Scannerless NSLR(1) Parsing of Programming Languages

Daniel J. Salomon & Gordon V. Cormack
Department of Computer Science, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
djsalomon@waterloo.edu & gvcormack@waterloo.edu

ABSTRACT

The disadvantages of traditional two-phase parsing (a scanner phase preprocessing input for a parser phase) are discussed. We present metalanguage enhancements for context-free grammars that allow the syntax of programming languages to be completely described in a single grammar. The enhancements consist of two new grammar rules, the exclusion rule, and the adjacency-restriction rule. We also present parser construction techniques for building parsers from these enhanced grammars, that eliminate the need for a scanner phase.

1. Introduction

Conventional wisdom tells us that to write a compiler we should split parsing into two phases: lexical analysis by an finite-state scanner and syntax analysis by a pushdown parser. Unfortunately both compiler writers and compiler-compiler writers suffer in this scheme. The compiler writer suffers in that he must partition the grammar for the programming language that he is implementing into two interelated grammars, and he must design an interface between the two phases. In addition, it is not always clear how much semantic analysis should be done in the scanner; specifically how much processing of literal constants should be done. (In attribute grammars this problem corresponds to devising a way to pass attributes between the two grammars.)

The designer of a compiler-compiler also faces difficult design decisions. He must choose what metalanguages are to be used by the scanner generator and the parser generator, implement two separate automata generators, design the form of the interface between them, and prepare user documentation for the two metalanguages and the interface.

The compiler writer attempting to prepare a complete grammar for a programming language, also suffers from the constraints of context-free grammars. BNF has only one type of rule, the production, and only one operator, the alternation bar. EBNF has the added features of square brackets for optional phrases and curly braces for repeated phrases. Tests on grammars for Pascal, Modula-2, and Ada have shown that due to these simple enhancements the same grammar can be expressed in EBNF using from 20% to 43% fewer rules than BNF. But these improvements are not enough, and due to the large size and complexity of the grammars that would be required to fully describe the syntax of existing programming languages, most programming-language-definition manuals omit parts of the syntax for brevity. Typical examples of omissions are:

1) the syntax of comments,
2) a description of what constitutes white space (usually blanks, tabs, new-line characters, and comments),
3) a description of where white space may appear, and where it is required,
4) a description of identifiers that excludes the reserved words, and
5) a description of how reserved words embeded in identifiers should be treated (henceforth called the *longest-match ambiguity*).

For examples of these shortcomings see the definitions of Pascal,[9] Modula-2,[16] and Ada.[1] These descriptions are omitted even though they are context-free.

The compiler-compiler writer does benefit from the limited nature of context-free grammars in that it has facilitated the theory of automatic parser generation. But the grammars provided in language-definition manuals typically require more sophisticated lookahead schemes than are afforded by conventional LL(1) and LR(1) parsers. The result is that the grammars must be carefully rewritten before they can be processed by the common compiler-writing tools.

The problem addressed in this paper is twofold. First, we want a succinct and unambiguous notation to specify the complete character-level syntax of a programming language, including comments and white space. Second, we want to be able to construct an efficient parser from such a notation. Our proposal for a more powerful metalanguage consists of enhancing context-free grammars with two new restrictive rules, the exclusion rule, and the adjacency restriction. The technique proposed is to write compact but ambiguous context-free grammars, augmented by restrictive rules that disambiguate them. To write unambiguous grammars using only productions would require awkward grammar transformations, resulting in an exponential size increase (thousands more rules for Pascal-like languages), and convoluting of the grammar's structure so as to

compromise its utility as a basis for semantic analysis and translation.

To provide a parser generator for our scheme, we propose an enhanced version of the noncanonical SLR(1) parser generator presented by Tai,[14] and describe how it can be modified to process the proposed new restrictive rules. We have tested our proposals by implementing the parser generator that we describe, and by preparing a complete and unambiguous description of ISO Pascal that has been successfully processed by our parser generator. Our parser generator also provides the basis for a new Modular Attribute Grammar system.[7]

2. Previous Work on Unified Metalanguages and Disambiguation

DeRemer[6] proposed that the scanner and the parser could be described by the same metalanguage, a context-free grammar, while keeping a multiphase parser. He deals with the keyword-identifier ambiguity problem by introducing another processing phase, called a *screener*, between the scanner and the parser for recognizing and reclassifying reserved identifiers as keywords. He does not, however, deal with other ambiguity problems in the scanner description, nor does he discuss the problem of interfacing the three phases.

Krzemien and Lukasiewicz[10] discuss the automatic extraction of the FSM scanner phase from a unified grammar. They do not, however, discuss the treatment of ambiguities in grammars.

Lalonde,[11] Baker,[4] and Bermudez and Schimpf[5] have all proposed powerful parsers for dealing with complex lookahead problems. The goal of parsing complete character-level grammars with these parsers is seldom discussed explicitly, but all seem reasonably well suitable for that purpose. Their parsers are fairly complex, however, and none of those authors deal with the common ambiguity problems.

The scanner generator LEX[12] is intended to be used in a two-phase parser and makes no attempt to unify metalanguages for the two phases. It does, however, have implied rules for dealing with ambiguities: the longest match and the earliest rule take precedence, in that order. In an LR parser this disambiguation strategy would have the effect of ignoring all shift-reduce and reduce-reduce conflicts, thus concealing unintentional ambiguities. On the other hand, reporting the conflicts would generate hundreds of messages that must each be carefully checked for correctness. This ad-hoc approach to disambiguation lacks any declarative specification of what *language* is accepted by the parser—the language accepted is determined operationally by means of the particular parsing method.

Aho, Johnson, and Ullman[2] discuss parser generation with disambiguation rules, but their notation is specialized to the operator-precedence problem and other specific problems, and is inadequate for disambiguating complete character-level grammars.

We propose that parsers for practical complete character-level grammars can be automatically generated by using an improved version of the NSLR(1) parser generator of Tai[14] to solve the lookahead problems, and that two kinds of disambiguation rules be used to resolve grammar ambiguities.

3. Noncanonical SLR Parsers

Tai presents a fairly simple enhancement to SLR parsers and parser generators that provides enhanced lookahead capabilities. Instead of only terminals as lookahead symbols, nonterminals are used as well.

An NSLR parser uses two stacks, a state stack and a symbol stack (see figure 1).

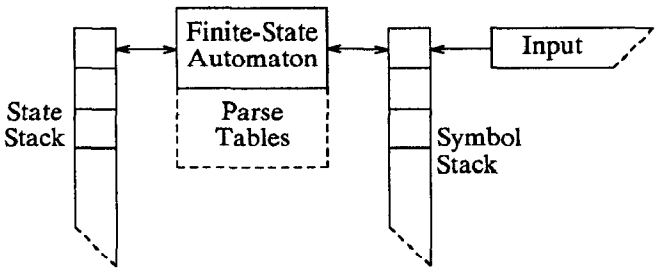

Figure 1. NSLR parsing automaton

The four usual parser actions, *shift*, *reduce*, *error*, and *accept* are used, but *shift* may also shift non-terminal symbols, and *reduce* is redefined so that the reduced symbol, instead of being shifted immediately, is pushed back onto the symbol stack to serve as lookahead for the next parser action. The NSLR(1) parsing algorithm contains the LR(1) algorithm in that it can execute ordinary LR(1) parse tables if the *goto* and *shift* action tables are merged. The NSLR(1) parsing algorithm is shown in figure 2.

```
state_stack  ← start state
symbol_stack ← empty
loop
   if symbol_stack = empty then
      let s = next input symbol
      push s on symbol_stack
   end if
   let x = top(state_stack)
   let y = top(symbol_stack)
   case action[x,y] in
      shift n
         pop symbol_stack
         push n on state_stack
      reduce A → α
         pop |α| symbols from state_stack
         push A on symbol_stack
      accept
         exit loop and accept
      error
         exit loop and reject
   end case
end loop
```

Figure 2. NSLR parsing algorithm.

The parser generator works by initially generating a standard SLR parser, state by state. When a conflict arises in a state, a *state-expansion* algorithm is invoked. All conflicts involve a lookahead symbol for a reduce action, so the parser generator resolves conflicts when it can by eliminating the offending lookahead symbols, and adding items which derive the offending symbols. This has the effect of generating shift actions for the conflicting lookahead symbols so that they can be shifted onto

the state stack, reduced based on further lookahead, and the reduced symbol pushed back onto the symbol stack to serve as higher-level lookahead. For many grammars, this reduced symbol provides the extra lookahead information needed to resolve the original conflict. This extra lookahead is achieved while preserving the linear time and space consumption characteristics of LR parsers.

To make NSLR(1) parsers useful for parsing single-phase grammars, some corrections and enhancements to Tai's algorithm were needed.

1) The lookahead sets of all states must be computed to include nonterminals as well as terminals. In the published algorithm, only lookahead sets for expanded states contain nonterminals, which produced incorrect parsers for some grammars.
2) The state expansion algorithm must be improved to avoid introducing irrelevant ϵ-reducing items. Tai's algorithm generates too many such items, introducing spurious conflicts.
3) Special treatment is needed for *invisible* symbols, symbols that cannot produce terminal symbols, either directly or indirectly.

Our version of the NSLR(1) construction, shown in figure 3, requires the prior computation of the following tables. (We follow the notation of Aho, Sethi, and Ullman[3] and a summary of this notation appears at the end of this paper.)

VISIBLE $= \{X \mid X \stackrel{*}{\Rightarrow} x$ for some $x \neq \epsilon\}$

FIRST$(X) = \{Y \mid X \stackrel{*}{\Rightarrow} Y\alpha$ for some $\alpha\}$

LAST$(X) = \{Y \mid X \stackrel{*}{\Rightarrow} \alpha Y$ for some $\alpha\}$

FOLLOW$(A) = \{X \mid B \Rightarrow \alpha Y \beta Z \gamma$ for some B, A is in LAST(Y), $\beta \stackrel{*}{\Rightarrow} \epsilon$, and X is in FIRST$(Z)\}$

NEEDED_FOLLOW$(A) = \{X \mid B \Rightarrow \alpha Y \beta X \gamma$ for some B, A is in LAST(Y), and $\beta \stackrel{*}{\Rightarrow} \epsilon\}$

UNRESOLVABLE$(A) = \{X \mid X$ is in VISIBLE, $A \Rightarrow \alpha X \beta$, $\alpha \neq \epsilon$ and $\alpha \stackrel{+}{\Rightarrow} \epsilon\}$

When the algorithm of Figure 3 marks a conflict as unresolvable, it may still be possible to resolve the conflict for some grammars by adding ϵ-reducing items and more shift items. We have judged that the extra class of grammars that would be accepted is not sufficiently important to justify the added complexity of the construction algorithm.

```
Loop until all states completed.
   Compute next SLR state q for grammar.
   To each complete item I_i = [A → α•] attach
      a lookahead set L_i = FOLLOW(A) ∩ VISIBLE
      and a needed-lookahead set
      NL_i = NEEDED_FOLLOW(A) ∩ VISIBLE.
   If state q has a conflict then
      For each conflicting lookahead symb X:
         resolved := true
         For each complete item I_i = [A → α•]
                  such that X is in L_i:
            If X is in NL_i or
                  X is in UNRESOLVABLE(B)
                  for some B in L_i
               then resolved := false
            else
               For each rule B → Xβ where
                  B is in L_i:
                  Add item [B → •Xβ] to q
               end for
            end if
         end for
         If resolved then remove X from L_i
         else grammar is not NSLR(1).
         end if
      end for
   end if
end loop.
```

Figure 3. NSLR(1) Construction algorithm.

4. Restrictive Grammar Rules

Our proposal is that a character-level grammar can be prepared by writing a concise but ambiguous BNF grammar, which uses restrictive rules to eliminate the ambiguity. We have found that two types of restrictive rules are adequate. The first rule, the exclusion rule, is used to resolve the reserved-identifier ambiguity and the second, the adjacency-restriction rule, solves the longest-match ambiguity as well as other ambiguity problems.

4.1. The Exclusion Rule

The form of an exclusion rule is the same as the form of an ordinary CFG production except that the operator "$\rightarrow$" is replaced by the operator "$\not\rightarrow$". In a BNF grammar the symbols " ::≠ " can be used as the exclusion operator. If a grammar must be represented in the ASCII character sets, then the character sequences "-X->", or " : :#" can be used for the exclusion operator.

An exclusion rule such as

id $\not\rightarrow$ *begin*

can be read as "*id* does not generate *begin*". It indicates that the language generated by the left part (*id*) should not include the language generated by the right part (*begin*). It would be used with a general description for *id*, to exclude the generation of specific sentential forms by *id*. Just as with ordinary BNF rules, multiple exclusion rules with the same left part can be joined by combining the right parts separated by OR bars "|". E.g.

id $\not\rightarrow$ *begin* | *end* | if | *then* | *else* | $\cdots$

More formally, if $L(A)$ is the sublanguage generated by the nonterminal A, and $L(\alpha)$ is the sublanguage generated by the string α, the exclusion rule $A \not\rightarrow \alpha$ restricts A to represent the sublanguage $L(A) \cap \overline{L(\alpha)}$.

Since CFL's are not closed under complementation, it is not possible to give a general construction for a CFG equivalent to the restricted grammar. If we restrict $L(\alpha)$ to being a regular set, the required algorithms exist in the literature (see for instance Hopcroft and Ullman[8]). Note that the language defined by a grammar using an exclusion rule, is still well defined, even if the language is not context-free.

4.2. Implementing the Exclusion Rule

Our construction algorithm builds the NSLR parser for the unrestricted ambiguous grammar, and uses the exclusion rules to eliminate actions that would parse the excluded sublanguages. The method consists of two modifications to the parser generator, and two tests for

membership in the class of grammars that are handled correctly.

Definition. The set DESCEND(A) contains the immediate descendants of A.

$$\text{DESCEND}(A) = \{X \mid A \Rightarrow \alpha X \beta \text{ for some } \alpha, \beta\}$$

Definition. The set DESCEND*(A) contains all symbols that can appear in any sentential form generated by A.

$$\text{DESCEND}^*(A) = \{X \mid A \stackrel{*}{\Rightarrow} \alpha X \beta \text{ for some } \alpha, \beta\}$$

Definition. The uniquely-LAST function, U_LAST(A), is the set of symbols that only appear as the last symbol of sentential forms generated by A.

$$\text{U\_LAST}(A) = \{Y \mid A \stackrel{+}{\Rightarrow} \beta Y \text{ for some } \beta, \text{ and every derivation } S \stackrel{*}{\Rightarrow} \alpha\beta Y\gamma \stackrel{*}{\Rightarrow} z \text{ is isomorphic to a derivation } S \stackrel{*}{\Rightarrow} \alpha A\gamma \stackrel{+}{\Rightarrow} \alpha\beta Y\gamma \stackrel{*}{\Rightarrow} z\}$$

For example, in the mini grammar at the end of this paper, *alpha_num* is in U_LAST(*id*). Figure 4 gives an algorithm for computing U_LAST(A).

$$\text{END\_SYM}(B, X) = \begin{cases} \textbf{true} & \text{if } B \Rightarrow \alpha X \\ \textbf{false} & \text{otherwise} \end{cases}$$

Compute the transitive closure END_SYM$^+$.

$$\text{ALT\_DESCEND}(B, Y) = \begin{cases} \textbf{true} & \text{for } Y \text{ in DESCEND}(B) \\ & \quad \text{and } B \neq A. \\ \textbf{false} & \text{otherwise} \end{cases}$$

Compute the transitive closure ALT_DESCEND$^+$

Initially U_LAST(A) = $\{X \mid \text{END\_SYM}^+(A, X)\}$.

```
For each rule A → αY:
    For each symbol X in α:
        U_LAST(A) := U_LAST(A) − DESCEND*(X).
    end for
end for

For each B such that ALT_DESCEND+(S, B) is
          true, where S is the start symbol:
    Remove B from U_LAST(A).
end for
```

Figure 4. An algorithm for computing U_LAST(A).

To implement the processing of exclusion rules, the following modifications are made, and tests added, to the parser generator:

Modification 1: For the rule $E \twoheadrightarrow\!\!\!\!\!\!\times\ F$, reduce-reduce conflicts between two items $I_i = [C \rightarrow \gamma X \cdot]$ and $I_j = [D \rightarrow \delta Y \cdot]$, where C is in $\{E\} \cup U\_LAST(E)$ or X is in U_LAST(E), and D is in $\{F\} \cup U\_LAST(F)$ or Y is in U_LAST(F), are resolved in favor of the second item, the reduction to D. The conflict resolution is made before each SLR state is tested for consistency. Specifically, for each X in L_j delete X from L_i, and for each Y in NL_j delete Y from NL_i. The effect is to prefer chained reductions that will yield the symbol on the right part of the exclusion rule rather than the left part.

Modification 2: For each exclusion rule $E \twoheadrightarrow\!\!\!\!\!\!\times\ F$, add the specially marked rule $E \rightarrow F$. These rules will be handled like any other by the parser generator except that complete items $[E \rightarrow F \cdot]$ have empty lookahead sets. Without these additions, the parser may not recognize that a string in $L(F)$ has occurred, where only a string in $L(E)$ is expected according to the grammar.

Test 1: This test enforces the restriction on the grammar that the right part of the exclusion rule must be a single nonterminal.

Test 2: The condition tested for in modification 1 ensures that the left context on the state stack represents the end of sentential forms generatable by E and F, but it does not verify that recognition of the two sentential forms starts in the same state. If recognition does not start simultaneously, then the modification may exclude valid parses, and the grammar should be rejected. The simplest way to test for this condition is to verify that for any two parses of E and F, either the parses are distinct, or that they start simultaneously. To ensure this property we verify that

(a) there are no states in the parser that contain an item of the form $[A \rightarrow \alpha \cdot E\beta]$ and (i) an item of the form $[B \rightarrow X\gamma \cdot Y\delta]$ where B is in DESCEND*(F), or (ii) an item of the form $[B \rightarrow \gamma \cdot]$ where B is in DESCEND*(F) and B is not in $\{F\} \cup$ U_LAST(F).

(b) Same as (a) above but exchange E and F.

These tests should be applied after state processing is complete, including state expansion, if any.

The above method accepts only a restricted class of grammars, which we call SE (*simple exclusion*) grammars, and which we have found adequate to describe programming languages. The adjective *simple* is used to describe this class of grammars because membership can be tested by examining each state of the generated parser individually without regard to the transitions between states. This property is shared by SLR grammars.

4.3. The Adjacency-Restriction Rule

The adjacency-restriction rule takes the form of a list of predecessor symbols, followed by the delimiter " $\not\dashv$ ", followed by a list of successor symbols. If the grammar is to be represented using the ASCII character set, the sequence of characters "-/-" can be used to represent the adjacency-restriction delimiter.

A sample usage of the adjacency restriction rule would be:

id begin do else ... $\not\dashv$ *id begin do else ...*

This example indicates that a string generated by one of the symbols in the predecessor list may not appear immediately adjacent to a string generated by one of the symbols in the successor list in any sentence in the language. The sample rule above indicates that an identifier or keyword may not abut another identifier or keyword. (If such symbols did abut each other, they should of course be parsed as a single identifier.)

More formally stated, the adjacency restriction $W \not\dashv X$ disallows all derivations of the form $S \stackrel{*}{\Rightarrow} \alpha W X \gamma \stackrel{*}{\Rightarrow} z$. Furthermore, the restriction excludes all derivations that would produce the same parse tree as a derivation of the above form.

Figure 5 gives a grammar transformation for constructing a pure context-free grammar from a restricted context-free grammar containing an adjacency restriction. The existence of this transformation demonstrates

that context-free grammars are closed under adjacency restrictions. Grammars G_1 and G_2 show a sample application of the transformation.

- Given a restricted CFG G' constructed from a CFG G by adding the restrictive rule $W \not\rightarrow X$, construct a CFG G'' such that $L(G'') = L(G')$.
- Initially $G'' = G$.
- Rewrite G'' to eliminate all ϵ-productions. (See Hopcroft and Ullman[8] in section 4.4, page 90.
- For each symbol C in $\text{LAST}^{-1}(W)$:
  - For each occurrence of C in a production P_i of G'':
    - Replace P_i by two productions, one using C_W in place of C, and the other using $C_{\overline{W}}$.
- For each C in $\text{FIRST}^{-1}(X) \cup \text{FIRST}^{-1}(X_W) \cup \text{FIRST}^{-1}(X_{\overline{W}})$:
  - For each occurrence of C in a production P_i of G'':
    - Replace P_i by two productions, one using C^X and one using $C^{\overline{X}}$.
- For each new symbol S_B^A whose basic part is S (the original start symbol), add a new rule $S \rightarrow S_B^A$ to G''. S is still the start symbol for grammar G''.
- Delete all productions using symbols whose basic part is W and whose subscript part is $\overline{W}$. Similarly, delete all productions using symbols whose basic part is X and whose superscript part is $\overline{X}$.
- Delete all productions of the form $C_W \rightarrow \alpha Z_{\overline{W}}$, $C_{\overline{W}} \rightarrow \alpha Z_W$, $C^X \rightarrow Z^{\overline{X}}\alpha$, or $C^{\overline{X}} \rightarrow Z^X\alpha$.
- Delete all productions of the form $C_W \rightarrow \alpha Y$ for Y not subscripted by W unless $C = W$. Similarly delete all productions $C^X \rightarrow Y\alpha$ for Y not superscripted by X unless $C = X$.
- Delete all productions $C \rightarrow \alpha Y_W Z^X \beta$. This is the main step of this algorithm; applying the principle of adjacency-restriction rule.
- Remove all useless symbols and productions.

Figure 5. A Grammar Transformation for Eliminating Adjacency-Restriction Rules

---

S → *oper* | *ID* | *S white oper* | *S white ID*
ID → *id*
id → *letter* | *id letter*
letter → **a** | **b** | **c** | ... | **z**
white → ϵ | **blank**
oper → * | / | + | −

ID $\not\rightarrow$ *ID*

Grammar G_1. A sample restricted CFG.

$S \rightarrow S_{ID}^{ID} \mid S_{\overline{ID}}^{ID} \mid S_{ID}^{\overline{ID}} \mid S_{\overline{ID}}^{\overline{ID}}$

$S_{ID}^{ID} \rightarrow ID_{ID}^{ID} \mid S_{ID}^{ID}\ white\ ID_{ID}^{ID} \mid S_{\overline{ID}}^{ID}\ white\ ID_{ID}^{ID} \mid S_{\overline{ID}}^{ID}\ ID_{ID}^{ID}$

$S_{\overline{ID}}^{ID} \rightarrow S_{ID}^{ID}\ white\ oper \mid S_{\overline{ID}}^{ID}\ white\ oper \mid S_{ID}^{ID}\ oper \mid S_{\overline{ID}}^{ID}\ oper$

$S_{ID}^{\overline{ID}} \rightarrow S_{ID}^{\overline{ID}}\ white\ ID_{ID}^{ID} \mid S_{\overline{ID}}^{\overline{ID}}\ white\ ID_{ID}^{ID} \mid S_{\overline{ID}}^{\overline{ID}}\ ID_{ID}^{ID}$

$S_{\overline{ID}}^{\overline{ID}} \rightarrow oper \mid S_{ID}^{\overline{ID}}\ white\ oper \mid S_{\overline{ID}}^{\overline{ID}}\ white\ oper \mid S_{ID}^{\overline{ID}}\ oper \mid S_{\overline{ID}}^{\overline{ID}}\ oper$

$ID_{ID}^{ID} \rightarrow id$

id → *letter* | *id letter*
letter → **a** | **b** | **c** | ... | **z**
white → **blank**
oper → * | / | + | −

Grammar G_2. The result of applying the adjacency-restriction-elimination algorithm to grammar G_1.

4.4. Implementing Adjacency Restrictions

The adjacency-restriction rule could be implemented in a parser generator by first applying the transformation of Figure 5 to the input grammar, but this would usually lead to an excessively large parser. Instead, we implement it by eliminating parser actions from the parser generated by the unrestricted grammar. For an NSLR parser generator we make two modifications to the the parser generation algorithm, and then apply three tests for cases where the method fails.

Modification 1: Eliminate reductions that clearly violate the adjacency restrictions. In an SLR or NSLR parser generator, the lookahead set for reduce actions of the form $A \rightarrow \alpha$ is the set $\text{FOLLOW}(A)$. In the SAR generator, the lookahead set is replaced by $\text{AR\_FOLLOW}(A)$ which is defined as:

$\text{AR\_FOLLOW}(W) = \{X \mid B \Rightarrow \alpha_0 Y_0 \beta Z_0 \gamma_0$ for some B,
$\beta \stackrel{*}{\Rightarrow} \epsilon$, and there exist derivations
$\alpha_0 Y_0 \Rightarrow \alpha_1 Y_1 \Rightarrow \cdots \alpha_n Y_n$ where $W = Y_n$
$Z_0 \gamma_0 \Rightarrow Z_1 \gamma_1 \Rightarrow \cdots Z_m \gamma_m$ where $X = Z_m$
such that no rule $Y_i \not\rightarrow Z_j$ exists for any $i, j\}$

The function $\text{NEEDED\_FOLLOW}(A)$ is replaced by
$\text{AR\_NEEDED\_FOLLOW}(A) = \{X \mid$
$B \Rightarrow \alpha_0 Y_0 \beta X \gamma$ for some B and $\beta \stackrel{*}{\Rightarrow} \epsilon$, and
there exists a derivation
$\alpha_0 Y_0 \Rightarrow \alpha_1 Y_1 \Rightarrow \cdots \alpha_n Y_n$ where $A = Y_n$
such that no rule $Y_i \not\rightarrow X$ exists for any i $\}$

The above definition of AR_FOLLOW includes only symbols that arise from at least one derivation that does not violate the adjacency restrictions. Note that this modification precludes some parser implementations that add spurious elements to lookahead sets; for example, the use of default reduce actions in YACC. In fact default reductions may be used except for actions of the form *reduce* $A \rightarrow \alpha$, where some exclusion rule $A \not\rightarrow X$ exists. Figure 6 gives an algorithm for computing AR_FOLLOW and AR_NEEDED_FOLLOW.

For each rule $B \rightarrow \beta$:
  For each j in the range $|\beta| > j \geq 1$ in descending order, (notice that when $\beta = \epsilon$ this loop is skipped):
    For each k in the range $j < k \leq |\beta|$ in ascending order, stopping after processing the first non-nullable $\beta[k]$:
      ProcessAdjacent($\beta[j]$, $\beta[k]$, $\emptyset$, $\emptyset$).

PROCEDURE *ProcessAdjacent*(W, X, AW, CAR)
Where:
  W, X = A pair of adjacent symbols.
  AW = Set of ancestors of W in current expansion.
  CAR = Collected adjacency restrictions,
    = $\{Z \mid \exists$ a restriction $Y \not\rightarrow Z$ and Y is in $AW\}$.
If W is not in AW then:
  $CAR' = CAR \cup \{Y \mid \exists$ a rule $W \not\rightarrow Y\}$.
  If X is not in CAR' then:
    If X is in VISIBLE then:
      Insert X into AR_NEEDED_FOLLOW(W).
    ProcessSuccessor(W, X, $\emptyset$, CAR').
    For each Y such that there is a rule $W \rightarrow \alpha Y$:
      ProcessAdjacent(Y, X, $AW \cup \{W\}$, CAR').

PROCEDURE *ProcessSuccessor*(W, X, AX, CAR)
Where:
  W, X, and CAR have the same meaning as in *ProcessAdjacent*.
  AX = Set of ancestors of X in current expansion.
If X is not in AX then:
  If X is in VISIBLE then:
    Insert X into AR_FOLLOW(W).
  For each Y such that $X \rightarrow Y\alpha$ is in P:
    If Y is not in CAR then:
      ProcessSuccessor(W, Y, $AX \cup \{X\}$, CAR).

Figure 6. Computing the adjacency-restricted follow sets AR_FOLLOW **and** AR_NEEDED_FOLLOW.

Modification 2: It is a property of LR parsers that any given state is reachable only by a *shift* action on a particular symbol. That is, for some unique Y, each item in S is either of the form $[C \rightarrow \alpha Y \cdot \beta]$ or $[B \rightarrow \cdot\, \delta]$. The SAR construction eliminates items of the form $[B \rightarrow \cdot\, X\delta]$ unless both B and X are in AR_FOLLOW(Y), and items of the form $[B \rightarrow \epsilon \cdot]$ unless unless B is in AR_FOLLOW(Y).

Test 1: For each adjacency restriction $W \not\rightarrow X$, W must be a nonterminal, and X may not be nullable. The principal mode of action of the above modifications is to block reductions to symbols on the left side of adjacency restrictions. If such symbols are terminals, those reductions cannot be blocked. If symbols on the right side of adjacency restrictions are nullable, then the method used to block the reductions fails, since the lookahead symbol can be part of context beyond the nullable symbol.

Test 2: For each adjacency restriction $A \not\rightarrow X$, and for each complete item $[A \rightarrow \alpha \cdot]$ with lookahead set L, Ensure that $L \cap$ FIRST(X) = $\emptyset$. This ensures that A cannot be reduced with lookahead symbols that could later be reduced to X.

Test 3: For each adjacency restriction $A \not\rightarrow X$, and each state q containing a complete item $[B \rightarrow \beta \cdot]$, with a lookahead set L, and with B in LAST(A), ensure that q does not also contain an item $[C \rightarrow \cdot\, \gamma]$ added during noncanonical state expansion, such that C is in L and X is in FIRST(γ), or $\gamma \stackrel{*}{\Rightarrow} \epsilon$. Parsers that violate this constraint may allow right context beginning with X (which does not appear in L) to be reduced to an ancestor symbol that does appear in L before the reduction of left context to C. Only noncanonical expansion items can lead to the reduction of right context before left context.

Because of the above tests, our method accepts a restricted class of grammars, which we call SAR (*simple adjacency-restriction*) grammars; we have found SAR grammars adequate for the programming-language examples we have tried. Again the adjective *simple* is used to describe this class of grammars because membership in the class can be tested by examining each state of the generated parser individually without regard to the transitions between the states.

5. The Efficiency of Single-Phase Parsers

A common objection to eliminating finite-state scanners, and using a pushdown automaton to parse raw input is that parsing would be slower. Actually, theory tells us that a finite-state machine, a deterministic pushdown automaton and an NSLR parser all run in linear time proportional to the length of the input, hence their performance can only vary by a constant factor. The size of that constant factor can vary greatly with implementation details, and can be accurately determined only with actual tests on production-quality systems. We have performed some tests that attempt to be as fair as possible.

Our first test consisted of writing three two-phase Pascal recognizers using the UNIX utilities LEX and YACC. These two utilities are both mature systems widely used for educational purposes and often used in production compilers. The first recognizer used the standard approach of a finite-state scanner generated by LEX, with a DPDA parser generated by YACC. Although LEX is a mature program, there are complaints that it produces slow scanners, so our second recognizer used FLEX,[13] a more recent version of LEX designed to generate faster scanners. Our third recognizer described the tokens in terms of a BNF grammar, and used YACC to generate the scanner as well as the parser. Ambiguities in the scanner grammar were handled by ad-hoc methods. Table 1 shows the results of these tests. The surprising result is that the running-time constant for a DPDA-DPDA recognizer can be as little as one seventh of that for an FSM-DPDA recognizer, and two thirds of the size.

Our second test was designed to compare two-phase recognizers with one-phase recognizers. The two-phase recognizer used two restricted CFG's to describe Pascal, and the one-phase recognizer used a single restricted CFG. All parse tables were generated using our NSLR(1) parser generator. Table 2 summarizes results on the relative sizes of the parsers. The results of Table 2 are not as favorable as those of Table 1, but they still show single-phase parsing to be a viable technique. Note, however, that we did not perform any table compression on the resulting parsers, and we believe that a single-phase parser will benefit more from table compression than will a two-phase parser.

At this writing, a comparison of running times of the two parsers is not available principally due to difficulties in designing the interface between the two phases of the two-phase recognizer. When timing test are performed, the results will not be directly comparable with the LEX and YACC generated parsers, since the programmers, source languages, and target languages are all different. It is hoped however that the high degree of similarity

	Rules	States	Object Size (Bytes)	Parse Time* (Sec.)
LEX–	97	545	21.9k	
YACC	204	369	6.8k	
Total	301	914	28.7k	48.2
FLEX–	57	625	47.0k	
YACC	204	369	6.8k	
Total	261	994	53.8k	15.5
YACC–	363	396	10.8k	
YACC	204	369	6.8k	
Total	567	765	17.6k	6.8

* For an 8,558 line Pascal program on a MicroVAX II.

Table 1. Comparison of performance of FSM-DPDA Versus DPDA-DPDA Syntax Analysis.

	Two-Phase Recognizer			One-Phase*
	Scanner*	Parser	Total	
Symbols	207	169	376	346
Rules	381	210	591	563
States	451	358	809	1006
Time to generate tables (seconds)	30.3	8.3	38.6	161.1
Total non-error parser actions	40,319	2,786	43,105	59,496
Total parser actions	93,357	60,502	153,859	348,076

* Noncanonical PDA.

Table 2. Comparison of one-phase versus two-phase Pascal recognizers.

between a two-phase DPDA-DPDA recognizer, and a noncanonical DPDA-DPDA recognizer will allow running time of the single-phase parser to be normalized and compared with the FSM-DPDA recognizer.

Why should one expect that a two-stack pushdown automaton could perform as well as a finite state machine? First of all, note that push and pop operations are not costly, since they can consist of simply moving a data value, and incrementing or decrementing a pointer. In contrast a table access on a two-dimensional table may involve a multiplication to compute addresses from subscripts, or a search operation if the table is large and stored using sparse-matrix techniques. While manipulating the stack consumes some time, considerable time is saved by eliminating the procedural interface between the scanner and the parser,* and by eliminating specialized buffering, lookahead, and re-scanning for semantics that are associated with a finite-state scanner like LEX. Also note that extremely fast LR parsing techniques are being developed, which we feel could be applied to an NSLR parser.[15]

* On modern computers, the procedural interface usually involves a great deal of pushing and poping of context information.

6. Results

The above proposals have been tested by the implementation of an SE-SAR-NSLR(1) parser generator and validated on a complete character-level grammar for ISO Pascal. Our grammar uses 567 rules (including 3 compound adjacency restriction rules and 35 exclusion rules) whereas an equivalent grammar written for LEX and YACC uses 309 rules. It should be noted that much of the difference in the number of rules required is due to compact specialized rule forms available in LEX (for example, we require one production for each printable ASCII character). Furthermore, in our grammar all the ambiguities are explicitly corrected, whereas in the LEX-YACC grammar many are handled by default actions.

The parser generated by our method had 1006 states, whereas the one generated by LEX and YACC has 914. Our parser generator has very few size optimizations; in particular, it has no combined shift/reduce actions and no table compression. Parser optimizations of the sophistication used by LEX and YACC might make our parser smaller than the one generated by LEX and YACC. This surprising prediction can be justified by noting that when given a complete grammar for a language a parser generator can make use of the extra information to produce a more compact parser. We predict that more work on the form of restrictive grammar rules and on parser generation techniques for these rules can further reduce the size of the grammars required and the parsers produced.

Our techniques have been used as the parsing engine for an integrated modular attribute grammar system.[7] We are able to parse programming language grammars whose structure reflects the desired attribute computations in a natural way. The mapping from concrete syntax to abstract attribute computations involves two straightforward and efficient operations: *elision* – ignoring irrelevant subtrees (such as those for white space) and *grouping* – applying common attribute computations to parse tree nodes generated from different nonterminals. It would also be possible to attach semantic actions to production rules, as in YACC, but the noncanonical parsing order would make the ordering of side-effects unpredictable. (In fact, the ordering of side-effects between the scanner and parser is already unpredictable in systems such as LEX/YACC, and noncanonical parsing does not significantly compound the problem.)

7. Sample Grammar

We present below a sample character-level SE-SAR-NSLR(1) grammar for a Pascal-like mini language. This grammar has some redundancy, but is still readable and usable. This grammar is not an optimal character-level grammar; many improvements in notation and parser generation strategy are possible that will permit more concise grammars. But even in its present form it is not much longer than the combined length of a scanner grammar and a parser grammar would be for a two-phase translator.

The restrictive disambiguation rules appear at the end of the grammar. They are few in number, and the purpose of each is quite clear. The exclusion rule lists the reserved keywords. The first adjacency restriction prevents the parser from breaking up a long identifier into identifiers and keywords. The last rule is used to solve the dangling-else ambiguity.

```
! A character-level grammar for a
! Pascal-like mini language.

prog       → PROGRAM ID SEMI decls cmpd_stmt DOT
id_list    → ID | id_list COMMA ID
decls      → decls VAR id_list COLON type SEMI | ϵ
type       → INTEGER
cmpd_stmt  → BEGIN opt_stmts END
opt_stmts  → stmt_list | ϵ
stmt_list  → stmt | stmt_list SEMI stmt
stmt       → variable ASSIGNOP expr | cmpd_stmt
           | ID | WHILE expr DO stmt
           | if_then | if_then_else
if_then    → IF expr THEN stmt
if_then_else → IF expr THEN stmt ELSE stmt
variable   → ID
expr_list  → expr | expr_list COMMA expr
expr       → term | sign term | expr ADDOP term
term       → factor | term MULOP factor
factor     → ID | NUM | LPAR expr RPAR | NOT factor

MINUS      → - white
PLUS       → + white
MULT       → * white
DIV        → / white
COLON      → : white
SEMI       → ; white
COMMA      → , white
LPAR       → ( white
RPAR       → ) white
DOT        → . white
ASSIGNOP   → := white
sign       → PLUS | MINUS
ADDOP      → sign
MULOP      → MULT | DIV

white      → real_white | ϵ
real_white → white_char | real_white white_char
white_char → blank | tab | newline | comment
comment    → cmt_body }
cmt_body   → { | cmt_body any_cmt_char
any_cmt_char → any_ASCII_char_except_}

BEGIN      → begin white
begin      → b e g i n
DO         → do white
do         → d o
ELSE       → else white
else       → e l s e
END        → end white
end        → e n d
IF         → if white
if         → i f
INTEGER    → integer white
integer    → i n t e g e r
NOT        → not white
not        → n o t
PROGRAM    → white program white
program    → p r o g r a m
THEN       → then white
then       → t h e n
VAR        → var white
var        → v a r
WHILE      → w h i l e white
while      → w h i l e

ID         → id white
id         → id_frag
id_frag    → letter | id_frag alpha_num
letter     → a | b | ... | z
digit      → 0 | 1 | ... | 9
alpha_num  → letter | digit
NUM        → digit_string white
digit_string → digit | digit_string digit

! Disambiguation section.

id  ↛ begin | do | else | end | if | integer | not
    | program | then | var | while

id begin do else end if integer not program then var while
    ⊣ id begin do else end if integer not
        program then var while NUM

if_then ⊣ ELSE
```

---

8. Enhancements

Our approach is based on SLR parsing techniques, and therefore rejects some grammars that might in fact be parsable using exactly the same state set. We have implemented a hybrid NLALR/SLR(1) parser generator which applies to a larger class of grammars than NSLR(1). The original construction uses the LALR(1) lookahead sets, and noncanonical state expansion is performed only for LALR(1) parsing conflicts. When new items are introduced in noncanonical expansion, they are given SLR(1) lookahead sets. In order to attach LALR(1) lookahead sets to these new items, it would be necessary to compute LALR(k) lookahead sets, but the resulting parser would still use only one lookahead symbol at execution time. We can show that NLR(1) parsers exist as well as hybrid NLR/SLR(1) parsers. We believe that enhancements, similar to those of LALR and LR parsers over SLR parsers, can be made to the processing of exclusion rules and adjacency restrictions by analyzing the paths between parser states, rather than just the items within each parser state.

9. Summary of Notation

A *context-free grammar* (CFG) G is a quadruple $G = (V_N, V_T, P, S)$, where V_N is the set of *nonterminals*, V_T is the set of *terminals*, P is the set of *productions*, and S in V_N is the *start symbol*. The set of all grammar symbols is represented by $V = V_N \cup V_T$. We assume that the grammar has no duplicate or useless productions, and no useless symbols.

Lowercase letters early in the alphabet, such as a, b, and c, represent a single terminal symbol. Uppercase letters early in the alphabet, such as A, B, and C, and also the letter S, represent nonterminals. Uppercase letters late in the alphabet, such as X, Y, and Z, represent terminals or nonterminals. The Greek letter ϵ

represents the empty string, and the other lowercase Greek letters, such as α, β, and γ, represent strings of terminals or nonterminals, or ϵ. Lowercase letters late in the alphabet, such as x, y, and z, represent strings of terminals or ϵ. The length of a string β is denoted by $|\beta|$, and therefore $|\epsilon| = 0$. Strings of symbols can be indexed, so that $\beta[i]$ represents the i^{th} symbol of β. The symbol $\varnothing$ represents the empty set.

The productions in P are numbered 1, 2, ..., t where $t = |P|$. Productions take the form $A \rightarrow \alpha$, where A in V_N is called the *left part*, and α in V^* is called the *right part*. The i^{th} production in P is denoted by P_i, and n_i denotes the length of the right part of P_i.

If $A \rightarrow \alpha$ is a production and $\beta A \gamma$ is a string in V^+, then we write $\beta A \gamma \Rightarrow \beta \alpha \gamma$ and say that $\beta A \gamma$ *derives* $\beta \alpha \gamma$. The transitive closure of $\Rightarrow$ is denoted by $\overset{+}{\Rightarrow}$, and the reflexive transitive closure of $\Rightarrow$ is denoted by $\overset{*}{\Rightarrow}$.

A *sentential form* of G is a string α such that $S \overset{*}{\Rightarrow} \alpha$ and α is in V^*. A *sentence* x of G is a sentential form of G consisting solely of terminals, i.e., x is in V_T^*. The *language* $L(G)$ generated by G is the set of sentences generated by G, i.e., $L(G) = \{x \mid S \overset{*}{\Rightarrow} x\}$. We can also refer to the language generated by some symbol X of grammar G as $L_G(X) = \{x \mid X \overset{*}{\Rightarrow} x\}$.

A symbol X in V is said to be *useless* if there is no derivation of the form $S \overset{*}{\Rightarrow} uXv \overset{*}{\Rightarrow} uxv$. A nonterminal A is called *nullable* if there exists a derivation $A \overset{*}{\Rightarrow} \epsilon$.

A *restricted context-free grammar*, an RCFG, is a quadruple $G = (V, P, R, S)$ where:

$V = V_N \cup V_T$
= The set of symbols in the grammar.
V_N = The set of nonterminal symbols.
V_T = The set of terminal symbols.
P = The set of productions.
$R = R_E \cup R_{AR}$ = The set of restrictive rules.
R_E = The set of exclusion rules.
R_{AR} = The set of adjacency-restriction rules.
S = A member of V_N, is the start symbol.

Every restricted context-free grammar $G = (V, P, R, S)$, has a corresponding unrestricted CFG $G^U = (V_N, V_T, P, S)$, which is the same a G but without the restrictive rules. The definitions of the restrictive rules imply that $L(G) \subseteq L(G^U)$.

10. Acknowledgements

We would like to thank the Natural Sciences and Engineering Research Council of Canada, and the Information Technology Research Centre of Ontario for their financial support of this work. We would also like to acknowledge Norbert Kusters for his contribution in programming parts of this project.

11. References

1. United States Department of Defense. *Reference Manual for the Ada Programming Language*. (1980).
2. Aho, A. V., Johnson, S. C., and Ullman, J. D., "Deterministic parsing of ambiguous grammars." *Communications of the ACM*, Vol. 18, No. 8, pages 441-452 (Aug. 1975).
3. Aho, Alfred V., Sethi, Ravi., and Ullman, Jeffrey D., *Compilers: Principles, Techniques, and Tools*. Addison Wesley, Reading, Mass. (1986).
4. Baker, Theodore P., "Extending lookahead for LR parsers." *Journal of Computer and System Science*, Vol. 22, No. 2, pages 243-259 (1982).
5. Bermudez, Manuel E. and Schimpf, Karl M., "A practical arbitrary look-ahead LR parsing technique." *ACM SIGPLAN Notices*, Vol. 21, No. 7, pages 136-144 (July 1986).
6. DeRemer, Franklin L., "Lexical analysis." in *Compiler Construction: An Advanced Course.*, ed. F. L. Bauer and J. Eickel, pp. 105-120, Springer-Verlag, Berlin (1976).
7. Dueck, Gerald D. P. and Cormack, Gordon V., "Modular Attribute Grammars." Technical Report CS-88-19, University of Waterloo, Waterloo, Canada (May 1988).
8. Hopcroft, John E. and Ullman, Jeffrey D., *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass. (1979).
9. Jensen, Kathleen and Wirth, Niklaus, *Pascal User Manual and Report. Third Edition*. p. 266, Springer-Verlag, New York (1985). Revised by: Andrew B. Mickel and James F. Miner
10. Krzemien, Roman. and Lukasiewicz, Andrzej., "Automatic generation of lexical analyzers in a compiler-compiler." *Information Processing Letters*, Vol. 4, No. 6, pages 165-168 (Mar. 1976).
11. LaLonde, Wilf R., "Regular right part grammars and their parsers." *CACM*, Vol. 20, No. 10, pages 731-741 (Oct. 1977).
12. Lesk, M. E., "LEX—A Lexical Analyzer Generator." Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J. (1975).
13. Paxson, Vern. *The FLEX Scanner Generator (Program)*. Real Time Systems, Bldg 46A, Lawrence Berkeley Laboratory, 1 Cyclotron Rd., Berkeley, CA 94720.
14. Tai, Kuo-Chung., "Noncanonical SLR(1) Grammars." *ACM TOPLAS*, Vol. 1, No. 2, pages 295-320 (Oct. 1979).
15. Whitney, M. and Horspool, R.N., "Extremely rapid LR parsing." in *Proceedings of Workshop on Compiler-Compiler and High-Speed Compilation.*, Berlin, G.D.R. (Oct. 1988).
16. Wirth, Niklaus, *Programming in Modula-2. Third Corrected Edition*. p. 202, Springer-Verlag, New York (1985).