Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 110 (2004) 75–96

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Language Analysis and Tools for Ambiguous Input Streams[1]

Andrew Begel, Susan L. Graham[2]

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

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 analyze 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.

Keywords: GLR, 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 [10] and Bison [3], 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

[1] This work was supported in part by NSF Grant CCR-0098314 and by an IBM Eclipse Innovation Grant.

[2] Email: {abegel, graham}@cs.berkeley.edu

doi:10.1016/j.entcs.2004.06.009

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:

(i) improved methods for syntax analysis that handle these kinds of ambiguities

(ii) a new combined lexer and parser generator and further parser enhancements that facilitate the description and analysis of embedded languages.

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 accurately process speech. Spoken input, however, contains many lexical ambiguities, such as homophones,[3] 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, natural language disambiguation rules do not apply. 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 can not distinguish between them; only the parser and static semantics have enough context to choose. Fortran's optional whitespace 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 – either the parser must help the lexer, or the lexer should pass all possible tokenizations to the parser.

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 hand-written analysis. The lack of modularity in Flex and Bison descriptions of embedded languages makes independent maintenance of each component language unwieldy and combined analysis awkward.

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

The methods described in this paper handle four kinds of input streams, three of which are ambiguous; our solutions are summarized in Section 3. Combinations of these ambiguities arise in different forms of embedded languages. The handling of this fifth kind of input stream is presented in Sections 4 to 7. Some of these ambiguities have also been addressed in related work, which is summarized in Section 8.

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 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.

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 analyzing the outer and inner (and any other nested) 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 analyzing 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.

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. 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 have strengthened the language analysis capabilities of our Harmonia analysis framework [2,5] 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 compose multiple language descriptions will enable us to create a voice-based command language for editing and navigating source code. This new input language will combine a command language written in a structured, natural-language style 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 enhanced GLR 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 analyzer, parse tables and syntax tree node C++ classes for representing syntax tree nodes in the parse tree. It enables language designers to easily describe many classes of embedded languages (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 [25] and Eclipse [4] 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 [21]. 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 unanalyzed. If the input was spoken, the words from the voice recognizer are turned into a new unanalyzed terminal node and added to the appropriate location in the parse tree. These changes make up the most recently edited version (a.k.a the last edited version). This version of the tree and the pre-edited version are used by an incremental lexer and parser to analyze and reconcile the changes in the tree.

Harmonia employs incremental versions of lexing and sentential-form GLR parsing [20,22,23,24] 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 nonterminals. 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 [12] and by Visser [19]. In GLR, conflict resolution is deferred to runtime, and all actions are placed in the table. When more than one action per lookup 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 lookup, 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 Figure 1.[4] 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 Figure 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 - 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

[4] The addition of incrementality is not essential to 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
```

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

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.

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.[5] The characters of the starting token are fed to the Flex-based lexical analyzer one at a time until a regular expression is matched. The action associated with the regular expression creates a single, 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 would be a token that is already in the edited version of the syntax tree. (The details of the 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

[5] Lookback is computed as a function of the number of lookahead characters used by the batch lexer when the token is lexed. [20]

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.

In Figure 2 is a restatement of the *PARSE-NEXT-SYMBOL()* function that has been modified with the changes above. 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 lookup must reference the associated lookahead – 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.

```
PARSE-NEXT-SYMBOL()
for each parse state $p \in$ active-parsers list
    $\mathbf{lex}_p$ one $lookahead_p$ token
    if $lookahead_p$ is ambiguous
        let $q_1 .. q_n$ = copy parse state $p$
        for each parse state $q \in q_1 .. q_n$
            assign one alternative from $lookahead_p$ to $q$
            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()
```

Figure 2. Part of the GLR parsing algorithm modified to support ambiguous lexemes.

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 contains the original identifier along with possible alternate spellings. Further analysis might be able to automatically choose the proper alternative 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. 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.[6] 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 analyzed by hand-written lexers and parsers, tend to have more fuzzy boundaries where

[6] 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.

either one of these boundary tokens may be absent or confused for whitespace. 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 whitespace 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, can not be edited or removed without removing the entire subdocument. The two languages can then be analyzed 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.

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 analyzed 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,[7] 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.

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

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 Harmonia, we can split these out into modular components and clean up the Java lexical specification in the process.

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.[8] This lack of analysis leaves the programmer at risk for runtime parse errors that should have been caught at compile-time. Similarly, in Flex, C code is passed along as text by the Flex analyzer, and subsequently packaged into a C program compiled by a conventional C compiler.

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

5 Blender 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.

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

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. To support descriptional modularity, one or more `%import-token` declarations are written to specify which lexical descriptions to load (of which one is specified as the default) in order to find tokens to use in this grammar. In addition to importing tokens, a grammar may import nonterminals from another grammar using the `%import-grammar` declaration. Grammar productions have no associated actions. The only action of the runtime 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 nonterminals are referred to in this paper as $\texttt{symbol}_{\mathit{language}}$. An imported symbol causes an inner language to be embedded in the outer language.

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:

$$\textsc{comment} \longrightarrow \texttt{SLASHSTAR}\ \texttt{COMMENTDATA}_{\mathit{comment\text{-}lang}}\ \texttt{STARSLASH}$$

In Blender, boundary tokens for an inner language are specified with the outer language, so that the outer analyzer can detect the boundaries. The data for the inner language is written in a different specification, named comment-lang, 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:

```
%{  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. In addition, since the `COMMENTDATA` lexeme would 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 (and thus be made 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.

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

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

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 nonterminal 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 nonterminals 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. Parser generation proceeds normally as for a GLR parser generator (*i.e.* LALR(1) with GLR conflict resolution).

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 together 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 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 runtime, as the parser analyzes 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

[9] GLR is closed under union.

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 simultaneously analyze the input with a number of logical language descriptions. We make two more changes to the GLR algorithm to handle embedded languages and illustrate the complete algorithm in Figures 3 and 4.

Before lexing the lookahead token for each parser in *PARSE-NEXT-SYMBOL()*, the lexical language(s) associated with each of the parse states is looked up 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 lookup 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: a new lexer instance is created for each lexical language (and set to the initial lexical state of that language), and a separate parser is created for each lexer instance. Each forked 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.

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 the same. 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 GLR algorithm given in Figures 1 and 2, only the *active-parsers* list is searched for mergeable parsers. If a parser *p* is more than one input token ahead of another 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

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

PARSE-NEXT-SYMBOL()
SETUP-LEXER-STATES()
for each parse state
    $p \in$ active-parsers list
  lex$_p$ one lookahead$_p$ token
  if lookahead$_p$ is ambiguous
    let $q_1$ .. $q_n$ = copy parse state $p$
    for each parse state $q \in q_1$ .. $q_n$
      assign one alternative from
          lookahead$_p$ to $q$
      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
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-LEXER-STATES()
for each parse state
    $p \in$ active-parsers list
  let langs = lexer-langs[$p$]
  if |langs| > 1
    let $q_1$ .. $q_n$ = 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 lex state of lex$_p$ to
      init-state[langs$_0$]

DO-ACTIONS(parse state p)
look up actions[$p \times$ lookahead$_p$]
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
```

Figure 3. A non-incremental version of the fully modified GLR parsing algorithm. Continued in Figure 4.

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 *GLR-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

```
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 $q \in$ lookahead-to-parse-state[$lookahead_p$]
         and $lookahead_q = lookahead_p$ and $lex_q = lex_p$
      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 with $lex_p$ and $lookahead_p$
      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)


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 with $lex_p$ and $lookahead_p$
      push x on stack p
      add x to active-parsers list
```

Figure 4. The remainder of a non-incremental version of the fully modified GLR parsing algorithm.

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 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.

If the entries in the map were never removed, the map would grow as the number of parsers created during that parse. In an incremental setting like ours, the number of parsers is bounded by the number of tokens examined during the parser (which is bounded by the size of the edited region of text). To be more memory efficient, entries are removed from the map when their lookaheads are no longer accessible. The *active-parsers* list is sorted by the offset in the input stream of each parser's lookahead terminal. Informed by this sorted list, as soon as the last parser shifts past a particular lookahead terminal, that lookahead (and its range of parsers) is removed from the map. Thus, the memory overhead of the map can be bounded by the dynamic separation of the parsers, rather than the entire size of the edited region.

8 Related Work

Yacc [6], Bison [3], and their derivatives, introduced in the late 1970s and 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 [8] for Yacc, Flex [10] for Bison, and others) to generate token data structures as input to the parser. Improvements on this fairly stable base include GLR parser generation [12,15], found in ASF+SDF [7], and more recently in Elkhound [9], D Parser [11], and Bison 1.50. Incremental GLR parsing was first described and implemented by Wagner and Graham [20,23,24] 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 [13,14], this idea has been adapted successfully by Visser to GLR parsing [18,19]. 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 [17].

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 of an edit upon re-analysis. In Harmonia's interactive setting, the maintenance of a persistent parse tree and the application of user edits to preexisting tokens in the parse tree contribute heavily to its interactive performance. 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-analyzed for each change would be considerably larger.

Aycock and Horspool [1] 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 each alternate token stream be synced up at all times to one another (inserting null tokens to pad out the varying token boundaries). Our mechanism is able to fluidly handle overlapping token boundaries in the alternate character streams without extraneous null tokens.

CodeProcessor [16] has been used to write language descriptions for lexically embedded languages. CodeProcessor also maintains persistent document boundaries between embedded documents.

9 Future Work and Conclusion

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 is more easily extended.)

Semantic analysis of embedded languages remains an interesting challenge. How can semantic analyses for independently-defined languages be composed as modularly as lexical and syntactic descriptions? The interaction of two language semantics on their document/subdocument boundaries must be defined by the language designer. The composition algorithm becomes complicated if semantic entities from the inner language can be seen or affected by the outer language (and vice versa).

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.

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 more easily describe and analyze the complex programming languages of today, and of tomorrow.

References

[1] John Aycock and R. Nigel Horspool. Schrödinger's token. *Software Practice and Experience*, 31(8):803–814, July 2001.

[2] M. Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Technical Report UCB/CSD-01-1149, Computer Science Division – EECS, University of California, Berkeley, 2001. M.S. Report.

[3] Charles Donnelly and Richard Stallman. *Bison: the Yacc-compatible parser generator*. Free Software Foundation, December 1990.

[4] Eclipse. http://www.eclipse.org.

[5] Harmonia Project Web Site. http://harmonia.cs.berkeley.edu.

[6] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report July 31, 1978.

[7] Paul Klint. A meta-environment for generating programming environments. *ACM Transactions of Software Engineering and Methodology*, 2(2):176–201, March 1993.

[8] Michael E. Lesk and Eric Schmidt. Lex — A lexical analyzer generator. In *UNIX Programmer's Manual*, volume 2, pages 388–400. Holt, Rinehart, and Winston, New York, NY, USA, 1979. AT&T Bell Laboratories Technical Report in 1975.

[9] Scott McPeak. Elkhound: A fast, practical GLR parser generator. In *Compiler Construction*, 2004.

[10] Vern Paxson. Flex – fast lexical analyzer generator. Free Software Foundation, 1988.

[11] John Plevyak. D Parser Homepage. http://dparser.sourceforge.net.

[12] Jan Rekers. *Parser Generation for Interactive Environments*. Ph.d. dissertation, University of Amsterdam, 1992.

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

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

[15] Masaru Tomita. *Efficient Parsing for Natural Language — A Fast Algorithm for Practical Systems*. Int. Series in Engineering and Computer Science. Kluwer, Hingham, MA, 1986.

[16] Michael L. Van De Vanter and Marat Boshernitsan. Displaying and editing source code in software engineering environments. In *Proceedings of Second International Symposium on Constructing Software Engineering Tools (CoSET'2000)*, pages 39–48, Limerick, Ireland, 2000.

[17] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation filters for scannerless generalized LR parsers. In *Compiler Construction (CC '02)*, pages 143–158, 2002. In Lecture Notes in Computer Science 2304.

[18] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, 1997.

[19] Eelco Visser. *Syntax Definition for Language Prototyping*. Ph.d. dissertation, University of Amsterdam, 1997.

[20] Tim A. Wagner. *Practical Algorithms for Incremental Software Development Environments*. Ph.d. dissertation, University of California, Berkeley, March 11, 1998. Technical Report UCB/CSD-97-946.

[21] Tim A. Wagner and Susan L. Graham. Efficient self-versioning documents. In *Proceedings of COMPCON '97*, San Jose, CA, 1997.

[22] Tim A. Wagner and Susan L. Graham. General incremental lexical analysis, 1997. Unpublished.

[23] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–43, 1997.

[24] Tim A. Wagner and Susan L. Graham. Efficient and flexible incremental parsing. *ACM Transactions on Programming Languages and Systems*, 20(5):980–1013, September 1998.

[25] XEmacs: The next generation of Emacs. http://www.xemacs.org.