An On-the-Fly Grammar Modification Mechanism for Composing and Defining Extensible Languages

Leonardo V.S. Reis[1,*]

Departamento de Computação e Sistemas, Universidade Federal de Ouro Preto, Brazil

Vladimir O. Di Iorio[*]

Departamento de Informática, Universidade Federal de Viçosa, Brazil

Roberto S. Bigonha[*]

Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Brazil

Abstract

Adaptable Parsing Expression Grammar (APEG) is a formal method for defining the syntax of programming languages. It provides an on-the-fly mechanism to perform modifications of the syntax of the language during parsing time. The primary goal of this dynamic mechanism is the formal specification and the automatic parser generation for extensible languages. In this paper, we show how APEG can be used for the definition of the extensible languages SugarJ and Fortress, clarifying many aspects of the syntax of these languages. We also show that the mechanism for on-the-fly modification of syntax rules can be useful for defining grammars in a modular way, implementing all types of language composition as defined by Erdweg et alii [15] in the context of specification of extensible languages.

Keywords: Parsing Expression Grammars, Extensible languages, Grammars, Language composition

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 [1, 2, 3]. Despite the various methods for implementing *DSLs*, extensible

[*]Corresponding authors

Email addresses: `leo@decsi.ufop.br` (Leonardo V.S. Reis), `vladimir@dpi.ufv.br` (Vladimir O. Di Iorio), `bigonha@dcc.ufmg.br` (Roberto S. Bigonha)

[1]Tel.: +55 31 3852 8709; fax: +55 31 3852 8702.

languages seem to have several advantages over 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 [1], by means of syntax units designated as *sugar* libraries, which specify a new syntax for a domain concept. Tobin-Hochstadt et alii also discuss the advantages of implementing *DSLs* by means of libraries [4].

Reis et alii observed that there is a lack of formalization when defining the syntax of extensible languages and proposed a new formal method to fill this gap, which is called *Adaptable Parsing Expression Grammars (APEG)* [5, 6]. The main feature of APEG is the ability for formally describing how the syntax of a language can be modified on the fly, while parsing a program. Although APEG was initially proposed as a formal method for defining the syntax of extensible languages and efficiently parsing them, its flexibility for dynamically changing the grammar during parsing time also accredits APEG to implement other important issues in language design.

In [7], we have shown how APEG can be used for the definition of extensible languages, using SugarJ [1] and Fortress [8, 9, 10] as examples. This paper is an extended version of that work, showing how the mechanisms that allow on-the-fly modifications on the syntax of the language can also be used for producing modular specifications and composing of languages.

The remaining of this paper starts giving a brief introduction on how APEG works, in Section 2. Section 3 discusses an APEG specification of the extensible languages SugarJ and Fortress. In Sections 4 and 5, we show how the mechanisms provided by APEG allow building modular specifications and language composition. Section 6 discusses the related work and, Section 7 presents the conclusions.

2. Adaptable Parsing Expression Grammar

Adaptable Parsing Expression Grammars or APEG [5] is an extension of PEG [11], so as to allow the set of grammar rules to be changed during parsing. APEG associates attributes with nonterminal symbols and achieves adaptability through a special inherited attribute called *language attribute*. The language attribute is the first attribute of every nonterminal. It represents the current grammar and contains the set of all its rules. For illustrating APEG, Figure 1 shows an example of an APEG grammar for parsing programs in a language initially containing only sum expressions and which is self-extended during parsing with a rule for minus expression.

The inherited attributes immediately follow the name of its associated nonterminal. The nonterminal `Sum` only has the language attribute, specified between the symbols [and]. The definition of this nonterminal consists of a number followed by a sequence of zero or more `Add_Num`. The nonterminal `Num` has two attributes: one inherited, which is its language attribute, and the other is a synthesized attribute, which is defined in the returns clause. In the definition of the nonterminal `Sum`, the grammar returned by `Num` is passed as the inherited attribute of the nonterminal `Add_Num`. This grammar may have a new

Figure 1 An example of an APEG grammar

```
1  Sum[Grammar g]:
2    Num<g,g1> (Add_Num<g1>)*
3  ;
4  Add_Num[Grammar g]:
5    '+' Num<g,g1>
6  ;
7  Num[Grammar g] returns[Grammar g1]:
8    [0-9]+ {g1 = g;} /
9    'extend' {g1 = adapt(g,'Add_Num[Grammar g]:\'-\' Num<g,g1>;');}
10 ;
```

rule for this nonterminal depending on the choice selected on the definition of `Num`. The list of attributes of a nonterminal occurring on the right hand side of a rule is enclosed by the symbols < and >. Each list begins with the inherited attributes followed by the synthesized ones. One example is the use of the nonterminal `Num` in the definition of the nonterminal `Sum`, in Figure 1.

The definition of the nonterminal `Num` presents two choices. The first one specifies a sequence of at least one digit 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 curly braces, { }. The second choice of `Num` is the keyword **extend**, 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 contains the new rules. APEG only permits modifying the grammar by creating new rules or adding new choices to the end of existing rules [6]. In this case, a new choice is added to the end of the rule that defines the nonterminal `Add_Num`.

Then, if we have the string "`extend+2-3-4+5`" as input, the parser will work as follows: 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 `g`. The second choice of the nonterminal `Num` will be used, consuming the prefix *extend* of the input and returns a new grammar `g1` 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 choice for minus expression, then the remaining of the input is correctly parsed.

This example illustrates 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, to assign a new grammar to 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 allows the specification of constraint expressions to check if an expression evaluates to the

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

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

true value. Also, APEG has bind expressions, which capture the expressions matched and bind them to variable names. For example, the rule term

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

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

A formal definition of the semantics of APEG is presented in [5].

3. Defining the Syntax of Extensible Languages Using APEG

As we stated in the introduction, extensible languages arise as a good method for implementing *DSLs*. However, analysing extensible languages which have the properties required for defining *DSLs* in a modular way, such as the languages SugarJ [1], Fortress [8, 9, 10] and XAJ [12], 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.

Due the flexibility of APEG to change the grammar definition on the fly, it is possible to formally define the syntax of extensible languages, including the extensibility mechanism, and automatically parse them. In order to show how extensible languages can be implemented using APEG, we specify the syntax of the extensible languages SugarJ and Fortress, described in Sections 3.1 and 3.2, respectively.

3.1. The Syntax of SugarJ

SugarJ [1] 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

Figure 3 Use of the pair syntax.

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

5, *type* → '(' *type* ',' *type*;')', and a rule for using pair expressions in line 6, *expr* → '(' *expr* ',' *expr*;')'. Note that the definition of a rule in SugarJ is in an order that is reverse to the one commonly used in context-free grammars.

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

We have defined the syntax of SugarJ in *APEG* and used an experimental version of an interpreter of the model to automatically perform parsing. As *APEG* is based on *PEG*, we adapted an implementation of the Java grammar for the Mouse project [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 is used later in an import statement to extend the grammar. Differently from the implementation of SugarJ, which defines the rules in SDF [14] 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 definition rule for this nonterminal has a new choice:

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

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

Figure 4 Syntax definition of sugar libraries.

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

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 definition in 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

Figure 5 Syntax definition of compilation units.

```
1 compilation_unit[Grammar g, Map m] returns[Map m1]:
2   package_decl<p>? (import_decl<g, m, g1> {g=g1;})*
3       (type_declaration<g, p,m,m1> {m=m1;})*
4 ;
5 import_decl[Grammar g, Map m] returns[Grammar g1]:
6   'import' n=qualified_id ';' {g1=adapt(g,m.get(n));}
7 ;
```

Figure 6 Composition of more than one sugar library.

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

declaration for parsing the program text in Figure 3, so that the grammar is modified with the new rules defining `Pair` syntax.

Composing sugar libraries

Sugar libraries are composed by importing more than one *sugar* library into the same file. As an example, Figure 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 declaration in line 2. The modified grammar, which has the syntactic rules of pairs and closures, is used for parsing the class `Partial`.

The implementation of SugarJ uses SDF [14] and it may be necessary to write disambiguation rules when composing various grammars. However, it is impossible to prevent all the possibilities of ambiguities and conflicts, consequently composing two or more *sugar* libraries is not always possible. APEG avoids ambiguities using ordered choice, so composition is, in principle, always possible using APEG. In fact, if there is some overlapping between the rules of two or more extensions, the first option on the ordered choice clause will prevail. As new choices are always inserted at the end of a rule definition, a user may change the priority altering the order of the import declarations. It seems a

Figure 7 Definition of the closure syntax.

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

simple task, but it is not always easy to understand the interactions between overlapping rules.

Paradox Syntax × Semantics

Erdweg et alii claim that is it not clear how to support "local" imports, which extend the language [1]. They give an example of extending the language with the statement s_1 `after` s_2 whose semantics is to swap the execution order of the statements s_1 and s_2. They argue that the code

(``12'', 34) `after import syntactic.Pair`

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

We claim that this is not a paradox. In fact, it is an error situation and the doubts arise only because of the lack of formalization of the language and a confusion between syntax and semantics. Given the definition of the syntax in APEG, which parses the program from left to right, it is possible to answer this question. Initially, the grammar has the rule *statement* → *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.

3.2. The Syntax of Fortress

The main goals of the design of the Fortress language were to emulate mathematical syntax and to be extensible [9]. These two goals impose additional difficulties to build a parser for the language. However, defining the extensibility system in a formalism like PEG [11], which supports unlimited lookahead would bring some advantages [9, 10].

Figure 8 Definition of a for loop in Fortress.

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

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. The parsing expression defines the syntax of the new construct in a PEG style and the action part specifies how to translate the syntax into the core language. The action part is everything after the symbol ⇒. It is possible to use aliases associated with terminal or nonterminal symbols, creating references for them, which can be used in the action part. Figure 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 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 parsing the action part (nonterminal `syntax`). Therefore, the action part may use the new syntax being defined.

The use of the and-operator, which allows an infinite lookahead, was very important to handle 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.

Figure 9 APEG formalization of Fortress language.

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

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 [9], the authors report that they need to resolve the set of extensions (for example, in grammar `C` it may use syntax defined in `C` or `B`, but not in `A`) to generate the table for parsing the action part and this is not an easy task.

Using the APEG model, defining the task described above is simple and clear. We adapt the grammar, adding the rules of the grammars specified in the `extends` part. For example, parsing the grammar `B`, we add only the rules of `A` and when parsing the grammar `C`, we add only the rules of `B`. Another difficulty reported in [9] 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).

Figure 10 Combining grammars.

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

Figure 11 APEG grammar for expressions

```
1 expr:
2   term (op term)*
3 ;
4 op:
5   '+' / '-'
6 ;
7 term:
8   factor (mul factor)*
9 ;
10 mul:
11   '*'
12 ;
13 factor:
14   '(' expr ')' / number
15 ;
16 number:
17   [0-9]+
18 ;
```

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.

4. Grammar Modularization

Grammars in APEG are first-class types in the sense that they can be used as inherited or synthesized attributes from which APEG fetches the parsing expression of the associated nonterminal during parsing. This feature enables to pass pieces of grammars as attributes and to use them to build other grammars.

For example, Figure 11 shows an APEG grammar for expressions, and Figure 12 shows an example of a language which uses the definition of the language of expressions. Observe that, in the definition of the nonterminal `stmt` in Figure 12, the nonterminal `expr` comes from the grammar `Exp`, which is an inherited attribute of nonterminal `stmt`. This is possible because of the APEG semantics of the use of a nonterminal on a parsing expression. The parsing expression

Figure 12 Example of a APEG grammar which uses the definition of another APEG grammar.

```
1  start[Grammar g, Grammar exp]:
2    'begin' stmt<g, exp>+ 'end'
3  ;
4  stmt[Grammar g, Grammar Exp]:
5      id ':=' expr<Exp> ';'
6   / 'read' '(' id (',' id)* ')'
7   / 'write' '(' expr<Exp> (',' expr<Exp>)* ')'
8  ;
9  id:
10   [a-zA-Z] [a-zA-Z0-9]*
11 ;
```

of the nonterminal is fetched from the language attribute being used. For example, when using `expr<Exp>` in Figure 12, the parsing expression associated with the nonterminal `expr` is defined by the grammar `Exp`, which is the language attribute in this case.

The APEG flexibility for changing grammars during parsing allows building grammars in a modular way. It is possible to define different pieces of grammars and use all of them together for building another language. So, we can think of an APEG grammar as a module, which defines a set of "syntactic functions". Thus, grammars can be passed on as inherited attributes and their "syntactic functions" can be used when these grammars are selected as the language attribute.

Another advantage of this semantics is that we can change the language just by using a different grammar definition. For example, the attribute `exp` of the nonterminal `start` could be associated with alternative grammars for expressions using postfix or prefix notation, creating different languages without modifying the text of Figure 12. This feature can be useful for describing the syntax of languages in which a symbol has different meanings in different contexts, such as the "if" expression in the AspectJ language.

5. Language Composition

Erdweg et alii proposed a new taxonomy for distinguishing different types of language composition, namely *language extension* and *restriction*, *self-extension*, *language unification* and *extension composition* [15]. Although APEG has been originally proposed as a formalism for defining the syntax of extensible languages [5, 6] (*self-extension* as defined by Erdweg et alii), its dynamic behaviour is able to specify these kinds of language composition, in the syntactic level. In this section, we discuss how each type of language composition can be defined using APEG.

5.1. Self-Extension

Erdweg et alii define that a language supports *self-extension* if the language can be extended by programs of the language itself without changing its implementation [15].

In Section 3, we showed how to define the entire syntax of two *self-extensible* languages, SugarJ and Fortress, using APEG. Our specifications define clearly what rules are available at a given moment during parsing. The SugarJ specification allows resolving the false paradox about the local import raised by Erdweg et alii [1]. 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. So, we have provided enough evidence that APEG is capable for specifying *self-extensible languages*. The original definition and implementation of the languages SugarJ and Fortress present a lack of formalization of the syntax, especially for the aspects related with the extensibility mechanism of the language. When comparing the APEG specifications with those definitions, it is even more clear that APEG is appropriate to specify *self-extensible* languages.

5.2. Language Extension and Restriction

Differently from *self-extensibility*, which is a property of the language, *language extensibility* is a property of the language-development system. Erdweg et alii [15] define that a system has this property if it allows extending a base language by reusing its definition without modifications.

APEG clearly has this property and it is used for defining the syntax of extensible languages, as in the case of SugarJ and Fortress. In fact, when the SugarJ grammar (the base language) is extended with a new DSL (for example, the *Pair* DSL of Figure 3), the *language extensibility* property provided by APEG is used for extending the language.

The initial formalization of APEG [5] does not restrict how the grammar can be extended, indicating that extensions will be performed by functions defined by the designer. Later, in a prototype interpreter that was developed [6], extensions on the base grammar were restricted by the addition of new rules or by the addition of new choices at the end of an existing rule. As APEG has ordered choices as in PEG, *language extension* could not be always possible because of the shadow problem. For example, suppose a grammar consisting of this APEG rule

$$rule: \ 'a';$$

If we extend this rule with the new choice 'ab', this second choice will never be used, because if the input starts with the symbol a, the first choice will always succeed and the second one will never be tested. Although APEG may present this shadow problem when extending a language, it can be avoided as we did when extending the SugarJ language with some DSLs [6].

Another limitation imposed by APEG when extending a grammar is that it is not possible to change the set of attributes (inherited or synthesized) of nonterminals. The attributes in APEG are syntactic, evaluated during parsing.

Figure 13 APEG grammar for a declaration language

```
1 declist: decl (',' decl)*;
2
3 decl: id '=' number;
4
5 id: [a-zA-Z] [a-zA-Z0-9]*;
6
7 number: [0-9]+;
```

When a nonterminal is used on a parsing expression, all the attributes must be specified. It is very similar to arguments in function calls. So, if APEG allowed adding a new attribute on a nonterminal, it would be necessary to change every use of this nonterminal on all parsing expressions for defining the value of the new attribute. To avoid redefining many rules, APEG does not allow changing the set of attributes.

For syntactic purposes, the restriction to extend the set of attributes is not a problem. However, if the attributes of APEG are also used for semantics purposes, extensions on the semantics by adding new attributes may be desired.

Erdweg et alii present another type of language composition, the *language restriction* [15]. The idea of *language restriction* is the opposite of *language extension*: it consists in the exclusion of features from a language. Erdweg et alii do not give special treatment to this type of language composition because they argue that *language restriction* can be implemented as an extension of the validation phase of the base language. APEG does not have any feature to restrict the base language, thus it does not have any support for *language restriction*.

5.3. Language Unification

A language-development system supports *language unification* when it is possible to reuse, unchanged, the implementation of two languages being unified only by the addition of glue code [15]. A possible solution for unifying languages using APEG is to use ideas similar to the ones presented in Section 4, for modularization of grammars. We may define a grammar which has the two other grammars being unified as inherited attributes and create a new one which uses or has the definition of these two grammars.

For example, Figure 13 shows a language for declaring variables. In Figure 14, we show how to combine the language of Figure 13 with the language of expressions of Figure 11, so as to build a new language which allows variables in expressions. In line 3 of Figure 14, a new grammar which has the rules of both grammars is created. This is done by adding all rules of the grammar `expr` to grammar `decl`. Note that, if there is any nonterminal in the second grammar which is already defined in the first grammar, the parsing expression of this nonterminal in the first grammar is extended with a new choice, consisting of the parsing expression of the second one. It is similar to add a new rule, as

Figure 14 A grammar which unify the declaration and expression languages.

```
1 unification[Grammar g, Grammar decl, Grammar expr]
2                                 returns[Grammar result]:
3   {unify = decl + expr;}
4   {glue = unify + 'factor: id;';}
5   {result = glue + 'exprdecl: declist expr;';}
6 ;
```

explained in Section 2, but, in this case, it may add a set of rules. In line 4, the resulted grammar is extended with a rule to allow variables in expressions and, in line 5, a new nonterminal definition, `exprdecl`, which defines a rule for allowing a list of declarations followed by an expression is added, completing the unification of the language of declaration with that of expressions. The grammar unified is stored in the synthesized attribute `result`, allowing other grammars to use it.

Creating a new grammar by extending a grammar with the rules of another one, such as in line 3 of Figure 14, resembles the idea of inheritance of object-oriented programming. Mernik shows that the notion of inheritance enables to implement all the types of language composition described by Erdweg et alii [16]. Thus, by passing grammars as inherited attributes and using the idea presented in the example above, which simulates inheritance, APEG can achieve *language unification*. However, APEG does not allow overriding a nonterminal definition as in object-oriented programming, or simulate it, thus it is not possible to use inheritance as discussed in [16] when it is needed to override a nonterminal definition. Also, unifying languages by creating a new grammar dynamically, as in Figure 14, is not efficient when the set of nonterminals and rules are static. However, in the context of defining the syntax of extensible languages, composing grammars in this way using APEG could be useful, because the syntax would change dynamically.

5.4. Extension Composition

The kind of language composition described above only defines how a system can be extended with a single extension. To refer to a system which allows composing more than one extension, Erdweg et alii [15] define a new term, *extension composition*. There are two interesting cases of *extension composition*: *incremental extension* and *extension unification*.

A system supports *incremental extension* if it is possible to extend a base language with a extension E_1 and also extends the result with another extension, E_2. In other words, the system allows *language extensibility* twice or more times in the base language. APEG supports *incremental extension* because it is possible to extend a grammar and, afterwards, to extend the result. In fact, this property was used to implement composition of sugar libraries of SugarJ, as shown in Figure 6. The only restriction to *incremental extension* of APEG is

Figure 15 An example of extension composition.

```
1 start[Grammar g, Grammar decl, Grammar, expr, Grammar unify]:
2   unification<unify, decl, expr, r>
3   {result = r + 'mul: \'/\';'}
4   exprdecl<result>
5 ;
```

the shadow problem, discussed in Section 5.2, that can occur between extensions too.

A system supports *extension unification* when it allows extending a language with the result of the unification of two other languages or extending the result of the unification. So it refers to the process of combining together the properties *language extension* and *language unification*. APEG also supports *extension unification*. As an example, Figure 15 shows an extension to the result of the unification shown in Figure 14. In line 2, the grammar of Figure 14 is used to unify the languages of declarations and expressions. In the sequel, line 3 extends the result with the rule `mul: '/'` to allow division expressions. Finally, line 4 uses the definition of the nonterminal `exprdecl`, fetching the parsing expression of this nonterminal from the new grammar, represented by the attribute `result`. This example shows a *language unification* and afterwards a *language extension*, giving the idea of how APEG supports *extension unification*. The same idea can be used to extend a language with the set of rules of a unification of other two languages.

The APEG ability to add new rules or rule choices and also to change the grammar during parsing provides a flexible mechanism to compose language. APEG is indeed a powerful mechanism to define extensible languages by means of features to compose and reuse definitions of DSLs.

6. Related Work

In this section, we discuss works related to ours and split them in four categories: *parsing of extensible languages*, *models for defining extensible languages*, *grammar modularization* and *language composition*. First, we discuss some implementations of extensible languages which allow a flexible mechanism to extend their own concrete syntax (Section 6.1) and some adaptable models to define them (Section 6.2). Afterwards, we show related work in grammar modularization (Section 6.3) and in the field of language composition (Section 6.4).

6.1. 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 [4]. These languages use the same notation for data and program, S-expressions, thus they allow the implementation of a flexible and powerful macro system. Racket implements macros by means of 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, and modern extensible languages avoid this approach.

The implementation of parsers for extensible languages which do not use the same notation 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 analyzes the program using the table generated.

For example, the SugarJ compiler [1] 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 [17] (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 [14] grammar (a syntax formalism whose parsing 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 replace 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 notation for data and program. To parse a program in the Fortress language, a two-phase approach is taken [9]: 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. Similarly, the parser of the XAJ language [12] collects the new syntactic constructions defined by *syntax classes* and generates a new parser using the PPG tool [18]. 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.

6.2. Models for Defining Extensible Languages

As extensible languages may change their own set of rules during parsing, the formalisms more appropriate to specify their syntax may be the ones which also allow modifying the own set of grammar rules. Christiansen [19] proposes a formalism with these features, called *Adaptable Grammars*, which is essentially an Extended Attribute Grammar [20] 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. APEG was inspired in Adaptable Grammars of Christiansen and the main difference between APEG and Adaptable Grammars are the models in which they are based [5].

Shutt [21] 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)* [21], 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*. As shown in Section 3.2, the and-predicate and the not-predicate operators allow APEG to model forward reference of Fortress.

Carmi [22] argues that existing adaptable formalisms do not handle well forward references, such as goto statements that precede label declarations, and extensible languages with features like macro syntax and its expansion. Thus, he proposes a new model, called *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, handling forward references, macro definitions and expansion accordingly.

6.3. Grammar Modularization

Mernik and Umer [23] present an approach to modularizing grammars by incorporating the idea of inheritance in Attribute Grammars (AG). Using inheritance, the definition of new grammars may reuse productions rules and attributes of other grammar definitions, as well extend or modify them. The tool LISA [24, 25] implements this notion of inheritance. An important difference between APEG and LISA is that LISA, as it is an AG system, models the syntax and semantics of the language. APEG is a formalism to define the syntax of languages and, although the attributes can be used for semantic purpose, APEG does not allow extending the set of attributes of a nonterminal as in LISA. Another difference between LISA (and also AG systems) and APEG is that the attributes of APEG are evaluated during parsing and not after it, traversing the AST. Also, APEG is L-attributed and does not allow definitions with circular dependency.

Rats! [26] is a parser generator tool based on PEG that allows modularizing grammars. The tool has a module system for organizing, modifying, and composing syntactic specifications. Every grammar module can use other module definitions and also modify them by adding, overriding or removing individual alternatives in a production. SDF [14] also can separate grammars in modules and allows building new grammar definitions importing and using the definition of other grammars. However, differently from LISA and Rats!, SDF cannot allow modifying or overriding the definitions of the grammar being used and only permit adding new production rules to them. The ANTLR [27] parser generator tool also has an import mechanism which resembles inheritance. It processes a list of imports of grammars in depth-first strategy, adopting the first definition of some rule that it encounters and ignoring subsequent instances. However, ANTLR does not have any mechanism for overriding or modifying rules of imported grammars. Johnstone et alii introduced the idea of *Modularized Grammar Specification*, which divides the grammar specification into modules [28]. The main difference to previous works is the treatment for module namespaces, allowing using or importing the same nonterminal name from different grammar modules.

The above tools work at source level and produce a global grammar from the modules, so they make it hard to produce separated pieces of compiled grammars (parsers) and use them when building a new parser. As APEG allows changing the grammar during parsing, it is possible to generate binary pieces of grammars and use them as libraries, but the prototype interpreter does have this feature yet. Several works have this goal and propose techniques for generating small parse tables for parts of the language, and combining them to form

the table for the language. As an example, Cervelle et alii [29] implements a system which supports to separate compilation of pieces of grammars and dynamic linkage of these pieces at runtime. Parse tables are generated using a bottom-up approach from incomplete grammars in which some nonterminals, those that come from other pieces of grammars, are treated as special terminals (branch points). During runtime, the parser switches between the parse tables when needed. The algorithm described by Bravenboer and Visser [30] for parse table composition supports separate compilation of grammars to parse table components, using modular definition of syntax. A prototype for this algorithm generates parse tables for scannerless Generalized LR (GLR) parsers [31], with input grammars defined in SDF [14]. Schwerdfeger and Van Wyk [32, 33] define conditions for composing parsing tables while guaranteeing deterministic parsing, allowing defining extension to a base language with the guarantee that problems will not occur when combining several extensions.

6.4. Language Composition

The increasing use of DSLs has brought new challenges for language development, requiring that languages and development systems can be planned to be composed and evolved. Many researches and systems have been developed to allow easy implementation and composition of languages, specially to DSLs.

Erdweg et alii [15] noticed that there is a lack of precise terminology and ambiguity about the many meanings of language composition, therefore they proposed a new terminology and classification to language composition, which we used to analyse APEG. Also, composing languages involves to compose syntax and semantics. We discussed how APEG can achieve language composition only in the syntactic level, however there is much work which goes beyond syntax.

Attribute Grammars are a model that allows defining the syntax and semantics of languages, and the mechanism of inheritance applied to AG, which is implemented in LISA, allows LISA to compose languages in both syntactic and semantic levels [16]. JastAdd [34] takes a similar approach to LISA and also allows all type of language composition in both levels, syntax and semantics. JastAdd works on an object-oriented representation of an AST, in which nonterminals act as abstract superclasses and their productions act as specialized concrete subclasses. The subclasses specify the syntactic structure, semantics rules and attributes, which can be specialized or overrided using inheritance. By means of aspect-oriented concepts, JastAdd allows combining language specifications.

Spoofax [35] is an approach based on SDF and Stratego. Using SDF, Spoofax is able to implement all types of language composition on the syntax level. By means of the Stratego tool, Spoofax supports language composition on the semantics level, however Stratego only supports the addition of new semantic rules to extend the base language semantics and does not support the adaptation of an existing rule, so Spoofax only supports extension unification on the semantic level.

There are many systems that allow only composition by language extension. The main purpose of these systems is to provide a way to extend a base language with DSLs (incremental extension, to use the taxonomy of Erdweg et alii [15]). Language boxes [36] and island grammar based approaches [37] are examples of approaches that allow only incremental extension. Language boxes specify extensions by defining the syntax, including modification on the base grammar to integrate with the new syntax, the transformation of the DSL AST to the AST of the base grammar and some integration with IDEs, such as highlighting code. The idea of language boxes is very similar to sugar libraries of SugarJ [1] and to syntax classes of XAJ [12], which define how to extend the syntax of the language (the difference is that SugarJ and XAJ are extensible languages) and the transformation of the extension to the base language code, encapsulated in a single definition.

The implementation of language boxes is based on PEG and parser combinators. However, language boxes also allow more four forms to modify a nonterminal definition, besides insertions of new choices at the end of the nonterminal parsing expression allowed by APEG. It also allows adding a new choice at the beginning, adding a sequence parsing expression at the beginning or the end of the pre-defined parsing expression or overriding the nonterminal definition.

Using island grammars, Dinkelaker et alii propose an approach to extend a host language with DSLs [37]. Due to the use of island grammars, they avoid to specify the complete grammar of the DSL and the host-language. It is necessary only to specify the parts of the grammars (DSL and host language) that are relevant to the DSL concrete syntax implementation. The Dinkelaker et alii's approach also allows the notion of grammar inheritance, allowing defining syntax and semantics based on other definitions. Their approach composes languages in the syntax level using a version of the Earley parser algorithm [38] which supports composable island grammars [39]. The semantics of DSLs is given by translation to code in the host-language.

7. Conclusion

The primary goal of designing APEG is to provide a formal method to define the syntax of extensible languages and also automatically generate efficient parser for such languages. In this paper, we proved that APEG may have achieved these goals by defining the syntax of SugarJ and Fortress. The specification of these languages shows that APEG is a powerful formalism, which permit a clear definition of what rules are available at a given moment during parsing. It allows us to resolve the false paradox about local imports raised by Erdweg et alii [1]. Also, the semantics of the combination of Fortress grammars is clear in the APEG specification, explicitly showing what set of rules is to be used when a grammar extends another.

Forward reference is reported as difficult to be handled with adaptable models [22]. The definition of grammars in Fortress has a kind of forward reference, in which the action part may use syntax that is defined later. Therefore, it is

necessary to use a multi-pass approach. We showed that the predicate operator & of APEG allows simulating a multi-pass parser, handling the forward reference.

In this work, we do not address the efficiency of parsers generated using APEG. However, an experiment on parsing programs with SugarJ using a prototype version of an APEG interpreter indicates that APEG may significantly improve the performance of parsing such programs, when compared to the original implementation built with SDF [6].

We also analyze the flexible mechanism of APEG to change the grammar on the fly with respect to its power to define grammars in a modular way and to compose languages. We showed that the flexibility to modify the grammar, by means of the language attribute, during parsing allows APEG to reuse definitions from other grammars. This mechanism makes possible to generate parsers from APEG grammars and distribute them as libraries. At this moment, we only have a prototype interpreter for APEG and we are working on a parser generator, therefore we still do not have the appropriate tool to make libraries from APEG grammars.

Erdweg et alii claim that implementing DSLs by means of libraries in an extensible language, such as SugarJ, is a better choice than other approaches [1]. Defining DSLs libraries may require the use of the definition of other DSLs and composing them, therefore it is important that a formalism for specifying extensible languages supports composition of DSLs. We showed that APEG allows implementing almost all types of language composition presented by Erdweg et alii [15], showing that the mechanism for changing grammars on-the-fly is very flexible.

Implementing and composing languages require more than only syntax. It involves semantics and also language-based tools, such as editors and debuggers. We have used APEG's attributes for syntactic purposes, but it may be used for giving semantics. Therefore, a first question to investigate is whether APEG is appropriate for giving semantics or we should use other formalism after building the AST, such as metaprogramming or rewrite rules. APEG does not allow modifications on the set of attributes and also the definition of them are embedded into the parsing expression, then we must investigate whether it is a severe restriction to compose semantics. As inheritance is a good solution to compose grammars incrementally and modularly [23, 16], we are planning to study how inheritance could be incorporated to APEG and how it would fit its dynamic mechanism.

References

[1] 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.

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

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

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

[5] L. V. Reis, R. S. Bigonha, V. O. D. Iorio, L. E. S. Amorim, Adaptable parsing expression grammars, in: F. H. Carvalho Junior, L. S. Barbosa (Eds.), Programming Languages, Vol. 7554 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 72–86.

[6] L. V. Reis, R. S. Bigonha, V. O. D. Iorio, L. E. S. Amorim, The formalization and implementation of adaptable parsing expression grammars, Science of Computer Programming (to appear) (2014) –.

[7] L. V. S. Reis, V. O. D. Iorio, R. S. Bigonha, Defining the syntax of extensible languages, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC'14, 2014, pp. 1570–1576.

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

[9] E. Allen, R. Culpepper, J. D. Nielsen, J. Rafkind, S. Ryu, Growing a syntax, in: International Workshop on Foundations of Object-Oriented Languages, 2009.

[10] 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.

[11] 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, ACM, New York, NY, USA, 2004, pp. 111–122.

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

[13] 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.

[14] J. Heering, P. R. H. Hendriks, P. Klint, J. Rekers, The syntax definition formalism sdf reference manual, SIGPLAN Not. 24 (11) (1989) 43–75.

[15] S. Erdweg, P. G. Giarrusso, T. Rendel, Language composition untangled, in: Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications, LDTA'12, ACM, New York, NY, USA, 2012, pp. 7:1–7:8.

[16] M. Mernik, An object-oriented approach to language compositions for software language engineering, Journal of Systems and Software 86 (9) (2013) 2451 – 2464.

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

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

[19] H. Christiansen, A survey of adaptable grammars, SIGPLAN Not. 25 (1990) 35–44.

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

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

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

[23] M. Mernik, V. Umer, Incremental programming language development, Comput. Lang. Syst. Struct. 31 (1) (2005) 1–16.

[24] M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, Multiple attribute grammar inheritance, Informatica (Ljubljana) 24 (3) (2000) 319–328.

[25] M. Mernik, M. Lenic, E. Avdicausevic, V. Zumer, Lisa: An interactive environment for programming language development, in: Proceedings of the 11th International Conference on Compiler Construction, CC'02, Springer-Verlag, London, UK, UK, 2002, pp. 1–4.

[26] R. Grimm, Better extensibility through modular syntax, in: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, PLDI'06, ACM, New York, NY, USA, 2006, pp. 38–51.

[27] T. Parr, K. Fisher, LL(*): the foundation of the ANTLR parser generator, in: Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation, PLDI'11, ACM, New York, NY, USA, 2011, pp. 425–436.

[28] A. Johnstone, E. Scott, M. van den Brand, Modular grammar specification, Science of Computer Programming 87 (0) (2014) 23 – 43.

[29] J. Cervelle, R. Forax, G. Roussel, A simple implementation of grammar libraries, Computer Science and Information Systems 4 (2) (2007) 65–77.

[30] M. Bravenboer, E. Visser, Parse Table Composition – Separate Compilation and Binary Extensibility of Grammars, in: D. Gasevic, E. van Wyk (Eds.), Software Language Engineering (SLE 2008), Vol. 5452 of Lecture Notes in Computer Science, Springer, Heidelberg, 2009, pp. 74–94.

[31] M. Tomita, Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems, Kluwer Academic Publishers, Norwell, MA, USA, 1985.

[32] 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.

[33] 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.

[34] G. Hedin, E. Magnusson, Jastadd: an aspect-oriented compiler construction system, Science of Computer Programming 47 (1) (2003) 37 – 58, special Issue on Language Descriptions, Tools and Applications (LDTA'01).

[35] L. C. Kats, E. Visser, The spoofax language workbench: Rules for declarative specification of languages and ides, in: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA'10, ACM, New York, NY, USA, 2010, pp. 444–463.

[36] L. Renggli, M. Denker, O. Nierstrasz, Language boxes: Bending the host language with modular language changes, in: Proceedings of the Second International Conference on Software Language Engineering, SLE'09, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 274–293.

[37] T. Dinkelaker, M. Eichberg, M. Mezini, Incremental concrete syntax for embedded languages with support for separate compilation, Science of Computer Programming 78 (6) (2013) 615 – 632.

[38] J. Earley, An efficient context-free parsing algorithm, Commun. ACM 13 (2) (1970) 94–102.

[39] L. Moonen, Generating robust parsers using island grammars, in: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), WCRE'01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 13–22.