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

ARTICLE INFO

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

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 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/

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`.

```
1  Start[Grammar g]:
2    (Sum<g> ';' / 'extend' rule<g,r> ';' {g = adapt(g,r);})+
3  ;
4  Sum[Grammar g]:
5    Num<g> (Add_Num<g>)*
6  ;
7  Add_Num[Grammar g]:
8    '+' Num<g>
9  ;
10 Num[Grammar g]:
11   [0-9]+
12 ;
13 rule[Grammar g] returns[String s]:
14   '"' s=(!"  .)* '"'
15 ;
```

Fig. 1. An example of an APEG grammar.