Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 07-012

Context Aware Scanning: Specification, Implementation, and Applications

Eric Van Wyk and August Schwerdfeger

April 25, 2007

Context-Aware Scanning: Specification, Implementation, and Applications — DRAFT*

Eric Van Wyk August Schwerdfeger
Department of Computer Science and Engineering
University of Minnesota
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 utilizes a slightly modified LR-style algorithm that passes to the scanner the set of valid symbols which the scanner may return at that point in parsing. This set is comprised of the terminal symbols that are valid for the current state, *i.e.*, those whose entry in the parse table are "shift," "reduce," or "accept" rather than "error." The scanner then only returns tokens in this set. Also, 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. We illustrate this approach with a declarative specification of a Java subset and extensions that embed SQL queries and Boolean expression tables into Java. We also give a test on extensions that can verify, at the time an extension is written, whether it will compose conflict-free with other extensions that pass the test.

1 Introduction

In previous work [13, 16] 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 as well as the generation of semantically equivalent host language constructs [15]. This paper addresses the specification, implementation, and composition of the syntactic aspects of extensible languages.

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* — indicating terminals which may not appear in a syntactically correct phrase at the

*This work are partially funded by NSF CAREER Award #0347860, NSF CCF Award #0429640, and the McKnight Foundation.

current state of the parser — 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. Thus, the scanner returns the expected symbol and the grammar describing type expressions is simplified.

```
class Demo {
 int demoMethod ( ) {
  List<List<Integer>> dlist ;
  int SELECT ;
  int T ;
  connection c "jdbc:derby:/home/evw/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.

Our motivation in developing these parsing and scanning algorithms, and their associated parser and scanner generation algorithms, was to parse extensible languages. An example of a program written in an extended language is shown in Figure 1. This program 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 to allow for the natural specification and the static syntax and type checking of SQL queries [13]. The `connection` construct specifies the database location as a String and lists the tables and their field names and types. This information is used for static typechecking of queries. The example query given in the `using` construct names the connection to the database on which the connection should execute.

The second extension adds *condition tables*, a construct for specifying complex Boolean conditions in a tabular format. These tables are used in specification languages such as SCR [6] and RSML^e [11] and simplify

the specification of complex Boolean conditions [19]. 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 we focus here on parsing, we will not concern ourselves further with the semantics of these extensions.

There are several aspects to this program that are difficult for traditional and other non-traditional scanning and parsing techniques, as the extended language's concrete syntax is the composition of the context free grammars defining the host language and extensions. Furthermore, 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$^-$ "==". "=" has precedence and associativity specifications for SQL expressions that are not valid for Java$^-$ assignment statements.

More generally, because extensions may be written independently of one another we must be concerned with the possibility of introducing syntactic ambiguities into the parser 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. absence of parse-table conflicts.)

- A test applied to each extension that can verify, at the time an extension is composed alone with the host language, whether the extension will not cause any conflicts when composed alongside other extensions that also pass the test.

Section 2 describes concrete syntax specifications used to generate the parsers and scanners that utilize context-aware scanning. Section 4 describes the modified LR parsing algorithm that uses the context-aware scanning algorithm described in Section 3. Section 5 shows how the specifications can be statically analyzed to guarantee that the parser and the scanner are deterministic. Section 6 describes a tool called Copper that implements these ideas. It also describes Copper's features and its performance. Section 7 describes related work and Section 8 concludes.

2 Extensible language specifications

We, along with our colleagues, 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 [16] and an extensible specification of Lustre [4], a synchronous language used in modelling safety-critical systems. In each language framework, the host language and the language extensions are written in Silver [14], 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 composed, extended language. Figures 2, 3, and 4 represent the concrete syntax (both syntactic and lexical syntax) extracted from their corresponding Silver specifications. The three specifications are combined to form the concrete specification of the extended language.

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 $H = \langle T_h, NT_h, P_h, S_h \rangle$ (consisting of a set of terminals, nonterminals, productions, and a start symbol) and a binary relation $\succ_h$ over terminals T_h. This relation 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.

Nonterminals are declared using `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; for the integer literal terminal `IntLit_t` and the string literal terminal `StrLit_t`, the regular expressions are the expected ones and written between forward slashes. If the regular expression defines a fixed string, it can be written inside single quotes, such as `'int'` on the `Int_t` keyword terminal. Fixed-string terminals such as `Int_t` and the punctuation terminals can be referenced by their lexemes in productions; as is done in the `Class` production and in the prose of this paper.

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.

BNF productions for Java$^-$ are shown last. These are self-explanatory.

Lexical precedence relation. In traditional approaches, a precedence ordering is placed on all terminal symbols so that if the regular expression for more than one terminal matches 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.

[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.

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.

We instead use the following more generalized manner of specifying precedence. A relation $\succ$ is set explicitly in the lexical specification of the language. The scanner uses this precedence relation when it matches two or more terminals to a given input. We will see in Sections 4 and 3 precisely how this relation is used; here we focus on its specification.

The relation $\succ$ is asymmetric and non-transitive, 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.

```
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 condition tables extension.

Language extension concrete syntax specs 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 4.2, there are contexts in which 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 simple union of the components of the grammars.

3 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 $regex : T \mapsto Regex$ — and a DFA is constructed as follows: (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, 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. Also, $poss(error_Q) = \emptyset$.

The scanner function *nextToken* is given in Figure 5; 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 regex is the Kleene star ($*$) of the disjunction of the regexes 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 6 contains the pseudocode for the context-aware scanner that takes the set of valid lookahead terminals as input. The set of terminals *validPoss* contains the 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 $poss(ss)$ 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 is called on the character at the current position *pos*, *pos* is incremented, and *validPoss* is computed for the new state. This continues until a state is entered for which no possible matches occur.

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 $t' \succ t \Rightarrow t' \neq t$.) Finally, the lexeme is copied from *input* and use to build the tokens returned to the parser.

This scan function 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. To convey further how this algorithm works we 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 passed and *validPoss* is set to {`GT_t`}; in the second iteration, the second

```
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.

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

Figure 5: The *nextToken* function called by the parser.

> is consumed. `GT_t` does not match any string of more than one character; *validPoss* is set to $\emptyset$, and the loop exits.

In scanning `int SELECT;` we showed that the valid lookahead set would contain the identifier terminal `Id_t` as well as `'int'` and five 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 not 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 8 we suggest a technique for requiring whitespace between terminals that addresses this issue.

```
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 6: The modified scanning function called by *nextToken*.

4 Modified LR parsing algorithm/scanner interface

We use a modified LR parsing algorithm and modified DFA-based scanning algorithm; the compile-time input thereto is generated from declarative specifications as in Figures 2, 3, and 4. Before describing our modified LR parsing algorithm, we briefly discuss traditional LR parsing.

4.1 Traditional LR parsing

In traditional LR parsing [8, 1] the input grammar is used to generate a *parse table* that drives the LR 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 identical to that of Figure 7 if one ignores lines 5–6 and replaces lines 10–23 with a traditional call to a disjoint scanner.

The parse table is indexed by pairs of 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 [8, 1].

We assume a global immutable input string *input* and the parser is a function that returns the CST corresponding to *input*. 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 accessor 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 ascertain the present parse state. Using this state and the lookahead token as indices, it will retrieve a parse action from the parse table, one of *shift*, *reduce*, *accept* or *error*, with the following effects.

- Shift: A shift action constructs a terminal CST node. The parser will move the input past the lookahead token and push it onto the parse stack paired with the shift action's parse state.
- Reduce: A reduce action constructs a nonterminal CST node. The parser will pop from the stack as many elements as there are symbols on the right-hand side of the reduce action's production. Then it will build a new CST node labeled with the production's left hand side with the freshly-popped CSTs for children. Using the state now on top of the stack and the production's left hand side nonterminal as indices, the parser will then retrieve a *goto action* (identical to a shift action) from the parse table. It will then push onto the stack the new CST and the goto action's parse state.
- Accept: An accept action stops the parser when parsing is complete.
- Error: An error action stops the parser when there is a syntax error, *i.e.*, when a lookahead token shows up for which there is no other action in the table.

4.2 Modified LR parsing

We modify the LR parsing algorithm in order to perform lexical disambiguation by context through 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-to-scanner interface.

4.2.1 Valid lookahead sets

A valid lookahead set is defined for each parse state ps as the subset of terminals that have 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 words 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.

4.2.2 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 $nextToken : \langle Set(T), int \rangle \rightarrow \langle Set(Tk), int \rangle$. 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 5.1.3) are rarely required.

4.2.3 Modified LR parsing algorithm

Figure 7 contains the pseudo-code of the modified LR parsing algorithm. The first seven lines of the function (1) initialize the parser start state, a flag *done*, and the position *pos* (to the beginning of the input), (2) declare variables for the parse state, valid lookahead terminals set and tokens returned from the scanner, and (3) push a starting state onto the parse stack.

Entering into 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. In this example, $validLA = \{$`'>'`$\}$ 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. In this example, nowhere is `'>'` incorporated into the lexical precedence relation $\succ$, and therefore $higherPrecTerms(\{$`'>'`$\}) = \emptyset$. (The definition of function *higherPrecTerms* is described below.) Line 12 calls the scanner on the present position, passing it the valid lookahead set. In our example the scanner will locate 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 contains exactly one element, and if not, it attempts to make it so by applying any disambiguation functions. If this fails 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*, `'>'`) indicates a shift action. The switch-statement occupying lines 25–40 provides cases for every kind of parse action. The code here is identical to that used in a traditional LR algorithm, as described above.

4.2.4 Other examples

Use of precedence relation $\succ$ This example shows how precedence relations are used in the parser. It occurs when parsing the program in Figure 1, at the beginning of the line `int SELECT;`. The parser is

```
function parse() returns CST
 1.  int startState = parser start state
 2.  boolean done = false
 3.  int pos = 0
 4.  int ps // current parse state
 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 code
21.        exit()
22.    // |lookAhead| = 1
23.    tk = first(lookAhead)
24.    action = table(ps, lookahead)
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 7: Modified parsing algorithm.

prepared to shift `int`, upon receipt of appropriate lookahead. At line 10, the parser retrieves *validLA*, here everything that can occur at the beginning of a statement in a method. This set contains precisely `Id_t`, `'int'`, `'Integer'`, `'String'`, `'boolean'`, `'while'`, and the connection terminal for `'connection'` from the SQL extension.

The relation $\succ$ specifies that in all contexts, $t_1 \succ t_2$ implies that 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. Let

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

Then let $validLA = validLA \cup higherPrecTerms(validLA)$: in any context in which t is valid, add all higher precedence terminals t'. This guarantees that a string matched by a higher-precedence terminal never matches a lower-precedence terminal, *even if the higher precedence terminal has no valid action.* For example, in a context where `Id_t` is valid, add all the appropriate keyword terminals. Then if a keyword showed up where only an identifier could occur, it would still be matched as a keyword and thus cause an error to be raised.

Thus, line 11 turns up some new additions to the set. 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. Their presence there prevents any string matching any of them from being matched as an `Id_t`.

For example, if the line had read "`class SELECT;`" instead of "`int SELECT;`," the new additions allow the terminal `'class'` to match, which causes a parse error as `'class'` has no parse action at that point; without them it would have matched `class` as an identifier instead of a reserved keyword. But in the actual case of "`int SELECT`", line 12 of the parser calls out to the scanner; the scanner matches the keyword `'int'` and returns it. There are no lexical ambiguities; the computation then branches to line 26, the start of taking a shift action. The parser pushes a new element on the stack containing the new parse state and the `'int'` token.

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.

Selectively reserved keywords This example, touching on the expressive power of the precedence relations, occurs directly after the conclusion of the previous example; the parser has just finished shifting `int` and is now ready to reduce it into a `Type` CST. At line 10, it retrieves *validLA* from the parse table. This consists of one element, `Id_t`. At line 11, the same eight keywords from above are added to *validLA*. At line 12, the parser calls the scanner; the scanner reads `SELECT` and matches it as an identifier. In a traditional scanner, the SQL keyword `'SELECT'` would have been matched, but `'SELECT'` $\notin$ *validLA*. Thus, this approach can implement selective keyword reservation based on grammar context. Execution then branches to line 30, the start of taking a reduce action. The production to be reduced upon is `Type ::= 'int'`. There is one symbol on the right-hand side of this production, so the algorithm pops one element off the stack; its *node* is of course `'int'`, which has just been shifted there. The `Type` CST is constructed in line 32; then the parse table is consulted. Once the new state ps' has been located it is pushed onto the stack along with the new tree.

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.

The use of valid lookahead also solves the other problems mentioned in Section 1. 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. Similarly, since the host language terminal `Assign_t` with regex `'='` is not in the valid lookahead in the context of SQL queries, but the SQL terminal `SQL_EQ_t` with regex `'='` is, the SQL terminal is returned and the effect of the traditional precedence and associativity settings on that terminal are seen.

5 Syntactic and lexical determinism

The integrated approach parsing and scanning which utilized context-aware scanning also supports two analyses used to determine if a language specification can be parsed and scanned deterministically. The first is a monolithic analysis that works on the parser and scanner specification for a complete language. It can statically verify that the scanner specification contains no lexical ambiguities with respect to the associated parser (specifically the set of valid terminals for each parser state) and thus will never return more than one token for a single input string. This analysis in combination with the standard check for conflicts in the LR parse table ensures that the parser and scanner are deterministic. The second is a modular analysis that checks a language extension specification against the specification of the host language that it extends. This analysis is a modular version of the monolithic one. It ensures that the combination of the host language and any set of extensions that pass the modular analysis will pass the monolithic analysis. This analysis is especially useful to language extension designers since it provides a guarantee that their extension will not be the cause of any lexical ambiguities or parse table conflicts in an extended language in which their extension is a component.

5.1 Monolithic determinism analysis

5.1.1 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. Our approach differs from the traditional in that we can produce more meaningful lexical tokens and hence parse a larger class of languages.

5.1.2 Lexical determinism

A scanner exhibits lexical determinism if for any input, it will match 0 or 1 terminals. Traditionally, a precedence relation is placed on all terminals, and if more than one terminal matches the same prefix of the input string, the one with higher precedence is selected. In our approach, we cannot base precedence relations on any order in the grammar specification since extensions (and the terminals thereof) are composed with the host language as an un-ordered set. Furthermore, any total ordering for precedence on all terminals in an extended language is not needed since many terminals never appear in the same valid lookahead set.

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 a DFA-state accept set ($acc(ss)$). Such a set with cardinality greater than 1 is an ambiguity. Therefore, the test is for every parse state s_p (row of the parse table), and for every state of the scanner DFA s_s, if $|validLA(s_p) \cap acc(s_s)| \leq 1$, there is no ambiguity; if it is greater than 1, it represents an lexical ambiguity.

5.1.3 Disambiguation functions

In our specifications of Java 1.4 and the various extensions there are no lexical ambiguities [16]. But in our specifications for ANSI C and AspectJ we have found a few lexical ambiguities. These can be resolved with a *disambiguation function* — a construct used to resolve a particular, otherwise unresolved, lexical ambiguity.

Our test detects if any intersection A between a valid lookahead set and a DFA-state accept set has a cardinality greater than 1. If such a set A exists, on some input string the scanner will return an ambiguous match set with a token for each terminal in A. To establish lexical determinism, a disambiguation function is then required in order to resolve A by selecting the single token to return. A disambiguation function is a pair $\langle A \subseteq T,\ f\ :\ \Sigma^\star \to A \rangle$. A is a particular ambiguity; f is a function that takes the matched lexeme (and, in practice, line and column information) and returns a token for 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 7 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.

We have used disambiguation functions to resolve C's typename/identifier ambiguity (see Section 6.1) and the lexical ambiguities in an AspectJ specification.

5.2 Modular determinsism analysis

5.2.1 Syntactic determinism

As the class of LALR(1) grammars is not closed under composition, standard methods will not work to ensure that a host grammar H and a set of extensions E_k for $k \in [1, n]$, though deterministic on their own, do not generate parse table conflicts when compiled together in some programmer-selected permutations.

It is desired that when composing several extensions, the addition of one extension E_k will not introduce shift-reduce or reduce-reduce conflicts in the parse table of a language composed from the host grammar H and several extensions $E_1, E_2, \ldots, E_k, \ldots, E_n$. This is ensured by making it incumbent upon the writer of E_k to resolve any conflicts between it and H, at the same time ensuring that should the other extensions match the same constraints, E_k will not conflict with *them* either.

The easiest way to ensure that any permutation of a set of deterministic extensions is also deterministic when applied is to separate the generated LR DFA into several distinct sub-automata, one for the host and one for each extension. This is done by mandating that each transition from a state in the host's partition to a state in an extension's partition be labeled with a unique terminal called a *marking token* that starts every expression in a particular extension.

The addition of the extension must also not affect the host's partition through addition of lookahead or uncontrolled new states, except for the addition of the items giving rise to the abovementioned transitions.

LR(1) DFA state basics

1. A state q in an LR(1) DFA [1] consists of several *items*, which we write as $I(q)$.
2. Each item i in q has attached to it a production, a bullet indicating the extent to which the parser will have parsed this production's right-hand side when in the item's state, and a set of lookahead terminals. We write the production as $P(i)$ and the lookahead set as $LA(q, i)$.

Formal definition of an extension

1. A *grammar extension* takes the form $E = \langle T^E, NT^E, P^E, H \rangle$.
2. T^E are the extension's terminals, NT^E are the extension's nonterminals, P^E are the extension's productions, and H is the host grammar being extended.
3. Productions in P^E may reference symbols from E or H; however, exactly one production therein must have on its left side a symbol from NT^H. Note that in practice, there may be more than on such production. The analysis presented here has a straightforward generalization to this case.

4. The one production with the symbol from NT^H (labeled H_P) on its left-hand side must have the form $H_P \rightarrow \mu_E H'_P$, where μ_E is the extension's marking token and H'_P is any positive-length sequence of symbols.

5. To state that an item i has a production of this form for a given extension E_k and that its bullet is at the beginning (*i.e.* $H_P \rightarrow \bullet\mu_E H'_P$) we write $entryitem(i, E_k)$.

Ownership of an LR DFA state

1. Ownership of a certain LR DFA state may be vested in the host grammar H or in exactly one of the extensions E_k.

2. The host grammar H owns a state S iff either there is no item in S with a production from an extension grammar in it, or for all such items i, $entryitem(i, E_k)$.

3. Otherwise, the production of one or more items in the state belongs to some extension E_k. In this case, the extension E_k owns the state.

4. Due to the separation the marking tokens provide, it is trivial that two extensions will not share items in the same state (one must cross a transition marked μ_E to reach a state with items belonging to E_k in it).

5. To convey grammar E_k's ownership of state s, we write $owns(s, E_k)$.

IL-subsets We now introduce the concept of the *IL-subset*. A state q' is an IL-subset of another state q iff q''s item set is a subset of q's, and for every one of q''s items, its lookahead set is a subset of the counterpart in q. Formally: $q' \subseteq_{IL} q \Leftrightarrow (I(q') \subseteq I(q) \wedge \forall(i \in I(q')).(LA(q', i) \subseteq LA(q, i)))$.

The crucial feature of IL-subsets is that if $p \subseteq_{IL} q$, and q produces a conflict-free row in a parse table, so will p.

Determinism test When a host grammar H and one extension E_k are merged and their parser compiled, the states in the resulting DFA will fall into one of three classes:

1. States belonging to E_k;

2. States belonging to H that are also present when the host grammar is compiled alone (ignoring *entryitem*s); and

3. States belonging to H that were introduced with E_k.

It is impossible for the members of the first class of states to be the source of conflicts with other extensions, as only E_k could possibly add it, and therefore if it is clear of conflicts when E_k is composed with the host, it will be clear no matter what other extensions are added.

As by definition the item sets of the second class of states have not been altered — except to add *entryitem*s — from a condition in which they are known not to introduce conflicts to any extension, the only way in which the addition of E_k could have altered them is to add terminals to the item's lookahead sets. We test to see that no lookahead has been added; if no extension does add lookahead, H's states remain the same as if it had been compiled individually with each extension, a condition in which no conflicts will arise.

The third class of states pose a slightly more difficult problem. If only one extension adds them, of course, they are in the same condition as E_k's states in that the addition of other extensions will not give rise to new conflicts in such a state; but if several extensions add them, the state's lookahead sets could spin out of control and generate a conflict.

We solve the problem in the following manner. If several extensions add the state to the LR DFA, the lookahead sets of the state's items will be the union of all those introduced when the extensions are compiled individually with the host. As the subset relation is closed under union — *i.e.* $A \subseteq C \wedge B \subseteq C \leftrightarrow A \cup B \subseteq C$ — if all the lookahead sets in the individual maps can be proven to be subsets of some lookahead set known to generate no conflicts, then their union will also generate none. In other words, a state is guaranteed deterministic if it is an IL-subset of a state of the second class.

The final test relates to the fringe case of the *entryitem*s, exempted above from the lookahead tests. As adding extensions mandates adding *entryitem*s, certain conflicts may be introduced when extensions' marking tokens are added as lookahead; these are unavoidable.

However, it is possible to write an extension so such additions will not affect its determinism. As other extensions' marking tokens are foreign to the host and the one extension being tested, there is no fundamental difference between them, and they may be regarded as one token for the purposes of determinism testing; to prepare for the addition of a new *entryitem* anywhere in the host grammar, when testing determinism with the host and one extension, a dummy marking token representing any possible extension should be added to the $first$ set of every nonterminal that can follow an extension expression.

5.2.2 Lexical determinism

The mandated partition of the DFA guarantees that the only lexical ambiguities rising from conflicts between extensions will be those between marking tokens. If the marking tokens from two extensions conflict, it is a situation that cannot be foreseen or prevented by extension writers, except if they are working in tandem.

We have developed a construct called *transparent prefixes* to address this specific problem. In the general case, a transparent prefix is a terminal p that may be specified as an attribute of a grammar terminal t. Then, any time t is in the valid lookahead set, before starting the scan for t the scanner will scan for p. If it matches p, the valid lookahead set will then be narrowed to only those terminals with p as a transparent prefix.

Now if p is different for each grammar — some variation on the grammar name, perhaps — then all that must be done for a programmer to differentiate between marking tokens is to put a marking token's prefix before it in his or her program.

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 is a metasyntax specifying both lexical and context-free syntax similar to those 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 [9, 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. An example of a place where this is useful is in the grammar of C. Typenames and identifiers in C share the same regex, and furthermore they may often occur in the same context. It is not possible to use Java's approach (one terminal type in both cases) as this introduces conflicts to the grammar. This ambiguity is usually resolved by maintaining a list of encountered `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` 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 approach 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.

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 in such a way as to be retrievable 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 theoretically 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 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 $rej : Q \mapsto P(T)$ 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 7 is modified by removing line 11 and the *scan* function of Figure 6 is modified by removing line 14 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.

7 Related Work

There is a vast amount of research on parsing and scanning techniques and we cannot discuss them all. Instead we describe some techniques that are also applicable extensible languages, 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 parse even ambiguous grammars. GLR has been greatly improved recently. Johnstone *et al.* [7] 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 [10] 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.

One major advantage of a nondeterministic system is that it allows for the complete separation of semantic analysis from parsing. The disambiguation of C typenames and identifiers, for example, could 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.

Eelco 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], which include:

- *Follow restrictions* to replace maximal munch (*e.g.*, a number cannot be directly followed by a digit).
- *Reject productions* to replace "prefer keywords" (in the identifier/`typedef` example, one would write $Id \rightarrow$ '`typedef`$'$, $\{reject\}$, which is analogous to our lexical precedence $\succ$).
- *Preference attributes*, which are identical to reject productions, except that the "non-preferred" or "avoided" production is not removed except in the presence of the "preferred" production.

  For example, to implement a *non-reserved* keyword `typedef`, one could write something such as: $Stmt \rightarrow$ '`typedef`$'$ and $Stmt \rightarrow Id$, $\{avoid\}$ This will only cause the $Stmt \rightarrow Id$ to match when there is no alternative (*i.e.*, when `typedef` has not been matched).

Operator precedence and associativity are still present, with the traditional effects, although specified by production rather than terminal. However, it still makes no guarantee of determinism, being based upon the GLR engine.

Aycock and Horspool: Schrödinger's Token Another less brazenly nondeterministic approach to this problem is the "Schrödinger's token." Aycock and Horspool [2] present non-reserved 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. The following properties apply to an individual Schrödinger's token: (1) it consists of several numbered terminal/lexeme pairs; (2) 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; (3) the content of its lexemes is not regulated except by its position in a sequence of one or more tokens. 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. A sequence of Schrödinger's tokens must represent the same section of the input and thus represents several scans thereof.

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 to compile *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." [3, 5] All constructs in PEGs are also greedy [5].

Being completely deterministic, they are closed under composition. Grimm [5] has written a packrat parser generator known as *Rats!* specifically for use with extensible languages, which outpaces two GLR

parse engines — Elkhound and SDF2. (It is outpaced, however, by two LL(k) parse engines, and no tests were performed against LALR(1) engines.)

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. It also includes support for constructs called *syntactic predicates*, which act as nonregular lookahead, in that they match expressions without consuming any input, and control what may appear after a certain grammatical construct. 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. The very idea of an ordered choice contradicts the concept of modular extensions. As extensions work presently by injecting new productions onto existing nonterminals in context-free grammars, so must they work with PEGs by injecting new choices. However, our extensions are intended to be insensitive to the order in which they are added. PEGs do not support this; therefore, the forced ordering requires that the writer of each extension know about all other extensions, so as to resolve this ordering issue.

8 Conclusion

In this paper we have described deterministic parsing and scanning algorithms in which the parse state is used by the scanner to disambiguate lexical syntax. This results in a system that 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 lesson here 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 parser state is used by the scanner to be more discriminating, there is a nice cooperation between the two.

Aho, Sethi, 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.

Future and Ongoing Work We are also interested in a 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 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 [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 3 by adding explicit whitespace between the 'for' and identifier terminals on the for-loop production.

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

[4] 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.

[5] 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.

[6] C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.

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

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

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

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

[11] J. M. Thompson, M. P. Heimdahl, and S. P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, September 1999.

[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, and P. Huntington. Adding syntax and static analysis to libraries via extensible compilers and language extensions. In *Proc. of LCSD 2006, Library-Centric Software Design*, 2006.

[14] 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.

[15] 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.

[16] 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.

[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: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 31–43, New York, NY, USA, 1997. ACM Press.

[19] M. K. Zimmerman, K. Lundqvist, and N. Leveson. Investigating the readability of state-based formal requirements specification languages. In *Proc. 24th Conf. on Software engineering*, pages 33 – 43, Orlando, Florida, May 2002. ACM Press.