

# Parsing Non-LR( $k$ ) Grammars with Yacc

GARY H. MERRILL

*SAS Institute Inc., SAS Campus Drive, Cary NC 27513, U.S.A.*

## SUMMARY

**Of the parser generating tools currently in use, yacc (or one of its several variants) is perhaps the most frequently employed. However, because of inherent ambiguities there are some languages (such as C++) that a yacc-generated parser cannot successfully compile. This paper describes a set of minor modifications to yacc-like tools that allows them to be used in a straightforward way to parse ambiguities and, more generally, grammars that require an indefinite amount of lookahead. Required changes to the lexical analyzer are also discussed, and the application of these techniques is illustrated within the context of specific examples.**

KEY WORDS: Yacc Parsers Parser generators Compilers Ambiguity

## INTRODUCTION

Among parser generators in common use today, the UNIX\* tool yacc is probably the most commonly used.<sup>1–3</sup> Yacc offers a number of obvious advantages to the compiler writer, including the following:

1. Yacc generates a powerful table-driven LALR(1) parser.
2. For relatively simple languages, writing a parser in yacc amounts to little more than simply writing the grammar for the language.
3. For more complex languages, yacc provides for a certain degree of explicit control in such areas as precedence and error handling and recovery.
4. A yacc parser is easily modified and maintained since the grammar of the language being parsed is explicit in the yacc input file.

These features all recommend the use of yacc, and a large number of academic and commercial products have been written with this tool.

Even given this list of impressive features and the fact that many successful products have been created with yacc, our experience shows that there are a number of serious disadvantages to yacc in favor of other parsing techniques (such as a handwritten recursive descent parser). Among these are:

1. Because of the ‘high level’ nature of the yacc language, it is frequently quite difficult to exert the desired degree of ‘local control’ over the action of the parser. A variety of tricks must be employed that amount to ‘fooling’ the parser into doing something it would not normally be inclined to do.

---

\* UNIX is a registered trademark of UNIX System Laboratories Inc.

2. A great deal of skill and effort is required in order to achieve what we regard as a satisfactory implementation of error detection and recovery at the syntactic level. In general it is easy to tell when a compiler has been written with yacc because of the brevity and lack of useful information in its messages pertaining to syntax errors. This is particularly frustrating in a commercial product where the quality of error, warning, or informational messages is of crucial importance.
3. The ease with which a yacc parser can be modified and maintained is frequently deceptive. This is because what appears to be a small change may have a 'ripple effect' that requires modifying a large number of productions or rewriting a portion of the grammar. On reflection, this should not be too surprising, since in the case of yacc (and other parser generators) one fundamentally writes the parser by writing the grammar, but a consequence of this is that a higher level of knowledge and skill is required to maintain and extend such a parser than to perform similar actions on a handwritten recursive descent compiler.
4. Yacc lacks support for resolving ambiguities in the language for which it is attempting to generate a parser. It does have a simple-minded approach to resolving shift/reduce and reduce/reduce conflicts, but this is not of sufficient power to solve the really thorny problems encountered in a genuinely ambiguous language.

Our own recent experience in using yacc on a significant programming project involved writing a translator for the C++ language. Initially yacc appealed to us because of the speed with which we believed we could construct a yacc-generated parser for the language, and indeed this belief appeared to be justified by rapid progress in that direction. We knew of the problems of ambiguity in C++ but felt that we could work around these within the framework of a yacc parser. After all, we also knew that others had used yacc to write C++ translators or compilers. What we did not know (but learned as time went by and the language evolved) was that these attempts were not entirely successful. We also felt that we had the ability to rewrite the parser without using yacc should this prove to be necessary.

One approach we considered was the construction of a C++ grammar that would be fully 'yacc-compatible' in the sense that it would be unambiguous and LR(1).<sup>\*</sup> We dismissed this approach for a variety of reasons. We felt that the resulting grammar would simply transfer the ambiguity problem to a different level. It seemed clear that such a grammar would employ syntactic classes so disparate from those of the published grammar (and the one likely to be enshrined in a language standard) that maintenance and modifications of the parser would be made unnecessarily difficult. And, finally, we felt that the use of such a grammar would result in inefficiencies of both space and time, since it appeared that a rather complete parse tree for a construct would need to be stored before semantic actions of a specific enough nature could be applied to some of its parts (those that would normally be thought of as ambiguous relative to the published grammar). Subsequent experiences of those who have attempted to use such a grammar for more than mere parsing seem to have borne out our initial suspicions.

When the time came to confront the difficulties of ambiguity in C++, serious

---

<sup>\*</sup> A grammar along these lines has in fact been written by James Roskind of Roskind Computing.

consideration was given to rewriting the parser and abandoning yacc. By this time, however, we had a yacc grammar that allowed us to handle almost all of C++, that had been undergoing fairly thorough testing, and that interfaced nicely with our semantic routines. Abandoning yacc at this point would have meant developing a new grammar along the lines of an LL grammar rather than the LR grammar fancied by yacc. This meant that we would have to develop a new grammar, write a parser for the grammar and interface it with our semantic routines, and of course replace our approach to error detection, reporting, and recovery. And all of this would need to be tested as well. At that point in our development cycle, and given the deadlines we had imposed on development and testing, we decided to stick with our existing yacc parser and do whatever was necessary to make it work.

In the sections that follow I will describe the changes to yacc, the grammar, and the lexical analyzer that allow the use of yacc in parsing non-LR(*k*) grammars. Wherever possible I will provide an illustration of the type of problem confronting us in C++, but it will be apparent that the technique is fully general and is not restricted to a particular language or type of ambiguity. There is nothing particularly innovative in a theoretical sense about the approach to be described since it is in fact an implementation of what is commonly referred to as *backtrack parsing*.<sup>\*</sup> What this approach offers is a relatively low-cost and practical alternative in solving problems associated with parsing non-LR(*k*) grammars. My goal here is to provide the reader with enough detail to understand and implement these changes while avoiding excessive detail peculiar to our own implementation and application.

### THE PROBLEM: AN ILLUSTRATION

The type of problem at issue here is easily illustrated by considering difficulties in parsing C++. One type of syntactic ambiguity found in C++ is the expression-statement/declaration-statement ambiguity. An expression-statement that begins with a function-style explicit type conversion cannot be distinguished (at that point) from a declaration in which the first declarator starts with a '(' . Typical examples given are

```
T(a)->m = 7;           // expression-statement
T(a)++;               // expression-statement

T(a) = { 0, 1 };      // declaration
T(f)[ ];              // declaration
```

and it may be necessary to examine the entire statement to determine whether such a construct is an expression or a declaration. In other cases, such as

```
T(a);
T(x) = 1;
```

---

<sup>\*</sup> For a theoretical discussion of this topic, see Reference 4. An approach similar to the one taken here, but with an underlying recursive descent parser, is described briefly in Reference 5.

No amount of parsing will tell the tale, and it is necessary to adopt a disambiguating convention: if something of this sort looks like a declaration, it is.\*

In discussing this problem Stroustrup remarks that

Note that simple lexical lookahead can help a parser disambiguate most cases. Consider analyzing a statement consisting of a sequence of tokens as follows:

*type-name (d-or-e) tail*

Here, *d-or-e* must be a declarator, an expression, or both for the statement to be legal. This implies that *tail* must be a semicolon, something that can follow a parenthesized declarator or something that can follow a parenthesized expression, that is, an initializer, const, volatile, (, [, or a postfix or infix operator.

The general cases cannot be resolved without backtracking, nested grammars or similar advanced parsing strategies. In particular, the lookahead needed to disambiguate this case is not limited (Reference 6, p. 93).

This leads to a related ambiguity in declaration contexts between, on the one hand, a function declaration with a redundant pair of parentheses surrounding its argument and, on the other, the declaration of an object by means of a function-style cast as the initializer, as shown in Figure 1.

Syntactically, there are two choices here:

1. A function *x* is being declared to take a single `int` parameter and return an object of type `S`; or
2. An object *x* of type `S` is being declared and initialized to *y* (cast to an `int`).

In the first case, the construct `int(y)` is a declaration of *y*, whereas in the second case it is a cast of *y*. Given the disambiguating convention, the former is chosen and the construct is taken to be a function declaration.

Note again that merely by seeing input of the form `type-name(` the parser cannot tell whether it is looking at a declaration or an expression. In the somewhat more

```

struct S {
    S(int); // constructor for class S
};

void fcn(int i)
{
    S x(int(y)); // function declaration or object declaration?
    // ...
}

```

Figure 1. Example of ambiguity in C++

---

\* Strictly speaking, the first set of examples here are not illustrations of genuine ambiguity (in the sense of permitting multiple parse trees). Rather, they are examples of how the usual published grammars for C++<sup>4</sup> are not LR(*k*). None the less, Stroustrup and others refer to these as ambiguities in a looser sense, and I continue that harmless practice here since it facilitates the discussion.

complex case in [Figure 2](#) the parser cannot know until it encounters the '+2' in the third argument that the previous constructs it has seen must be interpreted as casts rather than declarations. Again, no definite amount of lookahead suffices to determine such choices.

### RESOLVING AMBIGUITIES WITH TRIAL PARSING

The fundamental idea behind our approach to ambiguity resolution using yacc is quite simple:

When a potential ambiguity is detected, attempt to parse it as one of the possible alternatives. If this *trial parse* succeeds, then continue on. If the *trial parse* fails, then back up and attempt to parse it as the other alternative.

Notice that although this description is phrased in terms of two possible parses for the ambiguous construct in question, it is generalizable to an arbitrary number of alternatives. For the sake of simplicity throughout the remainder of this paper, we will assume that the ambiguity involves only two possibilities.

Before proceeding to a specific example I shall first discuss the modifications made to yacc and the lexical analyzer in support of trial parsing.

### A YACC BY ANY OTHER NAME

In the old days there was only one true yacc,<sup>1</sup> but times have changed and quite a variety of yaccs and yacc-like tools have been spawned by both academic and commercial establishments.<sup>2</sup> Although these tools share a certain cultural heritage and feature set, there are differences (sometimes significant ones) in terms of features, design, and implementation. The modifications discussed throughout the remainder of this paper may be applied to any yacc-like tool, but details of implementation will of course vary. In order to make the discussion coherent and useful, I shall proceed from the perspective of a single implementation of yacc. Applying similar modifications to other versions should be a straightforward exercise.

We began by using a version of UNIX yacc supplied with the standard UNIX tools on APOLLO\* workstations. As our development progressed we began to encounter odd problems with this tool: a small addition or change to the grammar would cause yacc to fail in a mysterious way on its input file. As the result of some experimentation (and based on experience with other tools of this sort) we

```

struct R {
    R(int, int, int); // constructor for R
};

void fcn(int i)
{
    R x(int(a), int(b), int(c) + 2); // object declaration
    // ...
}

```

*Figure 2. A more complex ambiguity example*

---

\* APOLLO is a registered trademark of Hewlett-Packard Co.

developed the hypothesis that this yacc was having trouble because of the size of our grammar (now over 700 productions). Since we did not have the source code for yacc, we faced the decision of either abandoning it for another tool or attempting to have this and subsequent bugs fixed by the suppliers of the tool. Being tool developers ourselves, we chose the former course.

Both bison (a product of the Free Software Foundation, Inc.) and Berkeley yacc (byacc) were readily available in source form, but only the latter came unencumbered by various ideologically driven restrictions on its use and the use of products created with its aid. Since we were developing a commercial product, we chose byacc. Our lexical analyzer is handwritten in C.

The discussion in later sections will be rendered more intelligible by a brief description of how the source for our base version of byacc was organized, the file names it uses, and how its output file is generated. I will provide only the minimal details necessary to enable an easy extension of my remarks to later versions of byacc and other yaccs. Throughout the remainder of the paper I will use the generic 'yacc' to refer to both the base version of byacc with which we started and its evolutionary stages.

In terms of the source files comprising yacc, only three are of importance for the modifications to be described:

- reader.c     This file contains a number of functions that support parsing and analysis of the yacc input file.
- output.c     This file contains a number of functions that support the emission of C source code to the yacc output files.
- skeleton.c   This file contains several arrays of strings that are used by yacc to create the yyparse() parser driver in its C source output file. I shall sometimes refer informally to this code as 'the parser skeleton'.

Of the files that yacc may generate, two are of interest for our purposes:

- y.tab.h     Internally, yacc thinks of this as the 'defines' file. It is populated with declarations of various variables and functions that the user may need to reference in order to control the parser. I shall sometimes refer to this file informally as 'the parser header'.
- y.tab.c     This is the C code for the parser that is generated by yacc. This file contains the parsing tables, the driver (yyparse()), and any additional source code the user may have specified in the yacc input file. I shall sometimes refer to this file informally as 'the parser source'.

With these details of source organization and nomenclature out of the way we can now proceed to a description of how support for trial parsing may be added to yacc.

### SUPPORT FOR TRIAL PARSING IN YACC

In order to support this approach to trial parsing, several simple but significant modifications were made to yacc.

**Recursively callable** `yyparse()`

The only thing prohibiting reasonable recursive calls to `yyparse()` is the fact that a certain set of variables are defined with global scope.\* These were collected together into a `PARSE_STATE` structure that contains all of the data pertaining to what might be thought of as the state of a single invocation of `yyparse()`. A new variable used to record the current size of the parser's stack (explained below) is also included in the `PARSE_STATE`.

In order to implement these changes without having to make wholesale editing changes in existing skeleton code, a number of `#defines` were added so that the structure fields could be accessed by using symbols that previously had been global variables. Finally, a declaration of the `PARSE_STATE` and an initialization of it were added to the beginning of `yyparse()`.

**Conditional and unconditional actions**

When a trial parse is being attempted we typically do not want the action associated with a production to be executed. Usually such actions result in the execution of various semantic routines that allocate data structures, add entries to tables, and otherwise perform a variety of bookkeeping tasks associated with compilation. In the case of a trial parse the attempted parse may fail, and then we would be required to 'undo' the sequence of actions that were executed during this abortive attempt. Such an approach would be terribly inefficient and would require a great deal of additional code to keep track of the actions we might later want to undo. If we consider the possibility that trial parses themselves might be recursive in nature, then such a scenario is especially unattractive. Finally, we do not want a trial parse to result in the generation of error or warning messages, since a later parse of the same construct may be successful and render such diagnostics spurious. All of these considerations argue that we should be able to suppress the execution of actions during a trial parse.

One approach to action suppression would be to require the programmer to code this explicitly in the parser specification file. Thus, instead of coding

```
x: y
    { /* action */ }
```

we could code

```
x: y
    { if ( !trial_parse ) { /* action */ } }
```

and switch the variable `trial_parse` on and off accordingly as we are trial parsing or not.

This solution to the problem is simple and offers the advantage that it requires no modification to the parser generator. However, in practice it is quite cumbersome,

---

\* Apparently, by means of an option bison supports recursive calls to `yyparse()`.<sup>6</sup> Since we decided against using bison we have not investigated how this feature is implemented or whether it might be used in the manner described here.

error prone, and burdensome to implement in a large existing parser when the need for trial parsing is discovered at a late stage in development. A better and more general approach is to have the parser generator emit code that makes the execution of actions conditional.

In the parser driver that yacc constructs, actions appear in the following context:

```
case n:
    {
        /* action */
    }
```

We simply alter the parser generator so that this becomes:

```
case n:
    #if YY_TRIAL_PARSE
        if (!yy_trial_parse )
    #endif
    {
        /* action */
    }
```

where the test is embedded in a conditional compilation context so that it will not appear if trial parsing is not necessary. The variable `yy_trial_parse` is defined as an int at global scope and an extern declaration of it is emitted into the header file generated by yacc. In this way the programmer has control over the value of this variable (which is incremented on entry to a trial parse and decremented on return) and hence the suppression of actions. We now refer to such actions as *conditional actions*.

Although we might desire the normal semantic actions of a parser to be suppressed during trial parsing, there may none the less be circumstances in which we require an action to be executed even during (or especially during) a trial parse. In fact, one such action is absolutely critical to the correct use of trial parsing: when a trial parse succeeds we will force `yparse()` to return by means of `YYACCEPT`, and this must be an unconditional action. Moreover, as mentioned previously, whether a given construction is ambiguous depends to at least a certain extent on context. Thus, for example,

T(a);

will be ambiguous in those places where a statement may occur, but not elsewhere (for example, at file scope, after an operator, etc.). As we shall see, it is the lexer's responsibility to determine when a trial parse is called for, but in order to make correct and efficient decisions about this the lexer needs some help from the parser (which has access to context information). We accomplish this by setting the variable `possible_ambiguity` to indicate a context in which ambiguity is possible, and resetting this variable when we are not within such a context. Since ambiguities may be nested, it is necessary to be able to perform these actions during a trial parse, and this requires that such actions be unconditional.

The problem with unconditional actions pertains to their relationship to conditional actions that apply to the same production. You easily might desire to have both a conditional and unconditional action apply to the same production:

```
a: b
    [ /* unconditional action */ ]
    { /* conditional action */ }
```

For example, during a trial parse you might want to retain context information by keeping track of what productions have been performed and in what order, whereas during a non-trial parse of the same construct you would want the usual semantic actions to take place as well. This is reasonable, and at least our version of yacc supported such rules as

```
a: b
    { /* action 1 */ }
    { /* action 2 */ }
```

in fundamentally the same manner as it supported rules with embedded actions:

```
a: b
    { /* intermediate action */ }
    c
    { /* final action */ }
```

The way in which yacc does this is to add additional nonterminals and null rules so that such a rule becomes

```
a: b $$ c
    { /* final action */ }

$$1:
    { /* intermediate action */ }
```

My original implementation of unconditional actions ran afoul of this technique when I simply modified yacc to recognize the new action delimiters and omit the conditional code for testing `yy_trial_parse`. I discovered that this approach led to some obvious inefficiency in the parser (the extra productions and nonterminals required additional cases to be generated in the driver) since a combination of actions such as

```
x: y
    [ /* unconditional action */ ]
    { /* conditional action */ }
```

would be expanded by yacc to

```
x: y $$1
    [ /* unconditional action */ ]
```

```

$$1:
  { /* conditional action */ }

```

(Note that embedded actions in yacc can also cause shift/reduce conflicts that do not occur in the absence of such actions.) In order to avoid the resulting proliferation of empty rules in such cases, unconditional actions have been implemented in such a way that when an unconditional action occurs immediately before another (conditional or unconditional) action, the two actions are treated as a single action and no extra production is generated. Thus the code generated for this example would look like that in [Figure 3](#).

In this sense an unconditional action preceding a conditional action may be thought of as an initial unconditional part of the conditional action. In the case that the unconditional action occurs after a conditional action it is handled in the ‘usual’ manner and a null production is generated.

For the sake of completeness it should be noted that there may indeed be circumstances in which some semantic actions need to be performed even during a trial parse. A candidate for such a circumstance is a C++ prototype such as

```
int foo( struct X {int i;} y, X z );
```

Now this is *not* an acceptable prototype in C++ since it violates the restriction that types may not be defined in return or argument types, but the error is not a *syntactic* one. However, if a trial parse is performed in this case (for example, in order to determine if the construct within the parentheses is an argument-declaration-list), and all semantic processing is suppressed during this parse, then when the parser encounters the second X it will not realize that this is a type-name, and so a syntax error will be diagnosed. Precisely how to handle such an example may be a matter of some debate, and several alternatives are available. Not all of these require performing semantic actions during trial parsing, and even if it is decided to perform such actions, the degree of analysis required in order to handle such cases may be considerably less than that required for the full semantic analysis of this declaration. The reader is left to ponder his own preferred response to problems of this sort with the observation that the features of trial parsing, the distinction between conditional and unconditional actions, and judicious testing of the `yy_trial_parse` variable allow for a range of solutions.

```

case n:
  {
    /* unconditional action */
  }
  #if YY_TRIAL_PARSE
    if ( !yy_trial_parse )
  #endif
  {
    /* conditional action */
  }

```

Figure 3. Driver code emitted for unconditional and conditional actions

### Returning from a trial parse

The parser (i.e. `yyparse()`) will not return until it encounters the EOF (end of file) token. Yet in a trial parse we want `yyparse()` to return as soon as it determines either that it has a successful parse or that it has encountered an error. Forcing a return of ‘success’ is simple enough since we can easily take advantage of the `YYACCEPT` macro available in `yacc`. However, forcing a return of ‘failure’ is not so straightforward. The use of `YYABORT`, for example, requires that a reduction has taken place to which the abort action may be attached, and ensuring the presence of such a reduction in just the right place can be troublesome. Instead, to achieve a failure return in an efficient manner, the code

```
if ( yy_trial_parse )
    goto yyabort;
```

is inserted immediately after the label `yyerrlab`. Thus if an error is encountered during trial parsing, `yyparse()` will return a failure code and not attempt error recovery.

### Reduction tracking

In `output.c` (within the function `out_rule_data()`), we emit to the `yacc` output file the definition of an enumeration type `yy_nonterm`, whose constants represent the nonterminals of our grammar. Each constant in the enumeration begins with the prefix `yy_nt_` and ends with the nonterminal name as this appears in the `yacc` specification file. For example, if our grammar contains the nonterminals declaration, statement, and expression, then the `yy_nonterm` enumeration will contain the constants

```
yy_nt_declaration
yy_nt_statement
yy_nt_expression
```

This allows an easy means of referring in C code to a nonterminal in the `yacc` specification file. Our version of `byacc` keeps track of the names of nonterminal symbols and their numeric representations (used in the parsing tables) by means of the arrays `symbol_name[]` and `symbol_value[]`. In defining the `yy_nonterm` enumeration we form the constants by using the `symbol_name[]` array and assign them their corresponding values in the `symbol_value[]` array.

Immediately after this definition of `yy_nonterm` we emit a definition

```
enum yy_nonterm yy_last_reduction;
```

to the output file as well. Similarly, both the definition of the enumeration type and an extern declaration of the `yy_last_reduction` enumeration are emitted to the `defines` file. In our version of `byacc` this is done within the function `out_defines()` in `output.c`. We accomplish reduction tracking by adding the single statement

```
yy_last_reduction = yylhs[yyn];
```

to the `skeleton.c` file immediately after the label `yyreduce`. Thus, each time a reduction

occurs, the value of the nonterminal on the left-hand side of the production used is assigned to `yy_last_reduction`.

This mechanism allows us at any point in our code to check in a very straightforward manner the symbol to which the last reduction was done. Such simple context information can sometimes be put to good use in determining whether a trial parse needs to be done. For example, if the lexical analyzer sees a left parenthesis followed by a type name, it needs to know whether it may be looking at the start of an argument-declaration-list. We can use such code as is shown in [Figure 4](#) to help determine whether we need to trial parse the construct in question as an argument-declaration-list.

### Dynamic allocation of the parser stack

A problem we encountered quite early in the testing of our parser was that some of the tests constructed by our testing group resulted in the dreaded ‘yacc stack overflow’ message. Since yacc’s stack is implemented as a statically sized array of size `YYSTACKSIZE` (whose default value is 300) it is always possible to exceed this limit. Typically such a result is achieved by highly nested expressions.

A number of considerations then compelled the implementation of a dynamically allocated parser stack and the several functions necessary to manage this stack. A field recording the current stack size of the parser is included in the `PARSE_STATE`. Since each call to `yyparse()` requires its own stack, and since trial parsing requires recursive invocation of `yyparse()`, the overhead involved in allocating and deallocating these stacks can become rather severe. However, we have discovered that it is rarely the case that a recursive depth greater than three is necessary to parse common C++ programs, and accordingly we take the approach of pre-allocating three parser stacks (which can of course be expanded upon demand) to be used by the primary invocation of `yyparse()` and the first two recursive calls to this routine. Any additional stacks required are allocated and freed as necessary.

```

/*
 * Looking at input of the form
 *      ( TYPENAME
 */
switch ( yy_last_reduction )
{
  case yy_nt_declarator:
  case yy_nt_abstract_declarator:
  case yy_nt_decl_specifiers:
  case yy_nt_decl_specifier_list:

    /* this could be an argument-declaration-list */

  default:
    /* fall through to other tests */
}

```

*Figure 4. Using `yy_last_reduction`*

## TRIAL PARSING AND THE LEXICAL ANALYZER

The role of the lexical analyzer in trial parsing cannot be overstated. Rather than acting simply as a ‘preprocessor’ that only tokenizes its input stream and hands off tokens to the parser, the lexical analyzer must work intimately with the parser in order to achieve the benefits of trial parsing. In effect, the lexical analyzer functions as a driver for a simple recursive descent parser that calls `yyparse()` to do the actual parsing. Let’s look a bit more closely at some of these responsibilities of the lexical analyzer.

### Determining the need for a trial parse

It is the lexer’s responsibility to determine when a trial parse is called for. The precise manner in which this is done will depend, in both detail and complexity, on the language being parsed, its grammar, and its ambiguities. Typically, of course, some degree of look-ahead will be involved, and it may be necessary also to maintain some sense of context. For example, in C++ certain constructions are ambiguous in some contexts but not in others. You may encounter a declaration/expression-statement ambiguity where a statement might occur, but not elsewhere. This means that immediately following each statement you might expect to encounter such an ambiguity, and you might see one as well in a `for-init-statement`. But otherwise, if you see a type-name followed by an open parenthesis the construction will not be subject to this particular ambiguity. Likewise, if you know you are in class scope, then you know that such a construction (even following a statement) cannot be an expression-statement and hence is not ambiguous.

By achieving a minimal sensitivity to context the occurrence of unnecessary trial parses can be reduced or eliminated. This requires a degree of communication between the parser and the lexer that will be illustrated in the examples of a later section.

### The token buffer

A trial parse may fail, and when it does we need to back up to where we started that trial parse so that we can begin the parse anew. Thus the parser input stream must be buffered, and it is most convenient and efficient to maintain a buffer of tokens (rather than to buffer the original source text for this purpose). Either as a consequence of trial parsing or as a requirement to force the parser in a certain direction, then in addition to the usual buffer operations we need to perform any of the following operations on the buffer:

1. Insert a token in the buffer. (For example, in order to ‘seed’ the buffer for a trial parse we need to add a guide token at or near the front of the buffer.)
2. Replace a token in the buffer with a different token. (If a given trial parse fails, this may indicate to us the path down which the parser must be forced, and so we need to replace the original guide token with a different one.)
3. Delete a token from the buffer. (Trial parses for complex expressions may become recursive. In such cases it may be necessary to delete guide tokens inserted by recursive calls to the parser.)

For these reasons we have implemented the buffer as a doubly linked list.

## The lexer state

The state saved and restored by the lexer must of course contain information pertaining to the current state of the token buffer. In addition to this information concerning the buffer, the value of the variable `yy_last_reduction` is also saved prior to trial parsing and restored afterward. Similarly, the value of `yyval` must be saved and restored as well.

## Guide tokens and guiding productions

*Guide* tokens are ‘artificial’ tokens used to coerce the parser down a path it would not otherwise be inclined to take. When an ambiguity is detected, a guide token is inserted at (or in some cases, near) the beginning of the token buffer and the `yyparse()` is called recursively to perform the trial parse.

The guide tokens (and their meanings) that we make use of in our C++ parser are

<code>START_DECL</code>	start parsing a declaration
<code>START_NON_DECL</code>	start parsing a non-declaration
<code>START_OLD_CAST</code>	start parsing an old-style cast
<code>START_ARG_DECL_LIST</code>	start parsing an argument-declaration-list
<code>START_ALLOC</code>	start parsing an allocation expression

The details of how such guide tokens are used will be illustrated in the example of the final section.

A *guiding production* (or *guiding rule*) is a production whose right-hand side begins with a guide token, and a *primary guiding production* is a guiding production whose left-hand side is the start symbol. Thus, for example, in our C++ grammar the productions

```
call_simple_type_name: START_NON_DECL simple_type_name
start_parser: START_DECL declaration ';'

```

are, respectively, a guiding production and a primary guiding production.

## EXAMPLES

Figure 5 is a fragment of the yacc specification for our C++ parser. I have changed this representation slightly from the one we employ to enhance readability, and the productions are numbered to facilitate reference to them. For example, for portability purposes (since we use yacc on an ASCII machine and then compile its output on an EBCDIC machine) we do not make use of character constants anywhere in our yacc specification. However, I use them here for the sake of clarity. Our actual grammar is in fact quite large, containing more than 700 productions, and of course most of these have been omitted. I have included only those that contribute directly to an understanding of the technique of trial parsing, and so the astute reader will notice that there are some nonterminals for which there are no rules. Actions have been omitted except for those (usually unconditional ones) that pertain to trial parsing. Finally, for a variety of reasons our grammar is dissimilar from the ANSI grammar for C++ at several points. I shall not describe these in detail here since the deviations are not germane to the current discussion. The knowledgeable reader

```

%{
int possible_ambiguity;
%}

%union {
    /* The various types of tokens are declared here */
};

%token <Token> START_ALLOC
%token <Token> START_DECL
%token <Token> START_NON_DECL
%token <Token> START_OLD_CAST
%token <Token> START_ARG_DECL_LIST

%start start_parser

%%

start_parser
[1] : translation_unit

[2] | START_DECL declaration ';'
    [
    /* Indicate to yylex() that a trial parse
    * is complete
    */
    YYACCEPT;
    ]

[3] | START_OLD_CAST '(' type_name ')'
    [ YYACCEPT; ]

[4] | START_ARG_DECL_LIST argument_declaration_list ')'
    [ YYACCEPT; ]

[5] | START_ALLOC '(' decl_specifiers ')'
    [ YYACCEPT; ]

[6] | START_ALLOC '(' decl_specifiers abstract_declarator ')'
    [ YYACCEPT; ]

[7] | error
    { /* Report error */ }
    ;

statement
[8] : simple_statement
    [ possible_ambiguity = 1; ]

[9] | labeled_statement
    [ possible_ambiguity = 1; ]

[10] | compound_statement

```

Figure 5. Fragment of a C++ grammar

```

        [ possible_ambiguity = 1; ]
[11] | selection_statement
      [ possible_ambiguity = 1; ]
[12] | iteration_statement
      [ possible_ambiguity = 1; ]
[13] | jump_statement
      [ possible_ambiguity = 1; ]
[14] | error
      [ possible_ambiguity = 1; ]
      ;

for_start

[15] : 'for' '('
      [ possible_ambiguity = 1; ]
      ;

for_init_statement

[16] : simple_statement
      [ possible_ambiguity = 0; ]
[17] | error ';'
      ;

type_simple_type_name

[18] : simple_type_name
[19] | START_DECL simple_type_name
      ;

decl_specifier

[20] : type_simple_type_name

      /* Additional rules for decl_specifier */
      ;

call_simple_type_name

[21] : simple_type_name
[22] | START_NON_DECL simple_type_name
      ;

postfix_expression:

[23] | call_simple_type_name '(' ')'
[24] | call_simple_type_name '(' expression_list ')'

      /* Additional rules for postfix_expression */
      ;

```

Figure 5. Continued.

```

paren_type_name
[25] : '(' type_name ')'
[26] | START_OLD_CAST '(' type_name ')'
;

argument_declaration_list
[27] : Arg_Decl_Lst
[28] | START_ARG_DECL_LIST Arg_Decl_Lst
;

```

Figure 5. Continued.

should have no difficulty in identifying these and even inferring the motivations for such deviations.

Notice that the symbol `start_parser` is an ‘artificial’ start symbol that is added in order to allow for reductions involving either the ‘genuine’ start symbol `translation_unit` or a number of guiding productions. The idea, of course, is that when `yyparse()` is called for a trial parse, the input stream will have been seeded with a guide token and the parser will proceed down a path indicated by one of the guiding productions. If the trial parse is successful, a reduction will be made to `start_parser` and (through the mechanism described in previous sections) `yyparse()` will return to `yylex()`. Otherwise, in the case that the genuine (non-trial) parse is in effect, these guiding productions will not be involved in the parse.

The possible\_ambiguity flag is set by an unconditional action when a context has been entered where trial parsing may be required. Typically this is at a point where a statement may begin. It is also set (by the lexer) when an open brace is encountered at block or function scope (since either a declaration-statement or expression-statement may occur at that point). This flag is reset by the lexer under several circumstances, such as when a trial parse is begun, or when one of the `START_DECL` or `START_NON_DECL` guide tokens is encountered (since this indicates that disambiguation has already been accounted for). It is ignored if what otherwise might be an ambiguity is encountered at class scope (since an expression-statement may not occur in that context).

The declaration-statement/expression-statement ambiguity referred to above manifests itself in our grammar as a reduce/reduce conflict involving productions 18 and 21. Yacc’s default resolution of such a conflict is to use the rule that occurs first. In this case, the result is that a `simple_type_name` will be reduced (by 18) to a `type_simple_type_name` rather than (by 21) to a `call_simple_type_name`, and consequently the ambiguous constructs in question would always be reduced to declarations rather than expressions. The guide token `START_NON_DECL` may be used to force (by means of production 22) reduction of a `simple_type_name` to a `call_simple_type_name` and subsequent reduction to a `postfix_expression` by way of one of the rules 23 or 24. Similarly, the guide token `START_DECL` can be used to force the parsing of input as a declaration in order to enforce the dictum that ‘if something could be a declaration, it is one’.

In representing the lexical buffer for the examples below *T* will be used to indicate a token of type `type-name`, *ID* to indicate an identifier token, *LIT* to indicate a

numeric literal, and punctuators and operators (commas, parentheses, increment operator, etc.) to represent themselves. As our first example we consider the C++ code in Figure 6.

Here the problematic construct is the `X(a)++` which must be parsed as an expression. When the lexical analyzer encounters the input

```
T (
```

at block scope, it realizes that this is the beginning of a potentially ambiguous construct.\* Since (by convention) anything that *could* be taken as a declaration *will* be taken as a declaration, an attempt is made to parse the impending input as a declaration. The guide token `START_DECL` is inserted prior to the type-name, and `yyparse()` is called with its input looking like

```
START_DECL T (
```

The idea is that if this is the beginning of a declaration, then a reduction by rule 2 will occur. When the type-name is read it is reduced to a `type_simple_type_name` (by rule 18), and all goes well until the increment operator is read. At this point, the parser enters an error state and `yyparse()` returns with a failure code. At this point the lexical buffer looks like

```
START_DECL T ( ID ) ++
```

Since the attempt at parsing the input as a declaration has failed, the lexer decides that what it is looking at is an expression (if it is well-formed at all). The `START_DECL` guide token is replaced with `START_NON_DECL`, the buffer read pointer is reset to point to this token, and the parse is allowed to continue from the state it was in prior to the abortive trial parse.

This time, with the buffer containing

```
START_NON_DECL T (ID) ++
```

rule 22 causes a reduction to a `call_simple_type_name` rather than to a `type_simple_`

```
class X {
public:
    X(int);
    operator++(int);
};

int a;

void foo(void)
{
    X(a)++;
}
```

Figure 6. Declaration/expression ambiguity

---

\* Note that if this sequence of tokens is encountered at file scope or class scope, there is no danger of ambiguity. Knowledge of this type is built into the lexer (which has access to scoping information).

type\_name and ultimately (by means of rule 24)  $X(a)++$  is reduced to a postfix\_expression. Note that if the increment operator were omitted in this example, the initial trial parse would succeed and the line would parse as a declaration.

Our second example involves two similar declarations in C++. In examining the code in Figure 7 we see that the two declarations within the function definition are quite similar, and yet one must be parsed as a function declaration and the other as an object declaration. The distinction between these two declarations hinges on whether a construction of the form  $\text{int}(x)$  is interpreted as a declaration or as a new-style cast (function-style explicit type conversion).

In looking at either of the declarations in Figure 7 the lexer sees the left parenthesis followed immediately by a type-name ( $\text{int}$ ),

```
( T
```

and determines that this is a potentially ambiguous construct. This could be the beginning of an argument\_declaration\_list or it could be the beginning of an expression\_list, and there is no limited amount of lookahead that will allow the parser to determine which of these is the case. A trial parse must be made in order to decide the issue.

In the case of Figure 7(a), the trial parse proceeds in the following manner. First the guide token `START_ARG_DECL_LIST` is inserted in the token buffer immediately following the '(' so that the buffer looks like

```
( START_ARG_DECL_LIST T
```

and (after saving states) `yyparse()` is called to start a parse beginning at this guide token. This will force the parser down the path determined by the guiding production 28 and the primary guiding production 4. As a consequence of this,  $\text{int}(a)$  will be parsed as a declarator (since  $\text{int}$  will be reduced to a `type_simple_type_name`), and

```
int(a),
```

```
// Assume that C has been previously defined
// as a class.

void foo(int i)
{
    // Figure 7a: a function declaration
    C f(int(a), int(b));

    // Figure 7b: an object declaration
    C g(int(c), int(d) + 2);

    // ...
}
```

Figure 7. Examples of ambiguity

as an `argument_declaration_list`. Then `int(b)` will be parsed also as a declarator and, when the final `' )'` is seen,

```
int(a), int(b)
```

will be reduced to an `argument_declaration_list`. At that point a reduction by means of rule 4 will take place to `start_parser`, and the unconditional action `YYACCEPT` will force a successful return.

The token buffer will then look like

```
( START_ARG_DECL_LIST T ( ID ), T ( ID ) )
```

Since the trial parse was successful, we know that the construct we were looking at is indeed an `argument_declaration_list`. Thus the guide token is left in place, the read pointer into the token buffer is backed up to the left parenthesis immediately preceding this guide token, and the original (top-level) invocation of `yyparse()` is allowed to continue on the buffer from that point. The final result is that the declaration in [Figure 7\(a\)](#) is parsed as a function declaration because of the occurrence of the `argument_declaration_list`.

In the second case the parse proceeds in exactly the same manner until the `' +'` is read. At that point `yyparse()` enters an error state and returns a failure code. At this point the buffer looks like

```
( START_ARG_DECL_LIST T ( ID ), T ( ID ) +
```

Since the trial parse of the input as an `argument_declaration_list` has failed, the lexer removes the `START_ARG_DECL_LIST` token, resets the lexical buffer read pointer to the initial left parenthesis and continues on from the state it was in prior to trial parsing. Without the guide token in the buffer, both `int(c)` and `int(d)` will be parsed as `cast_expressions` rather than declarations (since the `int` in each case will reduce to a `call_simple_type_name` by way of production 21 rather than a `type_simple_type_name` by way of production 18). Such a reduction to a `call_simple_type_name` will then bring about (in virtue of production 24) reduction to a `postfix_expression` and hence (through productions not shown) to a `cast_expression`. As a consequence, the input that appears as

```
int(c), int(d) + 2
```

will be parsed as an `expression_list` rather than an `argument_declaration_list`, and [Figure 7\(b\)](#) will be parsed as the declaration of an object (by means of reductions not shown).

These examples illustrate how guide tokens, guiding productions, and trial parsing may be used to resolve ambiguities and to achieve the effect of unlimited lookahead. The discussions of the earlier sections should provide sufficient detail to implement support for these features in any common version of `yacc`, endowing a `yacc`-based parser with substantially greater flexibility and power.

## SUMMARY

What can be said about the utility of this approach to parsing non-LR( $k$ ) grammars? It certainly is not the most elegant way to proceed. A more highly sophisticated parser generating tool is likely to provide for a more easily specifiable and maintainable parser, but of course the cost of acquiring (or writing) and maintaining such a tool must also be considered. The other obvious alternative is to turn to traditional recursive descent techniques, and it is likely that this is the approach we would choose in the future. In our experience a recursive descent compiler offers certain advantages in terms of diagnostics and error detection and recovery.

In terms of development costs, our use of the yacc-based trial parsing approach was relatively small. Modifications to yacc itself required little time and testing. Required modifications to the lexical analyzer consisted of approximately 650 lines of C code, including all the code necessary to support the buffering of tokens. A total of nine guiding productions and 11 unconditional actions were added to the C++ grammar, amounting to an increase of approximately one per cent.

Although required changes to the lexical analyzer were small in terms of lines of code, actually implementing those changes and testing them demanded considerable care. In addition, a lexical analyser that is so intimately tied to actions of the compiler as a whole (by using information concerning context and scoping, for example) is by its nature more difficult to maintain. The correctness of the parser is of course also sensitive to changes in the yacc specifications file. A developer unfamiliar with the close interrelations of the lexer and the parser may make what appears to be an innocuous change to the yacc grammar, and yet this can have serious consequences for the success of the parser through its effect on trial parsing. Shortly after the full implementation of trial parsing we encountered difficulties of this type. None the less, at this point we feel that we have a stable and robust implementation, and the ease with which subsequently discovered bugs have been repaired attests to this view. As an added benefit to this approach, it appears that we will be able to use the lexical analyzer and (a subset of) the yacc specification as the basis for a C++ expression evaluator to be used in our debuggers, thus achieving a substantial amount of code sharing and ensuring that the compiler and debugger handle expressions in the same way. This would not be so straightforward if the underlying parser were recursive descent.

## ACKNOWLEDGEMENTS

A number of people have made contributions to our C++ development effort, and I have benefitted greatly from helpful discussions concerning grammars and parsing techniques with Kevin Bond, Jack Rouse, and Gavin Koch. Special thanks in this regard are due to Gavin Koch who, as principal designer and implementor of the C++ translator, provided me with needed insight into the nature of ambiguity in C++ as well as some truly perverse test cases for my approach to ambiguity resolution. Despite the help of these very skilled people, any flaws in the design or implementation of this approach remain my own.

## REFERENCES

1. S. C. Johnson, 'Yacc: yet another compiler compiler', *Computing Science Technical Report 32*, Bell Laboratories, Murray Hill, NJ, July 1975.

2. T. Mason, D. Brown and J. Levine, *Lex & Yacc*, O'Reilly and Associates, Inc., Sebastopol, California, 1990.
3. A. T. Schreiner and H. G. Friedman, *Introduction to Compiler Construction with UNIX*, Prentice-Hall, Englewood Cliffs, N.J., 1985.
4. A. V. Aho and J. D. Ullman, *The Theory of Parsing, Translation and Compiling, Volume 1*, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
5. M. Ball, 'Inside templates', *C++ Report*, **4**, 36–40 (1992).
6. M. E. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1990.
7. B. Stroustrup, *The C++ Programming Language*, second edition, Addison-Wesley, Reading, Massachusetts, 1991.