Context-Aware Scanning for Parsing Extensible Languages

Eric R. Van Wyk August C. Schwerdfeger

Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN, USA
evw@cs.umn.edu, schwerdf@cs.umn.edu

Abstract

This paper introduces new parsing and context-aware scanning algorithms in which the scanner uses contextual information to disambiguate lexical syntax. The parser uses a slightly modified LR-style algorithm that passes to the scanner the set of valid symbols that the scanner may return at that point in parsing. This set is those terminals whose entries in the parse table for the current parse state are shift, reduce, or accept, but not error. The scanner then only returns tokens in this set. An analysis is given that can statically verify that the scanner will never return more than one token for a single input. Context-aware scanning is especially useful when parsing and scanning extensible languages in which domain specific languages can be embedded. It has been used in extensible versions of Java 1.4 and ANSI C. We illustrate this approach with a declarative specification of a subset of Java and extensions that embed SQL queries and Boolean expression tables into Java.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory:Syntax

General Terms Languages

Keywords extensible languages, context-aware scanning

1. Introduction

In previous work [15] we have argued that one of the fundamental challenges to developing robust, reliable software in a timely manner is the semantic gap between a programmer's high-level, domain-specific knowledge of a problem's solution and the comparatively low-level language in which the solution to the problem must be encoded. When general purpose languages, such as Java or C, can be extended with domain-specific language features, this gap can be narrowed. Our interests have been in the specification and implementation of extensible languages and composable language extensions, which add language features that define new syntax (notations) so that domain-specific concepts can be written in a natural way, and define new semantic analysis so that domain-specific analysis and error-checking can be performed. Our previous work has focused on the specification and composition of the semantic aspects of such language extensions [14]. This paper addresses the specification, implementation, and composition of the syntactic aspects of extensible languages.

GPCE'07, October 1–3, 2007, Salzburg, Austria.

In this paper we present new parsing and scanning algorithms in which the parsing context is used by the scanner in determining which terminal symbol among the possible matches it should return to the parser. The parsing algorithm is a slight modification of the LR algorithm. When calling the scanner for the next token, it passes thereto the set of terminals that, in the current parse state, may be part of a valid phrase in the language. This set is called the *valid lookahead* set, and for any state in the LR parsing table it consists of all terminals whose entry for that state in the table is *shift*, *reduce*, or *accept*. Terminals whose entry for the state are *error — i.e.*, those that are syntactically invalid at the parser's current position in the input — are not in the valid lookahead set. The scanning algorithm is a modification of traditional deterministic-finite-automaton-based algorithms that makes use of the valid lookahead set. As an example, consider parsing the Java 1.5 type expression "`List<List<Integer>>`". After recognizing `Integer` as a type, the parser is in a state in which the greater-than symbol `>` is in the valid lookahead set, but the right bit shift operator `>>` is not. This scanning algorithm subordinates the disambiguation principle of maximal munch to the principle of disambiguation by context; it will return a short valid match before a long invalid match. Thus, the scanner returns the expected symbol and the grammar describing type expressions is simplified.

Our motivation in developing these parsing and scanning algorithms, and their generation algorithms, was to parse extensible languages. An example of a program written in an extended language is shown in Figure 1. It is written in Java^{-}, a small subset of Java, augmented with two extensions. The first extension embeds SQL and relational-database type schema specifications into Java so that SQL queries can be written directly and also statically checked for syntax and type errors [15]. The `connection` construct specifies the database location and lists the tables and their field names and types, which are used to typecheck queries statically. The example query given in the `using` construct names the connection to the database on which the query should execute.

The second extension adds *condition tables*, a construct found in specification languages such as SCR [7] and used for specifying complex Boolean conditions in a tabular format. In each row of the table there are "truth value" entries `T` (true), `F` (false), or `*` (don't-care) indicating the desired truth value of the preceding Boolean expression. Tables can be read by taking for each column the conjunction of the truth-value modified expressions, and then taking the disjunction of these conjunction expressions. Therefore, the assignment to `b` in Figure 1 is semantically equivalent to the pure-Java code shown below:

```
b = (age > 40 && gender == "M") ||
    (true && !(gender == "M")) ;
```

As the focus of this paper is on parsing and scanning, we will not concern ourselves further with the semantics of these extensions.

```
class Demo {
 int demoMethod ( ) {
  List<List<Integer>> dlist ;
  int SELECT ;
  int T ;
  connection c "jdbc:derby:/home/derby/db/testdb"
   with table person  [ person_id  INTEGER,
                        first_name VARCHAR,
                        last_name  VARCHAR ] ,
        table details [ person_id  INTEGER,
                        age        INTEGER,
                        gender     VARCHAR ] ;
  Integer limit ;
  limit = 18 ;
  ResultSet rs ;
  rs = using c query
      { SELECT age, gender, last_name
        FROM   person , details
        WHERE  person.person_id =
               details.person_id
           AND phonebook.age > limit  } ;

  Integer age ;
  age = rs.getInteger("age");
  String gender ;
  gender = rs.getString("gender");
  boolean b ;
  b = table ( age > 40      : T * ,
              gender == "M" : T F ) ;
 }
}
```

Figure 1. Sample program in Java$^-$ extended with SQL and condition tables.

There are aspects of this program that are difficult for traditional (and other) scanning and parsing techniques to handle. This is partly due to the fact that the extended language's concrete syntax is the composition of that defining the host language and extensions, and that the extension grammars may be written independently of each other, the one knowing nothing of the other. Also, the embedded DSLs and language fragments may have their own notions of reserved keywords and operator precedence or associativity settings that are different from those in the host language. Some challenges in parsing the program in Figure 1 include:

1. Context-based preference of keywords: "SELECT" is recognized as a SQL keyword in the context of an SQL query, but will be recognized as Java$^-$ identifier in other contexts.
2. Context-based keyword disambiguation: The string "table" is recognized as either an SQL keyword or a condition table keyword, again, depending on the context.
3. Context-dependent operator precedence and associativity: The expressions in SQL "where" clauses use "=" for equality instead of the Java$^-$ "==". The "=" operator has precedence and associativity specifications for SQL expressions that are not valid for Java$^-$ assignment statements.

More generally, as extensions may be written independently of one another we must be concerned with the possibility of introducing syntactic ambiguities into the grammar when combining host and language extension specifications. Similarly, lexical ambiguities may be introduced, causing the scanner to match more than one terminal on some input string.

The primary contributions of this paper are:

- Deterministic parsing and scanning algorithms that can handle a wider range of languages and are especially appropriate for extensible languages in which DSLs can be embedded.
- Modifications to DFA-based scanner-generation algorithms to support context-aware scanning.
- An analysis that can verify at scanner-generation time that the scanner is deterministic, *i.e.*, for any input and for any parse state, it will never return more than one token.

Section 2 describes concrete syntax specifications used to generate the parsers and scanners that use context-aware scanning. Section 3 describes the modified LR parsing algorithm that uses the context-aware scanning algorithm described in Section 4. Section 5 shows how the specifications can be statically analyzed to guarantee that the parser and the scanner are deterministic. Section 6 describes the features and performance of a tool called Copper that implements these processes and discusses our experience in using these tool to build parsers and scanners for Java 1.4, several extensions to Java 1.4, AspectJ, ANSI C, and Promela. Section 7 describes related work and Section 8 concludes.

2. Extensible language specifications

Along with our colleagues, we have developed tools for building extensible language frameworks in which a programmer can import into his or her host language the combination of domain-specific (or general-purpose) language extensions that raise the level of abstraction of the language to that of the task at hand. We have used these tools to develop an extensible specification of Java [15] and an extensible specification of Lustre [5], a synchronous language used in modeling safety-critical systems. In each language framework, the host language and the language extensions are written in Silver [13], an attribute grammar (AG) specification language that also includes specifications for concrete syntax. The Silver tools support the modular specification of language extensions and their composition with the host language such that in many cases the programmer does not need any implementation-level knowledge of the language extensions — the composition of the host language and the programmer-selected extensions is automatic.

In this paper, we describe the specification of the concrete syntax of the host language and language extensions, the composition of these specifications, and the algorithms for parsing and scanning the extended (composed) language. Figures 2, 3, and 4 present the concrete syntax (syntactic and lexical) extracted from their corresponding Silver specifications. The three specifications are combined to form the concrete specification of the extended language. They are written in a notation very much like Silver, but without the AG aspects and with a few notational conveniences (such as "|" in production rules) that simplify the presentation.

Host language specifications: Figure 2 describes the concrete syntax of the host language Java$^-$, a subset of Java[1] used in this paper for explanatory purposes. This grammar, when combined with the extension grammars, is sufficient for parsing the example program in Figure 1. The host language is a context free grammar $H = \langle S_h, T_h, NT_h, P_h, \succ_h \rangle$, consisting of a start symbol, sets of terminals, nonterminals, and productions, and a binary relation $\succ_h$ over terminals T_h. $\succ_h$ is used to set lexical precedences for terminals whose regular expressions define overlapping languages. A common use is in specifying that keywords take precedence over identifiers. This relation is specified in the `lexer classes`, `submitsTo` and `dominates` clauses and is defined precisely below.

[1] This is not strictly true — although `Integer` and `String` are not actually reserved keywords in Java, we make them so here for pedagogical reasons.

```
grammar edu:umn:cs:melt:simplejava:host ;

-- NonTerminals
start nt Root ;
nt Class, ClassMem, ClassMems, Type, Params,
   Stmt, Stmts, Expr ;

-- Terminals
t IntLit_t /[0-9]+/ ;
t StrLit_t /[\"][^\"]*[\"]/ ;

lexer class host_kwd ;
t Int_t     'int'     lexer classes = { host_kwd } ;
t Integer_t 'Integer' lexer classes = { host_kwd } ;
t String_t  'String'  lexer classes = { host_kwd } ;
t Boolean_t 'boolean' lexer classes = { host_kwd } ;
t Class_t   'class'   lexer classes = { host_kwd } ;
t While_t   'while'   lexer classes = { host_kwd } ;

t Id_t /[a-zA-Z][a-zA-Z0-9]*/ submitsTo { host_kwd };

t LCurly_t  '{' ;         t RCurly_t  '}' ;
t LParen_t  '(' ;         t RParen_t  ')' ;
t LSquare_t '[' ;         t RSquare_t ']' ;

t Colon_t   ':' ;         t Semi_t    ':' ;
t Comma_t   ',' ;         t Assign_t  '=' ;
t Dot_t     '.' ;

t Star_t     '*'   prec=6, assoc=left;
t Plus_t     '+'   prec=5, assoc=left;
t BitShift_t '>>'  prec=4, assoc=left;
t GT_t       '>'   prec=3, assoc=none;
t EQ_t       '=='  prec=3, assoc=none;

ignore t LineComment_t /[\/][\/].*/ ;
ignore t SpaceTabNewLine_t  /[\ \t\n]+/ ;

-- Productions
Root ::= Class
Class ::= 'class' Id_t '{' ClassMems '}'
ClassMems ::= ClassMem ClassMems
            | ClassMem
ClassMem ::= Type Id_t ';'   -- field dcl
ClassMem ::= Type Id_t '(' Params ')' '{' Stmts '}'
Params ::=    -- no parameters
         |  Type  Id_t ','  Params
         |  Type  Id_t
Stmts ::=     -- no statements
        |  Stmt  Stmts
Stmt ::= Type Id_t ';'  -- local dcl
       | Id_t '=' Expr ';'
       | 'while' '(' Expr ')' Stmt
Type ::= 'int' | 'Integer' | 'String' | 'boolean'
       | Id_t
       | Id_t '<' Type '>'
Expr ::= Id_t | IntLit_t | StrLit_t | '(' Expr ')'
       | Expr '>' Expr
       | Expr '>>' Expr
       | Expr '*' Expr
       | Expr '+' Expr
       | Id_t '.' Id_t '(' Expr ')'
```

Figure 2. Specification of concrete syntax of Java$^-$ host language.

Nonterminals are declared with `nt` followed by the name of the nonterminal symbol. `start` indicates the start nonterminal (`Root`); there are also nonterminals for classes (`Class`), class members and sequences thereof (`ClassMem` and `ClassMems`), type expressions (`Type`), parameters, statements, and expressions.

Terminals are declared by `t`, in the same way. By convention, these names have a "`_t`" suffix, but this is not required; we do not follow the convention of using uppercase words for terminals and lower case words for nonterminals. Following the terminal name is its regular expression. Those of the integer literal terminal `IntLit_t` and the string literal terminal `StrLit_t` are written between forward slashes. If the regular expression defines a fixed string, it can instead be written inside single quotes, such as `'class'` on the `Class_t` keyword terminal. Regular expressions in single quotes can be used to reference their terminals in productions, as is done in the `Class` production and in the prose of this paper. (If several terminals sharing one fixed-string regular expression, however, one should refer to each one by its name.)

Following the specifications of the keyword terminals are those of the identifier terminal `Id_t` and the several punctuation terminals. Specifications of operator precedence and associativity are given on binary operator terminals; these are processed as in traditional LR parser generators. Finally, whitespace and comments are specified; the `ignore` modifier indicates that they are thrown away by the scanner.

Last are shown BNF productions for Java$^-$.

Lexical precedence relation: In traditional approaches, a precedence ordering is placed on all terminal symbols T so that if the regular expressions for more than one terminal match the same prefix of the input string, the one with the highest precedence is selected. In tools like Lex, this precedence ordering is a total ordering, implicitly specified by the textual order in which the terminals appear in the lexical specification.

In our approach, we cannot base any precedence ordering for terminals on their textual order in the grammar specification, as the extensions that contain the terminal specifications are composed with the host language as an un-ordered set, not an ordered list. Furthermore, many terminals never appear in the same valid lookahead set and thus do not need precedence orderings to disambiguate. Instead, an asymmetric, non-transitive relation $\succ$ is specified, consisting only of the relations explicitly specified by the `submitsTo` and `dominates` clauses in the specifications. To simplify the mechanics of the explicit specification, each terminal can be a member of various *lexical classes*, as specified by the `lexer classes` clause on terminal declarations. In Figure 2, the `host_kwd` lexical class is declared above `Int_t` and the keyword terminals are specified as its members. The identifier terminal `Id_t` specifies that it has lower precedence than all members of this class via the `submitsTo { host_kwd }` clause. Since this is the only such specification in this grammar, the relation $\succ_h$ specifies only that `Int_t` $\succ_h$ `Id_t`, `Integer_t` $\succ_h$ `Id_t`, `String_t` $\succ_h$ `Id_t`, `Boolean_t` $\succ_h$ `Id_t`, `Class_t` $\succ_h$ `Id_t`, and `While_t` $\succ_h$ `Id_t`. This information is used to prefer keywords to identifiers when scanning strings matching a keyword's regular expression.

If the relation $\succ$ specifies $t_1 \succ t_2$ then in all parsing contexts t_1 has precedence over t_2. The specification of $t_1 \succ t_2$ effectively adds t_1 to all valid lookahead sets containing t_2. For any context in which the set of terminals ts are valid, we will also consider the set $higherPrecTerms(ts)$ as valid, where

$$higherPrecTerms(ts) = \bigcup_{t \in ts} \{t' \in T \mid t' \succ t\}.$$

For example, in a context where `Id_t` is valid, all the appropriate keyword terminals would also be considered as valid. Thus, even in contexts where `Id_t` is valid but where a keyword such as

```
grammar edu:umn:cs:melt:simplejava:exts:tables ;
import  edu:umn:cs:melt:simplejava:host ;

nt TableRow, TableRows, TValue, TValues ;

t CondTable_t 'table' dominates { Id_t } ;
t TrueTV_t   'T'  ;
t FalseTV_t  'F'  ;
t StarTV_t   /\*/ ;

Expr ::= CondTable_t '(' TableRows ')'
TableRows ::= TableRow ',' TableRows  |  TableRow
TableRow  ::= Expr ':' TValues
TValues ::= TValue TValues  |  TValue
TValue ::= TrueTV_t | FalseTV_t | StarTV_t
```

Figure 3. Specification of concrete syntax of the tables extension.

`Class_t` cannot be part of a syntactically correct program, we need to scan "class" as the keyword terminal, not as an identifier. This would happen in processing a syntactically incorrect Java statement such as `class SELECT;`. To generate the appropriate parse error, "class" is recognized as `Class_t` and not as `Id_t`.

Language extension specifications: The extensions specify grammars with no start symbols, in relation to the host language specification, as is indicated by the `import` statement at the top of each specification. Figure 3 defines the condition tables grammar $CondTables = \langle T_t, NT_t, P_t, \succ_t, H \rangle$ and Figure 4 defines the SQL grammar $SQL = \langle T_s, NT_s, P_s, \succ_s, H \rangle$. Productions in language extensions will use host language (from H) terminals and nonterminals; also, their precedence relations will include terminals from the host language: $\succ_t \subseteq (T_h \cup T_t) \times (T_h \cup T_t)$ and $\succ_s \subseteq (T_h \cup T_s) \times (T_h \cup T_s)$. In the tables grammar in Figure 3 we see that the `CondTable_t` keyword dominates `Id_t`; thus `CondTable_t` $\succ_t$ `Id_t`. The SQL keywords are members of the `sql_kwd` lexer class; the terminal `Using_t` dominates `Id_t` but the other `sql_kwd` terminals do not, since these keywords and `Id_t` are never both valid lookahead, and thus we will never have to rely on precedence to distinguish between them. In fact, as illustrated in Figure 1 and described in Section 3.2, there are contexts where we want to recognize the string `SELECT` as an identifier `Id_t` and contexts where we want to recognize it as a SQL keyword.

Grammar composition: The grammar for the composed language Java^{-} + *SQL* + *CondTables* is $L = \langle T_h \cup T_s \cup T_t, NT_h \cup NT_s \cup NT_t, P_h \cup P_s \cup P_t, S, \succ_h \cup \succ_s \cup \succ_t \rangle$. This is just the set union of the components of the grammars.

```
grammar edu:umn:cs:melt:simplejava:exts:sql ;
import  edu:umn:cs:melt:simplejava:host ;

-- embedded SQL queries
nt SQL, SQL_Expr, SQL_Ids ;
TableRow, TableRows, TValue, TValues ;

lexer class sql_kwd ;
t Using_t  'using'  lexer classes = { sql_kwd },
                    dominates { Id_t } ;
t Query_t  'query'  lexer classes = { sql_kwd } ;
t Select_t 'SELECT' lexer classes = { sql_kwd } ;
t Where_t  'WHERE'  lexer classes = { sql_kwd } ;
t From_t   'FROM'   lexer classes = { sql_kwd } ;
t And_t    'AND'    lexer classes = { sql_kwd } ,
                    prec = 3, assoc = none ;
t SQL_EQ_t '='      prec = 4, assoc = none ;
t SQL_Id_t  /[a-zA-Z][a-zA-Z0-9]*/
            submitsTo { host_kwd, sql_kwd } ;

Expr ::= 'using' Id_t 'query' '{' SQL '}'
SQL ::= 'SELECT' SQL_Ids 'FROM' SQL_Ids
        'WHERE' SQL_Expr
SQL_Ids ::= SQL_Id_t ',' SQL_Ids  |  SQL_Id_t
SQL_Expr ::= SQL_Id_t
           | SQL_Id_t '.' SQL_Id_t
           | SQL_Expr SQL_EQ_t SQL_Expr
           | SQL_Expr And_t SQL_Expr
           | SQL_Expr GTE_t SQL_Expr

-- embedded connection/table schema
nt TableDcl, TableDcls, FieldDcls, FieldDcl ;

t Conn_t  'connection' lexer classes = { sql_kwd },
                       dominates { Id_t } ;
t With_t  'with'       lexer classes = { sql_kwd };
t Table_t 'table'      lexer classes = { sql_kwd };
t SQL_Type_t /(VARCHAR)|(INTEGER)/
                       lexer classes = { sql_kwd };

Stmt      ::= 'connection' Id_t StrLit_t 'with'
              TableDcls ';'
TableDcls ::= TableDcl ',' TableDcls  | TableDcl
TableDcl  ::= Table_t SQL_Id_t '[' FieldDcls ']'
FieldDcls ::= FieldDcl ',' FieldDcls  | FieldDcl
FieldDcl  ::= SQL_Id_t SQL_Type_t
```

Figure 4. Specification of concrete syntax of the SQL extension.

3. Modified LR parsing algorithm/scanner interface

Our approach uses a modified LR parsing algorithm and modified DFA-based scanning algorithm whose parse tables and DFAs are generated from declarative specifications such as those in Figures 2, 3, and 4. Before describing our modified LR parsing algorithm, we briefly discuss traditional LR parsing.

3.1 Traditional LR parsing

In traditional LR parsing [9, 1] the input grammar is used to generate a *parse table* that drives the parsing algorithm. While parsing, the algorithm maintains a *parse stack*, consisting of pairs of *parse states* and concrete syntax trees (CSTs). The traditional algorithm is the same as that of Figure 5 if one removes lines 5–6 and lines 10–23 and adds a call to a traditional disjoint scanner after line 29.

The parse table is indexed by parse states and grammar symbols. Its entries are called *parse actions* and have the form $error$, $accept$, $shift(ps \in ParseState)$, or $reduce(p \in P)$. Here we represent the parse table as a mapping $table : (ParseState, T \cup NT) \mapsto Action$ where *ParseState*s are typically represented as integers and correspond to states in the LR DFA [9, 1].

We assume a global immutable input string *input*; the parser is a function that returns the CST corresponding to *input* or exits with a parse error. We define a *token*, denoted by type Tk, as a tuple containing a terminal, a lexeme matching that terminal, and location information such as a line or column number. In our rendering of the algorithm we define the parse stack as a stack of $ParseState$/CST pairs, with associated operations $peek$, pop, $push$, and $multipop$. To access the $ParseState$ and CST inside a stack element, one uses the functions $state$ and $node$, respectively.

The execution of the traditional LR algorithm is directed by the parse table and parse stack. At each cycle, the parser calls to the scanner for the next token in the input, then called the *lookahead token*. The parser then looks at the top of the stack to get the present parse state. Using this state and the lookahead token as indices, it retrieves a parse action from the parse table, one of *shift*, *reduce*, *accept* or *error*. A shift action constructs a terminal CST node that is pushed onto the parse stack along with the shift action's parse state. The parser will also move the input past the lookahead token. A reduce action constructs a new CST using the action's production and the appropriate number of trees popped from the stack for children. It then pushes this tree onto the stack and changes the parser state to the state indicated by the goto action in the parse table. It then pushes the new CST and parse state onto the stack. An accept action stops the parser when parsing is complete. An error action stops the parser when there is a syntax error, *i.e.,* when the lookahead token has an error action in the table.

3.2 Modified LR parsing

We have modified the LR parsing algorithm in order to utilize context-aware scanning. We first define the two innovations around which all of our modifications to the traditional LR algorithm center: *valid lookahead* and the new parser-scanner interface.

Valid lookahead sets: A valid lookahead set is defined for each parse state ps as the set of terminals that have non-error parse actions associated with them: $validLA = \{t \in T \mid table(ps, t) \neq Error\}$. For example, in the initial state of a full Java parser, the valid lookahead would contain terminals such as `package`, `import`, `public`, and `class` because those can appear at the beginning of a Java file. It would *not* contain terminals such as `for`, because for-loops do not occur at the beginning of Java files.

Parser's interface to scanner: In the traditional LR algorithm, the type of the scanner function is $nextToken : int \rightarrow \langle Tk, int\rangle$. In context-aware scanning the interface is slightly different; the type of the scanner function is $nextToken : \langle \mathcal{P}(T), int\rangle \rightarrow \langle \mathcal{P}(Tk), int\rangle$. Both functions take the position from which to scan, but the latter also takes the set of valid lookahead terminals for the current parse state; both return the new position, while the traditional function returns exactly one token and the new function returns a set of tokens. If the returned token set is empty, then a lexical error has occurred. If its size is 1, then no error has occurred; if it is more than 1, a lexical ambiguity has occurred. The important invariant is that if $\langle ts', \_\rangle = nextToken(ts, \_)$ then $ts' \subseteq ts$ — the scanner only returns tokens in the set of valid lookahead.

Section 5 presents an analysis that is performed on the parse table and scanner DFA to ensure determinism — *i.e.,* that the scanner never returns more than one token, or if it does, a *disambiguation function* exists to select a single token from the set. Disambiguation functions (see section 5.2.1) are rarely required.

3.2.1 Modified LR parsing algorithm

Figure 5 contains the pseudo-code of our modified LR parsing algorithm. Lines 1–7: (1) initialize the parser start state, a flag $done$, and the position pos; (2) declare variables for the new position (np), parse state (ps), new parse state (ps'), current lookahead token (tk), valid lookahead terminals set, and tokens returned from the scanner; and (3) push a starting state onto the parse stack.

Entering the loop, line 9 retrieves the parse state from the top of the stack and stores it in ps. Line 10 retrieves from the parse table the valid lookahead for state ps. Consider the example "`List<List<Integer>>`" from section 1 in which the parser has just reduced "`Integer`" to `Type` in the previous loop iteration and input position pos indicates that "`>>`" is to be processed. $validLA = \{\texttt{'>'}\}$ since the current parse state is the one that contains only the LR item `Type ::= Id_t '<' Type • '>'`. Line 11 locates any terminals of higher precedence to what is already in the valid lookahead set, and places them therein. The terminal `'>'` is not used in the lexical precedence relation $\succ$, and therefore *higherPrecTerms*$(\{\texttt{'>'}\}) = \emptyset$. Line 12 calls the scanner on the present position, passing it the valid lookahead set. In this example the scanner locates a single greater-than sign and return it as the longest match. The input at that point also matches `BitShift_t`, but since `BitShift_t` $\notin validLA$, the scanner does not match it.

The three if-statements occupying lines 13–21 check if the match set returned by the scanner is comprised of exactly one element, and if not, attempt to make it so by applying the appropriate disambiguation function. If none exists an error is raised. In this example, there is only one match and line 23 is reached without incident; it extracts the single element of $lookAhead$ into a token variable tk. Line 24 retrieves an action from the parse table. In our example, parse table cell $(ps, \texttt{'>'})$ indicates a shift action. The switch-statement occupying lines 25–40 provides cases for every

```
function parse() returns CST
 1.  int startState = parser start state
 2.  boolean done = false
 3.  int pos = 0, np , ps, ps'
 4.  Tk  tk // current lookahead token
 5.  Set(T) validLA
 6.  Set(Tk) lookAhead
 7.  push(⟨startState, _⟩)
 8.  while ¬done do
 9.    ps = state(peek()) // copy parse state on top of parse stack
10.    validLA ={t ∈ T | table(ps, t) ≠ Error}
11.    validLA = validLA ∪ higherPrecTerms(validLA)
12.    ⟨lookAhead, np⟩ = nextToken(validLA, pos)
13.    if |lookAhead| = 0 then
14.      // generate parse error
15.      exit()
16.    if |lookAhead| > 1 then
17.      // possible lexical ambiguity
18.      lookAhead = applyDisambiguationFunctions(lookAhead)
19.      if |lookAhead| > 1 then
20.        // lexical ambiguity, unreachable for deterministic cases
21.        exit()
22.    // |lookAhead| = 1
23.    tk = first(lookAhead)
24.    action = table(ps, tk)
25.    switch action
26.      case shift(ps') :
27.        // perform semantic actions for tk
28.        push(⟨ps', tk⟩)
29.        pos = np
30.      case reduce(p : A ::= α) :
31.        children = map(node(multipop(|α|)))
32.        tree = p(children)
33.        // perform semantic actions for p
34.        ps' = table(ps, A)
35.        push(⟨ps', tree⟩)
36.      case accept :
37.        if lookAhead = EOF then done = true
38.        else // report error and exit
39.      case error :
40.        // report error and exit
41.  end while
42.  return node(pop())
```

Figure 5. Modified parsing algorithm.

kind of parse action. The code here is the same as that used in a traditional LR algorithm, as described above.

3.2.2 Use of precedence relation $\succ$

Consider parsing the program in Figure 1 when the parser is at the beginning of the line "`int SELECT;`" and the parser is prepared to shift `Int_t` for "`int`". At line 10, the parser retrieves $validLA$, here it is the set of terminals that can occur at the beginning of a statement in a method. This set contains exactly `Id_t`, `'int'`, `'Integer'`, `'String'`, `'boolean'`, `'while'`, and the SQL extension terminal `'connection'`. Line 11 adds new terminals to this set based on the precedence relation $\succ$. In the host grammar, `Id_t` submits to `'int'`, `'Integer'`, `'String'`, `'boolean'`, `'class'`, and `'while'`. In the condition-tables extension, `'table'` dominates `Id_t`. In the SQL extension, `'using'` dominates `Id_t`. The three that were not previously contained in *validLA* (`'class'`, `'table'`, and `'using'`) are then added to it. This ensures that in all parsing contexts the keywords have precedence over the identifier terminal. In the Java$^-$ statement "`class SELECT;`" discussed above, the new additions allow the terminal `Class_t` to match "`class`" and cause the desired parse error (since `Class_t` has no parse action in that state). In the case of "`int SELECT;`", the scanner returns the `Int_t` token. Since there are no lexical ambiguities the algorithm then branches to line 26 to shift the shift action. The parser pushes a new element on the stack containing the new parse state and the `Int_t` token.

The lexical precedence relation $\succ$ is not transitive because we need to avoid specifying an implicit relation between two terminal symbols when setting one to dominate a certain host terminal and the other to submit to it. In practice, the lack of transitivity has not been a problem, and with the use of lexer classes, specifying these implicit relations is quite simple.

In Section 6.2.2 we show how this same effect can be achieved though a modification to the scanner DFA that takes place at scanner-generation time instead of at parse time.

3.2.3 Example applications to parsing challenges

Here we illustrate how the parsing challenges enumerated in Section 1 are handled by context-aware scanning and parsing.

Context-based preference of keywords: Consider parsing "`int SELECT;`" at the point where the example in Section 3.2.2 has finished and the parser has just shifted `'int'` and is now ready to reduce it into a `Type` CST. At line 10, the algorithm computes $validLA$. This consists of one element, `Id_t`. At line 11, the keywords with higher precedence (`'int'`, `'Integer'`, `'String'`, `'boolean'`, `'while'`, `'class'`, `'table'`, `'using'`, and `'connection'`) are added to $validLA$. At line 12, the parser calls the scanner; the scanner reads `SELECT` and matches it as an identifier since `'SELECT'` $\notin validLA$. Thus, this approach can selectively reserve keywords based on the parsing context. In a traditional scanner, the SQL keyword `'SELECT'` would have been matched. Execution then branches to line 30 to perform the reduction using the production `Type ::= 'int'`. On the next iteration of the loop, the same situation occurs but the a shift action is taken to push `Id_t` onto the stack.

When processing has progressed to the SQL query and passed the "{" in the line "`{ SELECT age ...`," $validLA$ contains only the keyword terminal `'SELECT'` and not `Id_t` or `SQL_Id_t`. Thus "`SELECT`" is here recognized as a keyword disambiguated by context. Note that the SQL extension's inner productions do not reference `Id_t` but `SQL_Id_t`, which has lower precedence than (`submitsTo`) all host language and SQL (`sql_kwd`) keywords. Thus, the string "`SELECT`" will not be recognized as an identifier in the context of an SQL query and is disambiguated by precedence.

Context-based keyword disambiguation: The string "`table`" will be recognized as the SQL terminal `Table_t` in the context of an SQL connection construct but as the tables extension terminal `CondTable_t` in a Java$^-$ expression. These terminals are disambiguated by context and although they share a regular expression, they need not be related by the lexical precedence relation.

Operator precedence and associativity: A similar disambiguation by context happens for the host language terminal `Assign_t` and the SQL terminal `SQL_EQ_t` (both with the same regular expression `'='`) since `Assign_t` is not in the valid lookahead in the context of SQL queries and `SQL_EQ_t` is not in the valid lookahead in the context of Java. These terminals have different operator precedence and associativity settings and are handled in the traditional manner by the LR parsing engine. Context-aware scanning allows for finer distinctions in specifying operator precedence and associativity because it allows for the same lexeme (in this case "`=`") to be represented by two different terminals and therefore have different operator precedence and associativity.

4. Context-aware scanning

In this section we describe context-aware scanning, which is similar to traditional scanning techniques in that it is based on regular expressions for specification and deterministic finite automata (DFAs) for implementation. In both approaches, terminals have regular expressions associated with them, indicated by the mapping $regex : T \mapsto Regex$, and a DFA is constructed as follows [1]: (1) create a nondeterministic finite automaton (NFA) from each regular expression $regex(t \in T)$; (2) create a composite NFA from these constituent NFAs by introducing a new start state and adding epsilon transitions from that state to the start states of each constituent; (3) convert the NFA to a DFA.

The DFA has the form $\langle \Sigma, Q, \delta : Q \times \Sigma \to Q, start \in Q, acc : Q \mapsto \mathcal{P}(T), poss : Q \mapsto \mathcal{P}(T) \rangle$ where Σ is the alphabet, Q is the finite set of states (among which is a distinct error state $error_Q$), $start$ is the start state, acc indicates which terminals are accepted in each state of the DFA ($acc(error_Q) = \emptyset$), and:

- δ is the deterministic transition function. We augment δ such that for each $q \in Q$ where $\delta(q, s \in \Sigma)$ originally had no value, $\delta(q, s \in \Sigma) = error_Q$. The function $\delta^* : Q \times \Sigma^* \to Q$ is the natural extension of δ to sequences of symbols in Σ such that $\delta^*(q, x) = \delta(q, x)$ where $x \in \Sigma$ and $\delta^*(q, xv) = \delta^*(\delta(q, x), v)$.
- $poss$, a novel mapping in our approach, indicates what terminals may be accepted in *reachable* DFA states; *i.e.*, $poss(q) = \{t \in T | \exists w \in \Sigma^*.\ t \in acc(\delta^*(q, w))\}$. Thus, $acc(q) \subseteq poss(q)$ for any state q. Note that $\forall c \in \Sigma\ .\ (\delta(s_1, c) = s_2 \Rightarrow poss(s_2) \subseteq poss(s_1))$ — possible sets shrink monotonically along transition chains. Thus, $poss(error_Q) = \emptyset$.

The scanner function *nextToken* is given in Figure 6; it takes as input the valid lookahead set (*validLA*) and the position from which to scan (*whence*). This function first consumes whitespace. $WhiteSpace = \{ws\}$, where ws is a tool-generated terminal whose regular expression is the Kleene star ($*$) of the disjunction of the regular expressions of all terminals marked `ignore`. The *nextToken* function then scans for the tokens in the valid lookahead set from the new position. Note that *nextToken* and *scan* have identical type signatures.

Figure 7 contains the code for the context-aware scanner that takes the valid lookahead terminal set as input. The terminal set *validPoss* contains terminals that may possibly match after reading $pos - whence$ characters of the input; it is computed for each step through the DFA by intersecting the current state's possible set

```
function nextToken(Set(T) validLA, int whence)
returns ⟨Set (Tk), int ⟩
1.   ⟨_, np⟩ = scan(WhiteSpace, whence)
2.   return scan(validLA, np)
```

Figure 6. The *nextToken* function called by the parser.

$poss(ss)$ and the valid lookahead set ***validLA***. The loop terminates when it is not possible to match any terminals by reading further.

In the loop, we first check if a match is possible in the current state ss by checking if $acc(ss) \cap validLA \neq \emptyset$. If so, we save the current state and position as $lastMatch$ and $lastPos$. Next, the transition function δ is called on the current character, ***pos*** is incremented, and ***validPoss*** is computed for the new state. This continues until a state with no possible matches is entered.

After the loop, the set of matches is computed by intersecting $validLA$ with the accept set of $lastMatch$. Next, we filter from *matches* all terminals that have lower precedence than some other terminal in *matches*. (Note that $\succ$ is irreflexive so $t' \succ t \Rightarrow t' \neq t$.) The matched lexeme is used to build tokens returned to the parser.

To further understand this algorithm, consider some examples. First, in the example "`List<List<Integer>>`", after recognizing and shifting the token for "`Integer`" the scanner is called with the valid lookahead set {`GT_t`}. In the first iteration, the first `>` is read and $validPoss$ is set to {`GT_t`}. In the second iteration $acc(ss) \cap validLA \neq \emptyset$ and the match information for "`>`" is stored. The second `>` is consumed and since `GT_t` does not match any string of more than one character $validPoss$ is set to $\emptyset$ and the loop exits. The scan function subordinates the disambiguation principle of maximal munch to the principle of disambiguation by context; it will return a shorter valid match instead of a longer invalid match.

In scanning `int SELECT;` we showed that the valid lookahead set would contain the identifier terminal `Id_t` as well as `'int'` and several other keyword terminals. In this example, the scan function would consume "`int`" and record ***lastMatch*** as the state whose accept set is `Id_t` and `'int'`, then consume a space, which sets ss to $error_Q$ and causes the loop to exit. Thus, in line 14, the $\succ$ relation is used to remove `Id_t` from *matches*, since `'int'` $\succ$ `Id_t`. In Section 6.2.2 we will see how this precedence information can be used during scanner generation to partition the accept sets into restricted accept sets and *reject sets*, removing the need for scan-time match filtering of this kind.

Another interesting case occurs when one keyword is the prefix of another. Consider a Perl-Python hybrid in which both `for` and `foreach` are loop keywords followed directly by identifiers. Thus, in scanning "`foreach x`" when both `'for'` and `'foreach'` are in the valid lookahead, maximal munch dictates that `'foreach'` be recognized. There is no context in which only `'for'` is in the valid lookahead. But if there were, and the string "foreach" were scanned, *scan()* would scan for only `for`, and match it. It would then scan for an identifier and match "each," resulting in "foreach" having two separate meanings based on context. This seems odd, but it is identical to the example of scanning "`>>`" and returning 1 or 2 tokens depending on the context. In Section 6 we suggest a technique for requiring whitespace between terminals that addresses this issue.

5. Syntactic and lexical determinism

5.1 Syntactic determinism

The LR parsing algorithms (ours and the traditional one) guarantee syntactic determinism as they will not run on a parse table with shift-reduce or reduce-reduce conflicts. Conflicts are reported and resolved in the same manner in both approaches. Our approach differs from the traditional one in that we can produce more meaningful lexical tokens and hence parse a larger class of languages.

```
function scan(Set(T) validLA, int whence)
returns ⟨Set (Tk), int ⟩
1.   int ss = DFA start state
2.   int pos = whence
3.   Set(T) validPoss = poss(ss) ∩ validLA
4.   int lastMatch = error_Q
5.   while validPoss ≠ ∅ do
6.     if acc(ss) ∩ validLA ≠ ∅
7.       then lastMatch = ss; lastPos = pos;
8.     char ch = input[pos]
9.     ss = δ(ss, ch)
10.    pos = pos + 1
11.    validPoss = poss(ss) ∩ validLA
12.  end while
13.  Set(T) matches = acc(lastMatch) ∩ validLA
14.  matches = {t ∈ matches | ¬∃t' ∈ matches.t' ≻ t}
15.  string lexeme = getRange(input, whence, lastPos − 1)
16.  return {⟨t, lexeme⟩|t ∈ matches}
```

Figure 7. The modified scanning function called by *nextToken*.

5.2 Lexical determinism

A scanner exhibits lexical determinism if for any input, it will match 0 or 1 terminals. We have developed a simple analysis that tests whether or not lexical ambiguities exist in a lexical specification with respect to a given parser. Since the valid lookahead set plays a critical role in this analysis, lexical determinism can only be determined by examining both the lexical and context free syntax. In our algorithms, any set of tokens returned to the parser is the intersection of a valid lookahead set ($validLA$) and the accept set of a DFA-state ($acc(ss)$) from which is removed any terminal whose lexical precedence is lower than that of some other terminal in that set. Such a set with cardinality greater than 1 is a lexical ambiguity. Thus, the lexical syntax is free of lexical ambiguities if

$$\begin{array}{l}\forall\, sp \in ParseState\ .\\ \quad \forall\, ss \in Q\ .\\ \qquad |\ \{t \in validLA(sp) \cap acc(ss)\ |\\ \qquad\quad \neg\exists\ t' \in validLA(sp) \cap acc(ss)\ .\ t' \succ t\}\ | \leq 1.\end{array}$$

5.2.1 Disambiguation functions

In our specifications of Java 1.4 and the various extensions there are no lexical ambiguities [15]. But in our specifications for ANSI C and AspectJ there are a few lexical ambiguities. These can be resolved with *disambiguation functions* — a construct used to resolve a particular, otherwise unresolved, lexical ambiguity.

If the lexical determinism analysis finds a lexical ambiguity it will return the set $A \subset T$ of terminals that has cardinality greater than 1. To establish lexical determinism, a disambiguation function is then used in order to resolve A by selecting a single token to return. A disambiguation function is a pair $\langle A \subseteq T,\ f : \Sigma^\star \rightarrow Tk\rangle$. A is a set of terminals for a particular ambiguity; f is a function that takes the matched lexeme (and, in practice, line and column information) and returns a single token corresponding to a terminal in A. Any A with a disambiguation function does not need to be reported as a lexical ambiguity. Line 18 of the modified LR parsing algorithm in Figure 5 applies the relevant disambiguation function if the scanner returns more than 1 token. Since we can statically test that a disambiguation function exists for each ambiguity set A, we can be sure that lines 20 and 21

are unreachable and thus lines 19–21 can be safely removed from parsing algorithm generated for a language with no unresolved lexical ambiguities.

6. Discussion

6.1 Copper: a context-aware parser/scanner-generator tool

We have incorporated these algorithms in the form of a new parser and scanner generator tool, named Copper. Input to Copper specifies both lexical and context-free syntax in a manner similar to that shown in Section 2. Copper uses traditional algorithms for generating LALR(1) parse tables. It also performs the checks for determinism described above so that when the parser and scanner are generated, any conflicts or lexical ambiguities are reported. Augmenting the new algorithms are several features to handle "fringe cases" where a purely declarative paradigm may not suffice; among these are *parser attributes*. We have also worked to adapt error reporting to the new algorithms.

Parser attributes: Parser attributes are an abstracted version of the custom variables that can be found in any practical parser generator. In Yacc-like tools these are global variables updated by assignment statements in the semantic actions associated with the parse rules. In implementations for purely functional languages these occur in so called monadic parsers in which the values are threaded through the parsing process using a monad [10, Section 2.5]. Our approach is similar to these with the small difference being that they are declared directly as part of the syntax specifications. Parser attributes can be referenced as variables inside semantic actions (carried out at shift and reduce time) and in disambiguation functions.

Implementation of disambiguation functions: In addition to the set of tokens given to a disambiguation function, these functions can access parser attributes. This is useful is in the specification of ANSI C. Typenames and identifiers in C share the same regular expression, and they may often occur in the same context. It is not possible to use just one terminal for both cases for the standard grammar as this introduces conflicts to the grammar. This lexical ambiguity is usually resolved by maintaining a list of typenames created by `typedef` statements, which is checked whenever such an ambiguity arises. In this case one would declare a disambiguation function for $A = \{\texttt{typename}, \texttt{identifier}\}$ that returns one or the other based on the content of a parser attribute containing the typename list.

Error reporting: In a traditional scanner, if the input contains a valid token that is out of place (such as the "`class SELECT;`" example above) the parser will print out "Unexpected token `class`." With our algorithms such a message is impossible to produce: the scanner is blind to everything outside the valid lookahead set, and if nothing matches, it simply returns an empty match set. In such a case the only thing the parser has to display is the valid lookahead set, which is not very descriptive in that situation. A better thing to do is to run the scanner with a valid lookahead set containing all the grammar's terminals. The scanner will then return a (possibly ambiguous) match, which can be displayed along with the valid lookahead set.

Other features: We are also interested in a modular test that can be performed when a language extension designer specifies the concrete syntax of his or her extension, ensuring that when a programmer combines several language extensions that all pass this modular test no conflicts will occur in the parser and no lexical ambiguities occur in the scanner of the extended language. We have developed a prototype of such a test [15, 16] that is based on the principle of separating the parser DFA into partitions associated with precisely one grammar, either the host language grammar or an extension grammar.

Another enhancement to our approach allows different layout (whitespace and comments) for embedded languages. For example, in the tables extension we may like to use the newline instead of the comma to separate table rows. Layout can be specified on a per-production basis by allowing the explicit specification of the layout that is to appear between symbols on the right-hand side of a production. This can be used to address the situation with the `'for'` and `'foreach'` keywords described at the end of Section 4 by adding explicit whitespace between the `'for'` and identifier terminals on the for-loop production.

6.2 Performance

Our parsing algorithm, like all deterministic-LR algorithms, runs in linear time on the number of terminals shifted. Valid lookahead sets can be implemented to be retrieved in constant time.

6.2.1 Rescanning after reduce actions

Traditionally, the retrieval of each token of lookahead has been treated as taking constant time, since it is possible for the scanner to run as a pre-process, sending a ready-made token stream to the parser. Our integrated algorithm, on the other hand, requires a character string for input, and running the scanner at the same position in the string may produce a different match based on the valid lookahead set.

This poses a problem with chains of reduce actions. Naïvely, one would assume that after every reduce action, the algorithm would have to run the scanner again at the same position. However, there exists a property of lookahead sets that is very useful here: the lookahead set on an LALR(1) item is all symbols that may follow it. Thus, in all cases, the valid lookahead set facing the parser *after* a reduce action will be a (not necessarily proper) subset of the previous valid lookahead set. Furthermore, this subset will still contain the terminal that was matched on the previous scan.

Therefore, by memoizing the previous scan result to avoid a rescan, all rescanning can be eliminated except in cases where dynamic post-process disambiguation techniques such as disambiguation functions are used. Here it is necessary to re-scan, for two reasons: (1) while the terminal selected by the disambiguation function is still in the valid lookahead set, all members of the group may not be, meaning that the same disambiguation function would not be used on a rescan; (2) if all group members are still present, the reduce action may have altered parser attributes, causing the old disambiguation function to produce a different result.

Clearly, careless use of disambiguation functions could result in a more inefficient scan. However, in practical tests with a C grammar we have found that each token for which a disambiguation function was used is scanned an average of approximately 1.1 times, the maximum average for a file being 1.4.

6.2.2 Processing lexical precedence statically

Lexical disambiguation based on the lexical precedence relation $\succ$ shown in the parse function at line 11 and in the *scan* function in line 14 is done dynamically, *i.e.*, at parse and scan time. However, it can be done statically, at parser and scanner generation time, thus greatly increasing efficiency. Copper uses this optimization.

After the scanner DFA states have been generated and the states labeled with their accept sets acc and possible sets $poss$ have been computed, we add a new mapping $rej : Q \mapsto \mathcal{P}(T)$ that labels each state with a set of terminals called its *reject set*. We then move lower precedence terminals from a state's accept set to its reject set. Let $lower(ts) = \{t \in ts | \exists t' \in ts \,.\, t' \succ t\}$; this represents the terminals in ts that have a lexical precedence that is lower than some other terminal in ts. The reject set $rej(ss)$ is defined as $lower(acc(ss))$. The accept set is updated to remove from $acc(ss)$ those elements in $rej(ss)$.

The *parse* function in Figure 5 is modified by removing line 11 and the *scan* function of Figure 7 is modified by removing line 7 and adding the following statement after line 7:

`elseif` $rej(ss) \cap validLA \neq \emptyset$ `then`
$lastMatch = error_Q$

This is necessary in order to "drop" any matches of lesser length.

For example, consider scanning the string "while (..." in Java$^-$ when the valid lookahead set contains only the identifier terminal `Id_t`. This should result in a parse error since `'while'` $\succ$ `Id_t`. On the fifth iteration of the loop, line 7 will set *lastMatch* to a state whose accept set contains `Id_t` and pos such that the lexeme would be "whil". Line 9 sets ss to a state whose accept set contains `'while'` and whose reject set contains `Id_t`. On the next iteration, the new code above is utilized since its condition evaluates to *true*. We set *lastMatch* to $error_Q$ to remove the previous match of `Id_t` for lexeme `'whil'`. Thus, when the loop exits, no matches are found and the scanner returns an empty token set, triggering the parse error.

The effect is to remove code that computes sets of terminals based on the $\succ$ relation and replace it with code that computes sets of terminals based on set intersection.

This does not increase the overall complexity of *scan*, but it is necessary here to discuss scanner complexity. Although the scanner runs in linear time strictly with respect to the size of the input string, it should also be near constant-time with respect to the size of the *grammar*. Unlike with a traditional scanner, it is not strictly a constant-time operation with respect to grammar size to determine when the scan should stop, or to determine what to match in a given state. All superconstant operations, however, are based on intersections of sets of terminals; since the number of terminals is known at scanner generation time, efficient bit-vector implementations of these set operations can be generated.

6.3 Experience

We have used Silver and Copper to implement a number of languages and language extensions. Our ableJ framework [15] defines Java 1.4 and several language extensions that are parsed and scanned using our approach. One extension adds the constructs from AspectJ to provide a declarative and deterministic specification of that language. The disambiguation of lexical syntax is accomplished primarily by context and the precedence relation $\succ$; only five disambiguation functions are needed. All five contain the Java identifier terminal and a conflicting AspectJ keyword, and simply return the keyword. These AspectJ keywords are not specified to have lexical precedence over the Java identifiers because they can be used as Java identifiers in Java parts of an AspectJ program. We have also built parsers and scanners for ANSI C and Promela, the input language to the SPIN model checker. Promela incorporates embedded C code; our specification of Promela simply imports the ANSI C grammar to enable this. The context-aware scanning resolves all lexical ambiguities except for three dealing with comments, which are resolved by lexical precedence specifications.

It is conceivable that to specify the lexical precedence relation $\succ$ requires an understanding of LR parsing and the parse table for the grammar being defined. However, this is not the case, as the lexical precedence relation specifies a precedence for *all contexts* (parse states). For example, specifying that a keyword takes precedence over an identifier does not require any detailed knowledge of the algorithm or the parse table. In the SQL extension, we introduced `SQL_Id_t` with the same regular expression as `Id_t` from the host language instead of using `Id_t` in SQL expressions. This is because SQL keywords have lexical precedence over identifiers in the SQL constructs but not in Java constructs. To know to do one must understand that lexical precedence occurs in all contexts, but one need not understand the details of the algorithm or parse table. In our experience, context-aware scanning prevents most lexical ambiguities and resolving the remainder with precedence relations or disambiguation functions is easier than modifying the grammar to resolve conflicts in the LR parse table, which does require an understanding of parse table construction.

7. Related Work

There is a lot of research on parsing and scanning techniques and we will not attempt to discuss them all. Instead we describe some techniques that are also applicable to extensible languages and are either closely related to ours, or have specifications that can be easily composed to create a deterministic parser.

GLR: GLR (generalized-LR) parsing nondeterministically takes all actions when encountering a parse conflict; hence GLR can even parse ambiguous grammars. GLR has been greatly improved recently. Johnstone *et al.* [8] have taken its runtime down from exponential to cubic. Wagner and Graham [18] have shown empirically that most ambiguity occurs near the bottom of parse trees and that on practical languages the amount of time lost is very low. McPeak and Necula [11] have shown that GLR runs deterministically 70% of the time and capitalized on that fact in their optimized Elkhound system. Thus speed is not a primary factor for avoiding GLR.

An advantage of a nondeterministic system is that the disambiguation of C typenames and identifiers can be deferred to the semantic analysis phase [18]. However, despite being robust, GLR algorithms cannot guarantee determinism on grammars generating conflicting parse tables; ambiguities must be located by a mixture of debugging, intense empirical testing, and human intuition.

Visser has developed a novel system of parsing based upon the GLR engine. [12, 17] He turns nondeterminism to his advantage by eliminating the scanner and using character-level grammars, eliminating the scanner/parser dichotomy. Deterministic character-level grammars are nearly nonexistent; not so much in the nondeterministic system, which allows for unlimited lookahead capability as well as tolerance of ambiguity.

Adapting lexical syntax to the context free model requires some new footwork, as certain convenient features of lexical DFAs are not present to be exploited; they are replaced with "disambiguation filters" [12]. These include *follow restrictions* to replace maximal munch (*e.g.*, a number cannot be directly followed by a digit) and *reject productions* to replace the preference for keywords specified by lexical precedence. Operator precedence and associativity are still present, with the traditional effects. However, it still makes no guarantee of determinism, being based upon the GLR engine. Although this may be reasonable in some cases, for building extensible languages we prefer a guarantee of determinism since languages may be composed at the direction of the programmer, not a language expert who can resolve syntactic or lexical ambiguities.

XGLR [3] extends GLR to allow a different scanner state (e.g. input position) to be associated with each parse thread and thus supports lexically ambiguous input. As with scannerless GLR, the XGLR engine uses parser context to choose (disambiguate) the correct scan or tree at run time and provides no determinism analysis.

It is worth noting that context-aware scanning can also be used with GLR parsing algorithms. In fact, an early prototype implementation of Copper did just that. But since the LALR(1) version can handle a wide class of languages that includes the language extensions we have developed and because it provides the determinism analysis we seek, Copper does not use GLR.

Schrödinger's Token: Another approach to this problem is the "Schrödinger's token." Aycock and Horspool [2] present nonreserved keywords as a hurdle for traditional parsers. For example, in PL/I it is not known at scan-time whether a token such as `IF` is a keyword or identifier. The "Schrödinger's token" is a token

representing a lexical ambiguity; it may occur alone, representing an ambiguity on a particular lexeme, or in a sequence, representing a more profound ambiguity. An individual Schrödinger's token consists of more than one numbered terminal/lexeme pairs. The terminal may be from the grammar's terminal set or it may be a "null" terminal, in which case its corresponding lexeme is the empty string. The "null" tokens, which the parser ignores, are used as padding in an ambiguous scan where one interpretation has more tokens than the other, such as in the case where ">>" can be interpreted in C++ as a shift-right operator or two closing brackets.

Although Aycock and Horspool's system mandates a conflict-free parse table, the Schrödinger's tokens embody lexical ambiguity. When used by the parser as lookahead, they represent several parse actions; the parsing algorithm must take these concurrently, requiring a GLR parser to implement. Although this causes a minimal amount of ambiguity, there is, yet again, no guarantee of determinism. Also, in this system significant effort must be expended in order to pull off the nondeterministic scan. In the case where lexical boundaries are unclear, such as their example of the C++ templates, the addition of the "null" tokens must be carefully orchestrated.

PEGs: Another kind of scannerless parser is the novel *packrat parser*, which is used for *parsing expression grammars (PEGs)*. PEGs are deterministic by definition: there is exactly one production for each nonterminal. Productions are written in an EBNF-like format, with the only "branching" mechanism being an "ordered choice" written as "/" [4, 6]. All constructs in PEGs are also greedy [6]. Being completely deterministic, they are closed under composition. Grimm [6] has written a packrat parser generator known as *Rats!* specifically for use with extensible languages.

As well as being closed under composition, Grimm's system is scannerless and supports non-declarative specifications for fringe cases such as the C typename/identifier ambiguity. However, PEGs come with drawbacks: (1) Packrat parsers are character-level backtracking LL(1) parsers, and while they run in linear time, like any LL(1) parser they are unable to parse any left-recursive grammar without special modifications; (2) a context-free grammar extension connects itself to the host by inserting new productions; analogically a PEG extension must connect itself by inserting new "ordered choices." But our extensions are intended to be insensitive to the order of addition since they are added as un-ordered sets of extensions to the host language, not ordered lists. The "ordered choices" in PEGs do not support this.

8. Conclusion

We have presented deterministic parsing and scanning algorithms in which the scanner uses the parse state to disambiguate lexical syntax. This approach can deterministically parse and scan a class of languages that is strictly larger than what is possible with traditional LR parsers and disjoint scanners. The principle is that when the scanner can be more discriminating in the tokens that it returns it can help the parser to recognize a wider class of languages. Since the parse state is used by the scanner to be more discriminating, there is an effective cooperation between them.

Silver, Copper, and the language specifications shown in this paper are freely available on the Internet at `www.cs.umn.edu`.

Aho, Sethi, and Ullman [1, page 84-85] and Aycock and Horspool [2] mention several reasons why it is best to separate the parser and scanner into disjoint operations. We agree with them that a disjoint parser and scanner should be used for languages in which this is possible and a clean specification of the context-free and lexical syntax can be given. However, there are several languages for which this is not possible. We feel that the interaction between the parser and scanner as presented here has benefits that outweigh the drawback of tying them together.

Acknowledgments

We thank Lijesh Krishnan, Jimin Gao, Derek Bodin, and Yogesh Mali for their work on Silver and the extensible specifications for Java, Lustre, ANSI C, and Promela. We also thank Mark van den Brand for pointing out the related work of Aycock and Horspool.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers – Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[2] J. Aycock and R. N. Horspool. Schrödinger's token. *Software: Practice and Experience*, 31(8):803–814, 2004.

[3] A. Begel and S. L. Graham. XGLR - an algorithm for ambiguity in programming languages. *Science of Computer Programming*, 61(3):211–227, 2006.

[4] B. Ford. Parsing expression grammars: a recognition-based syntactic foundation. *SIGPLAN Not.*, 39(1):111–122, 2004.

[5] J. Gao, M. Heimdahl, and E. Van Wyk. Flexible and extensible notations for modeling languages. In *Fundamental Approaches to Software Engineering, FASE 2007*, volume 4422 of *LNCS*, pages 102–116. Springer-Verlag, March 2007.

[6] R. Grimm. Better extensibility through modular syntax. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 38–51, New York, NY, USA, 2006. ACM Press.

[7] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proc. of the* 10^{th} *Annual Conference on Computer Assurance, COMPASS 95*, 1995.

[8] A. Johnstone, E. Scott, and G. Economopoulos. Generalized parsing: Some costs. In *Proc. International Conf. on Compiler Construction*, volume 2985 of *LNCS*, pages 89–103. Springer-Verlag, 2004.

[9] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[10] S. Marlow and A. Gill. Happy user guide. Happy is a parser-generator for Haskell available at: `www.haskell.org/happy`.

[11] S. McPeak and G. C. Necula. Elkhound: a fast, practical GLR parser generator. In *Proc. International Conf. on Compiler Construction*, volume 2985 of *LNCS*, pages 73–88. Springer-Verlag, 2004.

[12] M. van den Brand, J. Scheerder, J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized lr parsers. In *Proc. 11th International Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 143–158, 2002.

[13] E. Van Wyk, D. Bodin, L. Krishnan, and J. Gao. Silver: an extensible attribute grammar system. In *Proc. of LDTA 2007,* 7^{th} *Workshop on Language Descriptions, Tools, and Analysis*, 2007.

[14] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142, 2002.

[15] E. Van Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for java. In *European Conference on Object Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, July 2007. To Appear.

[16] E. Van Wyk and A. Schwerdfeger. Context-aware scanning: Specification, implementation, and applications. Technical Report 07-012, Univ. of Minnesota, April 2007.

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

[18] T. A. Wagner and S. L. Graham. Incremental analysis of real programming languages. In *PLDI '97: Proc. of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 31–43, New York, NY, USA, 1997. ACM Press.