Baltic J. Modern Computing, Vol. 5 (2017), No. 1, pp. 136–145
http://dx.doi.org/10.22364/bjmc.2017.5.1.09

Critical Analysis of Extensible Parsing Tools and Techniques

Audrius ŠAIKŪNAS

Institute of Mathematics and Informatics, Vilnius University, Akademijos 4, LT-08663 Vilnius, Lithuania

tuxmarkv@gmail.com

Abstract. In recent years, a new type of compilers and programming languages has emerged, called extensible compilers and programming languages. These new tools are created in hope to extend lifetime and usability of programming languages by allowing users to define new language constructs with their own syntax and semantics. In this paper we focus on a subset of extensible programming languages, called reflectively extensible programming languages that allow definition of syntax and semantics extensions to be mixed with regular code. To facilitate the creation of such compilers and languages, new parsing and semantic analysis algorithms are required. This paper analyses available extensible parsers, semantic analysers, compilers and highlights further possible research directions in this field. We find that existing parsing, semantic analysis methods, compilers and compiler generators are insufficient for implementation of reflectively extensible programming languages and that creation of new parsing and semantic analysis methods with specific qualities is required to facilitate such implementations.

Keywords: programming languages, compilers, extensible parsing, semantic analysis, reflective grammars, survey

1 Introduction

Programming language and compiler research is one of the classical disciplines of computer science. Despite that the discipline exists for over 60 years, there is no shortage of new programming languages, compilers and ideas. New programming languages are usually created either to solve a new problem in a specific field (e.g., CSS programming language is used to define appearance of web pages) or to replace an existing or obsolete programming language with something more modern. In the latter case, a new programming language is usually created with specific use cases in mind. Whenever a programming language is used beyond of these use cases, source code becomes difficult to write or maintain and often full of boilerplate code. At this point, a new programming

language is created yet again to fix the problems of an existing language. And thus, the cycle repeats again.

To solve this issue, a new kind of compilers and programming languages has been introduced: extensible compilers and programming languages. Extensible compilers and languages allow users to write extensions (often in a form of a compiler plugin) for the compiler which define new language elements with respective syntax and semantics. For example, GCC compiler plugin for C++ language called ODB (WEB, a) introduces extra pragma directives to C++ language that allow mapping data structures to database objects. Another GCC plugin called GCC Python Plugin (WEB, b) embeds Python programming language interpreter into GCC compiler, which enables users to write Python code to analyse internal structure of C/C++ programs that are compiled using the GCC compiler.

In some cases, the extensions can be mixed with the regular code of an extensible programming language. As such, a user of such programming language can extend the programming language that is being used in the same source file which defines the behaviour of the program being written. We call such languages as reflectively extensible programming (REP) languages.

In order to implement compilers for REP languages, new parsing and semantic analysis algorithms and techniques are required, because unlike in traditional programming languages, the syntax and semantics of such REP languages is mutable.

This paper is a critical analysis of existing parsers, semantic analysis methods, compilers and languages that are partially or directly related to implementation of REP languages. The goal is to identify areas of research that need to be completed before a compiler for a reflectively extensible programming language can be implemented.

Just like in non-extensible programming languages, the compilation process can be divided into three discrete steps: parsing, semantic analysis and code generation. While in this paper we primarily focus on analysing available parsing methods that may be suitable for REP language parsing, in section 3 we also touch upon extensible syntax definition methods that are used in existing systems and tools for defining programming languages.

2 Extensible Parsing

Currently there are two main methods used to formally define a grammar: context-free grammars and parsing expression grammars (Ford, 2004) (PEGs). PEGs is a fairly new formalism to used to define grammars that can be unambiguously parsed in linear time using the Packrat parser. Because PEGs place fairly strict requirements on input languages and we wish to allow as non-restrictive grammar extensions as possible, Packrat parsing is outside of this paper's scope.

To support as flexible extensions as possible, we suggest the following requirements for the parser:

- Support for arbitrary context-free parsing. This requirement is necessary to support programming languages commonly used in practise.
- Support for scannerless parsing. There are multiple reasons for this requirement:

- It eliminates the need to separately define scanner tokens and thus allows to reduce overall complexity of the parser.
- Some languages (such as Python) have significant whitespace, while others ignore most of whitespace altogether. Mixing these two programming language styles in a compiler that uses a scanner would be very difficult.
- Scannerless parsing allows easier grammar composition that is necessary when composing base grammar with extension grammars.
- Reduced parser complexity. While there do exist scanner approaches that support ambiguity at lexical level and allow extensible parsing (such as (Wyk and Schwerdfeger, 2007)), the elimination of scanner reduces the overall complexity of the programming language's compiler.

While the requirement for the parser to be scannerless may be debatable, the primary reason why the scanners were used in the parsing process is performance: almost in all cases the scanners that are being used in various programming language compilers run in linear time. However with the advent of more powerful computers, newer parsing methods (such as scanner-less RNGLR (Economopoulos et al., 2009), Yakker (Jim et al., 2010) and Packrat (Ford, 2002)) are fast enough that the use of a dedicated scanner is unnecessary.

There are two parsing algorithm families that satisfy the requirements for a REP language parser: Earley (Earley, 1970) and GLR.

2.1 Naive Extensible Parsing

Theoretically it is possible to adapt any parsing algorithm to allow on-the-fly grammar extension using the following algorithm:

1. Let G_0 be the initial grammar and A_0 the respective parser data (e.g., such as transition tables used in LR parsers).
2. Divide input source into n top level blocks B_0 - B_{n-1} (such as top level declarations in C/C++).
3. Parse and semantically analyse B_i with current parser data A_i. If the current block contains a new language extension, then produce a new grammar composition G_{i+1} based on G_i and the extension. Update the new parser data A_{i+1} based on grammar G_{i+1}.
4. Parse the subsequent block B_{i+1} using parser data A_{i+1}.
5. Repeat steps 2-4 until completion.

Unfortunately, this algorithm is not useful for practical application, because even a trivial change to current grammar would require to fully update the current parser data. This is especially prohibitive when using GLR family parsers that require generating sizeable tables that are used by GLR parsers during parsing. Because of this issue, specialized parsing algorithms are required for reflectively extensible parsing that would support incremental updating of internal data structures used by the parser.

2.2 Parsing Reflective Grammars

The paper (Stansifer and Wand, 2011) describes an algorithm based on the Earley parser that is capable of parsing reflective grammars. Reflective grammars are a type of grammars that can modify their own syntax during parsing. Because this paper's algorithm is heavily based on Earley parser, it also inherits some of it's flaws. Specifically, the performance of parsing even unambiguous languages is fairly low in comparison with LR based parsers, because Earley's algorithm has to dynamically (at run time) traverse grammars in order to perform top-down recognition. Additionally, the parser described in this paper doesn't support data dependent constraints that are necessary to parse more complex programming languages. Both of these issues have been solved by a more modern parser called YAKKER, which performs much better compared to original Earley parser due to additional optimizations and supports arbitrary data dependent constraints.

2.3 Yakker Parser

The YAKKER parser (Jim et al., 2010) is a modern variation of Earley parser. It not only supports parsing arbitrary context-free grammars, and allows placing data dependent constraints. Because of this it is suitable for parsing languages that cannot be defined using pure CFGs, such as well-formed XML and Ruby: in XML language it is required that an opening tag would match a closing tag. Because of this restriction, it is impossible to define XML grammar that would only accept well-formed XML source files using purely context free grammars (see fig. 1). In similar fashion, Ruby's multiline string "here docs" also start and end with a user defined string, which both must be matched in order to determine, where the multi-line string terminates (see fig. 2).

```
<user_tag>
  <element1/>
  <element2/>
  <element3/>
</user_tag>
```

Fig. 1. An example of well-formed XML. The parser needs to store user-defined tag `user_tag` in order to determine where the specified tag ends.

Additionally, this parsing method contains increased performance in comparison with classic Earley parser. Authors achieve this by treating input grammar rules as nondeterministic finite automatas and performing a variation of subset construction of these automatas. This ensures that in unambiguous contexts, during parsing YAKKER parser is only in one state at a time. As a result, the performance of YAKKER makes it suitable for practical application.

However, YAKKER parser directly does not support any grammar extension mechanism that is required to parse REP languages. It still is possible to use a naive extensible

```
puts <<STR
  line1
  line2
  line3
STR
```

Fig. 2. An example of Ruby heredoc multi-line string. Parser needs to store user-defined token `STR` in order to identify where the multi-line string ends.

parsing technique with YAKKER parser, however mixing in multiple grammar extensions in sequence performance-wise would be ineffective, because YAKKER parser relies on subset construction to merge all the input grammar rules into a single automata. This means that YAKKER parser would have to reconstruct the whole parsing automata after inclusion of an additional extension, even when most of the parser's rules would not be used during parsing. Therefore, an incremental automata generation approach for YAKKER parser may be worth investigating in the future.

2.4 Incremental LR Generation

Paper (Cazzola and Vacchi, 2014) describes an algorithm that can be used to incrementally construct LR parser (Knuth, 1965) goto-graphs. While not directly applicable to reflective parsing or YAKKER, this idea may be used in conjunction with described parsing methods to create a new variation of YAKKER, which would be capable of reflective parsing with increased performance and without suffering grammar extension addition penalty (like mentioned in naive parsing section). Such parser would be suitable for implementing a compiler that is capable of parsing reflectively extensible programming languages.

2.5 Specificity parsing

Specificity parsing was first introduced in the METAFRONT System (Brabrand and Schwartzbach, 2007). According to this paper's authors, the METAFRONT system is a tool for specifying flexible, safe, and efficient syntactic transformations between languages defined by context-free grammars. This goal is achieved by using specificity parser to generate the abstract syntax tree for the input program, then using AST transformer to transform the AST from source to destination language and finally using an unparser that produces plain text code for destination language.

The specificity parser is a scannerless, top-down parser. It is unique compared to most other available parsing algorithms, because it allows defining grammars incrementally: each input grammar is essentially a set of grammar productions, which can be added dynamically. This aligns well with one of our goals to implement a language where new syntax can be added on-the-fly during parsing.

However, the specificity parser has several major restrictions:

- Because the specificity parser is a top-down parser, left recursion is disallowed in grammar production definitions. This goes directly against our goals, because we want to allow as non-restrictive input grammars as possible. Manual left-recursion elimination introduces an additional non-terminal with several right-hand alternatives, which causes the input grammar to become more verbose. Automatic left-recursion elimination can in some cases change the associativity of operators, which can introduce semantic errors and confusion when defining language element semantics.
- The specificity parser prohibits syntactic ambiguity. All ambiguous productions are identified when adding them to the parser's production set. While this is very useful for a tool that works only on syntactic level, as it helps to avoid grammar errors, it also restricts the possible set of input grammars. For example, some domain specific language extensions may allow reusing language keywords as identifiers. In such cases, the specificity parser would not be able to parse such grammars, whereas more general parsing algorithms, such as GLR or Earley, would construct an abstract syntax forest, which would represent all possible parse paths. Later on during parsing some of these parse trees may be discarded.
- Certain language productions require additional rules to be parsed correctly. For example, if the input language (such as Java) supports both `&` and `&&` operators, and `&` has higher precedence of the two, then the rule for parsing `&` would "steal" one of the two ampersands when parsing the input fragment `x && y`. Because of this, such fragment would never parse correctly. In order to parse such fragments correctly, the paper's authors introduce additional grammar definition elements called attractors that implement a limited form of lookahead that allows for the parser to identify which rule should be applied. While this solves the issue of parsing the `&` and `&&` operators, it further complicates definition of language/extension grammars.
- There is no direct support for data dependent constraints, which are necessary to allow definition of languages such as well-formed XML without using additional automata to match opening and closing tags.

3 Related Tools And Languages

There already exist multiple languages and compilers that support varying degree of syntactic and semantic extension. In this section we analyse the following tools and languages: JastAdd modular extensible compiler construction system (Ekman and Hedin, 2007), Neverlang framework (Vacchi and Cazzola, 2015) and Katahdin programming language (Seaton, 2007).

3.1 JastAdd compiler construction system

JastAdd (Ekman and Hedin, 2007) is a Java-based system for compiler construction. The tool allows to define new semantic extensions for the Java programming language. For example, it allows user to create an extension to Java programming language that adds nullable types. The extended language semantics are defined using rewritable reference attribute grammars (Ekman and Hedin, 2004) (or ReRAGs). This allows users

to define semantics in a concise and declarative fashion. However, JastAdd is not directly applicable as a parser and/or semantic analyser for a REP language, because of the following reasons:

- User must provide his/her own abstract syntax tree. The tool contains no integrated grammar definition mechanism.
- Extensions of JastAdd are composed into one monolithic compiler executable that contains fixed syntax and semantics after compilation. This goes directly against the idea of REP languages, where extensions can be added and defined directly during compile time of a program.
- Difficult or impossible to compose different syntax extensions, especially when different parsers are used. For example, there may arise the need to incorporate HAML, Python or any other programming language that uses significant whitespace into the base language.
- Extensions share the same variable and function scopes. ReRAGs permit aspect-oriented introduction of new variables and functions that can be used in extension definition. For example, user may define a new variable called `reference_count` in the AST node that represents an expression. Another extension may attempt to define yet another variable with the same name in the same node. While this may not be an issue in a monolithic compiler generator, because such errors are detected during generation of the resulting compiler, this is a more pressing issue in REP language compilers: such errors could only be detected during compiler runtime. This issue is further compounded, when considering the fact that extensions may be defined in external libraries that are authored by different users. As such, certain extensions may become incompatible with one another just because they share a variable with the same name.
- ReRAG extensions are global. If user creates an extension to an existing AST node, then that extension is applied to all instances of such node. In other words, there is no distinction between the original AST node and the extended one. This prohibits scoped extension application, where an extension is applied to only specific portion of input source code.

Even though JastAdd ideas are not directly applicable for use in REP languages, an adapted version of ReRAGs may be used in REP languages to allow dynamic definition of language semantics. However, for ReRAGs to be applicable, a new version of differently scoped ReRAGs is required, where each extension defines new functions and variables within AST tree in a separate namespace, thus isolating one extension from another. Additionally, in order to support scoped extension application, the ReRAGs should not directly modify existing AST node classes. More research is required on this topic in order to create such variation of ReRAGs.

3.2 Neverlang framework

Neverlang (Vacchi and Cazzola, 2015) is a framework that allows simplified programming language construction. It is somewhat similar to JastAdd compiler construction system, but in this case more focus is directed towards creating programming languages

from scratch, without basing them on an existing programming language (in case of JastAdd, it focuses mostly on extending Java programming language). New languages in Neverlang are defined by splitting programming language definition into separate aspects called slices, where individual programming language elements are defined. Each slice contains grammar rules of a specific language element, type checking semantics and evaluation rules. A language definition then combines different slices that are used to generate an interpreter for the new language. While this approach is more user friendly compared to JastAdd (there is no longer a need to use an external parser, different slices can be combined trivially), it too has several reasons why it can't be used in a REP language compiler:

- Only LALR grammars are supported.
- Slices are compiled into monolithic compiler, just like in JastAdd.
- All extensions are global: it is impossible to parse a specific fragment of code with one or more slices temporarily turned off.
- There is no encapsulation in slices. It is possible to define the same variable in multiple slices and to cause compilation error while generating an interpreter for a new language.

3.3 Katahdin programming language

Katahdin (Seaton, 2007) by Chris Seaton is a programming language where the syntax and semantics are mutable at runtime. In other words, it's a reflectively extensible programming language. Both syntax and semantic extensions can be mixed in with the regular Katahdin source code. In order to achieve dynamically mutable syntax, Katahdin uses PEG grammars and a variation of Packrat parser. Once a new rule is added to the grammar, Katahdin compiler creates a corresponding rule node in the tree that represents the current grammar. Once the appropriate `parse` method for the grammar node is called in order to parse given source fragment, the compiler just-in-time compiles the parser rule into native code in order to optimize parser process. However, even with such optimization the parsing process is still too slow to be used in practise. Additionally, the use of PEG grammars excludes too many languages and their extensions to make Katahdin a practically applicable language.

Semantic definition is performed using small-step semantics: each grammar rule is associated with a matching semantic definition of that element written in a high-level, C#-like programming language. The semantic definition has direct access to the parse tree and is directly executed when a code fragment that matches the grammar rule definition is found in the source code. Because of this, it is difficult or impossible to define complex semantics within Katahdin (such as introduction of a new type system; by default, Katahdin is duck-typed). Additionally, such method of semantic definition makes the language very poorly performing. Even the small illustrative example programs take multiple seconds to execute.

To summarize, the following flaws make Katahdin practically inapplicable:

- Slow parsing performance. More optimizations or a more performant algorithm is required.

- Input grammars are limited by PEG restrictions (debatable).
- Restricted semantic definition capabilities. Due to simplistic nature of Katahdin's language element definition, a more powerful semantic definition method is required (such as ReRAGs).
- Poor runtime performance. Compilation to bytecode or machine code is required to alleviate the poor performance.

4 Conclusions

Reflectively extensible programming languages are a fairly niche topic, as evidenced by a shortage of such languages and available parsing algorithms for implementing such languages as of the time of writing this paper. The closest language that fulfils our goals is Katahdin, but it is burdened by major parsing performance, runtime performance and semantic definition restrictions that make it unsuitable for any real-world application.

We find that current parsing and semantic definition methods are insufficient for implementation of REP languages:

- Naive Extensible Parsers lack performance when multiple grammar extensions are used.
- Parser from (Stansifer and Wand, 2011) incrementally extends upon the original Earley parser and as a result isn't efficient enough for practical use.
- The Yakker Parser (Jim et al., 2010) offers great performance, however it does not support reflective grammars.
- Specificity parser (Brabrand and Schwartzbach, 2007) imposes too strict limitations on supported grammars, thus reducing potential expressiveness of language extensions.

In order to construct a REP language compiler, further research of these subjects is required:

- YAKKER parser variation that: 1) is capable of parsing reflective grammars; 2) can construct the optimized grammar automata incrementally and on demand.
- ReRAG variation that preserves encapsulation across different extensions and allows dynamic composition and application of these extensions.

Additionally, none of the analysed compilers and programming languages allow on-the-fly modification of programming language syntax or semantics with acceptable performance and without recompiling or regenerating the whole compiler. Because of this, these tools are unsuitable for a reflectively extensible programming language implementation.

5 Acknowledgements

Thanks to Vilnius University, Institute of Mathematics and Informatics for financing this research.

References

Brabrand, C., Schwartzbach, M. (2007). *The Metafront System: Safe and Extensible Parsing and Transformation*. Science of Comp. Prog., 2–20.

Cazzola, W., Vacchi, E. (2014). *On the incremental growth and shrinkage of LR goto-graphs*. Acta Inf., 419–447.

Earley, J. (1970). *An Efficient Context-free Parsing Algorithm*. Commun. ACM, 94–102.

Economopoulos, G., Klint, P., Vinju, J. (2009). *Faster Scannerless GLR Parsing*. Springer, Berlin, Germany.

Ekman, T., Hedin, G. (2004). *Rewritable Reference Attributed Grammars*. Springer, Berlin, Germany.

Ekman, T., Hedin, G. (2007). *The JastAdd system modular extensible compiler construction*. Science of Comp. Prog., 14–26.

Ford, B. (2002). *Packrat parsing: a practical linear-time algorithm with backtracking*. M.S. Thesis, Massachusetts Institute of Technology, Cambridge, United States.

Ford, B. (2004). *Parsing Expression Grammars: A Recognition-based Syntactic Foundation*. SIGPLAN Not., 111–122.

Jim, T., Mandelbaum, Y., Walker, D. (2010). *Semantics and Algorithms for Data-dependent Grammars*. Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2010, Madrid, Spain), POPL '10, ACM, New York, United States.

Knuth, D. (1965). *On the translation of languages from left to right*. Information and Control, 607–639.

Seaton, C. (2007). *A Programming Language Where the Syntax and Semantics Are Mutable at Runtime*. M.S. Thesis, University of Bristol, Bristol, United Kingdom.

Stansifer, P., Wand, M. (2011). *Parsing Reflective Grammars*. Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications (2011, Saarbrucken, Germany), LDTA '11, ACM, New York, United States.

Vacchi, E., Cazzola, W. (2015). *Neverlang: A framework for feature-oriented language development*. Comp. Lang., Syst. & Struct., 1–40.

WEB (a). *ODB: C++ Object-Relational Mapping (ORM)* (2016). `http://www.codesynthesis.com/products/odb/`

WEB (b). *GCC Python Plugin* (2016). `https://fedorahosted.org/gcc-python-plugin/`

Wyk, E., Schwerdfeger, A. (2007). *Context-aware Scanning for Parsing Extensible Languages*. Proceedings of the 6th International Conference on Generative Programming and Component Engineering (2007, Salzburg, Austria), GPCE '07, ACM, New York, United States.

Zenger, M., Odersky, M. (2001). *Implementing Extensible Compilers*. Proceedings of the ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages (2001, Budapest).

Received September 27, 2016 , revised March 9, 2017, accepted March 13, 2017