The Purdue Compiler Construction Tool Set

Version 1.06 Update

December 1, 1992

1. Enhancements

This section describes the update of PCCTS Version 1.00 to Version 1.06 (1.01-1.05 were internal releases). The list of changes here is not complete since many ‘‘little’’ fixes were made (actually, the predicated parsing feature is major). Here, we concentrate on the enhancements to the system; file `BUGS100` contains a list of the bug fixes incorporated in the 1.06 release. In addition, note that the manual has not been updated and will not be until the next major release (or until we get around to it).

1.1. Predicated Parsing — ALPHA Version

Normally, parsing is a function of syntax alone. Semantics (meaning/value of the input) are not taken into account when parsing decisions are made—context-free parsing. Typically, languages, such as C++, have some rules which require the lexical analyzer to consult the symbol table to determine what token (e.g. classname verses enumerated name verses typedef name) is given to the parser. However, lexical analyzers have no context information except the current token of lookahead and must be coerced via flags to yield the various token types. Ad hoc approaches become increasingly difficult to implement as the number of ambiguities (due to lack of semantic information) in a grammar rises.

Context dictates the scope of variables in programming languages. Because only the parser has context information, it is reasonable to assume that the parser is the correct place to maintain and access the symbol table; there are other situtations in which it is useful to have context direct parsing. Unfortunately, most parser generators do not allow grammar actions to influence the parse; i.e. parsing is a function of syntax alone. Because PCCTS generates recursive-descent LL(k) parsers in C, allowing user-defined C actions to influence parsing is a straightforward and natural addition to the PCCTS description meta-language. Although bottom-up parsing strategies can allow actions to direct parsing, bottom-up parser have much less semantic information; i.e. LR and LALR techniques allow only *S*-attributed grammars whereas LL allows *L*-attributed grammars (for example, LL always knows which set of rules it is currently trying to match whereas LALR does not, in general; see your favorite book on parsing theory).

In this section, we introduce a rudimentary version of predicates, with a full implementation to appear in future releases. To support context-sensitive parsing PCCTS 1.06 allows a new action type called *predicate* (`-pr` must be used), which is denoted:

```
<<predicate>>?
```

or, optionally,

```
<<predicate>>?[fail_action]
```

where the second notation ‘‘passes’’ an argument of an action to the predicate operator ‘‘?’’, which is executed upon predicate failure. A predicate is a C action that evaluates to either true (success) or false (failure) and can be used for both *semantic validation* and for *disambiguating* syntactic conflicts in the underlying grammar.

Validation predicates are used as simple tests and are functionally equivalent to the following normal action:

```
if ( !predicate ) {error message; exit rule}
```

or, when a fail action is present:

```
if ( !predicate ) {fail_action}
```

For example,

```
a   :   A <<pred>>? B
    ;
```

Matches `A`, evaluates `pred`, and matches `B` if `pred` is true else an error is reported (i.e. ‘‘`failed predicate 'pred'`’’) and rule `a` is exited before matching `B`. To generate a more useful error message, the user may specify a fail action:

```
a   :   A <<pred>>?[fail] B
    ;
```

which executes `fail` when `pred` evaluates to false.

A predicate which validates a production can, at the same time, be used to disambiguate a syntactically ambiguous grammar structure. To be a candidate, a predicate must appear on the extreme left edge of a production. Disambiguating predicates are used in the alternative prediction expressions to enable or disable certain alternative productions of a block (rule or subrule). Parsing in this new environment can be viewed in this way:

step 1: Disable invalid productions

An *invalid* production is a production whose disambiguating predicate evaluates to false.

step 2: Parse as normal block

Once a list of *valid* productions has been found, they are parsed as a normal rule or subrule.

Essentially, disambiguating predicates ‘‘gate’’ alternative productions in and out depending on their ‘‘applicability’’. Productions without predicates have an implied predicate of `<<TRUE>>?`; i.e. they are always valid.

A single predicate can be used as both a validation and a disambiguating predicate—ANTLR determines which way to use them automatically. The role taken by a predicate depends on its location in the grammar. Predicates are all considered validation predicates as they must always evaluate to true for parsing of the enclosing production to continue. However, when a

grammatical ambiguity is discovered, ANTLR searches for *visible* predicates to disambiguate the construct. A visible predicate is one which can be evaluated in a production prediction expression without consuming a token or executing an action (except initialization actions); e.g. all disambiguating predicates appear on the left edge of some production. Fail actions are not used in prediction expressions because a failed predicate disables a production; it does not report a syntax error unless no other *viable* predicates are present; a viable production is one which is predicted by the lookahead. The action of moving a predicate into a prediction expression is called *hoisting*; currently, at most one predicate can be hoisted per ambiguous production. In general, predicates may only be a function of:

User variables

This is dangerous as hoisting may evaluate the predicate in a scope where the variable(s) does not exist; e.g. using rule parameters can be a problem if a predicate that references them is hoisted outside of the rule.

Attributes

Only attributes in the left context can be used; i.e. attributes of rules/tokens created before the predicate is evaluated.

Lookahead

The next *k* tokens of lookahead can be tested via `LA(i)`, `LATEXT(i)`.

Also, **predicates may not have side-effects** (user must undo anything that was changed if they do; in other words, they must ‘‘backtrack’’). See the Future Work section for information regarding the use of predicates and extremely large lookahead buffers.

Consider the following simple grammar:

```
a   :   <<pred1>>? A B
    |   <<pred2>>? A C
    ;
```

This grammar is LL(2) (`-k 2`) and ANTLR would not report an ambiguity. However, it is LL(1) ambiguous (`-k 1`) and `antlr t.g` would generate a warning message:

```
t.g, line 1: warning: alts 1 and 2 of rule ambiguous upon { A }
```

Now, if we allow predicates to be used with `antlr -pr t.g`, then ANTLR finds that both predicates are visible and can be used to disambiguate the production; it is up to the user to ensure that the predicates do, in fact, disambiguate the predicate. ANTLR would generate code similar to the following:

```
a()
{
    if ( pred1 && LA(1)==A ) {
    }
    else if ( pred2 && LA(1)==A ) {
    }
    else error;
}
```

The following grammars are syntactically and semantically equivalent to the above grammar, but the predicates are not hoisted trivially:

```
a   :   ( <<pred1>>? A B )
    |   <<pred2>>? A C
    ;
```

ANTLR looks inside the subrule of production one and finds that `pred1` can be evaluated before executing an action and before matching a token; it is hoisted for use in the prediction expression of rule `a`. Predicates can be hoisted from other rules as well:

```
a   :   b
    |   c
    ;

b   :   <<pred1>>? A B
    ;

c   :   <<pred2>>? A C
    ;
```

Rule `a` would still have the same production prediction expression. The following grammar does not have predicates that can be hoisted to disambiguate the prediction:

```
a   :   A <<pred1>>? B             /* cannot hoist past token */
    |   <<action>> <<pred2>>? A C  /* cannot hoist past action */
    ;
```

Now, consider how a simplified version of K&R C variable and function declarations could be specified:

```
decl:   /* variable definition, function DECLARATION/DEFINITION */
        type declarator ( func_body | ";" )
    |   /* function DEFINITION */
        declarator func_body
    ;

type:   "int"
    |   "float"
    |   WORD
    ;

declarator
    :   WORD { "\(" args "\)" }
    ;

func_body
    :   ( decl )* "\{" (decl)* (stat)* "\}"
    ;
```

unfortunately, rule `decl` is totally ambiguous in the LL(1) sense (and in the LL(k) sense) since `WORD` can begin both alternatives of rule `decl`. Increasing the lookahead to two tokens does not help as the alternatives are ambiguous upon `WORDWORD`. Alternative one could match `my_type var` and alternative two could match `func my_type`; hence, the second alternative matches at least one invalid sentence. This is because the `( decl )*` subrule of rule

`func_body`, which would match the second `WORD` of `WORD WORD`, does not know if an argument list was seen previously or not. The question of whether to match a parameter definition list after a declarator is context-sensitive because a function header may or may not have been seen immediately beforehand.

One way in which this subset could be recognized is to recognize a large superset of the language and then, using actions, verify that the input was valid:

```
decl:   {type} declarator ( func_body | ";" )
    ;
```

But, jamming everything together makes action introduction more complicated. It is better to specify the input language exactly with predicates.

To overcome the ambiguity in rule `decl` and to ensure that only a function declarator is followed by a parameter definition, we augment the grammar as follows:

```
decl:   <<int is_func;>>

        /* variable definition, function DECLARATION/DEFINITION */
        type declarator > [is_func]
        (   <<is_func>>?[printf("warning: declarator not func\n");]
            func_body
        |   ";"
        )

    |   /* function DEFINITION */
        declarator > [is_func]
        <<is_func>>?[printf("warning: expecting func header not %s\n", LATEXT(1));]
        func_body
    ;

type:   "int"
    |   "float"
    |   <<LA(1)==WORD ? isTYPE(LATEXT(1)) : 1>>?
        WORD
    ;

declarator > [int is_func]
    :   <<LA(1)==WORD ? !isTYPE(LATEXT(1)) : 1>>?
        WORD ( "\(" <<$is_func=1;>> args "\)" | <<$is_func=0;>> )
    ;

func_body
    :   ( decl )* "\{" (decl)* (stat)* "\}"
    ;
```

In rule `type`, we add a predicate that ensures that, if the lookahead is a `WORD`, that the `WORD` is, in fact, a user-defined type; nothing can be said if the lookahead is something else, so we return success in that case; i.e. the predicate does not apply for that lookahead (context). If rule `declarator` is called, a failure of the predicate will report a semantic error. In our example, the predicate is also hoisted for use as a disambiguating predicate in rule `decl`. The predicate in rule `declarator` ensures that, if the lookahead is a `WORD`, that the `WORD` is not a user-defined type. As before, it is also hoisted to rule `decl` to disambiguate the second production of

that rule. Both productions of rule `decl` have predicates that are hoisted into the prediction expressions for those productions; ANTLR assumes that the hoisted predicates **completely** disambiguate the decision, which is true in our case. The first predicate physically present in rule `decl` ensures that, if a function body is coming up, the declarator was a function header. The second predicate in rule `decl` again ensures that a function header was seen before trying to match a function body. The variable `is_func` is set to the return value of rule `declarator`, which is set by two simple actions in `declarator`.

Currently, the context under which predicates are evaluated may not be correct; i.e. currently ANTLR does not compute the context from which it hoists a predicate. For example,

```
a   :   b B
    |   (X Y | A C)
    ;

b   :   <<pred>>? A
    ;
```

The predicate `pred` will be hoisted for use in disambiguating rule `a`. However, the predicate will be evaluate regardless of whether `A`, which is its context, or `X` is on the input stream. It may be invalid to apply the predicate when the lookahead is `X`. The user may overcome this by changing `pred` thus:

```
a   :   b B
    |   (X Y | A C)
    ;

b   :   <<LA(1)==A ? pred : TRUE>>? A
    ;
```

which will ensure that the predicate is only evaluated when `A` is seen on the input stream.

Another problem area lies in that hoisting can occur in situations that will break the semantics. For example:

```
a   :   b[3]
    |   A
    ;

b[int i]
    : <<f($i)>>? A
    ;
```

The predicate `f(i)` will be hoisted to rule `a` where the parameter `i` is not even defined. Either do not do this or put a predicate (dummy or otherwise) between the decision and the predicate you do not wish to be hoisted:

```
a   :   <<1>>? b[3]
    |   A
    ;

b[int i]
    : <<f($i)>>? A
    ;
```

This is an alpha release feature in that it is not anywhere near what we want it to do and has NOT been thoroughly tested. User beware. We do not guarantee that the syntax of predicates will stay this way; however, the semantics will not change. We want users to play with predicates to find new, better or different ways of doing things. We **really** want user feedback on these critters before a full implementation is developed. For example, how do you want the parser to respond if no valid, viable production is found?

```
a   :   <<p1>>? A
    |   <<p2>>? A
    |   B
    ;
```

If the input were `C`, a correct error message would be reported:

```
line 1: syntax error at "C" missing { A B }
```

This is the correct message because it is truly a syntax, as opposed to semantic, error. On the other hand, upon input `A`, when predicates `p1` and `p2` are false, the parser reports the following bizarre message:

```
line 1: syntax error at "A"
```

which is not very meaningful. The fail function, `zzFAIL`, found that, in fact, `A` is syntactically valid and got confused because an error was raised; this is because the fail routine has no semantic information. A suitable definition of error reporting in the predicate environment must be created.

Please send any suggestions/questions to `parrt@ecn.purdue.edu`, `hankd@ecn.purdue.edu` or use the email server mechanism at `pccts@ecn.purdue.edu` (send a `Subject:` line of ‘‘question antlr’’ and fill in the body of the email with your suggestion/question).

1.2. 1.06 Is Written in 1.00

The grammar for the PCCTS meta-language (file format) has been implemented in Version 1.00, making heavy use of `#lexclass` directives. File `lexhelp.c` has been eliminated due to the superiority of 1.00 to 1.00B.

1.3. ANTLR Compiles Under ANSI C

Because of the rewrite of the grammar and some rewrites of ANTLR code, ANTLR now compiles with ANSI compilers without a wimper (except for two ‘‘`unknown escape sequence`’’ warnings). Your mileage may vary.

1.4. Grammar Files Distributed

`antlr.g` and `dlg_p.g` are now included in the source distribution for 1.06. Note that the 1.00 PCCTS (both ANTLR and DLG) are required to process these grammar files. DO NOT DESTROY YOUR OLD COPY OF 1.00 PCCTS (at least save the executables).

1.5. Script Generates Makefiles for PCCTS Projects

A C program called `genmk.c` is available in the `support` directory of the PCCTS release which has the following usage:

```
genmk project f1.g f2.g ... fn.g
```

It generates a makefile that creates an executable, `project`, from a set of grammar files. For example, `genmk t t.g` generates:

```
#
# PCCTS makefile for: t.g
#
DLG_FILE = parser.dlg
ERR_FILE = err.c
HDR_FILE = stdpccts.h
TOK_FILE = tokens.h
K = 1
ANTLR_H = .
BIN = .
ANTLR = $(BIN)/antlr
DLG = $(BIN)/dlg
CFLAGS = -I. -I$(ANTLR_H)
AFLAGS = -fe err.c -fh stdpccts.h -fl parser.dlg -ft tokens.h -k $(K)
-gk
DFLAGS = -C2 -i
GRM = t.g
SRC = scan.c t.c err.c
OBJ = scan.o t.o err.o

t: $(OBJ) $(SRC)
        cc -o t $(CFLAGS) $(OBJ)

t.c parser.dlg : t.g
        $(ANTLR) $(AFLAGS) t.g

scan.c : parser.dlg
        $(DLG) $(DFLAGS) parser.dlg scan.c
```

This program is handy when beginning a new PCCTS project or when first learning about PCCTS.

1.6. DLG Supports Case Insensitive Scanners

DLG has two new options which provide control over the case sensitivity of the generated scanner. Specifically, case insensitivity implies that when a character is referenced in a regular expression, DLG behaves as if the user had typed both upper and lower case versions of that character; i.e. `(`*a*`|`*A*`)` where *a* is some character. The new options are:

`-ci` Make lexical analyzer case insensitive

`-cs` Make lexical analyzer case sensitive (default).

1.7. Delayed Lookahead Fetches in Generated Parser

Currently, PCCTS generates parsers which always have *k* tokens of lookahead. This is done by following the strategy that another token is fetched (`zzCONSUME`) when one is matched (`zzmatch`). This can be a problem for actions that need to occur after a token has been matched, but before the next token of lookahead is fetched. This is somewhat overcome in PCCTS 1.00 by delaying the fetch if an action is found immediately after a token reference. With the new delayed lookahead scheme, the next token is not consumed until the next match is required. This means that any action before the next match (not necessarily adjacent to the previous match) will be executed before a lookahead fetch occurs. Turn on ANTLR option `-gk` and DLG option `-i` to enable this feature. This feature appears to work with AST's and attributes with the constraints mentioned below in the incompatibilities section (e.g. use of `LA(`*i*`)` and `LATEXT(`*i*`)` has been restricted). It has been tested with the C (*k*>1 and Pascal (*k*==1) examples provided in the release and with several other large grammars.

This feature is primarily useful for developers of interactive tools. Previously, it was **really** hard to get PCCTS to generate truly interactive tools. It appeared as if the parser was always waiting on a token fetch rather than executing an appropriate action. E.g. in PCCTS 1.00,

```
a : ( "A" "B" "C" "\n" ) <<printf("got it\n");>>
  ;
```

would not print `got it` until one token after the newline had been typed. PCCTS 1.06 will generate parsers that print the message immediately upon newline and will exit without waiting for another token as there are no token references following the action.

Another way in which delayed lookahead is useful lies in translators which add symbols to the symbol table which must be examined by a lexical action. If a lookahead fetch occurs too fast, the lexical action may miss the introduction of a symbol into the symbol table.

This feature is a bit flaky for the moment—`LA(`*i*`)` and `LATEXT(`*i*`)` will generally not have the same values as when the `-gk` option is not used (for *k*>=2). Use attributes to access token values; the lookahead buffer is not really a user object. If you insist upon accessing the lookahead buffer, use `LA(0)` and `LATEXT(0)`, which typically access the last token matched and last text matched respectively; this is distinguished from `LA(1)`, which means the next token of lookahead. Accessing the next token of lookahead is invalid because it will not be fetched from the input stream until needed (just before the next decision or match).

1.8. Tutorials Available

With release 1.06, we are distributing both a beginning and advanced tutorial. They have not been thoroughly ‘‘debugged’’ much are much better than nothing:

Beginning

This tutorial introduces the basic functionality of PCCTS by example. The user need not be familiar with parsing theory or other compiler tools, but any familiarity reduces the learning curve substantially.

Advanced

Constructing a translator can be viewed as an iterative refinement process moving from language recognition to intermediate-form transformation. This tutorial presents one possible sequence of refinements. It uses as many features of PCCTS as is reasonable without regards to optimality. It develops a compiler for a simple string manipulation language called *sc*. The resulting compiler generates code for a simple stack machine.

1.9. Error Messages for $k>1$

Previous versions of PCCTS did not handle error message correctly for $k>1$. For example, with two tokens of lookahead and the following grammar:

```
a : "A" "B" "D"
  | "A" "C" "E"
  ;
```

an incorrect input of A D D would yield:

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

which is wrong (and incredibly confusing). The new error mechanism generates the following error message upon the same incorrect input:

```
line 1: syntax error at "A D"; "D" not in { B C }
```

which is infinitely superior. Unfortunately, situations may arise when even this method will give an invalid message. This may occur when alternatives have lookahead sequences which are permutations of the same tokens.

The definition of the standard error reporting function, `zzsyn()` has been modified. The parameter list is now:

```
void
zzsyn(char *text,
      int tok,
      char *egroup,
      unsigned *eset,
      int etok,
      int k,
      char *bad_text);
```

Users can ignore this as it is transparent to them; unless, of course, the standard error reporting must be modified. In addition, `zzFAIL` is now a function rather than a macro.

1.10. Trace Facility has Exit Macro

Previously, only an entry trace macro was inserted in parsers when the `-gd` ANTLR option was used. An exit macro has been defined which resulted in `zzTRACE` becoming `zzTRACEIN`. Also, a default trace macro prints out ''`enter rule` *rule*'' if no default trace macros are defined. To define your own, the macro definitions must appear in the `#header` action. As before, the sole argument to the trace routines is a string representing the rule which has been entered or is about to be exited.

1.11. Resource Limitation

Occasionally, ANTLR is unable to analyze a grammar submitted by the user. This rare situation can only occur when the grammar is large and the amount of lookahead is greater than one. A nonlinear analysis algorithm is used by PCCTS to handle the general case of *LL(k)* parsing. The average complexity of analysis, however, is near linear due to some fancy footwork in the implementation which reduces the number of calls to the full *LL(k)* algorithm.

To avoid the situation where ANTLR takes 23 hours of CPU time and then runs out of virtual memory, use the `-rl` *n* resource limit option where *n* is the maximum number of tree nodes to be used by the analysis algorithm. An error message will be displayed, if this limit is reached, which indicates which grammar construct was being analyzed when ANTLR hit a nonlinearity. Use this option if ANTLR seems to go off to lunch and your disk start swapping; try *n*=10000 to start. Once the offending construct has been identified, try to remove the ambiguity that ANTLR was trying to overcome with large lookahead analysis. Future versions of PCCTS may incorporate a known algorithm that does not exhibit this exponential behavior.

1.12. Rule Prefix Option

An ANTLR option has been added that prefixes all functions corresponding to rules with a prefix. This can be used to provide symbol hiding in your project to isolate the parser. It can also be used to allow rule names that correspond to C keywords such as if and `typedef`.

1.13. Standard PCCTS Header

Two new ANTLR options have been added that control the creation of a standard PCCTS header file — `stdpccts.h`. Option `-gh` instructs ANTLR to create a file, `stdpccts.h` unless `-fh` is used, which contains all header information needed by non-PCCTS generated files that want to access PCCTS symbols. For example, it indicates the *k* of the parser, whether trees are being constructed, whether lookahead is to be delayed, and indicates what the user specified in the `#header` action in the grammar file. Previously, the user had to manually construct this information from the grammar file in order to place the information in a non-PCCTS C file. The `-fh` option is merely used to rename `stdpccts.h`.

1.14. Doubly-Linked AST's

A new function is available in `ast.c` which will doubly-link any tree that is passed in. To use this option, the user must define zzAST_DOUBLE in the `#header` directive or on the command-line of the C compile. This defines `left` and `up` fields automatically in the AST node typedef. ANTLR generated parsers, normally, only construct singly-linked trees. The fields can be filled in via code similar to the following:

```
#header <<
#define AST_FIELDS    user-fields;
>>

<<
main()
{
    AST *root = NULL;

    ANTLR(start(&root), stdin);
    zzdouble_link(root, NULL, NULL);
}
```

where the function is defined as:

```
zzdouble_link(AST *t, AST *left, AST *up);
```

1.15. C++ Compatible Parsers

PCCTS parsers may now be compiled with C++ compilers; i.e. the output is more ANSI C-like than before. It has been successfully compiled with GCC 2.2, but not with GCC 1.37. We do not guarantee anything. To be safe, use the `-ga` option so that PCCTS generates ANSI-style prototypes for functions generated from rules. As a simple example, consider:

```
#header <<
#include "charbuf.h"
/* stdio.h is included, by default, but doesn't seem to bother stream.h */
#include <stream.h>
#include <stdlib.h>
>>

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

<<
main()
{
        ANTLR(a(), stdin);
        cout << "end of C++ test\n";
}
>>

a       :       "A" "B" << cout << $1.text << $2.text << "\n"; >>
        ;
```

which does not do much:

```
% t
A B
AB
end of C++ test
%
```

but it does compile with G++ 2.2 except that a warning is generated concerning `strncpy()` not being declared before use. This is trivial to fix, of course — by modifying the `charbuf.h` file. We compiled this with:

```
antlr -k 2 -gk -ga t.g
Antlr parser generator   Version 1.06   1989-1992
dlg -C2 -i parser.dlg scan.c
dlg  Version 1.06   1989-1992
g++ -I. -I../1.06/h -g -c scan.c
g++ -I. -I../1.06/h -g -c t.c
t.c: In function ‘struct Attrib zzconstr_attr (int, char *)’:
t.c:19: warning: implicit declaration of function ‘strncpy’
g++ -I. -I../1.06/h -g -c err.c
g++ -o t -I. -I../1.06/h -g scan.o t.o err.o
```

We anticipate a rewrite to be more C++ sometime in the future.

2. Acknowledgements

We acknowledge Dan Lawrence of MDBS for the new error reporting facility concerning greater than one token of lookahead; Dana Hoggatt, also of MDBS, suggested the rule prefix idea (`-gp` option) and beta tested 1.06. We thank Ed Harfmann of MDBS for creating the `makefile.os2` files and porting it to the PC. We acknowledge the following beta testers for 1.06 (alphabetically): Thomas Buehlman (buehlman@iwf.mabp.ethz.ch), Peter Dahl (dahl@everest.ee.umn.edu), Chris Song (dsong@ncsa.uiuc.edu), Ariel Tamches (tamches@cs.UMD.EDU). We reference Russell Quong (quong@ecn.purdue.edu) of Purdue EE for his work with us on defining and refining predicates. Ariel Tamches (tamches@cs.UMD.EDU) deserves attention for hacking on the particulars of the alpha-release predicates.

3. Machine Compatibility

PCCTS Version 1.06 has been tested on the following platforms:

Sun 3/60

Sun SPARC I, II

Encore Multimax running Umax 4.3

Sun sparcstation IPX

NeXTstation

Decstation 3100 running Ultrix 4.2

DEC 5000

Linux on 386PC

Microsoft C 6.0 on PC OS/2, DOS

CSET/2 C compiler on PC OS/2

IBM RS6000

MIPS r2000

4. Incompatibilities

Due to the bug fixes in 1.06, ‘‘new’’ ambiguities may appear in your grammar. These were always there—ANTLR simply did not find them. The analysis is more correct.

Calls to `zzTRACE` are no longer generated by the `-gd` option. Now, `zzTRACEIN`, `zzTRACEOUT` are called at the beginning and end of functions, respectively.

The way in which PCCTS translates actions has been changed; before they were parsed with a C function, now the `#lexclass` facility is being used. Some differences in translation may be discovered; e.g. a character may need to be escaped with `\` whereas before the simple character was sufficient.

The user should no longer set the next token of lookahead or the text of the next token in the lexical analyzer using `LA(1)` and `LATEXT(1)`. This is incompatible with the `-gk` option; hence, `NLA` and `NLATEXT` should be used instead where the `N` means ‘‘next’’.

The `-ga` does not generate anything different as the code generator now dumps both K&R and ANSI with `#ifdef`’s around the ANSI code.

Previously, no prototype was given when `-ga` was off. Now, prototypes are always generated (with the appropriated `#ifdef`’s). These prototypes can conflict with the outside environment if the rule names are things like `if` and `stat` (which is a system call). Use the `-gp` *prefix* option to prefix all functions corresponding to rules with a string of your choice.

5. Future Work:

Predicates are still under development. We expect an enhanced version of PCCTS that computes context and hoists more aggressively.

Often a grammar construct cannot be left factored to remove an ambiguity. This typically arises in the situation that the common prefix can be arbitrarily long. Fortunately, input is typically finite and one could scan past these constructs given enough lookahead. This is not the same thing as backtracking as the parser never backs up; it simply looks ahead really far to make a decision. This can easily be handled with a predicate of the form:

`<<is_this_found_ahead(`*context*`)>>?`

which would look ahead in the lookahead buffer to see if the *context* occurred within some finite number of tokens. This concept is very similar to LR-Regular (LRR) parsers for those familiar

with parsing theory. Note that this is a very fast, cheap way to get something resembling the power of backtracking.

Attribute names are expected to be enhanced. For example, instead of $i, $*token_name* could be used:

```
a : WORD TYPE <<printf("%s %s\n", $WORD, $TYPE);>>
  ;
```

We expect to have a graphical user interface to PCCTS sometime in the future which allows entry of grammars using syntax diagram notation. The interface is expected to run under X Windows.

We anticipate a version that supports object-oriented programming and generates C++ instead of ANSI C. For the moment, PCCTS is compatible with C++, but future versions will support C++.

Future versions, both C and C++, will be able to refer to PCCTS symbols by `pccts.`*symbol* instead of `zz`*symbol*. E.g. `zzskip()` will become `pccts.skip()`.

DLG will soon use lookahead of its own to allow the recognition of more complicated expressions; specifically, those which have left substrings in common with other regular expressions.

We expect future versions of PCCTS to dump grammar analysis and parser construction statistics such as how many rules required 1 token of lookahead, how many ambiguities etc...