Computer Languages, Systems & Structures 42 (2015) 46–59

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

An on-the-fly grammar modification mechanism for composing and defining extensible languages

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

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

A R T I C L E I N F O

Article history:
Received 14 July 2014
Received in revised form
6 November 2014
Accepted 15 January 2015
Available online 24 January 2015

Keywords:
Parsing Expression Grammars
Extensible languages
Grammars
Language composition

A B S T R A C T

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 almost all types of language composition in the context of specification of extensible languages.

1. Introduction

The use of *Domain-Specific Languages (DSLs)* [1,2] has been considered a good way to improve readability of software, bridging the gap between domain concepts and their implementation, while improving productivity [3–5] and maintainability [3,6,7]. Despite the various methods for implementing *DSLs*, extensible languages seem to have several advantages over other approaches [8–10]. One of the advantages is the possibility of implementing *DSLs* in a modular way. For example, Erdweg et al. 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. Tobin-Hochstadt et al. also discuss the advantages of implementing *DSLs* by means of libraries [9].

The implementation of extensible languages requires adapting the parser every time the language is extended with a new construction. This task has been implemented in an ad-hoc way by regenerating a static grammar which accomplishes the new changes, compiling this grammar and using it for parsing the program [8,11,12]. Another way is to use models that provide mechanisms for dynamically changing grammar rules. The latter approach has several advantages [13].

Reis et al. observed this 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)* [14]. 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

[*] Corresponding author. Tel.: +55 31 3852 8709; fax: +55 31 3852 8702.
E-mail addresses: leo@decsi.ufop.br (L.V.S. Reis), vladimir@dpi.ufv.br (V.O. Di Iorio), bigonha@dcc.ufmg.br (R.S. Bigonha).

http://dx.doi.org/10.1016/j.cl.2015.01.002
1477-8424/© 2015 Elsevier Ltd. All rights reserved.

parsing them, its flexibility for dynamically changing the grammar during parsing time also accredits APEG to implement other important issues in language design.

The main application of on-the-fly grammar modification during parsing is on the definition of extensible languages. This work is an extension of [13], and shows that the mechanism for on-the-fly modification of syntax rules can implement almost all types of language composition defined in [8] and allows defining grammars in a modular way. It proves that the on-the-fly mechanism of APEG for changing grammars is powerful to define extensible languages. Composing grammars dynamically also may be useful for defining parameterized DSL libraries

Extending and composing languages require more than only syntax. It involves semantics and also language-based tools, such as editors and debuggers. In this paper, we only address syntactic aspects of extensibility. The semantic issues will be object of further research.

The remaining of this paper starts giving a brief introduction on how APEG works, in Section 2. Section 3 discusses APEG specifications 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, respectively. Section 6 discusses the related work and Section 7 presents the conclusions.

2. Adaptable Parsing Expression Grammar

Adaptable Parsing Expression Grammars or APEG [14] is an extension of PEG [15], 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 APEG grammar and contains the set of all its rules. This attribute can be changed, using update expressions, by a special function *adapt*. During the recognition process, when the APEG parser initiates the expansion of a nonterminal, its definition is obtained from its current language attribute.

Fig. 1 shows an example of an APEG grammar for parsing programs in a language initially containing only sum expressions and also a construction to extend itself with other rules. The nonterminals `Start`, `Sum`, `Add_Num` and `Num` only have the language attribute, enclosed by the symbols [and]. The nonterminal `rule` is the only one that has a synthesized attribute, which is defined by the `returns` clause. 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 `rule` in the definition of the nonterminal `Start`, in Fig. 1.

The language defined by the nonterminal `Start` is a sequence of one or more `Sum` `';'` or `'extend' rule <g,r >` ';' `{g=adapt(g,r);}`. The nonterminal `Sum` defines an arithmetic expression using only the addition operation. The parsing expression `'extend' rule <g,r >` ';' `{g=adapt(g,r);}` is a construction used in this example to extend the language. This parsing expression specifies a syntax that begins with the keyword `extend` followed by a nonterminal `rule` and ends with the semicolon symbol. The parsing expression `{g=adapt(g,r);}` is an update expression of APEG, and it is defined within the symbols {and}. That update expression extends the grammar `g` with new rules, encoded in the string `r`. The function `adapt` receives a grammar and a string representing the rules to be added to it and returns a new grammar, which contains the new rules [16]. APEG only permits modifying the grammar by creating new rules or adding new choices to the end of existing rules [16].

The definition of the nonterminals `Sum`, `Add_Numm` and `Num` is straightforward. The nonterminal `rule` just defines a string which encodes a rule. This string is a sequence of characters enclosed by "and". The parsing expression `s=(!".)*` in the definition of the nonterminal `rule` is a bind expression. It captures the text matched by the parsing expression `(!".)*` and binds it to the synthesized attribute `s`. The parsing expression `!".` uses the not-predicate operator, `!`, to assure that the next symbol is not ", without consuming it. Next, exactly one symbol is consumed using the any-expression symbol (a dot), which recognizes any character. Summarizing, the parsing expression `s=(!".)*` matches a sequence of symbols until it finds ", associating this sequence to the variable `s`.

```
Start[Grammar g]:
   (Sum<g> ';' / 'extend' rule<g,r> ';' {g = adapt(g,r);})+
;
Sum[Grammar g]:
   Num<g> (Add_Num<g>)*
;
Add_Num[Grammar g]:
   '+' Num<g>
;
Num[Grammar g]:
   [0-9]+
;
rule[Grammar g] returns[String s]:
   '“' s=(!” .)* '”'
;
```

Fig. 1. An example of an APEG grammar.

If we have the program of Fig. 2 as input, the parser will work as follows: it begins parsing with the nonterminal `Start` with the initial grammar, containing only the rules of Fig. 1. After this, it successfully matches the first expression `1+2` using the first rule choice, `Sum ';'`, and does not change the grammar. Next, the construction adding new rules is parsed using the second choice of the repetition. This time, the grammar `g` is modified with the new rule `Add_Num[Grammar g]: '-' Num <g >;`, by the update expression. Therefore, the new grammar has a new choice for the nonterminal `Add_Num`, allowing the minus expression. In the next iteration of the repetition, when parsing the last expression, the language attribute `g` passed to the nonterminal `Sum` has this new rule, so it is possible to correctly parse the expression in line 3.

It is important to highlight that although in this example we use the keyword `extend` in the construction that extends the language, it is not a requirement of the APEG model. A language designer can create a construct that extends the language using different keywords, or even no keyword at all.

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 the language attribute. However, the adaptability effectively takes place only when this attribute is passed to the nonterminal `Sum` as its language attribute in the next iteration of the repetition. APEG also has constraint expressions, which may indicate that a parsing expression fails if the given condition is not satisfied. 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 [14].

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, analyzing extensible languages which have the properties required for defining *DSLs* in a modular way, such as the languages SugarJ [8], Fortress [17,18,11] 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 [8] is a language recently developed by Erdweg et al. 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. Fig. 3 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*→ '('*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, Fig. 4 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 instance variable `p` of the class `Test` in line 5.

```
1 1+2;
2 extend "Add_Num[Grammar g]: '-' Num<g>;";
3 2-3-4+5;
```

Fig. 2. A program of the language defined by Fig. 1.

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

Fig. 3. A definition of sugar library for Pairs in SugarJ (borrowed from [8]).

```
import syntactic.Pair;

public class Test {

 private (String, Integer) p = ("12", 34);

}
```

Fig. 4. Use of the pair syntax (borrowed from [8]).

```
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 + ')'}
;
```

Fig. 5. Syntax definition of sugar libraries.

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 [19], which is also based on PEG, and extended it to allow the definition of *sugar* libraries. Fig. 5 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 Fig. 5 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 [20] 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. The nonterminal *peg_rule* defines a sequence of rules in the PEG style (lines 5 and 6 of Fig. 3 are a concrete example of the kind of syntax defined by this rule), but they are in an order that is reverse to the one commonly used in PEGs. The update expression of the *peg_rule*, {`rule +=id + ':' + s + ';'`;}, creates a string representing the rules being defined, in the APEG style. The other nonterminals define the syntax of parsing expressions and return their respective strings.

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]:

$\ldots/\text{sugar\_decl} < s, r > \{m1 = \text{add}(m, \text{pack}, s, r); \}$

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. Fig. 6 shows a new syntax definition for a compilation unit, highlighting the possible changes on the grammar rules. The nonterminal `compilation_unit` receives a map of *sugar* libraries and passes it to the nonterminal `import_decl`. The nonterminal

```
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));}
;
```

Fig. 6. Syntax definition of compilation units.

```
 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 }
```

Fig. 7. Composition of more than one sugar library (borrowed from [8]).

```
package javaclosure;

public sugar Closure {
  context-free syntax
    '#' type '(' type ')' -> type;
    '#' type formal_param block -> expr;
}
```

Fig. 8. Definition of the closure syntax (borrowed from [8]).

`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 Figs. 3 and 4, the compiler parses the definition in Fig. 3 with the nonterminal `compilation_unit`, which receives the initial grammar of the SugarJ language and an empty map without any definition of *sugar* libraries. As a result, the nonterminal `compilation_unit` returns a new map that has an entry for the new *sugar* library `Pair`. This new map is used in the import declaration for parsing the program text in Fig. 4, 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, Fig. 7 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 Fig. 8 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 [20] 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 simple task, but it is not always easy to understand the interactions between overlapping rules.

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 [18]. These two goals impose additional difficulties to build a parser for the language. However, defining the extensibility system in a formalism like PEG [15], which supports unlimited lookahead, would bring some advantages [18,11].

Fig. 9 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. Fig. 9 shows an example in which the nonterminal `forStart` is referenced by *b* in line 2.

Fig. 10 shows part of an APEG syntax definition of the Fortress language. Similar 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, different 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.

Combining grammars: Fig. 11 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 [18], 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 [18] is how to compose the rules with multiple extensions, as defined in grammar `D`. In APEG, to have the same behavior of the original Fortress implementation,

```
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
```

Fig. 9. Definition of a for loop in Fortress.

```
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 ']>'
;
```

Fig. 10. APEG formalization of Fortress language.

```
grammar A
   Nt ::= macroA => ...
end

grammar B extends A
   Nt |:= macroB => <[... macroA ...]>
end

grammar C extends B
   Nt |:= macroC => <[... macroB ...]>
end

grammar D extends {B,C}
   Nt |:= macroD => <[... macroB macroC ...]>
end
```

Fig. 11. Combining grammars (borrowed from [18]).

```
expr:
  term (op term)*
;
op:
  '+' / '-'
;
term:
  factor (mul factor)*
;
mul:
  '*'
;
factor:
  '(' expr ')' / number
;
number:
  [0-9]+
;
```

Fig. 12. APEG grammar for expressions (the language attribute was omitted).

```
start[Grammar g, Grammar exp]:
  'begin' stmt<g, exp>+ 'end'
;
stmt[Grammar g, Grammar g_exp]:
    id ':=' expr<g_exp> ';'
  / 'read' '(' id (',' id)* ')'
  / 'write' '(' expr<g_exp> (',' expr<g_exp>)* ')'
;
id:
  [a-zA-Z] [a-zA-Z0-9]*
;
```

Fig. 13. Example of a APEG grammar which uses the definition of another APEG grammar.

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 in the sequel, rules of `C`, for the example of Fig. 11).

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 us to pass pieces of grammars as attributes and to use them to build other grammars.

For example, Fig. 12 shows an APEG grammar for expressions, and Fig. 13 shows an example of a language which uses the definition of the language of expressions. In Fig. 14, we show a concrete example of an input string which conforms with grammar of Fig. 13. Observe that, in the definition of the nonterminal `stmt` in Fig. 13, the nonterminal `expr` comes from the grammar `g_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 of the nonterminal is fetched from the language attribute being used. For example, when using `expr` $<$`g_exp` $>$ in Fig. 13, the parsing expression associated with the nonterminal `expr` is defined by the grammar `g_exp`, which is the language attribute in this case.

```
begin
  x := 2+3*4;
  write(x);
end
```

Fig. 14. A concrete example which conforms with the grammar of Fig. 13.

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 APEG 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 Fig. 13. This feature can be useful for describing the syntax of languages by means of a parametrization mechanism in which a symbol has different meanings in different contexts, such as the "if" expression in the AspectJ language.

5. Language composition

Erdweg et al. proposed a new taxonomy for distinguishing different types of language composition, namely *language extension* and *restriction*, *self-extension*, *language unification* and *extension composition* [21]. Although APEG has been originally proposed as a formalism for defining the syntax of extensible languages [14,16] (*self-extension* as defined by Erdweg et al.), its dynamic behavior 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 al. define that a language supports *self-extension* if the language can be extended by programs of the language itself without changing its implementation [21].

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 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

Different from *self-extensibility*, which is a property of the language, *language extensibility* is a property of language definitions. Erdweg et al. define that a system has this property if it allows extending a base language by reusing its definition without modifications [21].

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 Fig. 4), the *language extensibility* property provided by APEG is used for extending the language.

The initial formalization of APEG [14] 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 [16], 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. For example, suppose a grammar consisting of the APEG rule

$$rule: \alpha.$$

If we extend this rule with the new choice $\alpha\ \beta$, this second choice will never be used, because if the input succeeds for the parsing expression α, the former choice will always be used. Otherwise, both choices fail. Although APEG may present this problem when extending a language, it can be avoided as we did when extending the SugarJ language with some DSLs [16].

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. When a nonterminal is used on a parsing expression, all its attributes must be specified. They are 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.

Erdweg et al. present another type of language composition, the *language restriction* [21]. The idea of *language restriction* is the opposite of *language extension*: it consists in the exclusion of features from a language. Erdweg et al. 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 provide any support for *language restriction*. In [22] it is shown how to give 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 [21]. 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, Fig. 15 shows a language for declaring variables. In Fig. 16, we show how to combine the language of Fig. 15 with the language of expressions of Fig. 12, so as to build a new language which allows variables in expressions. In line 3 of Fig. 16, 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 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 Fig. 16, resembles the idea of inheritance of object-oriented programming. Mernik shows that the notion of inheritance enables us to implement all the types of language composition described by Erdweg et al. [22]. Thus, by passing grammars as inherited attributes and using the idea presented in Fig. 16, 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 [22] when it is needed to override a nonterminal definition. Also, unifying languages by creating a new grammar dynamically, as in Fig. 16, 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 al. [21] 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 Fig. 7. The only restriction to *incremental extension* of APEG is the problem discussed in Section 5.2 that can occur between extensions too.

```
declist: decl (',' decl)*;

decl: id '=' number;

id: [a-zA-Z] [a-zA-Z0-9]*;

number: [0-9]+;
```

Fig. 15. APEG grammar for a declaration language.

```
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 ;
```

Fig. 16. A grammar which unifies the declaration and expression languages.

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, Fig. 17 shows an extension to the result of the unification shown in Fig. 16. In line 2, the grammar of Fig. 16 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 operation in 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`.

Fig. 18 shows a concrete example which conforms with the syntax defined by the grammar of Fig. 17. Note that the language defined (nonterminal `exprdecl`) is a list of variable declarations followed by an expression, as defined by the unification (grammar of Fig. 16). However, as the result of the unification was extended to allow the division operation, expressions can use this operation. Therefore, line 1 has a list of variable declarations, which comes from the nonterminal `declist` that was originally defined by the grammar of Fig. 15, and line 2 shows the extended version of expressions, which allows the division operation. This example shows a *language unification* and afterwards a *language extension*, giving a idea of how APEG supports *extension unification*. The same idea can be used to extend a language with the set of rules that results from the 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 languages. APEG is indeed a powerful mechanism to define extensible languages by means of features for composing and reusing definitions of DSLs.

6. Related work

In this section, we discuss works related to ours and split them into 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 [9]. 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 [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 [23] (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 [20] grammar (a syntax formalism whose parsing algorithm allows ambiguous grammars) and Stratego rules. SDF grammar and Stratego

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

Fig. 17. An example of extension composition.

```
1 x = 2, y = 3, z = 5
2 (7*x + 2*y) / z
```

Fig. 18. A concrete example which conforms with the grammar defined by Fig. 17.

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 class declarations 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 [18]: 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 [24]. 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, increases the complexity of writing the parser, and raises difficulties 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 most appropriate formalisms to specify their syntaxes may be the ones which also allow modifying the own set of grammar rules. Christiansen [25] proposes a formalism with these features, called *Adaptable Grammars*, which is essentially an Extended Attribute Grammar [26] 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 block structure declarations of several programming languages. APEG was inspired in Adaptable Grammars of Christiansen and the main difference between APEG and Adaptable Grammars is the models on which they are based [14].

Shutt [27] 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*) [27], 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 [28] argues that existing adaptable formalisms do not handle forward references well, such as goto statements that precede label declarations, and extensible languages with features like macro-syntax and its expansion. Thus, he proposes a new model, called *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 reparsing 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 Žumer [29] present an approach to write modular 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 [30,31] 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! [32] is a parser generator tool based on PEG that allows the production of modular 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 [20] also can separate grammars in modules and allows building new grammar definitions importing and using the definition of other grammars. However, different 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 [33] 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 al. introduced the idea of *Modularized Grammar Specification*, which divides the grammar specification into modules [34]. 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 not 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 al. [35] 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 [36] 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 [37], with input grammars defined in SDF [20]. Schwerdfeger and Van Wyk [38,39] 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 al. [21] 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 analyze 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 [22]. JastAdd [40] takes a similar approach to LISA and also allows all types 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 [41] 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 al. [21]). Language boxes [42] and island grammar based approaches [43] 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 [8] 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 than 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 al. propose an approach to extend a host language with DSLs [43]. 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 al.'s approach also uses the notion of grammar inheritance, which allows defining syntax and semantics based on other definitions. Their approach composes languages in the syntax level using a version of the Earley parser algorithm [44] which supports composable island grammars [45]. 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 argued 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 permits a clear definition of what rules are available at a given moment during parsing. Also, how to combine 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 [28]. The definition of grammars in Fortress has a kind of forward reference, in which the action part may use syntax that is defined later. Therefore, it is necessary to use a multi-pass approach. We showed that the predicate operator & of APEG allows simulating a multi-pass parser, handling forward reference properly.

In 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 [16].

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 al. claim that implementing DSLs by means of libraries in an extensible language, such as SugarJ, is a better choice than other approaches [8]. 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 al. [21], 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 [46–48]. 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 is 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 [22,29], we are planning to study how inheritance could be incorporated to APEG and how it would fit its dynamic mechanism.

Another important issue is error reporting. APEG reports errors based on the concrete syntax of the extension, however the prototype interpreter used does not have a mechanism for error recovery and abort on the first error found. Error reporting is also object of future work. Support for integrating APEG with IDEs and offering facilities to language-based tools, such as debugging and syntax highlighting, are also important future work.

References

[1] Fowler M. Domain-specific languages. Boston, USA: Addison-Wesley; 2010.

[2] Mernik M, Heering J, Sloane AM. When and how to develop domainspecific languages. ACM Comput Surv 2005;37(4):316–44.

[3] Kosar T, Mernik M, Carver JC. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. Empir Softw Eng 2012;17(3):276–304.

[4] Barišić A, Amaral V, Goulão M, Barroca B. Quality in use of domain-specific languages: a case study. In: Proceedings of the 3rd ACM SIGPLAN workshop on evaluation and usability of programming languages and tools, PLATEAU'11. New York, NY, USA:ACM; 2011. p. 65–72.

[5] Kieburtz RB, McKinney L, Bell JM, Hook J, Kotov A, Lewis J, et al. A software engineering experiment in software component generation. In: Proceedings of the 18th international conference on software engineering, ICSE'96. Washington, DC, USA: IEEE Computer Society; 1996. p. 542–52.

[6] Kosar T, Oliveira N, Marjan M, Pereira MJV, Črepinšek M, da Cruz D, et al. Comparing general-purpose and domain-specific languages: an empirical study. Comput Sci Inf Syst 2010;7:247–64.

[7] van Deursen A, Klint P. Little languages: little maintenance. J Softw Maint 1998;10(2):75–92.

[8] Erdweg S, Rendel T, Kästner C, Ostermann K. SugarJ: library-based syntactic language extensibility. In: Proceedings of the 2011 ACM international conference on object oriented programming systems languages and applications, OOPSLA'11. New York, NY, USA: ACM; 2011. p. 391–406.

[9] Tobin-Hochstadt S, St-Amour V, Culpepper R, Flatt M, Felleisen M. Languages as libraries. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation, PLDI'11. New York, NY, USA: ACM; 2011. p. 132–41.

[10] Kosar T, López PEM, Barrientos PA, Mernik M. A preliminary study on various implementation approaches of domain-specific language. Inf Softw Technol 2008;50(5):390–405.

[11] Ryu S. Parsing Fortress syntax. In: Proceedings of the 7th international conference on principles and practice of programming in Java. PPPJ'09. New York, NY, USA: ACM; 2009. p. 76–84.
[12] Reis LVS, Di Iorio VO, Bigonha RS, Bigonha MAS, Ladeira RC. XAJ: an extensible aspect-oriented language. In: Proceedings of the III Latin American workshop on aspect-oriented software development. Universidade Federal do Ceará, Brazil; 2009. p. 57–62.
[13] Reis LVS, Iorio VOD, Bigonha RS. Defining the syntax of extensible languages. In: Proceedings of the 29th annual ACM symposium on applied computing, SAC'14; 2014. p. 1570–6.
[14] Reis LV, Bigonha RS, Iorio VOD, Amorim LES. Adaptable parsing expression grammars. In: Carvalho Junior FH, Barbosa LS, editors. Programming languages. Lecture notes in computer science, vol. 7554. Berlin, Heidelberg: Springer; 2012. p. 72–86.
[15] Ford B. Parsing expression grammars: a recognition-based syntactic foundation. In: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'04. New York, NY, USA: ACM; 2004. p. 111–22.
[16] Reis LV, Bigonha RS, Iorio VOD, Amorim LES. The formalization and implementation of adaptable parsing expression grammars. Sci Comput Program 2014;96(Part 2):191–210 (selected and extended papers of the Brazilian Symposium on Programming Languages 2012 (SBLP 2012)), Elsevier, USA.
[17] Allen E, Chase D, Hallett J, Luchangco V, Maessen J-W, Ryu S, et al. The Fortress language specification version 1.0 (March 2008).
[18] Allen E, Culpepper R, Nielsen JD, Rafkind J, Ryu S. Growing a syntax. In: International workshop on foundations of object-oriented languages; 2009. p. 11.
[19] Redziejowski RR. Mouse: from parsing expressions to a practical parser. In: Proceedings of the CS&P 2009 workshop. Warsaw University; 2009. p. 514–25.
[20] Heering J, Hendriks PRH, Klint P, Rekers J. The syntax definition formalism sdf reference manual. SIGPLAN Not 1989;24(11):43–75.
[21] Erdweg S, Giarrusso PG, Rendel T. Language composition untangled. In: Proceedings of the twelfth workshop on language descriptions, tools, and applications, LDTA'12. New York, NY, USA: ACM; 2012. p. 7:1–8.
[22] Mernik M. An object-oriented approach to language compositions for software language engineering. J Syst Softw 2013;86(9):2451–64.
[23] Bravenboer M, Kalleberg KT, Vermaas R, Visser E. Stratego/xt 0.17. a language and toolset for program transformation. Sci Comput Program 2008;72(1–2):52–70.
[24] Brukman M, Myers AC. PPG: a parser generator for extensible grammars, ⟨http://www.cs.cornell.edu/Projects/polyglot/ppg.html⟩; 2003.
[25] Christiansen H. A survey of adaptable grammars. SIGPLAN Not 1990;25:35–44.
[26] Watt DA, Madsen OL. Extended attribute grammars. Comput J 1983;26(2):142–53.
[27] Shutt JN. Recursive adaptable grammars [Master's thesis]. Worchester Polytechnic Institute; 1998.
[28] Carmi A. Adaptive multi-pass parsing [Master's thesis]. Israel Institute of Technology; 2010.
[29] Mernik M, Žumer V. Incremental programming language development. Comput Lang Syst Struct 2005;31(1):1–16.
[30] Mernik M, Lenič M, Avdičaušević E, Žumer V. Multiple attribute grammar inheritance. Informatica 2000;24(3):319–28.
[31] Mernik M, Lenič M, Avdičaušević E, Žumer V. Lisa: an interactive environment for programming language development. In: Proceedings of the 11th international conference on compiler construction, CC'02. London, UK: Springer-Verlag; 2002. p. 1–4.
[32] Grimm R. Better extensibility through modular syntax. In: Proceedings of the 2006 ACM SIGPLAN conference on programming language design and implementation, PLDI'06. New York, NY, USA: ACM; 2006. p. 38–51.
[33] Parr T, Fisher K. LL(*): the foundation of the ANTLR parser generator. In: Proceedings of the 32nd ACM SIGPLAN conference on programming language design and implementation, PLDI'11. New York, NY, USA: ACM; 2011. p. 425–36.
[34] Johnstone A, Scott E, van den Brand M. Modular grammar specification. Sci Comput Program 2014;87:23–43.
[35] Cervelle J, Forax R, Roussel G. A simple implementation of grammar libraries. Comput Sci Inf Syst 2007;4(2):65–77.
[36] Bravenboer M, Visser E. Parse table composition – separate compilation and binary extensibility of Grammars. In: Gasevic D, vanWyk E, editors. Software language engineering (SLE 2008). Lecture notes in computer science, vol. 5452. Heidelberg: Springer; 2009. p. 74–94.
[37] Tomita M. In: Efficient parsing for natural languagea fast algorithm for practical systems. Norwell, MA, USA: Kluwer Academic Publishers; 1985.
[38] Schwerdfeger AC, Van Wyk ER. Verifiable composition of deterministic grammars. In: Proceedings of the 2009 ACM SIGPLAN conference on programming language design and implementation, PLDI'09. New York, NY, USA: ACM; 2009. p. 199–210.
[39] Schwerdfeger A, Van Wyk E. Verifiable parse table composition for deterministic parsing. In: Proceedings of the second international conference on software language engineering, SLE'09. Berlin, Heidelberg: Springer-Verlag; 2010. p. 184–203.
[40] Hedin G, Magnusson E. Jastadd: an aspect-oriented compiler construction system. Sci Comput Program 2003;47(1):37–58. (special Issue on Language Descriptions, Tools and Applications (LDTA'01)).
[41] Kats LC, Visser E. 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. New York, NY, USA: ACM; 2010. pp. 444–63.
[42] Renggli L, Denker M, Nierstrasz O. Language boxes: Bending the host language with modular language changes. In: Proceedings of the second international conference on software language engineering, SLE'09. Berlin, Heidelberg: Springer-Verlag; 2010. p. 274–93.
[43] Dinkelaker T, Eichberg M, Mezini M. Incremental concrete syntax for embedded languages with support for separate compilation. Sci Comput Program 2013;78(6):615–32.
[44] Earley J. An efficient context-free parsing algorithm. Commun ACM 1970;13(2):94–102.
[45] Moonen L. Generating robust parsers using island grammars. In: Proceedings of the eighth working conference on reverse engineering (WCRE'01), WCRE'01. Washington, DC, USA: IEEE Computer Society; 2001. p. 13–22.
[46] Chiş A, Gîrba T, Nierstrasz O. The moldable debugger: a framework for developing domain-specific debuggers. In: Combemale B, Pearce D, Barais O, Vinju J, editors. Software language engineering. Lecture notes in computer science, vol. 8706. Switzerland: Springer International Publishing; 2014. p. 102–21.
[47] Erdweg S, Kats LCL, Rendel T, Kästner C, Ostermann K, Visser E. Growing a language environment with editor libraries. In: Denney E, Schultz UP (Eds.), Proceedings of the 10th international conference on generative programming and component engineering, GPCE 2011, Portland, Oregon, USA, October 22–24, 2011. ACM, USA; 2011. p. 167–76.
[48] Wu H, Gray J, Mernik M. Grammar-driven generation of domain-specific language debuggers. Softw Pract Exp 2008;38(10):1073–103.