

# *Notes for New Users of PCCTS*

Thomas H. Moog  
Polhode, Inc.  
tmoog@polhode.com

*27 March 2000 Release 2.22*  
*PCCTS Version 1.33MR22*

© Copyright 2000 Polhode, Inc. These notes may be redistributed in electronic form or printed for personal use as long as there is no charge for them, proper credit is given to the author, any changes to the text are clearly marked, and the copyright and disclaimer are retained.

Disclaimer: These notes are provided "as is". They may include typographical or technical errors. The author disclaims all liability of any kind or nature for damages due to error, fault, defect, or deficiency in the notes regardless of cause. All warranties of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose are disclaimed.

**Where is**

- #1. The current maintenance release of PCCTS, these notes, and related examples are available on the net 1
- #2. Some other items available at <http://www.polhode.com>: 1
- #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 1
- #7. In #token regular expressions spaces and tabs which are not escaped are ignored 1
- #8. Never choose names which coincide with compiler reserved words or library names 1
- #9. Write <<predicate>>? not <<predicate *semi-colon*>>? (semantic predicates go in "if" conditions) —
- #10. Some constructs which cause warnings about ambiguities and optional paths 1

**Checklist**

- #11. Locate incorrectly spelled #token symbols using ANTLR -w2 switch or by inspecting *parserClassName.cpp* 1
- #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 2
- #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 2

**#token**

- #16. To change the token name appearing in syntax error messages: #token ID("identifier") "[a-z A-Z]+" 2
- #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) —
- #21. DLG wants to find the longest possible string that matches 2
- #22. When two regular expressions of equal length match a regular expression the first one is chosen 2
- #23. Inline regular expression are no different than #token statements 2
- #24. Watch out when you see ~ [list-of-characters] at the end of a regular expression 3
- #25. Watch out when one regular expression is the prefix of another 3
- #26. DLG is not able to backtrack (unlike flex) 3
- #27. The lexical routines mode(), skip(), and more() are *not* complicated ! 4
- #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() 4
- #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 4
- #32. For 8 bit characters use flex or in DLG make char variables unsigned (g++ option -funsigned-char) 4
- #33. The maximum size of a DLG token is set by an optional argument of the ctor DLGLexer() — default is 2000 4
- #34. If a token is recognized using more() and its #lexclass ignores end-of-file then the very last token will be lost 4
- #35. Sometimes the easiest DLG solution is to accept one character at a time. 5

**#tokclass**

- #36. #tokclass provides an efficient way to combine reserved words into reserved word sets 5
- #37. Use ANTLRParser::set\_el() to test whether an ANTLRTokenType is in a #tokclass or #FirstSetSymbol 5

**#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 5
- #40. Use a stack of #lexclass modes in order to emulate lexical subroutines 6
- #41. Sometimes a stack of #lexclass modes isn't enough 6

**Lexical Lookahead**

- #42. Vern Paxson's flex has more powerful features for lookahead than dlgl 6
- #43. Extra lookahead is available from class BufFileInput (subclass of DLGInputStream) 6
- #44. One extra character of lookahead is available to the #token action routine in ch (except in interactive mode) 7

#45.	There is no easy way in DLG to distinguish integer "1" from floating point "1." when "1.and.2" is valid	7
#46.	For lex operators "^" and "\$" (anchor pattern to start/end of line) use flex - don't bother with dlgl	—
<b>Line and Column Information</b>		
#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	7
#53.	Computing column numbers when using more() with strings that include tab characters and newlines	8
<b>Ambiguity Aid (options -aa, -aam, -aad)</b>		
#54.	Example with nested if statement	8
#55.	Example with cast expression	10
#56.	Example with ambiguity due to limitations of linear approximation	11
#57.	Summary of command line switches related to ambiguity aid	13
<b>C++ Mode</b>		
#58.	The destructors of base classes should be virtual in almost all cases	13
#59.	Why must the AST root be declared as ASTBase rather than AST ?	13
#60.	C++ mode makes multiple parsers easy	14
#61.	Use DLGLexerBase routines to save/restore DLG state when multiple parsers share a token buffer	14
#62.	In C++ mode ASTs and ANTLRTokens do not use stack discipline as they do in C mode	14
#63.	Summary of Token class inheritance in file AToken.h	14
#64.	Diagram showing relationship of major classes	14
#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"	15
#67.	The lexer can access parser information using member function getParser()	15
#68.	Additional notes for users converting from C to C++ mode	15
#69.	Use the macro mytoken( <i>expr</i> ) to convert an ANTLRTokenPtr to an ANTLRToken *	—
#70.	When using reference counted tokens be careful about saving a pointer generated by myToken()	15
#71.	LA( <i>i</i> ) is a cache of LT( <i>i</i> ) values used by the parser — it is valid only for $i \leq k$	—
#72.	To disable reference counting of ANTLRTokens use <i>parserName.noGarbageCollectTokens()</i>	—
#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	15
<b>ASTs</b>		
#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	16
#79.	Use symbolic tags (rather than numbers) to refer to tokens and ASTs in rules	16
#80.	Constructor AST(ANTLRTokenPtr) is automatically called for terminals when ANTLR -gt switch is used	16
#81.	If you use ASTs you have to pass a root AST to the parser	16
#82.	Use ast->destroy() to recursively descend the AST tree and free all sub-trees	—
#83.	Don't confuse #[...] with #( ... )	16
#84.	The make-a-root operator for ASTs ("^") can be applied only to terminals (#token, #tokclass, #tokdef)	17
#85.	An already constructed AST tree cannot be the root of a new tree	17
#86.	Don't assign to #0 unless automatic construction of ASTs is disabled using the "!" operator on a rule	17
#87.	The statement in Item #86 is stronger than necessary	17
#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	17
#90.	When passing #( ... ) or #[ ... ] to a subroutine it must be cast from "ASTBase *" to "AST *"	17
#91.	Some examples of #( ... ) notation using the PCCTS list notation	18
#92.	A rule which derives epsilon can short circuit its caller's explicitly constructed AST	18
#93.	How to use automatic AST tree construction when a token code depends on the alternative chosen	18

#94.	For doubly linked ASTs derive from class ASTDoublyLinkedBase and call <code>tree-&gt;double_link(0,0)</code>	18
#95.	When ASTs are constructed manually the programmer is responsible for deleting them on rule failure	18
<b>Rules</b>		
#96.	To refer to a field of an ANTLRToken within a rule's action use <code>&lt;&lt;... mytoken(\$x)-&gt;field...&gt;&gt;</code>	19
#97.	Rules don't return tokens values, thus this won't work: <code>rule: r1:rule1 &lt;&lt;...\$r1...&gt;&gt;</code>	19
#98.	A simple example of rewriting a grammar to remove left recursion	19
#99.	A simple example of left-factoring to reduce the amount of ANTLR lookahead	19
#100.	ANTLR will guess where to match "@" if the user omits it from the start rule	20
#101.	To match any token use the token wild-card expression "." (dot)	20
#102.	The "~" (tilde) operator applied to a #token or #tokclass is satisfied when the input token does <i>not</i> match	20
#103.	To list the rules of the grammar <code>grep parserClassName.h</code> for "_root" or edit the output from ANTLR <code>-cr</code>	—
#104.	The ANTLR <code>-gd</code> trace option can be useful in sometimes unexpected ways	20
#105.	Associativity and precedence of operations is determined by nesting of rules	20
#106.	#tokclass can replace a rule consisting only of alternatives with terminals (no actions)	21
#107.	Rather than comment out a rule during testing, add a nonsense token which never matches — See Item #110.	—
<b>Init-Actions</b>		
#108.	Don't confuse init-actions with leading-actions (actions which precede a rule)	21
#109.	An empty sub-rule can change a regular action into an init-action	22
#110.	Commenting out a sub-rule can change a leading-action into an init-action	22
#111.	Init-actions are executed just once for sub-rules: <code>(...)+</code> , <code>(...)*</code> , and <code>{...}</code>	22
<b>Inheritance</b>		
#112.	Downward inherited variables are just normal C arguments to the function which recognizes the rule	22
#113.	Upward inheritance returns arguments by passing back values	23
#114.	Be careful about passing via upward inheritance <code>LT(i)-&gt;getText()</code> if using <code>ANTLRCommonToken</code>	23
#115.	ANTLR <code>-gt</code> code will include the AST with downward inheritance values in the rule's argument list	—
#116.	Predefine the <code>PURIFY</code> macro if you are passing objects using upward inheritance	23
<b>Syntactic Predicates</b>		
#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	24
#120.	How to use init-actions to create side-effects in guess mode (despite Item #119)	24
#121.	With values of $k > 1$ or infinite lookahead mode one cannot use feedback from parser to lexer	24
#122.	Can't use interactive scanner (ANTLR <code>-gk</code> option) with ANTLR infinite lookahead	—
#123.	Syntactic predicates are implemented using <code>setjmp/longjmp</code> — beware C++ objects requiring destructors	—
<b>Semantic Predicates</b>		
#124.	Semantic predicates have higher precedence than alternation: <code>&lt;&lt;&gt;&gt;? A B</code> means <code>(&lt;&lt;&gt;&gt;? A) B</code>	—
#125.	Get rid of warnings about missing <code>LT(i)</code> by using a comment: <code>/* LT(i) */</code>	—
#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	24
#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	24
#130.	A semantic predicate which is not at the left edge of a rule becomes a validation predicate	24
#131.	Semantic predicates are not always hoisted into the prediction expression	25
#132.	Semantic predicates can't be hoisted into a sub-rule: <code>{x} y</code> is not exactly equivalent to <code>x y   y</code>	25
#133.	How to change the reporting of failed semantic predicates	25
#134.	A semantic predicate should be free of side-effects because it may be evaluated multiple times	25
#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?	26
#137.	Use ANTLR option <code>"-info p"</code> for information on how semantic predicates are being handled and hoisted	26
#138.	Semantic predicates, predicate context, and hoisting	27
#139.	Another example of predicate hoisting	28
#140.	Example of predicate hoisting and suppression with the ANTLR option <code>-mrhoist on</code>	29
#141.	The context guard <code>(...)? &amp;&amp; &lt;&lt;predicate&gt;&gt;? vs. (...) =&gt; &lt;&lt;predicate&gt;&gt;?</code>	31

#142. Experimental ANTLR option -mrhoistk on for suppression of predicates with lookahead depth $k > 1$	33
#143. Use #pred statement to describe the logical relationship of related predicates	34
#144. Disable predicate hoisting explicitly using the pseudo-action: rule: <<;>> <<nohoist>> ...	34
#145. Simplification of predicate expressions when there are multiple references to predicates	34
<b>Debugging Tips for New Users of PCCTS</b>	
#146. A syntax error with quotation marks on separate lines means a problem with newline	35
#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	35
#150. For complex problems use traceOption and traceGuessOption to control trace output	-
<b>Switches and Options</b>	
#151. Use ANTLR -gx switch to suppress regeneration of the DLG code and recompilation of DLGLexer.cpp	35
#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	35
#154. Use ANTLR option -glms to convert Microsoft file names like "..\foo.g" to "../foo.g" in generated files	-
#155. Use ANTLR option -treport <i>number</i> to locate alternatives using a lot of CPU time to resolve	35
#156. The ANTLR option -info ( p - predicate, t - tnodes, m - monitor, f - follow set, o - orphans)	35
<b>Multiple Source Files</b>	
#157. To see how to place main() in a .cpp file rather than a grammar file (.g) see pccts/testcpp/8/main.cpp	36
#158. How to put file scope information into the second file of a grammar with two .g files	36
<b>Source Code Format</b>	
#159. To place the C right shift operator ">>" inside an action use "\>>"	36
#160. One can continue a regular expression in a #token statement across lines (or use flex definitions)	36
#161. A #token without an action will attempt to swallow an action which immediately follows it - use ";"	36
<b>Miscellaneous</b>	
#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)	37
#164. To remake ANTLR after changes to the source code use make -f makefile1	37
#165. ANTLR reports "... action buffer overflow ..."	37
#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	37
#168. See pccts/testcpp/2/test.g and testcpp/3/test.g for examples of how to integrate non-DLG lexers with PCCTS	-
#169. Ambiguity, full LL( $k$ ), and the linear approximation to LL( $k$ )	37
#170. Ambiguity, #pragma, and ANTLR -rl switch (Contributed by John Lilley jlilley@empathy.com)	39
#171. What is the difference between "(...)? <<...>>? x" and "(...)? => <<...>>? x"?	41
#172. Memory leaks and lost resources	41
#173. Some ambiguities can be fixed by introduction of new #token numbers	41
#174. Subclassing DLGInputStream	42
<b>Changes From The Original 1.33 Which Are Not Part of Any Other Section</b>	
#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	42
#178. Use "class MyParser : public MyBaseParser ... { " to specify your own parser base class	42
#179. Use #FirstSetSymbol ( <i>symbol_name</i> ) to generate symbol for first set of an alternative	42
#180. Use -preamble and -preamble_first to insert macro code at the start of each rule or block	42
#181. Preprocessor option ZZDEFER_FETCH to defer token fetch for C++ mode	42
#182. Exception handling	42
<b>(C Mode) LA/LATEX and NLA/NLATEX</b>	
#183. Do not use LA( $i$ ) or LATEX( $i$ ) in the action routines of #token	43
#184. Care must be taken in using LA( $i$ ) and LATEX( $i$ ) in interactive mode (ANTLR switch -gk)	43
<b>(C Mode) Execution-Time Routines</b>	
#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	43
#187. Saving and restoring parser state in order to parse other objects (input files)	43
<b>(C Mode) Attributes</b>	
#188. Use symbolic tags (rather than numbers) to refer to attributes and ASTs in rules	44
#189. Rules no longer have attributes: <code>rule : r1:rule1 &lt;&lt;...\$r1...;&gt;&gt;</code> won't work	44
#190. Attributes are built automatically only for terminals	44
#191. How to access the text or token part of an attribute	44
#192. The \$0 and \$\$ constructs are no longer supported — use inheritance instead (Item #113)	—
#193. If you use attributes then define a <code>zsd_attr()</code> 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	44
#195. A <code>charptr.c</code> attribute must be copied before being passed to a calling rule	44
#196. Attributes created in a rule should be assumed <i>not</i> valid on entry to a fail action	45
#197. Use a fail action to destroy temporary attributes when a rule fails	45
#198. When you need more information for a token than just token type, text, and line number	45
#199. About the pipeline between DLG and ANTLR (C mode)	45
<b>(C Mode) ASTs</b>	
#200. Define a <code>zsd_ast()</code> to recover resources when an AST is deleted	—
#201. How to place prototypes for routines using ASTs in the <code>#header</code>	46
#202. To free an AST tree use <code>zsfree_ast()</code> to recursively descend the AST tree and free all sub-trees	46
#203. Use <code>#define zzAST_DOUBLE</code> to add support for doubly linked ASTs	46
<b>Extended Examples and Short Descriptions of Distributed Source Code</b>	
#1. DLG definitions for C and C++ comments, character literals, and string literals	46
#2. A simple floating point calculator implemented using PCCTS attributes and inheritance	46
#3. A simple floating point calculator implemented using PCCTS ASTs and C++ virtual functions	46
#4. An <code>ANTLRToken</code> class for variable length strings allocated from the heap	46
#5. How to extend PCCTS C++ classes using the example of adding column information	46
#6. Use of parser exception handling in C and C++ programs	46
#7. How to pass whitespace through DLG for pretty-printers	47
#8. How to prepend a newline to the <code>DLGInputStream</code> via derivation from <code>DLGLexer</code>	47
#9. How to maintain a stack of <code>#lexclass</code> modes	47
#10. When you want to change the token type just before passing the token to the parser	47
#11. Rewriting a grammar to remove left recursion and perform left factoring	47
#12. Processing counted strings in DLG	48
#13. How to convert a failed validation predicate into a signal for treatment by parser exception handling	49
#14. How to use Vern Paxson's flex with PCCTS in C++ mode by inheritance from <code>ANTLRTokenStream</code>	49
#15. Using the GNU <code>gperf</code> (generate perfect hashing function) with PCCTS	49
#16. Multiple files managed as a single token stream	49