The Use of Predicates In LL(k) And LR(k) Parser Generators (Technical Summary)

Article · January 1993
Source: OAI

3 authors, including:

Terence Parr
Google Inc.
39 PUBLICATIONS **2,161** CITATIONS

SEE PROFILE

Henry G. Dietz
University of Kentucky
131 PUBLICATIONS **1,150** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project   codebuff View project

Project   ANTLR parser generator View project

7-1-1993

The Use of Predicates In LL(k) And LR(k) Parser Generators (Technical Summary)

T. J. Parr
Purdue University School of Electrical Engineering

R. W. Quong
Purdue University School of Electrical Engineering

H. G. Dietz
Purdue University School of Electrical Engineering

THE USE OF PREDICATES IN LL(k) AND LR(k) PARSER GENERATORS

T. J. PARR
R. W. QUONG
H. G. DIETZ

SCHOOL OF ELECTRICAL ENGINEERING
PURDUE UNIVERSITY
WEST LAFAYETTE, INDIANA 47907-1285

The Use of Predicates In *LL(k)* And *LR(k)* Parser Generators[†]

(Technical Summary)

T. J. Parr, R. W. Quong, and H. G. Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
(317) 494-1739
quong@ecn.purdue.edu

Abstract

Although existing *LR*(1) or *LL*(1) parser generators suffice for many language recognition problems, writing a straightforward grammar to *translate* a complicated language, such as C++ or even C, remains a non-trivial task. We have often found that adding translation actions to the grammar is harder than writing the grammar itself. Part of the problem is that many languages are context-sensitive. Simple, natural descriptions of these languages escape current language tool technology because they were not designed to handle semantic information. In this paper, we introduce predicated *LR*(k) and *LL*(k) parsers as a solution. *Predicates* provide a general, practical means to utilize semantic tests in parsers. Used in conjunction with k>1 lookahead sets, a predicated parser simplifies the task of writing real translators.

Our approach differs from previous work in that (i) we allow multiple predicates to be placed arbitrarily within a production, *(ii)* we describe the construction of predicated *LR*(k) parsing tables, *(iii)* we automatically hoist predicates in an *LL(k)* parser from one production to aid in the recognition of another, and (iv) we have implemented predicate handling in a public-domain parser generator that offers k-token lookahead — The Purdue Compiler Construction Tool Set (PCCTS).

Keywords: parser generators, predicate testing, *LL*(k), *LR*(k).

1. Introduction

Although in theory, parsing is widely held to be a sufficiently solved problem, in practice, writing a grammar with embedded translation actions remains a non-trivial task. Most language applications involve translation rather than mere recognition. Translation presents two difficulties over recognition: (i) sentences must be tested for semantic as well as syntactic validity, and (ii) adding semantic actions to a grammar usually introduces syntactic ambiguities for *LR* based techniques.

Consider for example, the problem of creating an *LALR*(1) (yacc compatible) compatible grammar for a **C++** compiler. After many others had failed, J. Roskind finally succeeded in developing a carefully crafted grammar. Unfortunately, this story is not another success for yacc. **Roskind's** grammar is large (over 600 productions), is non-trivial, has no actions, and is broken easily when actions are added. Furthermore, the correlation between the grammar and the underlying language is weak, so that adding actions without breaking the *LALR*(1) grammar is not **easy.**

As another analogy, consider writing all software in assembly langauge. Although, in theory this idea could be done, in practice this idea is clearly infeasible. Similarly, although existing parser tools may be powerful enough in theory, in practice, creating a conformant grammar may involve significant user effort and tedium. Often the user must tweak the grammar via trial and error when adding semantic actions rendering the grammar fragile and **unreadable.** In particular, we have found ourselves doing manual left factoring or **inline** expansion of productions to get a yacc compatible grammar.

The other problem in real world translators is dealing with semantic information when parsing, such as deciding if a C++ identifier is a type or a variable name token Currently ad **hoc** techniques are used, such as having the lexical analyzer consult the symbol table to **determine** what token (`typeT` or `nameT`) is given to the parser. However, lexical analyzers have no context information except the current token of lookahead and must be coerced via flags to yield the various token types. Ad **hoc** approaches become increasingly difficult to implement as the number of ambiguities in a grammar rises.

We believe the user should be able to write a grammar (with actions) that has a simple and natural **correspondence** to the underlying language. To solve these two problems, we recommend augmenting existing parsers in two way: the use of k >1 lookahead and the use of semantic predicates as a general purpose method to handle semantic parse decisions. In this paper, we discuss the theory and practice of predicated *LL(k)* and *LR*(k) parsers and we illustrate how we added predicates to a public domain *LL(k)* parser generator. We also show that a predicated parser eliminates the **need** for ad **hoc** techniques in the scanner. The final result is parsing tools that simplify the users task.

Our summary is organized as follows. In Sections 2, 3, and 4 we define predicates and review previous work in the area In Section 5, we describe how to construct predicated *LR(k)* parsers. Next, in Section 6, we describe how the PCCTS generates predicated *LL(k)* parsers. Finally, in Section 7, we prove that parser predicates are stronger than scanner predicates.

2. Previous Work

Attributed grammars have received attention in the literature since their introduction [Knu68, Knu71]. [LRS74] considered the application of attribute grammars to compilers and characterized the types of attributed grammars that could be efficiently handled via bottom up and top down parsing methods. Despite the efforts in this area, attribute grammars have had little impact on compiler construction [Wai90].

[MiF79] introduced a class of top down grammars, *ALL(k),* which could be easily parsed by top **down** methods. *ALL(k)* specifications included two types of predicates, disambiguating and contextual, that were used to handle the context-sensitive portions of programming languages; the authors implemented an *ALL*(1) parser generator based upon their *ALL(k)* definition.

Our approach differs from [MiF79] in a number of ways. Whereas Milton and Fischer allow exactly one disambiguating predicate per production, we allow multiple predicates and do not distinguish between disambiguating and contextual predicates as this differentiation can be automatically determined. **Our** predicate definition permits the placement of predicates anywhere within a production and, more importantly, specifies the desired evaluation time by the location of the predicate. Also, the user need not determine when a structure is syntactically ambiguous and requires a disambiguating predicate; the grammar analysis phase has this information and can search for predicates that can be used to resolve the conflict (see the section on predicate hoisting and propagation). Further, the disambiguating predicates of [MiF79] require that the user specify the set of lookahead k-tuples over which the predicate is valid. **Our** predicates are automatically evaluated only when the **lookahead** buffer is consistent with the context surrounding the predicate's position. We have combined this predicate definition with an existing tool that generates *LL(k>1)* parsers. Although in theory, the predicates of [MiF79] and the predicates of this paper are equivalent in recognition strength, in practice our predicates allow for more concise and natural grammars.

Another group, [HCW82], developed a parser generator and language, S/SL, that allowed parsing to be a function of semantics. This was accomplished by allowing rule return values to predict future productions. Unfortunately, their system had a number of weaknesses that rendered it less interesting for very large applications; **e.g.** parsers could only see one token of **lookahead** and the user had to compute prediction lookahead sets by hand.

Our predicate definition is not restricted to top down parsing. We describe predicates as a general mechanism for semantic validation and context-sensitive parsing for which we define predicated *LL(k)* **and** *LR*(k) parsers; we also supply parser and parser generator construction details.

3. Background

A (context-free) *grammar* is denoted $G=(N, T, P, s)$, where N is the set of *nonterminals,* T is the set of *terminals,* P is a set of productions or rules, and *s* is a special nonterminal, the start symbol. The reserved **terminal** $ denotes the end of input and will not appear as normal input.

For the rest of this paper, we adopt the following symbol convention (similar to that used by YACC).

- **Nonterminal** names begin with a lower case letter (a, b, ..., z).

- Terminals or tokens are represented by names beginning with a capital letters (A, B, C, ..., Z). In addition, strings in quotes (**e.g.,** **"** **:** **"** and **"** w h i l e **"**) denote terminals.

- The lower case greek letters a , β, γ, 6, and ρ denote (possibly empty) **strings** of terminals and **nonterminals; i.e.** from $(N \cup T)$ *. w is used to represent sentences (w $\in$ T*, $s \Rightarrow^* w$).

- Lookahead tokens are referred to as $\lambda_1 ... \lambda_k$. A k tuple is a sequence of k tokens, usually referring to a lookahead sequence.

A standard left-to-right parser using a stack and k tokens of lookahead is a function,

$$\text{parser: (state } \mathbf{x} \, T^k) \rightarrow (\{\text{push x, pop } y, \text{ error, accept}), \text{ next} - \text{state}),$$

where state is the top of the stack, push x adds the **item(s)** x to the stack, pop y **pops** y items from the stack. Both *LL(k)* and *LR*(k) parsers fall into this category. The notation *Lx(k)* represents *LL(k)*, *LR*(k) and the variants of *LR(k),* such as *LALR*(k). Finally, an *Lx*(k) grammar is assumed not to be *Lx*(k−1).

Rules have the form:

$$\text{a} \; : \; \alpha_1 \; | \; \alpha_2 \; | \; ... \; | \; a_{rr} \, ;$$

where each α_i is considered an alternative production. In a rule, a : a $\beta \gamma$;, a is the *left-context* of β, and γ is the right-context of β. $FIRST_k(\text{a})$ is the set of k-tuple of terminals that can begin a sentence derived from a; $FOLLOW_k(a)$ is the set of k-tuple that can follow a in a **sentential** form.

4. Predicates

To allow context-sensitive parsing, parsers must be functions of semantics as well as syntax. Parser generators can support this type of parsing by permitting the specification of semantic **tests,** called predicates. These tests can be used for both semantic *validation* and for disambiguating syntactic conflicts in the underlying grammar.

A predicate is a function $\phi(x_1, x_2, ..., x_n)$ that returns either true (success) or false (failure). Predicates are enclosed in European quotes followed by a question mark, <<ϕ>>?. We use pred−*LR*(k) and pred−*LL*(k) to denote predicated *LR*(k) and *LL*(k) parsers, respectively. *Pred−LL(k)* parsers can efficiently handle L-attributed grammars and hence x_i in predicated *LL*(k) grammars can be functions of attributes at, below, or to the left of a given node in a derivation tree **[LRS74]**. Further, we allow x_i to be a function of the attributes for the next k tokens of **lookahead.** *Pred−LR*(k) parsers cannot manage inherited attributes in one pass (those derived **from** symbols in the left context); they are restricted to S-attributed grammars **[LRS74]** and, therefore, x_i in predicated *LR*(k) may only reference synthesized attributes. As with pred−LL(k)

parsers, x_i in *pred−LR(k)* may also reference the attributes of next k token of lookahead.

A successful predicate matches ε, the empty token; a failed predicate nullifies its production. A disambiguating predicate eliminating its production from consideration; a validation predicate terminates the parsing of that production. Given a syntactically ambiguous list of alternatives with embedded predicates, only those productions whose disambiguating predicates evaluate to true are considered applicable. The disambiguating predicates associated with a production must be mutually exclusive; **i.e.** exactly one production must succeed to uniquely resolve a syntactic conflict

4.1. Time of Evaluation for Predicates

A predicate ϕ is *viable* for lookahead $\lambda_1...\lambda_k$ if $s \Rightarrow^+ \alpha\,\phi\,\lambda_1 ... \lambda_k\,\beta$. We evaluate a predicate ϕ only if it is viable, namely if ϕ could be followed by the existing lookahead. Thus, a predicate only affects parsing when a normal non-predicated parser would have several ambiguous choices.

The placement of a predicate in a *pred−LR(k)* grammar indicates the time of evaluation; **e.g.** in a production of the form

$$a \; : \; \alpha \; <<\phi>>? \; \beta \; ;$$

ϕ is evaluated after its left context, a, has been shifted, but before its right context, β. A *pred−LR(k)* parser evaluates ϕ only if it is viable, namely if the lookahead $\lambda_1...\lambda_k \in FIRST_k(\beta\,\delta)$ where $s \Rightarrow^* w\,a\,\delta$.

Because an *LL(k)*-style parser is predictive, a *pred*−LL parser may need to *hoist* a predicate ϕ forward to the beginning of a production. However, we still evaluate ϕ only if it is viable from its *original position* in the grammar. For example, in the previous production, assume we hoist ϕ forward m tokens forward to the beginning of a, so that internally we get the production

$$a \; : \; <<\phi>>? \; \alpha \; \beta \; ;$$

In this case, we evaluate ϕ *iff* its left and right contexts are viable, namely if $\lambda_1...\lambda_k \in FIRST_k(\alpha\beta\,\delta)$ and $\lambda_{m+1}...\lambda_{m+k+1} \in FIRST_k(\beta\,\delta)$. Note that after hoisting ϕ, we need $m+k$ tokens of lookahead when ϕ is evaluated to ensure the the original right context $\beta\,\delta$ is present. We hoist a predicate at most k tokens forward, so that $m < k$, and thus at most $2k$ tokens of lookahead are needed in a *pred−LL(k)* parser.

Because a *pred−LL(k)* parser can move a predicate, we add the following definitions and restrictions. Consider a production of the form

$$a \; : \; \bullet \; \alpha_1 \; \phi_1 \; \alpha_2 \; | \; \bullet \; \beta_1 \; \phi_2 \; \beta_2 ;$$

A predicate is *visible* at some point, if it can be seen in the original grammar within the next k tokens of lookahead.

(i) Predicates may be a function only of their left context and tokens of their right context that will be within the lookahead buffer available at the left edge of a.

(ii) Predicates may not have side-effects.

(iii) Predicates may not be a function of semantic actions situated between themselves and the •
in rule a. **E.g.** a predicate cannot depend on an action over which it will be hoisted.

5. *Pred –LR (k)* Parsers

We now describe how to construct a *pred –LR (k)* parser. Predicates require special han-
dling to ensure that they are evaluated only once at the specified position in the grammar. Predi-
cates **appear** as a special symbol ϕ in the parsing tables and lead to an additional parsing action,
evaluate.

We demonstrate these ideas via the following grammar.

```
(1)  s    :    A <<φ₁>>? B
(2)       |    a <<φ₂>>? b
             ;
(3)  a    :    A ;
(4)  b    :    B ;
```

Grammar 1: Example *pred –LR* (1) Grammar

Assume ϕ_1 succeeds and ϕ_2 fails on input A B. The parser will shift A, evaluate both ϕ_1 and
ϕ_2, shift B and reduce by production (1). The *pred –LR* (1) *item-sets* are shown in Figure 1.
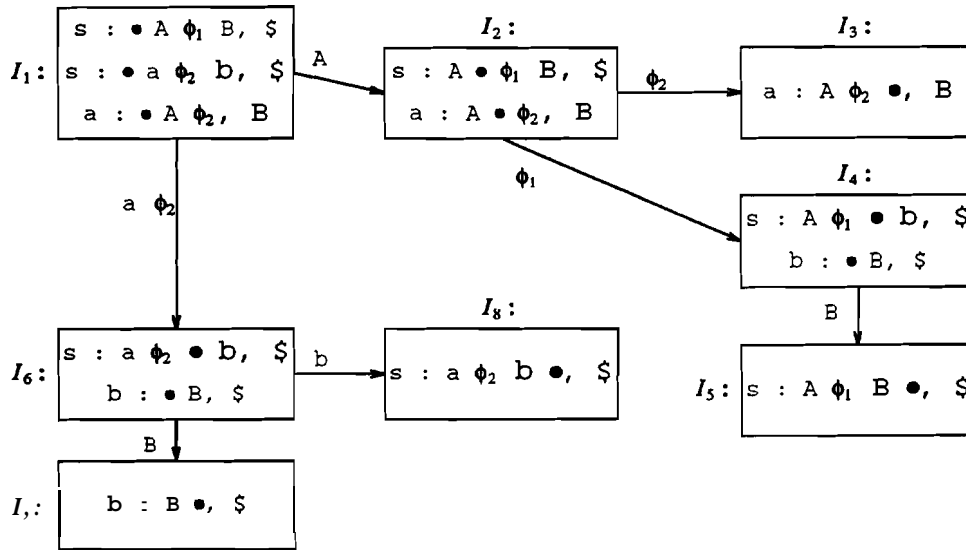

Figure 1: *Pred –LR* (1) Machine

Each unique predicate is a unique parsing symbol that matches no input. Moving the dot past a predicate corresponds to the predicate evaluating true. The **main** difference between pred$-$LR (k) item-sets and *LR*(k) item-sets occurs when there exists an item in which the dot precedes a **nonterminal** directly followed by a predicate ϕ as in production (2) of I_1. Closure of (2) adds (3) to I_1, : $\bullet$ A ϕ_2, B **in** the *pred$-$LR*(1) item-set. Without the predicates, the next item-set I_2 would contain a **shift/reduce** conflict. In order to disambiguate the **conflict,** the predicates must be evaluated before the next shift or reduce action. By appending the predicate to (3) in I_1 and I_2, we force the predicate to be evaluated before the next reduce action.

5.1. Pred$-$*LR*(k) Parser Construction

We now formally describe the construction of a *pred$-$LR*(k) parser. Due to the similarity of *pred$-$LR(k)* parsers to *LR*(k) parsers, we only discuss the differences between the two in constructing the action and **goto** table. We assume the reader is familiar with *LR*(k) parsing [**AhU86, FiL88**].

A *pred$-$LR(k)* parser consists of two two-dimensional tables, *action* and **goto,** where *action*[state, $\lambda_{1...k}$] $\in$ {shift, reduce, accept, error, evaluate($i_1, i_2, ..., i_n$)}, and each entry in *goto*[state, *N*] contains another state. The action evaluate($i_1, i_2, ..., i_n$) indicates that the *n* predicates ϕ_{i_1} through ϕ_{i_n} should be evaluated. If exactly one predicate succeeds, parsing continues along the corresponding production; otherwise a parse error occurs. The parse tables are constructed assuming exactly one predicate will succeed.

A pred$-$LR (k) item is simply a *LR*(k) item, namely a *pred$-$LR*(k) item is a double [$X, \lambda_{1...k}$], where X is a *LR*(0) item, and $\lambda_{1...k}$ = the **lookahead,** is a set of k-tuples. Construction of the *pred$-$LR*(k) item-sets is identical to that of *LR*(k) item-sets, except in the following two cases, when item-set *I'* contains *LR*(k) items of the following form.

$$a \; : \; \alpha \bullet b \; <<\phi>>? \; \gamma \qquad , \; \lambda_{1...k}$$
$$b \; : \; \beta \; ;.$$

1. Closure of item-set *I'* adds the following item to *I'*.

$$b \; : \; \bullet \; \beta \; <<\phi>>? , \; FIRST_k(\gamma \; \lambda_{1...k})$$

   Unlike a normal token, predicate ϕ is appended to the production for b and will remain in subsequent item-sets.

2. The entry goto(*I'*, b) is replaced by goto(*I'*, b ϕ) and contains the state for item-set

$$a \; : \; \alpha \; b \; <<\phi>>? \bullet \gamma, \; \lambda_{1...k}$$

   Because the predicate ϕ was appended to the production for b in the previous rule, the reduction b : β ; will not take place unless ϕ was true. Thus, we move past both b and ϕ in one transition, and evaluate ϕ only once.

Entries in the parsing table action[. , .] are derived from the item-sets identically to that of an LR(k) parsing table, except in the following cases.

1. **Add** e v a l u a t e ϕ to *action*[I',$FIRST_k$($\gamma\,\lambda_{1...k}$)], when the following item is in item set I'.

$$a \;:\; \alpha \bullet <<\phi>>? \;\; \gamma, \;\; \lambda_{1...k}$$

2. Add reduce b : β ; to *action*[I',$FIRST_k$($\gamma\,\lambda_{1...k}$)], when the following item is in item set $\mathbf{I'}$.

$$b \;:\; \beta \; <<\phi>>? \; \bullet, \;\; FIRST_k(\gamma\,\lambda_{1...k})$$

If an action table entry contains multiple actions, we can get **shift/evaluate** or **reduce/evaluate** conflicts. These conflicts occur when the parser has a choice between two productions, and only one production has a predicate. For example, the following item-set has a **reduce/evaluate** conflict.

```
a   :   A • <<φ>>?          (evaluate φ)
    '
b   :   A •                 (reduce b : A)
```

In the full paper, we show that our construction (1) evaluates predicates once, (2) only evaluates viable predicates, (3) evaluates predicates at the points specified by the grammar, (4) detects **shift/evaluate** and **reduce/evaluate** conflicts. We also discuss how to deal with *pred*–LR (k) conflicts.

6. *Pred–LL(k)* Parsers

In this extended abstract, we give an example of how predicates are implemented in The **Purdue** Compiler Construction Tool Set [PDC92], PCCTS, a public domain parser generator (currently, only an **internal** version has predicate capabilities). The example illustrates both the theoretical and practical issues of *pred–LL*(k) parsing.

The following *pred*–LL (2) grammar is not LL(2) (assuming the predicates were removed) because the terminal sequence A B predicts both productions of rule a. However, ϕ_1 and ϕ_2 serve as disambiguating predicates giving the parser a way to choose between the two productions. **Predicate** ϕ_3 is only evaluated on lookahead X,Y, and because there is no parsing ambiguity, it **serves** as a validation predicate.

```
a   :   A <<φ1>>? B C
    |   b D
    ;

b   :   A B <<φ2>>?
    |   <<φ3>>? X Y
```

Grammar 2: Example *pred–LL*(2) Grammar

To resolve the above *LL*(2) ambiguity, PCCTS searches for predicates *visible* to the parsing decision on the left edge of rule a. If there are no visible predicates, PCCTS reports an ambiguity. However, if there is at least one visible predicate, the analysis phase reports no ambiguity and supplies the disambiguating **predicate(s)** to the parser generation phase. **As** ϕ_1 is visible in production one and both ϕ_2 and ϕ_3 are visible in production two, PCCTS uses them to disambiguate rule a. PCCTS generates the following C code for the above grammar.

```
a()
{
    if ( (((LA(1)==A) && (LA(2)==B)) && (φ1)) &&
         ((LA(1)==A) && (LA(2)==B)) ) {
        zzmatch(A); zzCONSUME;
        zzmatch(B); zzCONSUME;
        zzmatch(C); zzCONSUME;
    }
    else if ( (((LA(1)==A&&(LA(2)==B))) && (φ2)) ||
            ((LA(1)==A || LA(1)==X) && (LA(2)==B || LA(2)==Y)) ) {
        b();
        zzmatch(D); zzCONSUME;
    }
    else {error;}
}


b()
{
    if ( (LA(1)==A) ) {
        zzmatch(A); zzCONSUME;
        zzmatch(B); zzCONSUME;
        if (!(φ2)) {error;}
    }
    else if ( (LA(1)==X) ) {
        if (!(φ3)) {error;}
        zzmatch(X); zzCONSUME;
        zzmatch(Y); zzCONSUME;
    }
    else {error;}
}
```

Listing 1: **PCCTS** *pred−LL*(2) code for Grammar 2

Predicate ϕ_1 is used to predict production one, but is only evaluated when viable, that is when its **lookahead** is consistent with its enclosing contexts. Similarly, ϕ_2 and its context are hoisted from rule b to help predict the second production of a. Predicate ϕ_3 is not hoisted because rule a is not ambiguous on **lookahead** X Y; therefore, ϕ_3 is used only for semantic validation within rule b.

Grammar 2 is not *LL*(2), but is *LL*(3). Thus, in a *pred−LL*(3) parser, the predicates would provide validation only and would not be hoisted. In this case, PCCTS generates the following *pred−LL*(3) parser code for rule a.

```
a()
{
    if ( (LA(1)==A) && (LA(2)==B) && (LA(3)==C) ) {
        zzmatch(A); zzCONSUME;
        if (!(φ₁)) {error;}
        zzmatch(B); zzCONSUME;
        zzmatch(C); zzCONSUME;
    }
    else if ( (LA(1)==A || LA(1)==X) && (LA(2)==B || LA(2)==Y) &&
              (LA(3)==D) ) {
        b();
        zzmatch(D); zzCONSUME;
    }
    else (error;}
}

b()
{
    // same code as in Listing 1.
}
```

<p align="center">**Listing 2: PCCTS** pred *−LL* (3) code for Grammar 2</p>

6.1. Pred *−LL*(k) Grammar **Analysis**

The previous section gave an example of how predicates are incorporated into the normal *LL(k)* parsing strategy without concern for how context sets and disambiguating predicates were extracted from the grammar. In this section, we present an extension to *LL*(k) grammar analysis that not.only detects ambiguities, but supplies lookahead information and disambiguating predicates to the code generation phase.

LL(k) grammars can be reduced to a set of parsing decisions of the form

$$
\begin{array}{l}
a \ : \ \alpha \\
\ \ | \ \beta \\
\ \ ;
\end{array}
$$

The decisions are syntactically ambiguous *iff* a and β generate phrases with at least one common k token prefix; i.e. for $s \Rightarrow^* wa\delta$, $S = FIRST_k(\alpha\, 6) \cap FIRST_k(\alpha\, 6) \# \varnothing$ where S represents the set of k-tuple that predict both productions. We consider a to be non-*pred−LL(k)* iff S is non-empty and no disambiguating predicates are available. A predicate is *disambiguating* if it is visible and resides in a production that generates at least one k-tuple in S. Hence, not all visible predicates aid in the disambiguation of a decision as was demonstrated in Grammar 2.

PCCTS automatically determines when disambiguating predicates are required and, more importantly, which of the visible predicates are disambiguating, by traversing a directed-graph representation of the grammar. Once the collection of visible predicates has been established, disambiguating predicates are isolated via algorithm 1 with S, 6 as above and:

$$
b \ : \ \gamma \ <<φ>>? \ \rho \ ;
$$

where b is derivable from rule a above.

```
function disambigqreds ( P : set of visibleqredicates ) : set of disambigqredicates;
begin
      D ← new set of disambigqredicates;
      for each p in P do
          d ← new disambigqredicate;
          d.expr ← p;
          d.k_distance ← distance of p from parsing decision;
          d.context ← right context of p;
          if ( (FIRST_k(γ ρ δ) ⋂ S) ≠ ∅ ) then
               D ← D ⋃ d;
      enddo
      return D;
end
```

Algorithm 1: Isolation of Disambiguating Predicates

Appendix I discusses the implementation of *pred−LL*(k) analysis in more detail.

7. Predicates: Scanner Versus Parser

Predicates in the parser are strictly more powerful than predicates in the scanner. For example, the standard way to parse C++ relies on the scanner to differentiate between type names and non-type names via symbol table access, i.e. the scanner uses a predicate. In Appendix II, we show that *L*(predicated *LR*(k) parser + simple scanner) ⊃ L(a predicated scanner + simple *LR*(k) parser), where $L(x)$ is the language recognized by x. The practical consequence of this theorem is that a predicated parser only needs a simple (non-predicated) scanner, such as those generated by the lex [Les75] or flex [Pax90] scanner generators.

8. Conclusions and Future Work

In this paper, we have defined predicated *LR*(k) and LL(k) parsers. Predicates provide a flexible general means of allowing parsing to be a function of semantics as well as syntax. We have described the construction of *pred*−LR (k) and *pred−LL*(k) parsers, the increased recognition strengths of predicated parsers, and the implementation of our public-domain predicated *LL(k)* parser generator within the Purdue Compiler Construction Tool Set (PCCTS). For ease of use, PCCTS generated *pred−LL*(k) parsers allow arbitrary predicate placement and automatically hoist predicates to prediction points

Currently, **PCCTS** automatically generates code to report syntax errors messages and to attempt recovery. We have no clear definition of how the error reporting facility should be augmented, but we anticipate allowing the user to specify an error string to print upon predicate failure. As an interim measure, PCCTS-generated parsers print "failed φ" where φ is the predicate that evaluated to False. We are also investigating the use of predicates that return a r e . value, rather than true or false, whereby the production with the largest predicate value is chosen.

We thank John **Interrante** for his feedback on using **Roskind's** grammar.

9. References

[ASU86]   Aho, A. V., Sethi, R., and Ullman, J. D., Compilers: Principles, Techniques, and Tools, Addison-Wesley, Reading MA, 1986.

[FiL88]   Charles N. Fischer and Richard J. LeBlanc, Crafting a Compiler, Benjamin/Cummings Publishing Company, 1988.

[HCW82]   R. C. Holt, J. R. Cordy, and D. B. Wortman, "An Introduction to S/SL: Syntax/Semantic Language,", ACM TOPLAS Vol. 4, No. 2, April 1982, pp 149-178.

[Joh78]   Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler," Bell Laboratories, Murray Hill, NJ, 1978.

[KeR88]   Brian W. Kernighan and Dennis M. Ritchie, "The C programming Language," Prentice Hall Inc., Englewood Cliffs, New Jersey, 1988.

[Knu68]   Donald E. Knuth, "Semantics of Context-Free Languages," Journal Mathematical Systems Theory Vol 2, No. 2, 1968, pp 127-145.

[Knu71]   Donald E. Knuth, "Semantics of Context-Free Languages: Correction," Journal Mathematical Systems Theory Vol 5, No. 1, 1971, pp 95-96.

[Knu65]   Donald E. Knuth, "On the Translation of Languages from Left to Right," Information and Control 8, 1965, pp 607-639.

[Les75]   M. E. Lesk, "LEX — a Lexical Analyzer Generator," CSTR 39 Bell Laboratories, Murray Hill, NJ, 1975.

[LRS74]   Lewis, P. M., Rosenkrantz, D. J., and Stearns, R. E., "Attributed Translations," Journal of Computer and System Science, Vol 9, No. 3, Dec. 1974, pp 279-307.

[MiF79]   D.R. Milton and C.N. Fischer, "LL(k) Parsing for Attributed Grammars," Automata, Languages and Programming, Sixth Colloquium, 1979, pp 422-430.

[MKR79]   D.R. Milton, L.W. Kirchhoff and B.R. Rowland, "An *ALL*(1) Compiler Generator," Conference Record of SIGPLAN Symposium on Compiler Construction, 1979.

[PaD92]   Terence J. Parr and Henry G. Dietz, "A Practical Approach to LL(k): $LL_m(n)$," Purdue Electrical Engineering Technical Report TR92-30, 1992.

[PDC92]   T.J. Parr, H.G. Dietz, and W.E. Cohen, "PCCTS Reference Manual Version 1.00," ACM SIGPLAN Notices, February 1992.

[Pax90]   V. Paxson, "Flex users manual," Ithaca, N.Y. Cornell University, 1990.

[Str87]   Bjarne Stroustrup, "The C++ Programming Language," Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[Wai90]   W. M. Waite, "Use of Attribute Grammars in Compiler Construction," Attribute Grammars and their Applications, Lecture Notes in Computer Science #461 (Proceedings of WAGA 90), Springer-Verlag, 1990, pp 254-265.

10. Appendix I — Implementation of *pred–LL*(k) analysis

This section provides a more detailed look at the analysis phase of PCCTS; specifically, we discuss hoisting distance and the syntactic context under which predicates may be evaluated.

PCCTS tracks the hoisting of disambiguating predicates via the following C structure:

```
struct PredicateRef {
    char *expr;              /* C code for predicate expression */
    Tree *context;           /* Context under which ok to eval predicate */
    int k_distance;          /* Offset from current token of lookahead */
};
```

Predicates may be a function of the next k tokens of lookahead relative to their position; therefore, the distance a predicate is hoisted must be recorded in `k_distance` to compensate for the shift in **lookahead** context; see [PaD92] for more information on k lookahead. To illustrate context and relative position, consider the following *pred–LL*(2) grammar:

$$
\begin{array}{lll}
a & : & A\ b \\
  & | & <<f_1(\text{LA}(1))>>?\ A\ B\ C \\
  & ; & \\
\\
b & : & B\ <<f_2(\text{LA}(1))>>?\ D \\
  & | & <<f_3(\text{LA}(1),\text{LA}(2))>>?\ B\ E \\
  & ; &
\end{array}
$$

where LA(i) is the ith token of lookahead. There are two predicate references visible from the start of production one in **rule a** and one reference from the start of production two:

expr	context	k_distance
$f_1(\text{LA}(1))$	(A,B)	0
$f_2(\text{LA}(1))$	(A,B)	2
$f_3(\text{LA}(1),\text{LA}(2))$	(A,B)	1

The LA(i) references in any predicate are translated to LA(i+k_distance) in the generated parser. For example, at the left edge of the second production of mle b, LA(1) and LA(2) are B and E respectively. However, when f_3 is hoisted for use in the prediction decision for mle a, LA(1) and LA(2) are A and B. References to lookahead in f_3 are compensated for this by adding the correct k_distance yielding $f_3(\text{LA}(1+1),\text{LA}(2+1))$. Because predicates may be hoisted forward k tokens and may reference k tokens of lookahead relative to their position, *pred–LL*(k) parsers actually need to maintain k+k lookahead.

Because multiple disambiguating predicates may be hoisted, each from a different context, PCCTS also records the context of predicates to ensure that the early evaluation of the predicate only occurs within the correct syntactic framework

The context of a predicate is $FIRST_k(\alpha\,\gamma\,\rho\,\beta\,6)$ where $s \Rightarrow^* wa6$ with

$$
\begin{array}{lll}
a & : & \alpha\ b\ \beta\ ; \\
b & : & \gamma\ <<\phi>>?\ \rho
\end{array}
$$

For a hoisted predicate, ϕ, to be syntactically valid, the **lookahead** tokens must be in context set computed for ϕ.

This section described what information is required to successfully evaluate a predicate early in order to disambiguate a parsing decision. For more information regarding the C code templates generated by PCCTS to test lookahead sets, consult [PaD92].

11. Appendix **II** — Predicates: Scanner Versus Parser

One standard way to parse context-sensitive constructs in languages like C++ is to have a ‘‘predicated **scanner**'' in which the scanner returns different tokens for the same input, based on a predicate (symbol table information). We now show that a predicated parser eliminates the need for a predicated scanner.

A simple scanner is a finite automaton that maps regular expressions of the input into tokens, without access to other information, such as a symbol table. For example, the Unix utility, lex [Les75], generates simple scanners if there is no embedded C code or functions calls. The interface from a simple scanner to the parser is a one-way stream of tokens. A *predicated* scanner is a simple scanner augmented with semantic predicates, such as access to a symbol table, that can **affect** the tokens returned. The next theorem shows that putting predicates in the parser is more powerful than putting predicates in the scanner. Let $L(X)$ be the languages recognized by X.

Theorem: *L*(predicated *LR*(k) parser + simple scanner) $\supset$ *L*(a predicated scanner + simple *LR*(k) parser).

Proofs:

(i) A simple impractical proof is to note that a scanner is not strictly necessary, as the grammar can be augmented so that the parser converts the input characters into terminals representing the original tokens. Predicates called by the scanner would now be called by the parser in the **corresponding** places.

(ii) For the second proof, we consider how scanner predicates would be used to disambiguate a grammar and we show how to duplicate this effect in an *LR(k)* predicated parser. Assume the scanner returns lookahead λ_{true} or λ_{false} based on a predicate, *pred*. The non-predicated grammar must have *LR(k)* an item-set that uses λ_{true} or λ_{false} to choose between two actions.

$$(1) \quad a \; : \; \alpha \; \beta \; \text{\textbf{,}} \; \gamma, \quad \lambda_{false} \; \delta \quad \text{(shift)}$$
$$(2) \quad b \; : \; \beta \; \bullet, \quad\quad\quad \lambda_{true} \; \delta \quad \text{(reduce)}$$

Grammar for predicated scanner

We can duplicate this effect with a simple scanner and predicated parser that uses ϕ. The **lookahead** will be λ for both productions.

```
(1) a : a β • <<!phi>>? γ,     λ 6   (shift)
(2) b : β • <<φ>>?,            λ 6   (reduce)
```

Grammar for predicated parser

Without the use of *pred* in the grammar, a **shift/reduce** conflict would result, as after seeing β, production (2) indicates reduce via B : β, but production (1) indicates shift. **A** nearly identical argument applies when **reduce/reduce** conflicts would result.

A predicated parser is strictly more powerful a predicated scanner, because the parser can wait longer before calling a predicate, ϕ, allowing ϕ to use synthesized attributes of the **looka**-head. Assuming ϕ affects token λ_m, the scanner must apply ϕ before a λ_m is placed in the looka-head **buffer,** namely immediately after λ_{m-k} is seen. In contrast, the parser may not need to call ϕ until λ_m is the **next lookahead** token. $\square$

As an example, consider the following grammar fragment using a predicated scanner, which handles variable and type declarations in C, when *k=2.* In this fragment, k=1 suffices, but suppose elsewhere in the grammar k=2 is needed. Let TYPE represents a type name, and NAME represent an unbound name. The predicated scanner returns either TYPE or NAME when it sees a C identifier.

```
(1) var_decl  : TYPE NAME << add-NAME-as-variable >> ";"

(2) .type-decl : "typedef" type-spec NAME << add-NAME-as-a-new-type>> ";"

(3) decl_list : ( type-decl | var_decl )*
```

On the following input, we declare a new type **boolean** and a variable **flag** of type boolean. We show the tokens returned by a predicated scanner. The subscripts are simply for ease of reference. Unfortunately, NAME (instead of TYPE) is returned for the second occurrence of boolean, as the **boolean$_2$** becomes part of the **lookahead** immediately after **boolean$_1$** is returned, before the parser has a chance enter boolean in the symbol table. Thus, a predicated scanner cannot handle this case properly.

Input:	typedef	int	boolean,		boolean$_2$	flag	;
Tokens:	"typedef"	TYPE	NAME	","	NAME	NAME	";"

In contrast, by adding the two following productions using a parser predicate to resolve TYPE from NAME, we no longer have a problem, because the predicate is evaluated *after* the identifier in question is seen.

```
(4) type :    <<isType (LA (1))>>? T_ID
            ,
(5) name :    <<!isType (LA (1))>>? T_ID
```

As the scanner simply returns `T_ID`, the lookahead always consists of `T_ID`'s, and we rely on the correct nonterminal (`type` or `name`) to be on the stack.