PCCTS 1.20 — Release Notes

Terence J. Parr[*]
University of Minnesota
Army High-Performance Computing
Research Center
Minneapolis, MN 55415
parrt@acm.org

Russell W. Quong
School of Electrical Engineering
Purdue University
W. Lafayette, IN 47907
quong@ecn.purdue.edu

William E. Cohen
School of Electrical Engineering
Purdue University
W. Lafayette, IN 47907
cohenw@ecn.purdue.edu

August 20, 1994

Abstract

This document describes the 1.20 release of the Purdue Compiler Construction Tool Set (PCCTS). A number of new features have been added since 1.10 (August 1993), but the main addition is the introduction of C++ support. The C++ support will be described in a forthcoming paper—it is merely summarized here.

PCCTS is in the public-domain and can be obtained at `marvin.ecn.purdue.edu` in `pub/pccts/1.20`. You can join the `pccts-users` mailing list dealing with tools ANTLR, DLG (and SORCERER) by emailing `pccts-users-request@ahpcrc.umn.edu` with a body of `subscribe pccts-users` *your-name-or-ret-addr*.

The authors make no claims that this software will do what you want, that this manual is any good, or that the software actually works—use PCCTS at your own risk. Bug reports and/or cheery reports of its usefulness are very welcome, however.

[*]Partial support for this work has come from the Army Research Office contract number DAAL03-89-C-0038 with the Army High Performance Computing Research Center at the U of MN.

1 Introduction

The 1.20 release is primarily to introduce C++ support for PCCTS, but includes some nice overall enhancements and important bug fixes. The C++ support is only of $beta$ quality and, as such, can be expected to change in future releases.

We anticipate more frequent releases to PCCTS in the future rather than big, twice-a-year, releases to provide faster bug fixes and feature enhancements.

The main 1.20 features and enhancements are

- Added "`#tokclass`" (define a set of tokens), "`.`" (wildcard operator), "`~`" (not operator).
- Added "T_1`..`T_n" token range operator.
- Added "`#tokdefs`" so programmers can have predefined token type values.
- A new, improved `genmk` program.
- Line numbers are now tracked when using infinite lookahead.
- Added C++ support for ANTLR and DLG output.
- Added "`-o` dir" option to specify where all output should go.
- A few nasty code generation bugs were fixed as well as a few bugs relating to semantic predicate hoisting.

2 Token Classes

A token class is set of tokens that can be referenced as one entity; they are equivalent to a subrule consisting of its member tokens separated by "|"s. The basic syntax is:

`#tokclass` $tclass$ $\{$ $T_1..T_n$ $\}$

where T_i is a token reference (either a token label or a regular expression in double-quotes) or a token class reference; token classes may have overlapping tokens.

The difference between a token class and a subule lies in efficiency. A reference to a token class is a simple set membership test during parser execution rather than a linear search of the tokens in a subrule. Furthermore, the set membership will be much smaller than a series of `if`-statements in a recursive-descent parser. Note that automaton-based parsers (both $LALR$ and LL) automatically perform this type of set membership (specifically, a table lookup), but lack the flexibility of recursive-descent parsers such as those constructed by ANTLR.

Figure 1 is a sample ANTLR 1.20 program that recognizes a simple mythical assembly language with statements such as:

```
#segment data
a  ds 42
b  ds 13
#segment code
   load r1, a
   load r2, b
   add  r1,r2,r3
   print r3
```

Referencing token class `REGISTER` is the same as referencing

```
( "r0" | "r1" | "r2" | "r3" )
```

A wildcard token is also available that refers to an implied token class consisting of all tokens referenced within a grammar. It can be used to ignore pieces of the input:

```
#header <<#include "charbuf.h">>

<<main() { ANTLR(prog(), stdin); }>>

#tokclass OPCODE   { "add" "store" "load" "call" "ret" "print" }
#tokclass REGISTER { "r0" "r1" "r2" "r3" }

#token "[\ \t]+"    <<zzskip();>>
#token "\n"         <<zzskip(); zzline++;>>

prog:   "#segment" "data" (data)*
        "#segment" "code" (stat)*
    ;

stat:   OPCODE operands
    ;

operands
    :   ID
    |   REGISTER
    |   REGISTER "," NUM
    |   REGISTER "," REGISTER "," REGISTER
    ;

data:   ID "ds" NUM
    ;

#token NUM     "[0-9]+"
#token ID      "[a-zA-Z]+"
```

Figure 1: ANTLR Recognizer for Simple Assembly Language

```
ig  :   "begin{ignore}"
        .
        "end{ignore}"
    ;
```

which matches any single token between begin{ignore} and end{ignore}, or the wildcard can be used for error detection:

```
if  :   "if" expr "then" stat
    |   .       <<fprintf(stderr, "malformed if-statement");>>
    ;
```

The programmer should be careful not to do things like this:

```
ig  :   "begin{ignore}"
        ( . )*
        "end{ignore}"
    ;
```

because the loop generated for the `( . )*` block will never terminate—"end{ignore}" is also matched by the wildcard.

Rather than using the wildcard to match large token classes, it is often best to use the *not* operator. For example,

```
ig  :   "begin{ignore}"
        ( ~"end{ignore}" )*
        "end{ignore}"
    ;
```

where ~ is the *not* operator. The `if` example could be rewritten as:

```
if  :   "if" expr "then" stat
    |    ~"if"    <<fprintf(stderr, "malformed if-statement");>>
    ;
```

The *not* operator may be applied to token class references and token references only (it may not be applied to subrules, for example).

The wildcard operator and the *not* operator never result in a set containing the end-of-file token type.

One final token operator has been introduced—the range operator of the form T_1..T_n. The vaue of T_1 must be less than T_n and the values in between should be valid token types. In general, this feature should be used in conjunction with `#tokdefs` so that the programmer controls the token type values.

An example range operator is:

```
#tokdefs "mytokens.h"

a : OpStart .. OpEnd operand ;
```

This feature is perhaps unneeded due to the more powerful token class directive.

3 New Directive `#tokdefs`

It is often the case that the user is interested in specifying the token types rather than having ANTLR generate its own; typically, this situation arises when the user wants to link an ANTLR-generated parser with a non-DLG-based scanner. To get ANTLR to use pre-assigned token types, specify

#tokdefs "*mytokens.h*"

before any token definitions where *mytokens.h* is a file with only a list of `#defines` or an `enum` definition with optional comments.

When this directive is used, new token label definitions will not be allowed (either explicit definitions like `#token A` or implicit definitions such as a reference to a token label in a rule). However, the programmer may attach regular expressions and lexical actions to the token labels defined in $mytokens.h$. For example, if $mytokens.h$ contained:

```
#define A 2
```

and `t.g` contained:

```
#tokdefs "mytokens.h"

#token A "blah"

a : A B;
```

ANTLR would report the following error message:

```
Antlr parser generator   Version 1.20   1989-1994
t.g, line 5: error: implicit token definition not allowed with #tokdefs
```

4 New `genmk`

The `genmk` program has been substantially upgraded. It now generates `clean` and `scrub` targets, generates more accurate file dependencies and, most importantly, generates makefiles for PCCTS C++ mode. The command line options are defined as follows:

- `-CC`: Generate C++ output from both ANTLR and DLG.
- `-class` *cl*: Name of the grammar class defined in the grammar files. This is only a valid option if `-CC` was seen before it on the command line.
- `-dlg-class` *cl*: Name of DLG lexer class (default is `DLGLexer`). This is only a valid option if `-CC` was seen before it on the command line. The option is placed on the DLG command line as the `-cl` option.
- `-header` *h*: Name of the ANTLR standard header information (default=no file).
- `-o` *dir*: Directory where output files should go (default="."). This is very nice for keeping the source directory clear of ANTLR and DLG spawn.
- `-project` *proj*: Name of executable to create (default=`t`).
- `-token-types` *t*: Token types are in this file; this option implies that the normal `tokens.h` is not to be generated or used for token type definitions (in C output mode, however, `tokens.h` may still be required because we have jammed some function prototypes in the file as well).
- `-trees`: Generate ASTs; basically turns on the `-gt` option of ANTLR, but also results in a target to compile the AST support file (in C++ mode only).
- `-user-lexer`: Do not create a DLG-based scanner. Turns on ANTLR `-gx` command line option. Turns off generation of targets for DLG scanners.

For example, to create a makefile for a grammar in `test.g` with project name `t` that uses a DLG-based scanner, use:

```
genmk -project t test.g
```

To specify multiple files (comprising the same grammar) use:

```
genmk -project t test.g test2.g test3.g
```

To create a basic C++ mode makefile for a grammar class `Expr` in file `test.g` and project `t` use:

```
genmk -project t -CC -class Expr test.g
```

To rename the default `DLGLexer` class and (associated files) to `Scanner` for DLG, use:

```
genmk -project t -CC -class Expr -dlg-class Scanner test.g
```

To create a makefile for a C++ parser that uses a non-DLG-based scanner, use:

```
genmk -project t -CC -class Expr -user-lexer test.g
```

To create a makefile that uses a hand-built parser for a grammar that uses "`#tokdefs "mytokens.h"`", use:

```
genmk -project t -CC -class Expr -user-lexer -token-types mytokens.h test.g
```

5 Line Numbers and Infinite Lookahead

Because ANTLR-generated parsers that use infinite lookahead (i.e., syntactic predicates) scan the entire input stream before actual parsing begins, the normal line number variable `zzline` is meaningless—it always "points" to the last line in the file. To overcome this, we have added a new macro (member function `inf_line(int` i`)` in C++ mode):

`ZZINF_LINE(`i`)`

for $i \geq 1$. `ZZINF_LINE(1)` is the line number of the token about to be matched. The following example illustrates the use of the new macro:

```
#header <<#include "charbuf.h">>

<<main() { ANTLR(stat(), stdin); }>>

#token "[\ \t]+"         <<zzskip();>>
#token "\n"              <<zzskip(); zzline++;>>
#token Equal "="
#token Semi  ";"

stat:   (list Equal)? list Equal list ";"   <<printf("list = list\n");>>
    |   list ";"                            <<printf("list\n");>>
    ;

list:   "\(" elem ("," elem)* "\)"
    ;

elem:   <<int line=0;>>
        <<line = ZZINF_LINE(1);>> WORD <<printf("WORD at line %d\n", line);>>
    |   <<line = ZZINF_LINE(1);>> INT <<printf("INT at line %d\n", line);>>
    ;

#token WORD "[a-zA-Z]+"
#token INT  "[0-9]+"
```

The line number recording before the recognition of `WORD` and `INT` in `elem` is done because `ZZINF_LINE(1)` refers to the line number for the token about to be matched.

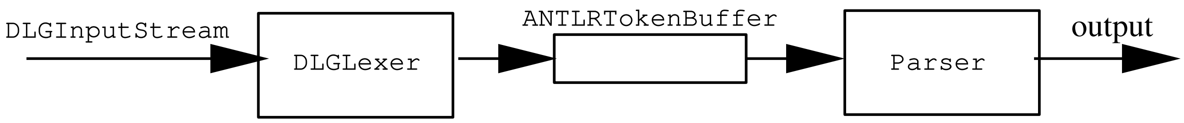

Figure 2: PCCTS C++ Parsing Block Diagram

This feature does not work correctly in C++ mode when `-gk` ANTLR option is used.

6 C++ Parser Organization

[*The C++ Output of PCCTS 1.20 is considered only "beta" quality. Expect a future release to require changes in most parsers written assuming version 1.20 C++ output.*]

[*Also note that the C++ output has only been tested under g++ 2.5.7 and cannot be guaranteed to compile under any other C++ compilers; indeed, g++ 2.4.5 will not compile our test cases.*]

Computer language recognition is generally viewed conceptually as a parser that parses tokens taken from a token stream filled by a lexical analyzer (scanner) that takes characters from an input stream. In practice, the token and character streams end up being merely function calls to an input routine while the separation between parser and scanner tends to become blurred.

For the C++ output of ANTLR and DLG, we have chosen to create a class hierarchy that reflects the nice conceptual separation between recognition subtasks. Figure 2 represents the class interactions that mirror the standard parsing block diagram. In C++ code, the block diagram is "constructed" by

1. Attaching an input stream to a DLG-based scanner,

   ```
   DLGFileInput in(stdin); /* create an input stream for DLG to get chars from */
   DLGLexer scan(&in,2000);/* create scanner reading from stdin w/bufsize==2000 */
   ```

2. Providing a token to DLG for it to continually fill in (the scanner is totally separated from the parser and, hence, has no idea how big the programmer's token is)[1]

   ```
   ANTLRToken aToken;
   scan.setToken(&aToken);
   ```

3. Attaching the token stream to a parser

   ```
   Parser p(&scan);
   ```

To start parsing, it is sufficient to call the `Parser` member function associated with the grammar rule:

`p.starting_rule(` *any_args*`);`

To specify the name of the parser class to construct, the programmer encloses all rules and actions in

```
class Parser {
```

[1]Note that if C++ allowed typenames to be passed around as objects, the scanner could simply be initialized with the correct typename. Also, since constructors cannot be virtual (polymorphic) a DLG-based token must be able to answer a message called `makeToken()`.

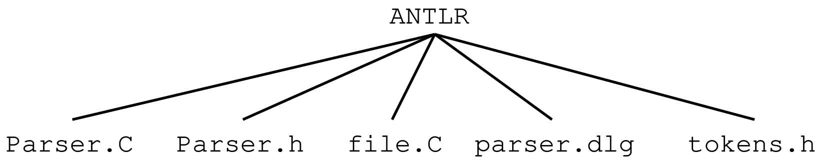

Figure 3: ANTLR-Generated Files in C++ Mode

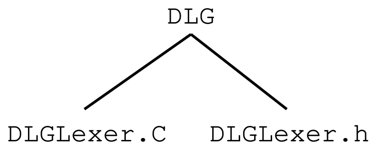

Figure 4: DLG-Generated Files in C++ Mode

```
    ...
}
```

Currently, exactly one class may be defined. For the defined class, ANTLR generates a derived class of `ANTLRParser` which accepts tokens (`ANTLRToken`) from a class derived from `ANTLRTokenBase`. This token stream can be either a user defined scanner or a DLG-based scanner. DLG generates a class (default name is `DLGLexer`) that is derived from `DLGLexerBase`. DLG-based scanners read input from derived classes of `DLGInputStream`, the most common being `DLGFileInput`.

Figures 3 and 4 describe the files constructed from ANTLR and DLG in C++ mode in the case where ANTLR was given a grammar called `file.g` containing "`class Parser {...}`" and DLG was given the usual `parser.dlg`. If the `#tokdefs` directive were used, then the `tokens.h` file would not be generated by ANTLR and if the ANTLR command line option for user-defined scanners (`-gx`) were used, `parser.dlg` would not be generated (`#tokdefs` and `-gx` operate independently of each other). Files `Parser.h` and `Parser.C` represent the class definition and support code for the output parser. The actual recursive-descent parser generated from `file.g` is placed in `file.C`; there may be multiple input files and for reasons of separate compilation, we do not put the parser functions into the `Parser.C` file. File `tokens.h` contains an enumerated type `TokenType` describing the set of defined token types. Files `DLGLexer.C` and `DLGLexer.h` embody the class definition of the scanner described by `parser.dlg`. The C++ support code for ANTLR and DLG output is not dependent on `#defines` as the C output is. For example, "`#define DEMAND_LOOK`" is not generated by ANTLR in C++ mode. The support code senses a flag in the parser structure that indicates whether to consume lookahead on demand or to continually fill the lookahead "pipe."

The communication medium between scanner and parser is an `ANTLRToken` (the name is fixed for the class name) where the communications channel is an `ANTLRTokenStream`. Tokens are no longer simply a token type; further, attributes, as defined by ANTLR 1.10, are no longer defined. For version 1.20, we combine the previous definitions to create an abstract token, `ANTLRToken` under the `ANTLRTokenBase` hierarchy, that contains at least the token type, but may include anything else required by the user. When using DLG-based scanners, `ANTLRToken` must be derived from `DLGBasedToken` which adds behavior needed by DLG (DLG-based scanners need to be able to set/get the text of a token). A common token definition, `ANTLRCommonToken`, is predefined to be the usual token type plus a fixed-size text buffer (similar to the old `Attrib` definition in `charbuf.h`). The minimal token definition is still the size of an integer (`sizeof(enum TokenType)`). To access the tokens in a token stream from the parser, `$`i variables are used. However, these variables are consistently pointers to type `ANTLRToken` in C++ mode whereas in C mode `$`i variables are of type `Attrib`.

The minimal programmer-supplied code requirements to get a C++ parser going are:

1. A function that constructs the various objects in the C++ parser block diagram in Figure 2 and invokes one of the parser member rules in your `Parser` class.

2. A definition for `ANTLRToken`. An easy way to accomplish this is to do the following:

```
typedef ANTLRCommonToken ANTLRToken; /* a token is token type and text */
```

3. A class definition in the grammar file(s).

The various definitions need not be placed in the `#header` action as in C output mode. The next section provides a complete example that illustrates all of the conventions described in this section.

6.1 A Simple C++ Example

Figure 5 is a simple example that illustrates a simple DLG-based scanner with an ANTLR grammar class. Assuming that the example is in a file called `test.g`, an executable parser may be obtained via the following command sequence:

```
antlr -CC test.g
dlg -CC parser.dlg
C++ -c -I/usr/local/pccts/include -o test.o test.C
C++ -c -I/usr/local/pccts/include -o Expr.o Expr.C
C++ -c -I/usr/local/pccts/include -o DLGLexer.o DLGLexer.C
C++ -o t test.o Expr.o DLGLexer.o \
         /usr/local/pccts/antlrx.o \
         /usr/local/pccts/dlgx.o \
```

where `antlrx.o` and `dlgx.o` are support code modules.

6.2 Grammar Classes

A grammar class is defined as follows

```
class Parser {
    actions
    rules
}
```

The actions may contain any normal C++ code that is valid within a C++ class definition[2]. For example,

```
class Parser {
<<public: int i;>>
<<int f() { blah; }>>

rule : A B ;

}
```

would result in a C++ class definition of:

[2]Actually, any normal ANTLR directive such as `#token` definitions may be placed inside, but it is best to separate them from the grammar class

```
<<
typedef ANTLRCommonToken ANTLRToken; /* user must define ANTLRToken */

/* "DLGLexer" must match DLG command line (-cl option); DLGLexer is default */
#include "DLGLexer.h"       /* include definition of DLGLexer.
main()
{
    DLGFileInput in(stdin); /* create an input stream for DLG to get chars from */
    DLGLexer scan(&in,2000);/* create scanner reading from stdin w/bufsize==2000
*/
    ANTLRToken aToken;      /* create a token for DLG to fill in */
    scan.setToken(&aToken);
    Expr parser(&scan);     /* create a parser of type Expr hooked to the scanner
*/

    parser.e();             /* start parsing at rule 'e' of that parser */
}
>>

#token "[\ \t\n]+"  <<skip();>>
#token Eof "@"

class Expr {                /* Define a grammar class */
e   :   IDENTIFIER NUMBER Eof
        <<printf("text is %s,%s\n", $1->getText(), $2->getText());>>
    ;
}

#token IDENTIFIER   "[a-z]+"
#token NUMBER       "[0-9]+"
```

Figure 5: Sample ANTLR Grammar for C++ Output

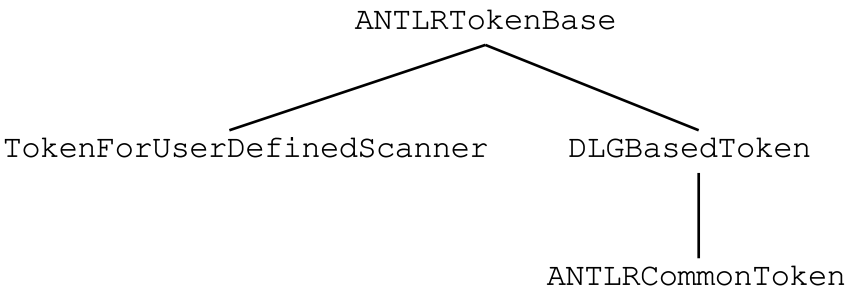

Figure 6: C++ Token Hierarchy

```
class Parser : public ANTLRParser {
protected:
    static ANTLRChar *_token_tbl[];
private:
    static SetWordType setwd1[4];
private:
public:
    Parser(ANTLRTokenStream *lexer);
    Parser(ANTLRTokenStream *lexer, TokenType eof);
    void rule();
public: int i;
int f() { blah; }
};
```

6.3 ANTLR Parser Classes

6.3.1 ANTLR Tokens

In C++ output mode, a token is an object that represents an abstraction of a lexical object found on the input stream by the scanner. An ANTLR-generated parser accepts input as sequence of tokens in the form of a token stream where the token stream is usually managed by a lexical analyzer (typically DLG). A token contains, minimally, a token type that is used by the parser to recognize grammatical structure. Optionally, the programmer can add fields specific to the token class to carry application-specific information. To facilitate error reporting, it is common practice to include a text string associate with each token and possibly the line number.

Figure 6 depicts the hierarchy of token base classes available to the programmer. Because an ANTLR parser must be able to make local copies of input tokens, the name of the programmer's token definition is fixed to be `ANTLRToken`; we do this for efficiency reasons as specifying a size to the parser and then asking it to allocate memory on the heap is much slower than defining a local object on the stack. The most basic token, `ANTLRTokenBase` knows how to set and get its token type and to get its text representation; the latter is needed to satisfy an ANTLR-generated parser's urge to print meaningful syntax error messages (because the text of a token is not required, this function returns the empty string unless redefined in a subclass). `ANTLRToken` must have a blank constructor:

```
ANTLRToken() ;
```

so that the parser can allocate local copies if required.

When using DLG-based scanners, additional behavior is required of a token. DLG must be able to set the text and token type for an `ANTLRToken`. A `virtual` function called

```
virtual void makeToken(TokenType t, ANTLRChar *s);
```

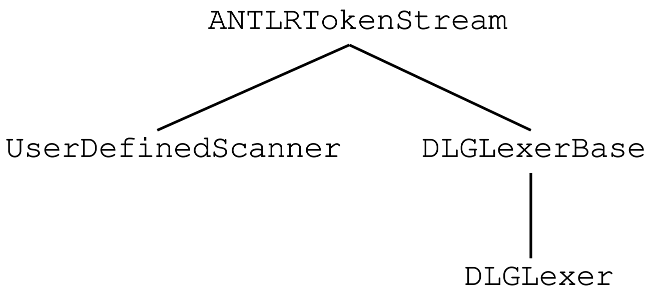

Figure 7: C++ Token Streams

is used because constructors cannot be `virtual` in C++ and we cannot pass the class to DLG so that it knows how to create and initialize an `ANTLRToken`. It is invoked in the following manner:

```
token_to_fill->makeToken(gettok(), _lextext);
```

where `token_to_fill` is a pointer to an `ANTLRToken`. This token, again, cannot be created by DLG because it's size is unknown to the DLG support code. It must be created by the programmer (normally just a local variable) and its address passed to the DLG-based scanner via member function `setToken()`.

The programmer defines an `ANTLRToken` class by deriving a class from `DLGBasedToken` (if DLG is being used—`ANTLRTokenBase` otherwise). For example, a common token is provided with PCCTS:

```
class ANTLRCommonToken : public DLGBasedToken {
protected:
    ANTLRChar _text[ANTLRCommonTokenTEXTSIZE+1];
public:
    ANTLRCommonToken(TokenType t, ANTLRChar *s) : DLGBasedToken(t,s)
            { setText(s); }
    ANTLRCommonToken() {;}
    ANTLRChar *getText() { return _text; }
    void setText(ANTLRChar *s) { strncpy(_text, s, ANTLRCommonTokenTEXTSIZE); }
};
```

Because the parser is attached to a DLG-based scanner, the parser has access to the current input line number; consequently, the line number is not stored in this token definition.

In C output mode, a type called `Attrib` is defined by the programmer and a required macro `zzcr_attr()` is invoked to set the attributes. In C++, these items correspond to `ANTLRToken` and `makeToken()`, respectively.

There is not a software stack of attributes or `ANTLRTokens` in C++ output mode—local token copies are local variables (on the hardware stack) for efficiency and ease of debugging.

6.3.2 ANTLR Token Streams

In C++ mode, an ANTLR-generated parser consumes `ANTLRTokens` from an `ANTLRTokenStream` which is an object that acts like a pipeline between the parser and lexical analyzer; the lexical analyzer, in turn, consumes characters from a text-based input stream. An `ANTLRTokenStream` supplies a stream of tokens by returning the "next" token for each call to `nextToken()` and also knows how to return the current text and line number of the current token. An `ANTLRTokenStream` can be anything from a simple array of `TokenTypes` to a full DLG-based scanner. Figure 7 depicts the token stream hierarchy.

To attach a non-DLG-based scanner to an ANTLR-generated parser, the programmer subclasses `ANTLRTokenStream`. For example,

```
ANTLRParser
     |
UserGrammarClass
```

Figure 8: C++ Parser Hierarchy

```
class MyTokenStream : public ANTLRTokenStream {
private:
   char c;
public:
   MyTokenStream() { c = getchar(); }
   ANTLRTokenBase *nextToken();
};
```

where `MyTokenStream::nextToken()` is some function that embodies a lexical analyzer; i.e., it breaks up the input stream into vocabulary symbols. `nextToken()` returns a pointer to an `ANTLRToken` that it has initialized.

6.3.3 ANTLR Parsers

All ANTLR-generated parsers in C++ mode are derived classes of `ANTLRParser`. No preprocessor symbols are used to define the structure of the parser as is done in C mode. In this way, the ANTLR-parser support code can be compiled separately and linked with different parsers. The support code senses object variables such as `demand_look` and `can_use_inf_look`. Figure 8 depicts the parser class hierarchy.

A few of the public interface functions are worth mentioning. To access the i^{th} token type of lookahead, use

```
inline TokenType LA(int i);
```

To access a pointer to the i^{th} `ANTLRToken` of lookahead, use

```
inline ANTLRTokenBase *LT(int i);
```

When using a non-DLG-based scanner, the user must inform the parser what token type should be considered end-of-input. This token type will then be used by the error recovery facilities to scan past bogus tokens without going beyond the end of input.

```
void setEofToken(TokenType t);
```

The programmer commonly wishes to modify the standard error reporting facility. To do so in C++ mode, simply subclass `Parser` and redefine `syn()` where `Parser` is your grammar class.

```
void syn(ANTLRTokenBase *tok, ANTLRChar *egroup,
         SetWordType *eset, TokenType etok, int k);
```

When `syn()` is redefined, so should `edecode()`:

```
void edecode(SetWordType *);
```

Upon catastrophic error, the following function is called (and should never return). The programmer can subclass `Parser` and redefine `ANTLRPanic`:

```
void ANTLRPanic(ANTLRChar *msg);
```

The following functions are the analog of the C mode infinite lookahead macros.

```
int inf_LA_valid(int i);
int inf_LA(int i);
inline inf_line(int i);
```

The following `protected` member functions can also be redefined:

```
virtual void tracein(ANTLRChar *r);
virtual void traceout(ANTLRChar *r);
```

The various ANTLR parser class definitions could have benefited greatly from templates and multiple inheritance, but no current C++ compiler implements these reliably; i.e., because of features like this C++ is totally nonportable due to compiler limitations.

6.4 AST Classes

To use ASTs with ANTLR in C++ mode, the user simply derives a class from either `ASTBase` or `ASTDoublyLinkedBase` and adds the desired fields. The AST classes automatically know how to do a preorder traversal of an AST, the programmer should redefine

```
virtual void preorder_action() { ; }
virtual void preorder_before_action() { printf(" ("); }
virtual void preorder_after_action() { printf(" )"); }
```

in their derived class if required. Figure 9 provides a simple example of how to use ASTs.

The `zzmk_ast()` and `zzcr_ast()` functions are now embodied by the AST class constructor.

#-variables are used as before and are pointers to the node(s) created by token and rule references.

6.5 DLG Classes

DLG generates a class (default name is `DLGLexer`) that is derived from `DLGLexerBase`. DLG-based scanners read input from derived classes of `DLGInputStream`, the most common being `DLGFileInput`. A number of functions can be redefined in derived classes; the interesting ones are:

```
virtual void erraction();
void    DLGPanic(DLGChar *msg);
virtual ANTLRTokenBase *nextToken();
```

The `nextToken()` function can be redefined in a subclass so that it knows what an `ANTLRToken` looks like. The standard `nextToken()` does not have this luxury and, hence, the user is required to provide DLG-based scanners with an `ANTLRToken` to fill in.

The function `trackColumns()` can be called to turn on column tracking. This is analogous to setting preprocessor symbol `ZZCOL` in C mode.

6.6 ANTLR Parsers and Hand-Built Scanners

Because of the clean separation of parsing subtasks followed by ANTLR, it is a trivial matter to link in a hand-built scanner or any non-DLG-based scanner. Simply turn on the `-gx` ANTLR command line option, which turns off the generation of DLG input, and attach an instance of your scanner to an instance of the parser class generated by ANTLR. Figures 10, 11, and 12 represent a complete example. The call to `setEofToken()` is done to inform ANTLR what token type is considered end of input; this is necessary for all hand-built parsers so that ANTLR does not try to resynchronize, after a syntax error, beyond the end of input.

6.7 New Supplied Files

- `antlrx.h`: All ANTLR parser support classes and the `ANTLRParser` class itself.
- `antlrx.C`: ANTLR parser support code.

```
<<
typedef ANTLRCommonToken ANTLRToken;
#include "DLGLexer.h"

class AST : public ASTBase {
public:
    ANTLRToken token;                    /* this is what I want inside */
    AST(ANTLRToken *t) { token = *t; }  /* how to make an AST node */
    void preorder_action() {
        char *s = token.getText();
        printf(" %s", s);
    }
};

main()
{
    ANTLRToken aToken;
    DLGFileInput in(stdin);
    DLGLexer scan(&in,2000);
    scan.setToken(&aToken);
    Expr parser(&scan);

    ASTBase *root = NULL;   // make a pointer to fill in
    parser.e(&root);        // don't forget to pass the ptr to the rule

    root->preorder();       // do a preorder traversal
    printf("\n");
}
>>

#token "[\ \t\n]+"  <<skip();>>
#token Eof "@"

class Expr {                /* Define a grammar class */

e   :   IDENTIFIER^n Eof!
        <<printf("text is %s,%s\n", #1->token.getText(),#2->token.getText());>>
    ;

n   :   NUMBER
    ;

}

#token IDENTIFIER   "[a-z]+"
#token NUMBER       "[0-9]+"
```

Figure 9: ANTLR Grammar Using ASTs

```
<<
typedef ANTLRCommonToken ANTLRToken;
#include "MyTokenStream.h"
main()
{
    MyTokenStream scan; /* create one of my scanners */
    Expr parser(&scan); /* create a parser of type Expr hooked to my scanner */
    parser.setEofToken(Eof);
    parser.e();
}
>>

class Expr {
e   :   IDENTIFIER NUMBER Eof
    ;
}
```

Figure 10: ANTLR Example That Uses Hand-Built Scanner

```
#include "ATokenStream.h"
class MyTokenStream : public ANTLRTokenStream {
private:
   char c;
public:
   MyTokenStream() { c = getchar(); }
   ANTLRTokenBase *nextToken();
};
```

Figure 11: Hand-Built Scanner Class Definition

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <iostream.h>
#include "tokens.h" /* include token defs */
#include "antlrx.h" /* include all the ANTLR yuck */
#include "MyTokenStream.h"
typedef ANTLRCommonToken ANTLRToken;

/* Recognizes Tokens IDENTIFIER and NUMBER */
ANTLRTokenBase *MyTokenStream::nextToken()
{
    static ANTLRToken resultToken; /* we will return a pointer to this guy */
    ANTLRChar TokenBuffer[100];
    int index=0;

    while ( c==' ' || c=='\n' ) c=getchar();

    if (c==EOF) {resultToken.setType(Eof); return &resultToken;}

    if (isdigit(c)) {
        while (isdigit(c)) ] {
            TokenBuffer[index++]=c;
            c = getchar();
        }
        TokenBuffer[index]='\0';
        resultToken.makeToken(NUMBER, TokenBuffer);
        return &resultToken;
    }

    if (isalpha(c)) {
        while (isalpha(c)) {
            TokenBuffer[index++]=c;
            c = getchar();
        }
        TokenBuffer[index]='\0';
        resultToken.makeToken(IDENTIFIER, TokenBuffer);
        return &resultToken;
    }
}
```

Figure 12: Hand-Built Scanner `nextToken()` Function

- `dlgx.h`: DLG scanner support classes and `DLGLexerBase` class.
- `dlgx.C`: DLG scanner support code.
- `astx.h`: AST class definition.
- `astx.C`: AST support code.
- `DLexer.C`: Support code that must be aware of the particular scanner generated by DLG. This is an ugly mechanism and will change in future versions.
- `AToken.h`: Definitions for classes `ANTLRTokenBase`, `DLGBasedToken`, and `ANTLRCommonToken`.
- `ATokenStream.h`: Definition of class `ANTLRTokenStream`.

6.8 $-Variables in C++ Mode

Because attributes do not exist in C++ mode, $-variables point to `ANTLRTokens`. Further, $-variables do not exist for rule references. Rule arguments and return values should be used instead. We anticipate the removal of $-variables all together in future releases in favor of labels for rule elements such as in the tree-parser generator SORCERER.

$-variables are pointers to `ANTLRTokens` exclusively in C++ mode.

6.9 Semantic Predicates

Semantic predicates should reference `LT(`*i*`)->getText()` instead of `LATEXT(`*i*`)` as `LATEXT` does not exist.

6.10 Converting a 1.10 Grammar to 1.20 C++ Grammar

This section describes the procedure we used to convert the PCCTS 1.10 C grammar from C to C++ mode (this covers most, but not all issues); "your mileage may vary."

- Remove `#parser` directive if any.
- `typedef ANTLRToken` to something or derive a class. Remove `#include "charbuf.h"` or other previous `Attrib` definition.
- Add a grammar class "wrapper" around your grammar rules.
- Convert AST stuff: (1) make a `class AST` definition. (2) Add fields in `AST_FIELDS` macro to the `class AST` definition as fields. (3) Remove `zzcr_ast` and make that the constructor. (4) Remove `zzmk_ast` if you have it and make it another constructor (no need to convert any `#[args]` references in the grammar).
- Add a `#include "DLGLexer.h"` in the parser (or whatever you call the lexical analyzer class).
- Convert all `$`$i.blah$ to `$`$i- > blah$.
- Convert all `LATEXT(`*i*`)` to `LT(`*i*`)->getText()`.
- Definitions in `#header` can come afterwards (i.e., do not use the directive anymore).
- Convert all `$0`=$blah$, `$$`=$blah$, or `$`$rule$=$blah$ to a return value or "by-reference" argument.
- Convert the `ANTLR()` macro reference into the series of object definitions outline in this document.

7 Semantic Predicate Hoisting

The hoisting of predicates in version 1.10 had a number of bugs that have been fixed. In addition, version 1.20 has changed the semantics of semantic predicate hoisting slightly to gain a useful feature. We begin by describing the bug fix.

The following grammar now behaves as advertised:

```
a : <<p1>>? b
  | ID
  ;
b : <<p2>>? ID
  | <<p3>>? ID
  ;
```

It results in the following code for `a`:

```
void a(void)
{
    ...
    if ( (LA(1)==ID)&&((p1)&&((p2)||(p3))) ) {
        if (!(p1)) zzfailed_pred((ANTLRChar *)"  p1"); //unused
        b();
    }
    else {
        if ( (LA(1)==ID) ) {
            zzmatch(ID); zzCONSUME;
        }
        else ...
}
```

Note that the following semantics indicate the correct semantic validity of production one of `a`: "`p1` and (`p2` or `p3`)." In version 1.10, the "or" was an "and."

In 1.10, we indicated that predicates were hoisting ONLY if the grammar was syntactically ambiguous. This had the unfortunate effect of making it impossible to include a predicate in the loop decision for `(..)+` and `(..)*` subrules if only one alternative was present. We have changed the meaning of semantic predicates slightly so that if only one alternative exists in a looping subrule, all visible predicates are ALWAYS hoisting. For example,

```
a : <<int i=5;>>            // match exactly 5 A's
    ( <<i>0>>? A <<i--;>> )+
  ;
```

In 1.10, this would have resulting in a loop that only tested the lookahead. In 1.20, the following is generated for the `(..)+` loop:

```
if (((i>0))) {
    do {
        if (!(i>0)) zzfailed_pred((ANTLRChar *)"  i>0"); // unused
        zzLOOP(zztasp2);
    } while ( (LA(1)==A)&&((i>0)) );
}
```

8 C AST Changes

In C mode, the programmer can define the preprocessor symbol `USER_DEFINED_AST` which allows the programmer to define the `AST` type themselves. The macro `AST_REQUIRED_FIELDS` is the minimum set of AST fields needed by ANTLR; as such, it functions like inheritance in C++. For example,

```
typedef struct _ast {
            AST_REQUIRED_FIELDS; // order is unimportant
            my stuff;
        } AST;
```

The structure name must be `_ast` for the `AST_REQUIRED_FIELDS` to work. If it is not used, the structure name can be anything.

9 New Command-Line Options

The following ANTLR command line options are new:

- `-CC`: Generate C++ output.
- `-o`: Directory where output files should go (default="."). This is very nice for keeping the source directory clear of ANTLR and DLG spawn.
- `-ct`: Do not make copies of tokens passed to the parser in C++ mode (default=to copy). When using DLG in conjunction with ANTLR, you will always want ANTLR to make copies because DLG only has space for one `ANTLRToken` (which is passed to the scanner with `setToken`); this address is always returned and, hence, without copies, all $-variables would point to the same `ANTLRToken`.

The `-ai` option has been removed in anticipation of an ANTLR graphical interface.

The following DLG command line options are new:

- `-CC`: Generate C++ output.
- `-o`: Directory where output files should go (default="."). This is very nice for keeping the source directory clear of ANTLR and DLG spawn.
- `-cl` *class*: Specify a class name for DLG to generate. The default is `DLGLexer`. *class* will be a subclass of `DLGLexerBase`.

Also note that in C++ mode, DLG now does not accept the output file name on the command line. The class name specified (or the default of `DLGLexer`) is used to derive the output file name:

```
dlg options parser.dlg
```

10 Miscellaneous New Additions

The following minor changes were made:

- A new character type is used for ANTLR and DLG `ANTLRChar` and `DLGChar` respectively in C++ mode. Normally these are 'char', but you can change the `typedef` to whatever you wish (you can even make it a class).
- ASTs were incorrectly handled in conjunction with 1.10 syntactic predicates—this has been fixed.
- The return values of rules were still assigned in guess mode (when using syntactic predicates); the arguments are still evaluated. This is perhaps not too bright.
- Warnings about missing `#header` now require that `-w2` ANTLR option be set.
- Allows `#parser` to come first (before `#header`)
- Fixed a bug so that "`(A B)+ A C`" will now terminate the loop upon "`A C`" whereas before it would just loop forever (it was not using enough lookahead).

- When a token label (for which there is no regular expression) is referenced in a rule, a warning is generated (if the ANTLR command line option `-w2` is specified).
- Fixed a nasty bug that caused ANTLR to loop forever (and a day) upon very large grammars with lots of optional subrules.
- ANTLR itself tends to give better error messages now; e.g., lexical errors give the file now and grammatical errors (employing `#errclasses`) are more readable.

11 Future

Our work on ANTLR continues to be heavily influenced by the feedback from our industrial and academic user community. As such, we are currently developing or planning the following improvements and tools.

- Good error recovery and reporting is notoriously difficult to achieve with parser generators, especially $LALR$-based tools. We are developing a sophisticated error handling mechanism analogous to C++ exception handling called *parser exception handling* that approaches the flexibility of hand-built parsers.
- The recognition strength of hand-built parsers arises from the fact that arbitrarily-complex expressions can be used to distinguish between alternative productions. We will introduce a new type of predicate called a *prediction predicate* that constitutes the entire prediction expression for a particular production; i.e., ANTLR does not generate code to test lookahead for the associated production. We anticipate the notation: "`<<`*this-is-the-entire-prediction-expression*`>>?!`".
- A graphical user interface is planned and a coder has been tentatively "pressed" into service. This "GUI" will display syntax diagrams on the screen and, hence, ambiguities in the grammar can be highlighted. The output of the GUI will be an ANTLR grammar or a PostScript representation of the syntax diagram.
- We intend to yank the infinite lookahead mechanism out of `ANTLRParser` and put it in `ANTLRTokenStream` where it belongs.

[*The users of PCCTS should be forewarned that we anticipate a break with total backward compatibility for a future release (perhaps PCCTS 2.00). This release is intended to fix the odious C output generated by the current version of ANTLR/DLG and will result in a modified grammar meta-language plus the removal of some parsing modes. Any book on PCCTS to be written will describe this version of reality. Also remember that the C++ output is going to change as we learn more about it.*]

12 Acknowledgements

Thanks are due to Sumana Srinivasan, Mike Monegan, and Steve Naroff of NeXT, Inc. for their extensive help in the initial definition of the ANTLR C++ parser. They are also instrumental in the ongoing design of parser exception handling.

We thank Gary Funck at Intrepid for his extensive testing of ANTLR and DLG plus his constant stream of excellent suggestions.

Steve Robenalt at Rockwell is single-handedly pushing the `comp.compilers.pccts` news group through, is helping with the workshop, and is porting PCCTS to a number of different platforms.

We thank Tom Moog (`moog@polhode.com`) for his fantastic `NOTES.newbie` information.

Ariel Tamches (`tamches@cs.wisc.edu`) deserves credit for spending a week of his Christmas vacation in the wilds of Minnesota helping me with the C++ output; he developed the majority of the code for the hand-built scanner C++ example.

The C++ output was also influenced by Thom Wood (`twood@tcis3.tcis.com`) and Randy Helzerman `helz@ecn.purdue.edu`.

Anthony Green at Visible Decisions, John Hall at Worcester Polytechnic Institute, Devin Hooker at Ellery Systems, Kenneth D. Weinert at Information Handling Services, and Roy Levow at Florida Atlantic University helped beta test 1.20.

Sriram Sankar at Sun Microsystems has help debug a number of features including the infinite lookahead line number tracking and has provided a fix to make DLG `char`-size independent, which we hope to include soon.

John Hall (`jhall@ivy.wpi.edu`) ported PCCTS to Visual C++.

We would also like to thank the multitude of other users of PCCTS for their excellent suggestions and beta-testing of the new C++ parsers.