Adding Semantic and Syntactic Predicates To $LL(k)$: *pred-LL(k)*[*]

Terence J. Parr[1] [**] and Russell W. Quong[2]

[1] University of Minnesota
Army High Performance Computing Research Center
parrt@acm.org
[2] Purdue University
School of Electrical Engineering
quong@ecn.purdue.edu

Abstract. Most language translation problems can be solved with existing $LALR(1)$ or $LL(k)$ language tools; e.g., YACC [Joh78] or ANTLR [PDC92]. However, there are language constructs that defy almost all parsing strategy commonly in use. Some of these constructs cannot be parsed without semantics, such as symbol table information, and some cannot be properly recognized without first examining the entire construct, that is we need "infinite lookahead."
In this paper, we introduce a new $LL(k)$ parser strategy, *pred-LL(k)*, that uses semantic or syntactic predicates to recognize language constructs when normal deterministic $LL(k)$ parsing breaks down. Semantic predicates indicate the semantic validity of applying a production; syntactic predicates are grammar fragments that describe a syntactic context that must be satisfied before application of an associated production is authorized. Throughout, we discuss the implementation of predicates in ANTLR—the parser generator of The Purdue Compiler-Construction Tool Set.

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. Ignoring arguments concerning the use of $LL(k)$ versus $LR(k)$ parsing strategies, it is often the case that semantic information (such as symbol table information) is required to parse a particular language correctly and naturally. While $LR(k)$-based parsers can be augmented with run time tests that alter the parse, bottom-up strategies have convenient access to much less semantic and context information than a top-down $LL(k)$ parser; hence, we have chosen to augment $LL(k)$ with predicates.

[*] *To appear at Int'l Conference on Compiler Construction; April, 1994*
[**] Partial support for this work has come from the Army Research Office contract number DAAL03-89-C-0038 with the Army High Performance Computing Research Center at the U of MN.

In this paper, we present *pred-LL(k)*, the class of languages recognized by conventional $LL(k)$ parsers augmented with semantic and syntactic predicates, which can specify the semantic and syntactic applicability of any given grammar production. *Semantic predicates* are run time tests that can resolve finite lookahead conflicts and syntactic ambiguities with semantic information. *Syntactic predicates* resolve finite lookahead conflicts by specifying a possibly infinite, possibly nonregular, lookahead language. Syntactic predicates are a form of selective backtracking that allow the recognition of constructs beyond the capabilities of conventional parsing; this capability is becoming necessary, e.g., [ES90] indicates that unbounded lookahead is required to correctly parse C++. We have implemented *pred-LL(k)* using an existing $LL(k)$ parser generator called ANTLR (the parser generator used in PCCTS—the Purdue Compiler-Construction Tool Set) [PCD93]. ANTLR is widely used tool; there are over 1000 registered users in over 37 countries and numerous universities are using it in compiler classes.

Our paper is organized as follows. Section 2 describes the previous work in this area and Section 3 provides numerous examples illustrating the utility of semantic and syntactic predicates. Section 4 describes the behavior of semantic and syntactic predicates more formally and how their introduction affects normal $LL(k)$ grammar analysis and parsing.

2 Previous Work

Attribute grammars have received attention in the literature since their introduction [Knu68] because they allow the specification of the grammar and the translation semantics in one description. Unfortunately translations implemented in this manner can be slow and many translations are difficult to express purely as functions of attributes; according to [Wai90], pure attribute grammars have had little impact on compiler construction.

[LRS74] considered the practical application of attribute grammars to compilers by characterizing the types of attribute grammars that could be efficiently handled via $LR(k)$ "bottom-up" and $LL(k)$ "top-down" parsing methods. They showed that $LL(k)$ has an advantage over $LR(k)$ in semantic flexibility (e.g., $LL(k)$ parsers may inherit attributes from invoking productions); despite this advantage of $LL(k)$, some researchers have argued that augmenting $LR(k)$ with predicates is more suitable than augmenting $LL(k)$ due to concerns over recognition strength (see LADE[3], YACC++[4], and [Gan89], [McK90]). However, with the addition of syntactic and semantic predicates, *pred-LL(k)* parsers can potentially recognize all context-free[5] and many context-sensitive languages beyond the ability of existing $LR(k)$ systems.

[3] LADE is a registered trademark of Xorian Technologies

[4] YACC++ is a registered trademark of Compiler Resources, Inc.

[5] ANTLR-generated parsers can be coerced into backtracking at all lookahead decisions to accomplish this, however, general backtracking is well known to be exponentially slow.

Other researchers have developed similar notions of predicated $LL(k)$ parsing. For example, [MF79] introduced the class of $ALL(k)$ grammars that could specify two types of semantic predicates, disambiguating and contextual, that were used to handle the context-sensitive portions of programming languages; the authors implemented an $ALL(1)$ parser generator [MKR79] based upon their $ALL(k)$ definition. Our approach differs from [MF79] in a number of ways. Whereas they 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 (the grammar analysis phase knows when a lookahead decision is nondeterministic and can search for semantic predicates that potentially resolve the conflict). 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. Further, the disambiguating predicates of [MF79] 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 (ANTLR grammar analysis can compute the k-tuples). Although in theory, the predicates of [MF79] and the predicates described herein are equivalent in recognition strength, in practice our predicates allow for more concise and more natural language descriptions; additionally, our predicate scheme has been implemented in an $LL(k)$ parser generator rather than in an $LL(1)$ parser generator, which dramatically increases the recognition strength of the underlying parsers.

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

3 Examples of Using Predicates

Programmers routinely handle many context-sensitive language constructs with context-free grammars through a variety of ah hoc "tricks" taught in most compilers courses. For example, when a syntactic structure is ambiguous, semantic information is often used by the lexical analyzer to return different token types for the same input symbol. In addition, grammars are often twisted (into an unreadable condition) in an effort to remove parser nondeterminisms and syntactic ambiguities. In this section, we provide two examples that illustrate the benefits of using semantic information in the parser rather than the lexical analyzer and we provide an example that illustrates how syntactic predicates can be used to resolve finite $LL(k)$ lookahead problems. In this paper, we use the terminology shown in the following table.

What	Example	Description
nonterminal	`a`, `varName`	starts with lower case
terminal	`ID`, `FOR`	starts with upper case
raw input	`"int" "for"`	regular expr in a string
action	`<< i++; >>`	enclosed in `<< ...>>`
predicate	`<< i==0 >>?`	construct followed by a `?`
semantic pred	`<< i==0 >>?`	enclosed in `<< ...>>?`
syntactic pred	`(declaration)?`	enclosed in `(...)?`
i^{th} lookahead symbol	`LA(3)`	`LA(`i`)`, $1 \leq i \leq k$
text of i^{th} lookahead symbol	`LATEXT(1)`	`LATEXT(`i`)`, $1 \leq i \leq k$

We now present some examples motivating the need for predicates during parsing. Consider FORTRAN array references and function calls, which are syntactically identical, but semantically very different:

```
expr : ID "\(" exprlist "\)" <<arrayref_action>>
     | ID "\(" exprlist "\)" <<fncall_action>>
     ;
```

There are two common approachs to resolving this ambiguity. First, one may merge the two alternatives and then examine the symbol table entry for the `ID` to determine which action to execute:

```
expr : ID "\(" exprlist "\)"
       <<if ( isvar($1) ) arrayref_action else fncall_action;>>
     ;
```

where the `$1` is the attribute (of type character string here) of `ID`.

In the second approach, the lexical analyzer modifies the token type according to whether the input `ID` is a variable or a function by examining the symbol table. The grammar then becomes context-free:

```
expr : VAR  "\(" exprlist "\)" <<arrayref_action>>
     | FUNC "\(" exprlist "\)" <<fncall_action>>
     ;
```

while these methods are adequate, they quickly become unmanageable as the complexity of the grammar increases. The lexical analyzer would be required to know more and more about the grammatical context in order to make decisions; in essence, the user has to hand code a significant part of the parser in the lexical analyzer. A more elegant solution is possible via semantic predicates, in which all code fragments pertaining to context-sensitivity are constrained to the grammar specification. The same expression that would normally be used to differentiate the two actions or to return different token types may be used to alter the normal $LL(k)$ parsing strategy by annotating the grammar, thereby, allowing array

references and function calls to be treated as grammatically different constructs, which was the original, fundamental goal:

```
expr : <<isvar(LATEXT(1))>>?  ID "\(" exprlist "\)" <<arrayref_action>>
     | <<isfunc(LATEXT(1))>>? ID "\(" exprlist "\)" <<fncall_action>>
     ;
```

where `LATEXT(1)` is the text of the first symbol of lookahead and the "?" suffix operator indicates that the preceding grammar element is a predicate. In this case, the predicates are semantic predicates that are to be used in the production prediction mechanism to differentiate the alternative productions.

As a more complicated example, consider the definition of classes in C++. Identifiers (`ID`s) are used as the class name, member function names and the constructor name. (In C++, a constructor is a special member function that has the same name as the class itself and does not have a return type.) For example,

```
class Box {
    Box() { /* constructor for class Box */ }
    draw() { /* member function */ }
    some_user_type val;  /* member variable */
};
```

A simplified grammar fragment for C++ class definitions such as:

```
class_def
    :   "class" ID "\{" ( member )+ "\}" ";"
    ;

member
    :   ID "\(" args "\)" func_body      /* constructor */
    |   ID "\(" args "\)" func_body      /* normal member function */
    |   ID declarator                    /* member variable */
    ;
```

is syntactically ambiguous (like the previous FORTRAN example), but the constructor requires special handling. The conventional method would be to have the lexical analyzer return different token types for the various `ID` references. There are numerous problems with this solution, but we will give only two here. First, because C++ class definitions can be nested, the lexical analyzer would need access to a stack of terminals that represent the enclosing class names. Only in this way can it decide between a normal function name and the class constructor. But, what is the purpose of the parser if the lexical analyzer is tracking the grammatical structure? Second, if the parser has lookahead depth greater than one, having the lexer return tokens based on semantic context would be problematic as symbols may not have been added to the symbol table before the lexical analyzer had to tokenize its input. E.g., when the lexer tokenizes `ID`, the symbol table must be up to date otherwise the lexer might incorrectly categorize `ID`. In the previous example, if k, the lookahead depth, equals 3, immediately

after seeing `"class"`, the lexer would have to tokenize `"Box"`, `"{"`, and `"Box"`. The second `"Box"` would not be recognized as the current class name, as we have not yet entered the class definition.

Semantic predicates, in contrast, would not be evaluated until the correct context—after the parser had passed the class header and had entered the class name into the symbol table—and could easily be added to `member` to resolve the ambiguity.

```
member[char *curclass]
    :   <<strcmp(curclass,LATEXT(1))==0>>?
        ID "\(" args "\)" func_body       /* constructor */
    |   <<!istypename(LATEXT(1) && strcmp(curclass,LATEXT(1))!=0>>?
        ID "\(" args "\)" func_body       /* normal member function */
    |   <<istypename(LATEXT(1))>>?
        ID declarator                     /* member variable */
    ;
```

where `istypename(LATEXT(1))` returns true if the text of the first symbol of lookahead is listed in the symbol table as a type name else it returns false; lookahead of $k = 2$ is used to differentiate between alternatives 1 and 3. Note that the current class can be passed into `member` because of the nature of top-down $LL(k)$ parsing and is used to determine whether a member function is the constructor. The predicates would be incorporated into the production prediction expressions and, hence, would resolve the syntactic ambiguity.

Occasionally a grammar developer is faced with a situation that is not syntactically ambiguous, but cannot be parsed with a normal $LL(k)$ parser. For the most part, these situations are finite lookahead nondeterminisms; i.e., with a finite lookahead buffer, the parser is unable to determine which of a set of alternative productions to predict. We again turn to C++ for a nasty parsing example. Quoting from Ellis and Stroustrup [ES90],

> "There is an ambiguity in the grammar involving *expression-statements* and *declarations*... The general cases cannot be resolved without backtracking... In particular, the lookahead needed to disambiguate this case is not limited."

The authors use the following examples to make their point, where `T` represents a type:

```
T(*a)->m=7;     // expression-statement with type cast to T
T(*a)(int);     // pointer to function declaration
```

Clearly, the two types of statements are not distinguishable from the left as an arbitrary number of symbols may be seen before a decision can be made; here, the `"->"` symbol is the first indication that the first example is a statement. Quoting Ellis and Stroustrup further,

> "In a parser with backtracking the disambiguating rule can be stated very simply:

1. If it looks like a *declaration*, it is; otherwise
2. if it looks like an *expression*, it is; otherwise
3. it is a syntax error."

The solution in ANTLR using syntactic predicates is simply to do exactly what Ellis and Stroustrup indicate:

```
stat:   (declaration)? declaration
    |   expression
    ;
```

The meaning of rule `stat` is exactly that of the last quote. Rule `stat` indicates that a `declaration` is the syntactic context that must be present for the rest of that production to succeed. As a shorthand, ANTLR allows the following alternative:

```
stat:   (declaration)?
    |   expression
    ;
```

which may be interpreted in a slightly different manner—"I am not sure if `declaration` will match; simply try it out and if it does not match try the next alternative." In either notation, `declaration` will be recognized twice upon a valid declaration, once as syntactic context and once during the actual parse. If an expression is found instead, the declaration rule will be attempted only once.

At this point, some readers may argue that syntactic predicates can render the parser non-linear in efficiency. While true, the speed reduction is small in most cases as the parser is mostly deterministic and, hence, near-linear in complexity. Naturally, care must be taken to avoid excess use of syntactic predicates. Further, it is better to have a capability that is slightly inefficient than not to have the capability at all. I.e., just because the use of syntactic predicates can be abused does not mean they should be omitted. In the following section, we formalize and generalize the notion of a predicate and discuss their implementation in ANTLR.

4 Predicated $LL(k)$ Parsing

We denote $LL(k)$ grammars that have been augmented with information concerning the semantic and syntactic context of lookahead decisions as *pred-*$LL(k)$. The semantic context of a parsing decision is the run time state consisting of attributes or other user-defined objects computed up until that moment; the syntactic context of a decision is a run time state referring to the string of symbols remaining on the input stream. As demonstrated in the previous section, *pred-*$LL(k)$ parsers can easily resolve many syntactic ambiguities and finite-lookahead insufficiencies. In this section, we discuss the behavior of semantic and syntactic predicates and describe the necessary modifications to the usual $LL(k)$ parsing.

4.1 Semantic Predicates

A semantic predicate is a user-defined action that evaluates to either true (success) or false (failure) and, broadly speaking, indicates the semantic validity of continuing with the parse beyond the predicate. Semantic predicates are specified via "<<*predicate*>>?" and may be interspersed among the grammar elements on the right hand side of productions like normal actions. For example,

```
typename
    : <<istype(LATEXT(1))>>?  ID
    ;
```

defines a rule that recognizes an identifier (`ID`) in a specific semantic context, namely `ID` must be a type name. We have assumed that `istype()` is a function defined by the user that returns true *iff* its argument is a type name determined by examining the symbol table. If rule `typename` were attempted with input `ID`, but that `ID` was not listed as a type name in the symbol table, a run-time parsing error would be reported; the user may specify an error action if the default error reporting is unsatisfactory. The predicate in this role performs *semantic validation*, but the very same predicate can take on a different role depending on the how `typename` is referenced. Consider the following C++ fragment, in which we need to distinguish between a variable declaration and a function prototype:

```
typedef int T;
const int i=3;
int f(i);    //  same as int f = i;  initialize f with i
int g(T);    // function prototype; same as " int g(int);"
```

The third declaration defines `f` to be an integer variable initialized to `3` whereas the last declaration indicates `g` is a function taking an argument of type `T` and returning an integer. The unpleasant fact is that there is no syntactic difference between the declarations of `f` and `g`; there is only a semantic difference, which is the type of the object enclosed in parentheses. Fortunately, a *pred-LL*(k) grammar for this small piece of C++ (i.e. for the last two declarations) can be written easily as follows,

```
def :   "int" ID ( var      <<var_decl_action>>
                 | typename <<fn_proto_action>>
                 )
        ";"
    ;

var :   <<isvar(LATEXT(1))>>?  ID
    ;

typename
    :   <<istype(LATEXT(1))>>?  ID
    ;
```

As before with `typename`, we define the rule `var` to describe the syntax and semantics of a variable. While `var` and `typename` are completed specified, the subrule in `def` appears syntactically ambiguous because in both cases the token stream (for the whole rule) would be `"int"`, `ID`, `"("`, `ID`, `")"`, `";"`, because `ID` would match both alternatives `var` and `typename` of the subrule. However, the subrule in `def` has referenced rules containing predicates that indicate their semantic validity. This information is available to resolve the subrule's ambiguous decision. Thus, the same predicates that validate `var` and `typename` can be used as *disambiguating predicates* in rule `def`. The action of copying a predicate upward to a pending production to disambiguate a decision is called *hoisting*. E.g., we must hoist the `<<isvar()>>?` predicate from the rule `var` upwards into rule `def`. Essentially, disambiguating predicates "filter" alternative productions in and out depending on their semantic "applicability." Productions without predicates have an implied predicate of `<<TRUE>>?`; i.e., they are always assumed to be valid semantically. Predicates are all considered validation predicates as they must always evaluate to true for parsing of the enclosing production to continue, but occasionally they are hoisted to aid in the parsing process. When ANTLR's grammar analysis phase detects a syntactic ambiguity (actually, any non-$LL(k)$ construct), it searches for *visible* semantic predicates that could be hoisted to resolve the ambiguous decision; the exact definition of visible is provided in a future section. In this way, ANTLR automatically determines which role each predicate should assume.

An important feature is that the hoisting of semantic predicates cannot result in nonlinear parsing complexity—the parser is only charged for the time to initiate execution of a predicate. There are a finite number of hoisted semantic predicates known at analysis time; the user actions within do not count just as normal actions executed from within the parser are not considered to effect fundamental parser complexity. In the following subsections, we describe more precisely the notions of *pred-*$LL(k)$ grammar analysis, predicate hoisting, and predicate context.

***pred-LL*(k) Parsing Strategy.** Semantic predicates, both validation and disambiguating, are easily added to the normal $LL(k)$ parsing strategy. Validation predicates are used as simple tests to determine whether the input is semantically correct and do not alter the $LL(k)$ parse. The predicate `<<`π`>>?` is functionally equivalent to the following normal action embedded in a grammar:

`<<if ( !`π `) {` *call standard predicate failure routine*`}>>`

Disambiguating predicates, namely those which are hoisted to resolve a syntactic ambiguity, are incorporated into the normal production prediction expression. For example, in the previous case regarding the C++ variable versus function prototype, ANTLR would generate the following C code for the the syntactically ambiguous subrule contained in rule `def`:

```
if ( LA(1)==ID && isvar(LATEXT(1)) ) {
    var();
} else if ( LA(1)==ID && istype(LATEXT(1)) ) {
    typename();
} else
     parse error;
```

Note that ANTLR-generated parsers attempt productions in the order specified. Parsing in this new environment can be conveniently viewed in the following manner:

1. *Disable invalid productions.* An *invalid* production is a production whose disambiguating predicate(s) evaluates to false.
2. *Parse as normal subrule.* Once a list of *valid* productions has been found, parse them according to normal $LL(k)$ rules.

In general, predicates must follow the following three rules for parsing to behave as we have described:

1. A predicate referenced in rule `a` can be a function only of its left context and tokens of its right context that will be within the lookahead buffer available at the left edge of `a`. When syntactic predicates are used, this lookahead buffer may be arbitrarily large.
2. Predicates may not have side-effects.
3. Disambiguating predicates may not be a function of semantic actions situated between themselves and the syntactically ambiguous decision. E.g., a predicate cannot depend on an action over which it will be hoisted; an action provides an easy way to prevent a predicate from being hoisted.

***pred-LL(k)* Grammar Analysis and Hoisting.** Semantic predicates are incorporated into the parsing process (hoisted) when grammar analysis indicates that normal $LL(k)$ is insufficient to differentiate alternative productions. This section provides a glimpse into how $LL(k)$ analysis is augmented to automatically determine predicate roles.

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

```
a   :   α
    |   β
    ;
```

It is well known that `a` is non-$LL(k)$ *iff* α and β generate phrases with at least one common k-symbol prefix; i.e., for $\mathtt{s} \Rightarrow^*_{lm} w\mathtt{a}\delta$, $T = FIRST_k(\alpha\delta) \cap FIRST_k(\beta\delta) \neq \emptyset$ where T represents the set of k-tuples that predict both productions, w is a terminal string, δ is a terminal and nonterminal string, `s` is the start symbol, and $\Rightarrow^*_{lm}$ is the closure of the usual leftmost derivation operator. In order to define *pred-*$LL(k)$, the set of predicates that are candidates for hoisting

into a prediction expression must be described. Consider the following grammar fragment.

```
a   :   b β | γ ;
b   :   <<π>>? α
```

A predicate is *visible* from a decision, such as that in `a`, if it can be evaluated without consuming a symbol of lookahead[6] and without executing a user action; i.e., all visible predicates appear on the left edge of productions derivable from `b` and γ, but predicates at the left edge of β would not be visible if α derives at least one token.

Rule `a` is *pred-*$LL(k)$ *iff* it is $LL(k)$ (T is empty) or the set of visible predicates for each production *covers* T. A predicate, π, covers a tuple, t, *iff* $t \in$ context(π) where context(π) is the set of lookahead k-tuples that predict the production from which π was hoisted; e.g., context(π) above is $FIRST_k(\alpha\beta\delta)$ where `s` $\Rightarrow^*_{lm}$ w`a`δ and `s` is the start symbol. Only those predicates that cover a tuple in T are used for disambiguation and if T is incompletely covered for a production (there exists a k-tuple t with no covering predicate), the enclosing decision is non-*pred-*$LL(k)$ and an ambiguity warning is given to the grammar developer. Further, to yield a deterministic *pred-*$LL(k)$ parser, exactly one syntactically viable production must be semantically valid (its visible predicates all evaluate to true); the predicates for any other syntactically viable productions must not succeed. Turning again to our ambiguous subrule in `def`, T is {`ID`}, both predicates are visible, their context is {`ID`}, and we note that, in that simplified grammar, we assume an `ID` can never be both a variable and a typename.

Hoisting a predicate, π, into a prediction expression in another rule is not as simple as copying the predicate. The predicate should only be evaluated under the syntactic context in which it was found—the current lookahead buffer must be a member of the context computed for π; for an example illustrating why predicate context is necessary see the ANTLR 1.10 release notes [PCD93].

Using our complete terminology, we can now summarize our *pred-*$LL(k)$ analysis approach: When a non-$LL(k)$ decision is encountered, all visible, covering predicates are hoisted, along with their context, into the prediction expressions for that decision; such predicates assume the role of disambiguating semantic predicate.

4.2 Syntactic Predicates

We saw in previous sections how disambiguating semantic predicates can be used to resolve many syntactic ambiguities. However, there are a number of non-$LL(k)$, unambiguous, grammatical constructs that semantic information cannot

[6] A generalization of this simple definition, allows hoisting of predicates up to k symbols ahead. We define the *hoisting distance* to be how many terminal symbols exist between the ambiguous decision and the predicate; in this terminology, our discussion in this paper is limited to hoisting distances of 0.

resolve. The most obvious example would be left-recursion, but left-recursion can be removed by well-known algorithms. The nastiest grammar construct is one in which two alternative productions cannot be distinguished without examining all or most of the production. While left-factoring can handle many of these cases, some cannot be handled due to action placement, non-identical left-factors, or alternatives productions that cannot be reorganized into the same rule. The *pred-*$LL(k)$ solution to the problem of arbitrarily-large common left-factors is simply to use arbitrary lookahead; i.e., as much lookahead as necessary to uniquely determine which production to apply. In this section we introduce *syntactic* predicates that, like disambiguating semantic predicates, indicate when a production is a candidate for recognition; the difference lies in the type of information used to predict alternative productions—syntactic predicates employ structural information rather than information about the "meaning" of the input.

Syntactic predicates are specified via "`(` α `)?`" and may appear on the left edge of any production of a rule or subrule. The required syntactic condition, α, may be any valid context-free grammar fragment except that new rules may not be defined. Consider how one might write a grammar to differentiate between multiple-assignment statements and simple lists such as:

```
(a,b) = (3,4);
(apple, orange);
```

One obvious grammar is the following:

```
stat:   list "=" list ";"
    |   list ";"
    ;
```

where `list` is a rule, defined elsewhere, that recognizes an arbitrarily long list of expressions. The grammar is not $LL(k)$ for any finite k, unfortunately, due to the common left-factor. E.g., upon seeing `"(red, green, blue, ...,"` an $LL(k)$ parser does not know which alternative to choose. Left-factoring would resolve this problem, but would result in a less readable grammar. We have found that having to constantly manually left factor rules leads to confusing and non-natural grammars. Furthermore, if we assume that the grammar cannot be factored because actions are needed on the left edge of the productions, nothing can be done to resolve the lookahead decision with normal $LL(k)$. In contrast, the nondeterministic decision is easily resolved through the use of a single syntactic predicate:

```
stat:   ( list "=" )?  <<action1>>  list "=" list ";"
    |                  <<action2>>  list ";"
    ;
```

The predicate specifies that the first production is only valid if "`list "="`" is consistent with (matches) an arbitrarily large portion of the infinite lookahead

buffer. ANTLR assumes that the production prediction expressions are evaluated in the order specified and, hence, the second production is the default case to attempt if the syntactic predicate `( list "=" )?` fails. A short form of the syntactic predicate exists that would allow a functionally equivalent, but less efficient, formalization of `stat`:

```
stat:   ( list "=" list ";" )?
    |   list ";"
    ;
```

This can be interpreted in a slightly different manner—that the first production may not match the input, but rather than reporting a parsing error, try the next viable production (here, the second production is also predicted by the next k symbols of lookahead and is, therefore, considered viable).

While syntactic predicates are an elegant means of extending the recognition strength of conventional $LL(k)$ through selective backtracking, one might argue that a $LALR(k)$ parser automatically left-factors alternative productions obviating the need for the previous syntactic predicates. However, recall that we assumed actions were inserted on the left edge of the two productions to inhibit left-factoring. After only a few moments thought, rule `stat` (with actions) is seen to be non-$LALR(1)$ because rule "cracking" forces actions to production right edges—producing a reduce-reduce conflict.

Because syntactic predicates are, by definition, not guaranteed to match the current input, they have an effect on user actions, semantic predicate evaluation, and normal $LL(k)$ grammar analysis. The following subsections briefly describe the issues in these areas.

Syntactic Predicates effect upon Actions and Semantic Predicates. While evaluating a syntactic predicate, user actions, such as adding symbol table entries, are not executed because in general, they cannot be "undone;" this conservative approach avoids affecting the parser state in an irreversible manner. Upon successful evaluation of a syntactic predicate, actions are once again enabled—unless the parser was in the process of evaluating another syntactic predicate (syntactic predicates may invoke rules that themselves evaluate syntactic predicates).

Because semantic predicates are restricted to side-effect-free expressions, they are always evaluated when encountered. However, during syntactic predicate evaluation, the semantic predicates that are evaluated must be functions of values computed when actions were enabled. For example, if your grammar has semantic predicates that examine the symbol table, all symbols needed to direct the parse during syntactic predicate evaluation must be entered into the table before this backtracking phase has begun.

Syntactic Predicates effect upon Grammar Analysis. ANTLR constructs normal $LL(k)$ decisions throughout predicated parsers, only resorting to arbi-

trary lookahead predictors when necessary. Calculating the lookahead sets for a full $LL(k)$ parsers can be quite expensive, so that, by default, ANTLR uses a linear approximation to the lookahead, called $LL^1(k)$ and only uses full $LL(k)$ analysis when required[7]. When ANTLR encounters a syntactic predicate, it generates the instructions for selective backtracking as you would expect, but also generates an $LL^1(k)$ decision. Although no finite lookahead decision is actually required (the arbitrary lookahead mechanism will accurately predict the production without it) the $LL^1(k)$ portion of the decision reduces the number of times backtracking is attempted without hope of a successful match. For example, referring to the C++ declaration versus expression grammar example in Section 3, if the current input token were `"42"`, rule `stat` would immediately attempt the second production—`expression`. On the other hand, if the current input token were `"abc"`, then the `declaration` rule would be attempted before attempting `expression`. If neither productions successfully matched the input, a syntax error would occur.

An unexpected, but important benefit of syntactic predicates is that they provide a convenient method for preventing ANTLR from attempting full $LL(k)$ analysis when doing so would cause unacceptable analysis delays.

5 Implementation

We have implemented a *pred-*$LL(k)$ parser generator in the latest version of ANTLR, which is part of the Purdue Compiler Construction Tool Set (PCCTS). Due to space constrains, we will discuss the ANTLR implementation of *pred-*$LL(k)$ in a future paper.

ANTLR is not just a research tool; many industrial and academic sites are currently using ANTLR for everyday use. Readers interested in obtaining ANTLR and other the other tools in PCCTS may contact the email server at `pccts@ecn.purdue.edu`.

6 Conclusion

In this paper, we have introduced a new predicated $LL(k)$ parser strategy, *pred-*$LL(k)$, that uses semantic and syntactic predicates to resolve syntactic ambiguities and parsing conflicts due to the limitations of finite lookahead. In theory, *pred-*$LL(k)$ can recognize all context-free languages (albeit, expensively) as well as many context-sensitive languages. Moreover, these methods have been cleanly integrated into ANTLR—a widely used public-domain $LL(k)$ parser generator. We believe that *pred-*$LL(k)$ represents a significant advance toward the development of natural, easy to read, grammars for difficult languages like C++.

[7] Polynomial approximation of higher degree, $LL^m(k)$, are also possible [Par93], but $LL^1(k)$ is sufficient in practice.

7 Acknowledgements

We would like to thank Ariel Tamches, Dana Hoggatt, Ed Harfmann, and Hank Dietz, who all helped shape the definition of semantic predicates.

References

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley Publishing Company, Reading, Massachusetts, 1990.

[Gan89] Mahadevan Ganapathi. Semantic Predicates in Parser Generators. *Computer Language*, 14(1):25–33, 1989.

[HCW82] R. C. Holt, J. R. Cordy, and D. B. Wortman. An Introduction to S/SL: Syntax/Semantic Language. *ACM TOPLAS*, 4(2):149–178, April 1982.

[Joh78] S. C. Johnson. *Yacc: Yet Another Compiler-Compiler.* Bell Laboratories; Murray Hill, NJ, 1978.

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[LRS74] P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns. Attributed Translations. *Journal of Computer and System Sciences*, 9:279–307, 1974.

[McK90] B. J. McKenzie. LR parsing of CFGs with restrictions. *Software–Practice & Experience*, 20(8):823–832, 1990.

[MF79] D.R. Milton and C.N. Fischer. $LL(k)$ Parsing for Attributed Grammars. In *Proceedings of Automata, Languages and Programming, Sixth Colloquium*, pages 422–430, 1979.

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

[Par93] Terence John Parr. *Obtaining Practical Variants of $LL(k)$ and $LR(k)$ for $k > 1$ by Splitting the Atomic k-Tuple.* PhD thesis, Purdue University, West Lafayette, Indiana, August 1993.

[PCD93] Terence Parr, Will Cohen, and Hank Dietz. The Purdue Compiler Construction Tool Set: Version 1.10 Release Notes. Technical Report Preprint No. 93-088, Army High Performance Computing Research Center, August 1993.

[PDC92] T.J. Parr, H.G. Dietz, and W.E. Cohen. PCCTS 1.00: The Purdue Compiler Construction Tool Set. *SIGPLAN Notices*, 1992.

[Wai90] W. M. Waite. Use of Attribute Grammars in Compiler Construction. In *Attribute Grammars and their Applications; Lecture Notes in Computer Science*, volume 461, pages 254–265. Springer-Verlag, 1990.

This article was processed using the LaTeX macro package with LLNCS style