Mail corrections, suggestions, or comments to tmoog@polhode.com

Where is

#1. The current maintenance release of PCCTS, these notes, and related examples are available on the net

Primary Site:

URL:	http://www.polhode.com/pccts.html
anonymous ftp:	ftp://ftp.enteract.com/users/tmoog

Europe:

anonymous ftp:	ftp://ftp.th-darmstadt.de/pub/programming/compiler-compiler/pccts/*

(This is updated weekly on Sunday.)

#2. Some other items available at http://www.polhode.com:

Link to the complete and unabridged version of T.J. Parr's book, *Language Translation Using PCCTS and C++*.
Link to the source code for the examples from this book.

Example grammars for C++, ANSI C, and Fortran 77. The Fortran 77 grammar (C mode) by Ferhat Hajdarpasic includes Sorcerer routines.

Log of all changes made as part of the maintenance releases: CHANGES_FROM_133*.TXT.
List of known problems: KNOWN_PROBLEMS.TXT.

#3. Newsgroup is comp.compilers.tools.pccts. Mailing list is pccts_1-33 at onelist.com.

Basics

#4. Invoke ANTLR or DLG with no arguments to get a switch summary

#5. Tokens begin with uppercase characters, rules begin with lowercase characters

#6. Even in C mode you can use C++ style comments in the non-action portion of ANTLR source code

Inside an action you have to obey the comment conventions of your compiler.

#7. In #token regular expressions spaces and tabs which are not escaped are ignored

This makes it easy to add white space to a regular expression:

```
        #token  Symbol  "[a-z A-Z] [a-z A-Z 0-9]*"
```

#8. Never choose names which coincide with compiler reserved words or library names

You'd be surprised how often someone has done something like one of the following:

```
        #token FILE    "file"
        #token EOF     "@"
        const:         "[0-9]*" ;
```

#9. Write <<predicate>>? not <<predicate *semi-colon*>>? (semantic predicates go in "if" conditions)

#10. Some constructs which cause warnings about ambiguities and optional paths

```
  rule : a { ( b | c )* } ;

  rule : a { b } ;
  b    : ( c )* ;

  rule : a c* ;
  a    : b { c } ;

  rule : a { b | c | } ;
```

Checklist

#11. Locate incorrectly spelled #token symbols using ANTLR –w2 switch or by inspecting *parserClassName*.cpp

If a #token symbol is spelled incorrectly ANTLR will assign it a new #token number which, of course, will never be matched.

#12. Be consistent with in-line token definitions: "`&&`" will not be assigned the same token number as "`\&\&`"

#13. Duplicate definition of a #token name is not reported if there are no actions attached

ANTLR will simply use the later definition and forget the earlier one. Using the ANTLR –w2 option does not help

#14. Use ANTLR option -info o to detect orphan rules when ambiguities are reported

#15. `LT`(*i*) and `LATEXT`(*i*) are magical names in semantic predicates — punctuation is critical

ANTLR wants to determine the amount of lookahead required for evaluating a semantic predicate. It does this by searching in C++ mode for strings of the form "`LT(`" and in C mode for strings of the form "`LATEXT(`". If there are spaces before the open "(" it won't make a match. It evaluates the expression following the "(" under the assumption that it is an integer literal (e.g."1"). If it is something like "`LT(1+i)`" then you'll have problems. With ANTLR switch –w2 you will receive a warning if ANTLR doesn't find at least one `LT`(*i*) in a semantic predicate.

#token

#16. To change the token name appearing in syntax error messages: #token ID("identifier") "[a-z A-Z]+"

The string appearing inside the parenthesis will be used for the token name in zztokens and _token_tbl

#17. To match any single character use: "`~[]`", to match everything to a newline use: "`~[\n]*`"

#18. To match an "`@`" in your input text use "`\@`", otherwise it will be interpreted as the end-of-file symbol

#19. The escaped literals in #token regular expressions are: `\t \n \r \b` (not the same as ANSI C)

#20. In #token expressions "\12" is decimal, "\012" is octal, and "\0x12" is hex (not the same as ANSI C)

Contributed by John D. Mitchell (johnm@jGuru.net).

#21. DLG wants to find the longest possible string that matches

The regular expression "`~[]*`" will cause problems — it will gobble up everything to the end-of-file.

#22. When two regular expressions of equal length match a regular expression the first one is chosen

Thus more specific regular expressions should appear in the grammar file before more general ones:

```
#token  HELP    "help"         /*  should appear before "symbol" */
#token  Symbol  "[a-z A-Z]*"   /*  should appear after keywords  */
```

Some of these may be caught by using the DLG switch –Wambiguity. In the following grammar the input string "HELP" will never be matched:

```
#token  WhiteSpace      "[\ \t]"        <<skip();>>
#token  ID              "[a-z A-Z]+"
#token  HELP            "HELP"

statement
      : HELP "@"      <<printf("token HELP\n");>>    /* a1 */
      | "inline" "@"  <<printf("token inline\n");>>  /* a2 */
      | ID "@"        <<printf("token ID\n");>>      /* a3 */
      ;
```

The best advice may be to follow the practice of TJP: place "#token ID" at the end of the grammar file.

#23. Inline regular expression are no different than #token statements

PCCTS code does *not* check for a match to "inline" (Item #22 line a2) before attempting a match to the regular expressions defined by #token statements. The first two alternatives ("a1" and "a2") will *never* be matched. All of this will be clear from examination of the file "parser.dlg" (the name does *not* depend on the parser's class name).

Another way of looking at this is to recognize that the conversion of character strings to tokens takes place in class DLGLexer, not class ANTLRParser, and that all that is happening with an inline regular expression is that ANTLR is allowing you to define a token's regular expression in a more convenient fashion — not changing the fundamental behavior.

If one builds the example above using the DLG switch –Wambiguity one gets the message:

```
dlg warning: ambigious regular expression  3  4
dlg warning: ambigious regular expression  3  5
```

The numbers which appear in the DLG message refer to the assigned token numbers. Examine the array _token_tbl in *parserClassName*.cpp to find the regular expression which corresponds to the token number reported by DLG:

```
ANTLRChar *Parser::_token_tbl[]={
      /* 00 */          "Invalid",
      /* 01 */          "@",
      /* 02 */          "WhiteSpace",
      /* 03 */          "ID",
      /* 04 */          "HELP",
      /* 05 */          "inline"
};
```

Well, there is one important difference for those using Sorcerer. With in-line regular expressions there is no symbolic name for the token, hence it can't be referenced in a Sorcerer rule. Contributed by John D. Mitchell (johnm@jGuru.com).

#24. Watch out when you see `~[`*list-of-characters*`]` at the end of a regular expression

What the user usually wants to express is that the regular expression should stop *before* the *list-of-characters*. However the expression will include the complement of that list as part of the regular expression. Often users forget about what happens to the characters which are in the complement of the set.

Consider for example a #lexclass for a C style comment:

```
/* C-style comment handling */
#lexclass COMMENT                                               /* a1 */
#token "\*/"            << mode(START); skip(); >>              /* a2 */
#token "~[\*]+"         << skip(); >>                           /* a3 */
#token "\*~[/]"         << skip(); >>  /* WRONG */              /* a4 */
             /* Should be "\*"                       */         /* a5 */
             /* Correction due to Tim Corringham     */         /* a6 */
             /*    tim@ramjam.u-net.com    20-Dec-94  */        /* a7 */
```

The RE at line a2 accepts "*/" and changes to #lexclass START. The RE at line a4 accepts a "*" which is *not* followed by a "/". The problem arises with comments of the form:

```
      /* this comments breaks the example **/
```

The RE at line a4 consumes the "**" at the end of the comment leaving nothing to be matched by "*/".

This is a relatively efficient way to span a comment. However it is not the simplest. A simpler description is:

```
#token "\*/"            << mode(START); skip(); >>              /* b1 */
#token "~[]"            << skip(); >>                           /* b2 */
```

This works because b1 ("*/") is two characters long while b2 is only one character long — and DLG always prefers the longest expression which matches.

For those who are concerned with the efficiency of scanning:

```
      #token      "[\n\r]"          <<skip();newline();>>
      #token      "\*/"             <<mode(START);skip();>>
      #token      "\*"              <<skip();>>
      #token      "~[\*\n\r]+"      <<skip();>>
```

Contributed by Brad Schick

#25. Watch out when one regular expression is the prefix of another

If the shorter regular expression is followed by something which can be the first character of the suffix of the longer regular expression, DLG will happily assume that it is looking at the longer regular expression. See Item #44 for one approach to this problem.

#26. DLG is not able to backtrack (unlike flex)

Consider the following example:

```
      #token            "[\ \t]*"       <<skip();>>
      #token ELSE       "else"
      #token ELSEIF     "else [\ \t]* if"
      #token STOP       "stop"
```

with input:

```
else stop
```

When DLG gets to the end of "else" it realizes that the space will allow it to match a longer string than "else" by itself. So DLG accept the spaces. Everything is fine until DLG gets to the initial "s" in "stop". It then realizes it has no match — but it can't backtrack. It passes back an error status to ANTLR which (normally) prints out something like:

```
invalid token near line 1 (text was 'else ') ...
```

There is an "extra" space between the "else" and the closing single quote mark.

This problem is not detected by the DLG option –Wambiguity.

The section, "Lexical Lookahead" has some additional information.

#27. The lexical routines mode(), skip(), and more() are *not* complicated !

All they do is set status bits in a structure owned by the lexical analyzer and then return immediately. Thus it is OK to call these routines anywhere from within a lexical action. You can even call them from within a subroutine called from a lexical action routine.

It is meaningless to call both more() and skip() in the same action.

#28. lextext() includes strings accumulated via more() — begexpr()/endexpr() refer only to the last matched RE

#29. Use `"if (_lextext != _begexpr) {...}"` to test for RE being appended to lextext using more()

To track the line number of the *start* of a lexical element that may span several lines I use the following test:

```
if (_lextext == _begexpr) {startingLine=_line;}  // user-defined var
```

#30. #token actions can access protected variables of the DLG base class

#31. When lookahead will break semantic routines in #token actions, consider using semantic predicates

In early versions on PCCTS it was common to change the token code based on semantic routines in the #token actions.

Old style:

```
#token TypedefName
#token ID  "[a-z A-Z]*"
      <<if (isTypedefName(lextext)) return TypedefName;>>
```

New Style C mode:

```
#token ID  "[a-z A-Z]*"
typedefName : <<isTypedefName(LA(1)->getText())>>? ID;
```

The old technique is appropriate for making *lexical* decisions based on the input: for instance, treating a number appearing in columns 1 through 5 as a statement label rather than a number. The new style is important because of the buffer between the lexer and parser introduced by large amounts of lookahead, especially syntactic predicates. For instance a declaration of a type may not have been entered into the symbol table by the parser by the time the lexer encounters a declaration of a variable of that type. An extreme case is infinite lookahead in C mode: parsing doesn't even begin until the entire input has been processed by the lexer. See Item #138 for an extended discussion of semantic predicates. Example #10 shows how some semantic decisions can be moved from the lexer to the token buffer.

#32. For 8 bit characters use flex or in DLG make `char` variables unsigned (g++ option –funsigned-char)

#33. The maximum size of a DLG token is set by an optional argument of the ctor DLGLexer() — default is 2000

The maximum size of a character string stored in an ANTLRToken is independent of the maximum size of a DLG token. See Item #60.

#34. If a token is recognized using more() and its #lexclass ignores end-of-file then the very last token will be lost

When a token is recognized in several pieces using more() it may happen that an end-of-file is detected before the entire token is recognized. Without treatment of this special case the portions of the token already recognized will be ignored and the error of a lexically incomplete token will be ignored. Since all appearances of the regular expression "@", regardless of #lexclass, are mapped to the same #token value, proper handling requires some workarounds.

Suppose one wants to recognize C style comments using:

```
#lexclass START
#token  Comment_Begin  "/\*"  <<skip();mode(LC_Comment);more();>>
#token  Eof            "@"
...
#lexclass LC_Comment
#token  Unexpected_Eof  "@"   <<mode(START);>>
#token  Comment_End     "\*/" <<skip();mode(START);>>
#token                  "~[]" <<skip();>>
...
```

The token code "Unexpected_Eof" will never be seen by the parser. The result is that C style comments which omit the trailing "*/" can swallow all the input to the end-of-file and not give any error message. My solution to this problem is to fool PCCTS by using the following definition:

```
#token  Unexpected_Eof  "@@"   <<mode(START);>>
```

This exploits a characteristic of DLG character streams: once they reach end-of-file they must return end-of-file to every request for another character until explicitly reset.

Another example of this pitfall, is the recognition of unterminated C style strings at the end of a file.

#35. Sometimes the easiest DLG solution is to accept one character at a time.

One example is the processing of Fortran style Hollerith constants. See Example #12.

Another example is recognizing radix expressions such as 2#1011 or 16#ffff. Given that the radix can vary between 2 and 36 the easiest way to handle it is to save the radix and then change to another #lexclass where the digits can be inspected one by one. Another alternative is to accept the entire string and then check all the characters at one time.

#tokclass

#36. #tokclass provides an efficient way to combine reserved words into reserved word sets

```
#token Read     "read"
#token Write    "write"
#token Exec     "exec"
#token ID       "[a-z A-Z] [a-z A-Z 0-9 \@]*"
#tokclass Any     {ID Read Write Exec}
#tokclass Verb    {Read Write Exec}
command: Verb Any ;
```

#37. Use ANTLRParser::set_el() to test whether an ANTLRTokenType is in a #tokclass or #FirstSetSymbol

To test whether a token "t" is in the #tokclass "Verb":

```
        if (set_el(t->getType(),Verb_set)) {...}
```

There are several variations of this routine in the ANTLRParser class.

#tokdef

#38. A #tokdef must appear near the start of the grammar file (only #first and #header may precede it)

#lexclass

#39. Inline regular expressions are put in the most recently defined lexical class

If the most recently defined lexical class is not START you may be surprised:

```
        #lexclass START
        ...
        #lexclass LC_Comment
        ...
        inline_example: symbol "=" expression ;
```

This will place "=" in the #lexclass LC_Comment (where it will never be matched) rather than the START #lexclass where the user meant it to be. Since it is okay to specify a #lexclass in several pieces it might be a good idea when using #lexclass to place "#lexclass START" just before the first rule — then any inline definitions of tokens will be placed in the START #lexclass automatically:

```
#lexclass START
...
#lexclass COMMENT
...
#lexclass START
```

#40. Use a stack of #lexclass modes in order to emulate lexical subroutines

Consider a grammar in which lexical elements have internal structure. An example of this is C strings and character literals which may contain elements like:

escaped characters	`\" and \'`
symbolic codes	`\t`
numbers	`\xff  \200  \0`

Rather than implementing a separate #lexclass to handle these sequences for both character literals and string literals it would be possible to have a single #lexclass which would handle both. To implement such a scheme one needs something like a subroutine stack to remember the previous #lexclass. See Example #9 for a set of such routines.

#41. Sometimes a stack of #lexclass modes isn't enough

Consider a log file consisting of clauses, each of which has its own #lexclass and in which a given word is reserved in some clauses and not others:

```
#1;1-JAN-94 01:23:34;enable;forge bellows alarm;move to station B;
#2;1-JAN-94 08:01:56;operator;john bellows;shift change at 08:00;
#3;1-JAN-94 09:10:11;move;old pos=5.0 new pos=6.0;operator request;
#4;1-JAN-94 10:11:12;alarm;bellows;2-JAN-94 00:00:01;
```

If the item is terminated by a separator then there is a problem because the separator will be consumed in the recognition of the most nested item — with nothing left over to be consumed by other elements which end at the separator. The problem appears when it is necessary to leave a #lexclass and return more than one level. To be more specific, a #token action can only be executed when one or more characters are consumed — so to return through three levels of #lexclass calls would appear to require the consumption of at least three characters. In the case of balanced constructs like `"..."` and `'...'` this is not a problem since the terminating character can be used to trigger the #token action. However, if the scan is terminated by a *separator* such as the semi-colon above (";"), one cannot use the same technique. Once the semi-colon is consumed it is unavailable for the other #lexclass routines on the stack to see.

One solution is to allow the user to specify (during the call to pushMode) a "lookahead" routine to be called when the corresponding element of the mode stack is popped. At that point the "lookahead" routine can examine `ch` to determine whether it also wants to pop the stack, and so on up the mode stack. The consumption of a single character can result in popping multiple modes from the mode stack based on a single character of lookahead.

For anything more complicated than this and you might as well write a second parser just to handle the so-called lexical elements.

Continuing with the example of the log file (above): each statement type has its fields in a specific order. When the statement type is recognized, a pointer is set to a list of the #lexclasses which is in the same order as the remaining fields of that kind of statement. An action is attached to every #token which recognizes a semi-colon (";") advances a pointer in the list of #lexclasses and then changes the #lexclass by calling mode() to set the #lexclass for the next field of the statement.

Lexical Lookahead

#42. Vern Paxson's flex has more powerful features for lookahead than dlg

Flex is a superset of lex. For an example of how to use flex with ANTLR in C++ mode see Example #14. For C mode download http://www.polhode.com/NOTES.flex.

#43. Extra lookahead is available from class BufFileInput (subclass of DLGInputStream)

Alexey Demakov has supplied this class to provide more than one character of lookahead for the input stream. The class is located in pccts/h/BufFileInput.*.

#44. One extra character of lookahead is available to the #token action routine in `ch` (except in interactive mode)

In interactive mode (DLG switch –i is not supported in C++ mode) DLG fetches a character only when it needs it to determine if the end of a token has been reached. In non-interactive mode the content of `ch` is always valid. The debug code described in Item #149 can help debug problems with interactive lookahead.

For the remainder of this discussion assume that DLG is in non-interactive mode.

Consider the problem of distinguishing floating point numbers from range expressions such as those used in Pascal:

range: 1..23   float: 1.23

As a first effort one might try:

```
        #token  Int      "[0-9]+"
        #token  Range    ".."
        #token  Float    "[0-9]+.[0-9]*"
```

The problem is that "1..23" looks like the floating point number "1." with an illegal "." at the end. DLG always takes the longest matching string, so "1." will always look more appetizing than "1". What one needs to do is to look at the character following "1." to see if it is another ".", and if it is to assume that it is a range expression. The flex lexer has trailing context, but DLG doesn't — except for the single character in `ch`.

A solution in DLG is to write the #token Float action routine to look at what's been accepted, and at `ch`, in order to decide what to do:

```
  #token Float    "[0-9]*.[0-9]*"
          <<if (*endexpr() == '.' && /* might use more complex test       */
                ch == '.') {
                  mode(LC_Range);   /* treat it like a range expression   */
                  return Int;       /* looks like an int followed by ".." */
            };
          >>

  #lexclass LC_Range
  #token Range    "."      <<mode(START);>>  // consume second "." of range
```

#45. There is no easy way in DLG to distinguish integer "1" from floating point "1." when "1.and.2" is valid

This differs from Item #44 in that two characters of lookahead are required before a decision can be made on whether the "." is part of ".and." or it is part of a floating point number. This is a frequent problem which can only be handled by using a more powerful lexer such as flex.

#46. For lex operators "^" and "$" (anchor pattern to start/end of line) use flex - don't bother with dlg

Line and Column Information

Most names in this section refer to members of class DLGLexerBase or DLGLexer

Before C++ mode the proper handling of line and column information was a large part of these notes.

#47. If you want column information for error messages (or other reasons) use C++ mode

#48. If you want accurate line information even with many characters of lookahead use C++ mode

#49. Call trackColumns() to request that DLG maintain column information

#50. To report column information in syntax error messages override ANTLRParser::syn() — See Example #5

#51. Call newline() and then set_endcol(0) in the #token action when a newline is encountered

#52. Adjusting column position for tab characters

Assume that tabs are set every eight characters starting with column 9.

Computing the column position will be simple if you match tab characters in isolation:

```
        #token Tab  "\t"    <<_endcol=((_endcol-1) & ~7) + 8;>>
```

This would be off by 1, except that DLG, on return from the #token action, computes the next column using:

```
          _begcol=_endcol+1;
```

If you include multiple tabs and other forms of whitespace in a single regular expression, the computation of _endcol by DLG must be backed out by subtracting the length of the string. Then you can compute the column position by inspecting the string character by character.

#53. Computing column numbers when using more() with strings that include tab characters and newlines

```
/* what is the column and line position when the comment includes
   or is followed by tabs tab  tab */  tab  tab  i++;
```

> Note: This code excerpt requires a change to PCCTS 1.33 file pccts/dlg/output.c in order to inject code into the DLGLexer class header. The modified source code is distributed as part of the notes in file notes/changes/dlg/output.c and output_diff.c. An example of its use is given in Example #7.

My feeling is that the line and column information should be updated at the same time more() is called because it will lead to more accurate position information in messages. At the same time one may want to identify the *first* line on which a construct begins rather than the line on which the problem is detected: it's more useful to know that an unterminated string started at line 123 than that is was still unterminated at the end-of-file.

```
void DLGLexer::tabAdjust () {       // requires change to output.c
   char * p;                        //   to add user code to DLGLexer
   if (_lextext == _begexpr) startingLineForToken=_line;
   _endcol=_endcol-(_endexpr-_begexpr)+1;  // back out DLG computation
   for (p=_begexpr;*p != 0; p++) {
      if (*p == '\n') {                    // newline() by itself
         newline();_endcol=0;              //   doesn't reset column
      } else if (*p == '\t') {
         _endcol=((_endcol-1) & ~7) + 8;   // traditional tab stops
      };
     _endcol++;
   };
   _endcol--;                  // DLG will compute begcol=endcol+1
}
```

See Example #7 for a more complete description.

Ambiguity Aid (options -aa, -aam, -aad

#54. Example with nested if statement

Consider the timeless and eternal beauty of the nested if statement:

```
        stmt        : if_stmt                          /* 1  */
                    | assign_stmt                      /* 2  */
                    ;                                  /* 3  */
        if_stmt     : IF expr                          /* 4  */
                      THEN stmt                        /* 5  */
                      { ELSE stmt }                    /* 6  */
                    ;                                  /* 7  */
        assign_stmt : expr EQUAL expr SC ;             /* 8  */
        expr        : E ;                              /* 9  */
```

This will be ambiguous regardless of the value of k and ck chosen. When analyzed with -k 1 and -ck 1 ANTLR will report:

```
ifstmt.g(6) : warning:  alts 1 and 2 of {...} ambiguous upon { ELSE }
```

We can specify the ambiguity of interest using a line number or rule name:

```
        antlr ifstmt.g -aa if_stmt #  invoked using a rule name
        antlr ifstmt.g -aa 6       #  invoked using a line number
```

The output is:

```
Ambiguity Aid      (-ck 1  -aa if_stmt    -aad 1)
```

This identifies the command line options relevant to the ambiguity analysis.

```
  Choice 1: if_stmt/3       line 6  file ifstmt.g
  Choice 2: if_stmt/3       line 6  file ifstmt.g
```

Choice 1 and Choice 2 are the two alternatives that have ambiguous prediction expressions. In this case, both choices are on line 6 of the file and the third line of rule if_stmt (i.e. if_stmt/3).

```
  Intersection of lookahead[1] sets:

     ELSE
```

The intersection of the lookahead sets is simply a restatement of the ambiguity set, but in an easier to read format. The number of sets should be equal to the -ck value. In this example, the ck value is 1 and there is only one such set. If the intersection were empty then there would be no ambiguity and no need for an ambiguity report.

```
   Choice:1  Depth:1  Group:1  (ELSE)
```

Choice 1: This is a traceback of one way one might run across the token ELSE.

```
 1 #token ELSE     if_stmt/3   line 6 ifstmt.g
```

Starting from line 6 one can easily find an ELSE by flowing into the optional element of the if_stmt ("{ ELSE stmt }"). The expression "if_stmt/3" says that this is on the third line of the rule "if_stmt".

```
   Choice:2  Depth:1  Group:2  (ELSE)
```

Choice 2: This is a traceback of another way one might run across the token ELSE. The ambiguity exists because there are two such paths. Sometimes I find it easier to work from the bottom to the top, other times from the top to the bottom. In this case I'll work from item 1 to item 4.

```
 1 end if_stmt     if_stmt/4   line 7 ifstmt.g
```

Item 1: Instead of flowing forward into the optional ELSE block (as with choice 1) we pass over it and flow off the end of the if_stmt (hence "end if_stmt") at line 7. Thus we have an if_stmt with no ELSE clause.

```
                IF e1 THEN s1
 2 end stmt        stmt/3      line 3 ifstmt.g
```

Item 2: The if_stmt (item 1) was referenced from stmt at line 3. The use of line 3 is confusing. The line number is 3 because we have flowed out of the if_stmt at line 1 and out of the implicit and invisible block which encloses the top level alternatives of a rule. It is as though the rule has been written:

```
stmt:   ( if_stmt
        | assign_stmt
        )
        ;
```

```
 3 in {...} block  if_stmt/3   line 6 ifstmt.g
```

Item 3: After flowing out of the stmt mentioned in item 2 we find ourselves entering the {...} block in if_stmt at line 6. The only way to flow into the {...} block on line 6 is by flowing out of the THEN clause on line 5. Recall that we have just flowed out of an if_stmt that had no ELSE clause (item 1). Thus we now have something like this:

```
IF e2 THEN IF e1 THEN s1 ...
```

where the underline identifies an inner if_stmt from item 1.

```
 4 #token ELSE     if_stmt/3   line 6 ifstmt.g
```

Item 4: In item 3 we flowed into the {...} clause on line 6. We have now found the ELSE token at the start of the {...} clause. Thus we now have something that like this:

```
                IF e2 THEN IF e1 THEN s1 ELSE ...
```

where the underline identifies an inner if_stmt from item 1.

This is not terribly clear, so let’s review the data and try to understand what the ambiguity aid is trying to say. When we are ready to recognize the ELSE there must be two plausible derivations for the ELSE token. The first one is the obvious one: the ELSE is part of the if_stmt being parsed. The second choice, the difficult one, arises when there is *no* ELSE clause for an if_stmt. Looking at

```
          IF e2 THEN IF e1 THEN s1 ELSE ...
```

with this hint and some thought we should be able to recognize that the ELSE can be interpreted as part of the underlined statement (choice 1) or as part of the non-underlined statement (choice 2).

I admit that this is far from ideal. However, I have found this an immense aid in trying to identify the source of an ambiguity in large grammars which may require dozens of rules to be examined in order to discover that rule g can follow rule f when rule f appears at the end of e, when e appears at the end of d, etc.

#55. Example with cast expression

This example illustrates the use of the -aad option which controls the number of lookahead tokens to match. The ambiguity aid first reports on the choices which match just one token, then on the choices which match the second token, and so on until the number of tokens specified by the -aad option is reached.

This ambiguity in this example is simple to diagnose without any aid, but I think it has educational value.

```
expr1 : expr2 { EQ_EQ expr2 } ;       /* 1  */
expr2 : expr3 ( ADD_OP expr3 )* ;     /* 2  */
expr3 : expr4 ( MUL_OP expr4 )* ;     /* 3  */
expr4 : expr5                         /* 4  */
      | LP Id RP expr5                /* 5  */
      ;                               /* 6  */
expr5 : Id                            /* 7  */
      | Number                        /* 8  */
      | LP expr1 RP                   /* 9  */
      ;                               /* 10 */
```

When run with -ck 2 ANTLR reports:

```
paren.g(4) : warning: alts 1 and 2 of the rule itself
                              ambiguous upon { LP }, { Id }
```

To diagnose this problem we use the command:

```
antlr paren.g -ck 2 -aa 4 -aad 2 #  ambiguity on line 4, match two tokens
```

The output is:

```
Ambiguity Aid          (-ck 2  -aa 4    -aad 2)

  Choice 1: expr4/1   line 4  file paren.g
  Choice 2: expr4/2   line 5  file paren.g
```

The ambiguity is a choice between line 4 (expr5) and line 5 (LP Id RP ...)

```
  Intersection of lookahead[1] sets:

     LP

  Intersection of lookahead[2] sets:

     Id

   Choice:1  Depth:1  Group:1  (LP)
```

The first choice at depth 1 starts at line 4 (item 1) and flows into expr5 (items 1 and 2). Once inside expr5 (item 2) we find the token LP (item 3).

```
 1 to expr5           expr4/1   line 4     paren.g
 2 expr5              expr5/1   line 7     paren.g
 3 #token LP          expr5/3   line 9     paren.g
```

```
  Choice:2  Depth:1  Group:2  (LP)
```

The second choice at depth 1 starts at line 5 and immediately finds the LP (item 1).

```
 1 #token LP          expr4/2   line 5     paren.g
```

```
  Choice:1  Depth:2  Group:3  (LP Id)
```

We are back with choice 1, but the depth is now 2, so it tries to match the Id in lookahead set 2. We start at line 4 (item 1) and flow into expr5 (items 1 and 2). In expr5 at line 7 we match the LP (item 3). Still within expr5 we flow into expr1 (item 4). From expr 1 we flow into expr 2 (items 6 and 7). This continues until we reach item 14 which contains an ID from expr5.

```
 1 to expr5           expr4/1   line 4     paren.g
 2 expr5              expr5/1   line 7     paren.g
 3 #token LP          expr5/3   line 9     paren.g
 4 to expr1           expr5/3   line 9     paren.g
 5 expr1              expr1/1   line 1     paren.g
 6 to expr2           expr1/1   line 1     paren.g
 7 expr2              expr2/1   line 2     paren.g
 8 to expr3           expr2/1   line 2     paren.g
 9 expr3              expr3/1   line 3     paren.g
10 to expr4           expr3/1   line 3     paren.g
11 expr4              expr4/1   line 4     paren.g
12 to expr5           expr4/1   line 4     paren.g
13 expr5              expr5/1   line 7     paren.g
14 #token Id          expr5/1   line 7     paren.g
```

```
  Choice:2  Depth:2  Group:4  (LP Id)
```

The second choice at depth 2. This is a trivial match to LP followed by Id.

```
 1 #token LP          expr4/2   line 5     paren.g
 2 #token Id          expr4/2   line 5     paren.g
```

This demonstrates how the ambiguity aid reports a chain of rule references.

#56. Example with ambiguity due to limitations of linear approximation

The example illustrates an ambiguity which is due solely to the limitations of linear lookahead:

```
rab : a    /* 1 */
    | b    /* 2 */
    ;      /* 3 */
a   : J X  /* 4 */
    ;      /* 5 */
b   : J P  /* 6 */
    | K X  /* 7 */
    ;      /* 8 */
```

When analyzed with -k 2 there is no ambiguity, but with -k 1 and -ck 2 ANTLR reports:

```
look.g(1) : warning: alts 1 and 2 of the rule itself
                               ambiguous upon { J }, { X }
```

To someone unfamiliar with linear lookahead, the problem is not obvious. Given the lookahead J followed by X surely the choice is obvious ? How does ambiguity aid help ?

We run the ANTLR ambiguity aid with depth 2 because the ambiguity involves two tokens of lookahead:

```
antlr look.g -aa rab -aad 2
```

The output is:

```
Ambiguity Aid       (-ck 2  -aa 1    -aad 2)

  Choice 1: rab/1    line 1  file look.g
  Choice 2: rab/2    line 2  file look.g

  Intersection of lookahead[1] sets:

     J

  Intersection of lookahead[2] sets:

     X

   Choice:1  Depth:1  Group:1  (J)
```

This is at depth 1 (match J, the first token,). The first choice starts at line 1 and flows into rule a at line 4 (items 1 and 2). It matches J at line 4.

```
 1 to a       rab/1  line 1     look.g
 2 a          a/1    line 4     look.g
 3 #token J   a/1    line 4     look.g

   Choice:2  Depth:1  Group:2  (J)
```

This is at depth 1 (match J, the first token). The second choice starts at line 2 and flows into rule b at line 6 (items 1 and 2). It matches J at line 6.

```
 1 to b       rab/2  line 2     look.g
 2 b          b/1    line 6     look.g
 3 #token J   b/1    line 6     look.g

   Choice:1  Depth:2  Group:3  (J X)
```

Choice 1 at depth2 finds a match to the second token of lookahead (X) at line 4 of rule a. Note that it follows the same path of references as choice 1 at depth 1.

```
 1 to a       rab/1  line 1     look.g
 2 a          a/1    line 4     look.g
 3 #token J   a/1    line 4     look.g
 4 #token X   a/1    line 4     look.g

   Choice:2  Depth:2  Group:4  (K X)
```

Choice 2 at depth 2 finds a match to the second token of lookahead (X) at line 7 of rule b. Note that the first lookahead token is not J, but K. Nonetheless, this explains why there is a collision for X which prevents linear lookahead from resolving the ambiguity.

```
 1 to b       rab/2  line 2     look.g
 2 b          b/1    line 6     look.g
 3 #token K   b/2    line 7     look.g
 4 #token X   b/2    line 7     look.g
```

The search at depth 1 shows how both alternatives find the token J in the first token lookahead set. The search at depth 2 shows how both alternatives find the token X in the second token lookahead set.

When ambiguity aid is enabled and an ambiguity is found in a rule or line number that matches the command line argument the ambiguity aid routine traverses the rules using the same routines which compute first sets. It searches for tokens which appear in the ambiguity set with appropriate depth. When it finds a match it reports the chain of rules that were traversed to reach the point at which the match occurs.

#57. Summary of command line switches related to ambiguity aid

The ambiguity aid is controlled by the following command line options:

-aa *ruleName* Selects reporting by name of rule

-aa *lineNumber* Selects reporting by line number (the file name is not used).

-aad *depth* Selects the depth of the search. The default value is 1.

The number of paths to be searched, and the size of the report can grow geometrically with the -ck value if a full search for all contributions to the source of the ambiguity is explored. The depth represents the number of tokens of lookahead which are matched against the sets of ambiguous tokens. A depth of 1 means that the search stops when a lookahead sequence of just one token is matched.

A k=1 ck=6 grammar might generate 5,000 items in a report if a full depth 6 search is made with the ambiguity aid. The source of the problem may be in the first token and obscured by the volume of data - I hesitate to call it information.

When the user selects a depth > 1, the search is first performed at depth=1 for both alternatives, then depth=2 for both alternatives, etc.

-aam Enables "multiple" reporting for a token in the intersection set of the alternatives. The default is "off".

A given token may appear dozens of times in various paths as the program explores the rules which are reachable from the point of an ambiguity. With option -aam every possible path the search program encounters is reported.

Without -aam only the first encounter is reported. This may result in incomplete information, but the information may be sufficient and much shorter.

C++ Mode

#58. The destructors of base classes should be virtual in almost all cases

If you don’t know why you should read Scott Meyers’ excellent book, "Effective C++, Fifty Specific Ways ...".

#59. Why must the AST root be declared as ASTBase rather than AST ?

The functions which implement the rules of the grammar are declared with the prototype:

```
void aRule(ASTBase ** _root) {...};
```

The underlying support code of ANTLR depends only on the behaviors of ASTBase. There are two virtues to this design:

No recompilation of the underlying routines is necessary when the definition of AST changes

The same object code can be used with multiple parsers in the same program each with its own kind of AST

This is in contrast to C++ templates which are designed to provide source code reuse, not object code reuse.

An "AST *" can be passed to an "ASTBase *" why not an "AST **" for an "ASTBase **" ?

This is a C++ FAQ. Consider the following (invalid) code fragment:

```
struct B {};                                                  /* a1 */
struct D1 : B {int i;};                                       /* a2 */
struct D2 : B {double d;};                                    /* a3 */
void func(B ** ppB) {*ppB=new D2;};      /* WRONG */          /* a4 */
D1 * pD1=new D1;                                              /* a5 */
func(&pD1);                                                   /* a6 */
```

At line a5, pD1 is declared to be a pointer to a D1. This pointer is passed to "func" at line a6. The function body at line a4 replaces a pointer to a D1 with a pointer to a D2, which violates the declaration at line a5.

The following *is* legal, although it may not do what is expected:

```
void func2(B * pB) {D1 d1; *pB=d1;};                      /* b1 */
func2(pD1);                                               /* b2 */
```

The assignment at line b1 *slices* d1 and assigns only the B part of d1 to the object pointed to by pB because the assignment operator chosen is that of class B, not class D1.

#60. C++ mode makes multiple parsers easy

pccts/testcpp/5/test.g	Uses multiple instances of a single parse class (thus a single grammar)
pccts/testcpp/6/main.cpp	Program uses parsers for two different grammars (test.g and test2.g)

If two parsers share the same DLG automaton it may be necessary to save DLG state. See Item #61.

#61. Use DLGLexerBase routines to save/restore DLG state when multiple parsers share a token buffer

When the second parser "takes control" the DLGLexer doesn't know about it and doesn't reset the state variables such as #lexclass, line number, column tracking, etc.

Use DLGLexerBase::saveState (DLGState *) and restoreState(DLGState *) to save and restore DLG state.

#62. In C++ mode ASTs and ANTLRTokens do not use stack discipline as they do in C mode

In C mode ASTs and attributes are allocated on a stack. This is an efficient way to allocates space for structs and is not a serious limitation because in C it is customary for a structure to be of fixed size. In C++ mode it would be a serious limitation to assume that all objects of a given type were of the same size because derived classes may have additional fields. For instance one may have a "basic" AST with derived classes for unary operators, binary operators, variables, and so on. As a result the C++ mode implementation of symbolic tags for elements of the rule uses simple pointer variables. The pointers are initialized to 0 at the start of the rule and remain well defined for the entire rule. The things they point to will normally remain well defined, even objects defined in sub-rules:

```
  rule ! : a:rule2 {b:B} <<#0=#(#a,#[$b]);>> ; // OK only in C++ mode
```

This fragment is not be well defined in C mode because "B" would become undefined on exit from "`{...}`".

#63. Summary of Token class inheritance in file AToken.h

```
ANTLRAbstractToken — (empty class) virtual table
       |
       V
       +-- ANTLRRefCountToken — (reference counter) virtual table
       |    |
       |    V
       |    +-- ANTLRCommonRefCountToken — (token type, text, line) virtual table
       |              using variable length text fields
       |
       +-- ANTLRCommonNoRefCountToken - (token type, text, line) virtual table
       |         using variable length text fields
       |
       +-- MyToken — (token type, text, line, ...) virtual table
              Examples:
                 NoLeakToken.h
                 SimpleToken.h
                 notes/calcAST/numToken.h - numeric field
                 notes/col/myToken.h — variable length text with column info
```

#64. Diagram showing relationship of major classes

```
                                        ANTLRTokenStream
                                        (ATokenStream.h)
                                               |
                                               V
 ANTLRParser --> ANTLRTokenBuffer --> DLGLexerBase ---> DLGInputStream
 (AParser.h)     (ATokenBuffer.h)      (DLexerBase.h)     |(DLexerBase.h)
      |                 |                   |             |
      |                 V                   V             +- DLGFileInput
      |           MyTokenBuffer          DLGLexer         |
      |                                (ANTLR generated)  |
      V                                                   +- DLGStringInput
  MyParser (generated by ANTLR from myFile.g)
     MyParser.h      (class header)
     MyParser.cpp    (static variable initialization)
     myFile.cpp      (implementation code for rules)
```

#65. Required AST constructors: AST(), AST(ANTLRTokenPtr), and AST(X x,Y y) for #[X x,Y y]

#66. Tokens are supplied as demanded by the parser. They are "pulled" rather than "pushed"

```
ANTLRParser::consume()
--> ANTLRTokenBuffer::getToken()
--> ANTLRTokenBuffer::getANTLRToken()
--> DLGLexer::getToken()
--> MyToken::makeToken(ANTLRtokenType,lexText,line)
```

#67. The lexer can access parser information using member function `getParser()`

The member functions ANTLRTokenBuffer::getParser() and DLGLexerBase::getParser() return a pointer to the current parser.

#68. Additional notes for users converting from C to C++ mode

In general:	zz*name*	=>	*name*, *_name*, or *name*()
example:	`zzlextext`	=>	`_lextext`, `lextext()`
except for:	`zzchar`	=>	`ch`
In DLGLexerBase:	`NLA=tokenCode`	=>	`return tokenCode`
	`line++`	=>	`newline()`
	`line=value`	=>	`_line=value`
	and	=>	`set_line(value)`
			set_line() is not available in vanilla 1.33
	zztokens[*i*]	=>	*parserClassName*::tokenName(*i*)
	and	=>	getParser()->parserTokenName(*i*)
			tokenName() is not available in vanilla 1.33 *parserTokenName is not available in vanilla 1.33*
	`zzendcol`	=>	`_endcol,set_endcol(),get_endcol()`
	`zzbegcol`	=>	`_begcol,set_begcol(),get_begcol()`

#69. Use the macro mytoken(*expr*) to convert an ANTLRTokenPtr to an ANTLRToken *

#70. When using reference counted tokens be careful about saving a pointer generated by `myToken()`

Reference counted tokens are deallocated when the reference count maintained by ANTLRTokenPtr reaches zero. Saving a pointer generated by `mytoken()` subverts the reference counting scheme because the reference is not counted. If such a pointer were stored in an AST it is likely that the token will soon be deallocated leaving you with a pointer to garbage.

#71. LA(*i*) is a cache of LT(*i*) values used by the parser — it is valid only for $i \le k$

Contributed by John Lilley (jlilley@empathy.com)

#72. To disable reference counting of ANTLRTokens use *parserName*.noGarbageCollectTokens()

#73. For string input use DLGStringInput(const DLGChar *string) for a DLGInputStream

#74. Use `#lexmember <<...>>` to insert code into the DLGLexer class

#75. Use `#lexprefix <<...>>` to insert `#include` statements into the DLGLexer file

#76. How to change the default error reporting actions of DLG and ANTLR

For DLG:

```
#lexmember <<
      virtual ANTLRTokenType erraction() {
            Your error action goes here.
            Normally you'll want to end by calling the standard action:
            return DLGLexerBase::erraction();
      };
>>
```

For ANTLR, start by adding a virtual member function "syn" to your parser:

```
class MyParser {
<<    public:
            virtual void syn(
                  ANTLRAbstractToken *        tok,
                  ANTLRChar *                 egroup,
                  SetWordType *               eset,
                  ANTLRTokenType              etok,
                  int                         k);
>>
...
}
```

The actual error recovery code for ANTLR is rather long. For an example of adding column information to the syntax error message see Example #5.

ASTs

#77. To enable AST construction (automatic or explicit) use the ANTLR –gt switch

#78. Use ANTLR option -newAST to make AST creation a member function of the parser

This allows a user to define a parser member function to create an AST object. This is useful for factory methods and cases where the AST are "owned" by a particular instance of a parser.

	Standard AST constructors	ANTLR -newAST option AST
Automatic conversion of token	`new AST(ANTLRTokenPtr)`	`newAST(ANTLRTokenPtr)`
Manual construction:	`#(X,Y,Z) => new AST(X,Y,Z)`	`#(X,Y,Z) => newAST(X,Y,Z)`

#79. Use symbolic tags (rather than numbers) to refer to tokens and ASTs in rules

prior to version 1.30:	`rule! : x y`	`<<#0=#(#1,#2);>>   ;`
with version 1.30:	`rule! : xx:x yy:y`	`<<#0=#(#xx,#yy);>> ;`

The symbolic tags are implemented as pointers to ASTs. The pointers are initialized to 0 at the start of the rule and remain defined for the entire rule. See Item #62. Rules no longer return pointers to tokens (Item #97

#80. Constructor AST(ANTLRTokenPtr) is automatically called for terminals when ANTLR –gt switch is used

This can be suppressed using the "!" operator.

#81. If you use ASTs you have to pass a root AST to the parser

```
ASTBase     *root=NULL;
...
Parser.startRule(&root,otherArguments);
root->preorder();
root->destroy();
```

#82. Use ast–>destroy() to recursively descend the AST tree and free all sub-trees

#83. Don't confuse `#[...]` with `#(...)`

The first creates a single AST node using an AST constructor (which is usually based on an ANTLRToken or an ANTLRTokenType). It converts lexical information to an AST.

The second creates an AST tree or list (usually more than a single node) from other ASTs by filling in the "down" field of the first node in the list to create a root node, and the "sibling" fields of each of the remaining ASTs in the lists. It combines existing ASTs to create a more complex structure.

```
#token ID          "[a-z]*"
#token COLON       ":"
#token Stmt_With_Label

id! : name:ID    <<#0=#[Stmt_With_Label,$name->getText()];>> ; /* a1 */
```

The new AST (a single node) contains Stmt_With_Label in the token field — given a traditional version of AST::AST(ANTLRTokenType,char *).

```
        rule! : name:id COLON e:expr   <<#0=#(#name,#e);>> ;      /* a2 */
```

Creates an AST list with "name" at its root and "e" as its first (and only) child.

The following example (a3) is equivalent to a1, but more confusing, because the two steps above have been combined into a single action:

```
  rule! : name:ID COLON e:expr                                     /* a3 */
              <<#0=#(#[Stmt_With_Label,$name->getText()],#e);>>;
```

#84. The make-a-root operator for ASTs ("^") can be applied only to terminals (#token, #tokclass, #tokdef)

A child rule might return a tree rather than a single AST. Were this to happen it could not be made into a root as it is *already* a root and the corresponding fields of the structure are in use. To make an AST returned by a called rule a root use the expression: #(root-rule, sibling1, sibling2, sibling3).

```
              addOp            : "\+" | "\-";
              #tokclass AddOp  { "\+"   "\-"}
  /* OK    */ add !            : expr ("\+"^ expr) ;
  /* Wrong */ addExpr !        : expr (addOp^ expr) ;
  /* OK    */ addExpr !        : expr (AddOp^ expr);
```

#85. An already constructed AST tree cannot be the root of a new tree

An AST tree (unless it's a trivial tree with no children) already has made use of the "down" field in its structure. Thus one should be suspicious of any constructs like the following:

```
  rule! : anotherRule:rule2........ <<#0=#(#anotherRule,...);>> ;
```

#86. Don’t assign to #0 unless automatic construction of ASTs is disabled using the "!" operator on a rule

```
  a! : xx:x yy:y zz:z <<#0=#(#xx,#yy,#zz);>> ; // ok
  a  : xx:x yy:y zz:z <<#0=#(#xx,#yy,#zz);>> ; // NOT ok
```

The reason for the restriction is that assignment to #0 will cause any ASTs pointed to by #0 to be lost when the pointer is overwritten.

#87. The statement in Item #86 is stronger than necessary

You can assign to #0 even when using automated AST construction if the old tree pointed to by #0 is part of the new tree constructed by `#(...)`. For example:

```
        #token Comma      ","
        #token Stmt_List

        stmt_list: stmt (Comma stmt)*  <<#0=#(#[Stmt_List],#0);>> ;
```

The automatically constructed tree pointed to by #0 is just put at the end of the new list, so nothing is lost. If you reassign to #0 in the middle of the rule, automatic tree construction will result in the addition of remaining elements at the end of the new tree. This is not recommended by TJP.

Special care must be used when combining the make-a-root operator (e.g. rule: expr Op^ expr) with this transgression (assignment to #0 when automatic tree construction is selected).

#88. A rule that constructs an AST returns an AST even when its caller uses the "!" operator

#89. (C++ mode) Without ANTLRRefCountToken, a token which isn’t used in an AST will result in lost memory

For a rule like the following:

```
        rule : FOR^ lValue EQ! expr TO! expr BY! expr ;
```

the tokens "EQ", "TO", and "BY" are not incorporated into any AST. In C mode the memory they occupied (they are called *attributes* in C mode) would be recovered on rule exit. In C++ mode their memory will be lost unless the ANTLRToken class is derived from ANTLRRefCountToken. Another approach is to use the NoLeakToken class from Example #4.

#90. When passing `#(...)` or `#[...]` to a subroutine it must be cast from "ASTBase *" to "AST *"

Most of the PCCTS internal routines are declared using ASTBase rather than AST because they don’t depend on behavior added by the user to class AST. Usually PCCTS hides this by generating explicit casts, but in the case of subroutine arguments the hiding fails and the user needs to code the cast manually. See also Item #163.

#91. Some examples of `#(...)` notation using the PCCTS list notation

See page 45 of the 1.00 manual for a description of the PCCTS list notation.

```
a: A ;
b: B ;
c: C ;

#token T_abc

r : a b c    <<;>>                   ;/* AST list (0 A B C) without root */
r!: a b c    <<#0=#(0,#1,#2,#3);>> ;/* AST list (0 A B C) without root */
r : a! b! c! <<#0=#(0,#1,#2,#3);>> ;/* AST list (0 A B C) without root */
r : a^ b c                           ;/* AST tree (A B C) with root A    */
r!: a b c    <<#0=#(#1,#2,#3);>>    ;/* AST tree (A B C) with root A    */
r!: a b c    <<#0=#(#[T_abc],#1,#2,#3);>>
                                     ;/* AST tree (T_abc_node A B C)     */
                                      /*   with root T_abc_node          */
r : a b c    <<#0=#(#[T_abc],#0);>>        ;  /* the same as above      */
r : a! b! c! <<#0=#(#[T_abc],#1,#2,#3);>> ;  /* the same as above      */
```

#92. A rule which derives epsilon can short circuit its caller's explicitly constructed AST

When a rule derives epsilon it will return an AST value of 0. As the routine which constructs the AST tree (ASTBase::tmake) has a variable length argument list which is terminated by 0, this can cause problem with `#(...)` lists that have more than two elements:

```
rule !     : DO body:loop_body END_DO  <<#0=#(#[DO],#body,#[END_DO];>> ;
loop_body : { statement_list } ;      /* can return 0 on DO END_DO  */
```

Although this particular example could be handled by automatic tree construction, the problem is a real one when constructing a tree by adding more than one sibling at a time. This problem does not exist for automatically constructed AST trees because those trees are constructed one element at a time. Contributed by T. Doan (tdoan@bnr.ca).

#93. How to use automatic AST tree construction when a token code depends on the alternative chosen

Suppose one wants to make the following transformation:

```
rule :  lv:lhs ;          => #(#[T_simple],#lv)
rule :  lv:lhs  rv:rhs ;  => #(#[T_complex],#lv,#rv)
```

Both lhs and rhs considered separately may be suitable for automatic construction of ASTs, but the change in token type from "T_simple" to "T_complex" appears to require manual tree construction. Use the following idiom:

```
        rule : lhs (
                     | rhs   <<#0=#(#[T_complex],#0);>>
                     ()      <<#0=#(#[T_simple],#0);>>
                   ) ;
```

Another solution:

```
        rule : <<ANTLRTokenType t=T_simple;>>
                    l:lhs { r:rhs <<t=T_complex;>> } <<#0=#(#[t],#0);>> ;
```

#94. For doubly linked ASTs derive from class ASTDoublyLinkedBase and call tree–>double_link(0,0)

The ASTDoublyLinkedBase class adds "up" and "left" fields to the AST definition, but it does not cause them to be filled in during AST construction. After the tree is built call tree->double_link(0,0) to traverses the tree and fill in the up and left fields.

#95. When ASTs are constructed manually the programmer is responsible for deleting them on rule failure

It is worth a little bit of extra trouble to let PCCTS construct the AST for a rule automatically in order to obviate the need for writing a fail action for a rule. A safer implementation might be to maintain a doubly linked list of all ASTs from which an AST is removed when it is destroyed. See class NoLeakAST from Example #5.

#96. To refer to a field of an ANTLRToken within a rule's action use `<<... mytoken($x)->field...>>`

ANTLR puts all "ANTLRToken *" variables in an ANTLRTokenPtr object in order to maintain reference counts for tokens. When the reference counter goes to zero the token is deleted (assuming that the ANTLRToken definition is derived from ANTLRRefCountToken). One result of this is that rule actions which need to refer to a real ANTLRToken field must first convert an ANTLRTokenPtr to an "ANTLRToken *" using the macro "mytoken":

```
number: n:Number <<if (mytoken($n)->value < 0) {...};>>
```

#97. Rules don't return tokens values, thus this won't work: `rule: r1:rule1  <<...$r1...>>`

In earlier versions of PCCTS (C mode) it was accepted practice to assign an attribute to a rule:

```
rule : rule1  <<$0=$1;>>
```

However, with the introduction of symbolic tags for labels (Item #79) this feature became deprecated for C mode (Item #189) and is not even supported for C++ mode. To return a pointer to a token (ANTLRTokenPtr) from a rule use inheritance (See Item #113):

```
statement
      : <<ANTLRTokenPtr t;>>  rule > [t] ;
rule > [ANTLRTokenPtr t]
      : x:X   <<$t=someAction($x);>>
```

It's still standard practice to pass back AST information using assignment to #0 and to refer to such return values using labels on rules. It's also standard practice to refer to tokens associated with *terminals*:

```
rule : xx:X    <<...$xx...>>    // okay: "X" is a terminal (token)
rule : xx:x    <<...$xx...>>    // won't work: "x" is a rule rather
x    : xx:X    <<$x=$xx;>>      //    than a terminal (token)
```

#98. A simple example of rewriting a grammar to remove left recursion

ANTLR can't handle left-handed recursion. A rule such as:

```
expr : expr Op expr
     | Number
     | String
     ;
```

will have to be rewritten to something like this:

```
expr : Number (Op expr)*
     | String (Op expr)*
     ;
```

#99. A simple example of left-factoring to reduce the amount of ANTLR lookahead

Another sort of transformation required by ANTLR is left-factoring:

```
rule : STOP WHEN expr
     | STOP ON expr
     | STOP IN expr
     ;
```

These are easily distinguishable when *k*=2, but with a small amount of work it can be cast into a *k*=1 grammar:

```
rule : STOP ( WHEN expr
            | ON expr
            | IN expr
            ) ;
```

or:

```
rule        : STOP rule_suffix
            ;
rule_suffix : WHEN expr
            | ON expr
            | IN expr
            ;
```

An extreme case of a grammar requiring a rewrite is in Example #11.

#100. ANTLR will guess where to match "@" if the user omits it from the start rule

ANTLR attempts to deduce "start" rules by looking for rules which are not referenced by any other rules. When it finds such a rule it assumes that an end-of-file token ("@") should be there and adds one if the user did not code one. This is the only case, according to TJP, when ANTLR adds something to the user's grammar.

#101. To match any token use the token wild-card expression "." (dot)

This can be useful for providing a context dependent error message rather than the all purpose message "syntax error".

```
if-stmt : IF "\(" expr "\)" stmt
        | IF .   <<printf("If statement requires expression "
                               "enclosed in parenthesis");
                   PARSE_FAIL;        // user defined
                 >>
        ;
```

This particular case is better handled by the parser exception facility.

A simpler example:

```
quoted : "quote" . ;            // quoted terminal
```

#102. The "~" (tilde) operator applied to a #token or #tokclass is satisfied when the input token does *not* match

```
anything : (~ t:Newline)* Newline ;
```

The "~" operator cannot be applied to rules. Use syntactic predicates to express the idea "if this rule doesn't match try to match this other rule".

The element label "t" in the example allows one to examine the token actually matched. Contributed by Tom Nurkkala (tom.nurkkala@powercerv.com).

#103. To list the rules of the grammar grep *parserClassName*.h for "_root" or edit the output from ANTLR –cr

#104. The ANTLR –gd trace option can be useful in sometimes unexpected ways

For example, by suitably defining the functions ANTLRParser::tracein and ANTLRParser::traceout one can accumulate information on how often each rule is invoked. They could be used to provide a traceback of active rules following an error provided that the havoc caused by syntactic predicates' use of setjmp/longjmp is properly dealt with.

#105. Associativity and precedence of operations is determined by nesting of rules

In the example below "=" associates to the right and has the lowest precedence. Operators "+" and "*" associate to the left with "*" having the highest precedence.

```
expr0   : expr1 {"="^ expr0} ;                             /* a1 */
expr1   : expr2 ("\+"^ expr2)* ;                           /* a2 */
expr2   : expr3 ("\*"^ expr3)* ;                           /* a3 */
expr3   : ID ;                                             /* a4 */
```

The more deeply nested the rule the higher the precedence. Thus precedence is "*" > "+" > "=". Consider the expression "x=y=z". Will it be parsed as "x=(y=z)" or as "(x=y)=z" ? The first part of expr0 is expr1. Because expr1 and its descendants cannot match an "=" it follows that all derivations involving a *second* "=" in an expression must arise from the "{...}" term of expr0. This implies right association.

In the following samples the ASTs are shown in the root-and-sibling format used in PCCTS documentation. The numbers in brackets are the serial number of the ASTs. This was created by code from Example #5.

```
a=b=c=d
( = <#2>  a <#1>  ( = <#4>  b <#3>  ( = <#6>  c <#5>  d <#7>  ) ) ) NL <#8>
a+b*c
( + <#2>  a <#1>  ( * <#4>  b <#3>  c <#5>  ) ) NL <#6>
a*b+c
( + <#4>  ( * <#2>  a <#1>  b <#3>  ) c <#5>  ) NL <#6>
```

#106. #tokclass can replace a rule consisting only of alternatives with terminals (no actions)

One can replace:

```
        addOp           : "\+" | "\-" ;
```

with:

```
        #tokclass AddOp { "\+"  "\-" }
```

This replaces a modest subroutine with a simple bit test. A #tokclass identifier may be used in a rule wherever a simple #token identifier may be used.

The other work-around is much more complicated:

```
  expr1! : left:expr2 <<#0=#l;>>
                (op:addOp right:expr2  <<#0=#(#op,#left,#right);>> )* ;
  addOp  : "\+" | "\-" ;
```

The "!" for rule "expr1" disables automatic constructions of ASTs in the rule. This allows one to manipulate #0 manually. If the expression had no addition operator then the sub-rule "(addOp expr)*" would not be executed and #0 will be assigned the AST constructed by #left. However if there *is* an addOp present then each time the sub-rule is rescanned due to the "`(...)*`" the current tree in #0 is placed as the first of two siblings underneath a new tree. This new tree has the AST returned by addOp as the root. It is a left-leaning tree.

#107. Rather than comment out a rule during testing, add a nonsense token which never matches — See Item #110.

Init-Actions

#108. Don't confuse init-actions with leading-actions (actions which precede a rule)

If the first element following the start of a rule or sub-rule is an action it is always interpreted as an init-action. An init-action occurs in a scope which includes the entire rule or sub-rule. An action which is *not* an init-action is enclosed in "{" and "}" during generation of code for the rule and has essentially zero scope — the action itself.

The difference between an init-action and an action which precedes a rule can be especially confusing when an action appears at the start of an alternative. These *appear* to be almost identical, but they aren't:

```
  b  : <<int i=0;>>  b1 > [i]     /* b1  <<...>> is an init-action       */
     | <<int j=0;>>  b2 > [j]     /* b2  <<...>> is part of the rule     */
     ;                            /*  and will cause a compilation error */
```

On line "b1" the `<<...>>` appears immediately after the beginning of the rule making it an init-action. On line "b2" the `<<...>>` does *not* appear at the start of a rule or sub-rule, thus it is interpreted as a leading action which happens to precede the rule.

This can be especially dangerous if you are in the habit of rearranging the order of alternatives in a rule.

For instance, changing this:

```
        b   : <<int i=0,j=0;>> <<i++;>>  b1 > [i]        /* c1 */
            | <<j++;>>  b1 > [i]                         /* c2 */
            ;
```

to this:

```
        b   : /* empty production */                     /* d1 */
            | <<int i=0,j=0;>> <<i++;>>  b1 > [i]        /* d2 */
            | <<j++;>>  b1 > [i]
            ;
```

or to this:

```
        b
            : <<j++;>>  b1 > [i]                         /* e1 */
            | <<int i=0,j=0;>> <<i++;>>  b1 > [i]        /* e2 */
            ;
```

changes an init-action into a non-init action, and vice-versa.

#109. An empty sub-rule can change a regular action into an init-action

A particularly nasty form of the init-action problem is when an empty sub-rule has an associated action:

```
rule!: name:ID (/* empty */
                     <<#0=#[ID,$name];>>
               | ab:array_bounds
                     <<#0=#[T_array_declaration,$name],#ab);>>
               );
```

Since there is no reserved word in PCCTS for epsilon, the action for the empty arm of the sub-rule becomes the init-action. For this reason it's wise to follow one of the following conventions

– Represent epsilon with an empty subrule "()"

– Put the null rule as the last rule in a list of alternatives:

```
rule!: name:ID (
                 ()  <<#0=#[ID,$name];>>
                 | ab:array_bounds
                       <<#0=#[T_array_declaration,$name],#ab);>>
               );
```

The cost of using "()" to represent epsilon is small.

#110. Commenting out a sub-rule can change a leading-action into an init-action

Suppose one comments out a rule in the grammar in order to test an idea:

```
rule                              /* a1 */
        : <<init-action;>>        /* a2 */
////      rule_a                  /* a3 */
        | rule_b                  /* a4 */
        | rule_c                  /* a5 */
        ;
```

In this case one only wanted to comment out the "rule_a" reference in line a3. The reference is indeed gone, but the change has introduced an epsilon production — which probably creates a large number of ambiguities. Without the init-action the ":" would have probably have been commented out also, and ANTLR would report a syntax error — thus preventing one from shooting oneself in the foot. See Item #107.

Commenting out a rule can create orphan rules which can lead to misleading reports of ambiguity in the grammar. To detect orphan rules use the ANTLR –info o switch.

#111. Init-actions are executed just once for sub-rules: `(...)+`, `(...)*`, and `{...}`

Consider the following example from section 3.6.1 (page 29) of the 1.00 manual:

```
a : <<List *p=NULL;>>                   // initialize list
    Type
    (  <<int i=0;>>                     // initialize index
       v:Var <<append(p,i++,$v);>>
    )*
    <<OperateOn(p);>>
  ;
```

Inheritance

#112. Downward inherited variables are just normal C arguments to the function which recognizes the rule

If one is using downward inheritance syntax to pass results back to the caller (really upward inheritance !) then it is necessary to pass the *address* of the variable which will receive the result.

#113. Upward inheritance returns arguments by passing back values

If the rule has more than one item passed via upward inheritance then ANTLR creates a `struct` to hold the result and then copies each component of the structure to the upward inheritance variables.

```
#token  T_int
#token  T_real
#token  T_complex

class P {
...
number : <<int useRadix=10;int iValue;double rValue;double rPart,iPart;>>
         { radix > [useRadix] }
                intNumber [useRadix] > [iValue]
        | realNumber > [rValue]
        | complexNumber > [rPart,iPart]
;
complexNumber > [double rPart,double iPart] :
        "\[" realNumber > [$rPart] "," realNumber > [$iPart] "\]"
;
realNumber > [double result] :
        v:"[0-9]+.[0-9]*"              <<$result=toDouble($v);>>
;
radix > [int i] : v:"%[0-9]+"          <<$i=toInt($v);>>
;
intNumber [int radix] > [int result] :
        v:"[0-9]+"                     <<$result=toInt($v);>>
;
}
```

This example depends on the use of several constructors for ASTs and user defined routines toInt() and toDouble().

#114. Be careful about passing via upward inheritance LT(i)->getText() if using ANTLRCommonToken

If the token is destroyed due to the reference count going to 0 will the text still be valid ?

#115. ANTLR –gt code will include the AST with downward inheritance values in the rule's argument list

#116. Predefine the PURIFY macro if you are passing objects using upward inheritance

The default PURIFY macro zeroes the memory occupied by objects passed via upward inheritance. If your object has a non-trivial default constructor this could cause problems.

Syntactic Predicates

The terms "infinite lookahead", "guess mode", and "syntactic predicate" all imply use of the same facility in PCCTS to provide a limited amount of backtracking by the parser. In this case we are *not* referring to backtracking in DLG or other lexers. The term "syntactic predicate" emphasizes that it is handled by the parser. The term "guess mode" emphasizes that the parser may have to backtrack. The term "guess mode" may also be used to distinguish two mutually exclusive modes of operation in the ANTLR parser:

— Normal mode: A failure of the input to match the rules is a syntax error. The parser executes actions, constructs ASTs, reports syntax errors it finds (or invokes parser exception handling) and attempts automatic recovery from the syntax errors. There is no provision for backtracking in this mode.

— Guess mode: The parser attempts to match a "`(...)?`" block and knows that it must be able to backtrack if the match fails. In this case the parser does *not* execute user-actions (except init-actions), nor does it construct ASTs. Failed semantic predicates cause backtracking, except when they are validation predicates.

In C++ mode, lookahead uses a sliding window of tokens whose initial size is specified when the ANTLRTokenBuffer is constructed. In C mode the entire input is read, processed, and tokenized by DLG before ANTLR begins parsing. The term "infinite lookahead" derives from the initial implementation in ANTLR C mode.

#117. Normal actions are suppressed while in guess mode because they have side effects

#118. Automatic construction of ASTs is suppressed during guess mode because it is a side effect

#119. Syntactic predicates should not have side-effects

If there is no match then the rule which uses the syntactic predicate won't be executed.

#120. How to use init-actions to create side-effects in guess mode (despite Item #119)

If you absolutely have to have side-effects from syntactic predicates one can exploit the fact that ANTLR always executes init-actions, even in guess mode:

```
rule   : (prefix)? A
       | B
       ;
prefix : <<regular-init-action-that's-always-executed>>
         A ( <<init-action-for-empty-subrule>> ) B
       ;
```

The init-actions in "prefix" will always be executed (perhaps several times) in guess-mode. Contributed by TJP.

#121. With values of *k*>1 or infinite lookahead mode one cannot use feedback from parser to lexer

As infinite lookahead mode can cause large amounts of the input to be scanned by DLG before ANTLR begins parsing one cannot depend on feedback from the parser to the lexer to handle things like providing special token codes for items which are in a symbol table (the "lex hack" for `typedefs` in the C language). Instead one *must* use semantic predicates which allow for such decisions to be made by the parser or place such checks in the ANTLRTokenBuffer routine getToken() which is called every time the parser needs another token. See Example #10.

#122. Can't use interactive scanner (ANTLR –gk option) with ANTLR infinite lookahead

#123. Syntactic predicates are implemented using setjmp/longjmp — beware C++ objects requiring destructors

Semantic Predicates

#124. Semantic predicates have higher precedence than alternation: `<<>>? A|B` means `(<<>>? A)|B`

#125. Get rid of warnings about missing LT(i) by using a comment: `/* LT(i) */`

#126. It is sometime desirable to use leading actions to inhibit hoisting of semantic predicates

#127. Any actions (except init-actions) inhibit the hoisting of semantic predicates

Here is an example of an empty leading action whose sole purpose is to inhibit hoisting of semantic predicates appearing in rule2 into the prediction for rule1. Note the presence of the empty init-action (See Item #108).

```
rule1     : <<;>> <<>> rule2
          | rule3
          ;
rule2     : <<semanticPred(LT(1)->getText())>>? ID ;
```

#128. Semantic predicates that use local variables or require init-actions must inhibit hoisting

#129. Semantic predicates that use inheritance variables must not be hoisted

You cannot use downward inheritance to pass parameters to semantic predicates which are *not* validation predicates. The problem appears when the semantic predicate is hoisted into a parent rule to predict which rule to call:

For instance:

```
a   :  b1 [flag]
    |  b2
    ;
b1 [int flag]
    : <<flag && hasPropertyABC(LT(1)->getText())>>? ID ;
b2 : ID ;
```

When the semantic predicate is evaluated within rule "a" to determine whether to call b1, b2, or b3 the compiler will discover that there is no variable named "flag" for procedure "a()". If you are unlucky enough to have a variable named "flag" in a() then you will have a *very* difficult-to-find bug.

#130. A semantic predicate which is not at the left edge of a rule becomes a validation predicate

Decisions about which rule of a grammar to apply are made before entering the code which recognizes the rule. If the semantic predicate is not at the left edge of the production then the decision has already been made and it is too late to change rules based on the semantic predicate. In this case the semantic predicate is evaluated only to verify that it is true and is termed a "validation predicate".

#131. Semantic predicates are not always hoisted into the prediction expression

Even if a semantic predicate is on the left edge there is no guarantee that it will be part of the prediction expression. Consider the following two examples:

```
a  :  <<semantic-predicate>>?  ID glob        /* a1 */
   |  ID glob                                 /* a2 */
  ;
b  :  <<semantic-predicate>>?  ID glob        /* b1 */
   |  Number glob                             /* b2 */
  ;
```

With *k*=1 rule "a" requires the semantic predicate to disambiguate alternatives a1 and a2 because the rules are otherwise identical. Rule "b" has a token type of Number in alternative b2 so it can be distinguished from b1 without evaluation of the semantic predicate during prediction. In both cases the semantic predicate will be validated by evaluation inside the rule.

#132. Semantic predicates can't be hoisted into a sub-rule: "`{x} y`" is not exactly equivalent to "`x y | y`"

Consider the following grammar extract:

```
class Expr {
    e1 : (e2)+ END ;
    xid: <<is_xid(LT(1)->getText())>>? ID ;
    yid: <<is_yid(LT(1)->getText())>>? ID ;

  /* Works        */  e2:  xid "." yid | yid ;      /* a1 */
  /* Doesn't work */  e2:  {xid "."} yid ;          /* a2 */
}
```

Alternatives a1 and a2 appear to be equivalent, but a1 works on input "abc" and a2 doesn't because only the semantic predicate of xid is hoisted into production e1 (but not the semantic predicate of yid).

Explanation by TJP: These alternatives are not really the same. The *language* described however is the same. The rule:

```
e2: {xid "."} yid ;
```

is shorthand for:

```
e2: (xid "." | /* epsilon */ ) yid ;
```

Rule e2 has no decision to make here — hence, yid does not get its predicate hoisted. The decision to be made for the empty alternative does not get the predicate from yid hoisted because one can't hoist a predicate *into* a subrule from beyond the subrule. The program might alter things in the subrule so that the predicate is no longer valid or becomes valid. Contributed by Kari Grano (kari.grano@varian.com).

#133. How to change the reporting of failed semantic predicates

To make a global change #define the macro zzfailed_predicate(string) prior to the #include of pccts/h/AParser.h

One can change the handling on a case-by-case basis by using the "failed predicate" action which is enclosed in "[" and "]" and follows immediately after the predicate:

```
a : <<isTypedef(LT(1)->getText())>>?
            [{printf("Not a typedef\n");};]  ID ;
```

For an example of conversion of a failed semantic predicate into a parser exception see Example #13.

#134. A semantic predicate should be free of side-effects because it may be evaluated multiple times

Even in simple grammars semantic predicate are evaluated at least twice: once in the prediction expression for a rule and once inside the rule as a validation predicate to make sure the semantic predicate is valid.

A semantic predicate may be hoisted into more than one prediction expressions.

A prediction expression may be evaluated more than once as part of syntactic predicates (guess mode).

#135. There’s no simple way to avoid evaluation of a semantic predicate for validation after use in prediction

#136. What is the "context" of a semantic predicate ?

Answer due to TJP: The context of a predicate is the set of *k*-strings (comprised of lookahead symbols) that can be matched following the execution of a predicate. For example,

```
a : <<p>>? alpha ;
```

The context of "p" is Look(alpha) where Look(alpha) is the set of lookahead *k*-strings for alpha.

```
class_name: <<isClass(LT(1)->getText())>>? ID ;
```

The context of `<<isClass ...>>?` is ID for *k*=1. Only *k*=1 is used since only LT(1) is referenced in the semantic predicate. It is important to use "–prc on" for proper operation. The old notation:

```
class_name: <<LT(1)==ID ? isClass(LT(1)->getText()) : 1>>? ID ;
                     /* Obsolete notation incompatiable with -prc on */
```

shouldn’t be used for new grammars — it is not compatible with "–prc on". The only reason "–prc on" is not the default is backward compatibility.

Here is an example that won't work because it doesn't have context check in the predicates:

```
a          : ( class_name | Num )
           | type_name
           ;
class_name : <<isClass(LT(1)->getText())>>? ID ;
type_name  : <<isType(LT(1)->getText())>>? ID ;
```

The prediction for production one of rule "a" will be:

```
if ( LT(1) in { ID, Num } && isClass(LT(1)->getText()))  {...}
```

Clearly, Num will never satisfy isClass(), so the production will never match.

When you ask ANTLR to compute context, it can check for missing predicates. With –prc on, for this grammar:

```
        a   : b
            | <<isVar(LT(1)->getText())>>?      ID
            | <<isPositive(LT(1)->getText()>>?  Num
            ;
        b   : <<isType(LT(1)->getText())>>?     ID
            | Num
            ;
```

ANTLR reports:

```
        warning alt 1 of rule itself has no predicate to resolve
                                    ambiguity upon { Num }
```

#137. Use ANTLR option "-info p" for information on how semantic predicates are being handled and hoisted

When the user selects option "-info p" the program will generate detailed information about predicates. If the user selects "-mrhoist on" additional detail will be provided explaining the promotion and suppression of predicates. The output is part of the generated file and sandwiched between #if 0/#endif statements.

Consider the following k=1 grammar:

```
        start : ( all ) * ;            /* 3  */
        all   : a                      /* 4  */
              | b                      /* 5  */
              ;                        /* 6  */
        a     : c B ;                  /* 7  */
        c     : <<pc(LT(1))>>?         /* 8  */
              | B                      /* 9  */
              ;                        /* 10 */
        b     : <<pb(LT(1))>>? X ;     /* 11 */
```

Below is an excerpt of the output for rule "start" when used with -mrhoist on and -info p.

```
#if 0

Hoisting of predicate suppressed by alternative without predicate.
The alt without the predicate includes all cases where the predicate is
false.

   WITH predicate: line 8  infop.g
   WITHOUT predicate: line 9  infop.g

The context set for the predicate:

     B

The lookahead set for the alt WITHOUT the semantic predicate:

     B

The predicate:

  pred  <<  pc(LT(1))>>?
                    depth=k=1  rule c  line 8  infop.g
    set context:
       B

Chain of referenced rules:

    #0  in rule start (line 3 infop.g) to rule all
    #1  in rule all (line 4 infop.g) to rule a
    #2  in rule a (line 7 infop.g) to rule c
    #3  in rule c (line 8 infop.g)

#endif
&&
#if 0

pred  <<  pb(LT(1))>>?
                  depth=k=1  rule b  line 11  infop.g
  set context:
     X

#endif
```

#138. Semantic predicates, predicate context, and hoisting

The interaction of semantic predicates with hoisting is sometimes subtle. Hoisting involves the evaluation of semantic predicates in a rule's parent in order to determine whether the rule associated with the semantic predicate is "viable". There are two ways to generate code for semantic predicates which are "hoisted" into a parent rule. With "–prc off", the default, the behavior of semantic predicates resembles gates which enable or disable various productions. With "–prc on" the behavior of semantic predicates resemble a token for which its token type is determined by run-tine information rather than by purely lexical information. It is important to understand what "-prc on" does, when to use semantic predicates, and when to choose an alternative method of using semantic information to guide the parse. We start with a grammar excerpt which does not require hoisting, then add a rule which requires hoisting and show the difference in code with predicate context computation off (the default) and on.

Consider:

```
statement
        : upper
        | lower
        | number
        ;
upper   : <<isU(LT(1)->getText())>>? ID ;
lower   : <<isL(LT(1)->getText())>>? ID ;
number  : Number ;
```

The code generated (with one ambiguity warning) resembles:

```
if (LA(1)==ID && isU) {
    upper();
} else if (LA(1)==ID && isL) {
    lower();
} else if (LA(1)==Number) {
    number();
...
```

Now the need for a non-trivial prediction expression is introduced:

```
parent   : statement
         | ID
         ;
statement
         : upper
         | number
         ;
```

Running ANTLR causes one ambiguity warning. The code for "statement" resembles:

```
if ( (LA(1)==ID || LA(1)==Number) && isU) {
      statement();
} else if (LA(1)==ID) {
...
```

Even if LA(1) is a Number the semantic predicate isU() will be evaluated. Depending on the way that isU is written it may or may not be meaningful. This is exactly the problem addressed by predicate computation. With "–prc on" one receives two ambiguity warnings and the code for "statement" resembles:

Code –prc on	Outline format –prc on
`if ( (LA(1)==ID ||`	`&&`
`      LA(1)==Number) &&`	`  ||`
`     ( !(LA(1)==ID) ||`	`    LA(1)==ID`
`        (LA(1)==ID && isU)) {`	`    LA(1)==Number`
`           statement();`	`  ||`
`} else if (LA(1)==ID) {`	`    !            <===== not ...`
`   ...`	`      LA(1)==ID    <===== an ID`
	`    isU(LT(1)->getText())`

The important thing to notice is the call to isU() is guarded by a test that insures that the token is indeed an ID.

The following does not change anything because ANTLR already knows that the lookahead context for the semantic predicates can only be "ID":

```
upper   : (ID)? => <<isU(LT(1)->getText())>>? ID ;
```

#139. Another example of predicate hoisting

Consider the following grammar fragment which uses semantic predicates to disambiguate an ID in rules ca and cb:

```
a : ( { b | X } Eol)* "@" ;                   /* a1 */
b : c ID ;                                    /* a2 */
c : {ca} {cb} ;                               /* a3 */

ca: <<pa(LT(1)->getText())>>? ID;             /* a4 */
cb: <<pb(LT(1)->getText())>>? ID;             /* a5 */
```

The code generated for rule c resembles:

```
if (LA(1)==ID) && pa(...)) {                  /* b1 */
   ca();                                      /* b2 */
} else {                                      /* b3 */
   goto exit;                                 /* b4 */
};                                            /* b5 */
```

The test of "pb" does not even appear. The problem is that the element "{cb}" is not at the left edge of rule c – even though "{ca}" is an optional element. Although "ca" may match epsilon, its presence in rule c still blocks the hoisting of the predicate in rule cb.

A first effort to solve this problem is to rewrite rule c so as to place "cb" on the left edge of the production:

```
c : ()                                     /* c1 */
  | ca {cb}                                /* c2 */
  | cb                                     /* c3 */
  ;                                        /* c4 */
```

The code generated for rule c now resembles:

```
if (LA(1)==ID) {                            /* d1 */
  ;                                         /* d2 */
} else if (LA(1)==ID && pa(...)) {          /* d3 */
   ...                                      /* d4 */
```

It is clear that rules ca and cb are now unreachable because any ID will always match the test at line d1. In fact, this will cause an error message because PCCTS is able to recognize the problem. The order of alternatives should be changed to:

```
c : ca {cb}                                 /* e1 */
  | cb                                      /* e2 */
  | ()                                      /* e3 */
  ;                                         /* e4 */
```

However our problems aren't over yet. The code generate for the "(...)*" test in rule "a" resembles:

```
while ( (LA(1)==X || LA(1)==Eol || LA(1)==ID) &&   /* f1 */
        (pa(...) || pb(...))) {                    /* f2 */
   ...                                             /* f3 */
```

If both pa and pb are false then the body of the rule is never entered even though it should match an X or and ID using the rule on line a2 when rule c derives epsilon.

This can be fixed by using the ANTLR "-mrhoist on" option which suppresses the test for pa and pb in rule "a" because ID is a viable alternative. See Item #140.

Contributed by Sigurdur Asgeirsson (sigurasg@menandmice.com).

#140. Example of predicate hoisting and suppression with the ANTLR option -mrhoist on

Consider the following grammar fragment:

```
a  : bc                         /* 2  */
   | d                          /* 3  */
   ;                            /* 4  */
bc : b                          /* 5  */
   | c                          /* 6  */
   ;                            /* 7  */
b  : <<pb(LT(1))>>? ID ;        /* 8  */
c  : ID ;                       /* 9  */
d  : <<pd(LT(1))>>? ID ;        /* 10 */
```

With no predicate context computation (-prc off) the generated code for rule "a" resembles:

```
if (LA(1)==ID && pb) {
   bc();
} else if (LA(1)==ID && pd) {
   d();
} ...
```

The code for the first alternative is incorrect because c (and therefore bc) is still a viable choice when pb is false. With the default predicate context computation (-prc on) the generated code for rule "a" resembles:

```
if (LA(1)==ID && ((! LA(1)==ID) || pb)) {
    bc();
} else if (LA(1)==ID && ((! LA(1)==ID) || pd)) {
    d();
} ...
```

This is more complex, but a close look at the code shows that this is no better than the code generated by -prc off.

The extra code checks that the predicate pb is evaluated only when the lookahead is ID. Since we always expect the predicate to be ID this provides no extra power.

With option -mrhoist on the code generated for rule "a" resembles:

```
        if (LA(1)==ID) {
          bc();
        } else if (LA(1)==ID && ((! LA(1)==ID) || pd)) {
          d();
        } ...
```

This is the correct code sequence. However, we have a warning:

```
  c.g(2) : warning: alt 1 line 2 and alt 2 line 3 of of the rule itself
       These alts have ambig lookahead sequences resolved by a predicate for
       the second choice. The second choice may not be reachable.
       You may want to use a complementary predicate or rearrange the alts
```

The problem is that the prediction expression for bc in rule a will always match an ID. This is the correct prediction expression because rule c is always viable when the lookahead is ID, but the alternative for rule d in rule a is no longer reachable when the lookahead is ID. Hence the warning.

We now return to the topic of predicate hoisting. Using the ANTLR -info p option we would find the following note in the generated code:

```
#if 0

Hoisting of predicate suppressed by alternative without predicate.
The alt without the predicate includes all cases where the predicate is
false.

   WITH predicate: line 5  c.g
   WITHOUT predicate: line 6  c.g

The context set for the predicate:

     ID

The lookahead set for the alt WITHOUT the semantic predicate:

     ID

The predicate:

  pred  <<  pb(LT(1))>>?
                    depth=k=1  rule b  line 8  c.g
    set context:
       ID

Chain of referenced rules:

    #0  in rule a (line 2 c.g) to rule bc
    #1  in rule bc (line 5 c.g)

#endif
  if ( (LA(1)==ID) ) {
    bc();
  }
  else {
    if ( (LA(1)==ID)&&
#if 0

pred  <<  pd(LT(1))>>?
                  depth=k=1  rule d  line 10  c.g
  set context:
     ID

#endif
(!(((LA(1)==ID)))||(pd(LT(1)))) ) {
      d();
    ...
```

There are additional examples in the CHANGES_FROM_133*.TXT files.

#141. The context guard `(...)? && <<predicate>>?` vs. `(...) => <<predicate>>?`

The idea for the new (...)? && <<predicate>>? is due to Reinier van der Born (reinier@vnet.ibm.com)

The `(...)? =>` predicate guard does not apply the predicate if the context guard doesn't match, whereas the `(...)? &&` form requires both the predicate guard and the predicate to be true to make the alternative viable. What is the significance ?

If you have a predicate which is *not* on the "leading edge" it cannot be hoisted. Suppose you need a predicate that looks at LA(2). You must introduce it manually. The classic example is:

```
castExpr : LP typeName RP
         | ....
         ;

typeName : <<isTypeName(LT(1))>>?  ID
         | STRUCT ID
```

```
                  ;
```

The problem is that isTypeName() isn't on the leading edge of castExpr because of the LP which precedes it. Thus it will not be hoisted into the prediction expression for castExpr. The *first* attempt to fix it is:

```
        castExpr : <<isTypeName(LT(2))>>? LP typeName RP
                 | ....
                 ;
```

Unfortunately, this won't work because it ignores the problem of STRUCT. The solution is to apply isTypeName() in castExpr if LA(2) is an ID and don't apply it when LA(2) is STRUCT:

```
        castExpr : (LP ID)? => <<isTypeName(LT(2))>>? LP typeName RP
                 | ....
                 ;
```

In conclusion, the "=>" style guarded predicate is useful when:

a. The tokens required for the predicate are not on the leading edge.

b. there are alternatives in the expression selected by the predicate for which the predicate is inappropriate.

If (b) were false, then one could use a simple predicate (assuming "-prc on"):

```
        castExpr : <<isTypeName(LT(2))>>? LP typeName RP
                 | ....
                 ;

        typeName : <<isTypeName(LT(1))>>?  ID
                 ;
```

So, when do you use the "&&" style guarded predicate ? The new-style "&&" predicate should always be used with predicate context. The context guard is in *addition* to the automatically computed context. Thus it useful for predicates which depend on the token type for reasons other than context.

The following example is contributed by Reinier van den Born.

This grammar has two ways to call functions:

A "standard" call syntax with parens and comma separated arguments.
A shell command like syntax (no parens and spaces separate arguments).

The former allows a variable to hold the name of the function, the latter can be used to call external commands. The grammar (simplified) looks like this:

```
fun_call    :     ID "(" { expr ("," expr)* } ")"
                             /* ID is function name */
            | "@" ID "(" { expr ("," expr)* } ")"
                             /* ID is var containing fun name */
            ;

command     : ID expr*       /* ID is function name */
            | path expr*     /* path is external command name */
            ;

path        : ID             /* left out slashes and such */
            | "@" ID         /* ID is environment var */
            ;

expr        : ....
            | "(" expr ")";

call        : fun_call
            | command
            ;
```

Obviously the call is wildly ambiguous. This is more or less how this is to be resolved:

A call begins with an ID or an "@" followed by an ID.

If it is an ID and:

a. it is an external command name => command
b. it is followed by a paren => fun_call
c. otherwise => command

If it is an @ and:

a. the ID is a var name => fun_call
b. otherwise => command

One can implement these rules quite neatly using && predicates:

```
call        : ("@" ID)? && <<isVarName(LT(2))>>? fun_call
            | (ID)?     && <<isExtCmdName>>?     command
            | (ID "(")?                          fun_call
            |                                    command
            ;
```

#142. Experimental ANTLR option -mrhoistk on for suppression of predicates with lookahead depth *k* > 1

The ANTLR option -mrhoist provides fairly complete handling of predicates with lookahead depth of 1. The handling of predicates with lookahead depth greater than one is more complicated and the solution provided by PCCTS is good, but not complete.

Consider the following grammar with -ck 2 and the predicate in rule "a" with depth 2:

```
r1  : (ab)* "@" ;                        /* 2 */
ab  : a                                  /* 3 */
    | b                                  /* 4 */
    ;                                    /* 5 */
a   : (A B)? => <<p(LT(2)>>? A B C ;     /* 6 */
b   : A B C ;                            /* 7 */
```

Without "-mrhoistk on" the predicate would be hoisted into rule r1 in order to determine whether to call rule "ab". However it should *not* be hoisted because, even if p is false, there is a valid alternative in rule b. With "-mrhoistk on" the predicate will be suppressed.

With the "-mrhoistk on" and "-info p" the following information will appear in the generated code:

```
while ( (LA(1)==A)
#if 0

Part (or all) of predicate with depth > 1 suppressed by alternative without
predicate

pred  <<  p(LT(2)>>?
                  depth=k=2  ("=>" guard)  rule a  line 6  t1.g
  tree context:
    (root = A
       B
    )

The token sequence which is suppressed: ( A B )
The sequence of references which generate that sequence of tokens:

   1 to ab       r1/1    line 2   t1.g
   2 ab          ab/1    line 3   t1.g
   3 to b        ab/2    line 4   t1.g
   4 b           b/1     line 7   t1.g
   5 #token A    b/1     line 7   t1.g
   6 #token B    b/1     line 7   t1.g

#endif
 ) {
      ab();
    }
  }
```

There is a second, more complex, example of the -mrhoistk option in the CHANGES_FROM_133*.TXT file.

#143. Use #pred statement to describe the logical relationship of related predicates

A problem with predicates is that each one is regarded as unique and capable of disambiguating cases where two alternatives have identical lookahead. For example, consider:

```
rule : <<pred(LATEXT(1))>>? A
     | <<pred(LATEXT(1))>>? A
     ;
```

Even though the two alternatives are identical, this will not cause any error messages or warnings to be issued by the original versions of PCCTS. One could compare the text of the predicate, but this is not a robust or satisfactory solution.

The #pred statement allows one to give a symbolic name to a "predicate literal" or a "predicate expression" in order to refer to it in other predicate expressions or in the rules of the grammar.

The predicate literal associated with a predicate symbol is C or C++ code which can be used to test the condition. A predicate expression defines a predicate symbol in terms of other predicate symbols using "!", "&&", and "||". A predicate symbol can be defined in terms of a predicate literal, a predicate expression, or both.

When a predicate symbol is defined with both a predicate literal and a predicate expression, the predicate literal is used to generate code, but the predicate expression is used to check for two alternatives with identical predicates in both alternatives.

Here are some examples of #pred statements:

```
#pred  IsLabel       <<isLabel(LATEXT(1))>>?
#pred  IsLocalVar    <<isLocalVar(LATEXT(1))>>?
#pred  IsGlobalVar   <<isGlobalVar(LATEXT(1)>>?
#pred  IsVar         <<isVar(LATEXT(1))>>?       IsLocalVar || IsGlobalVar
#pred  IsScoped      <<isScoped(LATEXT(1))>>?    IsLabel || IsLocalVar
```

The predicate IsLocalVar is related to IsGlobalVar (See the definition of IsVar). The #pred attempts to capture this for use in analyzing the predicates appearing that appear in prediction expressions. This is discussed in more detail in the file CHANGES_FROM_133_BEFORE_MR13.TXT.

#144. Disable predicate hoisting explicitly using the pseudo-action: `rule: <<;>> <<nohoist>> ...`

A common error, even among experienced PCCTS users, is to code an init-action to inhibit hoisting rather than a leading action. An init-action does not inhibit hoisting.

This was coded:

```
rule1 : <<;>> rule2
```

This is what was meant:

```
rule1 : <<;>> <<;>> rule2
```

Now the user can code:

```
rule1 : <<;>> <<nohoist>> rule2
```

The following will give an error message:

```
rule1 : <<nohoist>> rule2
```

If the <<nohoist>> appears as an init-action rather than a leading action an error message is issued.

#145. Simplification of predicate expressions when there are multiple references to predicates

When a rule containing a semantic predicate is referenced by more than one alternative of a grandparent rule or other ancestor, a large numbers of semantic predicate references can sometimes be generated. An effort has been made to simplify some of them. The table below summarized the kind of simplification performed. In the table, X and Y stand for single predicates (not trees).

```
(OR X (OR Y (OR Z)))             => (OR X Y Z)
(AND X (AND Y (AND Z)))          => (AND X Y Z)
(OR X  (... (OR  X Y) ... ))     => (OR X (... Y ... ))
(AND X (... (AND X Y) ... ))     => (AND X (... Y ... ))
(OR X  (... (AND X Y) ... ))     => (OR X (...   ... ))
```

```
(AND X (... (OR  X Y) ... ))      => (AND X (...   ... ))
(AND X)                           => X
(OR X)                            => X
```

Debugging Tips for New Users of PCCTS

#146. A syntax error with quotation marks on separate lines means a problem with newline

```
line 1: syntax error at "
" missing ID
```

#147. Use the ANTLR –gd switch to debug via rule trace

#148. Use the ANTLR –gs switch to generate code with symbolic names for token tests

#149. How to track DLG results

If the pre-processor symbol DEBUG_LEXER is defined when DLGLexerBase is compiled it will include code to assist in debugging your lexer. To turn on the debug output:

```
previousDebugValue=lexer.debugLexer(newDebugValue);
```

A value of 1 enables the debug output while a value of 0 disables output.

#150. For complex problems use traceOption and traceGuessOption to control trace output

Switches and Options

#151. Use ANTLR –gx switch to suppress regeneration of the DLG code and recompilation of DLGLexer.cpp

It is possible to maintain separate grammar files for the DLG definitions of #tokens and the ANTLR definition of rules so that the DLG related code (and routines having dependence on tokens.h) need not be re-compiled just because of a change in a rule's action.

#152. Can't use an interactive scanner (ANTLR –gk option) with ANTLR infinite lookahead

#153. To make DLG case insensitive use the DLG –ci switch

The analyzer does not change the text, it just ignores case when matching it against the regular expressions.

#154. Use ANTLR option -glms to convert Microsoft file names like "..\foo.g" to "../foo.g" in generated files

#155. Use ANTLR option -treport *number* to locate alternatives using a lot of CPU time to resolve

It can be difficult to determine which alternatives are causing PCCTS to work hard to resolve an ambiguity. In some cases the ambiguity is successfully resolved after much CPU time so there is no message at all.

A rough measure of the amount of work being performed which is independent of the CPU speed and system load is the number of tnodes created. Using "-info t" gives information about the total number of tnodes created and the peak number of tnodes.

```
Tree Nodes:  peak 1300k  created 1416k  lost 0
```

It also puts in the generated C or C++ file the number of tnodes created for a rule (at the end of the rule). However this information is not sufficient to locate the alternatives within a rule which are causing the creation of tnodes.

Using:

```
antlr -treport 100000 ....
```

causes antlr to list on stdout any alternatives which require the creation of more than 100,000 tnodes, along with the lookahead sets for those alternatives.

#156. The ANTLR option -info (p - predicate, t - tnodes, m - monitor, f - follow set, o - orphans)

The ANTLR -info options may be combined and may appear in any order:

```
antlr -info ptm -CC -gt -mrhoist on mygrammar.g
```

Summary:

-info p	Extra predicate information in generated file.
-info t	Information about tnode usage appears at the end of each rule in the generated file and on stderr. A tnode is used only when analyzing *k*>1 grammars. Compare with -treport *number*.
-info m	Monitor progress: Prints name of each rule as it is started. Flushes output at start of each rule.
-info f	First/follow set information sent to stdout.
-info o	(letter o) Orphan rules (rules not referenced by any other rule).
-info 0	(digit zero) No-operation

Multiple Source Files

#157. To see how to place main() in a .cpp file rather than a grammar file (".g") see pccts./testcpp/8/main.cpp

```
#include "tokens.h"
#include "myParserClass.h"
#include "DLGLexer.h"
```

#158. How to put file scope information into the second file of a grammar with two .g files

If one did place a file scope action in the second file, ANTLR would interpret it as the fail action of the last rule appearing in the first grammar file.

To place file scope information in the second file #include the generated file in yet another file which has the file scope declarations.

Source Code Format

#159. To place the C right shift operator ">>" inside an action use "\>\>"

If you forget to do this you'll get the error message:

```
warning: Missing <<; found dangling >>
```

No special action is required for the shift left operator.

#160. One can continue a regular expression in a #token statement across lines (or use flex definitions)

One can continue an antlr #token definition across a line boundary, by using a backslash, but if your regular expressions are that long, it might be wiser to use flex, which allows the definition of elements which can be combined to create complex regular expressions.

#161. A #token without an action will attempt to swallow an action which immediately follows it - use ";"

This is a minor problem when the #token is created for use with attributes or ASTs nodes and has no regular expression:

```
#token  CastExpr
#token  SubscriptExpr
#token  ArgumentList
<<
... Code related to parsing
>>
```

You'll receive the message:

```
warning: action cannot be attached to a token name
        (...token name...); ignored
```

To solve this problem one is now allowed to end a #token definition with a ";":

```
#token  ArgumentList ;
```

#162. A grammar may contain multiple start rules. They aren't declared.

#163. Given `rule[A a,B b] > [X x]` the proto is `X rule(ASTBase* ast,int* sig,A a,B b)`

The argument "sig" is the status value returned when using parser exception handling.

If automatic generation of ASTs is not selected, exceptions are not in use, or there are no inheritance variables then the corresponding arguments are dropped from the argument list. Thus with ASTs disabled, no parser exception support, and neither upward nor downward inheritance variables the prototype of a rule would be:

```
void rule()
```

#164. To remake ANTLR after changes to the source code use `make -f makefile1`

The first problem with the standard makefile is that generic.h does not appear in the dependency lists. The second problem is that the rebuild of antlr.c from antlr.g and of scan.c from parser.dlg have been commented out so as to allow building ANTLR on a machine without ANTLR the first time when there are problems with zip restoring modification dates for files.

#165. ANTLR reports "... action buffer overflow ..."

There are several approaches:

Usually one can bypass this problem with several consecutive action blocks. Contributed by M.T. Richter (mtr@ottawa.com).

One can place the code in a separate file and use #include. Contributed by Dave Seidel.

One can add -DZZLEXBUFSIZE=*value* to the command line.

#166. Exception handling uses status codes and `switch` statements to unwind the stack rule by rule

#167. For tokens with complex internal structure add #token expressions to match frequent errors

Suppose one wants to match something like a floating point number, character literal, or string literal. These have a complex internal structure. It is possible to describe them exactly with DLG. But is it wise to do so ? Consider:

`'\ff'` for `'\xff'` or `"\mThe result is: "` for `"\nThe result is: "`

If DLG fails to tolerate small errors like the ones above the result could be dozens of error messages as it searches for the closing quotation mark or apostrophe.

One solution is to create additional #token definitions which recognize common errors and either generates an appropriate error message or return a special #token code such as "Bad_String_Const". This can be combined with a special #lexclass which scans (in a very tolerant manner) to the end of the construct and generates no additional errors. This is the approach used by John D. Mitchell (johnm@jGuru.com) in the recognizer for C character and string literals in Example #1.

Another approach is to try to scan to the end of the token in the most forgiving way possible and then to validate the token's syntax in the DLG action routine.

#168. See pccts/testcpp/2/test.g and testcpp/3/test.g for examples of how to integrate non-DLG lexers with PCCTS

The examples were written by Ariel Tamches (tamches@cs.wisc.edu).

#169. Ambiguity, full LL(*k*), and the linear approximation to LL(*k*)

It took me a while to understand in an intuitive way the difference between full LL(*k*) lookahead given by the ANTLR –k switch and the linear approximation given by the ANTLR –ck switch. Most of the time I run ANTLR with –k 1 and –ck 2. Because I didn't understand the linear approximation I didn't understand the warnings about ambiguity. I couldn't understand why ANTLR would complain about something which I thought was obviously parse-able with the lookahead available. I would try to make the messages go away totally, which was sometimes very hard. If I had understood the linear approximation I might have been able to fix them easily or at least have realized that there was no problem with the grammar, just with the limitations of the linear approximation.

I will restrict the discussion to the case of "–k 1" and "–ck 2".

Consider the following example:

```
rule1   : rule2a | rule2b | rule2c ;
rule2a  : A X | B Y | C Z ;
rule2b  : B X | B Z ;
rule2c  : C X ;
```

It should be clear that with the sentence being only two tokens this should be parseable with LL(2).

Instead, because k=1 and ck=2 ANTLR will produce the following messages:

```
/pccts120/bin/antlr -k 1 -gs -ck 2 -gh example.g
ANTLR parser generator   Version 1.20   1989-1994
example.g, line 23: warning: alts 1 and 2 of the rule itself
         ambiguous upon { B }, { X Z }
example.g, line 23: warning: alts 1 and 3 of the rule itself
         ambiguous upon { C }, { X }
```

The code generated resembles the following:

```
if      (LA(1)==A || LA(1)==B || LA(1)==C) &&
        (LA(2)==X || LA(2)==Y || LA(2)==Z) then rule2a()
else if (LA(1)==B) &&
        (LA(2)==X || LA(2)==Z) then rule2b()
else if (LA(1)==C) &&
        (LA(2)==X) then rule3a()
        ...
```

This might be called "product-of-sums". There is an "or" part for LA(1), an "or" part for LA(2), and they are combined using "and". To match, the first lookahead token must be in the first set and the second lookahead token must be in the second set. It doesn't matter that what one really wants is:

```
if      (LA(1)==A && LA(2)==X) ||
        (LA(1)==B && LA(2)==Y) ||
        (LA(1)==C && LA(2)==Z) then rule2a()
else if (LA(1)==B && LA(2)==X) ||
        (LA(1)==B && LA(2)==Z) then rule2b()
else if (LA(1)==C && LA(2)==X) then rule2c()
```

The problem is that each product involves one element from LA(1) and one from LA(2) and as the number of possible tokens increases the number of terms grows as N^2. With the linear approximation the number of terms grows (surprise) linearly in the number of tokens.

ANTLR won't do this with k=1 (it would for k=2). It will only do "product-of-sums". However, all is not lost — you simply add a few well chosen semantic predicates which you have computed using your LL(*k*>1), mobile, water-resistant, all purpose, guaranteed-for-a-lifetime, carbon based, analog computer.

The linear approximation selects for each branch of the "if" a set which may include *more* than what is wanted but never selects a *subset* of the correct lookahead sets. We simply insert a hand-coded version of the LL(2) computation. It's ugly, especially in this case, but it fixes the problem. In large grammars it may not be possible to run ANTLR with k=2, so this fixes a few rules which cause problems. The generated parser may run faster because it will have to evaluate fewer terms at execution time.

```
<<
int use_rule2a() {
  if ( LA(1)==A && LA(2)==X ) return 1;
  if ( LA(1)==B && LA(2)==Y ) return 1;
  if ( LA(1)==C && LA(2)==Z ) return 1;
  return 0;
}
>>
rule1   : <<use_rule2a()>>? rule2a | rule2b | rule2c ;
rule2a  : A X | B Y | C Z ;
rule2b  : B X | B Z ;
rule2c  : C X ;
```

Correction due to Monty Zukowski (jaz@jGuru.com)

The real cases I've coded have shorter code sequences in the semantic predicate. I coded this as a function to make it easier to read. Another reason to use a function (or macro) is to make it easier to read the generated code to determine when your semantic predicate is being hoisted too high. It's easy to find references to a function name with the editor — but difficult to locate a particular sequence of "LA(1)" and "LA(2)" tests. Predicate hoisting is a separate issue which is described in Item #138.

In some cases of reported ambiguity it is not necessary to add semantic predicates because no *valid* token sequence could get to the wrong rule. If the token sequence were invalid it would be detected by the grammar eventually, although perhaps not where one might wish. In other cases the only necessary action is a reordering of the ambiguous rules so that a more specific rule is tested first. The error messages still appear, but one can ignore them or place a trivial semantic predicate (i.e. <<1>>?) in front of the later rules. This makes ANTLR happy because it thinks you've added a semantic predicate which fixes things.

#170. Ambiguity, `#pragma`, and ANTLR -rl switch (Contributed by John Lilley jlilley@empathy.com)

This note predates -treport (Item #155) and ambiguity aid (Item #54 ff.). However, it is still worth reading.

A significant problem is the exhaustion of ANTLR's lookahead-analysis resources, which is often related to the use of `(...)*`. Without the `-rl` option, ANTLR may consume a large, perhaps unlimited, amount of memory attempting to create the lookahead token sets for productions involving `(...)*`. If `-rl` is specified, then when ANTLR reaches the specified limit, it will exit with the error message "Ran out of resources while attempting to analyze *something*". I always use the `-rl` option; setting it to 600000 is usually sufficient even for large grammars. With an `-rl` setting of 60000 ANTLR can consume about 40Mbytes of memory before aborting.

Note: With recent versions of PCCTS one can use -treport to get information on tree node usage for each rule.

An ambiguity problem is hard to get a handle on because there is little indication of the cause of the problem. If you get this sort of error start looking for:

a. An `(...)*` where the "..." might be empty.

b. Two adjacent `(...)*`, perhaps brought together by related rules, where both `(...)` might match the same thing.

c. Something like `{A} (...)* {A}`.

d. Recursion can also be a problem if the recursion effectively implements (...)*.

Of course, the pieces of this puzzle might be sitting in different rules that just happen to get slapped together in some production, so it can be incredibly frustrating to diagnose. Sometimes, there is a real ambiguity underneath the problem (like a repetition of a possibly empty construct). Other times, the pattern just exceeds ANTLR's abilities. When this happens, ANTLR tries to compute an infinite or exponential number of combinations.

The following is the type of thing that drives ANTLR crazy, although this is too simple to exhibit any resource problem (you'll just get an ambiguity warning). But if this pattern were a lot more complex, you'd get resource problems:

```
a : (b)+ c
b : { X | Y | Z }
c : Z
```

There are several approaches I've taken to counter this problem:

a. Use source code control. Any time you introduce a significant change to a large grammar check in a snapshot first. Then, if you get this error, back out the changes and apply them incrementally.

b. Use `#pragma approx`. Be very careful with this! It is a sledgehammer for difficult cases that seem to have no other solution. It is dangerous because it easily masks problems in the grammar. The documentation for `#pragma approx` is sketchy, but it seems to do the following things in situations where there is ambiguity and/or exponential analysis.

1. It performs a *linear analysis*: the leading set of tokens will be computed for each lookahead slot (LA(1), LA(2), etc.) but no "token set tree" will be calculated. This is really only important for $k>1$.

2. `#pragma approx (...)*` or `#pragma approx (...)+` tells ANTLR to favor another iteration of the `(...)` over whatever rules may follow.

3. `#pragma approx {...}` tells ANTLR to favor the optional stuff over skipping it and matching what follows.

4. `#pragma approx ( A | B )` tells ANTLR to favor *A* over *B*.

You should only use `#pragma approx` when you are certain what is causing the problem, and there is no more specific way (such as predicates) to fix it.

c. Use the ambiguity aid options to locate rules which are contributing to the ambiguity that is driving ANTLR out of resources (*Item c added for 2.22 by THM)*.

d. Selectively disable or remove rules until the problem goes away. The purpose of this is to identify rules that participate in the ambiguity that is driving ANTLR out of resources. If you remove or disable a rule and the problem goes away, then the rule has *something* to do with the problem, although it requires more thought to diagnose the exact ambiguity. I sometimes create a special token that is never matched and use it in the grammar for the purpose of removing a rule from any potentially ambiguous interaction with other rules. For example, to remove rule "a" from consideration (in a certain sense), change:

```
a : b c (d)* ;
```

to:

```
a : Unmatched b c (d)* ;
```

where `Unmatched` is a token that is never generated. This will point out if the leading set of "a" is conflicting with other rules and causing problems.

Here's an example of something that drove me crazy in the C++ grammar. A fully-qualified identifier can have template arguments: this declares "x" to be a specialization of "A":

```
A<3> x;
```

The relevant part of the grammar was "qualified id", which matches a series of A::B::C:: followed by an ID:

```
scope_override    //A::B::C::...
  id:ID
  (
    "<"
        template_arguments
    ">"
  |
    // empty
  )
```

However, this was getting confused with relational expressions such as:

```
(A<3) > x;
```

The first task was identifying this as the problem. ANTLR actually blew up in an entirely different place, one that followed this rule sequentially in some productions. It took some selective disabling and rearrangement of rules to identify this rule as a culprit.

When this finally was pinpointed as an ambiguity, I couldn't see how this was, because "A" was a type, and types can't participate in relational expressions. However, after some more searching I dug up this case:

```
operator A < 3 > x;
```

The problem is that the type identifier "A" can be used as the "optor" for an operator function specification, so in that context (which is semantically invalid but permitted by the grammar), a type followed by "<" can be either a template specifier or a relational expression. There were a whole family of cases, all involving "operator", which allowed type names to show up where types were not allowed (or so I thought). The final solution was twofold: First, a semantic predicate had to categorize an identifier as "templated". Second, `#pragma approx` was used to avoid the complex processing that was bogging ANTLR down:

```
scope_override    //A::B::C::...
  id:ID
  #pragma approx
  (
    <<isTemplatedName($id->getText())>>?
    "<"
       template_arguments
    ">"
  |
    // empty
  )
```

Note that, if the brute-force `#pragma approx` had been used, then some relational expressions would never be parsed. The combination of `#pragma approx` and the predicate has (hopefully) solved the problem correctly.

#171. What is the difference between "`(...)? <<...>>? x`" and "`(...)? => <<...>>? x`" ?

The first expression is a syntactic predicate followed by a semantic predicate. The syntactic predicate can perform arbitrary lookahead and backtracking before committing to the rule. However it won't encounter the semantic predicate until already committed to the rule — this makes the semantic predicate merely a validation predicate. Not a very useful semantic predicate.

The second expression is a "guarded semantic predicate" with a convenient notation for specifying the look-ahead context. The context expression is used to generate an "if" condition similar to that used to predict which rule to invoke. It isn't any more powerful than the grammar analysis implied by the values you've chosen for the ANTLR switches –k and –ck. It doesn't have any of the machinery of syntactic predicates and does *not* allow arbitrarily large lookahead. If the syntax predicate is *true* the semantic predicate is evaluated – if *true* the parse of the alternative continues, and if *false* the parse of that alternative is aborted. If the syntax predicate is *false* then the semantic predicate is ignored and the parse continues. A common misconception is that the parse of the alternative is rejected when the syntax predicate is false.

#172. Memory leaks and lost resources

Syntactic predicates use setjmp/longjmp. They cause leaks even with reference counted tokens. (Item #123).
Delete temporary attributes on rule failure and exceptions (Item #193).
Delete temporary ASTs on rule failure and exceptions (Item #95).
A rule that constructs an AST returns an AST even when its caller uses the "!" operator (Item #88).
(C++ mode) A rule which applies "!" to a terminal loses the token (Item #89) unless the ANTLR reference counting option is enabled.
(C mode) Define a zzd_ast() routine if you define a zzcr_ast() or zzmk_ast() (Item #200).

#173. Some ambiguities can be fixed by introduction of new #token numbers

For instance in C++ with a suitable definition of the class "C" one can write:

```
C a,b,c                          /* a1 */
a.func1(b);                      /* a2 */
a.func2()=c;                     /* a3 */
a = b;                           /* a4 */
a.operator =(b);                 /* a5 */
```

Statement a5 happens to place an "=" (or any of the usual C++ operators) in a token position where it can cause a lot of ambiguity in the lookahead set. One can solve this particular problem by creating a special #lexclass for things which follow "operator" with an entirely different token number for such operator strings — thereby avoiding the whole problem.

```
//
//  C++ operator sequences (somewhat simplified for these notes)
//
//  operator <type_name>
//  operator <special characters>
//
//  There must be at least one non-alphanumeric character between
//  "operator" and operator name - otherwise they would be run
//  together - ("operatorint" instead of "operator int")
//

#lexclass LEX_OPERATOR
#token  FILLER_C1                "[\ \t]*"
          <<skip();
            if( isalnum(ch) ) mode(START);
          >>
#token  OPERATOR_STRING          "[\+\-\*\/%\^\&\|\~\!\=\<\>]*"
                         <<mode(START);>>
#token  FILLER_C2                "\(\) | \[\] "
                         <<mode(START);return OPERATOR_STRING;>>
```

#174. Subclassing DLGInputStream

To subclass DLGInputStream requires DLGLexerBase which is dependent on ANTLRTokenType. However the values of the ANTLRTokenType may not be known at the time. The workaround is to define a dummy ANTLRTokenType in the header file for your subclass. Because enum definitions have only file scope, this should not cause a problem as long as all files which include your class use the same size int to represent the enum.

Changes From The Original 1.33 Which Are Not Part of Any Other Section

A complete log of all but the most trivial changes are distributed in the PCCTS kits. The most recent changes are in CHANGES_FROM_133.TXT. Changes prior to 1.33MR13 are in CHANGES_FROM_133_BEFORE_MR13.TXT. Finally, for those who don't have the endurance to read these files, there is a CHANGES_SUMMARY.TXT which contains only the most interesting parts of those files. The following items are based on this last file.

#175. Use `#first <<...>` to place references to precompiled header files at the beginning of generated files

#176. Use DLGLexerBase::reset() to reset the input stream when parsing the input stream multiple times.

#177. Error counters are: ANTLRParser::syntaxErrCount and DLGLexerBase::lexErrCount

These counters are *not* reset to zero by ANTLRParser::init().

#178. Use "`class MyParser : public MyBaseParser ... {`" to specify your own parser base class

The information following the ":" is copied to the class declaration for MyParser. The constructor of the base class (e.g. MyBaseParser) must have the same signature as ANTLRParser.

#179. Use `#FirstSetSymbol(`*symbol_name*`)` to generate symbol for first set of an alternative

The set can be tested using:

```
        if (set_el(token->getType(),symbol_name)) {...}
```

A follow set can be generated by placing the #FirstSetSymbol operator after the block of interest:

```
        rr : (rule_a | rule_b) #FirstSetSymbol(ab_follow) () ;
```

#180. Use `-preamble` and `-preamble_first` to insert macro code at the start of each rule or block

The antlr option -preamble causes ANTLR to insert the code "`BLOCK_PREAMBLE`" at the start of each rule and block.

The option -preamble_first inserts the code "`BLOCK_PREAMBLE_FIRST(PreambleFirst_`*number*`)`" where the symbol `PreambleFirst_`*number* is equivalent to the first set defined by the #FirstSetSymbol described above.

#181. Preprocessor option `ZZDEFER_FETCH` to defer token fetch for C++ mode

When the ANTLRParser class is built with the pre-processor option ZZDEFER_FETCH defined, the fetch of new tokens by consume() is deferred until LA(i) or LT(i) is called. This is useful in cases where there is a potential for a deadlock between a supplier of data which is waiting for the parser to ask for more data and the parser which is waiting for another token because of prefetching.

The defer fetch feature cannot be used with the standard tracein() and traceout() routines because they display information from a prefetched token. The function isDeferFetchEnabled() tests whether this feature is enabled.

#182. Exception handling

There were significant problems in the handling of exceptions in 1.33 vanilla. The general problem is that it can only process one level of exception handler. For example, a named exception handler, an exception handler for an alternative, or an exception for a subrule always went to the rule's exception handler if there was no "catch" which matched the exception.

Now the exception handlers properly nest. If an exception handler does not have a matching "catch" then the nextmost outer exception handler is checked for an appropriate "catch" clause, and so on until an exception handler with an appropriate "catch" is found.

(C Mode) LA/LATEXT and NLA/NLATEXT

#183. Do not use LA(*i*) or LATEXT(*i*) in the action routines of #token

To refer to the token code (in a #token action) of the token just recognized use NLA. NLA is an lvalue (can appear on the left hand side of an assignment statement). To refer to the text just recognized use zzlextext (the entire text) or NLATEXT. One can also use zzbegexpr/zzendexpr which refer to the last regular expression matched. The char array pointed to by zzlextext may be larger than the string pointed to by zzbegexpr and zzendexpr because it includes substrings accumulated through the use of zzmore().

#184. Care must be taken in using LA(*i*) and LATEXT(*i*) in interactive mode (ANTLR switch –gk)

In interactive mode ANTLR doesn't guarantee that it will fetch lookahead tokens until absolutely necessary. It is somewhat safer to refer to lookahead information in semantic predicates, but care is still required.

In this table the entries "prev" and "next" means that the item refers to the token which precedes (or follows) the action which generated the output. For semantic predicate entries think of the following rule:

```
rule : <<semantic-predicate>>? Next NextNext ;
```

For rule-action entries think of the following rule:

```
rule : Prev <<action>> Next NextNext;
```

	k=1	k=1	k=3	k=3	k=3
	standard	infinite	standard	interactive	infinite
for a semantic predicate					
LA(0)	Next	Next	--	--	--
LA(1)	Next	Next	Next	Next	Next
zzlextext	Next	Next	Next	--	Next
ZZINF_LA(0)		Next			Next
ZZINF_LA(1)		NextNext			NextNext
for a rule action					
LA(0)	Prev	Prev	--	Prev	--
LA(1)	Prev	Prev	Prev	Next	Prev
zzlextext	Prev	Prev	Prev	--	Prev
ZZINF_LA(0)		Prev			Prev
ZZINF_LA(1)		Next			Next

(C Mode) Execution-Time Routines

#185. Calls to zzskip() and zzmore() should appear only in #token actions (or in subroutines they call)

#186. Use ANTLRs or ANTLRf in line-oriented languages to control the prefetching of characters and tokens

Write your own input routine and then use ANTLRs (input supplied by string) or ANTLRf (input supplied by function) rather than plain ANTLR which is used in most of the examples.

#187. Saving and restoring parser state in order to parse other objects (input files)

Suppose one wants to parse files that "include" other files. The code in ANTLR (antlr.g) for handling #tokdefs statements demonstrates how this may be done:

```
grammar:  ...
        | "#tokdefs"  QuotedTerm
          <<{
          zzantlr_state           st;     /* defined in antlr.h  */
          struct zzdlg_state      dst;    /* defined in dlgdef.h */
          FILE                    *f;
          UserTokenDefsFile = mystrdup(LATEXT(1));
          zzsave_antlr_state(&st);
          zzsave_dlg_state(&dst);
          f = fopen(StripQuotes(LATEXT(1)),"r");
          if ( f==NULL ) {
              warn(eMsg1("cannot open token defs file '%s'",
                          LATEXT(1)+1));}
          else {
              ANTLRm( enum_file(), f, PARSE_ENUM_FILE);
              UserDefdTokens = 1;
          }
          zzrestore_antlr_state(&st);
          zzrestore_dlg_state(&dst);
          }>>
```

The code uses zzsave_antlr_state() and zzsave_dlg_state() to save the state of the current parse. The ANTLRm macro specifies a starting rule for ANTLR of "enum_file" and starts DLG in the PARSE_ENUM_FILE state rather than the default state (which is the current state — whatever it might be). Because enum_file() is called without any arguments it appears that enum_file() does not use ASTs nor pass back any attributes. Contributed by TJP.

(C Mode) Attributes

#188. Use symbolic tags (rather than numbers) to refer to attributes and ASTs in rules

prior to version 1.30:	`rule : X Y`	`<<printf("%s %s",$1,$2);>> ;`
sinc version 1.30:	`rule : xx:X yy:Y`	`<<printf("%s %s",$xx,$yy);>> ;`

#189. Rules no longer have attributes: `rule : r1:rule1 <<...$r1...;>>` won't work

Actually this still works if one restricts oneself to C mode and uses *numeric* labels like $1 and $2. However numeric labels are a deprecated feature, can't be used in C++ mode, and can't be used in the same source file as symbolic labels, so it's best to avoid them.

#190. Attributes are built automatically only for terminals

To construct attributes under any other circumstances one must use `$[`*TokenCode*`,...]` or zzcr_attr().

#191. How to access the text or token part of an attribute

The way to access the text, token, (or whatever) part of an attribute depends on the way the attribute is stored. If one uses the PCCTS supplied routine "pccts/h/charbuf.h" then:

```
id : "[a-z]+"           <<printf("Token is %s\n",$1.text);>> ;
```

If one uses the PCCTS supplied routine "pccts/h/charptr.c" and "pccts/h/charptr.h" then:

```
id : "[a-z]+"           <<printf("Token is %s\n",$1);>> ;
```

If one uses the PCCTS supplied routine "pccts/h/int.h" (which stores numbers only) then:

```
number : "[0-9]+"       <<printf ("Token is %d\n",$1);>> ;
```

(Note the use of "%d" rather than "%s" in the printf() format).

#192. The $0 and $$ constructs are no longer supported — use inheritance instead (Item #113)

#193. If you use attributes then define a zzd_attr() to release resources (memory) when an attribute is destroyed

#194. Don't pass automatically constructed attributes to an outer rule or sibling rule — they'll be out of scope

The PCCTS generated variables which contain automatically generated attributes go out of scope at the end of the rule or sub-rule that contains them. Of course you can copy the attribute to a variable that won't go out of scope. If the attribute contains a pointer which is copied (e.g. pccts/h/charptr.c) then extra caution is required because of the actions of zzd_attr(). See Item #195.

#195. A charptr.c attribute must be copied before being passed to a calling rule

The pccts/h/charptr.c routines use a pointer to a string. The string itself will go out of scope when the rule or sub-rule is exited. Why ? The string is copied to the heap when ANTLR calls the routine zzcr_attr() supplied by charptr.c — however ANTLR also calls the charptr.c supplied routine zzd_attr() (which frees the allocated string) as soon as the rule or sub-rule exits. The result is that in order to pass charptr.c strings to outer rules via inheritance it is necessary to make an independent copy of the string (using strdup for example) or else by zeroing the original pointer to prevent its deallocation.

#196. Attributes created in a rule should be assumed *not* valid on entry to a fail action

Fail action are "... executed after a syntax error is detected but before a message is printed and the attributes have been destroyed. However, attributes are not valid here because one does not know at what point the error occurred and which attributes even exist. Fail actions are often useful for cleaning up data structures or freeing memory." (Page 29 of 1.00 manual)

Example of a fail action:

```
            a : <<List *p=NULL;>>
                ( v:Var <<append(p,$v);>> )+
                    <<operateOn(p);rmlist(p);>>
              ; <<rmlist(p);>>
                ^^^^^^^^^^^^^^     <--- Fail Action
```

#197. Use a fail action to destroy temporary attributes when a rule fails

If you construct temporary, local, attributes in the middle of the recognition of a rule, remember to deallocate the structure should the rule fail. The code for failure goes after the ";" and before the next rule. For this reason it is sometimes desirable to defer some processing until the rule is recognized rather than the most convenient place:

```
#include "pccts/h/charptr.h"
;statement!
        : <<char *label=0;>>
          {name:ID COLON  <<label=MYstrdup($name);>> }
                s:statement_without_label
                        <<#0=#(#[T_statement,label],#s);
                          if (label!=0) free(label);
                        >>
;<<if (label !=0) free(label);>>
```

In the above example attributes are handled by charptr.* (see the warning, Item #195). The call to MYstrdup() is necessary because $name will go out of scope at the end of the subrule "{name:ID COLON}". The routine written to construct ASTs from attributes (invoked by `#[int,char *]`) knows about this behavior and always makes a copy of the character string when it constructs the AST. This makes the copy created by the explicit call to MyStrdup redundant once the AST has been constructed. If the call to "statement_without_label" fails then the temporary copy must be deallocated.

#198. When you need more information for a token than just token type, text, and line number

Passing accurate column information along with the token in C mode when using syntactic predicates requires workarounds. P.A. Keller (keller@ebi.ac.uk) has worked around this limitation of C mode by passing the address of a user-defined struct (rendered as text using format codes "%p" or "%x") along with (or instead) of the token's actual text. This requires changes in syntax error routines and other places where the token text might be displayed.

#199. About the pipeline between DLG and ANTLR (C mode)

I find it helpful to think of lexical processing by DLG as a process which fills a pipeline and of ANTLR as a process which empties a pipeline. (This relationship is exposed in C++ mode because of the ANTLRTokenBuffer class).

With LL_K=1 the pipeline is only one item deep, trivial, and invisible. It is invisible because one can make a decision in ANTLR to change the DLG #lexclass with zzmode() and have the next token (the one following the one just parsed by ANTLR) parsed according to the new #lexclass.

With LL_K>1 the pipeline is not invisible. DLG will put a number of tokens into the pipeline and ANTLR will analyze them in the same order. How many tokens are in the pipeline depends on options one has chosen.

Case 1: Infinite lookahead mode ("`(...)?`"). The pipeline is as huge as the input since the entire input is tokenized by DLG before ANTLR even begins analysis.

Case 2: Demand lookahead (interactive mode). There is a varying amount of lookahead depending on how much ANTLR thinks it needs to predict which rule to execute next. This may be zero tokens (or maybe it's one token) up to *k* tokens. Naturally, it takes extra work by ANTLR to keep track of how many tokens are in the pipe and how many are needed to parse the next rule.

Case 3: Normal mode. DLG stays exactly *k* tokens ahead of ANTLR. This is a half-truth. It rounds *k* up to the next power of 2 so that with k=3 it actually has a pipeline of 4 tokens. If one says "-k 3" the analysis is still *k*=3, but the pipeline size is rounded up because TJP decided it was faster to use a bit-wise "and" to compute the next position in a circular buffer rather than (n+1) mod k.

(C Mode) ASTs

#200. Define a zzd_ast() to recover resources when an AST is deleted

#201. How to place prototypes for routines using ASTs in the #header

Add #include "ast.h" after the #define AST_FIELDS and before any references to AST:

```
#define AST_FIELDS int token;char *text;
#include "ast.h"
#define zzcr_ast(ast,attr,tok,astText) \
        create_ast(ast,attr,tok,text)
void create_ast (AST *ast,Attr *attr,int tok,char *text);
```

#202. To free an AST tree use zzfree_ast() to recursively descend the AST tree and free all sub-trees

The user should supply a routine zzd_ast() to free any resources used by a single node — such as pointers to character strings allocated on the heap.

#203. Use #define zzAST_DOUBLE to add support for doubly linked ASTs

There is an option for doubly linked ASTs in the module ast.c. It is controlled by #define zzAST_DOUBLE. Even with zzAST_DOUBLE only the right and down fields are filled while the AST tree is constructed. Once the tree is constructed the user must call the routine zzdouble_link(tree,0,0) to traverse the tree and fill in the left and up fields.

Extended Examples and Short Descriptions of Distributed Source Code

Examples mentioned in these notes are available as .zip files at the sites mentioned in Item #1. In keeping with the restrictions in PCCTS, I have used neither templates nor multiple inheritance in these examples.

All these examples use AST classes which are derived from NoLeakAST. Some of them use tokens which are derived from NoLeakToken. These classes maintain a doubly-linked list of all ASTs (or tokens) which have been created but not yet deleted making it possible to recover memory for these objects.

#1. DLG definitions for C and C++ comments, character literals, and string literals

See files in notes/cstuff/cstr.g (C mode) or notes/cstuff/cppstr.g (C++ mode). Contributed by John D. Mitchell (johnm@jGuru.com).

#2. A simple floating point calculator implemented using PCCTS attributes and inheritance

This is the PCCTS equivalent of the approach used in the canonical yacc example. See notes/calctok/*.

#3. A simple floating point calculator implemented using PCCTS ASTs and C++ virtual functions

See files in notes/calcAST/*.

In this example an expression tree is built using ASTs. For each operator in the tree there is a different class derived from AST with an operator specific implementation of the virtual function "eval()". Evaluation of the expression is performed by calling eval() for the root node of the AST tree. Each node invokes eval() for its children nodes, computes its own operation, and passed the result to its parent in a recursive manner.

#4. An ANTLRToken class for variable length strings allocated from the heap

This is no longer useful since ANTLRCommonToken uses the heap.

#5. How to extend PCCTS C++ classes using the example of adding column information

See files in notes/col/*

This demonstrates how to add column information to tokens and to produce syntax error messages using this information. This example derives classes from ANTLRToken and ANTLRTokenBuffer.

#6. Use of parser exception handling in C and C++ programs

For C see notes/exc/extestc.g and test1.dat
For C++ see notes/excpp/extestcpp.g and test1.dat

#7. How to pass whitespace through DLG for pretty-printers

See files in notes/ws/*

This demonstrates how to combine several separate DLG tokens (whitespace for this example) into a single ANTLR token. It also demonstrates careful processing of tab characters to generate accurate column information even within comments or other constructs which use more().

#8. How to prepend a newline to the DLGInputStream via derivation from DLGLexer

See files in notes/prependnl/*

This demonstrates how to derive from DLGLexer in order to replace a user-supplied DLGInputStream routine with another which can perform additional processing on the input character stream before the characters are passed to DLG. In this case a single newline is prepended to the input. This is done to make it easier to treat the first non-blank token on a line as a special case, even when it appears on the very first line of the input file.

#9. How to maintain a stack of #lexclass modes

See files in notes/modestack/*

This is based on routines written by David Seidel which allow the user to pass a a routine to be executed when the mode is popped from the stack.

#10. When you want to change the token type just before passing the token to the parser

See files in notes/predbuf/*

This program shows how to reassign token codes to tokens at the time they are fetched by the parser by deriving from class ANTLRTokenBuffer and changing the behavior of getToken().

#11. Rewriting a grammar to remove left recursion and perform left factoring

The original grammar:

```
command := SET var BECOMES expr
                |  SET var BECOMES QUOTE QUOTE
                |  SET var BECOMES QUOTE expr QUOT
                |  SET var BECOMES QUOTE command QUOTE

expr    := QUOTE anyCharButQuote QUOTE
                |  expr AddOp expr
                |  expr MulOp expr
```

The repetition of "SET var BECOMES" for command would require k=4 to get to the interesting part. The first step is to left-factor "command":

```
command := SET var BECOMES
                ( expr
                | QUOTE QUOTE
                | QUOTE expr QUOTE
                | QUOTE command QUOTE
                )
```

The definition of expr uses left recursion which must be eliminated when using ANTLR:

```
op      := AddOp
        |  MulOp
expr    := QUOTE anyCharButQuote QUOTE (op expr)*
```

Since expr begins with QUOTE and all the alternatives of the sub-rule of command also start with QUOTE. This too can be left-factored:

```
 command := SET var BECOMES QUOTE
                ( expr_suffix
                | QUOTE
                | expr QUOTE
                | command QUOTE
                )
expr_suffix := anyCharButQuote QUOTE (op expr)*
expr        := QUOTE expr_suffix
```

The final grammar can be built by ANTLR with k=2.

```
#token Q          "\""
#token SVB        "svb"                     //  "SET var BECOMES"
#token Qbar       "[a-z A-Z]*"
#token AddOp      "\+"
#token MulOp      "\*"
#token WS         "\ "    <<skip();>>
#token NL         "\n"    <<skip();>>

repeat       : ( command )+ "@";
command      : SVB Q ( expr_suffix
                     | expr Q
                     | Q           <<printf("null command\n");>>
                     | command Q   <<printf("command\n");>>
                     );
expr_suffix : Qbar Q <<printf("The Qbar expr is (%s)\n",$1.text);>>
                 { op expr };
expr         : Q expr_suffix;
op           : AddOp | MulOp ;
```

#12. Processing counted strings in DLG

Sometimes literals are preceded by a count field.

```
        3abc identifier 4defg
```

This example works by storing the count which precedes the string in a local variable and then switching to a #lexclass which accepts characters one at a time while decrementing a counter. When the counter reaches zero (or a newline in this example) the DLG routine switches back to the usual #lexclass.

```
...
#lexaction <<static int count;>>

#token HOLLERITH        "[0-9]*"
        <<count=atoi(lextext());
         printf("Count is %d\n",count);
         mode(COUNT);
        >>
#token Eof              "@"
#token ID               "[a-z]*"  <<printf("ID is %s\n",lextext());>>
#token WS               "\ "      <<skip();>>
#token NL               "\n"
#lexclass COUNT
#token  STRING          "~[]"
        <<count--;
          if (count == 0) {
              mode(START);
              printf ("Hollerith string is \"%s\"\n",lextext());
          } else if (ch == '\n') {
              mode(START);
              printf("Hollerith string %s terminated by newline\n",
                lextext());
          } else {
              more();
          };
        >>
class P {
  statement      : ( (HOLLERITH STRING | ID )* NL)+ "@";
}
```

See files in notes/hollerith/*

#13. How to convert a failed validation predicate into a signal for treatment by parser exception handling

See files notes/sim/*

This program intercepts a failed validation predicate in order to pass a signal back up the call tree. The example includes code which takes the signal returned by the start rule and invokes the default handler

This example is not as clean as I would like because of the difficulty of adding new behavior to a parser class.

#14. How to use Vern Paxson's flex with PCCTS in C++ mode by inheritance from ANTLRTokenStream

See files example.flex and flexLexer.* in notes/flex/*

#15. Using the GNU gperf (generate perfect hashing function) with PCCTS

See files in notes/gperfex/*

The scanner generated by DLG can be very large. For grammars which contain a large number of keywords it might make sense to the use of the GNU program "gperf". The gperf programs attempts to generate a "minimal perfect hash function" for testing whether an argument is among a fixed set of strings such as those used in the reserved words of languages it has a large number of options to specify space/time trade-offs and the style of the code generated (e.g. C++ vs. C, case sensitivity, arrays vs. case statements, etc.).

As a test I found that a grammar with 25 keywords caused DLG to generate a file DLGLexer.cpp with 22,000 characters. Changing the lexical analysis code to use gperf resulted in a file DLGLexer.cpp that was 2,800 characters. The file generated by gperf was about 3,000 characters.

The gperf program was originally written by Douglas C. Schmidt. It is based on an algorithm developed by Keith Bostic. The gperf program is covered by the GNU General Public License. I do not know what restrictions there are on the output of gperf. The source code can be found in comp.sources.unix, volume 20.

Among the many FTP sites with comp.sources.unix here are two:

ftp.cis.ohio-state.edu/pub/comp.sources.unix/Volume20/gperf
ftp.informatik.tu-muenchen.de/pub/comp/usenet/comp.sources.unix/gperf

#16. Multiple files managed as a single token stream

See files in notes/nested/*

This example manages a file which may have an #include statement as a single stream of tokens.