Defining the Syntax of Extensible Languages

Leonardo V. S. Reis
Universidade Federal de Ouro Preto
leo@decea.ufop.br

Vladimir O. Di Iorio
Universidade Federal de Viçosa
vladimir@dpi.ufv.br

Roberto S. Bigonha
Universidade Federal de Minas Gerais
bigonha@dcc.ufmg.br

ABSTRACT

The interest in *Domain-Specific Languages (DSLs)* has been increasing as a way of improving the productivity and readability of software. Some modern extensible languages offer facilities for building modular specifications for extensions, so they may be considered an interesting option for implementing domain specific languages. But there are at least two disadvantages that currently affect most extensible languages. First, their syntax is usually defined informally, because there is a lack of formal tools for the definition of extensible languages. Second, extensible languages are usually implemented in an ad-hoc and inefficient way.

In this paper, we show how the syntax of extensible languages like Fortress and SugarJ can be formally defined using a novel model designated *Adaptable Parsing Expression Grammars (APEG)*. The formal definitions of the languages help clarifying many aspects of their syntax. We also use an interpreter of the APEG model to parse programs of the languages, showing that the model can be used in practice to implement parsers for extensible languages.

Categories and Subject Descriptors

D.3.2 [**Programming Language**]: Language Classifications—*design languages, extensible languages*

General Terms

Languages

Keywords

extensible languages, APEG, PEG

1. INTRODUCTION

The use of *Domain-Specific Languages (DSLs)* has been considered a good way to improve readability of software, bridging the gap between domain concepts and their implementation, while improving productivity and maintainability [8, 10, 12]. However, the main weakness of *DSLs* is the cost of implementing and maintaining them [10, 12].

There are various methods for implementing *DSLs*, however, extensible languages seem to have several advantages when compared to the other approaches. One of the advantages is the possibility of implementing *DSLs* in a modular way. For example, Erdweg et alii show how *DSLs* can be implemented using the extensible language SugarJ [8], by means of syntax units designated as *sugar* libraries, which specify a new syntax for a domain concept. In [8], there is a careful discussion on the methods for implementing *DSLs* and why extensible languages with *sugar* libraries are a better choice. Tobin-Hochstadt et alii [18] also discuss the advantages of implementing *DSLs* by means of libraries.

As extensible languages arise as a good method for implementing *DSLs*, we have to answer the following question: how are extensible languages formally defined and implemented? Analysing extensible languages which have the properties required for defining *DSLs* in a modular way, such as the languages SugarJ [8], Fortress [2, 3, 16] and XAJ [15], we have noted a lack of formal tools for their definition, leading to ad-hoc implementations. The parsers available for these languages use a mix of a handwriting approach and automatic generation, first collecting all definitions of new syntax and, next, generating a new parser table at compile-time for parsing the code that uses the new syntax. Therefore, a new question arises: is there any advantage in formally defining extensible languages?

Adams [1] argues that a lack of standard formalization increases the complexity of writing parsers, and the parsers built often have different structure from the specification, which makes it difficult to determine if the implementation reflects the specification. These claims were made to motivate a grammar formalism to specify indentation rules of languages such as Haskell and Python, but they are also valid to motivate a formalism for extensible languages.

In this paper, we address the question if there is any advantage in formally defining extensible languages. We specify the extensible languages SugarJ and Fortress in *Adaptable Parsing Expression Grammars (APEG)* [14], an adaptable formalism which is based on *Parsing Expression Grammars (PEG)* [9] designed for formally defining extensible languages. We show that a formal definition has at least two advantages: it avoids doubts about the concrete syntax of the language; and it allows the automatic generation of a parser for the language, making it easier to construct compilers for them. As another result, we show how some features of the formalism chosen for specifying the languages, *APEG*,

SAC'14 March 24-28, 2014, Gyeongju, Korea.

http://dx.doi.org/10.1145/2554850.2554898

facilitate the definition of these languages.

The remaining of this paper starts discussing related work and the *APEG* model in Section 2. In Sections 3 and 4, we discuss the specification of the languages SugarJ and Fortress in *APEG*, respectively. Finally, Section 5 presents the conclusions.

2. RELATED WORK

Parsing of Extensible Languages

The idea of offering facilities to add syntactic constructions to a language remotes to the Lisp language and its dialects, such as Scheme and Racket [18]. These languages use the same format for data and program, S-expressions, thus they allow the implementation of a flexible and poweful macro system. Racket implements macros by functions from syntax to syntax that are executed at compile time when a macro use is reached by the macro expander. However, S-expressions impose restrictions on macro syntax and Racket lacks support for a high-level syntax formalism.

The implementation of parsers for extensible languages which do not use the same format for data and program is similar. In general, it uses a stepwise approach, which collects the grammar definitions and generates a parse table from the new rules collected. Then, the parser analyses the program using the table generated.

For example, the SugarJ compiler [8] uses a stepwise approach for parsing its syntax: parsing, desugaring, splitting and adaptation; and the compiler uses an incremental compilation process, in which every top-level entry is parsed at a time. A top-level entry in SugarJ is either a package declaration, an import statement, a Java type declaration, a declaration of syntactic sugar or a user-defined top-level entry introduced with a *sugar* library. Every top-level entry passes through the four stages before parsing other top-level entries.

In the parsing phase, a top-level entry is parsed with the current grammar, which reflects all sugar libraries currently in scope, and the other entries are parsed as a string. As a result of this stage, an abstract syntax tree is constructed with nodes of SugarJ and user-defined extension nodes. In the desugaring stage, user-defined extension nodes are desugared in nodes of SugarJ. The desugaring is done by the Stratego tool [4] (a language for program transformation) with the rules defined in a sugar library. In the splitting stage, the compiler splits every top-level entry into fragments of Java code, SDF [11] grammar (a syntax formalism whose parse algorithm allows ambiguous grammars) and Stratego rules. SDF grammar and Stratego rules produced in the splitting stage are used in the adaptation stage for modifying the current grammar of the parsing stage and the desugar rules in the desugaring stage. In the adaptation stage of the SugarJ compiler, the SDF grammar needs to be compiled to generate a parsing table at compile time, which will be used for changing the current grammar to parse the other top-level entries. This approach only works because the current grammar in SugarJ is only changed after parsing top-level entries, which are disposed according to the structure of a file. For example, a file starts with a package declaration, next is the import statements, then classes declaration and so forth. This allows parsing, for example, an import statement, changing the current grammar and parsing the next top-level entry, that could be a new syntax defined by the user.

Fortress is also an extensible language which does not use the same format for data and program. To parse the Fortress language, a two-phase approach is taken [3]: in the first step, all the grammars except the action part (a rule that describes how to desugar the extension in terms of Fortress core syntax) and the main expression are parsed. In this step, the action part and the main expression are parsed as Unicode Strings. Next, the parser computes the set of extensions that are available and generates another parser that is used for parsing the action part and the main expression, which may use the new syntax. This strategy only works because all the grammar definitions must come before the main expression, so they can be processed first. Similary, the parser of the XAJ language [15] collects the new syntactic constructions defined by *syntax classes* and generates a new parser using the PPG tool [5]. The generated parser is used for parsing the program, which may use the new syntax.

The approach described for parsing SugarJ, Fortress and XAJ has some problems: the lack of a formalism for defining the extensibility aspects of the language makes it impossible to automatically generate the parser, increasing the complexity of writing the parser, and it makes it difficult to understand the language; the parser implementation may not conform with the language specification; and the generation of the entire parser table every time the language is extended with few rules may be inefficient.

Models for Defining Extensible Languages

As extensible languages may change their own set of rules during the derivation process, the formalisms more appropriate to specify their syntax may be the ones which also allow modifying the own set of grammar rules. Christiansen [7] proposes a formalism with these features, called *Adaptable Grammars*, which is essentially an Extended Attribute Grammar [19] where the first attribute of every nonterminal symbol is inherited and represents the *language attribute*. The language attribute contains the set of rules allowed in each derivation. The initial grammar works as the language attribute for the root node of the parse tree, and new language attributes may be built and used in different nodes. Each grammar adaptation is restricted to a specific branch of the parse tree. One advantage of this approach is that it is easy to define statically scoped dependent relations, such as the block structure declarations of several programming languages.

Shutt [17] observes that Christiansen's *Adaptable Grammars* inherit the lack of 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 (RAGs)* [17], where a single domain combines the syntactic elements (terminals), meta-syntactic (nonterminals and the language attribute) and semantic values (all other attributes). One problem of RAG is the difficulty to check for forward references, which is important for defining the syntax of the Fortress language, for example. Modelling forward references is also difficult with Christiansen's *Adaptable Grammars*.

Carmi [6] argues that existing adaptable formalisms do not handle well forward references, such as goto statements that precede label declarations, and extensible languages

```
Sum[Grammar g]:
   Num<g,g1> (Add_Num<g1>)*;

Add_Num[Grammar g]:
   '+' Num<g,g1>;

Num[Grammar g] returns[Grammar g1]:
     [0-9]+ {g1 = g;}
   / 'adapt' {g1 = adapt(g,
        'Add_Num[Grammar g]: \'-\' Num<g,g1>;');};
```

Figure 1: An example of grammar using APEG

with features like macro syntax and its expansion. Thus, he proposes a new model, called *AMG*. *AMG* is driven by the parsing algorithm and the derivation must be rightmost. Nonterminal symbols of *AMGs* may have annotations and a special type of rule, a multi-pass rule. A multi-pass rule is similar to a simple rule, however, when the parser reduces using this type of rule, the annotation of the rule is put as a prefix of the input to be parsed. The multi-pass rules together with nonterminal annotations allow parsing a prefix of the input string and then reparse it using the same grammar rules or a different set of rules, allowing to handle forward references, macro definitions and expansion.

Adaptable Parsing Expression Grammar

Adaptable Parsing Expression Grammars or APEG [14] is an extension of PEG [9], which allows to change the set of rules during parsing. APEG adds attributes to the nonterminal symbols and achieves adaptability through a special inherited attribute, called *language attribute*. The language attribute is the first attribute of every nonterminal and it represents the current grammar, containing all set of rules. This is the same idea of the attribute grammar used for Christiansen's *Adaptable Grammars* [14], thus it has the same advantages. For illustrating APEG, Figure 1 shows a toy example of a grammar for parsing a sum expression, which may extend itself with a rule for minus expression during parsing. The concrete syntax of all APEG examples presented in this paper is the one adopted by the current version of the available APEG interpreter[1]. It is slightly different from the original syntax proposed [14].

The inherited attributes immediately follow the nonterminal name. The nonterminal *Sum* has only the language attribute, specified following the nonterminal name between the symbols [and]. The definition of this nonterminal begins with a number followed by a sequence of zero or more *Add_Num*. The nonterminal *Num* has two attributes. One inherited, which is the language attribute, and a synthesized attribute, which is a possibly modifiable grammar (the synthesized attributes are defined in the returns clause). In the definition of the nonterminal *Sum*, the grammar returned by *Num* is passed to the nonterminal *Add_Num*, which may have a new rule for this nonterminal. The attributes of a nonterminal in a right hand side of a rule are specified between the symbols < and >, beginning with the inherited attributes following by the synthesized ones, as the use of the nonterminal *Num* in the definition of the nonterminal *Sum*.

[1]The interpreter and all code developed is found in github repository at http://github.com/leovieira/APEG.git

The definition of the nonterminal *Num* has two choices. The first one specifies a sequence of at least one number followed by an update expression, which sets the value of the synthesized attribute *g1* to the same value of *g*. An update expression is defined between braces. The second choice of *Num* is a string literal, **adapt**, representing a mark to extend the grammar, in this example. The function adapt, used in the update expression, receives a grammar and a string representing the rules to be added to the grammar and returns a new grammar, which includes the new rules. It is only possible to modify the grammar creating new rules or adding new alternatives at the end of existing rules. In this case, a new alternative is added at the end of the rule for the nonterminal *Add_Num*.

Then, if we have the string

$$adapt + 2 - 3 - 4 + 5$$

as input, the parser will work as following: it begins parsing with the nonterminal *Sum* with the initial grammar, containing only the rules of Figure 1. After this, it tries to match the nonterminal *Num*, which receives the initial grammar. The second choice of the nonterminal *Num* will be used, consuming the prefix *adapt* of the input and returns a new grammar which has a new rule for *Add_Num*. The nonterminal *Sum* passes this grammar to the nonterminal *Add_Num*. Now, the new grammar has a rule for minus expression, then the remaining of the input is correctly parsed.

This toy example shows some important features that APEG adds to the PEG model. The update expression is a new feature added by APEG, which is used, in the example, for changing the grammar and to return its value as a synthesized attribute. However, the adaptability is effectively done only when this attribute is passed to the nonterminal *Add_Num* as its language attribute. APEG has two other features that help to understand the examples in this paper. APEG has constraint expressions, which allow to check if an expression is evaluated to the *true* value. Also, APEG has bind expressions, which capture the expression matched and bind it to a variable name. For example, the rule

$$ch1 = [A - Z]\ ch2 = [A - Z]\ \{?\ ch1\ !=\ ch2\ \}$$

binds the first symbol of the input to the variable $ch1$ and the second symbol of the input to the variable $ch2$. At the end, this rule checks if the value of these variables are different. Additionally, we may omit the language attribute if it is only passed on without modifications.

A formal definition of the semantics of APEG and more details of the formalism is presented in [14].

3. DEFINING THE SYNTAX OF SUGARJ

SugarJ [8] is a language recently developed by Erdweg et alii to experiment and validate their idea of *sugar* libraries. The main aim of *sugar* libraries is to encapsulate the definition of extensions for the Java language in units that may be imported or composed for creating other extensions, in a modular way. Figure 2 shows an example of a definition of a *sugar* library for a new syntax for pairs, creating two new rules: a rule for the definition of pair types in line 5, $type \rightarrow$ '(' *type* ',' *type*;')', and a rule for using pair expressions in line 6, $expr \rightarrow$ '(' *expr* ',' *expr*;')'. Note that the definition of a rule in SugarJ is in an order that is reverse to the one commonly used in context-free grammars.

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

Figure 2: A definition of sugar library for Pairs in SugarJ.

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

Figure 3: Use of the pair syntax.

A definition of a *sugar* library does not extend immediately the language, an extension is created only when a module or file imports a *sugar* library. As an example, Figure 3 shows a program that imports the *sugar* library *Pair* in line 1. After this import statement, the parser effectively extends the language, adding the two rules defined by the *sugar* library. The rules added are used for correctly parsing the attribute *p* of the class *Test* in line 5.

We have defined the syntax of SugarJ in *APEG* and used an experimental version of an interpreter of the model to automatically parse the language. As *APEG* is based on *PEG*, we adapted an implementation of the Java grammar for the Mouse project [13], which is also based on PEG, and extended it to allow the definition of *sugar* libraries. Figure 4 shows the syntax definition of *sugar* libraries. As a definition of a *sugar* library does not extend immediately the grammar, the nonterminal *sugar_decl* only collects the name of the *sugar* library and the rules in a single string. This information is passed through the rules of Figure 4 as synthesized attributes and are used later in an import statement to extend the grammar. Differently from the implementation of SugarJ, which defines the rules in SDF [11] syntax, we have decided to use the PEG style for defining the rules of SugarJ, because of the base model. Otherwise, we would have to translate the context-free rules to PEG and this would add complexity that is out of the scope of the project.

We have also modified the nonterminal that represents type declarations to allow declarations of *sugar* libraries. Therefore, the rule of this nonterminal has a new choice:

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

A type declaration has two inherited attributes, the package name and a map from names to rules, and one synthesized attribute, a map from *sugar* names to their corresponding definition. So, when a *sugar* library is defined by the user, a type declaration returns a new map associating the *sugar* library to its rules. Figure 5 shows a new syntax definition for

```
sugar_decl returns[String name, String rules]:
   'sugar' name=Id '{' defining_syntax<rules> '}';

defining_syntax returns[String rules]:
   'context-free syntax' peg_rule<rules>;

peg_rule returns[String rule]:
   {rule = '';} (peg_expr<s> '->' id=Id ';'
                  {rule += id + ':' + s + ';'; })*;

peg_expr returns[String rule]:
   peg_seq<rule> ('/' peg_seq<r>
                      { rule += ' / ' + r; })*;

peg_seq returns[String s]:
   peg_predicate<s> (peg_predicate<s1>
                       { s += ' ' + s1; })*
   / { s = '';};

peg_predicate returns[String r]:
     '!' peg_unary_op<s> { r = '!' + s; }
   / '&' peg_unary_op<s> { r = '&' + s; }
   / peg_unary_op<r>;

peg_unary_op returns[String r]:
     peg_factor<s> '*' { r = s + '*'; }
   / peg_factor<s> '+' { r = s + '+'; }
   / peg_factor<s> '?' { r = s + '?'; }
   / peg_factor<r>;

peg_factor returns[String r]:
     r=(peg_literal / Id / '.')
   / '(' peg_expr<s> ')' { r = '(' + s + ')' };
```

Figure 4: Syntax definition of sugar libraries.

```
compilation_unit[Grammar g, Map m] returns[Map m1]:
   package_decl<p>? (import_decl<g, m, g1> g=g1;)*
   (type_declaration<g, p,m,m1> m=m1;)*;

import_decl[Grammar g, Map m] returns[Grammar g1]:
   'import' n=qualified_id ';' { g1=adapt(g,m.get(n)); };
```

Figure 5: Syntax definition of compilation units.

a compilation unit, highlighting the possible changes on the grammar rules. The nonterminal *compilation_unit* receives a map of *sugar* libraries and passes it to the nonterminal *import_decl*. The nonterminal *import_decl* checks if the file is importing a *sugar* library and adapts the grammar, if necessary, using the function *adapt*. The adaptable grammar is returned as a synthesized attribute and passed to the nonterminal *type_declaration*, which may use the new syntax.

Every file is parsed by the nonterminal *compilation_unit*. So, for parsing our examples of Figures 2 and 3, the compiler parses the file of Figure 2 with the nonterminal *compilation_unit*, which receives the initial grammar of the SugarJ language and an empty map without any definition of *sugar* libraries. As a result, the nonterminal *compilation_unit* returns a new map that has an entry for the new *sugar* library *Pair*. This new map is used in the import declaration for parsing the file of Figure 3, which modifies the grammar with the new rules of the *Pair* syntax.

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

Figure 6: Composition of more than one sugar library.

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

Figure 7: Definition of the closure syntax.

Composing sugar libraries

Sugar libraries are composed by importing more than one *sugar* library into the same file. As an example, Figure 6 shows a program that uses the syntax of pairs and closures. The compiler extends the grammar with the rules of the syntax of closures defined in Figure 7 when parsing the first import statement, in line 1. Next, the grammar is also changed with the syntax of pairs when parsing the import in line 2. The modified grammar, which has the rules of pairs and closures syntax, is used for parsing the class *Partial*.

The implementation of SugarJ uses SDF [11] and it may be necessary to write rules for disambiguities when composing various grammars. However, it is impossible to prevent all the possibilities of ambiguities and conflicts, consequently composing two or more *sugar* libraries is not always possible. APEG avoids ambiguities using ordered choice, so composition is, in principle, always possible using APEG. On the other hand, if there is some overlapping between the rules of two or more extensions, the first option on the ordered choice clause will prevail. As new options are always inserted at the end of a rule, a user may change the priority altering the order of the import declarations. It seems a simple task, but it is not always easy to understand the interactions between overlapping rules.

Paradox Syntax × Semantics

In [8], the authors claim that is it not clear how to support "local" imports, which extend the language. They give an example of an extension s_1 after s_2 whose semantics swap the execution order of the statements s_1 and s_2. They argue that the code

("12", 34) after **import** syntactic.Pair

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

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

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

Figure 8: Definition of a for loop in Fortress.

4. THE SYNTAX OF FORTRESS

The main goals of the design of the Fortress language were to emulate mathematical syntax and to be extensible [3]. These two goals imposed additional difficulties to build a parser for the language. Defining the extensibility system in a formalism which supports unlimited lookahead would bring some advantages [3, 16], therefore, the core language and extensions to it are specified in PEG [9].

Figure 8 shows an example of the definition of an extension in Fortress. Line 1 defines a new grammar, called *ForLoop*, which may use symbols of two other grammars, *Expression* and *Identifier*. The Fortress language has two types of nonterminal specifications: the extension of an existing nonterminal, using the symbol |:= (line 2) or the definition of a new one (line 4). The right hand side of a rule has two parts, a parsing expression and an action part. The parsing expression defines the syntax of the new construct in a PEG style and the action part specifies how to translate the syntax into the core language. The action part is everything after the symbol ⇒. It is possible to use aliases associated to terminal or nonterminal symbols, creating references for them, which can be used in the action part. Figure 8 shows an example in which the nonterminal *forStart* is referenced by *b* in line 2.

Figure 9 shows part of an APEG syntax definition of the Fortress language. Similarly to the SugarJ definition, the nonterminal *gram_def* defines the syntax of an extension in Fortress and returns a map with the new entry for it. However, differently from the SugarJ definition, a grammar in Fortress allows self recursion and may use the new syntax

```
gram_def[Grammar g, Map m] returns[Map m1]:
   'grammar' n=id gram_ext<m,l>? &collect_gram<r>
        {g1 = adapt(g, r + allRules(l));}
        nonterm_def<g1>* 'end' {m1 = put(m,n,r);};

gram_ext[Map m] returns[List l]:
   'extends' qualified_names<m,l>;

collect_gram returns[String r]:
   {r = '';} (non_def<n,r> {r += 'n : r;';})*;

non_def return[String n, String r]:
     n=id '|:=' syn<r> ('/' syn<r1> r += '/' + r1;)*
   / n=id '::=' syn<r> ('/' syn<r1> r += '/' + r1;)*;

syn returns[String r]:
   peg_seq<r> '=>' '<[' !']>' . ']>';

nonterm_def[Grammar g]:
     id '|:=' syntax ('/' syntax)*
   / id '::=' syntax ('/' syntax)*;

syntax:
   peg_seq<r> '=>' '<[' expr ']>';
```

Figure 9: APEG formalization of Fortress language.

in the action part. Therefore, it is necessary to collect the grammar rules before parsing the code. We use the and-predicate operator "&" to specify this, collecting the grammar rules while ignoring the action part. Next, we reparse the program with the modified grammar. Note that when collecting the grammar rules using the and-predicate operator, the action part is parsed as a string, ignoring every symbol between '<[' and ']>' (nonterminal *syn*). After collecting the rule definitions, we adapt the grammar and generate a new grammar *g1*. This new grammar is passed to the nonterminal *nonterm_def*, which passes it to its children, allowing to parse the action part (nonterminal *syntax*). Therefore, the action part may use the new syntax being defined.

The using of the and-operator, which allows an infinite lookahead, is very important to handle the self recursion, a kind of forward reference. This operator is inherited by APEG from PEG and it is implemented efficiently with the packrat algorithm, using memoization.

Combining Grammars

Figure 10 shows an example of composition of grammars in Fortress. Grammar A defines a new nonterminal Nt and grammar B extends grammar A. Fortress allows the use of the syntax of A in the action part of B, as in line 6. Grammar C extends B and can use its syntax, however, C cannot use the syntax of A because it does not explicitly extend grammar A. In [3], the authors report that they need to resolve the set of extensions (for example, in grammar C it may use syntax defined in C or B, but not in A) to generate the table for parsing the action part and this is not an easy task.

Using the APEG model, defining the task described above is simple and clear. We adapt the grammar, adding the rules of the grammars specified in the *extends* part. For

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

Figure 10: Combining grammars.

example, parsing the grammar B, we add only the rules of A and when parsing the grammar C, we add only the rules of B. Another difficulty reported in [3] is how to compose the rules with multiple extensions, as defined in grammar D. In APEG, to have the same behaviour of the original Fortress implementation, we must adapt the grammar in the following order: first, we add the rules of the grammar which is currently being defined (rules of D in the example), next the grammars in the extends part in the same order that is specified (first, it adds rules of B and next of C, for the example of Figure 10).

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

Other Features

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

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

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

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

The parser available for Fortress parses an operator name as an identifier and calls an external function to check if it is a valid operator name. This procedure could also be implemented in Fortress using a definition similar to the one presented above, but the developers apparently favored

efficiency at the expense of clarity.

This example is presented only to show the power of APEG, using it for problems other than adapting grammars.

5. CONCLUSIONS

We introduced the paper inquiring if there are any advantages in using an adaptable model for defining the syntax of extensible languages. In order to answer this question, we implemented two relevant extensible languages in the adaptable model APEG and used its interpreter for automatically parsing programs of these languages. Therefore, the main advantage provided is the possibility of automatic generation of the parser. It is well-known that automatic generation of parsers reduces the complexity of building parsers, makes it easier to correct bugs and to change the concrete syntax of the language during the development phase. Finally, the implementation conforms with the specification, so if the specification is correct, then also the generated parser will be.

Moreover, our specifications define clearly what rules are available at a given moment during the parsing. It allows to resolve the false paradox about the local import raised by Erdweg et alli [8]. Also, the semantics of the combination of Fortress grammars is clear in the APEG specification, showing explicitly what set of rules will be used when a grammar extends another.

Forward reference is reported as difficult to handle with adaptable models. The definition of grammars in Fortress has a kind of forward reference, where the action part may use syntax that is defined later. Therefore, it is necessary to use a multi-pass approach. We showed that the predicate operator & of APEG allows simulating a multi-pass parse, handling the forward reference. APEG is powerful enough to formalize features that are difficult with CFG, as the definition of Fortress operator names.

We have provided enough evidence about using an adaptable model for specifying and implementing extensible languages and also showed that the APEG model is a good choice for it. However, this work does not study how is the efficiency of the implementation of these languages, comparing with the original one. Our experience using APEG on this work seems that APEG may be efficient enough for using in practice, but a careful study in this direction is needed. This is the immediate future work.

6. ACKNOWLEDGMENTS

The authors would like to thank the Programa de Pós Graduação em Ciência da Computação – Universidade Federal de Minas Gerais and Funarbe (Fundação de Apoio à Universidade Federal de Viçosa).

7. REFERENCES

[1] M. D. Adams. Principled parsing for indentation-sensitive languages: revisiting landin's offside rule. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '13, pages 511–522, New York, NY, USA, 2013. ACM.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification Version 1.0, Mar. 2008.

[3] E. Allen, R. Culpepper, J. D. Nielsen, J. Rafkind, and S. Ryu. Growing a syntax. In *International Workshop on Foundations of Object-Oriented Languages*, 2009.

[4] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Science of Computer Programming*, 72(1-2):52–70, June 2008.

[5] M. Brukman and A. C. Myers. PPG: a parser generator for extensible grammars, 2003. http://www.cs.cornell.edu/Projects/polyglot/ppg.html.

[6] A. Carmi. Adaptive multi-pass parsing. Master's thesis, Israel Institute of Technology, 2010.

[7] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Not.*, 25:35–44, November 1990.

[8] S. Erdweg, T. Rendel, C. Kästner, and 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, pages 391–406, New York, NY, USA, 2011. ACM.

[9] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, New York, NY, USA, 2004. ACM.

[10] M. Fowler. *Domain-Specific Languages*. Addilson-Wesley, Boston, USA, 2010.

[11] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdfreference manual. *SIGPLAN Not.*, 24(11):43–75, Nov. 1989.

[12] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[13] R. R. Redziejowski. Mouse: from parsing expressions to a practical parser. In *Proceedings of the CS&P 2009 Workshop*, pages 514–525. Warsaw University, 2009.

[14] L. V. S. Reis, R. S. Bigonha, V. O. Di Iorio, and L. E. A. de Souza. Adaptable parsing expression grammars. In *Proceedings of the 16th Brazilian conference on Programming Languages*, SBLP'12, pages 72–86, Berlin, Heidelberg, 2012. Springer-Verlag.

[15] L. V. S. Reis, V. O. Di Iorio, R. S. Bigonha, M. A. S. Bigonha, and R. C. Ladeira. XAJ: An extensible aspect-oriented language. In *Proceedings of the III Latin American Workshop on Aspect-Oriented Software Development*, pages 57–62. Universidade Federal do Ceará, 2009.

[16] S. Ryu. Parsing fortress syntax. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 76–84, New York, NY, USA, 2009. ACM.

[17] J. N. Shutt. Recursive adaptable grammars. Master's thesis, Worchester Polytechnic Institute, 1998.

[18] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.

[19] D. A. Watt and O. L. Madsen. Extended attribute grammars. *Comput. J.*, 26(2):142–153, 1983.