Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

The formalization and implementation of Adaptable Parsing Expression Grammars

Leonardo V.S. Reis [a,*], Roberto S. Bigonha [b], Vladimir O. Di Iorio [c], Luis Eduardo S. Amorim [c]

[a] Departamento de Computação e Sistemas, Universidade Federal de Ouro Preto, Brazil
[b] Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Brazil
[c] Departamento de Informática, Universidade Federal de Viçosa, Brazil

HIGHLIGHTS

- We propose an adaptable model based on Parsing Expression Grammars.
- We added attributes to PEG so extensibility is achieved by means of grammar attributes.
- The model is formally defined.
- The implementation of the adaptable model is detailed.
- The adaptable model preserves PEG legibility.

ARTICLE INFO

Article history:
Received 30 March 2013
Received in revised form 8 February 2014
Accepted 21 February 2014
Available online xxxx

Keywords:
Extensible languages
Adaptable grammars
PEG

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 analysers do not offer any adequate resources for the specification of extensible languages. When used in the implementation of features like syntax macro definitions, these tools usually impose severe restrictions. For example, it may be required that macro definitions and their use reside in different files; or it may be impossible to perform the syntax analysis in one single pass. We claim that one of the main reasons for these limitations is the lack of appropriate formal models for the definition of the syntax of extensible languages.

This paper presents the design and formal definition of *Adaptable Parsing Expression Grammars*, an extension to the *Parsing Expression Grammar* (PEG) model that allows the manipulation of its own production rules during the analysis of an input string. The proposed model compares favourably with similar approaches for the definition of the syntax of extensible languages. An implementation of the model is also presented, simulating the behaviour of packrat parsers. Among the challenges for this implementation is the use of attributes and on the fly modifications on the production rules at parse time, features not present in standard PEG. This approach has been used on the definition of a real extensible language, and initial performance tests suggest that the model may work well in practice.

© 2014 Published by Elsevier B.V.

* Corresponding author.
E-mail addresses: leo@decsi.ufop.br (L.V.S. Reis), bigonha@dcc.ufmg.br (R.S. Bigonha), vladimir@dpi.ufv.br (V.O. Di Iorio), luis.amorim@ufv.br (L.E.S. Amorim).

http://dx.doi.org/10.1016/j.scico.2014.02.020
0167-6423/© 2014 Published by Elsevier B.V.

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

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

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

1. Introduction

In recent years, we have witnessed important advances in parsing theory. For example, Ford created *Parsing Expression Grammars (PEG)* [1], an alternative to Context Free Grammars for describing syntax, and packrat parsers [2], top-down parsers with backtracking that allow unlimited lookahead and a linear parse time. Parr has devised a new parsing strategy called LL(*) for the ANTLR tool, which allows arbitrary lookahead and recognizes some context-sensitive languages [3]. Visser proposed SGLR [4,5], combining scannerless parsing with generalized LR parsing, and addressing important issues related to extensibility and compositionality of parsers. New engines have been created, such as YAKKER [6], which allows the control of the parsing process by arbitrary constraints, using variables bound to intermediate parsing results. All the advances listed above, however, do not include important features for the definition of extensible languages, especially when extensibility may be considered at parse time.

As a simple example of desirable features for the implementation of extensible languages, Fig. 1 shows an excerpt from a program written in the Fortress language [7]. Initially, a new syntax for loops is defined, and then this syntax is used in the same program. In details, line 1 introduces the definition of *ForLoops* as an extension of *Expression* and *Identifier*. Line 2 says that the grammatical rule that defines *Expr* is to be extended with the new right hand side defined on line 3. The new alternative, defined in line 3, begins with a terminal symbol *for*, followed by a symbol group, another terminal *do*, a nonterminal *Expr* and a terminal *end*. The names *i*, *e* and *block* are only aliases to the nonterminal, which may be used in the transformation part, not showed in Fig. 1. The expression between the symbols { and } is a sequence of zero or more patterns of the form $Id \leftarrow Expr$ with an optional comma and a required white space (nonterminal Space) at the end. The lines 12 to 14 show a use of this new syntax.

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. The Fortress interpreter, written with the tool Rats! [8], 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 [9]. A tool that is able to parse the program in Fig. 1 in one pass must be based on a model that allows syntax extensions.

Our concern is the lack of tools for helping the definition and implementation of extensible languages, such as Fortress. Our goal is to provide a solution for this problem, which may be used in practice. 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:

- to offer facilities for adapting the grammar during the parsing process, without adding too much complexity to the base model (in this case, PEG);
- to allow an implementation with reasonable efficiency.

Satisfying the first requirement, the model may be considered a viable formal approach to describe the syntax of extensible languages. An efficient implementation allows automatic generation of parsers that may be used in practice.

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, but 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 are variations on *Attribute Grammars* (AGs) [10]. First introduced by Knuth to assign semantics to context-free languages, AGs have subsequently been used for several other purposes, such as the specification of static semantics of programming languages [11,12].

Authors like Christiansen [13] and Shutt [14] 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

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

grammar for ADA, in which a single rule representing function calls has two and a half pages associated with it, just to describe the context conditions. He proposes an approach called *Adaptable Grammars* [15], which explicitly provides mechanisms within the formalism to allow production rules to be manipulated.

In an adaptable grammar, the task of checking whether a program variable used in an expression has been previously declared may be performed as follows. Instead of having a general rule like `variable -> identifier`, each variable declaration may cause the addition of a new rule to the grammar. For example, the declaration of a variable with name `x` causes the addition of 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 Fig. 1. However, as the model is based on CFG, undesirable ambiguities may arise when the set of production rules is modified. This problem must be properly addressed such as in [4,5,16]. 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 the same environment [14], although this task can be easily accomplished using attribute grammars and a symbol table.

1.2. From Adaptable Grammars to Adaptable PEGs

Similarly to *Extended Attribute Grammars* (EAGs) [17], in APEG, attributes are associated to the symbols of production rules. And similarly to *Adaptable Grammars* [15], the first attribute of every nonterminal symbol represents the grammar to be used when the respective nonterminal is interpreted. Whenever a nonterminal is to be interpreted, the production rule is fetched from its own first attribute, and not from a global static grammar, as in a standard PEG or standard CFG. Different grammars may be associated with other nonterminal symbols.

A fundamental difference between our approach and previous attempts to develop adaptable models is that we have chosen PEG as the base model, instead of CFG. This decision is based on some of the PEG standard features 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 will show that this feature may allow a simple solution for specifying the constraint that an identifier cannot be defined twice in the same environment, a problem related as hard to be solved by adaptable models based on CFG;
- The choice operator of PEG is ordered, giving more control of which alternative will be used. Our implementation relies on this feature to restrict the changes on existing rules: it is only allowed to perform additions on the end of a list of alternatives. This restriction allowed us to provide an implementation that does not need to build complex parsing tables whenever a rule is updated, favouring efficiency.

Choosing PEG and the restrictions explained above, our work forgoes the benefits of the modularity and declarative approach offered by CFGs. However, we argue that these decisions were important to provide an implementation that works with enough efficiency to be used in practice. As an evidence, we present the description of an extensible language and the results of performance tests.

1.3. Summary of contributions of the work

The main contributions of the work may be summarized as:

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 comparison with adaptable models based on CFG, which exhibits the advantages of the proposal.
4. A detailed description of an implementation.
5. The presentation of a specification of a real extensible language, using the implementation developed.
6. Performance tests on the implementation developed.
7. Formal proofs of some properties of the model.

Items 1 to 3 are the main contributions of the previous work [18]. The main contributions of the present work are items 4 to 7 and extensions to items 1 and 2. More specifically:

- New features are proposed for the model, such as facilities for binding variables to intermediate parse results (improvement on items 1 and 2 listed above).
- The development of an implementation for the model, including a concrete syntax for the language (item 4).
- The specification of the complete syntax of the extensible language SugarJ, using the developed implementation (item 5).

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

- Performance tests on the SugarJ implementation, comparing with an available implementation of the language (item 6). In order to accomplish this task, several examples of DSLs defined using SugarJ were tested.
- Formal proof of properties that are important for the memoization mechanism used by our implementation (item 7).

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. Section 5 describes the developed implementation and Section 6 describes performance tests. Conclusions and future work are discussed in Section 7. Appendix A formally defines and proves properties associated with memoization. Appendix B presents a simplified version of a grammar for the language implemented.

2. Related work

It seems that Wegbreit was the first to formalize the idea of grammars that allow the manipulation of the own set of rules [19], 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 allows 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* [15]. In previous works, he had used the term *Generative Grammar* [13]. Although Shutt has designated his own model as *Recursive Adaptable Grammar* [14], he has later used the term *Adaptive Grammar*. 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 [20], 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 [14], Shutt classifies adaptable models as *imperative* and *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* [21], using the model in the tool USSA [22]. 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 [23] 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* [24] 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 independence 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 [25]. It is essentially an Extended Attribute Grammar where the first attribute of every nonterminal 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 scoped 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* [14], where a single domain combines the syntactic elements (terminals), meta-syntactic (nonterminals and the language attribute) and semantic values (all other attributes).

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* [26], which provide developers with tools for importing syntax extensions and their desugaring as libraries. The authors use SDF [16] and Stratego [27] 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.

There are several other recent works involving grammar extensions. For example, in [28] and [29], the authors define a way to efficiently compose pre-compiled parse tables of a host language and extensions, producing a resulting parse table which is free of conflicts. Works like Silver [30], Kiama [31] and JastAdd [32] allow building attribute grammars that may be extended or embedded in a programming language. We will not discuss these works in detail because our focus is mainly adaptable models, i.e., the ones which allow modifying the own set of rules on the fly, at parse time.

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

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

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

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

Fig. 3. An example of an attribute PEG.

3. Definition of the model

Our work is inspired by *Adaptable Grammars*. The main difference is that, instead of a CFG as the base model, we use PEG. The adaptability of Adaptable PEGs is achieved by associating an attribute with every nonterminal to represent the grammar to be used when the respective nonterminal is interpreted. 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 of *Adaptable PEG*. Basic knowledge about Extended Attribute Grammars and Parsing Expression Grammars is desirable – we recommend [17] and [1].

3.1. PEG with attributes

Extended Attribute Grammar (EAG) is a model proposed by Watt and Madsen [17] for formalizing context sensitive features of programming languages. Compared to *Attribute Grammar* (AG) [10] and *Affix Grammar* [33], EAG is more readable and generative in nature.

Fig. 2 shows an example of an 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 a right side of a rule are called *applying positions*.

A reader not familiar with the EAG notation can use the following association with procedure calls of imperative programming languages, at least for the examples presented in this paper. The left side of a rule mimics 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 Fig. 2) represents 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 mimics the body of a procedure, in which every nonterminal is a new procedure call. Inherited attributes represent expressions that define the values of the arguments, and synthesized attributes are variables that store the returned values. For example, $\langle T \downarrow 2 * x_0 + x_1 \uparrow x_2 \rangle$ (second line of Fig. 2) represents 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 on their left sides indicate that the synthesized attribute is to be 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 compliance with the PEG's facilities that allow replacing recursion by the *repetition* operator "*", we propose that evaluation rules in Attribute PEGs may update the values of the attribute variables, treating them as variables as in an imperative language, so to provide an efficient implementation.

Fig. 3 shows an Attribute PEG equivalent to the EAG presented in Fig. 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 $[x_0 = 2 * x_0 + x_1]$.

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

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

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 [6].

Another improvement provided by EAG is the possibility of using the same attribute variable in more than one defining position in a rule. 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 [1] 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. $A : V_N \to \mathcal{P}(\mathbb{Z}^+ \times \{\uparrow, \downarrow\})$ is an attribute function that maps every nonterminal to a set of attributes. We denote the number of attributes of a nonterminal N as $arity(N)$. 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 symbol $\uparrow$ labels inherited attributes and $\downarrow$ labels synthesized attributes. $R : V_N \to \mathcal{P}_e$ 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 $\varepsilon_1, \ldots, \varepsilon_n$ are attribute expressions, then $f(\varepsilon_1, \ldots, \varepsilon_n)$ is also an attribute expression.

Suppose that e, e_1 and e_2 are *parsing expressions*, ε and ε_i are attribute expressions, which may use functions in F, and $\dagger$ is an element of the set $\{\downarrow, \uparrow\}$. The set of valid *parsing expressions* $(\mathcal{P}_e)$ can be recursively defined as:

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

In the list above, the seven first expressions are already present in the standard PEG model. Only *nonterminal expressions* are different because of the use of attributes. In Section 3.3, a formal definition of the semantics is presented, but for now it is interesting to highlight some aspects of the PEG model, when compared to context free grammars. The *ordered choice expression* represents an important difference between PEG and CFG. In a derivation step, two or more alternatives may be viable for a nonterminal symbol in CFGs. On the other hand, in PEG, alternatives of ordered choice expressions are always processed in the established order, providing a convenient mechanism to give a specific semantic interpretation in case of ambiguities. Another interesting construct that has no correspondence in the CFG model is the *not-predicate expression*. It allows unlimited lookahead, without consuming the input. A not-predicate results in success if the expression **does not** match the input. A *positive* predicate expression may be built applying *not-predicate* twice.

To the set of standard *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, using F functions. 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.

The example of Fig. 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 Fig. 3.

Nonterminal expressions are nonterminal symbols with attribute expressions. Without loss of generality, we will assume that all inherited attributes are listed 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 $\langle A \downarrow \vartheta_1 \downarrow \vartheta_2 \ldots \downarrow \vartheta_p \uparrow \varepsilon_1 \uparrow \ldots \uparrow \varepsilon_q \rangle \leftarrow e$ represents the rule for A and its attributes. We will also assume that the attribute expressions in defining positions are always a single variable.

In the rest of this paper, 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

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

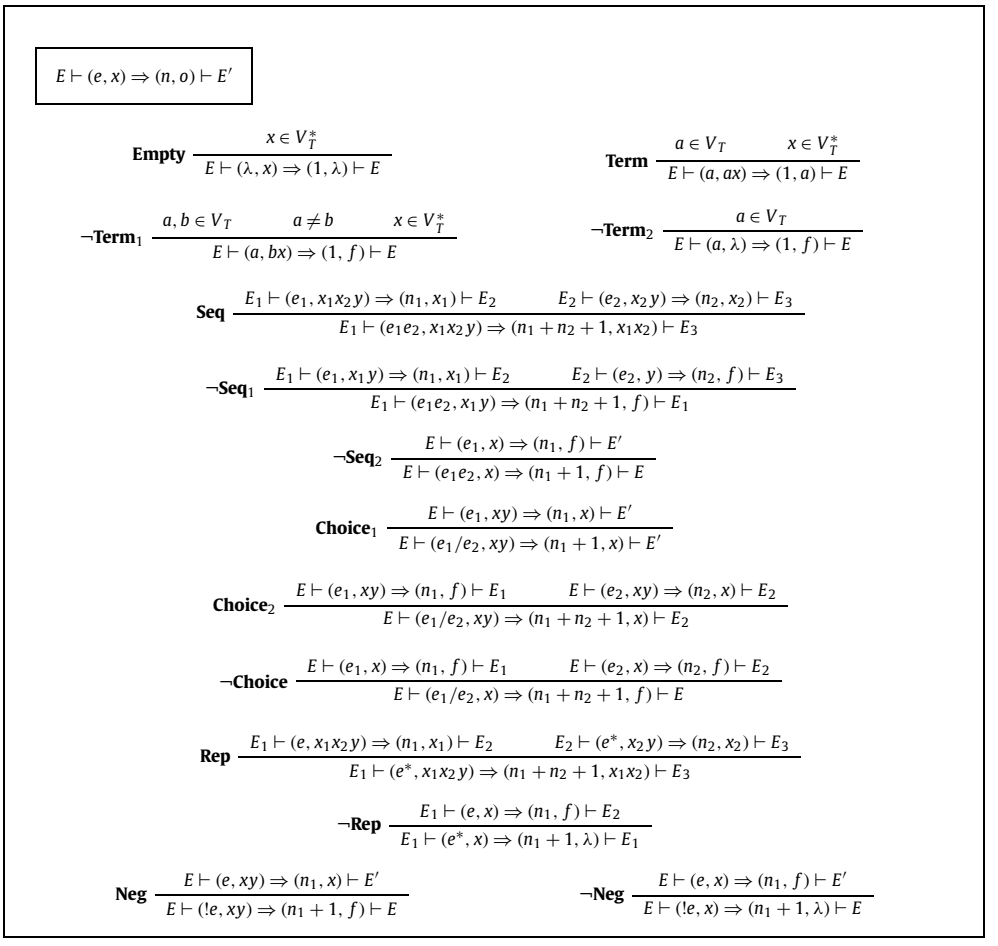

Fig. 4. Semantics of Adaptable PEG – Part I.

- 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.

3.3. Semantics of Adaptable PEG

An Adaptable PEG (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). Figs. 4 and 5 present the semantics of an Adaptable PEG. Almost all the formalization is related to PEG with attributes. Only the last equation in Fig. 5 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;
- $[\vartheta_1/v_1, \ldots, \vartheta_n/v_n]$ maps ϑ_i to v_i, $1 \leqslant i \leqslant n$, and other variables to *unbound*;
- $E[\vartheta_1/v_1, \ldots, \vartheta_n/v_n]$ is an environment which is equal to E, except for the values of ϑ_i which map to v_i, $1 \leqslant i \leqslant n$.

We write $E[\![\varepsilon]\!]$ to indicate the value of the attribute expression ε evaluated in the environment E. Figs. 4 and 5 define 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.

The equations in Fig. 4 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 is computed when it succeeds, a situation represented by rule **Seq**. If the first or the second subexpression of

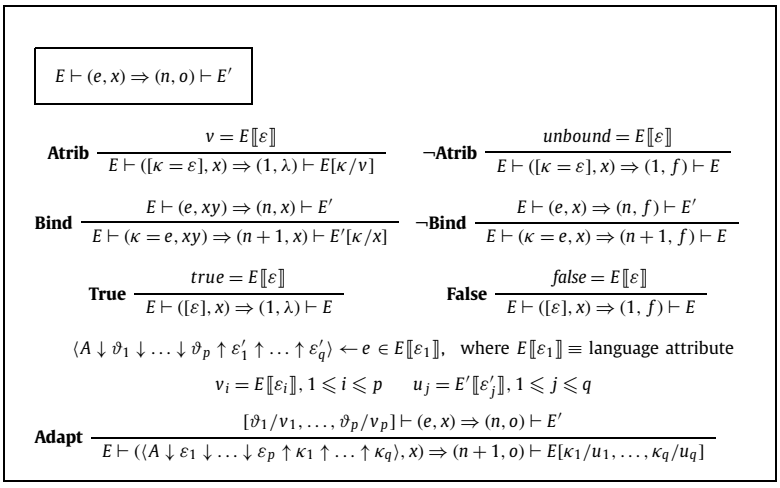

Fig. 5. Semantics of Adaptable PEG – Part II.

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 ¬**Seq**$_1$ and ¬**Seq**$_2$. A similar behaviour is defined for ¬**Term**$_1$ and ¬**Term**$_2$, when a terminal expression fails, for ¬**Rep**, when a repetition fails, and for **Choice**$_2$, when the first alternative of a choice fails. Rules **Neg** and ¬**Neg** show that the environment changes computed inside a not-predicate expression are not considered in the outer level, allowing arbitrary lookahead without collateral effects.

The equations in Fig. 5 describe the semantics of the operations that our model has added to the PEG model, namely *update*, *constraint* and *bind* expressions, and also the semantics of the evaluation of *nonterminal expressions*. Rules **Atrib** and ¬**Atrib** define the behaviour for update expression, and rules **True** and **False** represent predicate evaluation in constraint expressions. Rules **Bind** and ¬**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. Attribute values are associated with variables using an approach similar to EAG, but in a more operational way; it is also similar to *parametrized nonterminals* described in [6], 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. Rule **Adapt** is the only point in all the rules of Figs. 4 and 5 that is associated with the property of adaptability.

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

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

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. The language $L(G)$ is the set of words w that do not produce f (failure).

4. Usage examples

In this section, we present three examples of usage of Adaptable PEG. The first example is a definition of context dependent constraints commonly required 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 al. [6] 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]".

Fig. 6 shows how a similar language may be defined in an Attribute (nonadaptable) 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. 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 nonnegative, inside the repetition expression,

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

Fig. 6. An example PEG with attributes for a data dependent language.

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

Fig. 7. Using adaptability in the PEG of Fig. 6.

$$
\begin{array}{rcl|rcl}
block & \leftarrow & \mathbf{\{}\ dlist\ slist\ \mathbf{\}} & decl & \leftarrow & \mathbf{int}\ id\ \mathbf{;} \\
dlist & \leftarrow & decl\ decl^* & stmt & \leftarrow & id = id\ \mathbf{;} \\
slist & \leftarrow & stmt\ stmt^* & id & \leftarrow & alpha\ alpha^*
\end{array}
$$

Fig. 8. Syntax of block with declaration and use of variables (simplified).

$$
\begin{array}{lll}
\langle block \downarrow g \rangle & \leftarrow & \mathbf{\{}\ \langle dlist \downarrow g \uparrow g_1 \rangle \ \langle slist \downarrow g_1 \rangle\ \mathbf{\}} \\
\langle dlist \downarrow g \uparrow g_1 \rangle & \leftarrow & \langle decl \downarrow g \uparrow g_1 \rangle \ [g = g_1] \ (\langle decl \downarrow g \uparrow g_1 \rangle \ [g = g_1])^* \\
\langle decl \downarrow g \uparrow g_1 \rangle & \leftarrow & !(\ \mathbf{int}\ \langle var \downarrow g \rangle\ )\ \mathbf{int}\ \langle id \downarrow g \uparrow n \rangle\ \mathbf{;} \\
 & & \quad [g_1 = g \oplus rule(\text{“}\langle var \downarrow g \rangle \leftarrow \#n\ !alpha\text{”})] \\
\langle slist \downarrow g \rangle & \leftarrow & \langle stmt \downarrow g \rangle \ \langle stmt \downarrow g \rangle^* \\
\langle stmt \downarrow g \rangle & \leftarrow & \langle var \downarrow g \rangle = \langle var \downarrow g \rangle\ \mathbf{;}
\end{array}
$$

Fig. 9. Adaptable PEG for declaration and use of variables.

ensuring that the loop will end when n becomes 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 Adaptable PEG in the same language, we could replace the first two rules of Fig. 6 by the rule in Fig. 7. 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. The formalization of these functions is straightforward, and thus omitted here to save space. 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: $\langle strN \downarrow g \rangle \leftarrow CHAR\ CHAR\ CHAR$.

4.2. Static semantics

Fig. 8 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 has not been declared before, and a variable cannot be declared more than once. The Adaptable PEG in Fig. 9 implements these context dependent constraints.

In the rule that defines *dlist*, the language attribute synthesized by each *decl* symbol is passed on to the next *decl* symbol. The rule that defines *decl* first checks whether the input matches a declaration generated by the current PEG g. This is done through the nonterminal *var*, which is the nonterminal modified during the parsing to recognize only valid variables. If so, it is an indication that the variable has already been declared. Using the PEG operator "!", this checking is performed without consuming the input and it indicates a failure, in case of a repeated declaration. Otherwise, a new declaration is processed and the name of the identifier is collected in n. Finally, a new PEG is built, by the addition of a rule $\langle var \downarrow g \rangle \leftarrow \#n\ !alpha$ which states that the nonterminal *var* may derive the name n. The symbol "#" indicates that the identifier following it (n, in this case) must be treated as a variable inside a string. This operator is similar to the anti-quote operator found in most meta-programming systems.

The use of the PEG operator "!" on the rule defining *decl* prevents multiple declarations and 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.

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

$$
\begin{array}{lcl}
\langle gram \downarrow g \downarrow t_1 \uparrow t_2 \rangle & \leftarrow & \textbf{grammar}\ \langle Id \downarrow g \uparrow id \rangle\ \langle extends \downarrow g \uparrow l \rangle \\
 & & \quad (\langle nonterm \downarrow g \downarrow t_1 \downarrow l \uparrow g_1 \rangle \\
 & & \quad\ [t_2 = [id\ /\ t_2(id) \bigcup g_1]])^*\ \textbf{end} \\
 & & \\
\langle extends \downarrow g \uparrow l \rangle & \leftarrow & \textbf{extends}\ \{\ \langle Id \downarrow g \uparrow id_1 \rangle\ [l = [id_1]] \\
 & & \quad (\textbf{,}\ \langle Id \downarrow g \uparrow id_2 \rangle\ [l = l : [id_2]])^*\ \} \\
 & / & \lambda\ [l = []] \\
 & & \\
\langle nonterm \downarrow g \downarrow t_1 \downarrow l \uparrow g_1 \rangle & \leftarrow & \langle Id \downarrow g \uparrow id_1 \rangle\ \textbf{|:=}\ \langle syntax \downarrow g \uparrow e_1 \rangle \\
 & & \quad [g_1 = \{id_1 \leftarrow \otimes(t_1, l, id_1, e_1)\}] \\
 & / & \langle Id \downarrow g \uparrow id_2 \rangle\ \textbf{::=}\ \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 \\
 & & \quad (\textbf{|}\ (\langle part \downarrow g \uparrow e_2 \rangle\ [x = x\ e_2])^* \\
 & & \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 \textbf{?}\ [e \leftarrow e_1\ /\ \lambda] \\
 & / & \langle single \downarrow g \uparrow e_2 \rangle \textbf{*}\ [e = e_2^*] \\
 & / & \langle single \downarrow g \uparrow e_3 \rangle \textbf{+}\ [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)] \\
 & / & \{\ (\langle part \downarrow g \uparrow e_7 \rangle [x = x\ e_7])^*\ \}\ [e = (x)] \\
 & & \\
\langle single \downarrow g \uparrow e \rangle & \leftarrow & \langle Id \downarrow g \uparrow id \rangle\ \textbf{:}\ \langle Base \downarrow g \uparrow e \rangle \\
 & / & \langle Base \downarrow g \uparrow e \rangle
\end{array}
$$

Fig. 10. Fortress syntax grammar.

4.3. Fortress language

In Fig. 10, we show how extensions 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 Fig. 1.

The 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.

The 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 Fig. 10 do not change the Fortress grammar directly; the extensions are only accomplished when an import statement is used.

5. Implementation

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

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

Ford developed *packrat parsers* [2], which can be used for efficient implementation of PEGs. Packrat parsers allow unlimited lookahead with linear time processing, with 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

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

challenges not faced by a standard PEG. For instance, the use of attributes required 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 about tools implementing ideas related to the PEG model. Then we describe details of our own implementation, analysing the impacts of the new proposed features.

5.1. PEG-related implementations

As stated before, Ford developed *packrat parsing*, a method for implementing linear-time top-down parsers [2]. 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 [34], 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 developed Mouse [35], 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 LGI (Language Generator by Instil) [36], 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.

ANTLR [37] is a parser generator that recently introduced a novel parsing approach, called LL(*) [3]. 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! [8] 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 [38] 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 parsing expression grammars, 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

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

```
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] : digit<x2> ( digit<x1> { x2 = x2 * 10 + x1; } )* ;
8
9  digit returns [int x1] :
10   '0' { x1 = 0; }
11   / '1' { x1 = 1; }
12   / '2' { x1 = 2; }
13   / '3' { x1 = 3; }
14   / '4' { x1 = 4; }
15   / '5' { x1 = 5; }
16   / '6' { x1 = 6; }
17   / '7' { x1 = 7; }
18   / '8' { x1 = 8; }
19   / '9' { x1 = 9; } ;
20
21 CHAR : . ;
```

Fig. 11. Concrete syntax for the example of Fig. 6.

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.

5.2. Implementing an interpreter for APEG

The implementation of an interpreter for APEG was developed using the ANTLR parser generator [37], 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 abstract syntax tree is generated using the operators provided by ANTLR for AST construction. The generated AST is then interpreted, implementing the semantics of the model.

The source code of the project is available on a gitHub repository,[1] together with several examples.

5.2.1. Examples showing the concrete syntax

A grammar for the concrete syntax adopted is presented in Appendix B, inspired on the syntax of ANTLR. Inherited attributes are defined as parameters for nonterminal symbols. Synthesized attributes are defined as a list of return values. We defined also local attributes, which are synthesized attributes whose values are manipulated only locally. All attributes are typed, anticipating our desire for an efficient code generation. Assignment expressions are defined between `{...}` and conditional expressions are defined between `{?...}`. Desugaring syntax for optional, and-predicate, positive Kleene closure and character classes are also available.

In Fig. 11, the APEG of Fig. 6 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. The value of a synthesized attribute must always be the name of a variable. Character sequences are defined inside single quotes (e.g. `'['` and `']'`, in line 3). The dot operator ("`.`") matches any single character. All other elements in Fig. 11 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 Fig. 11 could, in principle, be used as a start symbol, each one describing a different language.

Fig. 12 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

[1] The link of the repository is https://github.com/leovieira/APEG.git.

```
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 : . ;
```

Fig. 12. APEG with user-defined functions and character classes.

```
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(concat('strN : ', concatN('CHAR  ', n)), ';')) ; }
8    '[' 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 : . ;
```

Fig. 13. Example with the set of production rules changed at parse time.

string into its integer representation. Also in line 8, there is a usage of a *character class*, defining the symbols `'0'` ... `'9'`, and a usage of the feature for binding a variable to the input matched by an APEG expression ("`t=...`").

Fig. 13 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 Fig. 7. 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 Section 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 Fig. 13, so the user does not need to declare it.

Consider the APEG expression in lines 6 to 8, in Fig. 13, 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 launched.
- 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.

5.2.2. 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 ⟨*nonterminal*, *position*⟩ to results, where a result is either *fail* or an integer *newPos*. Because of backtracking, a nonterminal may be applied to a 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.

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

As discussed in Section 5.1, 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 impossible. A nonterminal symbol may be used with different values for the inherited attributes. A result memoization associated with a pair $\langle nonterminal, position\rangle$ 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 $\langle nonterminal, inhAttr, position\rangle$ to either *fail* or $\langle synAttr, newPos\rangle$. 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 $\langle inhAttr, position\rangle$ to results of the form *fail* or $\langle synAttr, newPos\rangle$.

It is important to note that we consider that a set of values for a 3-tuple $\langle nonterminal, inhAttr, position\rangle$ 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 Appendix A, 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 3.3, some equations of Fig. 4 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 Fig. 4, 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.

5.2.3. Implementing adaptability

In the APEG model, the set of productions may change at parse 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 5.2.1, the APEG of Fig. 13 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 this 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 a 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 Fig. 13. 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 Fig. 14. 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 by the two Grammar objects.

In Section 5.2.2, 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.

6. Experimental 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 [26] in our model. SugarJ is an extension of the Java language with *sugar*

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

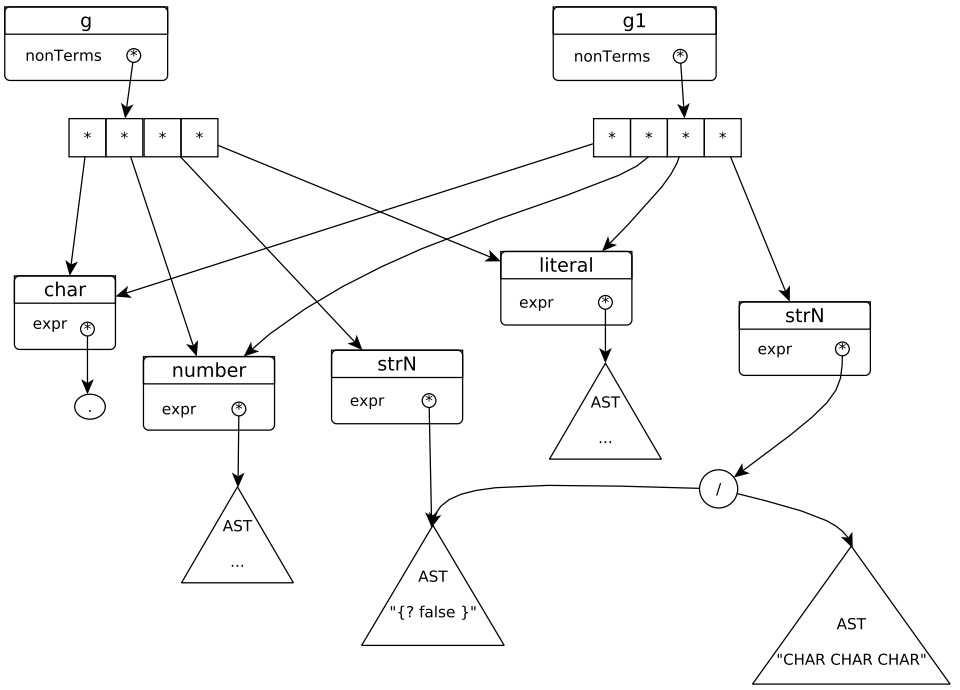

Fig. 14. Data structures representing a copy of an APEG with a modified production rule.

Table 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.

DSL	SugarJ implemented with SDF			SugarJ implemented with APEG		
	Adapt	Parse	Total	Adapt	Parse	Total
xml	17 948	280	18 228	16	1048	1064
closure	17 858	84	17 942	3	582	585
pair	22 920	41	22 961	3	368	371
n-xml	–	–	–	70	19 029	19 099
n-closure	22 975	378	23 353	33	1261	1294
n-pair	29 680	111	29 791	37	1130	1167
closure-pair-xml	39 537	401	39 938	52	1604	1656

libraries. Sugar libraries are units that encapsulate the definition of Java language extensions and they may be imported or composed for creating other extensions, in a modular way. 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 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 [26] and available in the SugarJ website. They define, respectively, an 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 code 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. The implemented programs and also the modified version of the SugarJ compiler for measuring the parse times are available in the gitHub repository of the APEG project.

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 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

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

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, 2.4 GHz Intel Core i5 running Ubuntu 12.04 with 6 GB of RAM. We have repeated the execution 30 times in a row and measured the average for *adapt* and *parse* time.

In Table 1, the first three lines, *xml*, *closure* and *pair*, present the parsing of a program that imports and uses only one DSL, namely the one listed in the first column. The results give 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. Although the time for actually parsing the files is smaller when using the original SugarJ compiler, the overall result (*adapt* + *parse*) shows that 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. In order to evaluate the impact of this feature, we have performed tests with several modules using the same DSL. In Table 1, the lines labelled *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. We collected large XML examples from an XML repository[2] 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 problem seems to be the large amount of memory required by the parser. 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. However, the performance of the APEG implementation was still better.

Finally, we tested the performance of the implementations when the input programs use different DSLs. The last line of Table 1 presents the parse time of a program which is a collection of modules that use different combinations of the three DSLs. 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. As we expected, the original SugarJ compiler was faster than our APEG prototype interpreter, when parsing the programs after the grammar was changed. However, in all scenarios, the overall execution time in our implementation was better, because the SugarJ compiler takes a considerable amount of time for adapting the grammar.

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

7. Conclusions and future work

The main motivation for this work was the current lack of appropriate tools for the definition and implementation of extensible languages. In order to solve this problem, we have proposed a new adaptable model based on Parsing Expression Grammar. The main goals of the proposed model, as stated in Section 1, are:

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

We present below arguments that may convince the reader that we have succeeded.

7.1. 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

[2] http://www.cs.washington.edu/research/xmldatasets/www/repository.html.

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

familiar with Extended Attribute Grammars and Parsing Expression Grammars. 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 (EAG). Additionally, we keep 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 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.

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, we believe that it is not a disadvantage as it is for imperative adaptable models based on CFG. Even for standard PEG (nonadaptable), 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 PEG has some advantages. Extending a language specification may require the extension of the set of its lexemes. PEG 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. PEGs are closed under union [1], so two or more language specifications can be combined without the restrictions imposed by models such as standard LR or LL. These advantages are highlighted in the development of the extensible language Fortress [9,39].

In [40], 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, that 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 can avoid most problems associated with subtle changes on a language definition, caused by the PEG semantics for ordering of alternatives. The examples in Sections 4, 5.2.1 and 6 present evidences that the restrictions are not so severe to the point of causing difficulties for the definition of extensions.

7.2. A reasonably efficient implementation

Our second 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 Section 5, we have presented an interpreter for our model. The tests described in Section 6 indicate that this interpreter may have a better performance than the available compiler for the extensible language SugarJ. These results show initial evidence that our approach may be used in practice.

7.3. Future work

One of the main tasks to be immediately performed is running tests with definitions of other extensible languages in APEG. The first one may be with a full description of the Fortress language. We expect to get results similar to the ones presented in Section 6.

Regarding the developed implementation, we are aware that generating code directly, which is the approach taken by the tools Pappy, Mouse, Rats! and ANTLR, might produce better efficiency results, but an important restriction to efficient code generation is the adaptability of the model. When production rules are changed at parse time, parts of the generated code may be invalidated. For the next implementations, we are considering a mixed approach, generating code for the initial grammar, and interpreting production rules that may be inserted at parse time. It is important to note that our implementation is a preliminary version of an interpreter. Applying a mixed approach for parser generation for APEG, we may achieve even better results than the ones presented in Section 6, but we consider that the developed APEG interpreter was already an important tool to validate our model, and also to investigate techniques for efficient implementation when production rules may change at parse time.

The semantics described in Figs. 4 and 5 define 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 infinite, it is not possible 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 for memoing Definite Clause Grammars [41] is similar to ours and they suggest that a super-linear behaviour does not happen in practice. They also show that, in general,

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

it is better memoing none or few nonterminals than all nonterminals. Our initial tests with the definition of the Fortress language suggest that it may be a case in which memoization is really important for the performance of the parser. This subject requires more investigation.

Currently, rules inserted in a PEG at parse time are represented as plain strings. In the future, we will evaluate the use of meta-programming techniques for the definition of the set of new rules. A promising approach may be based on the techniques used in tools like MetaAspectJ [42].

Appendix A. APEG properties associated with memoization

In order to our memoization scheme proposed in Section 5.2.2 work, we mentioned that the following property must hold: 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.

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. Follows directly from the definitions in Figs. 4 and 5. □

This lemma states that the model is deterministic. Suppose an APEG (V_N, V_T, A, R, S, F) and the judgement $E \vdash (e, x) \Rightarrow (n, o) \vdash E'$. Given values for e, x and E, if it is possible to calculate the values for n, 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 $\langle A \downarrow \vartheta_1 \downarrow \ldots \downarrow \vartheta_p \uparrow \varepsilon_1 \uparrow \ldots \uparrow \varepsilon_q \rangle$ 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 $\langle A \downarrow \varepsilon'_1 \downarrow \ldots \downarrow \varepsilon'_p \uparrow \kappa_1 \uparrow \ldots \uparrow \kappa_q \rangle$ 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 $p \geqslant 1$ and expressions $\varepsilon_1, \ldots, \varepsilon_p$, $\varepsilon'_1, \ldots, \varepsilon'_p$. 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 $1 \leqslant i \leqslant p$, 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 $1 \leqslant i \leqslant p$. Consider that the interpretation, if exists, of the parsing expression e in the environment $[\kappa_1/v_1, \ldots, \kappa_p/v_p]$ for an input string x is given by $[\kappa_1/v_1, \ldots, \kappa_p/v_p] \vdash (e, x) \Rightarrow (n, o) \vdash E''$. 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 $1 \leqslant j \leqslant q$, 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:

$$\textbf{Proof Tree 1}\quad \frac{\begin{array}{c}\langle A \downarrow \vec{\kappa_p} \uparrow \vec{\varepsilon''_q}\rangle \leftarrow e \in E[\![\varepsilon_1]\!],\ \text{where } E[\![\varepsilon_1]\!] \equiv \text{language attribute} \\ v_i = E[\![\varepsilon_i]\!], 1 \leqslant i \leqslant p \qquad u_j = E''[\![\varepsilon''_j]\!], 1 \leqslant j \leqslant q \\ [\kappa_1/v_1, \ldots, \kappa_p/v_p] \vdash (e, x) \Rightarrow (n, o) \vdash E''\end{array}}{E \vdash (\langle A \downarrow \vec{\varepsilon_p} \uparrow \vec{\vartheta_q}\rangle, x) \Rightarrow (n+1, o) \vdash E[\vartheta_1/u_1, \ldots, \vartheta_q/u_q]}$$

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

The interpretations above consume the same prefix o of the input x, using the same number of passes. 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 $[\kappa_1/v_1, \ldots, \kappa_p/v_p]$, 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'. □

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 5.2.2.

Appendix B. APEG grammar

We present below 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.

```
1  grammarDef: 'apeg' ID ';' functions? rule+ ;
2
3  functions : 'functions' (ID)+ ';' ;
4
5  rule : ID decls? ('returns' decls)? ('locals' declas)? ':' peg_expr ';' ;
6
7  decls : '[' varDecl (',' varDecl)* ']' ;
8
9  varDecl : type ID ;
10
11 type : ID ;
12
13 peg_expr : peg_seq ('/' peg_expr | ) ;
14
15 peg_seq : ( (ID '=')? peg_unary_op )+ ;
16
17 peg_unary_op :
18 peg_factor ('?' | '*' | '+')?
19 | '&' peg_factor
20 | '!' peg_factor
21 | '{?' cond '}'
22 | '{' (ID '=' expr ';')+ '}'
23 ;
24
25 peg_factor :
26 '\'' ALPHA* '\''
27 | ID ('<' actPars '>')?
28 | '[' (ALPHA '-' ALPHA)+ ']'
29 | '.'
30 | '(' peg_expr ')'
31 ;
32
33 actPars: (expr (',' expr )* )? ;
34
35 cond : ...
36 expr : ...
```

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020

References

[1] B. Ford, Parsing expression grammars: a recognition-based syntactic foundation, SIGPLAN Not. 39 (1) (2004) 111–122.
[2] B. Ford, Packrat parsing: simple, powerful, lazy, linear time, functional pearl, SIGPLAN Not. 37 (9) (2002) 36–47.
[3] T. Parr, K. Fisher, LL(*): the foundation of the ANTLR parser generator, SIGPLAN Not. 46 (6) (2011) 425–436.
[4] E. Visser, Scannerless generalized-LR parsing, Tech. Rep. P9707, Programming Research Group, University of Amsterdam, July 1997.
[5] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, E. Visser, Disambiguation filters for scannerless generalized LR parsers, in: R.N. Horspool (Ed.), Compiler Construction, CC 2002, in: Lecture Notes in Computer Science, vol. 2304, Springer, 2002, pp. 143–158.
[6] T. Jim, Y. Mandelbaum, D. Walker, Semantics and algorithms for data-dependent grammars, SIGPLAN Not. 45 (2010) 417–430.
[7] E. Allen, R. Culpepper, J.D. Nielsen, J. Rafkind, S. Ryu, Growing a syntax, in: International Workshop on Foundations of Object-Oriented Languages, 2009.
[8] R. Grimm, Better extensibility through modular syntax, SIGPLAN Not. 41 (6) (2006) 38–51.
[9] S. Ryu, Parsing Fortress syntax, in: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ '09, ACM, New York, NY, USA, 2009, pp. 76–84.
[10] D.E. Knuth, Semantics of context-free languages, Math. Syst. Theory 2 (2) (1968) 127–145.
[11] T. Reps, Optimal-time incremental semantic analysis for syntax-directed editors, in: Proceedings of the 9th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, POPL '82, ACM, New York, NY, USA, 1982, pp. 169–176.
[12] T. Reps, T. Teitelbaum, A. Demers, Incremental context-dependent analysis for language-based editors, ACM Trans. Program. Lang. Syst. 5 (3) (1983) 449–477, http://dx.doi.org/10.1145/2166.357218.
[13] H. Christiansen, The Syntax and semantics of extensible languages, Roskilde datalogiske skrifter, Computer Science, Roskilde University Centre, 1987, http://books.google.com.br/books?id=pd4ONAAACAAJ.
[14] J.N. Shutt, Recursive adaptable grammars, Master's thesis, Worchester Polytechnic Institute, 1998.
[15] H. Christiansen, A survey of adaptable grammars, SIGPLAN Not. 25 (1990) 35–44.
[16] E. Visser, A family of syntax definition formalisms, Tech. Rep. P9706, Programming Research Group, University of Amsterdam, August 1997.
[17] D.A. Watt, O.L. Madsen, Extended attribute grammars, Comput. J. 26 (2) (1983) 142–153.
[18] L. Santos Reis, R. Silva Bigonha, V. Iorio, L. Souza Amorim, Adaptable parsing expression grammars, in: F. Carvalho Junior, L. Barbosa (Eds.), Programming Languages, in: Lecture Notes in Computer Science, vol. 7554, Springer, Berlin, Heidelberg, 2012, pp. 72–86.
[19] B. Wegbreit, Studies in Extensible Programming Languages, Outstanding Dissertations in the Computer Sciences, Garland Publishing, New York, 1970.
[20] P. Stansifer, M. Wand, Parsing reflective grammars, in: Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications, LDTA '11, ACM, New York, NY, USA, 2011, pp. 10:1–10:7.
[21] B. Burshteyn, Generation and recognition of formal languages by modifiable grammars, SIGPLAN Not. 25 (1990) 45–53.
[22] B. Burshteyn, USSA – universal syntax and semantics analyzer, SIGPLAN Not. 27 (1992) 42–60.
[23] S. Cabasino, P.S. Paolucci, G.M. Todesco, Dynamic parsers and evolving grammars, SIGPLAN Not. 27 (1992) 39–48.
[24] P. Boullier, Dynamic grammars and semantic analysis, Rapport de recherche RR-2322, INRIA, projet CHLOE, 1994, http://hal.inria.fr/inria-00074352.
[25] H. Christiansen, Adaptable grammars for non-context-free languages, in: Proceedings of the 10th International Work-Conference on Artificial Neural Networks: Part I: Bio-Inspired Systems: Computational and Ambient Intelligence, IWANN '09, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 488–495.
[26] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann, SugarJ: library-based syntactic language extensibility, in: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, ACM, New York, NY, USA, 2011, pp. 391–406.
[27] M. Bravenboer, K.T. Kalleberg, R. Vermaas, E. Visser, Stratego/XT 0.17. a language and toolset for program transformation, Sci. Comput. Program. 72 (1–2) (2008) 52–70.
[28] A.C. Schwerdfeger, E.R. Van Wyk, Verifiable composition of deterministic grammars, in: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09, ACM, New York, NY, USA, 2009, pp. 199–210.
[29] A. Schwerdfeger, E. Van Wyk, Verifiable parse table composition for deterministic parsing, in: Proceedings of the Second International Conference on Software Language Engineering, SLE'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 184–203.
[30] E.V. Wyk, D. Bodin, J. Gao, L. Krishnan, Silver: an extensible attribute grammar system, Electron. Notes Theor. Comput. Sci. 203 (2) (2008) 103–116, http://dx.doi.org/10.1016/j.entcs.2008.03.047.
[31] A.M. Sloane, L.C.L. Kats, E. Visser, A pure object-oriented embedding of attribute grammars, Electron. Notes Theor. Comput. Sci. 253 (7) (2010) 205–219, http://dx.doi.org/10.1016/j.entcs.2010.08.043.
[32] T. Ekman, G. Hedin, The jastadd system – modular extensible compiler construction, Sci. Comput. Program. 69 (1–3) (2007) 14–26, http://dx.doi.org/10.1016/j.scico.2007.02.003.
[33] C.H.A. Koster, Affix grammars, in: Algol 68 Implementation, North-Holland, 1971, pp. 95–109.
[34] B. Ford, Packrat parsing: a practical linear-time algorithm with backtracking, Ph.D. thesis, Massachusetts Institute of Technology, 2002.
[35] R.R. Redziejowski, Mouse: from parsing expressions to a practical parser, in: Proceedings of the CS&P 2009 Workshop, Warsaw University, 2009, pp. 514–525.
[36] E.A.D. Guzmán, LGI (Language Generator by Instil), http://sourceforge.net/projects/instil-lang/, 2009.
[37] T.J. Parr, R.W. Quong, ANTLR: A predicated-LL(k) parser generator, Softw. Pract. Exp. 25 (1994) 789–810.
[38] C. Seaton, A programming language where the syntax and semantics are mutable at runtime, Tech. Rep. CSTR-07-005, University of Bristol, June 2007, http://www.cs.bris.ac.uk/Publications/Papers/2000702.pdf.
[39] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu Jr., S. Tobin-Hochstadt, The Fortress language specification, Tech. rep., Sun Microsystems, Inc., 2007, http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf.
[40] L.C.L. Kats, E. Visser, G. Wachsmuth, Pure and declarative syntax definition: paradise lost and regained, in: W.R. Cook, S. Clarke, M.C. Rinard (Eds.), Proceedings of OOPSLA 2010, ACM, 2010, pp. 918–932.
[41] R. Becket, Z. Somogyi, DCGs + memoing = packrat parsing but is it worth it?, in: P. Hudak, D. Warren (Eds.), Practical Aspects of Declarative Languages, in: Lecture Notes in Computer Science, vol. 4902, Springer, Berlin, Heidelberg, 2008, pp. 182–196.
[42] D. Zook, S.S. Huang, Y. Smaragdakis, Generating AspectJ programs with Meta-AspectJ, in: Generative Programming and Component Engineering: Third International Conference, GPCE 2004, in: LNCS, vol. 3286, Springer, 2004, pp. 1–19.

Please cite this article in press as: L.V.S. Reis et al., The formalization and implementation of Adaptable Parsing Expression Grammars, Science of Computer Programming (2014), http://dx.doi.org/10.1016/j.scico.2014.02.020