PCCTS 1.21 — 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`

[*This author list only includes active participates for 1.21]*

August 20, 1994

Abstract

This document describes the 1.21 release of the Purdue Compiler Construction Tool Set (PCCTS). Aside from a few bug fixes, this release merely cleans up the C++ output—the C++ interface has been changed slightly since 1.20 (March 31, 1994). The original 1.00 manual and all release notes are required for a complete documentation set for PCCTS. A book is in the works and the papers provided at the `ftp` site don't hurt.

PCCTS is in the public-domain and can be obtained at `marvin.ecn.purdue.edu` in `pub/pccts/1.21`. The newsgroup `comp.compilers.tools.pccts` provides a discussion forum. Alternatively, 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*. To receive future release broadcast messages, register yourself by sending email to `pccts@ecn.purdue.edu` with a "`Subject:`" line of "`register`".

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.

From the 1.21 release forward, the maintenance and support of all PCCTS tools will be primarily provided by Parr Research Corporation, Minneapolis MN—an organization founded on the principles of excellence in research and integrity in business; we are devoted to providing really cool software tools. Please see file `PCCTS.FUTURE` for more information. All PCCTS tools currently in the public domain will continue to be in the public domain.

[*]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.

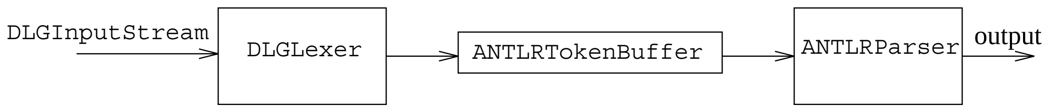

Figure 1: Overview of the C++ classes generated by ANTLR.

1 Introduction

The PCCTS 1.21 release is mainly an upgrade for the C++ output. The test examples have now been successfully compiled under a number of C++ compilers. Further, C++ mode may now use arbitrary lookahead with a "sliding window" of lookahead rather than reading in the entire input file before commencement of parsing. Any grammars written with 1.20 C++ output should be easy to convert to 1.21. The C++ interface section provides the new parser definition and invocation sequence. The C++ output is still considered alpha quality, but within a release or two, it should be up to "parr".

A number of bug fixes have been done and a very nice configuration file has been produced to aid in porting PCCTS.

Scott Haney at Lawrence Livermore National Labs has done a fabulous port of PCCTS 1.21 to the Macintosh (primarily MPW).

2 Configuration File

A new file, `config.h`, is provided to make porting ANTLR, DLG, and SORCERER easier. This file defines file names, standard symbols such as `__USE_PROTOS` and `CPP_FILE_SUFFIX`, directory characters, and various things for standard ports such as MPW.

The file `support/set.h` need no longer be modified for 16 bit compilers.

Define preprocessor symbol `PC` to enable a bunch of PC stuff like `.obj` for object files and "
" for the directory symbol.

Define preprocessor symbol `MPW` to enable a bunch of Mac stuff.

3 C++ Interface

[Warning: The C++ output is still in a state of change as we learn more about what it should look like. This should stabilize soon].

When generating recursive-descent parsers in C++, ANTLR creates separate C++ classes for the input stream, the lexical analyzer (scanner), the token buffer, and the parser. Conceptually, these these classes fit together as shown in Figure 1, and in fact, the ANTLR-generated classes "snap together" in an identical fashion. To initialize the parser, the programmer simply

1. attaches an input stream object to a DLG-based scanner[1],
2. attaches a scanner to a token buffer object, and
3. attaches the token buffer to a parser object generated by ANTLR.

The following code illustrates, for a parser object `Expr`, how these classes fit together.

[1]If the user has constructed their own scanner, they would attach it here.

```
main()
{
    DLGFileInput in(stdin);           // get an input stream for DLG
    DLGLexer scan(&in,2000);          // connect a scanner to an input stream
    ANTLRTokenBuffer pipe(&scan, k);  // connect scanner and parser via ``pipe''
    ANTLRToken aToken;
    scan.setToken(&aToken);           // give DLG access to a virtual table
    Expr parser(&pipe);               // make a parser connected to the pipe
    parser.init();                    // initialize the parser
    parser.e();                       // begin parsing;  e = start symbol
}
```

where `ANTLRToken` is programmer-defined and must be a subclass of `ANTLRAbstractToken` (or one of the predefined classes below it). To start parsing, it is sufficient to call the `Expr` member function associated with the grammar rule; here, `e` is the start symbol.

To specify the name of the parser class in an ANTLR grammar description, enclose the appropriate rules and actions in a C++ class definition, as follows.

```
class Expr {
<<int i;>>
<<
public:
    void print();
>>
e   :   INT ("\*" INT)* ;
    ⋮                 // other grammar rules
}
```

Thus, a parser object is simply a set of actions and routines for matching a rule. Consequently, it is natural to have many separate parser objects. For example, if parsing C code, we might have different parser classes for C expressions, for C function definitions, and for assembly code. Parsing multiple languages or parts of languages simply involves switching parsers objects. For example, assume you have a working C language front-end. To evaluate C expressions in a debugger, just use the parser object for C expressions (assuming the semantic actions were flexible enough).

Currently, ANTLR only allows one class definition per grammar. This will change in future versions when we figure out how grammar class inheritance should work.

To ensure compatibility among different input streams, lexers, token buffers, and parsers, all objects are derived from one of the four common bases classes `DLGInputStream`, `DLGLexer` (or `ANTLRTokenStream` if you roll your own lexer), `ANTLRTokenBuffer` or `ANTLRParser`.

Please see the C++ sample files collected in the `testcpp.tar` file.

3.1 C++ Token Definitions

To increase flexibility, the token class hierarchy has changed since release 1.20; see Figure 2. We did this mainly so that the minimal token is simply an `int`. Some programmers have existing scanners which cannot be modified. We encountered one such scanner that defined tokens to be integers; hence, a mandatory virtual table pointer inside each token object would render ANTLR and their system incompatible.

The classes are described as follows:

- An `ANTLRAbstractToken` cannot be instantiated and can be subclassed for truly unusual token definitions; this requires changes to the functions in `ANTLRParser` which obtain the token type from a token object such as `ANTLRParser::LT()`.
- If a programmer wants a token object that merely holds a token type and does not contain a virtual table, they may subclass `ANTLRLightweightToken`. However, because ANTLR wants to know the text and line number associated with a token for error reporting purposes, `ANTLRParser` must be subclassed to redefine the error handling routines such as `ANTLRParser::syn()`.

$$\begin{tikzcd}
\texttt{ANTLRAbstractToken} \arrow[d] \\
\texttt{ANTLRLightweightToken} \arrow[d] \\
\texttt{ANTLRTokenBase} \arrow[d] \\
\texttt{DLGBasedToken} \arrow[d] \\
\texttt{ANTLRCommonToken} \arrow[d] \\
\texttt{ANTLRCommonBacktrackingToken}
\end{tikzcd}$$

Figure 2: C++ Token Class-Hierarchy.

- Class `ANTLRTokenBase` describes the behavior of token objects as desired by ANTLR. Specifically, ANTLR token objects know their token type, line number, and associated input text. This is mainly for error reporting purposes.
- Any ANTLR grammar that uses DLG to produce a scanner must derive a token class from `DLGBasedToken`. This class adds a `int _line` field to the token object.
- Class `ANTLRCommonToken` is derived from `DLGBasedToken` and contains a text field.
- Class `ANTLRCommonBacktrackingToken` is the same as `ANTLRCommonToken` except that it should be used when syntactic predicates are used in the ANTLR grammar **OR** when lookahead $k > 1$. See the C++ test files and Section 3.3.

The programmer must still define type `ANTLRToken` to be one of the predefined classes or to one of their own just as in 1.20.

If you defined your own `ANTLRToken::makeToken()` for the 1.20 release, it's return type must be changed to

```
virtual ANTLRLightweightToken *makeToken(TokenType, ANTLRChar *, int);
```

3.2 The Mysterious `makeToken()` Function

Some readers may wonder why function `makeToken()` is required at all and why the programmer has to pass the address of an `ANTLRToken` into DLG during parser initialization. Why cannot the constructor be used to create a token and so on? The reason lies with the scanner, which must construct the token objects. The DLG support routines are typically in a precompiled object file that is linked in regardless of your token definition. Hence, DLG must be able to create tokens of any type.

Because objects in C++ are not "self-conscious" (i.e., they don't know their own type), DLG has no idea what the appropriate constructor is. Constructors cannot be virtual anyway; so, we had to come up with a "constructor" that is virtual and that acts like a factory—it returns the address of a (possibly new) token object upon each invocation rather than just initializing an existing object.

Because classes are not first-class objects in C++ (i.e., you cannot pass class names around), we must pass DLG the address of an `ANTLRToken` token object so DLG has access to the appropriate virtual table and is, thus, able to call the appropriate `makeToken()`.

This weirdness would disappear if all objects knew their type or if class names were first-class objects.

Here is the code fragment in DLG that constructs the token objects that are passed to the parser via the `ANTLRTokenBuffer`:

```
ANTLRAbstractToken *DLGLexerBase::
getToken()
{
    if ( token_vtbl==NULL ) DLGPanic("NULL token_vtbl");
    TokenType tt = nextTokenType();
    DLGBasedToken *tk;
    tk = (DLGBasedToken *)token_vtbl->makeToken(tt, _lextext, _line);
    tk->setLine(_line);
    return tk;
}
```

3.3 ANTLR Token Buffers and Streams

The 1.20 release of PCCTS connected an `ANTLRTokenStream` to the parser to provide tokens. In an effort to isolate the arbitrary lookahead mechanism from the parser class, 1.21 introduces `ANTLRTokenBuffers`. The parser is "attached" to an `ANTLRTokenBuffer` via interface functions `getToken()` and `bufferedToken()`. The object that actually consumes characters and constructs tokens, a derivatived of `ANTLRTokenStream`, is connected to the `ANTLRTokenBuffer` via interface function `getToken()` where `ANTLRTokenStream` is really just a behavior (class with no data). [C++ does not have this abstraction and hence we simply have come up with a fancy name for "`void *`"].

Define `DEBUG_TOKENBUFFER` to have ANTLR do extra checking to ensure your arguments to the `ANTLRTokenBuffer` constructor make sense. You must specify enough minimum arbitrary lookahead to cover the finite lookahead specified on the ANTLR command line.

The `ANTLRTokenBuffer` class maintains a "sliding window" of lookahead into the `ANTLRTokenStream`; a minimum window size must be specified during token buffer construction. This set of pointers can point to a single or multiple token objects. The following scenarios are possible:

1. **The parser does not need to backtrack (no syntactic predicates)**.

    - If DLG is used to produce an `ANTLRTokenStream`, then `makeToken()` can simply fill in a `static`, local copy of an `ANTLRToken` and return the same address continuously. Here is how an `ANTLRCommonToken` "computes" a token for DLG.

    ```
    virtual ANTLRLightweightToken *makeToken(TokenType tt, ANTLRChar *txt, int line)
            {
                static ANTLRCommonToken t;
                t.setType(tt); t.setText(txt); t.setLine(line);
                return &t;
            }
    ```

    - If you make your own scanner, you can return the same object just like DLG does. For example, a scanner that simply returned tokens with increasing integer token types could be defined as follows:

    ```
    ANTLRAbstractToken *MyLexer::getToken()
    {
        static MyToken t;
        static int i=1;
        t.setType(i++);
        return &t;
    }
    ```

    where `MyLexer` is the name of your scanner class.

In both cases, the scanner has the option to return the same physical object (modified each time the scanner is called) or return a stream of physical different token objects.

ANTLR by default makes a copy of all objects sent to the parser and so the `$`-variables point to distinct, local copies of the token objects passed to the token buffer. This is unnecessary if you pass distinct objects to the token buffer in the first place; the ANTLR `-ct` command line option can be used to turn this redundant copying off. [*Warning*: It is very possible that in a future version, the token buffer will do the copying rather than the parser].

2. **The parser must be able to backtrack**. In this case, physically distinct tokens must be passed to the `ANTLRTokenBuffer` by `ANTLRTokenStream::getToken()`. During backtracking, the parser reloads its local token type cache from the `ANTLRTokenBuffer`'s sliding window of token pointers. If the token pointers all pointed to the same token, the lookahead cache in the parser could not be reloaded correctly.

   - If DLG is used to produce an `ANTLRTokenStream`, DLG cannot simply modify and return the same token object. Currently, the `new` operator is used by the predefined token objects such as `ANTLRCommonBacktrackingToken`. The programmer is free to subclass and redefine `makeToken()` to use a more efficient memory allocator. Here is the definition of the `ANTLRCommonBacktrackingToken`.

```
class ANTLRCommonBacktrackingToken : public ANTLRCommonToken {
public:
    virtual ANTLRLightweightToken *makeToken(TokenType tt, ANTLRChar *txt, int
line)
        {
            ANTLRCommonToken *t = new ANTLRCommonToken;
            t->setType(tt); t->setText(txt); t->setLine(line);
            return t;
        }
    ANTLRCommonBacktrackingToken(TokenType t, ANTLRChar *s) : ANTLRCommonToken(t,s)
        {;}
    ANTLRCommonBacktrackingToken() {;}
};
```

   - If the programmer defines his/her own scanner, then they must return distinct token objects as well. For example, we could modify our `getToken()` above in the following way:

```
virtual ANTLRAbstractToken *MyLexer::getToken()
        {
            static MyToken *t = new MyToken;
            static int i=1;
            t->setType(i++);
            return t;
        }
```

For backtracking parsers, the `-ct` command line option should be used to turn off redundant copying token object copying because the `ANTLRTokenBuffer` will already contain distinct objects.

It is the programmers responsibility to track and to `delete` *any* `ANTLRToken` *objects that they create.*

3.4 Access to the Lookahead Buffer

ANTLR parsers may access the i^{th} token object of lookahead via the `ANTLRParser::LT(`i`)` function. `LT(1)` always returns a pointer to the token object for the next lookahead symbol. You may look ahead until end-of-file if necessary. Define `DEBUG_TOKENBUFFER` to have ANTLR do extra checking to ensure your arguments to the `ANTLRTokenBuffer` constructor make sense and that your calls to `LT()` are ok. You may check i for validity by calling `inputTokens->bufferSize()` where `inputTokens` is an member variable of your `ANTLRParser`; References to `LT(`i`)` beyond the end of file returns a pointer to the `ANTLRToken` for end of file as you have prescribed.

Semantic predicates in C++ mode should use the following rather than `LATEXT()`. For example,

```
typename : <<isType(LT(1)->getText())>>? ID ;
```

The `LA(`i`)` function access the local parser token type cache and, hence, is only valid for $i = 1..k$.

3.5 Arbitrary Lookahead and Semantic Predicates

There are language constructs that truly need a combined syntactic and semantic predicate to be parsed correctly. One possible solution is to use a semantic predicate that calls a function that "spins" ahead looking at the infinite token buffer and checks for semantic validity as well. For example,

```
int isQualClassName()
{
    int i=1;
    ANTLRToken *tk = LT(1);

    if ( LA(1)!=ID || tk==NULL ) return 0;

    while ( tk->getType()!=Eof &&
            (tk->getType()==ID || tk->getType()==COLON_COLON) )
    {
        tk = LT(++i);
    }
    if ( isClassName( LT(i-1)->getText() ) ) return 1;
    return 0;
}

qualified_classname
    :   <<isQualClassName()>>? ( ID "::" )* ID
    ;
```

where `isClassName()` is some function that examines the symbol table to determine whether or not the argument is a valid class name. This is not completely robust, but demonstrates the idea.

A mechanism like this is required to distinguish between qualified class names and qualified identifiers in C++.

3.6 Global Variables

If the programmer requires a variable which is visible to all rules, they may define a global variable inside the `class` definition. Doing so renders the variable a member of the class and, hence, a real global variable in the C/C++ sense is not defined—resulting in a cleaner (and re-entrant) program. For example,

```
class MyClass {
<<char *current_file_name;>>

a : ... <<current_file_name = "blah";>> ;

}
```

3.7 $-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.

3.8 DLG Classes

The DLG C++ interface has not changed from a programmer's point of view.

4 Miscellaneous Changes

- The `genmk` program has been upgraded in a few minor ways. For example, it is now sensitive to the `config.h` file; hence, it will now do the "right thing" for the PC (such as using `.obj` instead of `.o`).
- The ANTLR `-gk` option is now a warning not an error when used with semantic predicates.
- In C++ mode, the programmer can change the `_line` member, but should call the `newline()` member function instead. Access to the current line number can be obtained via `line()`. The parser normally does not access the line number directly from the lexer in C++ mode, however. Typically, the `ANTLRToken` object contains the line number on which it was found.
- In C++ mode, DLG now assumes `DLGInputStream::nextChar()` returns an `int` so that -1 (`EOF`) is handled correctly.

5 New or Renamed Supplied Files

- `AParser.h`: All ANTLR parser support classes and the `ANTLRParser` class itself.
- `AParser.C`: ANTLR parser support code.
- `DLexerBase.h`: DLG scanner support classes and `DLGLexerBase` class.
- `DLexerBase.C`: DLG scanner support code.
- `ASTBase.h`: AST class definition.
- `ASTBase.C`: AST support code.
- `DLexer.C`: Support code that must be aware of the particular scanner generated by DLG. This is an ugly mechanism for including the DFA automaton and will change in future versions.
- `AToken.h`: Definitions for classes `ANTLRTokenBase`, `ANTLRLightweightToken`, `ANTLRAbstractToken`, `DLGBasedToken`, and `ANTLRCommonToken`.
- `ATokenStream.h`: Definition of class `ANTLRTokenStream`.
- `ATokenBuffer.h`: Definition of class `ANTLRTokenBuffer`.
- `ATokenBuffer.C`: Code for class `ANTLRTokenBuffer`.

6 Bugs Fixed for 1.21

- Fixed a hideous bug in the `-gl` generate line info option that sometimes put the line directive not at the left edge of a line.
- Fixed a bug in `_match` with `-gk` mode; it didn't work before.
- Added `_setmatch()` to C++ output mode.
- The `genmk` program had a number of small bugs.
- In the old `antlrx.h` file for C++ output, `zzfailed_pred` was missing a semi-colon.
- The line number stuff for infinite lookahead was screwed up even for C mode. This has changed so that the `zzline` variable (in C mode) consistently for any mode. For C++ output, the `newline()` and `line()` macros may be used. To access the line number for a particular token from the parser, use the `getLine()` `ANTLRToken` function.
- The return type for `erraction` in the DLG C++ output should have been `TokenType` to be consistent with the other lexical actions.
- The `enum TokenType` definition no longer has a trailing comma.
- DLG sometimes defined `DfaState` inconsistently for C++ output.
- The AST stuff didn't work correctly with syntactic predicates because of a missing `zzNON_GUESS_MODE` in the `zzEXIT` macro (C mode).
- Previously, the `-w2` generated a warning for basically every token definition indicating that it has no associated regular expression.
- The `DLGFileInput` class now always immediately returns `EOF` once that condition has been detected once. It is no longer necessary to type more than one end of file character in C++ mode.

7 Future

- Good error recovery and reporting is notoriously difficult to achieve with parser generators, especially $LALR$-based tools. The previously mentioned *parser exception handling* was discussed at the first annual PCCTS workshop. A reasonable implementation/syntax has been obtained and we anticipate their introduction by the end of 1994.
- 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 using a multi-platform window library. 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. The GUI would be an actual product sold by Parr Research Corporation, but would be tremendously cool.

[*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 a number of other little things.*

We anticipate an intermediate break that will change the grammar meta-language. 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.]

8 Acknowledgements

As usual, there are a large number of people to thank for their help with PCCTS (more than we can hope to acknowledge here).

Thanks are due to Sumana Srinivasan, Mike Monegan, and Steve Naroff of NeXT, Inc. for their continuing help in the definition of the ANTLR C++ output. Further, Sumana's work on the C++ grammar (no, we don't have a release date set yet) is fabulous.

We thank Gary Funck at Intrepid Technology for his work on SORCERER, which we hope to release soon with PCCTS as standard baggage. He always has good suggestions for ANTLR as well.

Steve Robenalt at Rockwell single-handedly pushed the `comp.compilers.tools.pccts` news group through. Further, he continues to maintain the FAQ.

We thank Scott Haney at Lawrence Livermore National Labs for his Macintosh port of PCCTS.

We thank Tom Moog (`moog@polhode.com`) for his continued efforts on the `NOTES.newbie` information file.

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

The planning group for the first annual PCCTS workshop included:

```
Gary Funck      <gary@intrepid.com>,            Intrepid Technology, Inc.
Steve Robenalt  <steve@molly.dny.rockwell.com>, Rockwell International
Ivan Kissiov    <ivan@cadence.com>,             Cadence
```