A Mixed Approach for Building Extensible Parsers

Leonardo Vieira dos Santos Reis[1], Vladimir Oliveira Di Iorio[2], and Roberto S. Bigonha[3]

[1] Departamento de Computação e Sistemas, Universidade Federal de Ouro Preto, João Monlevade, Brazil
leo@decsi.ufop.br
[2] Departamento de Informática, Universidade Federal de Viçosa, Viçosa, Brazil
vladimir@dpi.ufv.br
[3] Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, Brazil
bigonha@dcc.ufmg.br

Abstract. For languages whose syntax is fixed, parsers are usually built with a static structure. The implementation of features like macro mechanisms or extensible languages requires the use of parsers that may be dynamically extended. In this work, we discuss a mixed approach for building efficient top-down dynamically extensible parsers. Our view is based on the fact that a large part of the parser code can be statically compiled and only the parts that are dynamic should be interpreted for a more efficient processing. We propose the generation of code for the base parser, in which hooks are included to allow efficient extension of the underlying grammar and activation of a grammar interpreter whenever it is necessary to use an extended syntax. As a proof of concept, we present a prototype implementation of a parser generator using Adaptable Parsing Expression Grammars (APEG) as the underlying method for syntax definition. We show that APEG has features which allow an efficient implementation using the proposed mixed approach.

1 Introduction

Parser generators have been used for more than 50 years. Tools like YACC [10] can automatically build a parser from a formal definition of the syntax of a language, usually based on context-free grammars (CFG). The main motivation for automatic parser generation is compiler correctness and recognition completeness, since with manual implementation it is very difficult to guarantee that all programs in a given language will be correctly analysed. The parsers that are generated from the language formal definition usually consist of a fixed code driven by a parse table. Different languages are associated with distinct parse tables. With top-down parsers, it is common to produce the code directly as a recursive descent program instead of using parse tables.

Extensible parsers [28] impose new challenges to automatic parser generation, since they have to cope with on-the-fly extensions of their own concrete syntax,

F. M. Quintão Pereira (Ed.): SBLP 2014, LNCS 8771, pp. 1–15, 2014.

requiring that the parser must be dynamically updated. If these changes are frequent, the use of interpretation techniques may be less time-consuming than to reconstruct the parser. Thus, at first, instead of building a fixed code for the parser, it may be interesting to use an interpreter for parsing the input based on a suitable representation of the syntax of the language, which may be dynamically updated. However, this solution is not entirely satisfactory. A mixed approach, compilation and interpretation, offers the best of both worlds.

For automatically building a parser, either using code generation or interpretation, it is necessary to use a formal model that is powerful enough to appropriately describe the syntax of the language, including possible on-the-fly modifications. Adaptable Parser Expression Grammar (APEG) [17,18] is a new formal model that satisfies these requirements. It is based on PEG (Parsing Expressions Grammars) [7], a formalism similar to CFG which formally describes a recursive descent parser. APEG extends PEG with the notion of *adaptability*, which is implemented by means of operations that allow the own syntax of the language to be extended. In previous work [19], we have shown that a prototype interpreter for APEG allows the implementation of extensible parsers which are more efficient than the traditional approaches used by other tools.

In this work, we discuss a mixed approach for building extensible parsers. Our view is based on the fact that a large part of the syntax of an extensible language is stable, so it is appropriate for a parser whose code can be statically generated. The parts of the syntax specification that are dynamically extended may use grammar interpretation for a more efficient processing. We propose the generation of code for the base syntax, including hooks that may allow an efficient extension, activating an interpreter whenever it is necessary to use the extended syntax. As a proof of concept, we present a prototype implementation of a parser generator for APEG. We show that APEG has features which allow an efficient implementation using the proposed mixed approach.

Section 2 presents a definition of the APEG model to help understanding the concepts used in the following sections. Section 3 discusses our approach in detail, showing how it works with the APEG model. In Section 4, we describe some efficiency tests with parsers generated using the mixed approach. Section 5 lists some works similar to ours. Section 6 presents our final conclusions and discusses future works.

2 Adaptable Parsing Expression Grammar

Parsing Expression Grammar (PEG) [7] is a model for describing the syntax of programming languages using a recognition-based approach instead of the generative system typical of context-free grammars (CFG). Similar to CFG, formally a PEG is a 4-tuple (V_N, V_T, R, S), where V_N is the set of nonterminal symbols, V_T is the set of terminal symbols and S is the initial symbol. R is a rule function which maps nonterminals to *parsing expressions*. A parsing expression is very similar to the right hand side of a CFG production rule in the extended Backus-Naur form, except for the addition of two new operators: the *not-predicate* and the *prioritized choice*. The not-predicate operator

checks whether the input string matches some syntax, without consuming it, thus allowing unrestricted lookahead. The prioritized choice lists alternative patterns to be tested in order.

Adaptable PEG (APEG) [18] is an adaptable model based on PEG. It also uses the idea of attributes of Attributes Grammars [27] to guide parsing. The attributes of APEG are L-Attributed and its purporse is syntactic and not semantics as in attributes grammars. APEG possesses a special attribute called *language attribute*, which represents the set of rules that are currently used. Language attribute values can be defined by means of embedded semantic actions and can be passed to different branches of the parse tree. This allows a formal representation of a set of syntactic rules that can be modified on-the-fly, i.e., during the parsing of an input string. APEG allows to extend the grammar by addition of new choices at the end of an existing list of choices of the definition rule of a given nonterminal or by addition of new nonterminal definitions.

As a concrete example, Figure 1 shows a PEG definition of a toy block structured language in which a block consists of a list of declarations of integer variables, followed by a list of assignment statements. An assignment statement consists of a variable on the left side and a variable on the right side. For simplicity, the whitespaces are not considered.

block	←	{ *dlist slist* }	*decl*	←	**int** *id* ;
dlist	←	*decl decl**	*stmt*	←	*id* = *id* ;
slist	←	*stmt stmt**	*id*	←	*alpha alpha**

Fig. 1. Syntax of block with declaration and use of variables (simplified)

Suppose that the context dependent constraints of this language are: a variable cannot be used if it has not been declared before, and a variable cannot be declared more than once. Using APEG, we implemented these context dependent constraints as shown in Figure 2. As mentioned before, every nonterminal has a special inherited attribute, the language attribute, which is a representation of a grammar. In Figure 2, this attribute is always the first one and it has type Grammar. The inherited attributes of a nonterminal are enclosed in the symbols "[" and "]" occurring after the name of the nonterminal, and the synthesized attributes are specified after the *returns* keywords. For example, line 20 defines the nonterminal *id* with one inherited attribute of type *Grammar* named as *g* (it is the language attribute) and one synthesized attribute named as *s* of type *String*. To refer to a nonterminal on a parsing expression, we use the name of the nonterminal followed by the list of inherited attributes and synthesized attributes, in this order, enclosed in the symbols "<" and ">". As an example, in line 21, *alpha<g, ch1>* refers to the nonterminal *alpha* followed by its attributes *g* and *ch1*. APEG also has *constraint*, *binding* and *update expressions*. The *constraint expression* is a boolean expression enclosed by the symbols "{?" and "}", such as the expression in the definition of the nonterminal *var* (line 12). A *constraint expression* succeeds if it evaluates to true, otherwise it fails. A *binding*

expression assigns the input string matched by a parsing expression to a variable. It is used on line 24 of Figure 2 to assign to variable *ch* the value of the character matched by the given parsing expression. An *update expression* is enclosed by the symbols "{" and "}" and it is used to assign the value of an expression to an attribute. The parsing expression $\{g = g1;\}$ in line 5 of Figure 2 is an example of an update expression.

In this example, the idea of implementing the context dependent constraints is to adapt the nonterminal *var* on the fly in order to allow only declared variables to be recognized. Note that, in the beginning, the nonterminal *var* does not recognize any symbols (lines 11-12). However, when a variable is declared (nonterminal *decl*, defined in lines 7-9), a new grammar rule is produced by the addition of a new choice in the definition of nonterminal *var*, which allows the recognition of the new variable name. The resulting new grammar is passed as the language attribute, in the definition of the nonterminal block, to the nonterminal *slist*, and, in the sequel, to *stmt*. As a result, the nonterminal *stmt* now can recognize the declared variable.

```
1  block[Grammar g]:
2    '{' dlist<g, g1> slist<g1> '}' !.;
3
4  dlist[Grammar g] returns[Grammar g1]:
5    decl<g, g1> {g = g1;} (decl<g, g1> {g = g1;})*;
6
7  decl[Grammar g] returns[Grammar g1]:
8    !('int␣' var) 'int␣' id<s> ';'
9     {g1 = g + 'var:␣\'' + s + '\' !alpha<g, ch>;';};
10
11 var[Grammar g]:
12   {? false };
13
14 slist[Grammar g]:
15   stmt<g> stmt<g>*;
16
17 stmt[Grammar g]:
18   var<g> '=' var<g> ';';
19
20 id[Grammar g] returns[String s]:
21   alpha<g, ch1> {s = ch1;} (alpha<g, ch2> {s = s + ch2;})*;
22
23 alpha[Grammar g] returns[String ch]:
24   ch=[a-zA-Z0-9_];
```

Fig. 2. Example with the set of production rules changed at parse time

For example, suppose the input string $\{int\ a;int\ b;a=b;b=a;\}$. The recognition of this string starts with the nonterminal *block* and its language attribute is the grammar in Figure 2. After recognizing the first symbol, {, the parser proceeds to recognize a list of declarations (nonterminal *dlist*), passing down the same grammar as the language attribute to the nonterminal *dlist*. During the recognition of the nonterminal *dlist*, it first tries to match a variable declaration through the nonterminal *decl*, passing to it the same language attribute. The nonterminal *decl* first checks if the variable is already declared using the parsing expression *!('int ' var)*. The not-lookahead operator, !, succeeds if the expression enclosed in parentheses fails, and it does not consume any symbol from the input. In order to check whether the variable "a" has already been declared, the parsing expression enclosed in parentheses matches the "int" string, but the nonterminal *var* does not recognize the variable "a", because it does not have any rule for it yet. In the sequel, the parsing expression *'int ' id⟨s⟩ ';'* recognizes the declaration of variable 'a'. Note the use of the nonterminal *id* instead of the nonterminal *var*. The nonterminal *id* is used here to recognize any valid variable name, since it is a new one. Then, a new grammar is built from the current grammar by the addition of a new choice, *var* : 'a' *!alpha⟨ch⟩;*, on the definition of nonterminal *var*. This new grammar becomes the value of the synthesized attribute *g1*.

The grammar synthesized by the nonterminal *decl* is used in the nonterminal *dlist* as language attribute of other calls of the nonterminal *decl*. Proceeding, the next variable declaration will be recognized, and the nonterminal *dlist* synthesizes a new grammar with these two choices, in this order, *var* : 'a' *!alpha⟨ch⟩;* and *var* : 'b' *!alpha⟨ch⟩;*, for the nonterminal *var*. This new grammar is used by the nonterminal *block* to pass it as the language attribute of the nonterminal *slist*. As a result, the two statements, $a = b$ and $b = a$, can be recognized, because the nonterminal *var* in the language attribute passed to the nonterminal *stmt* has rules to recognize the variables 'a' and 'b'.

Usually, parser generators for PEG produce a top-down recursive descent parser. Every nonterminal is implemented by a function whose body is a code for its parsing expression. The return value of each function is an integer value representing the position on the input that it has got moved on or the value -1 if it fails. It is straightforward to extend this idea to include attributes: the inherited attributes become parameters of functions and synthesized attributes are return values. For example, Figure 3 shows the code generated for the nonterminal *var* of Figure 2. The function *var* has one parameter, the language attribute, and returns an object of type *Result*, which must contain fields representing the portion of the input consumed and the values of the synthesized attributes, when specified.

Complications with this scheme arise when the base grammar is dynamically extended during parsing. When new choices are added to the *var* nonterminal, the function of Figure 3 does not represent anymore the correct code for this nonterminal, then this function must be updated. However, it is cumbersome regenerating all the parser code on the fly to reflect these small changes. In

```
1 Result var(Grammar g) {
2   if(false) {
3       // do nothing
4   } else
5     return new Result(-1); // a fail result
6 }
```

Fig. 3. Example of code generated by a PEG

cases where the grammar changes several times, as in extensible languages, the on-the-fly regeneration of all the parser is very expensive [17]. An alternative solution is to interpret the whole grammar directly. However, this may cause a great loss in parsing efficiency. So, we propose, in this paper, an approach to efficiently adapt the grammar. We propose to generate the code from the base grammar and include hooks to jump from the generated code to interprete the parts that have been added dynamically.

3 Mixing Code Generation and Interpretation

Since APEG only allows changes in the definition of nonterminal symbols by insertion of new choices at the end of the rules [19], we generate a recursive descent parser from an APEG grammar, so there is a function for each nonterminal and, whenever necessary, we place at the end of the body of these functions a call to the interpreter.

Figure 4 shows a scratch of the code generated for the grammar of Figure 2. As shown, we generate a Java class which has a function to each nonterminal definition on the grammar. The generated class extends the predefined class *Grammar* that has the implementation of standard functions, such as the function *interpretChoice* to interpret an AST and functions to add rules to the grammar or to clone the grammar itself.

The vector *adapt* (line 3 in Figure 4) stores a possible new choice for each nonterminal. Notice the hook at the end of the function body of each nonterminal (lines 29 and 40) to call the interpreter with its possible choice. This hook will be reached only if its preceding code fails, indicating that we must interpret the new choice. For example, if the code representing the parsing expression *'{' dlist<g, g1> slist<g1> '}'* on lines 6 to 24 fails, then we call the interpreter passing its new choice. So, the action of adapting a grammar is just an action of including a new choice rule on the vector *adapt*.

Using this strategy, all the base code for the grammar is generated and compiled, and only the choices that are added dynamically must be interpreted. Our strategy is based on the assumption that the code for the base grammar is expected to be large and used many times. Therefore, the expected result shall be a faster parser than the interpreter we have proposed in previous work [19], and will still allow an efficient method for managing syntactic extensions.


```
1  public class BlockLanguage extends Grammar {
2    // vector of new choices
3    private CommonTree[] adapt = new CommonTree[8];
4    ...
5    public Result block(BlockLanguage g) {
6      BlockLanguage g1; // local attribute
7      Result result;
8      int position = g.match("{", currentPos);
9      if(position > 0) {
10       g.currentPos = position;
11       result = g.dlist(g);
12       if(!result.isFail()) {
13         g1 = (BlockLanguage) result.getAttribute(0);
14         g1.currentPos = result.getNext_pos();
15         result = g1.slist(g1);
16         if(!result.isFail()) {
17           position = g.match("}", result.getNext_pos());
18           if(position > 0) {
19             char ch = g.read(position);
20             if(APEGInputStream.isEOF(ch))
21               return new Result(position);
22           }
23         }
24       }
25     }
26     Environment env;
27     ... // set the environment to start the interpreter
28
29     // interpreter the choice of block (index 0)
30     return g.interpretChoice(adapt[0], env);
31   }
32
33   public Result var(BlockLanguage g) {
34     if(false) {
35       // do nothing
36     }
37     Environment env;
38     // set the environment to start the interpreter
39
40     // interpreter the choice of var (index 3)
41     return g.interpretChoice(adapt[3], env);
42   }
43   ... // other functions
44 }
```

Fig. 4. Generated code for the block language

In the APEG model, a parsing expression of a nonterminal is fetched from its language attribute. Using different language attributes, it is possible to get different parsing expressions for the same nonterminal, thus effectively adapting the grammar. To have this behaviour, each function generated from a nonterminal has the language attribute as a parameter. The type of this parameter is the type of the grammar generated. In our example, Figure 4 shows the language attribute, whose type is *BlockLanguage*, of the functions *block* (line 5) and *var* (line 32).

We use the *dot* notation to call a nonterminal function associated with its correct language attribute. For example, the nonterminals *dlist* and *slist* on the definition of the function *block* are called as *g.dlist(g)* (line 11) and *g1.slist(g1)* (line 15). Note that, as the language attribute passed to each nonterminal is different, we call each nonterminal function from a different language attribute. We must call *slist* from the object *g1* instead of *g* because the vector *adapt* of *g1* has a different value of choices for the function *var*. So, the interpreter is called, the new choice is passed, allowing the use of the variables that have been declared.

A restriction to this approach is that as we use the generated class as the language attribute type, e.g. the *BlockLanguage* type in Figure 4, it is not possible to pass a different grammar which is not subtype of the generated class, as the language attribute. For example, suppose a grammar with other definitions for the same nonterminals presented in Figure 4. One may want to pass as the language attribute this grammar in a specific context on the definition of the block language of Figure 4. However, as this grammar is not a subclass of *BlockLanguage*, there will be a type error. Instead of using the generated class as the language attribute, we could use the base type, *Grammar*, as the language attribute and use reflection on runtime to invoke the nonterminal functions. However, as the use of reflection may result in a slower program than the use of the dot notation to call functions, we avoid this solution.

During the interpretation process of a parsing expression, it is possible to encounter a reference to a predefined nonterminal. In this case, the interpreter must execute the function code of this nonterminal. For example, suppose an input to the block language example of Figure 2 which adds the choice *var* : 'a' !*alpha*⟨*ch*⟩; to the definition of the nonterminal *var*. The nonterminal referenced in this choice, *alpha*, is the one defined in Figure 2 and has a code generated for it. So, when the interpreter reaches this nonterminal, it must stop interpreting and invoke the function of this nonterminal. We implemented this feature using the reflection mechanism of the Java language. Whenever interpreting a nonterminal, the interpreter checks whether the nonterminal is a method of the language attribute object, and if so, the interpreter invokes the method code by reflection. Otherwise, it continues the interpretation.

The code presented in Figure 4 was not automatically generated. In order to test our approach, we first produced handwritten code for some APEG specifications, such as the one presented in Figure 2 and another one discussed in Section 4. For the interpretation, we modified a prototype interpreter we had

developed in a previous work [19]. One of the main modifications was the code for calling, from the interpreter, functions on the generated code. This feature was implemented using reflection in the Java language, as previously discussed. After our mixed approach proposal be proved useful, we will write the code generator to automatically produce a recursive descent parser from an APEG specification.

4 Evaluation of the Mixed Extensible Parser

We have performed preliminary experimental tests to evaluate whether our mixed approach is feasible. We were interested in the performance of the mixed code, i.e., the cost of switching to the interpreter and turning back to the compiled code. So, we built tests which exercises these features. We used two language definitions to evaluate our approach: the block language presented in Section 2 and a version of a data dependent language presented in [9]. The syntax of the data dependent language is an integer followed by the same number of characters enclosed by the symbols "[" and "]". The input *3[abc]* is an example of valid string of this language. Figure 5 shows an APEG grammar for this language. Note that it adds a new choice to the nonterminal *strN* with exactly the number of characters given by the integer value just read, e.g, for the input *3[abc]*, the nonterminal *strN* is extended with the rule *strN* → *CHAR CHAR CHAR*.

```
1  literal[Grammar g]:
2    number<n>
3    { g1 = g + 'strN␣:␣' + concatN('CHAR␣', n) + ';'; }
4    '[' strN<g1> ']'
5  ;
6
7  strN[Grammar g]: {? false } ;
8
9  number returns[int r]: t=[0-9]+ { r = strToInt(t); } ;
10
11 CHAR : . ;
```

Fig. 5. APEG grammar for a data dependent language

We used these APEG grammars because they are simple and demand switching between the compiler and the interpreter. The data dependent example will adapt the grammar once and force the interpreter to return to the code of the *CHAR* function many times. The block language example adapts the grammar several times and also turns back from the interpreter to the compiler code every time a variable is used or declared. We have performed the experiments in a 64-bit, 2.4 GHz Intel Core i5 running Ubuntu 12.04 with 6 GB of RAM on

the Eclipse environment. We have repeated the execution 10 times in a row and computed the average execution time.

Table 1 shows the result for the data dependent language. The inputs used were automatically generated by setting an integer number and then randomically generating this set of characters. The first column shows the value of the integer used in the input string; in this case, the size of the input is proportional to this value. The second column shows the time for parsing the input string, using a prototype interpreter we have developed in a previous work. We have shown that this interpreter presents better performance than similar works, when used for languages requiring extensibility [19]. The third column presents the time for parsing the same input string using our new approach, mixing code generation and interpretation. This result shows that, even though using reflection to switch between interpreter and compiled code is expensive, the efficiency of the compiled code compensates it.

Table 1. Time in milliseconds for parsing data dependent programs. The performance of the interpreter and the mixed approach are compared.

Size	Interpreter	Mixed approach
1000	258	276
10000	1615	2584
100000	68744	36015
150000	181338	86576
200000	445036	164041

Table 2 shows the results of parsing programs of the block structure language of Figure 2. The first column shows the number of variables declared and the second column shows the number of assignment statements in the programs used as input string. These programs were also automatically generated by creating a set of variables and formed assignment statement by chosing two variables of this set. The third and the fourth columns present the time for parsing the programs using the prototype interpreter and using a mixed approach, respectively. The results show that the mixed approach executes slightly faster than the interpreter.

Table 2. Time in milliseconds for parsing block language programs. The performance of the interpreter and the mixed approach are compared.

Variable	Statements	Interpreter	Mixed approach
100	1	176	176
500	1	754	498
1000	1	912	725
100	100	232	263
100	500	448	295
100	1000	506	394

These examples force the use of the slow mechanism of reflection several times. In real cases, we expect the activation of interpretation and reflection is not too frequent, thus the performance of the parser using the mixed approach would be even better. So, our preliminary experiments indicate that the mixed approach can really improve the APEG performance.

5 Related Work

In 1971, based on data from empirical studies, Knuth [12] observed that most programs spend the majority of time executing a minority of code. Using these observations, Dakin and Poole [4] and Dawson [5] (both works published in a same journal issue) independently proposed the idea of "mixed code". The term refers to the implementation of a program as a mixture of compiled native code, generated for the frequently executed parts of the program, and interpreted code, for the less frequently executed parts of the program. Their main motivation was to save memory space with little impact on the execution speed of programs. The interpreted source code is usually smaller although slower than the generated native code. Although with different motivations, this approach has some similarities to our work in the sense that the native code is statically generated (in our case, code for a large part of the parser is statically generated) and interpretation is also used.

Plezbert [15] proposes the use of a mixture of compiled and interpreted code in order to improve programming efficiency during software development. His purpose is to reduce the time spent in the "make" process, considering that programmers repeatedly use the cycle edit-make-execute when developing software. Here, "make" stands for the compilation of files that have been changed, compilation of files dependent on the ones which have been changed, and linking of separately compiled objects into an executable image. The increasingly use of aggressive compiler optimizations causes the make process to take longer. A solution may be the use of interpretation for prototyping, during the development phase, or to turn off optimizations until a final release is to be produced. Plezbert suggests an alternative approach which he calls "continuous compilation". After editing a program, the execution phase can immediately start using interpretation, while the "make" phase is performed concurrently with program execution. The interpreted code is gradually replaced by natively-executable code. Performance increases until, eventually, the entire program has been translated to a fully optimized native form. In our work, extensions for the parser are always interpreted. We could benefit from the "continuous compilation" approach if code is concurrently generated for extensions, without stopping the parsing process. When the code is completely generated and compiled, it could replace the interpreted parts of the parser. Further investigation is necessary, but we believe the opportunities for the performance gains in parsing are smaller then the ones reported by Plezbert.

Just-in-time compilation (JIT) [1], also known as *dynamic translation*, is compilation done during execution of a program – at run time – rather than prior to

execution. JIT is a means to improve the time and space efficiency of programs, using the benefits of compilation (compiled programs run faster) and interpretation (interpreted programs are typically smaller, tend to be more portable and have more access to runtime information). Thompson's paper [23] is frequently cited as one of the first works to use techniques that can be considered JIT compilation, translating regular expressions into IBM 7094 code.

The success of Java has increased the attention to JIT compilation, highlighting the tradeoff between portability of code and efficiency of execution. Several Java implementations have been developed using JIT, such as the ones provided by Sun [3], IBM [22] and the Harissa environment [13,14]. Several improvements to JIT have been proposed, extending the possibilities of mixing interpretation and compilation. For example, Plezbert has shown that good results could be achieved combining JIT compilation with his approach of "continuous compilation", which he called "smart JIT" [16]. Dong Heon Jung et alli [11] add to JIT the approaches of "ahead-of-time compilation" and "idle-time compilation", building a hybrid environment in order to increase the efficiency on a Java-based digital TV platform. It should be noted that our approach is the reversal of the traditional JIT in the sense that we perform a "just-in-time interpretation" during the execution of a compiled code. However, we share the objective of improving execution speed.

All the works discussed so far in this section use a combination of interpretation and code generation, having goals such as performance gain and portability, but not specifically involving parsing. The works discussed in the following are related to improvements on the parsing process, when extensions are considered.

Parsers using a bottom-up approach are usually built as a small, fixed code that is driven by a large parse table, generated from a formal specification of a language. When an extension to the language is required, the entire parse table must be recalculated, which is an expensive process. Several works propose techniques for generating small parse tables for the extended parts of the language, and combining them with the table originally generated for the base language. As an example, Schwerdfeger and Van Wyk [21,20] define conditions for composing parsing tables while guaranteeing deterministic parsing. The algorithm described by Bravenboer and Visser [2] 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 [24], with input grammars defined in SDF [25]. The use of GLR imposes no restrictions on the input grammars, allowing a more natural definition for the syntax than methods based on PEG, such as our approach. On the other hand, GLR does not guarantee linear time processing for the generated parsers. In [17], we have shown that a prototype interpreter using APEG may present better performance results than works based on GLR, when dynamic parser extensions are required, even not considering yet the improvements provided by mixing interpretation with code generation. The obvious reason is that these works are not designed having dynamic extensibility

as an important goal. A disadvantage of the current version of APEG is that it does not offer facilities for modular specifications.

OMeta [26] is a fully dynamic language extension tool, allowing lexically scoped syntax extensions. Similarly to our work, it is based in Parsing Expression Grammar, but it can make use of a number of extensions in order to handle arbitrary kinds of data (not limited to processing streams of characters). Also similarly to our work, OMeta extends PEG with semantic predicates and semantic actions, which can be written using the host language. Programmers may create syntax extensions by writing one or more OMeta productions inside {}s. This creates a new parser object (at parsing time) that inherits from the current parser. In current implementations, everything is processed during parsing time, so they have more in common with our previous work, using only interpretation.

Hansen [8] designed a dynamically extensible parser library and two new algorithms: an algorithm for incremental generation of LL parsers and a practical algorithm for generalized breadthfirst top-down parsing. This work is similar to ours in the sense that the parsers produced may be modified on-the-fly, during parsing time, and it also uses top-down parsing methods. Although the algorithms proposed by Hansen have an exponential worst-case time complexity, the author showed that they may work well in practice. Our approach is based on PEG, so it always produces parsers with linear-time processing, provided by the use of memoization. It may be interesting to implement the examples used by Hansen (a Java grammar and several extensions) in APEG and compare the performance of the two approaches, for parser generation and for parser execution.

6 Conclusion and Future Work

Automatic generation of an extensible parser is difficult because extensions on the syntax may invalidate the generated parser code. In order to ameliorate this problem, in this paper, we propose a novel mixed approach to generate an extensible parser, which combines compilation with interpretation, using Adaptable Parsing Expression Grammars (APEG) as the underlying formal model. The greatest virtue of this proposal is its simplicity, which comes from the APEG model.

Preliminary experiments indicate that the mixed approach can improve the performance of the APEG parser. Our goals in the experiments were to evaluate the mixed code, thus we used languages and examples that exercise this feature. The next steps are to evaluate the performance of the mixed extensible parser in real situations, such as parsing programs of extensible languages like SugarJ [6]. Our strategy is based on the assumption that the code for the base grammar is expected to be large and used many times than the extensions. We still have to prove this assumption.

As APEG allows changing the grammar during the parsing, it opens several possibilities to compose grammars, and in the field of grammar extensibility, allowing, for example, simulate a kind of backbones described in [9]. We plan to investigate and evaluate the performance of the parser in these situations.

As explained in Section 3, we have not developed yet a code generator that may automatically produce a recursive descent parser for the static part of the language specification written in APEG. The examples used in this paper were handwritten and served only for a preliminary tests of the proposed approach. Our next steps include the implementation of this code generator, which will make possible to test for the entire syntax of real extensible languages. Several optimizations on the generated parsers may also be introduced. For example, we can apply techniques to generate code for rules that are used more frequently during interpretation.

References

1. Aycock, J.: A brief history of just-in-time. ACM Comput. Surv. 35(2), 97–113 (2003)
2. Bravenboer, M., Visser, E.: Parse Table Composition – Separate Compilation and Binary Extensibility of Grammars. In: Gašević, D., Lämmel, R., Van Wyk, E. (eds.) SLE 2008. LNCS, vol. 5452, pp. 74–94. Springer, Heidelberg (2009)
3. Cramer, T., Friedman, R., Miller, T., Seberger, D., Wilson, R., Wolczko, M.: Compiling java just in time. IEEE Micro 17(3), 36–43 (1997)
4. Dakin, R.J., Poole, P.C.: A mixed code approach. Comput. J. 16(3), 219–222 (1973)
5. Dawson, J.L.: Combining interpretive code with machine code. Comput. J. 16(3), 216–219 (1973)
6. 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 2011, pp. 391–406. ACM, New York (2011)
7. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. SIGPLAN Not. 39(1), 111–122 (2004)
8. Hansen, C.P.: An Efficient, Dynamically Extensible ELL Parser Library. Master's thesis, Aarhus Universitet (2004)
9. Jim, T., Mandelbaum, Y., Walker, D.: Semantics and algorithms for data-dependent grammars. In: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, pp. 417–430. ACM, New York (2010)
10. Johnson, S.C.: Yacc: Yet Another Compiler Compiler. In: UNIX Programmer's Manual, vol. 2, pp. 353–387. Holt, Rinehart, and Winston, New York (1979)
11. Jung, D.-H., Moon, S.-M., Oh, H.-S.: Hybrid compilation and optimization for java-based digital tv platforms. ACM Trans. Embed. Comput. Syst. 13(2s), 62:1–62:27 (2014)
12. Knuth, D.E.: An empirical study of fortran programs. Software – Practice and Experience 1(2), 105–133 (1971)
13. Muller, G., Moura, B., Bellard, F., Consel, C.: Harissa: A flexible and efficient java environment mixing bytecode and compiled code. In: Proceedings of the 3rd Conference on USENIX Conference on Object-Oriented Technologies (COOTS 1997), vol. 3, p. 1. USENIX Association, Berkeley (1997)
14. Muller, G., Schultz, U.P.: Harissa: A hybrid approach to java execution. IEEE Softw. 16(2), 44–51 (1999)
15. Plezbert, M.: Continuous Compilation for Software Development and Mobile Computing. Master's thesis, Washington University, Saint Louis, Missouri (1996)

16. Plezbert, M.P., Cytron, R.K.: Does "just in time" = "better late than never"? In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1997, pp. 120–131. ACM, New York (1997)
17. Reis, L.V.S., Di Iorio, V.O., Bigonha, R.S.: Defining the syntax of extensible languages. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC 2014, pp. 1570–1576 (2014)
18. dos Santos Reis, L.V., da Silva Bigonha, R., Di Iorio, V.O., de Souza Amorim, L.E.: Adaptable parsing expression grammars. In: de Carvalho Junior, F.H., Barbosa, L.S. (eds.) SBLP 2012. LNCS, vol. 7554, pp. 72–86. Springer, Heidelberg (2012)
19. Reis, L.V.S., Bigonha, R.S., Di Iorio, V.O., Amorim, L.E.S.: The formalization and implementation of adaptable parsing expression grammars. Science of Computer Programming (to appear, 2014)
20. Schwerdfeger, A., Van Wyk, E.: Verifiable parse table composition for deterministic parsing. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 184–203. Springer, Heidelberg (2010)
21. Schwerdfeger, A.C., Van Wyk, E.R.: Verifiable composition of deterministic grammars. In: Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, pp. 199–210. ACM, New York (2009)
22. Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., Nakatani, T.: Overview of the ibm java just-in-time compiler. IBM Syst. J. 39(1), 175–193 (2000)
23. Thompson, K.: Regular expression search algorithm. Commun. ACM 11(6), 419–422 (1968)
24. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Norwell (1985)
25. Visser, E.: Syntax Definition for Language Prototyping. PhD thesis, University of Amsterdam (1997)
26. Warth, A., Piumarta, I.: OMeta: An Object-oriented Language for Pattern Matching. In: Proceedings of the 2007 Symposium on Dynamic Languages, DLS 2007, pp. 11–19. ACM, New York (2007)
27. Watt, D.A., Madsen, O.L.: Extended attribute grammars. Comput. J. 26(2), 142–153 (1983)
28. Wilson, G.V.: Extensible programming for the 21st century. Queue 2(9), 48–57 (2004)