Adaptable Parsing Expression Grammars

Leonardo V. S. Reis[1], Roberto S. Bigonha[1],
Vladimir O. Di Iorio[2], and Luis Eduardo S. Amorim[2]

[1] Departamento de Ciência da Computação, Universidade Federal de Minas Gerais
`leo@dcc.ufmg.br, bigonha@dcc.ufmg.br,`
[2] Departamento de Informática, Universidade Federal de Viçosa
`vladimir@dpi.ufv.br, luis.amorim@ufv.br`

Abstract. The term "extensible language" is especially used when a language allows the extension of its own concrete syntax and the definition of the semantics of new constructs. Most popular tools designed for automatic generation of syntactic analyzers do not offer any desirable resources for the specification of extensible languages. When used in the implementation of features like syntax macro definitions, these tools usually impose severe restrictions. We claim that one of the main reasons for these limitations is the lack of formal models that are appropriate for the definition of the syntax of extensible languages.
This paper presents the design and formal definition for *Adaptable Parsing Expression Grammars (APEG)*, an extension to the PEG model that allows the manipulation of its own production rules during the analysis of an input string. It is shown that the proposed model may compare favorably with similar approaches for the definition of the syntax of extensible languages.

Keywords: extensible languages, adaptable grammars, PEG

1 Introduction

In recent years, we have witnessed important advances in parsing theory. For example, Ford created *Parsing Expression Grammars (PEG)* [13], an alternative formal foundation for describing syntax, and packrat parsers [12], top-down parsers with backtracking that guarantee unlimited lookahead and a linear parsing time. Parr has devised a new parsing strategy called LL(*) for the ANTLR tool, that allows arbitrary lookahead and recognizes some context-sensitive languages [20]. The parser generator YAKKER presents new facilities for applications that operate over binary data [16]. These advances do not include important features for the definition of extensible languages, although the importance of extensible languages and the motivation for using it have been vastly discussed in recent literature [1, 10, 11, 25].

As a simple example of desirable features for the implementation of extensible languages, Figure 1 shows an excerpt from a program written in the Fortress language [1]. Initially, a new syntax for loops is defined, and then the new syntax is used in the same program. Standard tools for the definition of the syntax of programming languages are not well suited for this type of extension, because the syntax of the language is modified while the program is processed. A Fortress interpreter, written with the tool Rats! [14], uses the following method: it collects only the macro (extension) definitions in a first pass, processes the necessary modifications to the grammar, and then parses the rest of the program in a second pass [21]. Another solution for similar problems, used in a compiler for the extensible language OCamL [19], is to require that macro definitions and their usage always reside in different files [15].

```
grammar ForLoop extends {Expression, Identifier}
  Expr | :=
    for {i:Id ← e:Expr, ?Space}* do block:Expr end ⇒
        // ... define translation to pure Fortress code
end
...
// Using the new construct
g1 = < 1, 2, 3, 4, 5 >
g2 = < 6, 7, 8, 9, 10 >
for i ← g1, j ← g2 do println "(" i "," j ")" end
```

Fig. 1. A Fortress program with a syntax macro.

A tool that is able to parse the program in Figure 1 in one pass must be based in a model that allows syntax extensions. We propose *Adaptable Parsing Expression Grammars (APEG)*, a model that combines the ideas of *Extended Attribute Grammars*, *Adaptable Grammars* and *Parsing Expression Grammars*. The main goals that the model has to achieve are: legibility and simplicity for syntactic extension, otherwise it would be restricted to a very small set of users; and it must be suitable for automatic generation of syntactic analyzers.

1.1 From Context-Free to Adaptable Grammars

Context Free Grammars (CFGs) are a formalism widely used for the description of the syntax of programming languages. However, it is not powerful enough to describe context dependent aspects of any interesting programming language, let alone languages with extensible syntax. In order to deal with context dependency, several augmentations to the CFG model have been proposed, and the most commonly used is *Attribute Grammars* (AGs) [17]. In AGs, *evaluation rules* define the values for attributes associated to symbols on production rules, and *constraints* are predicates that must be satisfied by the attributes.

Authors like Christiansen [7] and Shutt [22] argue 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. Christiansen gives as an example an attribute grammar for ADA, in which a single rule representing function calls has two and a half pages associated to it, to describe the context conditions. He proposes an approach called *Adaptable Grammars* [8], explicitly providing mechanisms within the formalism to allow the production rules to be manipulated.

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. There is no need to use an auxiliary symbol table and additional code to manipulate it.

Adaptable grammars are powerful enough even for the definition of advanced extensibility mechanisms of programming languages, like the one presented in Figure 1. However, as the model is based on CFG, undesirable ambiguities may arise when the set of production rules is modified. There are also problems when using the model for defining some context sensitive dependencies. For example, it is hard to build context free rules that define that an identifier cannot be declared twice in a same environment [22], although this task can be easily accomplished using attribute grammars and a symbol table.

1.2 From Adaptable Grammars to Adaptable PEGs

We propose an adaptable model that is based on *Parsing Expression Grammars* (PEGs) [13]. Similarly to *Extended Attribute Grammars* (EAGs) [26], attributes are associated to the symbols of production rules. And similarly to *Adaptable Grammars* [8], the first attribute of every nonterminal symbol represents the current valid grammar. Every time a nonterminal is rewritten, the production rule is fetched from the grammar in its own attribute, and not from a global static grammar, as in a standard CFG. Different grammars may be built and passed to other nonterminal symbols.

A fundamental difference between CFGs and PEGs is that the choice operator in PEG is ordered, giving more control of which alternative will be used and eliminating ambiguity. PEG also defines operators that can check an arbitrarily long prefix of the input, without consuming it. We will show that this feature may allow for a simple solution for specifying the constraint that an identifier cannot be defined twice in a same environment.

The main contributions of this paper are: 1) the design of an adaptable model based on PEG for definition of the syntax of extensible languages; 2) a careful formalization for the model; 3) a comparation with adaptable models based on CFG, that exhibits the advantages of the proposal.

The rest of the paper is organized as follows. In Section 2, we present works related to ours. Section 3 contains the formalization of Adaptable PEG. Examples of usage are presented in Section 4. Conclusions and future works are discussed in Section 5.

2 Related Work

It seems that Wegbreit was the first to formalize the idea of grammars that allow for the manipulation of their own set of rules [27], so the idea has been around for at least 40 years. Wegbreit proposed *Extensible Context Free Grammars* (ECFGs), consisting of a context free grammar together with a finite state transducer. The instructions for the transducer allow the insertion of a new production rule on the grammar or the removal of an existing rule.

In his survey of approaches for extensible or adaptable grammar formalisms, Christiansen proposes the term *Adaptable Grammar* [8]. In previous works, he had used the term *Generative Grammar* [7]. Although Shutt has designated his own model as *Recursive Adaptable Grammar* [22], he has later used the term *Adaptive Grammar* [23]. The lack of uniformity of the terms may be one of the reasons for some authors to publish works that are completely unaware of important previous contributions. A recent example is [24], where the authors propose a new term *Reflective Grammar* and a new formalism that has no reference to the works of Christiansen, Shutt and other important similar models.

In [22], Shutt classifies adaptable models as *imperative* or *declarative*, depending on the way the set of rules is manipulated. 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. So the grammar designer must know exactly the order of decisions made by the parser. One example is the ECFG model mentioned above.

The following works may also be classified as imperative approaches. Burshteyn proposes *Modifiable Grammars* [4], using the model in the tool USSA [5]. 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. Because of the dependency on the parser algorithm, Burshteyn presents two different formalisms, one for bottom-up and another one for top-down parsing. Cabasino and Todesco [6] propose *Dynamic Parsers and Evolving Grammars*. Instead of a transducer, as works mentioned above, each production of a CFG may have an associated rule that creates new nonterminals and productions. The derivations must be rightmost and the associated parser must be bottom-up. Boullier's *Dynamic Grammars* [2] is another example that forces the grammar designer to be aware that derivations are rightmost and the associated parser is bottom-up.

One advantage of declarative adaptable models is the relative independency from the parsing algorithm. Christiansen's *Adaptable Grammars*, mentioned above, is an example of a declarative model. Here we refer to the first formalization presented by Christiansen – later, he proposed an equivalent approach, using definite clause grammars [9]. It is essentially an Extended Attribute Grammar where the first attribute of every non terminal symbol is inherited and represents the *language attribute*, which contains the set of production 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.

Shutt observes that Christiansen's *Adaptable Grammars* inherits the non orthogonality of attribute grammars, with two different models competing. The CFG kernel is simple, generative, but computationally weak. The augmenting facility is obscure and computationally strong. He proposes *Recursive Adaptable Grammars* [22], where a single domain combines the syntactic elements (terminals), meta-syntactic (nonterminals and the language attribute) and semantic values (all other attributes).

Our work is inspired on *Adaptable Grammars*. The main difference is that, instead of a CFG as the base model, we use PEG. Defining a special inherited attribute as the language attribute, our model keeps the advantage of easy definitions for block structured scope. The use of PEG brings additional advantages, such as coping with ambiguities when modifying the set of rules and more powerful operators that provide arbitrary lookahead. With these operators is easy, for example, to define constraints that prevent multiple declarations, a problem that is difficult to solve in other adaptable models.

Our model has some similarities also with the imperative approaches. PEG may be viewed as a formal description of a top-down parser, so the order the productions are used is important to determine the adaptations our model performs. But we believe that it 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 model, so adaptability is not a significant increase on the complexity of the model.

We believe that problems regarding efficient implementation are one of the reasons that adaptable models are not used yet in important tools for automatic parser generation. Evidence comes from recent works like *Sugar Libraries* [11], that provide developers with tools for importing syntax extensions and their desugaring as libraries. The authors use SDF and Stratego [3] for the implementation. They mention that adaptable grammars could be an alternative that would simplify the parsing procedures, but indicate that their efficiency is questionable. One goal of our work is to develop an implementation for our model that will cope with the efficiency demands of parser generators.

3 Definition of the Model

The adaptability of Adaptable PEGs is achieved by means of an attribute associated with every nonterminal to represent the current grammar. 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. In this section, we discuss our design decisions on how to combine PEG and attributes (*Attribute PEGs*), and then present a formal definition for *Adaptable PEG*. Basic knowledge about Extended Attribute Grammars and Parsing Expression Grammars is desirable – we recommend [26] and [13].

3.1 PEG with Attributes

Extended Attribute Grammar (EAG) is a model for formalizing context sensitive features of programming languages, proposed by Watt and Madsen [26]. Compared to *Attribute Grammar* (AG) [17] and *Affix Grammar* [18], EAG is more readable and generative in nature [26].

Figure 2 shows an example of a EAG that generates a binary numeral and calculates its value. Inherited attributes are represented by a down arrow symbol, and synthesized attributes are represented by an up arrow symbol. Inherited attributes on the left side and synthesized attributes on the right side of a rule are called *defining positions*. Synthesized attributes on the left side and inherited attributes on the right side of a rule are called *applying positions*.

$$\boxed{\begin{array}{lcl}
\langle S \uparrow x_1 \rangle & \rightarrow & \langle T \downarrow 0 \uparrow x_1 \rangle \\
\langle T \downarrow x_0 \uparrow x_2 \rangle & \rightarrow & \langle B \uparrow x_1 \rangle\ \langle T \downarrow 2 * x_0 + x_1 \uparrow x_2 \rangle \\
\langle T \downarrow x_0 \uparrow 2 * x_0 + x_1 \rangle & \rightarrow & \langle B \uparrow x_1 \rangle \\
\langle B \uparrow 0 \rangle & \rightarrow & \mathbf{0} \\
\langle B \uparrow 1 \rangle & \rightarrow & \mathbf{1}
\end{array}}$$

Fig. 2. An example of an EAG that generates binary numerals.

A reader not familiar with the EAG notation can use the following association with procedure calls of an imperative programming language, at least for the examples presented in this paper. The left side of a rule may be compared to a procedure signature, with the inherited attributes representing the names of the formal parameters and the synthesized attributes representing expressions that define the values returned (it is possible to return more than one value). For example, $\langle T \downarrow x_0 \uparrow 2 * x_0 + x_1 \rangle$ (third line of Figure 2) would represent the signature of a procedure with name T having x_0 as formal parameter, and returning the value $2 * x_0 + x_1$, an expression that involves another variable x_1 defined in the right side of the rule. The right side of a rule may be compared to the body of a procedure, with every symbol being a new procedure call. Now inherited attributes represent expressions that define the values for the arguments, and synthesized attributes are variables that store the resulting values. For example, $\langle T \downarrow 2 * x_0 + x_1 \uparrow x_2 \rangle$ (second line of Figure 2) would represent a call to procedure T having the value of $2 * x_0 + x_1$ as argument, and storing the result in variable x_2.

One of the improvements introduced by EAG is the use of *attribute expressions* in applying positions, allowing a more concise specification of AG *evaluation rules*. For example, the rules with B as left side indicate that the synthesized attribute is evaluated as either 0 or 1. Without the improvement proposed by EAG, it would be necessary to choose a name for an attribute variable and to add an explicit evaluation rule defining the value for this variable.

We define *Attribute PEGs* as an extension to PEGs, including attribute manipulation. Attribute expressions are not powerful enough to replace all uses of explicit evaluation rules in PEGs, so we propose that Attribute PEGs combine attribute expressions and explicit evaluation rules. In PEGs, the use of recursion is frequently replaced by the use of the *repetition* operator "*", giving definitions more related to an imperative model. So we propose that evaluation rules in Attribute PEGs may update the values of the attribute variables, treating them as variables of an imperative language.

Figure 3 shows an Attribute PEG equivalent to the EAG presented in Figure 2. 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 because the options are defined in a single parsing expression. 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 $[x_0 := 2 * x_0 + x_1]$.

$$\boxed{\begin{array}{lcl}
\langle S \uparrow x_0 \rangle & \leftarrow & \langle T \uparrow x_0 \rangle \\
\langle T \uparrow x_0 \rangle & \leftarrow & \langle B \uparrow x_0 \rangle \;\; (\langle B \uparrow x_1 \rangle [x_0 := 2 * x_0 + x_1]) * \\
\langle B \uparrow x_1 \rangle & \leftarrow & (\mathbf{0}\ [x_1 := 0]) \quad / \quad (\mathbf{1}\ [x_1 := 1])
\end{array}}$$

Fig. 3. An example of an attribute PEG.

Besides explicit evaluation rules, AGs augment context-free production rules with *constraints*, predicates which must be satisfied by the attributes in each application of the rules. In Attribute PEGs, we allow also the use of constraints, as predicates defined in any position on the right 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 in [16].

Another improvement provided by EAG is the possibility of using the same attribute variable in more than one defining rule position. It 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.

3.2 Formal Definition of Attribute PEG

We extend the definition of PEG presented in [13] and define Attribute PEG as a 6-tuple (V_N, V_T, A, R, S, F), where V_N and V_T are finite sets of nonterminals and terminals, respectively. $A : V_N \rightarrow \mathcal{P}(\mathbb{Z}^+ \times \{\uparrow, \downarrow\})$ is an attribute function that maps every nonterminal to a set of attributes. Each attribute is represented by a pair (n, t), where n is a distinct attribute position number and t is an element of the set $\{\uparrow, \downarrow\}$. The use of positions instead of names makes definitions shorter [26]. The symbol $\uparrow$ represents an inherited attribute and $\downarrow$ represents a synthesized attribute. $R : V_N \rightarrow \mathcal{P}_e$ is a total rule function which maps every nonterminal to a *parsing expression* and $S \in V_N$ is an initial *parsing expression*.

F is a finite set of functions that operate over the domain of attributes, used in attribute expressions. We assume a simple, untyped language of attribute expressions that include variables, boolean, integer and string values. If $f \in F$ is a function of arity n and $e_1, \ldots, e_n$ are attribute expressions, then $f(e_1, \ldots, e_n)$ is also an attribute expression.

Suppose that e, e_1 and e_2 are *attribute parsing expressions*. The set of valid *attribute parsing expressions* ($\mathcal{P}_e$) can be recursively defined as:

$$\begin{array}{rl}
\lambda \in \mathcal{P}_e & \text{(empty expression)} \\
a \in \mathcal{P}_e, \text{ for every } a \in V_T & \text{(terminal expression)} \\
A \in \mathcal{P}_e, \text{ for every } A \in V_N & \text{(nonterminal expression)} \\
e_1 e_2 \in \mathcal{P}_e & \text{(sequence expression)} \\
e_1 / e_2 \in \mathcal{P}_e & \text{(ordered choice expression)} \\
e^* \in \mathcal{P}_e & \text{(zero-or-more repetition expression)} \\
!e \in \mathcal{P}_e & \text{(not-predicate expression)} \\
[v := exp] \in \mathcal{P}_e & \text{(update expression)} \\
[exp] \in \mathcal{P}_e & \text{(constraint expression)}
\end{array}$$

To the set of standard *parsing expressions*, we add two new types of expressions. *Update expressions* have the format $[v := exp]$, where v is a variable name and *exp* is an attribute expression, using functions from F. They are used to update the value of variables in an environment. *Constraint expresssions* with the format $[exp]$, where *exp* is an attribute expression that evaluates to a boolean value, are used to test for predicates over the attributes.

Nonterminal expressions are nonterminals symbols with attribute expressions. Without losing generality, we will assume that all inherited attributes are represented in a nonterminal before its synthesized attributes. So, suppose that $e \in R(A)$ is the parsing expression associated with nonterminal A, p is its number of inherited attributes and q the number of synthesized attributes. Then $\langle A \downarrow a_1 \downarrow a_2 \ldots \downarrow a_p \uparrow b_1 \uparrow \ldots \uparrow b_q \rangle \leftarrow e$ represents the rule for A and its attributes. We will also assume that the attribute expressions in defining positions of nonterminals are always represented by a single variable.

The example of Figure 3 can be expressed formally as $G = (\{S, T, B\}, \{\mathbf{0}, \mathbf{1}\}, \{(S, \{(1, \uparrow)\}), (T, \{(1, \uparrow)\}), (B, \{(1, \uparrow)\})\}, R, S, \{+, *\})$, where R represents the rules described in Figure 3.

3.3 Semantics of Adaptable PEG

An Adaptable PEG is an Attribute PEG whose first attribute of all nonterminals is inherited and represents the language attribute. Figure 4 presents the semantics of an Adaptable PEG. Almost all the formalization is related to PEG with attributes. Only the last equation defines adaptability.

An environment maps variables to values, with the following notation: . (a dot) represents an empty environment, i.e., all variables map to the *unbound* value; $[x_1/v_1, \ldots, x_n/v_n]$ maps x_i to v_i, $1 \leq i \leq n$; $E[x_1/v_1, \ldots, x_n/v_n]$ is an environment which is equal to E, except for the values of x_i that map to v_i,

$1 \leq i \leq n$. We write $E[\![e]\!]$ to indicate the value of the expression e evaluated in the environment E.

Figure 4 defines the judgement $E \vdash (e, x) \Rightarrow (n, o) \vdash E^{'}$, which says that the interpretation of the parsing expression e, for the input string x, in an environment E, results in (n, o), and produces a new environment $E^{'}$. In the pair (n, o), n indicates the number of steps for the interpretation and $o \in V_T^* \cup \{f\}$ indicates the prefix of x that is consumed, if the expression succeeds, or $f \notin V_T^*$, if it fails.

Note that the changes in an environment are discarded when an expression fails. For example, in a sequence expression, a new environment 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 $\neg$**Seq**$_1$ and $\neg$**Seq**$_2$. A similar behaviour is defined for $\neg$**Term**$_1$ and $\neg$**Term**$_2$, when a terminal expression fails, and for $\neg$**Rep**, when a repetition fails.

Rules **Neg** and $\neg$**Neg** show that the environment changes computed inside a not-predicate expression are not considered in the outer level, allowing arbitrary lookahead without colateral effects. Rules **Atrib** and $\neg$**Atrib** define the behaviour for update expression, and rules **True** and **False** represent predicate evaluation in constraint expressions.

The most interesting rule is **Adapt**. It defines how nonterminal expressions are evaluated. Attribute values are associated with variables using an approach similar to EAG, but in a way more operational; it is also similar to *parameterized nonterminals* described in [16], but allowing several return values instead of just one. When a nonterminal is processed, the values 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, that is always the first attribute of the symbol. It is indeed the only point in all the rules of Figure 4 associated with the property of adaptability.

Now we can define the language accepted by an Adaptable PEG as follows. Let $G = (V_N, V_T, A, R, S, F)$ be an Adaptable PEG. Then

$$L(G) = \{w \in V_T^* \ \mid \ . \vdash (\langle S \downarrow G \ldots \rangle, w) \Rightarrow (n, w^{'}) \vdash E^{'}\}$$

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, n represents the number of steps for the derivation, $w^{'}$ is the prefix of w matched and $E^{'}$ is the resulting environment as in [13]. The language $L(G)$ is the set of words w that do not produce f (failure).

4 Empirical Results

In this section, we present three examples of usage of Adaptable PEG. The first example is a definition of context dependent constraints commonly required

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

$$\textbf{Empty}\ \frac{x \in V_T^*}{E \vdash (\lambda, x) \Rightarrow (1, \lambda) \vdash E} \qquad \textbf{Term}\ \frac{a \in V_T \qquad x \in V_T^*}{E \vdash (a, ax) \Rightarrow (1, a) \vdash E}$$

$$\neg\textbf{Term}_1\ \frac{a, b \in V_T \qquad a \neq b \qquad x \in V_T^*}{E \vdash (a, bx) \Rightarrow (1, f) \vdash E} \qquad \neg\textbf{Term}_2\ \frac{a \in V_T}{E \vdash (a, \lambda) \Rightarrow (1, f) \vdash E}$$

$$\textbf{Seq}\ \frac{E_1 \vdash (e_1, x_1x_2y) \Rightarrow (n_1, x_1) \vdash E_2 \qquad E_2 \vdash (e_2, x_2y) \Rightarrow (n_2, x_2) \vdash E_3}{E_1 \vdash (e_1e_2, x_1x_2y) \Rightarrow (n_1 + n_2 + 1, x_1x_2) \vdash E_3}$$

$$\neg\textbf{Seq}_1\ \frac{E_1 \vdash (e_1, x_1y) \Rightarrow (n_1, x_1) \vdash E_2 \qquad E_2 \vdash (e_2, y) \Rightarrow (n_2, f) \vdash E_3}{E_1 \vdash (e_1e_2, x_1y) \Rightarrow (n_1 + n_2 + 1, f) \vdash E_1}$$

$$\neg\textbf{Seq}_2\ \frac{E_1 \vdash (e_1, x) \Rightarrow (n_1, f) \vdash E_2}{E_1 \vdash (e_1e_2, x) \Rightarrow (n_1 + 1, f) \vdash E_1}$$

$$\textbf{Choice}_1\ \frac{E \vdash (e_1, x_1y) \Rightarrow (n_1, x_1) \vdash E'}{E \vdash (e_1/e_2, x_1y) \Rightarrow (n_1 + 1, x_1) \vdash E'}$$

$$\textbf{Choice}_2\ \frac{E_1 \vdash (e_1, x) \Rightarrow (n_1, f) \vdash E_2 \qquad E_1 \vdash (e_2, x) \Rightarrow (n_2, o) \vdash E_3}{E_1 \vdash (e_1/e_2, x) \Rightarrow (n_1 + n_2 + 1, o) \vdash E_3}$$

$$\textbf{Rep}\ \frac{E_1 \vdash (e, x_1x_2y) \Rightarrow (n_1, x_1) \vdash E_2 \qquad E_2 \vdash (e^*, x_2y) \Rightarrow (n_2, x_2) \vdash E_3}{E_1 \vdash (e^*, x_1x_2y) \Rightarrow (n_1 + n_2 + 1, x_1x_2) \vdash E_3}$$

$$\neg\textbf{Rep}\ \frac{E_1 \vdash (e, x) \Rightarrow (n_1, f) \vdash E_2}{E_1 \vdash (e^*, x) \Rightarrow (n_1 + 1, \lambda) \vdash E_1}$$

$$\textbf{Neg}\ \frac{E \vdash (e, xy) \Rightarrow (n_1, x) \vdash E'}{E \vdash (!e, xy) \Rightarrow (n_1 + 1, f) \vdash E} \qquad \neg\textbf{Neg}\ \frac{E \vdash (e, xy) \Rightarrow (n_1, f) \vdash E'}{E \vdash (!e, x) \Rightarrow (n_1 + 1, \lambda) \vdash E}$$

$$\textbf{Atrib}\ \frac{v = E[\![e]\!]}{E \vdash ([x := e], y) \Rightarrow (1, \lambda) \vdash E[x/v]} \qquad \neg\textbf{Atrib}\ \frac{\mathit{unbound} = E[\![e]\!]}{E \vdash ([x := e], y) \Rightarrow (1, f) \vdash E}$$

$$\textbf{True}\ \frac{\mathit{true} = E[\![e]\!]}{E \vdash ([e], x) \Rightarrow (1, \lambda) \vdash E} \qquad \textbf{False}\ \frac{\mathit{false} = E[\![e]\!]}{E \vdash ([e], x) \Rightarrow (1, f) \vdash E}$$

$$\textbf{Adapt}\ \frac{\begin{array}{c}\langle A \downarrow a_1 \downarrow \ldots \downarrow a_p \uparrow e_1' \uparrow \ldots \uparrow e_q' \rangle \leftarrow e \in E[\![e_1]\!], \quad \text{where } E[\![e_1]\!] \equiv \text{language attribute} \\ v_i = E[\![e_i]\!], 1 \leq i \leq p \qquad v_j' = E_1[\![e_j']\!], 1 \leq j \leq q \\ [a_1/v_1, \ldots, a_p/v_p] \vdash (e, x) \Rightarrow (n, o) \vdash E_1\end{array}}{E \vdash (\langle A \downarrow e_1 \downarrow \ldots \downarrow e_p \uparrow b_1 \uparrow \ldots \uparrow b_q \rangle, x) \Rightarrow (n + 1, o) \vdash E[b_1/v_1', \ldots, b_q/v_q']}$$

Fig. 4. Semantics of Adaptable PEG.

in binary data specification. The second illustrates the specifications of static semantics of programming languages. And the third one shows how syntax extensibility can be expressed with Adaptable PEG.

4.1 Data Dependent Languages

As a motivating example of a context-sensitive language specification, Jim et alii [16] present a data format language in which an integer number is used to define the length of the text that follows it. Figure 5 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 sequel. The terminal *CHAR* represents any single character.

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

Fig. 5. An example of a data dependent language.

Using features from Adaptable PEG in the same language, we could replace the first two rules of Figure 5 by:

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

In an Adaptable PEG, every nonterminal has the language attribute as its first inherited attribute. The attribute g of the start symbol is initialized with the original PEG, 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, then g_1 will be equal to g together with a new rule that indicates that *strN* can generate a string with length n. These two functions are not formalized here for short.

4.2 Static Semantics

Figure 6 presents a PEG definition for a language where a block starts with a list of declarations of integer variables, followed by a list of update commands. For simplification, white spaces are not considered. An update command is formed by a variable on the left side and a variable on the right side.

Suppose that the context dependent constraints are: a variable cannot be used if it was not declared, and a variable cannot be declared more than once. The Adaptable PEG in Figure 7 implements these context dependent constraints.

block	$\leftarrow$	$\mathbf{\{}\ \mathit{dlist}\ \mathit{slist}\ \mathbf{\}}$	decl	$\leftarrow$	$\mathbf{int}\ \mathit{id}\ \mathbf{;}$
dlist	$\leftarrow$	$\mathit{decl}\ \mathit{decl}^*$	stmt	$\leftarrow$	$\mathit{id} = \mathit{id}\ \mathbf{;}$
slist	$\leftarrow$	$\mathit{stmt}\ \mathit{stmt}^*$	id	$\leftarrow$	$\mathit{alpha}\ \mathit{alpha}^*$

Fig. 6. Syntax of block with declaration and use of variables (simplified).

$\langle \mathit{block} \downarrow g\rangle$	$\leftarrow$	$\mathbf{\{}\ \langle \mathit{dlist} \downarrow g \uparrow g_1\rangle\ \langle \mathit{slist} \downarrow g_1\rangle\ \mathbf{\}}$
$\langle \mathit{dlist} \downarrow g \uparrow g_1\rangle$	$\leftarrow$	$\langle \mathit{decl} \downarrow g \uparrow g_1\rangle\ [g := g_1]\ (\langle \mathit{decl} \downarrow g \uparrow g_1\rangle\ [g := g_1])^*$
$\langle \mathit{decl} \downarrow g \uparrow g_1\rangle$	$\leftarrow$	$!(\ \mathbf{int}\ \langle \mathit{var} \downarrow g\rangle\ )\ \mathbf{int}\ \langle \mathit{id} \downarrow g \uparrow n\rangle\ \mathbf{;}$
		$[g_1 := g \oplus \mathit{rule}(\text{“}\langle \mathit{var} \downarrow g\rangle \leftarrow \#n\text{”})]$
$\langle \mathit{slist} \downarrow g\rangle$	$\leftarrow$	$\langle \mathit{stmt} \downarrow g\rangle\ \langle \mathit{stmt} \downarrow g\rangle^*$
$\langle \mathit{stmt} \downarrow g\rangle$	$\leftarrow$	$\langle \mathit{var} \downarrow g\rangle\ =\ \langle \mathit{var} \downarrow g\rangle\ \mathbf{;}$
$\langle \mathit{id} \downarrow g \uparrow n\rangle$	$\leftarrow$	$\langle \mathit{alpha} \downarrow g \uparrow \mathit{ch}_1\rangle[n = \mathit{ch}_1]((\langle \mathit{alpha} \downarrow g \uparrow \mathit{ch}_2\rangle[n = n + \mathit{ch}_2])^*$

Fig. 7. Adaptable PEG for declaration and use of variables.

In the rule that defines *dlist*, the PEG synthesized by each *decl* is passed on to the next one. The rule that defines *decl* first checks whether the input matches a declaration generated by the current PEG g. If so, it is an indication that the variable has already been declared. Using the PEG operator “!”, it is possible to perform this checking without consuming the input and indicating a failure, in case of repeated declaration. Next, a new declaration is processed and the name of the identifier is collected in n. Finally, a new PEG is built, adding a rule that states that the nonterminal *var* may derive the name n. The symbol “#” indicates that the string n must be treated as a variable.

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

4.3 Fortress Language

In Figure 8, we show how extensions defined to Fortress could be integrated into the language base grammar using Adaptable PEG. It is an adapted version of the original grammar proposed in the open source Fortress project, considering only the parts related to syntax extension. The rules can derive grammar definitions as the one presented in Figure 1.

Nonterminal *gram* defines a grammar which has a name specified by nonterminal *Id*, a list of extended grammars (*extends*) and a list of definitions. The grammar declared is located in the synthesized attribute t_2, which is a map of names to grammars. Note that language attribute is not changed, because the nonterminal *gram* only declares a new grammar that can be imported when needed. The attribute t_1 is also a map, and it is used for looking up available grammars.

Nonterminal *extends* defines a list of grammars that can be used in the definition of the new grammar. Every nonterminal of the imported grammar can be extended or used in new nonterminals definitions. Nonterminal *nonterm* defines a rule for extending the grammar, either extending the definition of a nonterminal or declaring a new one, depending whether the symbol used is |:= or ::=. If the definition of a nonterminal is extended, the function $\otimes$ is used to put together the original rule and the new expression defined. Otherwise, a grammar that has only one rule is created and stored in attribute g_1.

The rule of the nonterminal *syntax* has two parts: one is a parsing expression of a nonterminal (sequence of *part*) and the other is a transformation rule. A transformation rule defines the semantics of an extension, which is specified by nonterminal *sem*. Nonterminal *part* defines the elements that can be used in a parsing expression with addition that nonterminal can have aliases. Nonterminal *Base* generates nonterminal names, terminals and strings.

The rules in Figure 8 do not change the Fortress grammar directly; the extensions are only accomplished when an import statement is used.

$$
\begin{array}{ll}
\langle gram \downarrow g \downarrow t_1 \uparrow t_2 \rangle & \leftarrow \mathbf{grammar}\ \langle Id \downarrow g \uparrow id \rangle\ \langle extends \downarrow g \uparrow l \rangle \\
 & \qquad (\langle nonterm \downarrow g \downarrow t_1 \downarrow l \uparrow g_1 \rangle \\
 & \qquad\quad [t_2 := [id\ /\ t_2(id) \bigcup g_1]])^*\ \mathbf{end} \\
 & \\
\langle extends \downarrow g \uparrow l \rangle & \leftarrow \mathbf{extends}\ \pmb{\{}\ \langle Id \downarrow g \uparrow id_1 \rangle\ [l := [id_1]] \\
 & \qquad (,\ \langle Id \downarrow g \uparrow id_2 \rangle\ [l := l : [id_2])^*\ \pmb{\}} \\
 & /\ \ \lambda\ [l := []] \\
 & \\
\langle nonterm \downarrow g \downarrow t_1 \downarrow l \uparrow g_1 \rangle & \leftarrow \langle Id \downarrow g \uparrow id_1 \rangle\ |{:=}\ \langle syntax \downarrow g \uparrow e_1 \rangle \\
 & \qquad [g_1 := \{id_1 \leftarrow \otimes(t_1, l, id_1, e_1)\}] \\
 & /\ \ \langle Id \downarrow g \uparrow id_2 \rangle\ {::=}\ \langle syntax \downarrow g \uparrow e_2 \rangle\ [g_1 := \{id_2 \leftarrow e_2\}] \\
 & \\
\langle syntax \downarrow g \uparrow e \rangle & \leftarrow (\langle part \downarrow g \uparrow e_1 \rangle\ [e := e\ e_1])^* \Rightarrow \langle sem \downarrow g \rangle \\
 & \qquad (|\ (\langle part \downarrow g \uparrow e_2 \rangle\ [x := x\ e_2])^* \\
 & \qquad\quad [e := e\ /\ x] \Rightarrow \langle sem \downarrow g \rangle)^* \\
 & \\
\langle part \downarrow g \uparrow e \rangle & \leftarrow \langle single \downarrow g \uparrow e_1 \rangle?\ [e \leftarrow e_1\ /\ \lambda] \\
 & /\ \ \langle single \downarrow g \uparrow e_2 \rangle *\ [e := e_2^*] \\
 & /\ \ \langle single \downarrow g \uparrow e_3 \rangle +\ [e := e_3\ e_3^*] \\
 & /\ \ \langle single \downarrow g \uparrow e_4 \rangle\ [e := e_4] \\
 & /\ \ \neg \langle single \downarrow g \uparrow e_5 \rangle *\ [e := !e_5] \\
 & /\ \ \wedge \langle single \downarrow g \uparrow e_6 \rangle\ [e := !(!e_6)] \\
 & /\ \ \pmb{\{}\ (\langle part \downarrow g \uparrow e_7 \rangle [x := x\ e_7])^*\ \pmb{\}}\ [e := (x)] \\
 & \\
\langle single \downarrow g \uparrow e \rangle & \leftarrow \langle Id \downarrow g \uparrow id \rangle\ \mathbf{:}\ \langle Base \downarrow g \uparrow e \rangle \\
 & /\ \ \langle Base \downarrow g \uparrow e \rangle
\end{array}
$$

Fig. 8. Fortress syntax grammar

5 Conclusion and Future Work

The main goals for the model proposed in this work, as stated in Section 1, are: legibility and simplicity; and it must be suitable for automatic generation of syntactic analyzers. We have no proofs that these goals have been attained, however we believe that we have presented enough evidence for the first goal. Our model has a syntax as clear as Christiansen's Adaptable Grammars, since 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 Extended Attribute Grammars and Parsing Expression Grammars.

We keep some of the most important advantages of declarative models, such as an easy definition of context dependent aspects associated to static scope and nested blocks. 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.

When defining the syntax of extensible languages, the use of PEG has at least two important advantages. The production rules can be freely manipulated without the insertion of undesirable ambiguities, since it is not possible to express ambiguity with PEG. Extending a language specification may require the extension of the set of its lexemes. PEGs 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.

In order to know exactly the adaptations performed by an Adaptable PEG, 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.

We have not developed yet any proof that our model is suitable for automatic generation of syntactic analyzers. So the immediate next step of our work is to develop an efficient implementation for Adaptable PEG, considering frequent modifications on the set of production rules. In this implementation, we must offer an appropriate set of operations to manipulate the grammar. Grimm proposes an interesting mechanism for the tool Rats! [14], inserting labels in places that the rules may be modified. We may use a similar approach in our future implementation.

References

1. Eric Allen, Ryan Culpepper, Janus Dam Nielsen, Jon Rafkind, and Sukyoung Ryu. Growing a syntax. In *Proceedings of FOOL'2009*, 2009.
2. Pierre Boullier. Dynamic grammars and semantic analysis. Rapport de recherche RR-2322, INRIA, 1994. Projet CHLOE.
3. Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, June 2008.

4. Boris Burshteyn. Generation and recognition of formal languages by modifiable grammars. *SIGPLAN Not.*, 25:45–53, December 1990.
5. Boris Burshteyn. Ussa – universal syntax and semantics analyzer. *SIGPLAN Not.*, 27:42–60, January 1992.
6. S. Cabasino, Pier S. Paolucci, and G. M. Todesco. Dynamic parsers and evolving grammars. *SIGPLAN Not.*, 27:39–48, November 1992.
7. H. Christiansen. *The Syntax and Semantics of Extensible Languages.* Roskilde datalogiske skrifter. Computer Science, Roskilde University Centre, 1987.
8. H. Christiansen. A survey of adaptable grammars. *SIGPLAN Not.*, 25:35–44, November 1990.
9. Henning Christiansen. Adaptable grammars for non-context-free languages. In *Proceedings of IWANN'09*, pages 488–495. Springer-Verlag, 2009.
10. Tom Dinkelaker, Michael Eichberg, and Mira Mezini. Incremental concrete syntax for embedded languages. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC'11, pages 1309–1316, New York, NY, USA, 2011. ACM.
11. Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. Sugarj: library-based syntactic language extensibility. In *Proceedings of OOP-SLA'11*, pages 391–406, New York, NY, USA, 2011. ACM.
12. Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time, functional pearl. *SIGPLAN Not.*, 37(9):36–47, September 2002.
13. Bryan Ford. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, January 2004.
14. Robert Grimm. Better extensibility through modular syntax. *SIGPLAN Not.*, 41(6):38–51, June 2006.
15. Martin Jambon. How to customize the syntax of ocaml, using camlp5.
URL: http://mjambon.com/extend-ocaml-syntax.html, 2011.
16. Trevor Jim, Yitzhak Mandelbaum, and David Walker. Semantics and algorithms for data-dependent grammars. *SIGPLAN Not.*, 45:417–430, January 2010.
17. Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
18. Cornelius H. A. Koster. Affix grammars. In *Algol 68 Implementation*, pages 95–109. North-Holland, 1971.
19. Yaron Minsky. Ocaml for the masses. *Commun. ACM*, 54(11):53–58, November 2011.
20. Terence Parr and Kathleen Fisher. LL(*): the foundation of the ANTLR parser generator. *SIGPLAN Not.*, 46(6):425–436, June 2011.
21. Sukyoung Ryu. Parsing fortress syntax. In *Proceedings of PPPJ'09*, pages 76–84, New York, NY, USA, 2009. ACM.
22. John N. Shutt. Recursive adaptable grammars. Master's thesis, Worchester Polytechnic Institute, 1998.
23. John N. Shutt. What is an adaptive grammar?
URL: http://www.cs.wpi.edu/ jshutt/adapt/adapt.html, 2001.
24. Paul Stansifer and Mitchell Wand. Parsing reflective grammars. In *Proceedings of LDTA'11*, pages 10:1–10:7, New York, NY, USA, 2011. ACM.
25. Guy L. Steele, Jr. Growing a language. In *Addendum to OOPSLA'98*, pages 0.01–A1, New York, NY, USA, 1998. ACM.
26. David A. Watt and Ole Lehrmann Madsen. Extended attribute grammars. *Comput. J.*, 26(2):142–153, 1983.
27. Ben Wegbreit. *Studies in Extensible Programming Languages.* Outstanding Dissertations in the Computer Sciences. Garland Publishing, New York, 1970.