. An extension (or DSL) modifies all phases providing information to them, informing how to parse and give the semantics for the extension.

As the most important current tools for automatic generation of lexers and parsers do not have resources to implement this behavior of extensible languages, language designers usually handwrite code to emulate it. As an example, Figure 5.1 shows the SugarJ compiler strategy to modify the parser when using an extension and to give its semantics. Observe that when parsing a program which uses an extension (box SugarJ + extensions), the parser builds a mixed AST containing nodes of the original SugarJ (blue circles) and extension nodes (red squares). After that, the desugar module

Figure 5.2 A definition of sugar library for Pairs in SugarJ.

```

1 package syntactic;
2
3 public sugar Pair {
4     context-free syntax
5     '(' type ',' type ')' -> type;
6     '(' expr ',' expr ')' -> expr;
7 }

```

translates extension nodes into nodes of the base language (SugarJ in this case). The SugarJ compiler takes an incremental approach and the task presented in this scheme is done for every top-level entry (the SugarJ compiler splits the program file in a sequence of logical parts called top-level entries, otherwise its approach would not work). Note that after analysing a top-level entry, the SugarJ compiler may modify the code of the parser and the desugar modules, allowing parsing the next top-level entries.

APEG was planned to describe the syntax of extensible languages and automatically parse them, including the mechanism of on-the-fly grammar modification presented in these languages. Therefore, instead of handwriting an incremental mechanism for parsing parts of the input, adapting the grammar and resuming the parser for the remaining input, the designer only specify the language syntax in APEG and automatically generates an extensible parser which is capable to parsing a program of this language. The generated parser also may produce a mixed AST, containing base language and extension nodes. The semantics of the extension nodes can be given using the same strategy applied by the SugarJ compiler, desugaring the extension nodes into base language nodes, or using any other desired technique.

In the following, we discuss only the syntactic aspects of the definition of extensible languages. As APEG does not provide any special support for giving semantics, the reader may suppose that the semantics may be given by translating the AST extension nodes into base language nodes after parsing the program using APEG.

5.2 The Syntax of SugarJ

SugarJ [Erdweg et al., 2011] is a language developed by Erdweg et alii to experiment and validate their idea of *sugar* libraries. The main aim of *sugar* libraries is to encapsulate the definition of extensions for the Java language in units that may be imported or composed for creating other extensions, in a modular way. Figure 5.2¹ shows an

¹All examples of programs in SugarJ languages presented here are borrowed from [Erdweg et al., 2011]

Figure 5.3 Use of the pair syntax.

```

1  import syntactic.Pair;
2
3  public class Test {
4
5      private (String, Integer) p = ('12', 34);
6
7  }
```

example of a definition of a *sugar* library for a new syntax for pairs, creating two new rules: a rule for the definition of pair types in line 5, $type \rightarrow '(type\ ',\ type;)'$, and a rule for using pair expressions in line 6, $expr \rightarrow '(expr\ ',\ expr;)'$. Note that the definition of a rule in SugarJ is in an order that is reverse to the one commonly used in context-free grammars.

A definition of a *sugar* library does not immediately extend the language, an extension is only created when a module or file imports a *sugar* library. As an example, Figure 5.3 shows a program that imports the *sugar* library `Pair` in line 1. After this import statement, the parser effectively extends the language, adding the two rules defined by the *sugar* library. The rules added are used for correctly parsing the attribute `p` of the class `Test` in line 5.

We have defined the syntax of SugarJ in APEG and used an experimental version of an interpreter of the model to automatically perform parsing. As APEG is based on PEG, we adapted an implementation of the Java grammar for the Mouse project [Redziejowski, 2009], which is also based on PEG, and extended it to allow the definition of *sugar* libraries. Figure 5.4 shows the syntax definition of *sugar* libraries. As a definition of a *sugar* library does not extend immediately the grammar, the nonterminal *sugar_decl* only collects the name of the *sugar* library and the rules in a single string. This information is passed through the rules of Figure 5.4 as synthesized attributes and is used later in an import statement to extend the grammar. Differently from the implementation of SugarJ, which defines the rules in SDF [Heering et al., 1989] syntax, we have decided to use the PEG style for defining the rules of SugarJ, because of the base model. Otherwise, we would have to translate the context-free rules to PEG and this would add a complexity that is out of the scope of this thesis.

We have also modified the nonterminal that represents type declarations to allow declarations of *sugar* libraries. Therefore, the definition rule for this nonterminal has

a new choice:

```
type_declaration[String pack, Map m] returns[Map m1]:
  .../ sugar_decl<s,r> {m1 = add(m,pack,s,r);}
```

The nonterminal `type_declaration` has two inherited attributes, the package name and a map from names to rules, and one synthesized attribute, a map from *sugar* names to their corresponding definitions. So, when a *sugar* library is defined by the user, a `type_declaration` returns a new map associating the *sugar* library to its rules. Figure 5.5 shows a new syntax definition for a compilation unit, highlighting the possible changes on the grammar rules. The nonterminal `compilation_unit` receives a map of *sugar* libraries and passes it to the nonterminal `import_decl`. The nonterminal `import_decl` checks if the file is importing a *sugar* library and adapts the grammar, if necessary, using the function *adapt*. The adaptable grammar is returned as a synthesized attribute and passed to the nonterminal `type_declaration`, which may use the new syntax.

Every file is parsed by the nonterminal `compilation_unit`. So, for parsing our examples of Figures 5.2 and 5.3, the compiler parses the definition in Figure 5.2 with the nonterminal `compilation_unit`, which receives the initial grammar of the SugarJ language and an empty map without any definition of *sugar* libraries. As a result, the nonterminal `compilation_unit` returns a new map that has an entry for the new *sugar* library `Pair`. This new map is used in the import declaration for parsing the program text in Figure 5.3, so that the grammar is modified with the new rules defining `Pair` syntax.

Composing sugar libraries

Sugar libraries are composed by importing more than one *sugar* library into the same file. As an example, Figure 5.6 shows a program that uses the syntax of pairs and closures. The compiler extends the grammar with the rules of the syntax of closures defined in Figure 5.7 when parsing the first import statement, in line 1. Next, the grammar is also changed with the syntax of pairs when parsing the import declaration in line 2. The modified grammar, which has the syntactic rules of pairs and closures, is used for parsing the class `Partial`.

The implementation of SugarJ uses SDF [Heering et al., 1989] and it may be necessary to write disambiguation rules when composing various grammars. However, it is impossible to prevent all the possibilities of ambiguities and conflicts, consequently composing two or more *sugar* libraries is not always possible. APEG avoids ambiguities

Figure 5.4 Syntax definition of sugar libraries.

```

1 sugar_decl returns [String name, String rules]:
2   'sugar' name=Id '{' defining_syntax<rules> '}'
3 ;
4 defining_syntax returns [String rules]:
5   'context-free syntax' peg_rule<rules>
6 ;
7 peg_rule returns [String rule]:
8   {rule = "}; (peg_expr<s> '->' id=Id ';'
9     {rule += id + ':' + s + '};')}*
10 ;
11 peg_expr returns [String rule]:
12   peg_seq<rule> ('/' peg_seq<r> {rule += ' / ' + r;})*
13 ;
14 peg_seq returns [String s]:
15   peg_predicate<s> (peg_predicate<s1> {s += ' ' + s1;})*
16   / {s = "};
17 ;
18 peg_predicate returns [String r]:
19   '!' peg_unary_op<s> {r = '!' + s;}
20   / '&' peg_unary_op<s> {r = '&' + s;} / peg_unary_op<r>
21 ;
22 peg_unary_op returns [String r]:
23   peg_factor<s> '*' {r = s + '*';}
24   / peg_factor<s> '+' {r = s + '+';}
25   / peg_factor<s> '?' {r = s + '?';}
26   / peg_factor<r>
27 ;
28 peg_factor returns [String r]:
29   r=(peg_literal / Id / '.')
30   / '(' peg_expr<s> ')' {r = '(' + s + ')'}
31 ;

```

Figure 5.5 Syntax definition of compilation units.

```

1 compilation_unit [Grammar g, Map m] returns [Map ml]:
2   package_decl<p>? (import_decl<g, m, g1> {g=g1;})*
3     (type_declaration<g, p,m,ml> {m=ml;})*
4 ;
5 import_decl [Grammar g, Map m] returns [Grammar g1]:
6   'import' n=qualified_id ';' {g1=adapt(g,m.get(n));}
7 ;

```

Figure 5.6 Composition of more than one sugar library.

```

1  import javaclosure.Closure;
2  import syntactic.Pair;
3
4  public class Partial {
5      public static <R,X,Y> #R(Y)
6          invoke(final #R((X,Y)) f, final X x) {
7              return #R(Y y) {
8                  return f.invoke((x,y));
9              };
10     }
11 }

```

Figure 5.7 Definition of the closure syntax.

```

1  package javaclosure;
2
3  public sugar Closure {
4      context-free syntax
5      '#' type '(' type ')' -> type;
6      '#' type formal_param block -> expr;
7  }

```

using ordered choice, so composition is, in principle, always possible using APEG. In fact, if there is some overlapping between the rules of two or more extensions, the first option on the ordered choice clause will prevail. As new choices are always inserted at the end of a rule definition, a user may change the priority altering the order of the import declarations. It seems a simple task, but it is not always easy to understand the interactions between overlapping rules.

Syntax × Semantics Paradox

Erdweg et al. [2011] claim that it is not clear how to support “local” imports, which may extend the language. They give an example of extending the language with the statement s_1 **after** s_2 whose semantics is to swap the execution order of the statements s_1 and s_2 . They argue that the code

(“12”, 34) **after** import syntactic.Pair

is a paradox, because only after swapping the two statements, the import statement comes before the expression (“12”, 34), so it becomes a valid expression. However, they claim that the parser should already know how to parse the pair expression (“12”, 34), before it can even consider parsing the import.

We claim that this is not a paradox. In fact, it is an error situation and the doubts arise only because of the lack of formalization of the language and a confusion between syntax and semantics. Given the definition of the syntax in APEG, which parses the program from left to right, it is possible to answer this question. Initially, the grammar has the rule *statement* \rightarrow *expr* ‘after’ *expr*, then the parser tries to use this rule to parse the statement. Next, the parser tries to parse the first expression with the current grammar and fails, because the current grammar was not extended yet and there is not a rule for correctly parsing the pair expression (“12”, 34). Note that the meaning of the statement (“12”, 34) `after import syntactic.Pair` was not considered because the objective of the parser is only to check if the program conforms with the grammar rules available at the moment and the semantics of any expression is considered afterwards only if the program is valid.

5.3 The Syntax of Fortress

The main goals of the design of the Fortress language were to emulate mathematical syntax and to be extensible [Allen et al., 2009]. These two goals impose additional difficulties to build a parser for the language. However, defining the extensibility system in a formalism like PEG, which supports unlimited lookahead would bring some advantages [Allen et al., 2009; Ryu, 2009].

Figure 5.8 shows an example of the definition of an extension in Fortress. Line 1 defines a new grammar, called `ForLoop`, which may use symbols of two other grammars, `Expression` and `Identifier`. The Fortress language has two types of nonterminal specifications: the extension of an existing nonterminal, using the symbol `|:=` (line 2) or the definition of a new one (line 4). The right hand side of a rule has two parts, a parsing expression and an action. The parsing expression defines the syntax of the new construct in a PEG style and the action part specifies how to translate the syntax into the core language. The action part is everything after the symbol `=>`. It is possible to use aliases associated with terminal or nonterminal symbols, creating references for them, which can be used in the action part. Figure 5.8 shows an example in which the nonterminal `forStart` is referenced by `b` in line 2.

Figure 5.9 shows part of an APEG syntax definition of the Fortress language. Similarly to the SugarJ definition, the nonterminal `gram_def` defines the syntax of an extension in Fortress and returns a map with the new entry for it. However, differently from the SugarJ definition, a grammar definition in Fortress allows recursion and may use the new syntax in the action part. Therefore, it is necessary to collect the grammar

Figure 5.8 Definition of a for loop in Fortress.

```

1  grammar ForLoop extends{ Expression , Identifier }
2    Expr ::= for b:forStart => <[ b ]>
3
4    forStart ::=
5      i:Id <- e:Expr d:doFront => <[ ... ]>
6    | e:Expr d:doFront => <[ ... ]>
7    ...
8  end

```

rules before parsing the code. We use the and-predicate operator “&” to specify this, collecting the grammar rules while ignoring the action part. Next, we reparse the program with the modified grammar. Note that, when collecting the grammar rules using the and-predicate operator, the action part is parsed as a string, ignoring every symbol between ‘<[’ and ‘]>’ (nonterminal **syn**). After collecting the rule definitions, we adapt the grammar and generate a new grammar **g1**. This new grammar is passed to the nonterminal **nonterm_def**, which passes it to its children, allowing parsing the action part (nonterminal **syntax**). Therefore, the action part may use the new syntax being defined. The use of the and-operator, which allows an infinite lookahead, was very important to handle recursion, a kind of forward reference.

Combining Grammars

Figure 5.10 shows an example of composition of grammars in Fortress. Grammar **A** defines a new nonterminal **Nt**, and grammar **B** extends grammar **A**. Fortress allows the use of the syntax of **A** in the action part of **B**, as in line 6. Grammar **C** extends **B** and can use its syntax, however, **C** cannot use the syntax of **A** because it does not explicitly extend grammar **A**. In [Allen et al., 2009], the authors report that they need to resolve the set of extensions (for example, in grammar **C** it may use syntax defined in **C** or **B**, but not in **A**) to generate the table for parsing the action part and this is not an easy task.

Using the APEG model, defining the task described above is simple and clear. We adapt the grammar, adding the rules of the grammars specified in the **extends** part. For example, parsing the grammar **B**, we add only the rules of **A** and when parsing the grammar **C**, we add only the rules of **B**.

Another difficulty reported in [Allen et al., 2009] is how to compose the rules with multiple extensions, as defined in grammar **D**. In APEG, to have the same behaviour of the original Fortress implementation, we must adapt the grammar in the following

Figure 5.9 APEG formalization of Fortress language.

```

1 gram_def[Grammar g, Map m] returns [Map m1]:
2   'grammar' n=id gram_ext<m,l>? &collect_gram<r>
3     {g1 = adapt(g, r + allRules(l));} nonterm_def<g1>* 'end'
4     {m1 = put(m,n,r);}
5 ;
6 gram_ext[Map m] returns [List l]:
7   'extends' qualified_names<m,l>
8 ;
9 collect_gram returns [String r]:
10  {r = ";"} (non_def<n,r> {r += 'n : r;';})*
11 ;
12 non_def return [String n, String r]:
13   n=id '|' :=' syn<r> ('/' syn<r1> {r += '/' + r1;})*
14   / n=id '::=' syn<r> ('/' syn<r1> {r += '/' + r1;})*
15 ;
16 syn returns [String r]:
17   peg_seq<r> '=>' '<[' !]>' . ']'>'
18 ;
19 nonterm_def[Grammar g]:
20   id '|' :=' syntax ('/' syntax)* / id '::=' syntax ('/' syntax)*
21 ;
22 syntax:
23   peg_seq<r> '=>' '<[' expr ']'>'
24 ;

```

order: first, we add the rules of the grammar which is currently being defined (rules of D in the example), next the grammars in the extends part in the same order that is specified (first, it adds rules of B and in the sequel, rules of C, for the example of Figure 5.10).

The combination of extensions is difficult in the Fortress implementation because it must generate an entire grammar which must contain the definitions of all grammars used. As in the APEG model the grammar is changed locally and only as needed, combining grammars is easy and clear.

Other Features

When we were defining the Fortress language in APEG, we have noted that the language is powerful enough for defining other aspects of itself. For example, an operator name in Fortress must be defined by a sequence containing only uppercase letters and underscore. This sequence must not begin or end with underscore, and must have at least two different symbols. It is not simple to define this in a CFG formalism,

Figure 5.10 Combining grammars.

```

1  grammar A
2    Nt ::= macroA => ...
3  end
4
5  grammar B extends A
6    Nt |:= macroB => <[... macroA ...] >
7  end
8
9  grammar C extends B
10   Nt |:= macroC => <[... macroB ...] >
11  end
12
13  grammar D extends {B,C}
14   Nt |:= macroD => <[... macroB macroC ...] >
15  end

```

leading to a very large grammar. Using the APEG model, it is possible to define this syntactically, as follows.

```

op_name:
  ch1=[A-Z] ('_'* ch2=[A-Z] {? ch1 != ch2}
    tail_op_name / '_'* &[A-Z] op_name);

tail_op_name:
  ![A-Za-z0-9_]
  / ('_'* [A-Z])+ ![A-Za-z0-9_];

```

The nonterminal `op_name` tests if the first two uppercase letters are different and if so, the nonterminal `tail_op_name` is called. Otherwise, it calls itself ignoring all symbols until the second uppercase letter. The nonterminal `tail_op_name` recognizes a sequence of uppercase letters and underscore, in which the last symbol is an uppercase letter.

The parser available for Fortress treats an operator name as an identifier and calls an external function to check if it is a valid operator name. This procedure could also be implemented in Rats! using a definition similar to the one presented above, but the developers apparently favoured efficiency at the expense of clarity. This example is presented only to show the power of APEG, using it for problems other than adapting grammars.

DSL	Implementation With SDF		Implementation With APEG	
	Adapt	Parse	Adapt	Parse
xml	17948	280	16	1048
closure	17858	84	3	582
pair	22920	41	3	368
n-xml	-	-	70	19029
n-closure	22975	378	33	1261
n-pair	29680	111	37	1130
closure-pair-xml	39537	401	52	1604

Table 5.1: Time in milliseconds for parsing programs written in the SugarJ language. The performance of the original SugarJ compiler and the APEG version are compared.

5.4 Performance Evaluation

In order to verify if our approach can be efficient enough to be used in a real implementation of extensible languages, we have implemented a parser for the SugarJ language [Erdweg et al., 2011] in our model. The SugarJ language has the adequate features for testing our approach, because the language must be dynamically extended whenever a sugar library is imported by a module. The SugarJ compiler implemented by the developers of the language serves as basis for a performance comparison. We have run performance tests comparing the parser generated by our implementation, based on the APEG SugarJ description, and the parser provided by the SugarJ developers. The results are summarized in Table 5.1.

The most interesting features to be evaluated are the ones related to the extensibility mechanisms, so we have tested the two parsers with three DSLs (specified as sugar libraries): *xml*, *closure* and *pair*. These DSLs are presented in [Erdweg et al., 2011] and available in the SugarJ website. They define, respectively, a XML language embedded in the Java language, Java extended with closures, and Java extended with pair type. In the original SugarJ implementation, the syntax of each DSL is defined using the SDF style and in the APEG version the syntax is defined in the PEG style. The programs tested as input for both SugarJ parsers are identical, except for the codes related to the definition of the syntax of DSLs, in Sugar libraries, which are defined in the PEG style in the APEG version and in the SDF style in the original version.

We have measured the time for syntax analysis of programs written in the selected DSLs. The parsers first process the rules that specify the DSL, and then parse the code that may include the new constructs. The results in Table 5.1 detail the time spent into adapting the grammar (*adapt*) and actually parsing the program, including new constructs (*parse*). In the original SugarJ compiler, the *adapt* time is the time

for compiling the generated SDF file which has the modified grammar, generating a new parse table, and for loading this new parse table. In our APEG implementation, the *adapt* time is the time for interpreting the rules and changing the grammar. Everything else is considered as *parse* time. As our main focus is adaptability, we have not computed the time for the desugaring phase, after parsing the programs. We have performed the experiments in a 64-bit, 2.4GHz Intel Core i5 running Ubuntu 12.04 with 6GB of RAM. We have repeated the execution 30 times in a row and measured the average for *adapt* and *parse* time.

We may analyse the result considering three types of users: DSL library designers; users of the DSL library; and system end-users. For a system end-user, it does not matter how the system was implemented, using APEG or SDF, because the executable code is, potentially, the same. For the other two types of users, however, how the compiler is implemented does matter, because each one will use the compiler several times during the development cycle (write-compile-execute).

We expect a DSL library designer to edit several times the code of the DSL being developed and use this DSL in small programs to test it, so the *adapt* task may be performed several times in a row. The first three lines, *xml*, *closure* and *pair*, of Table 5.1 give an idea of this common work of a DSL designer. These lines present the parsing of a program that imports and uses only one DSL, namely the one listed in the first column, giving an idea of the overhead to generate a new parser table which includes the rules of the DSL. The original SugarJ compiler takes a long time to perform adaptation because the compiler generates an entire parse table and loads it every time a program uses a sugar library (DSL). On the other hand, the time for adapting the grammar in our APEG implementation is the time taken for parsing the string representing the sugar library rules, which is proportional to the length of this string, and the time for setting a few pointers that will change the language grammar. Therefore, for a DSL library designer the APEG implementation presents a better performance.

The original SugarJ compiler uses a caching system for parse tables, avoiding the generation of a same parse table several times. A DSL user would take advantage of this feature, because it only edits programs that use the DSL library and it pays the *adapt* price only once. In order to evaluate the impact of this feature, we have performed tests with several modules using the same DSL. In Table 5.1, the lines labeled *n-closure* and *n-pair* present results of testing 20-module programs using only the DSL *closure* and DSL *pair*, respectively. The overhead for adapting the grammar occurs only when parsing the first module of a program, but this overhead is so large that the results with the APEG parser are still better. However, if, in a day of work,

the programs which use the DSL were recompiled several times (perhaps, more than 200 times would be enough), the overall performance using SDF would be better.

We also collected large XML examples from a XML repository² and built large programs for testing the performance of the compilers in this situation. We built a test, namely *n-xml*, with data we have collected containing 10 modules using embedded XML. The total size of the programs is 5.4 MB of code. Our APEG implementation parses all the code for the *n-xml* example in a reasonable time, however we could not compare with the original SugarJ compiler because it was not able to process the large files even after several minutes of execution. The examples with results on lines *n-pair*, *n-closure* and *n-xml* were designed with the goal of creating situations which could minimize the overhead for adapting the grammar on the original SugarJ compiler, simulating a situation more often faced by a DSL library user. These tests show that although the *parse* task in the APEG implementation is slower than the SDF one, the time taken by our approach is also reasonable to be used in practice.

Finally, we tested the performance of the implementations when the input programs use different DSLs. In a real system, it would use several DSLs and combinations of them in different parts of the system, requiring to adapt the grammar many times with different rules. The last line of Table 5.1 presents the parse time of a program which is a collection of modules that use different combinations of the three DSLs, testing this situation. The first six tests impose few modifications in the grammar and this last one requires several modifications, but the results are similar.

The tests show that the adaptation time in the original SugarJ compiler is responsible for more than 93% of the execution time and the adaptation time in our APEG implementation is responsible for less than 2% of the execution time, in average. As we expected, the original SugarJ compiler executed faster than our APEG prototype interpreter, when parsing the programs after the grammar was changed. It shows that the time for adapting the grammar in our approach is not significant and the major time is the *parse* time. Parsing the program after adapting the grammar takes a time similar to the execution of the regular PEG algorithm³ which has a linear complexity. Moreover, our interpreter does not implement any optimization, so we could apply several techniques to optimize the *parse* task, getting results near to the SDF performance. Therefore, it is clear that our approach is feasible to use in real implementations of extensible languages.

Besides the performance of the parsers, there are other points to highlight when

²<http://www.cs.washington.edu/research/xmldatasets/www/repository.html>

³We assume that the values of the nonterminal inherited attributes when backtracking will often repeat, allowing memoization

comparing the two SugarJ implementations. Using APEG, the formal definition of the language's syntax is totally specified, whereas it is only partially defined in the original implementation that uses SDF. In the latter, the extensibility mechanism is provided by additional programming to direct the parser to use the current parse table for processing the input until a sugar library import is found. At this point, the library is to be parsed to generate the SDF file grammar, which will be used to produce the parse table from this new grammar. Next, the parser resumes the analysis of the remaining input string. In APEG, a language designer does not need to be concerned with all these procedures, because the adaptation mechanism is automatically performed.

5.5 Conclusion

We showed the definition of the extensible languages SugarJ and Fortress using APEG. The specification of these languages shows that APEG is a powerful formalism which permits a clear definition of what rules are available at a given moment during parsing. The implementation of these languages uses a mixed approach of handwriting code and automatic code generation. Compared with APEG, the syntax of these languages is fully defined and programs are automatically parsed. Using an automatic approach, we may assure correctness and recognition completeness, since with the manual implementation it is very difficult to guarantee that all programs will be correctly analysed. The formal specification of the SugarJ language allows us to resolve the false paradox about the local imports raised by Erdweg et al. [2011]. Also, the semantics of the combination of Fortress grammars is clear in the APEG specification, explicitly showing what set of rules is to be used when a grammar extends another. Although we have discussed only how to define the syntax of these languages using APEG, the semantics of the constructs can be done as usual in other models, such as based on an AST built after parsing the program.

Forward reference is reported as difficult to be handled with adaptable models [Carmi, 2010]. The definition of grammars in Fortress has a kind of forward reference, in which the action part may use syntax that is defined later. Therefore, it is necessary to use a multi-pass approach. We showed that the predicate operator $\&$ of APEG allows simulating a multi-pass parser, handling forward reference properly.

In order to evaluate the feasibility of APEG to be used in practice, we performed an experimental evaluation on parsing programs of the SugarJ languages. The results indicate that APEG may significantly improve the performance of parsing such programs, when compared to the original implementation built with SDF.

Chapter 6

Conclusion and Future Work

The main motivation for this work was the current lack of appropriate models for the definition and implementation of features for allowing on-the-fly modification of the grammar rules. In order to solve this problem, we have proposed a new adaptable model based on PEG. The main goals of the proposed model are:

1. to offer facilities for adapting the grammar during the parsing process, without adding too much complexity to the PEG model;
2. to assure that grammar extensions take effect immediately;
3. to allow an implementation with reasonable efficiency.

Below we present arguments intended to convince the reader that we have succeeded.

6.1 Adaptability at a Low Complexity Cost

Our model allows changing the grammar at parse time and it has a syntax as clear as Christiansen's Adaptable Grammars, because the same principles are used. In order to explore the full power of the model, it is enough for a developer to be familiar with AGs and PEGs. So it is reasonable to say that not much complexity was added to the PEG model, or at least, we used principles that are well known by most language designers (AG). Additionally, we keep some of the most important advantages of declarative adaptable models, such as an easy definition of context dependent aspects associated with static scope and nested blocks. We showed that the use of PEG as the basis for the model allowed a very simple solution for handling forward reference, which is

important to define the syntax of languages like Fortress. Forward reference is reported as very difficult issue to be solved with adaptable models based on CFG.

Our model also has some similarities with the imperative approaches of adaptable grammars. PEG may be viewed as a formal description of a top-down parser, so the order in which the productions are used is important to determine the adaptations our model performs. However, this is not a disadvantage as it is for imperative adaptable models based on CFG. Even for standard PEG (non adaptable), designers must be aware of the top-down nature of the model, so adaptability is not a significant increase on the complexity of the model.

When defining the syntax of extensible languages, the use of APEG has some advantages. Extending a language specification may require the extension of the set of its lexemes. APEG is scannerless so the extension of the set of lexemes in a language is performed with the same features used for the extension of the syntax of the language. APEG enables the use of automatic parser generation, reducing the complexity of building parsers for such language. It makes easier to correct bugs and to change the concrete syntax of the language during the development phase. As the implementation conforms with the specification, if the specification is correct, then also the generated parser will be. Moreover, our specifications clearly define what rules are available at a given moment during the parsing. It allows avoiding confusions about the language syntax, as the false paradox regarding local import raised by Erdweg et al. [2011].

In [Kats et al., 2010], the authors indicate that pure and declarative syntax definitions have significant advantages over parser definitions. They claim that, using the PEG model, it is not possible to be completely oblivious about the parser implementation. A language developer must be aware of the effects of the ordering of alternatives in production rules, which are especially complex for larger, modular grammars with injection productions. In our work, changes to a language definition are restricted to the insertion of new rules, or the extension of a rule by adding an alternative at the end of this rule. These restrictions prevent most problems associated with subtle changes on a language definition, caused by the PEG semantics for ordering of alternatives. The examples in sections 5.2, 5.3, and 5.4 present evidences that the restrictions are not so severe to the point of causing difficulties for the definition of extensions. In fact, this semantics provides a simple mechanism to resolve ambiguities and conflicts when importing several sugar libraries in programs written in SugarJ, by just swapping the order of the imports statements. Perhaps, it would be less annoying to the user than subtle parser table conflicts errors that may arise when using several libraries.

6.2 A Reasonably Efficient Implementation

Another goal was to show that the model allows building an implementation efficient enough to be used in practice. This was another reason for choosing PEG as the base model.

An adaptable parser must deal with changes of the production rules at parse time. If a model such as LR, LL or SGLR is used, it may be necessary to carry on a large amount of computations when a rule is changed, recalculating lookahead functions and updating parse tables. Considering the restrictions we defined for the extension of production rules, it is possible to efficiently produce new PEG rules at parse time. In this case, the PEG semantics for ordering of alternatives is an advantage.

In Chapter 4, we have presented an interpreter for our model. The tests described in Section 5.4 indicate that this interpreter may have a better performance than the available compiler for the extensible language SugarJ. These results show initial evidences that our approach may be used in practice.

Automatic generation of an adaptable model is difficult because extensions on the syntax may invalidate the code of the generated parser. In Section 4.4, we also propose a mixed approach to generate an extensible parser from an APEG grammar, combining compilation with interpretation. This approach has the virtue of being very simple, as opposed to other models based on CFG, since LL, LR and SGLR would require to recalculate parser table decisions.

6.3 Future Work

As explained in Section 4.4, we have not yet developed a code generator that may automatically produce a recursive descent parser for the static part of the language specification written in APEG. The examples used here were handwritten and served only as a proof of concept of the proposed approach. Our next steps include the implementation of this code generator, which will make possible to test for the entire syntax of real extensible languages and evaluate the performance of the generated code when parsing real programs. Our strategy is based on the assumption that the code for the base grammar is expected to be large and used many times than the extensions. We still have to prove this assumption. Several optimizations on the generated parsers may also be introduced. For example, we can apply techniques to generate code for rules that are used more frequently during interpretation.

The semantics of APEG, described in Figures 3.2 and 3.3, defines that, for some operations, changes on the environment must be discarded when a failure occurs. We

are not sure that this semantics is really useful, but it is clear that it may be expensive. Some investigation is still necessary in order to reach a better conclusion. The interpreter developed allows turning on or switching off this semantics, so it may be an appropriate tool for testing the utility of such semantics.

In order to implement memoization for the APEG model, for each nonterminal symbol, it is necessary to store the values of the inherited attributes used for each position of the input string. As the range of the inherited attributes can be unbound, it is not practical to create a simple two dimensional table, such as in packrats parsers. Because of this, we cannot ensure that the algorithm is linear-time. The approach taken by Becket and Somogyi [2008] for memoization on Definite Clause Grammars is similar to ours and they suggest that a super-linear behaviour does not happen in practice. They also show that, in general, it is better memoizing none or few nonterminals than all nonterminals. This subject requires more investigation.

Currently, rules inserted in a nonterminal at parse time are represented as plain strings. In the future, we will evaluate the use of metaprogramming techniques for the definition of the set of new rules. A promising approach may be based on the techniques used in tools like MetaAspectJ [Zook et al., 2004].

As APEG has a flexible mechanism for changing the grammar during the parsing, it opens several possibilities to compose grammars and define grammars in a modular way. It may enable the generation of parsers from APEG grammars and to distribute them as libraries. We have plans to investigate this.

There are some problems related to the PEG approach which are also inherited by our approach, such as left-recursion grammars [Medeiros et al., 2012] and error reporting [Maidl et al., 2013]. Therefore, we have to study how to deal with these problem in APEG, specially when adapting the grammar.

We also plan to investigate if it would be beneficial to use a parsing machine, such as LPeg [Ierusalimschy, 2009; Medeiros and Ierusalimschy, 2008], instead of generating a recursive descent parser from an APEG grammar.

6.4 Publications

The development of this thesis has produced publications at the 16th Brazilian Symposium on Programming Languages [Reis et al., 2012], the 18th Brazilian Symposium on Programming Languages [Reis et al., 2014c], the 29th Annual ACM Symposium on Applied Computing [Reis et al., 2014b] and at the Science of Computer Programming journal [Reis et al., 2014a].

Appendix A

Adaptable Parsing Expression Grammar

We present below, in Figure A.1, a simplified version of our grammar for APEG, written in ANTLR. The terminal symbol `ID` represents an identifier and `ALPHA` is any character. The definitions of the nonterminal symbols `cond` and `exp` are omitted. They represent a condition and an expression, respectively, written in the embedded language.

Figure A.1 Simplified version of the APEG grammar

```
1 grammarDef: 'apeg' ID ';' functions? rule+ ;
2 functions : 'functions' (ID)+ ';' ;
3
4 rule : ID decls? ('returns' decls)? ('locals' declas)? ':'
5       peg_expr ';' ;
6 decls : '[' varDecl (',' varDecl)* ']' ;
7
8 varDecl : type ID ;
9 type : ID ;
10
11 peg_expr : peg_seq ('/' peg_expr | ) ;
12 peg_seq : ( (ID '=')? peg_unary_op )+ ;
13
14 peg_unary_op :
15     peg_factor ('?' | '*' | '+')?
16   | '&' peg_factor | '!' peg_factor
17   | '{?' cond '}' | '{' (ID '=' expr ';')+ '}'
18 ;
19 peg_factor :
20   '\'' ALPHA* '\'' | ID ('<' actPars '>')?
21   | '[' (ALPHA '-' ALPHA)+ ']' | '.'
22   | '(' peg_expr ')'
23 ;
24 actPars: (expr (',' expr )* )? ;
```

Bibliography

- Adams, M. D. (2013). Principled parsing for indentation-sensitive languages: revisiting landin’s offside rule. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL ’13*, pages 511--522, New York, NY, USA. ACM.
- Aho, A. V. (1968). Indexed grammars – an extension of context-free grammars. *J. ACM*, 15(4):647--671. ISSN 0004-5411.
- Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G. L., and Tobin-Hochstadt, S. (2008). The Fortress Language Specification Version 1.0.
- Allen, E., Culpepper, R., Nielsen, J. D., Rafkind, J., and Ryu, S. (2009). Growing a syntax. In *International Workshop on Foundations of Object-Oriented Languages*.
- Barišić, A., Amaral, V., Goulão, M., and Barroca, B. (2011). Quality in use of domain-specific languages: A case study. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU’11*, pages 65--72, New York, NY, USA. ACM.
- Becket, R. and Somogyi, Z. (2008). DCGs + Memoing = Packrat Parsing but Is It Worth It? In Hudak, P. and Warren, D., editors, *Practical Aspects of Declarative Languages*, volume 4902 of *Lecture Notes in Computer Science*, pages 182–196. Springer Berlin Heidelberg.
- Boullier, P. (1994). Dynamic grammars and semantic analysis. Rapport de recherche RR-2322, INRIA. Projet CHLOE.
- Burshteyn, B. (1990a). Generation and recognition of formal languages by modifiable grammars. *SIGPLAN Not.*, 25:45--53. ISSN 0362-1340.

- Burshteyn, B. (1990b). On the modification of the formal grammar at parse time. *SIGPLAN Not.*, 25:117--123. ISSN 0362-1340.
- Cabasino, S., Paolucci, P. S., and Todesco, G. M. (1992). Dynamic parsers and evolving grammars. *SIGPLAN Not.*, 27:39--48. ISSN 0362-1340.
- Carmi, A. (2010). Adaptive multi-pass parsing. Master's thesis, Israel Institute of Technology.
- Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2:113--124. <http://www.chomsky.info/articles/195609--.pdf> – last visited 14th January 2009.
- Chomsky, N. (1959). On certain formal properties of grammars. *Information and Control*, 2(2):137 – 167. ISSN 0019-9958.
- Christiansen, H. (1987). *The Syntax and Semantics of Extensible Languages*. Roskilde datalogiske skrifter. Computer Science, Roskilde University Centre.
- Christiansen, H. (1990). A survey of adaptable grammars. *SIGPLAN Not.*, 25:35--44. ISSN 0362-1340.
- Christiansen, H. (2009). Adaptable grammars for non-context-free languages. In *Proceedings of IWANN'09*, pages 488--495. Springer-Verlag.
- de Chastellier, G. and Colmerauer, A. (1969). W-grammar. In *Proceedings of the 1969 24th national conference*, ACM '69, pages 511--518, New York, NY, USA. ACM.
- Erdweg, S., Rendel, T., Kästner, C., and Ostermann, K. (2011). SugarJ: library-based syntactic language extensibility. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 391--406, New York, NY, USA. ACM.
- Ford, B. (2002a). *Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking*. PhD thesis, Massachusetts Institute of Technology.
- Ford, B. (2002b). Packrat parsing:: Simple, powerful, lazy, linear time, functional pearl. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 36--47, New York, NY, USA. ACM.
- Ford, B. (2004). Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111--122, New York, NY, USA. ACM.

- Fowler, M. (2010). *Domain-Specific Languages*. Addison-Wesley, Boston, USA. ISBN 0321712943.
- Grimm, R. (2006). Better extensibility through modular syntax. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, PLDI'06*, pages 38--51, New York, NY, USA. ACM.
- Heering, J., Hendriks, P. R. H., Klint, P., and Rekers, J. (1989). The syntax definition formalism sdf-reference manual-. *SIGPLAN Not.*, 24(11):43--75. ISSN 0362-1340.
- Ierusalimschy, R. (2009). A text pattern-matching tool based on parsing expression grammars. *Software – Practice and Experience*, 39(3):221--258. ISSN 0038-0644.
- Jim, T., Mandelbaum, Y., and Walker, D. (2010). Semantics and algorithms for data-dependent grammars. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'10*, pages 417--430, New York, NY, USA. ACM.
- Kats, L. C. L., Visser, E., and Wachsmuth, G. (2010). Pure and declarative syntax definition: paradise lost and regained. In Cook, W. R., Clarke, S., and Rinard, M. C., editors, *Proceedings of OOPSLA 2010*, pages 918–932. ACM.
- Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I., and Walton, L. (1996). A software engineering experiment in software component generation. In *ICSE '96: Proceedings of the 18th International Conference on Software Engineering*, pages 542--552, Washington, DC, USA. IEEE Computer Society.
- Knuth, D. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145. ISSN 0025-5661.
- Kosar, T., López, P. E. M., Barrientos, P. A., and Mernik, M. (2008). A preliminary study on various implementation approaches of domain-specific language. *Information and Software Technology*, 50(5):390 – 405. ISSN 0950-5849.
- Kosar, T., Mernik, M., and Carver, J. C. (2012). Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical Software Engineering*, 17(3):276–304. ISSN 1382-3256.
- Kosar, T., Oliveira, N., Marjan, M., Pereira, M. J. V., Črepinšek, M., da Cruz, D., and Henriques, P. R. (2010). Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems*, 7:247–264.

- Koster, C. (1991a). Affix grammars for natural languages. In Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 469–484. Springer Berlin Heidelberg.
- Koster, C. (1991b). Affix grammars for programming languages. In Alblas, H. and Melichar, B., editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 358–373. Springer Berlin Heidelberg.
- Koster, C. H. A. (1970). Affix-grammars. In Peck, J. E. L., editor, *ALGOL 68 Implementation: Proceedings of the IFIP Working Conference on ALGOL 68 Implementation, Munich, Germany, July 20-24, 1970*, pages 95–109. North-Holland.
- Maidl, A., Mascarenhas, F., and Ierusalimschy, R. (2013). Exception handling for error reporting in parsing expression grammars. In Du Bois, A. and Trinder, P., editors, *Programming Languages*, volume 8129 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg.
- Mayer, O. (1972). Some restrictive devices for context-free grammars. *Information and Control*, 20(1):69 – 92. ISSN 0019-9958.
- Medeiros, S. and Ierusalimschy, R. (2008). A parsing machine for pegs. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, pages 2:1--2:12, New York, NY, USA. ACM.
- Medeiros, S., Mascarenhas, F., and Ierusalimschy, R. (2012). Left recursion in parsing expression grammars. In *Proceedings of the 16th Brazilian Conference on Programming Languages*, SBLP'12, pages 27--41, Berlin, Heidelberg. Springer-Verlag.
- Mercer, D. B. (2008). Attributed parsing expression grammars. Master's thesis, University of South Alabama.
- nez Guzmán, E. A. D. (2009). LGI (Language Generator by Instil).
<http://sourceforge.net/projects/instil-lang/>.
- Parr, T. and Fisher, K. (2011). LL(*): the foundation of the antlr parser generator. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 425–436, New York, NY, USA. ACM.
- Parr, T., Harwell, S., and Fisher, K. (2014). Adaptive LL(*) parsing: The power of dynamic analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'14, pages 579–598, New York, NY, USA. ACM.

- Parr, T. J. and Quong, R. W. (1994). ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25:789–810.
- Rahien, A. (2010). *DSLs in BOO: Domain-Specific Languages in .NET*. Manning, Stanford, USA. ISBN 978-1-933988-60-3.
- Redziejowski, R. R. (2009). Mouse: from parsing expressions to a practical parser. In *Proceedings of the CSE&P 2009 Workshop*, pages 514–525. Warsaw University.
- Reis, L. V. S., Bigonha, R. S., Di Iorio, V. O., and Amorim, L. E. S. (2012). Adaptable parsing expression grammars. In Carvalho Junior, F. and Barbosa, L. S., editors, *Programming Languages*, volume 7554 of *Lecture Notes in Computer Science*, pages 72–86. Springer Berlin Heidelberg.
- Reis, L. V. S., Bigonha, R. S., Di Iorio, V. O., and Amorim, L. E. S. (2014a). The formalization and implementation of adaptable parsing expression grammars. *Science of Computer Programming*, 96, Part 2:191–210. ISSN 0167-6423.
- Reis, L. V. S., Di Iorio, V. O., and Bigonha, R. S. (2014b). Defining the syntax of extensible languages. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1570–1576, New York, NY, USA. ACM.
- Reis, L. V. S., Di Iorio, V. O., and Bigonha, R. S. (2014c). A mixed approach for building extensible parsers. In Quintão Pereira, F. M., editor, *Programming Languages*, volume 8771 of *Lecture Notes in Computer Science*, pages 1–15. Springer International Publishing.
- Reis, L. V. S., Di Iorio, V. O., Bigonha, R. S., Bigonha, M. A. S., and Ladeira, R. C. (2009). XAJ: An extensible aspect-oriented language. In *Proceedings of the III Latin American Workshop on Aspect-Oriented Software Development*, pages 57–62. Universidade Federal do Ceará.
- Rozenberg, G. and Wood, D. (1980). Context-free grammars with selective rewriting. *Acta Informatica*, 13(3):257–268. ISSN 0001-5903.
- Ryu, S. (2009). Parsing fortress syntax. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 76–84, New York, NY, USA. ACM.
- Salomaa, A. (1972). Matrix grammars with a leftmost restriction. *Information and Control*, 20(2):143 – 149. ISSN 0019-9958.

- Seaton, C. (2007). A programming language where the syntax and semantics are mutable at runtime. Technical report CSTR-07-005, University of Bristol.
- Shutt, J. N. (1998). Recursive adaptable grammars. Master's thesis, Worcester Polytechnic Institute.
- Sintzoff, M. (1967). Existence of a van wijngaarden syntax for every recursively enumerable set. *Annales de la Société Scientifique de Bruxelles*, 81(2):115–118.
- Slonneger, K. and Kurtz, B. L. (1995). *Formal Syntax and Semantics of Programming Languages: a Laboratory Based Approach*. Addison-Wesley. ISBN 0201656973.
- Stansifer, P. and Wand, M. (2011). Parsing reflective grammars. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications, LDTA '11*, pages 10:1--10:7, New York, NY, USA. ACM.
- Tobin-Hochstadt, S., St-Amour, V., Culpepper, R., Flatt, M., and Felleisen, M. (2011). Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI'11*, pages 132--141, New York, NY, USA. ACM.
- van Deursen, A. and Klint, P. (1998). Little languages: Little maintenance. *Journal of Software Maintenance*, 10(2):75--92. ISSN 1040-550X.
- Watt, D. A. and Madsen, O. L. (1983). Extended attribute grammars. *Comput. J.*, 26(2):142–153.
- Wegbreit, B. (1970). *Studies in Extensible Programming Languages*. Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York. ISBN 0-8240-4423-1.
- Wijngaarden, A. v. (1969). *Report on the algorithmic language ALGOL 68*. Printing by the Mathematisch Centrum. ISBN B0007IUUXM.
- Wilson, G. V. (2004). Extensible programming for the 21st century. *Queue*, 2(9):48--57. ISSN 1542-7730.
- Zook, D., Huang, S. S., and Smaragdakis, Y. (2004). Generating AspectJ programs with Meta-AspectJ. In *Generative Programming and Component Engineering: Third International Conference, GPCE 2004, volume 3286 of LNCS*, pages 1--19. Springer.