Parsing Composed Grammars with Language Boxes

Lukas Diekmann and Laurence Tratt

Software Development Team, King's College London
http://soft-dev.org/

Abstract. Parsing uncovers if, and how, an input stream conforms to a grammar. Language composition requires combining grammars together, yet all traditional parsing techniques have limitations when parsing composed grammars. We augment an incremental parser with *language boxes*, which allows us to retain the feel of traditional parsing whilst allowing arbitrary grammar composition.

1 Introduction

Parsing is the act of taking a stream of textual input and uncovering its underlying structure with respect to a grammar, making later processing much easier. Context Free Grammars (CFGs) are the pragmatic compromise which are most commonly used in parsing—they are fairly expressive, while having a solid theoretical grounding which allows us to prove several desirable properties about them. The two major approaches to CFG parsing are LL and LR parsing. Both approaches can only parse restricted subsets of the CFGs, with the LR subset being less restrictive than the LL subset. While LL parsers can be easily built by hand (typically as recursive descent parsers), LR parsers require large tables to be built, which requires tooling. LL and LR parsing approaches are fairly efficient, while accepting a large enough subset of the CFGs to be usable for programming languages. They are therefore the most commonly used parsing approaches in compilers and related tools.

More recently, there has been renewed interest in languages whose syntaxes can be changed or extended. Such languages come in several different forms (e.g. using textual transformation [1], macros [2], or run-time meta-programming [3]). The basic idea is that languages can be *composed* together i.e. languages A and B are formed to make a new language C (see [4] for some suggestions as to different forms of language composition). Language composition raises many thorny issues, and in this paper we tackle only one: what it means for parsing composed grammars. Since A and B already have their own grammars, they need to be 'put together' in order for C to have a valid parser.

It is commonly assumed that existing parsing approaches are already well suited to this task, or can be made to do so relatively easily. Unfortunately, this is not the case. Every major, extant parsing approach has compromises which make it less than ideal for parsing with composed grammars. Some of these

compromises may be less commonly experienced and some are less severe, but every approach has troubling corner cases.

We are drawn to conclude that arbitrary, safe language composition cannot rely on traditional parser combination. Fortunately, there is a solution: Syntax Directed Editing (SDE). Generally speaking, SDE systems do not allow users to enter text at random: Abstract Syntax Tree (AST) elements are instantiated as templates with holes, which are then filled in. This means that programs being edited are always syntactically valid and unambiguous (though there may be holes with information yet to be filled in). Since the SDE systems of the 70s and 80s were rejected by programmers as restrictive and clumsy, our challenge has been to retain SDE's benefits while ridding ourselves of its problems.

In this paper we extend Wagner and Graham's incremental parsing algorithm [5] with the new concept of *language boxes*. The resulting editor allows users to compose languages in a way which retains SDE's benefits whilst retaining, in the general case, the feel of a normal text editor. A video of the editor in action can be found at `http://www.youtube.com/watch?v=LMzrTb22Ot8`.

2 Parsing composed grammars

Since parsing using composed grammars is a largely unexplored area, there is no good definition of exactly what is meant by that term. For this paper, two conditions provide a reasonable intuition. First, a composed grammar must be able to accept *all* the inputs of both its constituent grammars, though it need not accept such inputs at the top-level; in other words, one language may be embedded in another. Second, any meaningful composition will require 'glue' to combine the two constituent grammars. CFG union is one example of a widely known grammar composition operator which satisfies these criteria. In the rest of this section we briefly survey the major design points in the parsing universe.

2.1 LL and LR parsing

LL and LR parsers are the most common type of parsers in practice and are thus the natural place to start when considering parser composition. LR(k) are the most powerful of this class and have the useful property that they accept only unambiguous CFGs. However, Parikh's theorem [6] shows that the composition of two LR/LL grammars does not, in general, result in a valid LR/LL grammar.

2.2 Parsing arbitrary CFGs

It was not until the early 1970s that a practical algorithm for parsing the whole class of CFGs was introduced by Earley [7]. It, and its descendents, seem well suited to parsing composed grammars as we know that two composed CFGs result in a valid CFG. However, parsing composed CFGs is problematic for several reasons. The first relates to tokenization, the second to ambiguity.

Composed grammars may have conflicting tokenization rules. Scannerless parsing [8] – a necessary prerequisite for most language composition – aims to

solve this, but the 'reject' rules of ASF+SDF can specify context sensitive languages, which change the guarantees that can be made about such grammars in ways that are not yet fully understood [9]. Furthermore, combining tokenizers and parsers means that the resultant grammars are potentially ambiguous e.g. due to the 'longest match' problem [10].

Even if tokenization issues were solved, the issue of ambiguity raises its head. Ambiguity is disastrous for programming language tools, which can hardly ask of a user "which parse of many did you intend?" Two unambiguous grammars, when composed, may become ambiguous. However, we know that, in general, we can not tell if an arbitrary CFG is ambiguous or not without trying all the possible inputs [11]. Although heuristics for detecting ambiguity exist, all existing approaches fail to detect at least some ambiguous grammars [12].

2.3 PEGs

Parsing Expression Grammars[1] (PEGs) are a modern update of a classic parsing approach [13]. Unlike the approaches previously discussed, PEGs have no relation at all to CFGs. The chief reason for this is the PEG *ordered choice* operator e_1 `/` e_2. This means 'try e_1 first; if it succeeds, the ordered choice immediately succeeds and completes. Only if e_1 fails should e_2 be tried.'

PEGs are appealing for parsing composed languages because PEGs are closed under composition and inherently unambiguous. However, the ordered choice operator, which must be used when composing grammars, can cause the resulting PEG not to parse inputs valid in its constituent grammars. Assume the LHS and RHS of the ordered choice in the well known PEG `S ::= a / ab` are composed grammars. If the LHS matches `a`, the ordered choice matches and returns success to its container; the RHS is not given a chance to match anything, even if it could have matched a longer input sequence.

3 Syntax directed editing

Because we can't parse files written using composed languages reliably, language composition has remained something of a pipe-dream. If a good solution to editing composed languages cannot be found, the other problems that language composition raises are irrelevant.

SDE offers a solution, because by working on an AST at all times, arbitrary languages can be mixed together trivially. However, SDE tools of the 70s and 80s (see e.g. [14,15]) were almost universally rejected by programmers as restrictive, clumsy, and unnecessary. Fortunately for us, recent tools have shown that SDE can be made much less restrictive and clumsy. Perhaps the best example is JetBrain's MPS system[2], which provides a generic SDE environment into which a reasonable variety of syntaxes can be combined. While the user experience is good enough to be acceptable to some programmers as-is, it is considerably different to using a traditional text editor.

[1] Packrat parsers are an optimisation of PEGs, and thus of little interest to this paper.
[2] `http://jetbrains.com/mps/`

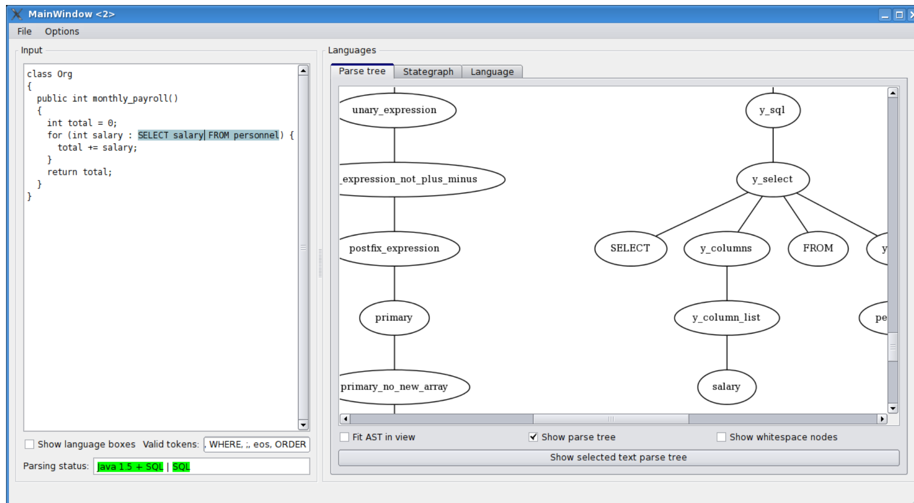

Fig. 1. The editor tool displaying a composed Java and SQL program. The two green parsing statuses in the bottom left indicate that both the Java and SQL programs are valid. The cursor is inside the SQL language box, so it is highlighted. On the right ('Parse Tree') you can see a visualization of the parse tree, containing both Java (on the left e.g. 'primary') and SQL (on the right e.g. 'SELECT') fragments.

4 Language boxes

Our solution to these problems is to extend the concept of incremental parsing, an 'online' parsing approach which builds and maintains a parse tree (i.e. it still contains tokens that might be considered unnecessary in an AST) as the user types into an editor. In other words, parsing is a continuous activity, rather than being performed occasionally as part of a compilation process.[3] We use Wagner and Graham's incremental parsing algorithm [5] as our base.

We extend the incremental parser with *language boxes*, which are tokens with arbitrary contents. This is different from traditional context free parsing, which first tokenizes text into *(type, value)* pairs and then parses them into a parse tree. We make use of a simple observation: while the value of a token is used when deriving its type in the tokenization phase, the value is opaque to the parsing phase (to do otherwise would be to make the parser context sensitive). In an incremental parser, where a parse tree is persistent, we can then untangle a token's type from its value. Provided we can create a token of a given type, its contents are irrelevant—we need not be constrained by the convention that a token's value is a sequence of ASCII/Unicode characters related to its type. Language boxes are thus tokens whose value need have no relation to its type.

[3] Although slower than using a non-incremental parser, an incremental parser can be used to read in existing files.

The resulting prototype editor is shown in Figure 1. Language boxes always encompass their contents, growing as needed; a language box's contents can never leak to the outside. Since they must, in general, be manually created, they force the user to explicitly disambiguate which language is which. They require no special start or end markers to denote their existence; there is never any possibility of ambiguity in composition; nor of text accidentally 'leaking' from the language box. Users can, though, manually copy and paste code from inside a language box to outside (and vice versa). In the basic version of the editor, each language box has a 'fresh' incremental parser which is unaware of the existence (or not) of an outer language box. Language boxes can be nested arbitrarily deep, and any given language box can contain multiple nested language boxes at different points. In general, language boxes are not explicitly highlighted, because we hope that programs using composed grammars make sense without the seams between grammars constantly being highlighted; however, moving the cursor into a language box does highlight its extent.

Using language boxes, we can trivially solve the 'classic' grammar composition problem of allowing SQL expressions to be used wherever a Java RHS expression can be used. Let us assume the existence of an SQL grammar named `SQL` and a standard Java 1.5 grammar derived from the Java standard. We extend the Java rule `unary_expression`[4] to reference the SQL grammar as follows:

```
unary_expression ::=
      preincrement_expression | predecrement_expression | ... | <SQL>
```

Since one cannot type an `<SQL>` token in the normal fashion, pressing `Ctrl+Space` opens up a drop-down box from which a language box of a given type can be created; this implicitly creates a parse tree node of type `SQL`. Language boxes can be placed anywhere the user desires, even where they are not, by the grammar's rules, syntactically valid. This loosens traditional SDE's restrictions, and allows users to edit files in any way they choose.

In a further departure from traditional editors, the only representation in our editor is the parse tree. What looks like a standard textual GUI component on the left of Figure 1 is actually a raw canvas, on which the text in the parse tree is then manually rendered. Since language boxes are opaque nodes in the the parse tree (akin to terminals), editors for non-textual languages can be used in them. Our editor thus has experimental support for visualising, but not editing, non-textual languages. Allowing editing of, and embedding within, non-textual languages is one of our major future tasks.

5 Conclusions and future work

Language boxes allow us to achieve the benefits of SDE without the downsides. Like SDE tools, language boxes make it possible to neatly sidestep the problems associated with ambiguity (when parsing CFGs) and shadowing (when parsing

[4] Because of the sheer size of the Java grammar, many other rules could also have been used as the target.

PEGs). Unlike normal SDE tools, language boxes only need to be used at the boundary between composed languages: when not using language boxes, our editor feels just like a normal text editor. Importantly, language boxes can have non-incremental parser editors: non-textual languages are naturally supported.

The chief limitation of our approach is that the incremental parser is inherently LR-based. For languages which need less restricted grammars, a language box with a different parsing mechanism could be used. At worst, one can simply embed an arbitrary parser in a language box. It may, however, be awkward to allow nested language boxes in such a scenario.

An interesting challenge will be to see if, and to what degree, our editor can automatically detect from the user's input that a language box is needed.. Clearly, the theory from Section 2 tells us that this is impossible in general: it remains to be seen whether it is practical in enough cases to be worthwhile, and whether it feels like too much like 'magic' to be tolerable.

Acknowledgements: We thank Naveneetha Vasudevan and Edd Barrett for comments. This research was graciously funded by a grant from Oracle.

References

1. M. Bravenboer and E. Visser, "Concrete syntax for objects. DSL embedding and assimilation without restrictions," in *Proc. OOPSLA'04*, Oct. 2004.
2. L. Tratt, "Domain specific language implementation via compile-time metaprogramming," *TOPLAS*, vol. 30, no. 6, pp. 1–40, 2008.
3. C. Seaton, "A programming language where the syntax and semantics are mutable at runtime," Master's thesis, University of Bristol, May 2007.
4. S. Erdweg, P. G. Giarrusso, and T. Rendel, "Language composition untangled," in *Proc. LDTA*, 2012.
5. T. A. Wagner and S. L. Graham, "Efficient and flexible incremental parsing," *ACM TOPLAS*, vol. 20, pp. 980–1013, Sept. 1998.
6. R. J. Parikh, "On context-free languages," *J. ACM*, vol. 13, pp. 570–581, Oct. 1966.
7. J. Earley, "An efficient context-free parsing algorithm," *Communications of the ACM*, vol. 13, Feb. 1970.
8. E. Visser, "Scannerless generalized-LR parsing," Tech. Rep. P9707, Programming Research Group, University of Amsterdam, July 1997.
9. J. van Eijck, "Let's accept rejects, but only after repairs," in *Liber Amicorum Paul Klint*, pp. 117–128, Nov. 2007.
10. D. J. Salomon and G. V. Cormack, "Scannerless NSLR(1) parsing of programming languages," *SIGPLAN Not.*, vol. 24, pp. 170–178, June 1989.
11. D. G. Cantor, "On the ambiguity problem of backus systems," *J. ACM*, vol. 9, pp. 477–479, Oct. 1962.
12. N. Vasudevan and L. Tratt, "Search-based ambiguity detection in context-free grammars," in *Proc. ICCSW*, pp. 142–148, Sept. 2012.
13. B. Ford, "Parsing expression grammars: a recognition-based syntactic foundation," in *Proc. POPL*, pp. 111–122, Jan. 2004.
14. W. J. Hansen, "User engineering principles for interactive systems," in *Proc. AFIPS '71*, pp. 523–532, 1971.
15. T. Teitelbaum and T. Reps, "The Cornell program synthesizer: a syntax-directed programming environment," *Commun. ACM*, vol. 24, no. 9, pp. 563–573, 1981.