Available online at www.sciencedirect.com

SCIENCE DIRECT®

Science of Computer Programming 61 (2006) 211–227

Science of Computer Programming

www.elsevier.com/locate/scico

XGLR—an algorithm for ambiguity in programming languages[☆]

Andrew Begel[*], Susan L. Graham

Computer Science Division – EECS, University of California, Berkeley, CA 94720-1776, USA

Available online 19 May 2006

Abstract

Automatically generated lexers and parsers for programming languages have a long history. Although they are well suited for many languages, many widely used generators, among them Flex and Bison, fail to handle input stream ambiguities that arise in embedded languages, in legacy languages, and in programming by voice. We have developed Blender, a combined lexer and parser generator that enables designers to describe many classes of embedded languages and to handle ambiguities in spoken input and in legacy languages. We have enhanced the incremental lexing and parsing algorithms in our Harmonia framework to analyse lexical, syntactic and semantic ambiguities. The combination of better language description and enhanced analysis provides a powerful platform on which to build the next generation of language analysis tools.
© 2006 Elsevier B.V. All rights reserved.

Keywords: GLR; XGLR; Embedded languages; Harmonia; Programming-by-voice

1. Introduction

Automatically generated lexers and parsers for programming languages have long been essential tools for constructing language analysis environments. Many widely used lexer and parser generators, among them Flex [1] and Bison [2], are well suited for describing a broad class of programming languages that are designed to be unambiguous. These tools are ill suited for handling input stream ambiguities that arise from legacy languages, from non-keyboard-based input such as programming by voice, and from embedded languages. The ambiguities may be lexical, syntactic or semantic.

The contributions reported in this paper are two-fold:

(1) improved methods for syntax analysis that handle these kinds of ambiguities;
(2) a new combined lexer and parser generator and further parser enhancements that facilitate the description and analysis of embedded languages.

[☆] The authors thank the members of the Harmonia Group, past and present, for their contributions to this work. The referees made helpful suggestions that improved the paper. They also pointed out some additional references. A preliminary report on this work appeared in the LDTA 2004 workshop. This research was supported in part by NSF Grant CCR-0098314 and by an IBM Eclipse Innovation Grant.

[*] Corresponding address: One Microsoft Way, 98052 Redmond, WA, United States. Tel.: +1 425 705 1816; fax: +1 425 936 7329.
E-mail addresses: Andrew.Begel@microsoft.com (A. Begel), graham@cs.berkeley.edu (S.L. Graham).

Programming by voice, a novel form of user interface enabling the user to edit, navigate, and dictate code using voice recognition software, is a recent programming technique supported by the increased power of desktop computers to process speech accurately [3]. It uses a combination of commands and program fragments, rather than full-blown natural language. Spoken input, however, contains many lexical ambiguities, such as homophones,[1] misrecognized words, and an inability to recognize unpronounceable or concatenated words. When the input is for an English or other natural language document, it can be disambiguated by a hidden Markov model provided by the speech recognition vendor. However, when the input is a computer program, the natural language disambiguation rules provided by commercial speech tools cannot be applied. Not only do these ambiguities affect the voice-based programmer's ability to introduce code, they also affect the ability of the voice-based programmer to use similar sounding words in different contexts.

Some legacy languages like PL/I and Fortran present difficulties to both a Flex-based lexer and an LALR(1) based parser. PL/I, in particular, does not have reserved keywords, meaning that `IF` and `THEN` may be both keywords and variables. A lexer cannot distinguish between them; only the parser and static semantics have enough context to choose. Fortran's optional white-space rule leads to insidious lexical ambiguities. For example, `DO57I` can designate either a single identifier or `DO 57 I`, the initial portion of a Do loop. Without syntactic support, a particular character sequence could be interpreted using several sets of token boundaries. Feldman [4] summarizes other difficulties that arise in analysing Fortran programs.

Embedded languages, in which fragments of one language can be embedded within another language, are in widespread use in common application domains such as Web servers (e.g. PHP embedded in XHTML), data retrieval engines (e.g. SQL embedded in C), and structured documentation (e.g. Javadoc embedded in Java). The boundaries between languages within a document can be either fuzzy or strict; detecting them might require lexical, syntactic, semantic or customized analysis.

The lack of composition mechanisms in Flex and Bison for describing embedded languages makes independent maintenance of each component language unwieldy and combined analysis awkward. Other language analyser generators, such as ANTLR [5], SPARK [6], or ASF+SDF [7], provide better structuring mechanisms for language descriptions, but differing language conventions for comments, white-space, and token boundaries complicate both the descriptions of embedded languages and the analyses of their programs, particularly in the presence of errors.

Section 2 of this paper summarizes the Harmonia framework within which our enhanced methods are implemented. The methods described in Section 3 handle four kinds of input streams: (1) single spelling, single lexical type; (2) multiple spellings, single lexical type; (3) single spelling, multiple lexical types; and (4) multiple spellings, multiple lexical types. The last three are ambiguous. Combinations of these ambiguities arise in different forms of embedded languages. The handling of input streams containing such combinations is presented in Sections 4–7. Some of these ambiguities have also been addressed in related work, which is summarized in Section 9.

Single spelling, single lexical type. This is normal, unambiguous lexing (i.e. a sequence of characters produces a unique sequence of tokens). We illustrate this case to show how lexing and parsing work in the Harmonia analysis framework.

Multiple spellings, single lexical type. Programming by voice introduces potential ambiguities into programming that do not occur when legal programs are typed. If the user speaks a homophone which corresponds to multiple lexemes (for example, `i` and `eye`), and all the lexemes are of the same lexical type (the token `IDENTIFIER`), using one or the other homophone may change the meaning of the program. Multiple spellings of a single lexical type might also be used to model voice recognition errors or lexical misspellings of typed lexemes (e.g. the identifier `counter` occurring instead as `conter`).

Single spelling, multiple lexical types. Most languages are easily described by separating lexemes into separate categories, such as keywords and identifiers. However, in some languages, the distinction is not enforced by the language definition. For instance, in PL/I, keywords are not reserved, leading a simple lexeme like 'IF' or 'THEN' to be interpreted as both a keyword and an identifier. In such cases, a single character stream is interpreted by a lexer as a unique sequence of lexemes, but some lexemes may denote multiple alternate tokens, which each have a unique lexical type.

[1] Homophones are words that sound alike but have different spellings.

Multiple spellings, multiple lexical types. Sometimes a user might speak a homophone (e.g. ‘for’, ‘4’ and ‘fore’) that not only has more than one spelling, but that have distinct lexical types (e.g. keyword, number and identifier).

Embedded languages. Two issues arise in the analysis of embedded languages—identifying the boundaries between languages, and analysing the outer and inner languages according to their differing lexical, structural, and semantic rules. Once the boundaries are identified, any ambiguities in the inner and outer languages can be handled as if embedding were absent. However, ambiguity in identifying a boundary leads to ambiguity in which language’s rules to apply when analysing subsequent input. Virtually all programming languages admit simple embeddings, notably strings and comments. The embedding in an example such as Javadoc within Java is more complex. These embeddings are typically processed by ad hoc techniques. When properly described, they can be identified in a more principled fashion. For example, Synytskyy et al. [8] use island grammars to analyse multilingual documents from web applications. Their approach is summarized in Section 9.

The results described in this paper require modifications to conventional lexers and parsers, whether batch or the incremental versions used in interactive environments. Our approach is based on GLR parsing [9], a form of general context free parsing based on LR parsing, in which multiple parses are constructed simultaneously. Even without input ambiguities, the use of GLR instead of LR parsing enables support for ambiguities in the *analysis* of an input stream. GLR tolerates local ambiguities by forking multiple parsers, yet is efficient because the common parts of the parsers are shared. In addition, for the syntax specifications of most programming languages, the amount of ambiguity that arises is bounded and fairly small. Our contribution is to generalize this notion of ambiguity, and the GLR parsing method, to parse inputs that are locally different (whether due to the embedding of languages, the presence of homophones or other lexically identified ambiguities). We call this enhanced parser XGLR.

We have strengthened the language analysis capabilities of our Harmonia analysis framework [10,11] to handle these kinds of ambiguities. Our research in programming by voice requires interactive analysis of input stream ambiguities. Harmonia can now identify ambiguous lexemes in spoken input. In addition, Harmonia’s new ability to embed multiple formal language descriptions enables us to create a voice-based command language for editing and navigating source code. This new input language combines a command language written in a structured, natural-language style (with a formally specified syntax and semantics) with code excerpts from the programming language in which the programmer is coding.

To realize these additional capabilities, the parser requires additional data structures to maintain extra lexical information (such as its own private lookahead token and its own private lexer state), as well as an enhanced interface to the lexer. These changes enable the XGLR parser to resolve *shift–shift* conflicts that arise from the ambiguous nature of the parser’s input stream. The lexer must be augmented with a bit of extra control logic. A completely new lexer and parser generator called *Blender* was developed. Blender produces a lexical analyser, parse tables and syntax tree node C++ classes for representing syntax tree nodes in the parse tree. It enables language designers to describe many classes of embedded languages easily (including recursively nested languages) and supports many kinds of lexical, structural and semantic ambiguities at each stage of analysis. In the next section, we summarize the structure of incremental lexing and GLR parsing, as realized in Harmonia. The changes to support input ambiguity and the design of Blender follow.

2. Lexing and parsing in Harmonia

Harmonia is an open, extensible framework for constructing interactive language-aware programming tools. Programs can be edited and transformed according to their structural and semantic properties. High-level transformation operations can be created and maintained in the program representation. Harmonia furnishes the XEmacs [12] and Eclipse [13] programming editors with interactive, on-line services to be used by the end user during program composition, editing and navigation.

Support for each user language is provided by a plug-in module consisting of a lexical description, syntax description and semantic analysis definition. The framework maintains a versioned, annotated parse tree that retains all edits made by the user (or other tools) and all analyses that have ever been executed [14]. When the user makes a keyboard-based edit, the editor finds the lexemes (i.e. the terminal nodes of the tree) that have been modified and updates their text, temporarily invalidating the tree because the changes are unanalysed. If the input was spoken, the words from the voice recognizer are turned into a new unanalysed terminal node and added to the appropriate location in the parse tree. These changes make up the most recently edited version (also called the last edited version). This version of the tree and the pre-edited version are used by an incremental lexer and parser to analyse and reconcile the changes in the tree.

Harmonia employs incremental versions of lexing and sentential-form GLR parsing [15–18] in order to maintain good interactive performance. For those unfamiliar with GLR, one can think of GLR parsing as a variant of LR parsing. In LR parsing, a parser generator produces a parse table that maps a parse state/lookahead token pair to an action of the parser automaton: shift, reduce using a particular grammar rule, or declare error. The table contains only one action for each parse state/lookahead pair. Multiple potential actions (conflicts) must be resolved at table construction time. In addition to the parse table and the driver, an LR parser consists of an input stream of tokens and a stack upon which to shift grammar terminals and non-terminals. At each step, the current lookahead token is paired with the current parse state and looked up in the parse table. The table tells the parser which action to perform and, in the absence of an error, the parse state to which it should transition.

The GLR algorithm used in Harmonia is similar to that described by Rekers [19] and by Visser [20]. In GLR parsing, conflict resolution is deferred to run-time, and all actions are placed in the table. When more than one action per look-up is encountered, the GLR parser forks into multiple parsers sharing the same automaton, the same initial portion of the stack, and the same current state. Each forked parser performs one of the actions. The parsers execute in pseudo parallel, each executing all possible parsing steps for the next input token before the input is advanced (and forking additional parsers if necessary), and each maintaining its own additional stack. When a parser fails to find any actions in its table look-up, it is terminated; when all parsers fail to make progress, the parse has failed, and error recovery ensues. Parsers are merged when they reach identical states after a reduce or shift action. Thus, conceptually, the forked parsers either construct multiple subtrees below a common subtree root, representing alternative analyses of a portion of the common input, or they eventually eliminate all but one of the alternatives.

The basic non-incremental form of the GLR algorithm (before any of our changes) is shown in Fig. 1.[2] In GLR parsing, each parser stack is represented as a linked structure so that common portions can be shared. Each parser state in a list of parsers contains not only the current state recorded in the top entry, but also pointers to the rest of all stacks for which it is the topmost element. In Fig. 1, the algorithm is abstracted to show only those aspects changed by our methods. In particular, parse stack sharing is implicit. Thus *push* q *on stack* p means to advance all the specified parsers with current state p to current state q. The current lookahead token is held in a global variable *lookahead*.

In a batch LR or GLR parse, the sentential form associated with a parser at any stage is the sequence of symbols on its stack (read bottom-to-top) followed by the sequence of remaining input tokens. Conceptually, they represent a parse forest that is being built into a single parse tree. In an incremental parser, both the symbols on the stack and the symbols in the input may be parse (sub)trees (see Fig. 2)—one can think of them as potentially a non-canonical sentential form. The goal of an incremental or change-based analysis is to preserve as much as possible of the parse prior to a change, updating it only as much as is needed to incorporate the change.

The result of lexing and parsing is sometimes a parse forest made up of all possible parse trees. Semantic analysis must be used to disambiguate any valid parses that are incorrect with respect to the language semantics. For example, to disambiguate identifiers that ought to be concatenated (but were entered as separate words because they came from a voice recognizer) the semantic phase can use symbol table information to identify all in-scope names of the appropriate kind (method name, field name, local variable name, etc) that match a concatenated sequence of identifiers that is semantically correct. Care with analysis must be taken if an inner language can access the semantics of the outer (e.g. Javascript can reference objects from the HTML code in which it is embedded). Semantic analyses techniques are interesting and important, but an in-depth discussion of this topic is beyond the scope of this paper.

3. Ambiguous lexemes and tokens

In Section 1, we classified token ambiguities into four types (including unambiguous tokens). We next explain how these situations are handled.

[2] The addition of incrementality is not essential for understanding the changes made here and is not shown.

```
GLR-PARSE()
init active-parsers list to parse state 0
init parsers-ready-to-act list to empty
while not done
  PARSE-NEXT-SYMBOL()
  if accept before end of input
    invoke error recovery
accept

PARSE-NEXT-SYMBOL()
lex one lookahead token
init shiftable-parse-states list to empty
copy active-parsers list to
      parsers-ready-to-act list
while parsers-ready-to-act list $\neq \emptyset$
  remove parse state p from list
  DO-ACTIONS(p)
SHIFT-A-SYMBOL()

DO-ACTIONS(parse state p)
look up actions[$p \times$ lookahead]
for each action
  if action is SHIFT to state x
    add <p, x> to shiftable-parse-states
  if action is REDUCE by rule y
    if rule y is accepting reduction
      if at end of input return
      if parsers-ready-to-act list $= \emptyset$
        invoke error recovery
      return
    DO-REDUCTIONS(p, rule y)
    if no parsers ready to act or shift
      invoke error recovery and return
  if action is ERROR and no parsers
        ready to act or shift
    invoke error recovery and return

DO-REDUCTIONS(parse state p, rule y)
for each parse state $p^-$ below RHS(rule y)
      on a stack for parse state p
  let q = GOTO state for
      actions[$p^- \times$ LHS(rule y)]
  if parse state $q \in$ active-parsers list
    if $p^-$ is not immediately below stack
        for parse state q
      push q on stack $p^-$
      for each parse state r such that
          $r \in$ active-parsers list and
          $r \notin$ parsers-ready-to-act list
        DO-LIMITED-REDUCTIONS(r)
  else
    create new parse state q
    push q on stack $p^-$
    add q to active-parsers list
    add q to parsers-ready-to-act list

DO-LIMITED-REDUCTIONS(parse state r)
look up actions[$r \times$ lookahead]
for each REDUCE by rule y action
  if rule y is not accepting reduction
    DO-REDUCTIONS(r, rule y)

SHIFT-A-SYMBOL()
clear active-parsers list
for each <p, x> $\in$ shiftable-parse-states
  if parse state $x \in$ active-parsers list
    push x on stack p
  else
    create new parse state x
    push x on stack p
    add x to active-parsers list
```

Fig. 1. A non-incremental version of the unmodified GLR parsing algorithm.

3.1. Single spelling—one lexical type

Unambiguous lexing and parsing is the normal state of our analysis framework. Programming languages have mostly straightforward language descriptions, only incorporating bounded ambiguities when described using GLR. Thus, the typical process of the lexer and parser is as follows. The incremental parser identifies the location of the edited node in the last edited parse tree and invokes the incremental lexer. The incremental lexer looks at a previously computed *lookback* value (stored in each token) to identify how many tokens back in the input stream to start lexing due to the change in this token.[3] The characters of the starting token are fed to the Flex-based lexical analyser one at a time until a regular expression is matched. The action associated with the regular expression creates a single,

[3] Lookback is computed as a function of the number of lookahead characters used by the batch lexer when the token is lexed [15].

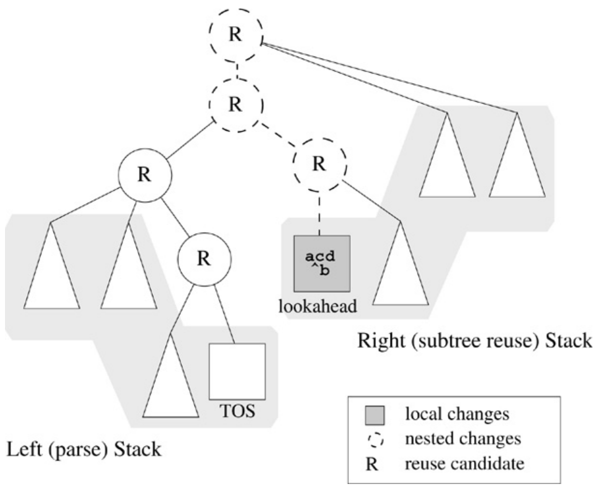

Fig. 2. A change in the spelling of an identifier has resulted in a split of the parse tree from the root to the token containing the modified text. In an incremental parse, the shaded portion on the left becomes the initial contents of the parse stack. The shaded portion on the right represents the potentially reusable portion of the input stream. Parsing proceeds from the TOS (top of stack) until the rest of the tree in the input stream has been re-incorporated into the parse.

unambiguous token, which is returned to the parser to use as its lookahead symbol. In response to the parser asking for tokens, lexing continues until the next token is a token that is already in the edited version of the syntax tree. (The details of parser incrementality are not essential to this discussion and are omitted for brevity. Notice that additional information must be stored in each tree node to support incrementality.)

3.2. *Single spelling—multiple lexical types*

If a single character sequence can designate multiple lexical types, as in PL/I, tokens are created for each interpretation (containing the same text, but differing lexical types) and are all inserted into an *AmbigNode* container. When the lexer/parser interface sees an AmbigNode, namely, multiple alternate tokens, that AmbigNode represents a shift–shift conflict for the parser. A new lexer instance is created for each token, and a separate parser is created for each lexer instance. Thus each parser has its own (possibly shared) lexer and its own lookahead token. The GLR parse is carried out as usual, except that instead of a global lookahead token, the parsers have local lookaheads with a shared representation. Due to this change, the criteria for merging parsers includes not only that the parse states are equal, but that the lookahead token and the state of each parser's lexer instance are the same as well.

Fig. 3 shows our modification of the *PARSE-NEXT-SYMBOL()* function. Note that both *lex* and *lookahead* are now associated with a parser p rather than being global. Not shown are the changes to the parser merging criteria in *DO-REDUCTIONS()* and to the creation of new parse states (which should be associated with the current *lex* and *lookahead*). In addition, each look-up must reference the lookahead associated with its parser—for example, $actions[p \times lookahead_p]$.

3.3. *Multiple spellings—one lexical type*

Harmonia's voice-based editing system looks up words entered by voice recognition in a homophone database to retrieve all possible spellings for that word. The lexer is invoked on each word to discover its lexical type and create a token to contain it. If all alternatives have the same lexical type (e.g. all are identifiers), they are returned to the parser in a container token called a *MultiText*, which, to the parser, appears as a single, unambiguous token of a single lexical type. Once incorporated into the parse tree, semantic analysis can be used to select among the homophones.

A similar mechanism could be used for automated semantic error recovery. Identifiers can easily be misspelled by a user when typing on a keyboard. Compilers have long supported substituting similarly spelled (or phonetically similar) words for the incorrect identifier. In an incremental setting, where the program, parse, and symbol table information are persistent, error recovery could replace the user's erroneous identifier with an ambiguous variant that

```
PARSE-NEXT-SYMBOL()
for each parse state $p \in$ active-parsers list
   set lookahead$_p$ to first token lexed by lex$_p$
   if lookahead$_p$ is ambiguous
      let each of $q_1$ .. $q_n$ = copy parse state $p$
      for each parse state $q \in q_1$ .. $q_n$
         for each alternative $a$ from lookahead$_p$
            set lookahead$_q$ to $a$
         add $q$ to active-parsers list
init shiftable-parse-states list to empty
copy active-parsers list to parsers-ready-to-act list
while parsers-ready-to-act list $\neq \emptyset$
   remove parse state $p$ from list
   DO-ACTIONS(p)
SHIFT-A-SYMBOL()
```

Fig. 3. Part of the XGLR parsing algorithm modified to support ambiguous lexemes.

contains the original identifier along with possible alternate spellings. Further analysis might be able to choose the proper alternative automatically based on the active symbol table. We have not yet investigated this application.

3.4. Multiple spellings—multiple lexical types

If the alternate spellings for a spoken word (as described above) have differing lexical types (such as 4/`for`/`fore`), they are returned to the parser as individual tokens grouped in the same AmbigNode container described above. When the lexer/parser interface sees an AmbigNode, it forks the parser and lexer instance, and assigns one token to each lexer instance.[4] The state of each lexer instance must be reset to the lexical state encountered after lexing its assigned alternative, since each spelling variant may traverse a different path through the lexer automaton.[5] Once each token is re-lexed, it is returned to its associated parser to be used as its lookahead token and shifted into the parse tree.

4. The nature of embedded languages

Using Blender, the outer and inner languages that constitute an embedded language can be specified by two completely independent language definitions, for example, one for PHP and another for XHTML, which are composed to produce the final language analysis tool. Embedded language descriptions may be arbitrarily nested and mutually recursive. It is the job of the language description writer to provide appropriate boundary descriptions.

4.1. Boundary identification

In embedded languages, boundaries between languages may be designated by context (e.g. the format control in C's `printf` utility), or by delimiter tokens before and after the inner language occurrence. The delimiters may or may not be distinct from one another; they may or may not belong to the outer (resp. inner) language, and they may or may not have other meanings in the inner (resp. outer) language. We refer to these delimiters as a *left boundary token* and a *right boundary token*. Older legacy languages, usually those analysed by hand-written lexers and parsers, tend to have more fuzzy boundaries where either one of these boundary tokens may be absent or confused for white-space.

[4] Note that the main characteristic distinguishing AmbigNodes from MultiTexts is that AmbigNodes have multiple lexical types, whereas MultiTexts have only one. Since all spellings of a MultiText have the same lexical type, the parser need not (in fact, must not) fork when it sees one. The parser only forks when the aggregate token it receives contains multiple lexical types that could cause the forked parsers to take different actions.

[5] Note that we do not reset the lexical state on a single spelling—multiple lexical type ambiguity, because the text of each alternative (and thus the lexer's path through its automaton) is the same, ending up in the same lexical state.

For example, in the description format used by Flex, the boundary between a regular expression and a C-based action in its lexical rules is simply a single character of white-space followed by an optional left curly brace.

One technique for identifying boundaries is to use a special program editor that understands the boundary tokens that divide the two languages (e.g. PHP embedded in XHTML) and enforces a high-level document/subdocument editing structure. The boundary tokens are fixed and, once inserted, cannot be edited or removed without removing the entire subdocument. The two languages can then be analysed independently.

Another technique is to use regular expression matching or a simple lexer to identify the boundary tokens in the document and use them as an indication to switch analysis services to or from the inner language. These services are usually limited to lexically based ones, such as syntax highlighting or imprecise indentation. More complex services based on syntax analysis cannot easily be used, since the regular expressions are not powerful enough to determine the boundary tokens accurately. In some cases, it might be possible to use a coarse parse such as Koppler's [21], but we have not explored that alternative.

Some newer embedded languages maintain lexically identifiable boundaries (e.g. PHP's starting token is `<?php` and its ending token is `?>`). Others contain boundaries that are only structurally or semantically detectable (e.g. Javascript's left boundary is `<script language=javascript>`).

4.2. Lexically embedded languages

Lexically embedded languages are those where the inner language has little or no structure and can be analysed by a finite automaton. To give an example, the typical lexical description for the Java language includes standard regular expressions for keywords, punctuation, and identifiers. The most complicated regular expressions are reserved for strings and comments. A string is a sequence of characters bounded by two double quote characters on either side. A comment is a sequence of characters bounded by a `/*` on the left and a `*/` on the right. Inside these boundary tokens, the traditional rules for Java lexing are suspended—no keywords, punctuation or identifiers are found within. Most description writers will "turn off" the normal Java lexical rules upon seeing the left boundary token, either by using lexer "condition" states,[6] or by storing the state in a global variable. When the right boundary token is detected, the state is changed back to the initial lexer state to begin detecting keywords again.

From the perspective of an embedded language, it is obvious that strings and comments form inner languages within the Java language that use completely different lexical rules. Using Blender, we can split these out into separate components and thereby clean up the Java lexical specification.

In the case of a string within a Java program, the two boundary tokens are identical, and lexically identifiable by a simple regular expression. However, aside from a rule that double quote may not appear unescaped inside a string, the double quotes that form the boundaries are not part of the string data. This is also true for comments—the boundary tokens identify the comment to the parser, but do not make up the comment data.

4.3. Syntactically embedded languages

Syntactically embedded languages are those where the inner language has its own grammatical structure and semantic rules. Compilers for syntactically embedded languages typically use a number of ad hoc techniques to process them. One common technique is to ignore the inner language, for example, as is done with SQL embedded in PHP. PHP analysis tools know nothing about the lexical or grammatical structure of SQL and, in fact, treat the SQL code as a string, performing no static checking of its correctness.[7] Similarly, in Flex, C code is passed along as unanalysed text by the Flex analyser, and subsequently packaged into a C program compiled by a conventional C compiler. The lack of static analysis leaves the programmer at risk for run-time errors that could have been caught at compile-time.

It is sometimes possible to analyse the embedded program. The embedded program can be segmented out and analysed as a whole program independently of the outer program. This technique will not work, however, if the embedded program refers to structures in the outer program or vice versa. In addition, the embedded program may not

[6] Condition states are explicitly declared automaton states in Flex-based lexical descriptions. They are often used to switch sub-languages.

[7] This incomplete and inappropriate lexing forces programmers to escape characters in their embedded SQL queries that would not be necessary when using SQL alone.

be in complete form. For example, it may be pieced together from distinct strings or syntactic parts by the execution of the outer program. Gould et al. [22] describe a static analysis of dynamically generated SQL queries embedded in Java programs that can identify some potential errors. In general, to analyse a particular embedding of one language in another, a special-purpose analysis is required, and often may not yet exist.

In the next section, we show how language descriptions are written in Blender, our combined lexer and parser generator tool.

5. Language descriptions for embedded languages

Lexical descriptions are written in a variant of the format used by Flex. The header contains a set of token declarations which are used to name the tokens that will be returned by the actions in this description. At the beginning of a rule is a regular expression (optionally preceded by a lexical condition state) that, when matched, creates a token of the desired type(s) and returns it to the parser.

Grammar descriptions are written in a variant of the Bison format. Each grammar consists of a header containing precedence and associativity declarations, followed by a set of grammar productions. One or more `%import-token` declarations are written to specify which lexical descriptions to load (one of which is specified as the default) in order to find tokens to use in this grammar. In addition to importing tokens, a grammar may import non-terminals from another grammar using the `%import-grammar` declaration. Grammar productions do not have user-described actions.[8] The only action of the run-time parser is to produce a parse tree/forest from the input. The language designer writes a tree-traversing semantic analysis phase to express any desired actions.

Imported (non-default) terminals and non-terminals are referred to in this paper as $\texttt{symbol}_{\textit{language}}$. An imported symbol causes an inner language to be embedded in the outer language.

5.1. Lexically embedded example

An example of a comment embedded in a Java program is:

```
/* Just a comment */
```

To embed the comment language in the outer Java grammar, the following rule might be added:

$$\textsc{comment} \quad \rightarrow \quad \texttt{SLASHSTAR}\ \texttt{COMMENTDATA}_{\textit{comment-lang}}\ \texttt{STARSLASH}$$

In Blender, boundary tokens for an inner language are specified with the outer language, so that the outer analyser can detect the boundaries. The data for the inner language is written in a different specification, named comment-lang in the example, which is imported into the Java grammar. In this simple case, the embedding is lexical. Comment boundary tokens are described by regular expressions that detect the tokens `/*` and `*/`. They are placed in the main Java lexical description (the one that describes keywords, identifiers and literals).

The comment data can be described by the following Flex lexical rule which matches all characters in the input including the carriage returns:

```
.|[\r\n]    { yymore(); break; }
```

However, this specification would read beyond the comment's right boundary token. Our solution, which is specialized to the peculiarities of a Flex-based lexer (and might be different in a different lexer generator), is to introduce a special keyword, `END_LEX`, into any lexical description that is intended to be embedded in an outer language. `END_LEX` will stand in for the regular expression that will detect the `*/`. Blender will automatically insert this regular expression based on the right boundary token following the `COMMENTDATA` terminal. For those familiar with Flex, the finalized description would look like:

[8] Because there are multiple parses with differing semantics, some of which may fail, it is tricky to get those actions right for GLR parsing, as discussed by McPeak [23].

```
%{  int comment_length;  %}
%token COMMENTDATA
%%
END_LEX    { yyless(comment_length); RETURN_TOKEN(COMMENTDATA); }
.|[\r\n]   { yymore(); comment_length = yyleng; break;          }
```

We must be careful to insert this new END_LEX rule before the other regular expression due to Flex's rule precedence property (lexemes matching multiple regular expressions are associated with the first one), or Flex will miss the right boundary token. Also, since the COMMENTDATA lexeme will only be returned once the right boundary token has been seen, its text would accidentally include the boundary token's characters. We use Flex's yyless() construct to push the right boundary token's characters back onto the input stream, making it available to be matched by a lexer for the outer language, and then return the COMMENTDATA lexeme.

This sort of lexical embedding enables one to reuse common language components in several programming languages. For example, even though Smalltalk and Java use different boundary tokens for strings (Java uses " and Smalltalk uses '), their strings have the same lexical content. Lexically embedding a language (such as this String language) enables a language designer to reuse lexical rules that may have been fairly complex to create, and might suffer from maintenance problems if they were duplicated.

5.1.1. *Syntactically embedded example*

Syntactic embedding is easier to perform because of the greater expressive power of context-free grammars. One simply uses non-terminals from the inner language in the outer language. The following is an example of a grammar for Flex lexical rules:

$$\begin{array}{lcl}
\text{RULE} & \rightarrow & \text{REGEXP\_ROOT}_{\mathit{regexp}}\ \texttt{WSPC}\ \text{CCODE} \\
\text{CCODE} & \rightarrow & \texttt{LBRACE}\ \text{COMPOUND\_STMT}_{C}\ \texttt{RBRACE}\ \texttt{NEWLINE} \\
 & | & \text{COMPOUND\_STMT}_{C}\ \texttt{NEWLINE}
\end{array}$$

A Flex rule consists of a regular expression followed by an optionally braced C compound statement. The regular expression is denoted by the REGEXP_ROOT non-terminal from the regexp grammar. The symbol WSPC denotes a white-space character. The compound statement is denoted by the COMPOUND_STMT from the C grammar.

We can now show one of the lexical ambiguities associated with legacy embedded languages. A left brace token is described by the character {, in both Flex *and* in C. A compound statement in C may or may not be bracketed by a set of curly braces. When a left brace is seen, it can belong either to the outer language for Flex or to the inner C language. Choosing the right language usually requires contextual information that is only available to a parser. Even the parser can only choose properly when presented with both choices: a Flex left brace token and a C left brace token. This is another example of a single lexeme with multiple lexical types; its resolution requires enhancements to both the lexer and parser generators, as well as enhancements to the parser.

In the next section, we show how embedded terminals and non-terminals are incorporated in our tools.

6. Blender lexer and parser table generation for embedded languages

When a Blender language description incorporates grammars for more than one language, the grammars are merged.[9] Each grammar symbol is tagged with its language name to ensure its uniqueness. Blender then builds an LALR(1) parse table, but omits LALR(1) conflict resolution. Instead, it chooses one action (arbitrarily) to put in the parse table, and puts the other action in a second so-called 'conflict' table to be available to the parser driver at run-time.

When a Blender language description incorporates more than one lexical description, all of them are combined. In each description, any condition states declared (including the default initial state) are tagged with their language name to ensure their uniqueness. All rules are then merged into a single list of rules. Each rule whose condition state was not explicitly declared is now declared to belong to the tagged initial condition state for its language. The default

[9] Since any context-free grammar can be parsed using GLR, merging causes no difficulty for the analyser.

lexical description's initial condition state is made the initial condition state of the combined specification. Rules that were declared to apply to all condition states (denoted by <*> at the beginning of the rule) are subsetted to apply only to those states declared for that particular language. This state-renaming scheme avoids any problems that the reordering of the rules may cause to the semantics of each language's lexical specification.

However, now each embedded lexical description's initial condition state is disconnected from the new initial state. It falls to the parser to set the lexer state before each token is lexed. For each parse state created by the GLR parser generator, the lexical descriptions to which the shift and reduce lookahead terminals belong are determined. This information is written into a table mapping a parse state to a set of lexical description IDs. At run-time, as the parser analyses a document described by an embedded language description, it uses this table to switch the lexer instance into the proper lexical state(s) before identifying a lookahead token. If there is more than one lexical state for a particular parse state, the parser has to tell the lexer instance to switch into *all* of the indicated lexical states. However, any parse state that has more than one lexical state causes the input stream to become ambiguous. The analysis of this ambiguity is described in the next section.

7. Lexing and parsing for embedded languages

Embedded languages add to the variety of input stream ambiguities described in Section 3 by enabling the lexer and parser to analyse the input simultaneously with a number of logical language descriptions. We make several more changes to the GLR algorithm to handle embedded languages and illustrate the complete XGLR algorithm in Figs. 4–6.

Before lexing the lookahead token for each parser in *SETUP-LOOKAHEADS()*, *SETUP-LEXER-STATES()* looks up the lexical language(s) associated with each of the parse states in the *active-parsers* list. If the language has changed, the state of the parser's lexer instance is reset to the initial lexical state of that language (via a look-up table generated by Blender). When there is more than one lexical language associated with the parse state, it implies that there is a lexical ambiguity on the boundary between the languages. This situation is handled in the same way as the other input stream ambiguities: for each ambiguity, a new parser is forked, and its lexer instance is set to the initial lexical state of that language. Each lexer instance will then read the same characters from the input stream but will interpret them differently because it is in a different lexical state. The ambiguous lookahead tokens that caused the parsers to fork are joined into an equivalence class for later use during parser merging (explained below). After shifting symbols, parser merging may cause multiple parsers incorrectly to share a lexer. One function of *SETUP-LEXER-STATES()* is to ensure that each parser's lexer instance is unique.

Next, if each parser has its own private lexer instance, and each lexer instance is in a different lexical state when reading the input stream, then the input streams may diverge at their token boundaries, with some streams producing fewer tokens, some producing more. This may cause each parser to be at a different position in the input stream than the others, which is a departure from the traditional GLR parsing algorithm in which all parsers are kept in sync shifting the same lookahead token during each major iteration. Unless we are careful, this could have serious repercussions on the ability of parsers to merge, as well as performance implications if one parser were forced to repeat the work of another.

To solve this problem, we observe that any two parsers that have forked will only be able to merge once their parse state, lexer state and lookahead tokens are equivalent.[10] For out-of-sync parsers, this can only happen when the input streams converge again after the language boundary ambiguities have been resolved. However, in the XGLR algorithm given in Fig. 1, only the *active-parsers* list is searched for mergeable parsers. If a parser *p* is more than one input token ahead of a parser *q*, *q* will no longer be in the *active-parsers* list when *p* will be ready to merge with it. If the merge fails to occur, parser *p* may end up repeating the work of parser *q*.

We introduce a new data structure, a map from a lookahead token to the parsers with that lookahead. The map is initialized to empty in *XGLR-PARSE()*, and is filled with each parser in the *active-parsers* list after each lookahead has been lexed in *PARSE-NEXT-SYMBOL()*. Any new parsers created during *DO-REDUCTIONS()* are added to the map. In *DO-REDUCTIONS()*, when a parser searches for another to merge with, instead of searching the *active-parsers* list, it searches the list of parsers in the range of the map associated with the parser's lookahead. In the case

[10] At the end of the input stream when there is no more input to lex, it is not important to check for lexer state equality.

```
XGLR-PARSE()
init active-parsers list to parse state 0
init parsers-ready-to-act list to empty
init parsers-at-end list to empty
init lookahead-to-parse-state map
      to empty
init lookahead-to-shiftable-parse-states
      map to empty
while active-parsers list $\neq \emptyset$
  PARSE-NEXT-SYMBOL(false)
copy parsers-at-end list to
  active-parsers list
clear parsers-at-end list
PARSE-NEXT-SYMBOL(true)
accept

SETUP-LEXER-STATES()
for each pair of parse states
      $p, q \in$ active-parsers list
  if lexer state of $lex_p =$
        lexer state of $lex_q$
      set $lex_p$ to copy $lex_q$
for each parse state
      $p \in$ active-parsers list
  let $langs$ = lexer-langs[$p$]
  if $|langs| > 1$
      let each of $q_1 .. q_n =$
            copy parse state $p$
      for each parse state $q_i \in q_1 .. q_n$
        if $langs_i \neq$ lexer language of $lex_p$
            set lex state of $lex_{q_i}$ to
                  init-state[$langs_i$]
      add $q_i$ to active-parsers list
  else if $langs_0 \neq$ lexer language of $lex_p$
      set lexer state of $lex_p$ to
        init-state[$langs_0$]

PARSE-NEXT-SYMBOL(bool finish-up?)
SETUP-LEXER-STATES()
SETUP-LOOKAHEADS()
if not finish-up?
  FILTER-FINISHED-PARSERS()
  if active-parsers list is empty? return
init shiftable-parse-states list to empty
copy active-parsers list to
      parsers-ready-to-act list
while parsers-ready-to-act list $\neq \emptyset$
  remove parse state $p$ from list
  DO-ACTIONS(p)
SHIFT-A-SYMBOL()

SETUP-LOOKAHEADS()
for each parse state
      $p \in$ active-parsers list
  set $lookahead_p$ to
        first token lexed by $lex_p$
  add <offset of $lookahead_p \times lookahead_p$>
        to offset-to-lookaheads map
  if $lookahead_p$ is ambiguous
    let each of $q_1 .. q_n =$
          copy parse state $p$
    for each parse state $q \in q_1 .. q_n$
      for each alternative $a$
            from $lookahead_p$
          set $lookahead_q$ to $a$
          add $lookahead_q$ to
            equivalence class for $a$
      add $q$ to active-parsers list
for each parse state $p \in$
      active-parsers list
  add <$lookahead_p \times p$>
      to lookahead-to-parse-state map
```

Fig. 4. A non-incremental version of the fully modified XGLR parsing algorithm. Continued in Fig. 5.

where all parsers remained synchronized at the same lookahead terminal, this degenerates to the old behavior. But for parsers that get out of sync, this enables the late parser to merge with a parser that has already moved past that terminal, thereby avoiding repeated work.

Parser merging in XGLR contains one more potential pitfall that must be addressed in the implementation of the algorithm. The criteria for parser merging compares two lookahead tokens for equivalence. Usually, equivalence is an equality test, but for tokens that caused the parsers to fork, the algorithm tests each token for membership in the same equivalence class (assigned in *SETUP-LEXER-LOOKAHEADS()*). We use this equivalence to properly merge the parse trees formed by the reduction of each parser in *DO-REDUCTIONS*. Normally, both parsers involved in successful merge would share a p^- during the reduce action. Parsers that were created by forking at an input stream ambiguity do not, because the parser fork occurred *before* the shift of the equivalent tokens, not after. Even though all the conditions for parser merging are met, the implementation of the algorithm must ensure an equivalence between all possible parsers p^- that could shift any of the lookahead tokens in the equivalence class. We use a map to record

```
DO-ACTIONS(parse state p)
look up actions[p×lookahead_p]
for each action
  if action is SHIFT to state x
    add <p, x> to shiftable-parse-states
    add <lookahead_p×p> to
        lookahead-to-shiftable-parse-states
        map
  if action is REDUCE by rule y
    if rule y is accepting reduction
      if lookahead_p is end of input
        return
      if no parsers ready to act or
          shift or at end of input
        invoke error recovery
      return
    DO-REDUCTIONS(p, rule y)
    if no parsers ready to act or shift
      invoke error recovery and return
  if action is ERROR and no parsers
      ready to act or shift or
      at end of input
    invoke error recovery and return
```

```
FILTER-FINISHED-PARSERS()
for each parse state
    p ∈ active-parsers list
  if lookahead_p = end of input?
    remove p from active-parsers list
    add p to parsers-at-end list

SHIFT-A-SYMBOL()
clear active-parsers list
for each <p, x> ∈ shiftable-parse-states
  if p is not an accepting parser
    if parse state x ∈ active-parsers list
      push x on stack p
    else
      create new parse state x
        with lookahead_p and copy of lex_p
      push x on stack p
      add x to active-parsers list
```

Fig. 5. The second portion of a non-incremental version of the fully modified XGLR parsing algorithm. Continued in Fig. 6.

all parsers that can immediately shift a particular lookahead token (the *lookahead-to-shiftable-parse-states* map). The set of all *equivalent* parsers p^- is the range of the *lookahead-to-shiftable-parse-states* map with the domain being all lookahead tokens in the equivalence class of token $lookahead_p$.

Since any parser may be out of sync with other parsers, the end of the input stream may be reached by some parsers before others. These parsers are stored separately in the *parsers-at-end* list because it simplifies the control flow logic of the algorithm to have all parsers that are ready to accept the input accept in the same call to *PARSE-NEXT-SYMBOL()*. We add a Boolean argument *finish-up?* to *PARSE-NEXT-SYMBOL()* to indicate this final invocation and we call the *FILTER-FINISHED-PARSERS()* function to move the finished parsers to the *parsers-at-end* list.

XGLR uses more memory in practice than GLR. In addition to the two maps above, which cannot be pruned during the parse (reductions may require looking up any already parsed token in the map), the lack of synchronization of parsers requires each parser to hold extra state that is global in GLR. This memory requirement grows linearly as the number of parsers, or equivalently, as the number of dynamic ambiguities in the program discovered during the parse.

8. Implementation status

This algorithm has been implemented in the Harmonia analysis framework and is the parser used for analyses of spoken ambiguities in the SPEED spoken program editor [24]. The SPEED prototype parses a language called Spoken Java, a semantically identical variant of Java that is easier to say out loud. The Spoken Java design was based on a study of how developers naturally verbalize Java code [25]. The syntax of Spoken Java is considerably more ambiguous than Java (which has no syntactic ambiguities that survive parsing), but in fact was derived directly from the Java grammar. Most of the structural ambiguities arise from optional punctuation and reordered phrase structure, while the lexical ambiguities originate in alternate spellings for the same lexeme (e.g. saver, savor), multiple interpretations of the same word (e.g. equals can be equality or assignment), or a combination (e.g. to, 2, too and two).

```
DO-REDUCTIONS(parse state p, rule y)
for each equivalent parse state $p^-$ below RHS(rule $y$)
      on a stack for parse state $p$
  let $q$ = GOTO state for actions[$p^- \times$ LHS(rule $y$)]
  if parse state $q \in$ lookahead-to-parse-state[lookahead$_p$]
        and lookahead$_q \cong$ lookahead$_p$
        and (lookahead$_p$ is end of input or
           lexer state of lex$_q$ = lexer state of lex$_p$)
    if $p^-$ is not immediately below $q$ on stack for parse state $q$
      push $q$ on stack $p^-$
      for each parse state $r$ such that $r \in$ active-parsers list
            and $r \notin$ parsers-ready-to-act list
        DO-LIMITED-REDUCTIONS(r)
  else
    create new parse state $q$ with lookahead$_p$ and copy of lex$_p$
    push $q$ on stack $p^-$
    add $q$ to active-parsers list
    add $q$ to parsers-ready-to-act list
    add <lookahead$_q \times q$> to lookahead-to-parse-state map

DO-LIMITED-REDUCTIONS(parse state r)
look up actions[$r \times$ lookahead$_r$]
for each REDUCE by rule y action
  if rule $y$ is not accepting reduction
    DO-REDUCTIONS(r, rule y)
```

Fig. 6. The remainder of a non-incremental version of the fully modified XGLR parsing algorithm.

We also use this parser algorithm to handle language descriptions of Java dialects. For example, Titanium, a parallel programming language [26], is a superset of Java 1.4. We describe Titanium using two grammars and two lexical descriptions. The "outer" grammar and lexical description is for Java 1.4; the "inner" language consists of extra (and altered) grammar productions as well as new lexical rules for Titanium's new keywords.

Performance measurements of the parser are dependent on the nature of the grammar used and the input provided. In Spoken Java, punctuation is optional. Consequently, any number of implicit punctuation symbols (e.g. comma, period, left paren, right paren, quote) must be considered between any two identifiers. This blows up the number of ambiguities during parsing to astronomical levels for an entire program. In contrast, the possible lexical ambiguities in the specification rarely increase the ambiguity of the language dramatically, since they typically correspond to only a few structural ambiguities. In practice, the user interface limits the input between incremental analyses to 30 or 40 words. Limiting the input in this way makes the parse time tractable, even though it results in a large number (tens) of ambiguous parses. When filtered by semantic analysis, the number of semantically valid parses drops to a small number, usually one.

9. Related work

Yacc [27], Bison [28,2], and their derivatives, which are widely used, make the generation of C-, C++- and Java-based parsers for LALR(1) grammars relatively simple. These parsers are often paired with a lexical generator (Lex [29] for Yacc, Flex [1] for Bison, and others) to generate token data structures as input to the parser. Improvements on this fairly stable base include GLR parser generation [19,9], found in ASF+SDF [7], and more recently in Elkhound [23], D Parser [30], and Bison 1.50. Incremental GLR parsing was first described and implemented by Wagner and Graham [15,17,18] and has been improved in the last few years by our Harmonia project.

There has been considerable work in the ASF+SDF research project [7] on the analysis of legacy languages, as well as language dialects. One central aspect of this work increases the power of the analyses by moving the lexer's work into the parser and simply parsing character by character. Originally described as scannerless parsing [31,32], this idea has been adapted successfully by Visser to GLR parsing [20,33]. Visser merges the lexical description into the grammar and eliminates the need for a special-purpose analysis for ambiguous lexemes. Some of the messiness of Flex interaction that we describe for embedded languages can be avoided. In making this change, however, some desirable attributes of a separate regular-expression-based lexer, such as longest match and order-based matching, are lost, requiring alternate, more complex, implementations based on disambiguation filters that are programmed into the grammar [34].

In the Harmonia project, a variant of the Flex lexer is used—historically because of the ability to re-use lexer specifications for existing languages, but more importantly because a separate incremental lexer limits the effects that an edit has on re-analysis. In Harmonia's interactive setting, the maintenance of a persistent parse tree and the application of user edits to pre-existing tokens in the parse tree contribute heavily to its interactive performance. For example, a change to the spelling of an identifier may often result in no change to the lexical type of the token. Thus, the change can be completely hidden by the lexer, preventing the parser from doing any work to re-analyse the token. In addition, the incremental lexer affords a uniform interface of tokens to the parser, even when the lexer's own input stream consists of a variety of characters, normal tokens and ambiguous tokens created by a variety of input modes.

In principle, both incrementality and the extensions described in this paper could be added to scannerless GLR parsers. However, as always, the devil is in the details. In an incremental setting, parse tree nodes have significant size because they contain data to maintain incremental state. If the number of nodes increases, even by a linear factor, performance can be affected. More significantly, incremental performance is based on the fact that the potentially changed region of the tree can be both determined and limited prior to parsing by the set of changed tokens reported from the lexer. For example, only a trivial amount of reparsing is needed if the spelling of an identifier changes, since the change does not cross a node boundary. Although we have not done a detailed analysis, our intuition is that, without a lexer, the potentially changed regions that would end up being re-analysed for each change would be considerably larger.

Aycock and Horspool [35] propose an ambiguity-representing data structure similar to our AmbigNode. They discuss lexing tokens with multiple lexical types, but do not discuss how to handle other lexical ambiguities. Their scheme also requires that all token streams be synced up at all times (inserting null tokens to pad out the varying token boundaries). Our mechanism is able to handle overlapping token boundaries fluidly in the alternate character streams without extraneous null tokens.

CodeProcessor [36] has been used to write language descriptions for lexically embedded languages. CodeProcessor also maintains persistent document boundaries between embedded documents. Gould et al. [22] describe a static analysis of potentially dynamically generated SQL query strings embedded in Java programs. Specialized fragment analyses are likely to be required to analyse this kind of embedded language semantically.

Synytskyy et al. [8] provide a cogent discussion of the difficulties that arise with embedded languages, and describe the use of island grammars to parse multi-language documents. They also summarize related research into the use of coarse parsing techniques for that purpose. Unlike the approach that we have taken, they handle some of the boundary difficulties, such as those concerning white-space and comments, by a lexical preprocessor prior to parsing.

10. Future work

Blender, our lexer and parser generator, is built using language descriptions for its Flex and Bison variant input files. Flex, in particular, is made up of three languages: the Flex file format, regular expressions, and C. The three languages combine to form several kinds of interesting ambiguities. First, white-space forms the boundary between regular expressions and C code in each Flex rule. In many parser frameworks, white-space is either filtered by the lexer or discarded by the parser, but certainly not included in the parse tables. However, in this case, white-space must be considered by the parser in order to properly switch among lexical language descriptions at run-time. Second, white-space takes on additional significance in Flex since rules are required to be terminated by carriage returns, even though carriage returns *are* allowed as general white-space characters within rules. Third, it is possible to have non-obvious shift–shift conflicts between multiple interpretations of the same character sequence, because they are interpreted in different lexical descriptions. For example, the following is the real grammar production for Flex rules (first described in Section 5.1.1):

$$\begin{array}{lll} \text{RULE} & \rightarrow & \text{STATE}?\ \text{REGEXP\_ROOT}_{regexp}\ \texttt{WSPC}\ \text{CCODE} \\ \text{STATE} & \rightarrow & <\texttt{ID}> \end{array}$$

The optional STATE can begin with a $<$ token. But $<_{regexp}$ is a valid regular expression token as well. Since the STATE is optional, the $<$ character may be lexed as two separate tokens, leading to a lexical ambiguity. However, the Flex manual states that if a Flex rule begins with a $<$, it must be the beginning of an optional STATE, not a regular expression. If the input is not actually a proper state, it is an error, not a regular expression. We are currently upgrading our language analysis technology and the grammar transforms used in Blender to handle these three kinds of ambiguities.

New techniques being developed in our research group for batch GLR parser error recovery do not yet take into account the ambiguities discussed in this paper. Extension of the work above to incorporate batch error recovery is ongoing. Incremental error recovery is change-based and has already been extended.

Automated semantic disambiguation of both homophones and syntactic ambiguities will require integration with name resolution and type checking. In addition, to handle ambiguities that arise in an interactive setting (e.g. via edits in a program editor), semantic information must be persistent and incrementally updateable. Such persistence will enable analysis of edits to a portion of the program to use semantic information from surrounding code to help disambiguation (for example, by providing a list of all legal visible bindings at the edit location). A MultiText identifier token appearing in a variable-use position can be disambiguated if one of its alternatives matches a definition that is in scope and has the right static type. Our solutions to these problems are still in progress.

11. Conclusion

In this paper, we have described tools and analyses to handle embedded languages, programming by voice, and support for legacy languages—situations that are poorly supported by contemporary language analysis tools. We classified the lexical ambiguities caused by these situations into four types, and developed both a lexer and parser generator and a set of lexing and parsing analysis enhancements to address each one. We then extended these methods to embedded languages. Our work gives language designers several more tools with which to describe and analyse more easily the complex programming languages of yesterday, today, and tomorrow.

References

[1] V. Paxson, Flex—fast lexical analyzer generator, Free Software Foundation, 1988.
[2] C. Donnelly, R. Stallman, Bison: The Yacc-compatible parser generator, Free Software Foundation, December 1990.
[3] A. Begel, Spoken language support for software development, Ph.D. Dissertation, University of California, Berkeley, Technical Report EECS-2006-8, December 2005 (in press).
[4] S.I. Feldman, Implementation of a portable Fortran 77 compiler using modern tools, in: SIGPLAN '79: Proceedings of the 1979 SIGPLAN Symposium on Compiler Construction, ACM Press, New York, NY, USA, 1979, pp. 98–106.
[5] T.J. Parr, R.W. Quong, ANTLR: A predicated-LL(*k*) parser generator, Software—Practice and Experience 25 (7) (1995) 789–810.
[6] J. Aycock, The design and implementation of SPARK, a toolkit for implementing domain-specific languages, Journal of Computing and Information Technology 10 (2002) 55–66.
[7] P. Klint, A meta-environment for generating programming environments, ACM Transactions of Software Engineering and Methodology 2 (2) (1993) 176–201.
[8] N. Synytskyy, J.R. Cordy, T.R. Dean, Robust multilingual parsing using island grammars, in: CASCON '03: Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative Research, IBM Press, 2003, pp. 266–278.
[9] M. Tomita, Efficient Parsing for Natural Language—A Fast Algorithm for Practical Systems, in: Int. Series in Engineering and Computer Science, Kluwer, Hingham, MA, 1986.
[10] M. Boshernitsan, Harmonia: A flexible framework for constructing interactive language-based programming tools, Tech. Rep. UCB/CSD-01-1149, Computer Science Division–EECS, University of California, Berkeley, M.S. Report, 2001.
[11] Harmonia Project Web Site, http://harmonia.cs.berkeley.edu.
[12] XEmacs: The next generation of Emacs, http://www.xemacs.org.
[13] Eclipse, http://www.eclipse.org.
[14] T.A. Wagner, S.L. Graham, Efficient self-versioning documents, in: Proceedings of COMPCON '97, San Jose, CA, 1997, pp. 62–67.
[15] T.A. Wagner, Practical algorithms for incremental software development environments, Ph.D. Dissertation, University of California, Berkeley, Technical Report UCB/CSD-97-946, March 11 1998.

[16] T.A. Wagner, S.L. Graham, General incremental lexical analysis, unpublished, 1997.

[17] T.A. Wagner, S.L. Graham, Incremental analysis of real programming languages, in: Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation, 1997, pp. 31–43.

[18] T.A. Wagner, S.L. Graham, Efficient and flexible incremental parsing, ACM Transactions on Programming Languages and Systems 20 (5) (1998) 980–1013.

[19] J. Rekers, Parser generation for interactive environments, Ph.D. Dissertation, University of Amsterdam, 1992.

[20] E. Visser, Syntax definition for language prototyping, Ph.D. Dissertation, University of Amsterdam, 1997.

[21] R. Koppler, A systematic approach to fuzzy parsing, Software—Practice and Experience 27 (6) (1997) 637–649.

[22] C. Gould, Z. Su, P. Devanbu, Static checking of dynamically generated queries in database applications, in: Proceedings of the 26th International Conference of Software Engineering, IEEE Press, Edinburgh, Scotland, UK, 2004, pp. 645–654.

[23] S. McPeak, Elkhound: A fast, practical GLR parser generator, Lecture Notes in Computer Science 2985 (2004) 73–88.

[24] A. Begel, Programming by voice: A domain-specific application of speech recognition, in: AVIOS Speech Technology Symposium–SpeechTek West, 2005.

[25] A. Begel, S.L. Graham, Spoken programs, in: IEEE Symposium on Visual Languages and Human-Centric Computing, 2005.

[26] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, A. Aiken, Titanium: A high-performance Java dialect, Concurrency: Practice and Experience 10 (1998) 825–836.

[27] S.C. Johnson, Yacc: Yet another compiler compiler, in: UNIX Programmer's Manual, vol. 2, Holt, Rinehart, and Winston, New York, NY, USA, 1979, pp. 353–387. AT&T Bell Laboratories Technical Report July 31, 1978.

[28] R.P. Corbett, Static semantics and compiler error recovery, Ph.D. Dissertation, Computer Science Division—EECS, University of California, Berkeley, CA, June 1985.

[29] M.E. Lesk, E. Schmidt, Lex—A lexical analyzer generator, in: UNIX Programmer's Manual, vol. 2, Holt, Rinehart, and Winston, New York, NY, USA, 1979, pp. 388–400. AT&T Bell Laboratories Technical Report in 1975.

[30] J. Plevyak, D Parser Homepage, http://dparser.sourceforge.net.

[31] D.J. Salomon, G.V. Cormack, Corrections to the paper: Scannerless NSLR(1) Parsing of Programming Languages, ACM SIGPLAN Notices 24 (11) (1989) 80–83.

[32] D.J. Salomon, G.V. Cormack, Scannerless NSLR(1) parsing of programming languages, in: Proceedings of the SIGPLAN'89 Conference on Programming Language Design and Implementation, 1989, pp. 170–178.

[33] E. Visser, Scannerless generalized-LR parsing, Tech. Rep. P9707, Programming Research Group, University of Amsterdam, 1997.

[34] M.G.J. van den Brand, J. Scheerder, J.J. Vinju, E. Visser, Disambiguation filters for scannerless generalized LR parsers, in: Compiler Construction, CC '02, in: Lecture Notes in Computer Science, vol. 2304, 2002, pp. 143–158.

[35] J. Aycock, R.N. Horspool, Schrödinger's token, Software Practice and Experience 31 (8) (2001) 803–814.

[36] M.L. Van De Vanter, M. Boshernitsan, Displaying and editing source code in software engineering environments, in: Proceedings of Second International Symposium on Constructing Software Engineering Tools, CoSET'2000, Limerick, Ireland, 2000, pp. 39–48.