PCCTS Reference Manual

Version 1.00

T. J. Parr, H. G. Dietz, and W. E. Cohen

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
August 1991
pccts@ecn.purdue.edu

[*Note: Latest release is 1.10; please see release notes for 1.06 and 1.10*]

This document describes the structure and use of the Purdue Compiler-Construction Tool Set (PCCTS), a set of public domain software tools designed to facilitate the implementation of compilers and other translation systems.

Distribution of PCCTS is managed by an email server that reads the `Subject:` line of each email sent to `pccts@ecn.purdue.edu`. A missing or undecipherable `Subject:` line will cause the system to reply with email explaining how to use the server. Full C source code for PCCTS, various examples, and on-line documentation are available.

PCCTS was developed within the School of Electrical Engineering at Purdue University, supported in part by NSF award number 8624385A3-CDR. As a public domain software release, Purdue University and the authors provide the system strictly on an ‘‘as-is’’ basis, without warranty or liability. Despite this, bug reports and fixes are welcome.

1. Introduction

This document describes structure and use of the Purdue Compiler-Construction Tool Set (PCCTS). In many ways, PCCTS is similar to a highly integrated version of YACC [Joh78] and LEX [Les75]; where ANTLR (ANother Tool for Language Recognition) corresponds to YACC and DLG (DFA-based Lexical analyzer Generator) functions like LEX. However, PCCTS has many additional features which make it easier to use for a wide range of translation problems.

PCCTS grammars contain specifications for lexical and syntactic analysis, intermediate-form construction and error reporting. Future releases will include an integrated code-generator generator to support code-generation and other intermediate-form translations. Rules may employ Extended BNF (EBNF) grammar constructs and may define parameters, return values and local variables. Languages described in PCCTS are recognized via Strong LL(k) parsers constructed in pure, human-readable, C code.

The Version 1.00 Release PCCTS package is considered public-domain and includes the following items:

- C source code for `antlr` — parser generator.
- C source code for `dlg` — DFA-based lexical analyzer generator.
- C source for a symbol table manager with arbitrary scoping capabilities.
- Grammars for ISO PASCAL and ANSI C with symbol table management.
- C source for `rexpr(char *expr, char *s)` which answers whether `s` is in the language (regular expression) described by `expr`.
- On-line documents — PCCTS Version 1.00 Reference Manual

PCCTS was developed within the School of Electrical Engineering at Purdue University to aid in constructing various prototype and specialized compilers. After using the system for over a year, we decided that it was mature enough to warrant release to a *few* test sites. Hence, in Spring 1990, an experimental version of the system, known as PCCTS β, was made available via anonymous `ftp`. Although the system was only announced via two short postings in network newsgroups, by January 1991, there were over 450 known user sites worldwide — and we were hopelessly behind in responding to requests for hardcopy manuals.

Now, only a year older but much wiser, we are finally ready for a full release of PCCTS — Version 1.00. Experience has proven that we are not able to deal with high-quantity hardcopy requests; instead, the basic documentation (this document) is being printed by SIGPLAN Notices and all documentation is available on-line.

The rest of this section describes the overall characteristics of PCCTS including its programming interface, translation to C code and input file format. Also, an example is presented to illustrate the development of parsers using PCCTS.

1.1. PCCTS programming interface

PCCTS accepts language descriptions in the form of a set of grammatical rules and token/lexeme definitions. Significant emphasis was placed on making the PCCTS programming interface simple and flexible. Much of the notation was derived from previous language tools and/or programming languages in order to bring a sense of familiarity and to attenuate the learning curve. For example, attribute notation ($*i*) and grammar syntax were derived mainly from YACC, though they have been augmented for use in the EBNF environment. Rules may define return types, parameter lists and local variables just as in a programming language. A mechanism for describing abstract-syntax-trees is also provided.

PCCTS Grammars tend to be smaller and more readable than grammars written for other parser-generators because of the EBNF notation and because precedence rules are defined inherently within the grammar rather than delineated via pseudo-instructions (e.g. `%left`, `%right` in YACC). In addition, PCCTS translators (parser plus actions) are generally easier to develop because of the extensive programming support.

For example, init-actions allow the user to define local stack-based variables or execute a piece of C initialization code. Often, a distinct copy of a variable is required for each invocation of a rule. This requires the overhead of a software stack when using other language tools. PCCTS constructs a function for each rule which allows each rule invocation to automatically create its own copy of the user variables on the hardware stack.

PCCTS parsers are constructed in pure, human-readable, C code. As a result, standard debugging tools can be used to trace and debug PCCTS parsers. Breakpoints can be set so that parser execution stops before or after certain grammar fragments of interest have been recognized. PCCTS parsers consist of `if`, and `while` statements which can be easily traced by a human observer.

PCCTS supports intermediate-form (such as expression-trees) construction via a flexible abstract-syntax-tree (AST) mechanism which allows trees to be built explicitly or automatically. The user explicitly creates trees via a LISP-like tree constructor or directs the automatic tree construction facility via simple grammar directives. AST tree nodes are user-defined and are generally a function of attributes or terminals. A default transformation from attributes/terminals ($-variables) to AST nodes can be specified. Alternatively, each tree node can be defined explicitly via an AST node constructor.

1.2. PCCTS information flow

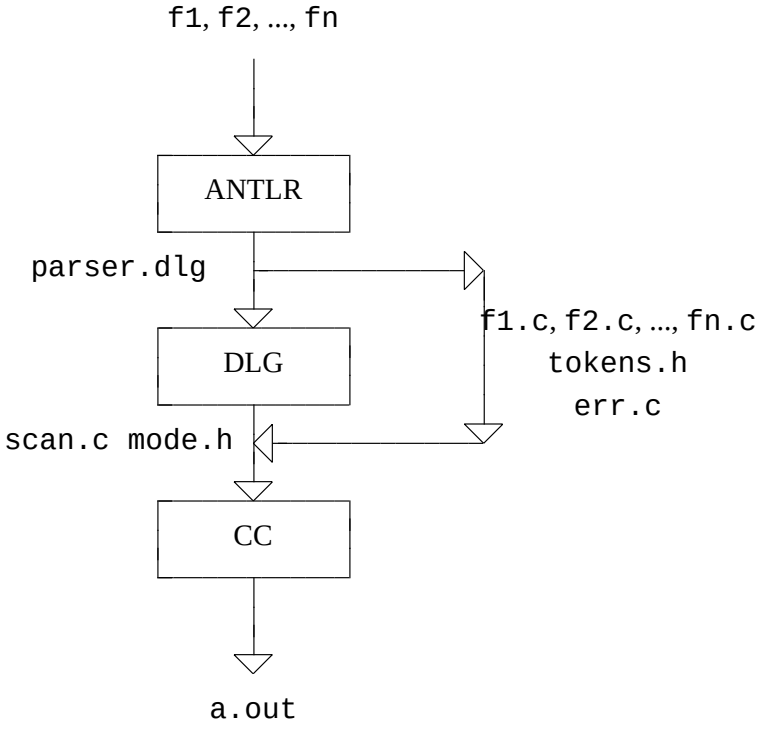
PCCTS currently consists of ANTLR (ANother Tool for Language Recognition) and DLG (DFA-based lexical analyzer generator) which construct parsers and lexical analyzers respectively.

ANTLR accepts as input a complete language description including rules for both lexical and syntactic analysis and produces a C program that recognizes sentences in that language. A C file is constructed for each grammar description file. Also, three additional files — `err.c`,

`tokens.h` and `parser.dlg` (all of which can be renamed) are generated as support files. `err.c` contains error information and token class definitions. `tokens.h` is a list of `#define`'s representing all of the token names defined or referenced in the grammar description files as well as function prototypes for rules that have return types. `parser.dlg` is a specification of the lexical analyzer generator derived from the user's language description and is translated to C by DLG.

DLG creates an automaton that breaks up the input character stream into tokens used by the ANTLR-generated parser and it creates `mode.h` which is a list of `#define`'s corresponding to the separate automatons defined by your grammar.

The following is a graphical representation of the information flow during translation from parser description to executable parser.



FILE	DESCRIPTION
`f1` ... `fn`	Input ANTLR grammar description files
`f1.c` ... `fn.c`	C code generated by ANTLR
`tokens.h`	List of token names found in `f1` ... `fn`
`err.c`	File containing error sets and error support routines
`parser.dlg`	Scanner description prepared for DLG
`scan.c`	Actual scanner written in C
`mode.h`	Definitions for lexical modes (`#lexclass`'s)
`a.out`	Final executable parser

1.3. PCCTS file(s) format

PCCTS descriptions adhere to a fairly flexible format and can be spread out over many files. The actual grammar for PCCTS input can be defined in itself as:

```
grammar     :   { "#header" Action }
                ( Action | tokenDef | eclassDef )*
                ( rule | tokenDef | eclassDef )+
                ( Action | tokenDef | eclassDef )*
                Eof
            ;
```

where `Action` is a user action (C code), `tokenDef` is a `#token` definition and `eclassDef` is an `#errclass` definition. Lexical actions found in a language description are passed on to the `parser.dlg` file. The `( )*` and `( )+` EBNF operators indicate 0 or more and 1 or more respectively. `{ }` implies that the items within are optional. Rule `grammar` indicates that a PCCTS file has a `#header` action first (if at all) followed by actions, token definitions and error class definitions with the except that actions may not appear between rules. Also, at least one rule must be defined. There is no start symbol specification since any rule can be invoked first.

The minimum required user C code is a main program that initializes the PCCTS-generated parser and calls the starting rule. The lexical analyzer is derived automatically (from the user's language description) by ANTLR and a user routine is not necessary to supply tokens to the parser. The main program does not need to be located in a PCCTS grammar file.

ANTLR descriptions may be broken up into many different files, but the sequence mentioned above in `grammar` must be maintained. For example, if file `f1.g` contained

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

<< main() { ANTLR(start(), stdin); } >>
```

and file `f2.g` contained

```
start   :   "begin" VAR "=" NUM ";" "end" "." "@" ;
```

and file `f3.g` contained

```
#token VAR "[a-z]+"
#token NUM "[0-9]+"
```

the correct ANTLR invocation would be

```
antlr f1.g f2.g f3.g
```

Note that the order of files `f2.g` and `f3.g` could be switched because a `rule` is either a `#token` or an actual rule definition. In this case, to comply with ANTLR's grammar, the only restriction is that file `f1.g` must be mentioned first on the command line.

Other files may be included into the parser files generated by ANTLR via actions containing a `#include` directive. For example,

```
<<#include "support_code.h">>
```

If a file must be included in all parser files generated by ANTLR, the `#include` directive must be placed in the `#header` action. In other words,

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

Note that `#include`'s can be used to define any ANTLR object (`Attrib`, `AST`, etc...) by placing it in the `#header` action.

1.4. Makefile template

The template presented in this section can be used to construct makefiles for single-file grammars. The order of execution will follow that illustrated in section 1.2.

```
#
# Makefile for a single-file grammar
#
ANTLR_H= path to antlr.h, dlgdef.h, dlgauto.h, standard attribute defs
CFLAGS= -I$(ANTLR_H) -DLL_K = $(K)
AFLAGS=
GRM=user_grammar_file_name
SRC=scan.c $(GRM).c err.c
OBJ=scan.o $(GRM).o err.o
K=number_of_tokens_of_lookahead

# build an executable with same name as grammar file minus .g
proj: $(OBJ) $(SRC)
    cc -o $(GRM) $(OBJ)

# how to build parser files from grammar files
$(GRM).c parser.dlg : $(GRM).g
    antlr $(AFLAGS) $(GRM).g

# how to build scanner from lex description built by ANTLR
scan.c : parser.dlg
    dlg -C2 parser.dlg scan.c
```

1.5. PCCTS component command-line options

The PCCTS system is composed of ANTLR and DLG both of which have command-line options. Since each component is executed independently, each may be invoked with different options which are summarized in this section.

ANTLR accepts the following command-line options

`-cr` Generate cross reference of all rules. The default is FALSE.

`-e1` Only the first 3 tokens of any ambiguity set are displayed upon warning/error. i.e. `{ A B C ...}`.

`-e2` Every token in an ambiguity set is displayed upon warning/error.

`-e3` Error levels 1 and 2 (-e1, -e2) are sometimes misleading for the sake of brevity. For instance, when a block is ambiguous upon the sequences

```
A B
C D
```

only { A C }, { B D } is displayed when in fact `A D` and `C B` are not ambiguous. The `-e3` option makes ANTLR generate permutation trees in LISP-like notation

```
(root child1 child2 ... childn)
```

The following grammar illustrates the difference

```
a   :   (A B|C D|A D|C B)
    |   (A D|C B)
    ;
```

ANTLR with error level 1 or 2 reports

```
Antlr parser generator   Version 1.00   1989, 1990, 1991
"t.g", line 1: warning: alts 1 and 2 rule ambiguous upon { A C }, { B D }
```

ANTLR with error level 3 reports

```
Antlr parser generator   Version 1.00   1989, 1990, 1991
"t.g", line 4: warning: alts 1 and 2 rule ambiguous upon ( C B ) ( A D )
```

which indicates that rule `a` is only ambiguous upon the two sequences `C B` and `A D`.

`-fe` *fname*
Rename the file where error information, token sets, and `zztokens[]` are placed. Default is `err.c`.

`-fl` *fname*
Rename the file where ANTLR places the lexical description. Default is `parser.dlg`.

`-ft` *fname*
Rename the file where ANTLR places the `#define`'s for labeled constants and the function prototypes. Default is `tokens.h`.

`-ga` Generate ANSI prototypes for functions constructed from rules. Default is not to generate ANSI-style prototypes.

`-gt` Generate code for Abstract-Syntax-Trees. Default is not to generate trees.

`-gc` Do not generate output parser code or lexical description (implies `-gx`). This option can be used to check a grammar for ambiguities. Default is to generate parsers.

`-ge` Generate an error class for each non-terminal of the form

```
#errclass Rule { rule }
```

Default is not to generate the classes.

`-gs` When a token in the current look-ahead needs to be compared against two or more tokens, code for set membership is generated rather than a set of integer comparisons. This renders the parsers more efficient, but much more difficult to read for a human. When debugging an ANTLR parser with a source level debugger, this command-line option should be used to create more readable parsers. The default is to generate sets for any token expression list with two or more members.

`-gd` Generate debugging code; trace rule invocation. This command-line option forces ANTLR to generate code that calls `zzTRACE()` with the rule name in which it appears as an

argument. `zzTRACE()` can be a macro or a function. The default is not to generate debugging code.

`-gl` Generate line information in PCCTS-generated parser that associates a line in the user's grammar with a line in the C code. This is useful when the C compiler reports an error in a user action. It will report an error in the grammar file rather than an error in the C file. This dumps line information like the C preprocessor.

`-gx` Do not generate a lexical description (dlg-related files). This is useful if you are positive nothing has changed in the lexical portion of your language description. For example, if you simply change an action in one of your rules, nothing has changed lexically and you need not run DLG to recreate the lexical analyzer. The default is to generate a description.

`-k` *n*

Set k, the number of look-ahead tokens, to *n*. Default is 1.

`-p` Print out the grammar to `stdout` without actions. This option is useful for documentation purposes or to simply make viewing/understanding your grammar easier. The default is not to generate a grammar printout.

`-pa` This option is identical to the -p option accept that the grammar is annotated with the results of grammar analysis. For example, the set of all tokens that can be matched at k tokens into the future are displayed after each point in your grammar where a decision has to be made. ANTLR prints out only the sets of tokens that would be needed to construct a parser. Rule `a` below illustrates the annotations.

```
#header <<;>>
a   :   (A B | A D)
    |   Q
    ;
```

Executing ANTLR with

```
antlr -k 2 -pa t.g
```

yields (cleaned up slightly)

```
Antlr parser generator   Version 1.00   1989, 1990, 1991

a   :   ( A B       /* [1] { A }, { B } */
        | A D       /* [2] { A }, { D } */
        )           /* [1] { A } */
        | Q         /* [2] { Q } */
    ;
```

The first alternative of rule `a` has a subrule with two alternatives which requires a parsing decision. Since both alternatives start with an `A`, another token of look-ahead must be examined to determine which alternative to choose. In this case, the second token in each alternative allows us to choose. Therefore, two sets of tokens (singleton sets here) are

printed out for each alternative in the subrule. The alternatives of rule `a` can be matched with the knowledge of only the first token and so only one set is printed. The default is not to generate a grammar printout with annotations.

DLG has the following command-line options.

`–C`*n*

Specify level of character class compression. 0 means no compression, 1 removes unused characters from DFA states, and 2 specifies that DLG is to combine characters into equivalence classes. It is suggested that `-C2` be used since it will result in much smaller output file and does not require much additional processing time.

`–ga` Produce ANSI C compatible code.

`–m` *filename*

Uses *filename* rather than `mode.h` for the lexical mode definition file.

`–i` Attempts to build a lexical scanner for interactive programs. The non-interactive scanner always looks one character past the end of a token. This look-ahead is avoided whenever possible in the interactive version so that the pattern is recognized without any additional characters being read. Look-ahead is always used when necessary to correctly match input tokens.

1.6. Introductory example

As a simple concrete example of the PCCTS system, consider the following translator which accepts a date in the typical American format of *mm/dd/yy* and prints to `stdout` the same date, but in the European format of *dd/mm/yy*.

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

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

#token "[\t\ \n]"     << zzskip(); >>          /* Ignore White */
#token Digits2 "[0-9][0-9]"                    /* Define mm, dd or yy */

date    :   Digits2 "/" Digits2 "/" Digits2 "@"
            <<printf("%02d/%02d/%02d\n", $3, $1, $5);>>
        ;
```

This example is complete in the sense that PCCTS can generate all C files necessary to lexically and syntactically analyze a date and convert it to the European format. An executable can be obtained by performing the following sequence of commands (assuming the above grammar is in file `date.g` and `antlr.h`, the standard ANTLR header, is in the current directory).

```
antlr date.g
dlg parser.dlg scan.c
cc date.c scan.c err.c
```

Executing the file `a.out` (on UNIX [Trademark AT&T] systems) will allow the user to type in a date and have the European version printed out (after end of file is typed).

Rule `date` is defined to be a sequence of two digits followed by a `/` followed by a sequence of 2 digits, and so on, terminated by the end of the input file (represented by `"@"`). Note that rule names begin with a lower-case letter. The action

```
<<printf("%02d/%02d/%02d\n", $3, $1, $5);>>
```

contains C code that is executed after all elements in rule `date` have been recognized. C code for the action is delimited by `<<` and `>>` (European quotes). Because the `>>` delimiter otherwise would be confused with C's right-shift operator within the C code of an action, right-shift is written as `\>\>`. Similarly, the `$` and `#` characters normally have special meanings within the C code of a PCCTS action; hence, the literal character `$` in the C code would be written as `\$` and `#` would be written as `\#`.

Although the placement of an action within a grammar determines when that action will be applied, it must also be possible for the action to be a function of the text which has been recognized. For example, PCCTS automatically makes `$1` in rule `date` have a value, or attribute, which is a function of the text matched by the first occurrence of `Digits2`. In general, the attribute associated with the i^{th} item in a rule is called `$`*i*. Both the type of the attributes and the mapping from text matched into its attribute value is specified by the user. The directive

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

includes code that makes the values of ‘‘$ variables’’ be of the C type `int` with values defined by applying the C function `atoi()` to the text matched.

Each item in the rules refers either to the text matched by a simple pattern (a regular expression) or to the collection of items matched by a rule (represented by the rule's name). Simple patterns can be stated directly, such as `"/"` in the example, or can be given names using the `#token` directive. The token definitions

```
#token "[\t\ \n]"     << zzskip(); >>          /* Ignore White */
#token Digits2 "[0-9][0-9]"                    /* Define mm, dd or yy */
```

cause tab (`\t`), space (`\ `), and newline (`\n`) characters between the other tokens to be ignored, and allow use of the name `Digits2` as a shorthand for `"[0-9][0-9]"`.

Since the above is meant to be a stand-alone complete example and PCCTS does not generate a C `main` function, the user must provide a `main`. The `main` also specifies which rule is to be used to recognize the input and where that input will come from. The code

```
ANTLR(date(), stdin);
```

specifies that the `date` rule will be used to recognize input taken from `stdin` (the standard input stream).

Notice that, although there is no specification of how incorrect input should be handled by the generated program, PCCTS will automatically generate appropriate error messages and attempt to recover from the errors.

The above example should give the reader a "feel" for how PCCTS can be used, however, the system supports much more than the example uses. This document is organized into six major sections: lexical analysis, syntactic analysis, attribute handling, error detection, miscellaneous items and PCCTS history.

2. Lexical analysis and token definition

The task of language recognition is conveniently broken down into two phases — lexical and syntactic analysis. Each phase is considered a separate operation and, therefore, most language tools require separate specifications. For example, Coco-2 [DoP90] requires the user to create separate, but consistent, specifications of character sets, keywords and tokens (and even non-terminals). In contrast, PCCTS derives all this information from a single description. This both ensures that the information is consistent and makes it easier to view a PCCTS language specification as a whole.

This section of the manual describes general lexical issues associated with PCCTS' integrated description including lexical action definitions, regular expression specification, and resolution of ambiguities. Also, more specialized issues such as multiple automatons, interactive scanners, token positional information and input character supply are presented.

2.1. Token Labeling and token actions

Tokens are defined either explicitly with `#token` or implicitly by using them as rule elements. Implicitly defined tokens can be either regular expressions (non-labeled tokens) or token names (labeled). Token names begin with an upper-case letter (rules begin with a lower-case letter). More than one occurrence of the same regular expression in a grammar description produces a single regular expression in `parser.dlg` and is assigned one token number. Quoted and non-quoted tokens that refer to the same lexical object (expression) may be used interchangeably. Token names that are referenced, but not attached to a regular expression are assigned a number and given a `#define` definition in `tokens.h`. It is not necessary to label any tokens in PCCTS. However, all tokens that you wish to explicitly refer to in an action, must be declared with a `#token` instruction. From PCCTS's point of view, it merely needs to know that a word is a token (as opposed to a non-terminal). Its value is irrelevant at the symbolic level of PCCTS's meta-language.

The user may introduce tokens, lexical/token actions, and token labels via the `#token` instruction. Specifically,

- Simply declare a token for use in a user action:

```
#token VAR
```

- Associate a token with a regular expression and, optionally, an action:

```
#token VAR "[a-zA-Z][a-zA-Z0-9]*"
#token Eof "@" << printf("Eof Found\n"); >>
```

- Specify what must occur upon a regular expression:

```
#token "[0-9]+"  << printf("Found int %s\n", zzlextext); >>
```

Token actions may employ two functions, `zzreplchar()` and `zzreplstr()`, to replace the text for the regular expression matched on the input stream. For example, if `\n` is found in a string for C, it needs to be converted to the actual newline character. Strings in C can be handled in the following manner.

```
/* START lexclass is in effect */
#token "\""           <<zzmode(STRINGS); zzmore();>>

#lexclass STRINGS   /* define a new automaton */
#token STRING "\""  <<zzmode(START);>>
#token "\\\""       <<zzmore();>>
#token "\\n"        <<zzreplchar('\n'); zzmore();>>
#token "\\r"        <<zzreplchar('\r'); zzmore();>>
#token "\\t"        <<zzreplchar('\t'); zzmore();>>
#token "\\[1-9][0-9]*"
                    <<zzreplchar((char)strtol(zzbegexpr,NULL,10)); zzmore();>>
#token "\\0[0-7]*"  <<zzreplchar((char)strtol(zzbegexpr,NULL,8)); zzmore();>>
#token "\\0x[0-9a-fA-F]+"
                     <<zzreplchar((char)strtol(zzbegexpr,NULL,16)); zzmore();>>
#token "\\~[\n\r]"  <<zzmore();>>
#token "[\n\r]"     <<zzline++; zzmore(); /* print warning about \n in str */>>
#token "~[\"\n\r\\]+" <<zzmore();>>

#lexclass START
```

Note the use of `zzbegexpr` which points to the beginning of the text matched for the currently recognized regular expression. `zzendexpr` is also available and points to the end of the expression. The text always has a `'\0'` on the end and `zzendexpr` points to the character just before.

There is a big difference between a token action and a normal action found outside of the `#token` instructions. An action not included as part of one of the lexical commands is considered a syntactic action and is placed in the C parser file associated with that grammar file.

If a grammar action follows a `#token` definition with no associated lexical action, the `#token` instruction will assume that the action is really meant as a lexical action. This is sort of analogous to the dangling-else-clause dilemma. Example:

```
#token VAR "[a-zA-Z][a-zA-Z0-9]*"

<<
    /*
     * This will be associated with the #token even though it is
     * probably a grammar action not a lexical action
     */
>>
```

This ambiguity can be solved by appending a null action to the `#token`:

```
#token VAR "[a-zA-Z][a-zA-Z0-9]*" << ; >>
```

2.2. Lexical actions via `#lexaction`

Often, it is convenient to have a section of user C code placed in the lexical analyzer constructed automatically by DLG (PCCTS's DFA-based lexical analyzer generator). Normally, actions not associated with a `#token` pseudo-op or embedded within a rule are placed in the parser generated by ANTLR. However, preceding an action appearing outside of any rule, with the `#lexaction` pseudo-op directs the C code to the lexical analyzer. For example,

```
<< /* a normal action outside of the rules */ >>

#lexaction
    << /* this action is inserted into the lexical
        * analyzer created by DLG
        */
    >>
```

All `#lexaction` actions are collected and placed as a group into the C file where the lexer resides. Typically, this code consists of functions or variable declarations needed by `#token` actions.

2.3. Multiple lexical classes

ANTLR parsers employ DFA's (Deterministic Finite Automatons) created by DLG to match tokens found on the character input stream. More than one automaton (lexical class) may be defined in PCCTS. Multiple scanners are useful in two ways. First, more than one grammar can be described within the same PCCTS input file(s). Second, multiple automatons can be used to recognize tokens that seriously conflict with other regular expressions within the same lexical analyzer (e.g. comments, quoted-strings, etc...).

Actions attached to regular expressions (which are executed when that expression has been matched on the input stream) may switch from one lexical analyzer to another. Each analyzer

(lex class) has a label which is used to enter that automaton. A predefined lexical class called `START` is in effect from the beginning of the PCCTS description until the user issues a `#lexclass` directive or the end of the description is found.

When more than one lexical class is defined, it is possible to have the same regular expression and the same token label defined in multiple automatons. Regular expressions found in more than one automaton are given different token numbers, but token labels are unique across lexical class boundaries. For instance,

```
#lexclass A
#token LABEL "expr1"

#lexclass B
#token LABEL "expr2"
```

In this case, `LABEL` is the same token number (`#define` in C) for both `"expr1"` and `"expr2"`. A reference to `LABEL` within a rule can be matched by two different regular expressions depending on which automaton is currently active.

Hence, the `#lexclass` directive marks the start of a new set of lexical definitions. Rules found after a `#lexclass` can only use tokens defined within that class — i.e. all tokens defined until the next `#lexclass` or the end of the PCCTS description, whichever comes first. Any regular expressions used explicity in these rules are placed into the current lexical class. Since the default automaton, `START`, is active upon parser startup, the start rule must be defined within the boundaries of the `START` automaton. Typically, a multiple-automaton PCCTS grammar will begin with

```
#lexclass START
```

immediately before the rule definitions to insure that the rules use the token definitions in the "main" automaton.

Tokens are given sequential token numbers across all lexical classes so that no conflicts arise. This also allows the user to reference `zztokens[`*token_num*`]` (which is a string representing the label or regular expression defined in the grammar) regardless of which class *token_num* is defined in.

2.3.1. Multiple grammars, multiple lexical analyzers

Different grammars will generally require separate lexical analyzers to break up the input stream into tokens. What may be a keyword in one language, may be a simple variable in another. The `#lexclass` PCCTS meta-op is used to group tokens into different lexical analyzers. For example, to separate two grammars into two lexical classes,

```
#lexclass GRAMMAR1
```

rules for grammar1

```
#lexclass GRAMMAR2
```

rules for grammar2

All tokens found beyond the `#lexclass` meta-op will be considered of that class.

2.3.2. Single grammar, multiple lexical analyzers

Again the `#lexclass` meta-op is used to indicate the beginning of a new lexical analyzer. For instance to define C-style comments, the following will suffice:

```
/* lexclass START is in effect by default */
/* switch to COMMENT automaton */
#token "/\*"         <<zzmode(COMMENT); zzskip();>>
 ...
#lexclass COMMENT
#token "\*/"         <<zzmode(START); zzskip();>>  /* switch back */
#token "~[\*]*"      <<zzskip();>>  /* ignore all stuff inside */
#token "\*~[/]"       <<zzskip();>>
 ...
```

2.4. Handling end of input

There is only one automatically defined regular expression — end-of-file which is represented by the regular expression `@`. An implicit

```
#token "@"
```

is executed at the start of each lexical class (`#lexclass`). A label may be attached via the following in the user's grammar:

```
#token Eof "@"
```

A user could then refer to end-of-file as the `#define` constant `Eof`.

2.5. Token order and lexical ambiguities

The order in which regular expressions are found in the grammar description file(s) is significant. When the input stream contains a sequence of characters that match more than one regular expression, (e.g. one regular expression is a subset of another) DLG is confronted with a dilemma. DLG does not know which regular expression to match, hence, it does not know which action should be performed. To resolve the ambiguity, DLG assumes that the regular expression which is defined earliest in the DLG input should take precedence over later definitions. Therefore, tokens that are special cases of other regular expressions should be defined before the more

general regular expressions.

ANTLR automatically constructs the input to DLG by copying regular expressions into the file `parser.dlg`. These definitions are copied in the order in which they are encountered among the rules and `#token` definitions. For example, a keyword is a special case of a variable and thus needs to occur before the variable definition.

```
#token KBegin "begin"
    .
    .
    .
#token VAR "[a-zA-Z][a-zA-Z0-9]*"
```

2.6. Quoted tokens

The regular expressions in a PCCTS grammar appear between quote marks and are literally copied into the input for DLG. Therefore, the regular expressions must use the notation accepted by DLG — a relatively standard notation using a few meta-symbols.

`"`*a*`|`*b*`"` Matches either the pattern *a* or the pattern *b*.

`"(`*a*`)"`
Matches the pattern *a*. Pattern *a* is kept as an indivisible unit.

`"{`*a*`}"`
Matches *a* or nothing, i.e., the same as `"(`*a*`|)"`.

`"[`*a*`]"`
Matches any single character in character list *a*. e.g. `"[abc]"` matches either an `a`, `b` or `c` and is equivalent to `"(a|b|c)"`.

`"[`*a*`-`*b*`]"`
Matches any of the single characters whose ASCII codes are between *a* and `b` inclusively, i.e., the same as `"(`*a*`|... |`*b*`)"`.

`"~[`*a*`]"`
Matches any single character except for those in character list *a*.

`"~[]"`
Matches any single character; literally "not nothing."

`"`*a*`*"` Matches zero or more occurrences of pattern *a*.

`"`*a*`+"` Matches one or more occurrences of pattern *a*, i.e., the same as `"`*aa*`*"`.

`"\`*a*`"` Matches the single character *a* — even if *a* by itself would have a different meaning, e.g., `"\+"` would match the `+` character. This is consistent with the use of `\` in the C language.

In addition, the symbols `%%`, `<<`, and `>>` are used to specify control for DLG; hence, these patterns must be escaped if they are to appear as literals within a regular expression. For example, `<<` would be `\<\<`. The following characters are also special.

`@` Matches end-of-file.

`\t` Tab character

`\n` Newline character

`\r` Carriage return character

`\b` Backspace character

`\0`*nnn*Matches character that has octal value *nnn*

`\0x`*nn*
Matches character that has hexadecimal value *nnn*

`\`*mnn*
Matches character with decimal value *mnn*, $1 \leq m \leq 9$

`\`*c* Character escape

White space characters (tab and space) are ignored within the quotation marks. To specify the actual space character use `"\ "`.

2.7. Interactive lexical analyzers

Normally, the scanners produced by DLG always have one character of look-ahead available. This look-ahead causes the scanner to look one character past the end of the current pattern and wait for the next character if it is not yet available.

This can cause unexpected behavior from interactive programs such as command line interpreters which must respond to input upon newline. Normally, the scanner would seek a character beyond the newline; forcing the user to type an additional character before returning the carriage return token to the parser.

The user may employ the DLG `-i` command-line option to request that unneeded look-ahead be removed whenever possible. This allows interactive parsers such as the interpreter mentioned above to get tokens without the user having to type additional characters. On some systems special flags need to be set, so that the scanner actually gets the characters when they are received rather than having the operating system buffer them. On BSD unix the `cbreak` mode needs to be used to accomplish this (e.g. `stty cbreak`).

There are situations where the character look-ahead is still required. Typically, this occurs when two patterns have the same prefix. For example, patterns α and α`b` can only be distinguished by examining the character following pattern α. A `b` character would indicate that the token associated with the second pattern should be returned. Also, any pattern that has a closure, `+` or `*`, must have look-ahead to determine the end of the pattern, since the pattern by definition can be repeated indefinitely.

2.8. DLG lexical input

The user can specify that DLG is to take its input from a stream or from a function. The function must behave like `getchar()` by returning a new character per invocation. The function must return `-1` when no more characters are available to signal "end-of-file." Presumably, the function would read characters from some buffer and hand them one by one to DLG upon request. For example, a user might define the following.

```
char text_edit_buf[BIG];

int nextchar()
{
    static char *p = &text_edit_buf[0];
    return (p < &text_edit_buf[BIG]) ? *(p++) : -1;
}
```

Then, when invoking the parser, the user would call:

```
{
    ...
    ANTLRf(start_symbol(), nextchar);
    ...
}
```

2.9. Tracking pattern position

There are several global variables available in the scanners produced by DLG to keep track of the position of a pattern within an input stream. `zzline` tracks vertical position and must be explicitly updated by the user's grammar. `zzbegcol` and `zzendcol` track horizontal position—column number within a line. They are maintained by the DLG automaton for characters that are normally one character wide. `zzendcol` needs to be corrected for cases when the character is not one character wide, e.g. tabs or carriage return.

To enable the position tracking, a `#define ZZCOL` needs to be put in an action in the lexical specification or `–DZZCOL` needs to be included in the command line when compiling the C code for the lexical analyzer.

This section described how a stream of input characters can be collected into a sequence of tokens and how one can act upon the recognition of particular tokens. Sentences in a language are composed of tokens and are described using a set of rules. The next section presents the PCCTS meta-language for describing language syntax.

3. Syntactic analysis

Once the lexical structure of a language has been specified, the way in which tokens can be combined to form sentences must be described. The set of valid sentences in a language is described by a set of rules, written in an appropriate meta-language, that impose a structure on the sequence of input tokens.

In most parser-generator tools, context-free grammars are specified in a notation which is equivalent to BNF [Nau63]. However, the notation used in regular expressions to permit repetition, alternatives, etc. can be used to extend BNF (EBNF) so that context-free grammars can be expressed more concisely. This also has the advantage of being more consistent with the regular expression notation and providing additional information which can be used to create a more efficient parser. For these reasons, PCCTS accepts an EBNF-like grammar notation.

The input for PCCTS differs from EBNF primarily in that `::=` is replaced by `:` and non-terminals are distinguished from terminals by requiring all non-terminals to begin with a lower-case letter, rather than by bracketing non-terminals with `<>`. Further, since rules can become long when actions are embedded, each rule is terminated by a `;` symbol. These modifications are not arbitrary, but mirror the differences between YACC grammars and BNF (although PCCTS provides a number of extensions beyond YACC).

As an example rule, consider:

```
block: "begin" (statement)* "end" ;
```

It indicates that rule `a` is defined to be the word `"begin"`, followed by zero or more `statement`'s, followed by the word `"end"`.

This section describes the syntax of PCCTS rules and the method by which rules communicate at run-time.

3.1. PCCTS rule definitions

A rule is formally defined, in PCCTS's own notation, as:

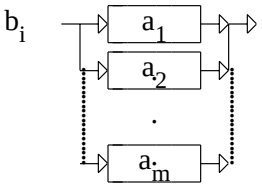
```
rule    :   NonTerminal                 /* rule name */
            { "!" }                     /* turn off automatic AST construction */
            { {"\<"} InheritAction }    /* downward-inheritance ("input") */
            { "\>" InheritAction }      /* upward-inheritance ("output") */
            { QuotedTerm }              /* name for error reporting */
            ":" block ";"               /* actual rule elements */
            { Action }                  /* fail action */
        ;

block   :   alt ("\|" alt)*
        ;

alt     :   ( element )*
        ;

element :   TokenTerm  { "^" | "!" }
        |   QuotedTerm { "^" | "!" }
        |   NonTerminal { "!" }
            { {"\<"} InheritAction } { "\>" InheritAction }
        |   Action
        |   "\(" block "\)" { "\*" | "\+" } { InheritAction }
        |   "\{" block "\}" { InheritAction }
        ;
```

where `InheritAction` is a C expression enclosed in `[ ]` used for both upward- and downward-inheritance. `QuotedTerm` is a regular expression and `TokenTerm` is a label associated with some lexeme.

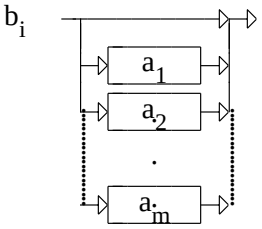
3.1.1. Subrules

Rules that employ EBNF constructs implicitly define subrules according to the following syntax:

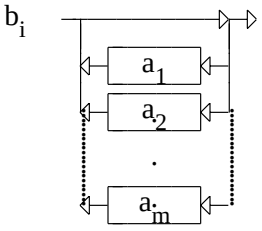
(i) $b_i = (a_1 \mid a_2 \mid ... \mid a_m)$; match 1 time



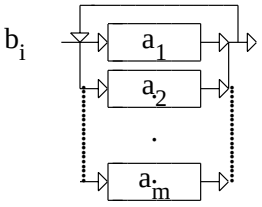
(ii) $b_i = \{a_1 \mid a_2 \mid ... \mid a_m\}$; match 0 or 1 times



(iii) $b_i = (a_1 \mid a_2 \mid ... \mid a_m)^*$; match 0 or more times



(iv) $b_i = (a_1 \mid a_2 \mid ... \mid a_m)+$; match 1 or more times



where a_i represents an alternative (list of rule elements).

With each enclosing set of meta-symbols, a new level of alternatives is permitted. Each rule or subrule thus created is called a ‘‘block.’’ In terms of the recognizer generated, blocks generated for rules and subrules are virtually indistinguishable. Subrules are like macro expansions of rules. However, there is no label associated with a subrule and thus cannot be referred to

by another rule or from elsewhere within that rule.

Consider the following rule which uses EBNF constructs:

```
e12 :   {"\+" | "\-"} VAR
    |   (FuncName | ProcName) parameterList
    |   "[0-9]+"
    ;
```

Rule `e12` has 3 alternatives. The first alternative has an optional block with 2 alternatives as its first element. The `{ }` is an optional subrule. `VAR` is a simple token. The optional block says that a `VAR` can have a plus or minus in front, but it is not necessary. The second alternative of rule `a` also begins with a subrule. The block indicates that this alternative must begin with a function or procedure name. `parameterList` is some other rule which presumably looks for a list of parameters. Alternative 3 is simply a token which recognizes an integer number.

Subrule meta-symbols alter the associativity of the `|` alternative operator and can be nested to any number of levels. `|`'s have the lowest precedence.

```
a: B | C  D ;
```

Rule `a` specifies that the parser should match either `B` or `C D`. It will not match a `D` preceded by a `B` or `C`. The following version of `a` will accomplish that:

```
a: (B | C) D;
```

Empty alternatives called epsilon transfers can also be used to imply optionality. They are useful when you want an action to be performed when nothing is matched by the other alternatives of the block.

```
a: B | C | ;
b: { B | C } ;
```

Rules `a` and `b` are identical in what they match. However, the resulting C code could be very different. Also note that with rule `a`, an action can placed in the empty alternative to be executed when a `B` or `C` is not matched. Rule `b` does not have this capability.

Subrules are allowed at most one parameter which must be of type `Attrib`. The attribute `$0` receives the inherited parameter for subrules. Subrules return values in `$0` as well (which become the appropriate `$`*i* in the enclosing subrule/scope above). `$0` is initially undefined in rules and in subrules with no explicit parameter or default `$0` initialization. See section 3.1.2.

3.1.2. Rule communication

An advantage of LL parsers over LR parsers is that inheritance (rule/subrule communication) is relatively simple and efficient. PCCTS allows each non-terminal to have an arbitrary set of downward-inherited (parameters) and upward-inherited (return value) attributes (see section

4.1.6). The specification syntax is presented in this section whereas the use of rule communication is illustrated in section 4.1.6.2:

rule[*parameters*] > [*return_type(s)*] : ξ_1 | ξ_2 | ... | ξ_m ;

which provide more powerful attribute handling than can be achieved with the ‘‘$ variables’’ found in systems like YACC (and also supported in PCCTS, primarily to ease the transition for users familiar with YACC and to communicate with the lexical analyzer).

Communicating with other rules should be as flexible as it is in a programming language. Hence, in addition to the standard attribute mechanism, PCCTS supports the familiar high-level language parameter passing scheme. The arguments to a rule must be consistent with the parameter specifications given with the definition of the rule. For example,

```
<<
#define GLOBAL 1
#define LOCAL 2
>>

prog    :    <<Sym *vars;>>
             TYPE dlist[GLOBAL] > [vars]
             <<OperateOn(vars);>>
             ...
        ;

/* Recognize a list of declarations.  The scoping level is passed
 * in and the list of symbols is returned.
 */
dlist[int level] > [Sym *list]
        :    <<$list=NULL;>>
             VAR           <<add($list, $1, $level);>>
             (    ","
                 VAR       <<add($list, $2, $level);>>
             )*
        ;
```

where `add()` is some function that adds an element to a list. Here we have used the attributes only as a means of communicating with the lexical analyzer.

In contrast to most programming languages, more than one value can be returned. Each return value is set by simply referring to it by the name given in the return value definition (prefixed with `$`) — e.g. `$list` in the `dlist` example.

Subrules may not have return types nor parameter list definitions. They may only pass a single attribute which becomes the initial value of `$0` in the subrule.[1]

To illustrate the effect of rule communications upon local variables and scoping, consider:

[1] An earlier version of PCCTS treated subrules and rules identically with respect to parameters and return values. The current distinction represents an extension to the handling of rules.

```
rule[int a, Attrib b] > [float q, int r]        /* a,b visible to rule */
   :  <<int i,j; printf("starting rule a\n");>> /* i,j visible to rule  */
      ...
      (   <<int i;>>        /* i hides rule-local i */
          ...
          (    <<int i,r;>> /* def hides outer scope i; local r can't be used */
               <<$r=5;>>    /* $r is return value not local var */
          )
      )
   ;
```

Note that return values and parameters are always given priority over local variables. While generating C code for a rule, ANTLR scans the actions and translates references to return values and parameters irrespective of scope.

3.1.3. Miscellaneous notes

The user may attach a string to a non-terminal that is passed on to the error reporting macro/function `zzsyn()`. See section 5 on error reporting for more details.

The user may elect to have PCCTS automatically construct abstract-syntax-trees (AST's). To disable automatic AST construction for a rule, employ the `!` as indicated above in the rule grammar. See section 4.2 on abstract-syntax-trees.

3.1.4. LL(k) parsing

ANTLR attempts to build parsers that use as little look-ahead as possible. Ambiguities force ANTLR to attempt resolution using more and more tokens of look-ahead. For example, ANTLR tries to resolve LL(1) ambiguities with LL(2). If the grammar construct is still ambiguous, LL(k>2) is tried until either all ambiguities are resolved or the user-defined maximum k has been reached. A subrule or rule which requires LL(k>1) does not constrain all others to LL(k>1). ANTLR builds mixed model parsers where each parsing decision is made with the smallest of amount of look-ahead possible. The `-k` command-line option is used to bound the maximum number of tokens of look-ahead to be used by ANTLR-generated parsers.

Most programming language constructs that cannot be described in LL(1) can be easily described in LL(2). Typically, LL(2) is only required in a handful of rules yielding a general LL(1) parser with a few anomalies. For example, when parsing the C programming language, a left-parenthesis could be the beginning of a type-cast or the beginning of an expression. With two tokens of look-ahead a parser could use the token following the parenthesis to determine whether an expression or type-cast followed. A more common problem lies in the definition of labels. The following grammar fragment illustrates the problem.

```
stat    :    { WORD ":" }          /* label definition */
             (
             |    "if" ...
             |    "while" ...
             .
             .
             |    WORD "\(" ...    /* forward ref to function */
             |    FUNC "\(" ...    /* ref to function already seen */
             )
        ;
```

With only one token of look-ahead, function `stat()` could not determine whether a `WORD` on the input stream was to be matched as a label or a yet to be seen function. However, if two tokens or more were available, the colon or left-parenthesis following `WORD` would unambiguously indicate which alternative to match.

3.2. Grammar ambiguities

PCCTS parsers are of the top-down, recursive descent variety which implies that all parsing decisions must be left-decidable—decidable given a finite sequence of input tokens looking from left to right across productions. Because of this limitation, there are grammars for which PCCTS cannot construct a parser. However, any language can be described by several, equivalent grammars. Ambiguities that arise in a grammar may often be circumvented by modifying that grammar while still recognizing the same language.

A grammar is considered ambiguous (in PCCTS's opinion) if one input sequence of k tokens can be matched by two different productions/alternatives within the same block. For example, if k equals one,

```
a   :   A B
    |   A C
    ;
```

is ambiguous because if all the parser "sees" is A, it will be unable to choose between alternatives one and two. ANTLR reports the following error message:

```
Antlr parser generator   Version 1.00   1989, 1990, 1991
"t.g", line 1: warning: alts 1 and 2 ambiguous upon { A }
```

If we allow two tokens of look-ahead (with `-k 2`), a parser would see either `A B` or `A C` and could uniquely determine which production to match. If subrules and/or rule references are used, the set of possible input permutations grows quickly.

```
a   :   A (B | D) E F H
    |   A b E G H
    ;

b   :   B
    ;
```

Rule `a` is ambiguous when k equals one or two since `A B` can start either production. If we increase k to three, rule `a` is still ambiguous because `A B E` begins both productions. Only when k is set to four is `a` unambiguous. ANTLR applies a similar logic when constructing parsers. When an ambiguity occurs at k==*i*, k=*i*+1 is tried until either the ambiguity is resolved or the maximum look-ahead defined by the user is reached (in which case the ambiguity is reported to the user). ANTLR reports the following for k==2,

```
Antlr parser generator   Version 1.00   1989, 1990, 1991
"t.g", line 1: warning: alts 1 and 2 ambiguous upon { A }, { B }
```

Note the sequence `A B` is denoted by the singleton set `{ A }` followed by the singleton set `{ B }`. In general, ambiguous input sequences of k tokens are represented by a sequence of k sets.

The `-pa` command-line option can be used to print out the user's grammar (minus actions) annotated with the input token permutations. The set sequence `{ A B }, { C D }` means that the following four input token permutations are possible:

```
A C
A D
B C
B D
```

3.3. Look-ahead size

The user's grammar can be analyzed using any integer k value (limited by machine size and how long the user wants to wait). However, parsers constructed by ANTLR always use k values equal to the nearest power of two greater than or equal to the k specified by the user. ANTLR parsers treat the look-ahead as a circular buffer in which a "modulo k" operation is performed repeatedly. If k is a power of two, this operation reduces to a logical-and operation whereas other values generally require a divide — i.e. a time consuming operation. Since the modulo operation is performed at least once for each input token, a divide operation would seriously degrade parser performance. The `#define` constant `LL_K` is set by ANTLR to the k associated with parser execution. Note that files not constructed by ANTLR will need to define `LL_K` if they include `antlr.h`. This can be handled with a "`-DLL_K=k`" command-line option on most C compilers.

3.4. ANTLR parser construction

ANTLR generates parsers in pure C code--i.e. non-interpretive, mutually-recursive sets of functions. There is exactly one C function for every rule present in an ANTLR grammar description. Transformations exist to convert grammar constructs to C language constructs. To perform lexical analysis, ANTLR prepares a DLG input file that will be converted to a C scanner by DLG and then linked into the parser. Because of ANTLR's flexibility, C code generation templates vary depending on command-line options and `#define`'s [Par90]. ANTLR parsers are fast primarily because they never have to back up and map EBNF grammar constructs to efficient target language constructs. The current k tokens of look-ahead always tells the parser which alternative to choose. Also, C's efficient function invocation mechanism allows references to other rules to be handled quickly.

3.4.1. Efficiency

The way grammars are described has a big impact on phrase recognition speed. Rules invoking other rules with tail-recursion (right-recursion) like

```
/* S l o w  A n d  U g l y */
a: B c ;
c: a | ;
```

can be replaced with

```
/* F a s t e r  B u t  S t i l l  U g l y */
a: B (a | ) ;
```

or, better yet,

```
/* F a s t e s t  A n d  R e a d a b l e */
a: B (B)* ;
```

which is more concise and infinitely more readable.

This example leads us to two important generalizations:

- Subrules are much faster than invoking another rule to match the same subgrammar.
- Use the `( )*` and `( )+` closure operators instead of tail recursion. This generates a `while` loop instead of recursive function calls.

Rules or subrules with many alternatives tend to generate bigger and slower programs. This arises from the fact that a linear search implemented by a sequence of `if` statements is used to determine which alternative to choose. Although general LL(k) parsing makes it difficult to optimize this search for the correct alternative, future versions of PCCTS will probably make a number of special-case improvements, such as generating a `switch` statement for each alternative selection problem which is LL(1).

In many parser-generators, such as YACC, adding actions to a grammar can cause rules to be split, introducing ambiguities and/or extra states in the generated parser. In contrast, actions have no affect on the structure of the recognizers constructed by PCCTS. PCCTS simply generates C code templates that implement the recognizer and embeds C code for the user-supplied actions at the proper points. User actions have no significant affect on parsing speed.

3.4.2. Debugging parsers

PCCTS parsers have one function for each rule. The function name will coincide with the rule it was constructed from and therefore can be referenced from within a source-level debugger. Breakpoints can be set so that parser execution stops before or after certain grammar fragments of interest have been recognized. PCCTS parsers are easy to follow because parsing decisions are made using simple `if` and `while` statements.

Often it is convenient to track the sequence of rule invocations made in an PCCTS parser. The `-gd` command-line option forces PCCTS to generate code that calls `zzTRACE()` with the rule name in which it appears as an argument. `zzTRACE()` can be a macro or a function.

3.5. PCCTS parser template

The following code template can be used to begin parser development.

```
/*
 * A N T L R  P a r s e r  T e m p l a t e
 *
 * Terence Parr, Hank Dietz and Will Cohen
 * CARP Research Group
 * Purdue University Electrical Engineering
 * PCCTS Version 1.00
 */
#header << /* Define Attrib/AST or include standard package */ >>

/* O p t i o n a l */
#token "[\t\ ]+"    << zzskip(); >>               /* Ignore White */
#token "\n"         << zzline++; zzskip(); >>     /* Track Line # */
```

```
<<
main() { ANTLR(startSymbol(), stdin); }

zzcr_attr(attr, token, text)
Attrib *attr;
int token;
char *text;
{
    /* *attr = Function(text, token) */
}

zzd_attr(attr) Attrib *attr; { /* destroy attribute */ }

zzcr_ast(ast, attr, tok, text)
AST *ast;
Attrib *attr;
int tok;
char *text;
{
    /* *ast = Function(attr) */
}


/* node constructor for use with #[] */
AST zzmk_ast(node, user_args)
AST *node;               /* newly created node */
{
    /* fill in node->fields */
    return node;
}

zzd_ast(ast) AST *ast; { /* destroy AST node pointed to by 'ast' */ }

#define zzdef0(zero) /* *(zero) = initial $0 attrib */
>>

/* P u t  R u l e s  H e r e */
```

3.6. Grammatical actions

Actions are blocks of C code enclosed in `<<` and `>>`. They are placed in the ANTLR-generated C files according to position in the corresponding grammar file. In general, an action following a token or non-terminal reference is executed after that grammar element has been recognized. There is no restriction concerning their placement within rules. However, actions cannot be specified between rules. They may only be used before and after the set of rules.

3.6.1. Init Actions

If an action is specified before any grammar element is found after the `:` in a rule definition or after the start of a subrule, it is placed before any ANTLR generated C code for that rule or subrule. This allows the user to define variables and execute C code in a scope local to that rule or subrule; any C code given is executed just before the rule or subrule is applied. For example,

consider the definition of a rule `a`:

```
a   :   <<List *p=NULL;>>   /* Get a type followed by a list of vars */
        Type
        (   <<int i=0;>>
            Var <<append(p, i++, $1);>>
        )*
        <<OperateOn(p);>>
    ;
```

The init action `<<List *p=NULL;>>` defines a local `auto` variable called `p` and initializes it to `NULL`. The subrule also has an init action, `<<int i=0;>>`, which is executed every time the subrule is entered.

The init actions of optional blocks are executed whether or not the elements in the block are recognized. This is due to the fact that subrules (C code block) must be entered in order to determine whether or not that grammar construct can be matched on the input stream. This can be confusing, but on the other hand, we can force a variable to have a value even if nothing is matched in an optional block. For example,

```
a   :   <<Sym *p;>>
        ...
        { <<p=NULL;>> ":" Type <<p=$2;>> }
        <<OperateOn(p);>>
        ...
    ;
```

The init-actions of closure blocks (`( )*` subrules) are executed only once — upon entry to the block. This allows local variable initialization within closure blocks.

Only `$0` can be referenced in an init-action since nothing will have been matched when the action executes. Default `$0` initialization occurs before init-actions are executed.

3.6.2. Fail actions

Fail actions are actions that are placed immediately following the `;` rule terminator. They are executed after a syntax error has been detected but before a message is printed and the attributes have been destroyed (optionally with `zzd_attr()`; see the section on attributes). However, attributes are not valid here because one does not know at what point the error occurred and which attributes even exist. Fail actions are often useful for cleaning up data structures or freeing memory. For example,

```
a   :   <<List *p=NULL;>>
        ( Var <<append(p, $1);>> )+
        <<OperateOn(p); rmlist(p);>>
    ;
    <<rmlist(p);>>
```

The `( )+` loop matches a lists of variables (`Var`'s) and collections them in a list. The fail-action `<<rmlist(p);>>` specifies that if and when a syntax error occurs, the elements are to be `free()`'ed.

3.6.3. Actions appearing outside of rules

When PCCTS is used to construct a complete compiler, it is useful to be able to incorporate support code written in C in the same file. For example, there must always be a `main` specified and the ANTLR-generated parser must be invoked using the `ANTLR` macro. Hence, PCCTS allows arbitrary C code to appear as ‘‘actions’’ outside of the grammar rules. The PCCTS description

```
<<
#include "junk"
static int local_to_the_parser;
main() {ANTLR(a(), stdin);}
>>

a: a_stuff ;
b: b_stuff ;

<< hello() { printf("Hello?"\n"); } >>
```

will result in the following output C code:

```
#include "junk"
static int local_to_the_parser;
main() {ANTLR(a(), stdin);}

a()
{
    ... code to recognize a_stuff ...
}

b()
{
    ... code to recognize b_stuff ...
}

hello() { printf("Hello?"\n"); }
```

Note that although these segments of C code look like ‘‘actions,’’ and the same rules apply concerning the use of escape characters, they are not actions per se. Hence, no attribute values are accessible.

4. Attribute handling

Given a grammar, PCCTS constructs a C program that recognizes phrases in the language described by that grammar. No translation from source language to output language is performed; the resulting parser merely complains if a syntax error is detected. To effect a transformation, actions must be embedded within the grammar that either perform an immediate

translation or create an intermediate-form which is translated at a later time. Grammar actions have access to lexical information via run-time objects called attributes which are a function of the token number and the text of the lexeme found on the input stream. PCCTS also supports the creation and manipulation of abstract-syntax-trees (AST's) to handle more complicated transformations. This section describes the use of attributes, attribute inheritance as well as other forms of rule communication, and the handling of AST's.

4.1. General attribute handling

Attributes are objects that are associated with all rules and rule elements including block elements like `{ }`. Attributes are generally identified by `$`*i* (where *i* is a positive integer) and are used in actions to identify the values of rule elements. Attributes behave like variables. They are mentioned and labeled at analysis-time, but have values only at run-time. The positive integers uniquely identify an element within the currently active block and within the current alternative of that block. With the invocation of each new block, a new set of attributes becomes active and the previously active set is temporarily inactive (unless `$`*i*`.`*j*, `$$` or $*inheritance_attr* attributes are used; see sections 4.1.4). Attributes are scoped exactly like local stack-based variables in C.

`$`-variables refer to user-defined objects associated with each rule element. The `zzcr_attr()` function transforms lexical objects into grammar attributes. Attributes associated with terminals are implicitly created and are a function of lexemes. However, attributes of non-terminals and subrules are explicitly manipulated by actions provided by the user. Terminals, which are lexical tokens, simply have a value associated with them and are generally referenced not reset by actions. Attributes that are returned from or passed to blocks are considered synthetic attributes. Rules and subrules may communicate via attributes or with familiar high-level language parameter passing techniques (see sections 3.1.2 and 4.1.6).

PCCTS requires that the user define the data type or structure of the attributes as well as specify how to convert from lexemes to attributes. This provides significant control over what information attributes carry and how the lexical analyzer should react to lexical objects found on the input stream. The type of the attributes is always defined by a C `typedef` named `Attrib` and must be defined before `antlr.h` is included in your C program. Using the PCCTS `#header` instruction, the user may specify the C definition or `#include` the file needed to define `Attrib`. The action associated with `#header` is placed in every C file generated from the input grammar files. Any C file created by the user that includes `antlr.h` must once again define `Attrib` before `#include`'ing `antlr.h`.

Attributes are stored and accessed in stack fashion. With the recognition of each element in a rule, a new attribute is pushed on the stack—i.e. becomes available. In other words, the ith attribute is not available until the ith rule element has been matched. If a rule element is something other than a simple token, many attributes may be pushed on the attribute stack until they reduce to a single attribute for that rule reference or subrule.

For example, if attributes are to be simple integers, the following C definition is sufficient and necessary

```
typedef int Attrib;.
```

However, this does not specify how those integer attributes are converted from character strings in the input stream.

4.1.1. Attribute creation

The attributes associated with terminals must be a function of the text and the token number associated with that lexeme. These values are passed to `zzcr_attr()` which computes the attribute to be associated with that lexeme. The user must define a function or macro that has the following form:

```
zzcr_attr(attr, token, text)
Attrib *attr;   /* Pointer to attribute associated with this lexeme */
int token;
char *text;     /* text associated with lexeme */
{
    /* *attr = F(text,token); */
}
```

Consider the following `Attrib` and `zzcr_attr()` definition.

```
typedef union {int ival; float fval;} Attrib;

zzcr_attr(attr, token, text)
Attrib *attr;
int token;
char *text;
{
    switch ( token )
    {
        case INT   : attr->ival = atoi(text); break;
        case FLOAT : attr->fval = atof(text); break;
    }
}
```

The `typedef` specifies that attributes are integer or floating point values. When the regular expression for a floating point number (which has been labeled `FLOAT`) is matched on the input stream, `zzcr_attr()` converts the string of characters representing that number to a C `float`.

The next attribute example is more involved. Attributes are either symbol table entries or integer values. Variables (`VAR`), intrinsic types (`BuiltInType`) and parameters (`PARAMETER`) found on the input stream are converted to symbol table pointers. Integers are converted to C `int`'s.

```
/* Assume Sym, zzsymget() are defined elsewhere */
#header <<typedef union { Sym *var; int val; } Attrib;>>

<<
Sym *cursym;

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


zzcr_attr(attr, token, text)
Attrib *attr;
int token;
char *text;
{
    switch ( token )
    {
        case WORD :
        case BuiltInType :
        case PARAMETER :
        case VAR : attr->var = cursym;
                   break;
        case INT : attr->val = atoi(text);
                   break;
    }
}
>>


#token WORD "[a-zA-Z_][a-zA-Z_]*"
                <<{ extern Sym *cursym;

                    cursym = zzsymget(zzlextext);
                    if ( cursym == NULL ) cursym = zzsymnew(zzlextext);
                    else LA(1) = cursym->token;
                 }>>

prog :   ... VAR ... ;
```

When a `WORD` is matched on the input stream, the character string is looked up in the symbol table. If it exists, a pointer to it is placed in `cursym` and the current look-ahead token (`LA(1)`) is modified accordingly. Otherwise, a symbol table record is created leaving `cursym` pointing to it. Also of note, we cannot place the `zzsymget()` call into `zzcr_attr()` because of the look-ahead. `zzcr_attr()` is called after the token has been recognized; modifying `LA(1)` in `zzcr_attr()` would have no effect. Parsing decisions are made based upon current look-ahead and, therefore, the lexical action must "compute" the token value before the parser receives it.

4.1.2. Attribute destruction

The user may elect to "destroy" all attributes created with `zzcr_attr()`. A macro or function called `zzd_attr()`, is executed once for every attribute. This occurs collectively at the end of every block. `zzd_attr()` is passed the address of the attribute to destroy. This can

be useful when memory is allocated with `zzcr_attr()` and needs to be `free()`'ed. For example, sometimes `zzcr_attr()` needs to make copies of some lexical objects temporarily. Rather than explicitly inserting code into the grammar to free these copies, `zzd_attr()` can be used to do it implicitly. Notice that this concept is similar to the constructors and destructors of C++ [Str87].

Attributes are often character strings. Copies of the lexical text buffer (managed by DLG) are made which later need to be deallocated. This can be accomplished with code similar to the following.

```
#header <<#define zzd_attr(attr) {free(*(attr));}
          typedef char *Attrib;>>

<<
zzcr_attr(attr, token, text)
Attrib *attr;
int token;
char *text;
{
    extern char *malloc();

    if ( token == StringLiteral )
    {
        *attr = malloc( strlen(text)+1 );
        strcpy(*attr, text);
    }
}
>>
```

4.1.3. Standard attribute definitions

Some typical attribute types are defined in the PCCTS include directory. These standard attribute types are contained in the following include files:

`charbuf.h`
: Attributes are fixed-size buffers, each of 32 characters in length. If a string longer than 31 characters (31 + 1 `'\0'` terminator) is matched for a token, it is truncated to 31 characters. The user can change this buffer length from the default 32 by redefining `ZZTEXTSIZE` before the point where `charbuf.h` is included. The text for an attribute must be referenced as `$`*i*`.text`.

`int.h`
: Attributes are `int` values derived from tokens using the `atoi()` function.

`charptr.h, charptr.c`
: Attributes are pointers to dynamically-allocated variable-length strings. Although generally both more efficient and more flexible than `charbuf.h`, these attribute handlers use `malloc()` and `free()`, hence, they can cause memory fragmentation. The file `charptr.c` must be `#include`'d, or linked with, the C code PCCTS generates for any grammar using `charptr.h`.

In addition to the standard library of attribute handlers, the user may define new attribute types and handlers — which should also be placed in the PCCTS include directory. The makefile

template given in section 1.4 knows where the include files live and will inform the C compiler.

The following example uses the `charptr` package:

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

<<
#include "charptr.c"
main() { ANTLR(a(), stdin); }
>>

#token "\n"      <<zzskip();>>

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

4.1.4. Attribute references

PCCTS accepts five basic attribute referencing notations:

$*i* This format refers to the i^{th} rule element within the current alternative which can be a rule reference, token or subrule.

$*i.j* This format extends the $*i* notation to allow the user to refer to attributes at scopes above the current scope. *i* is the scope and *j* is the j^{th} rule element within the alternative that encloses the current scope.

$$ This is a shorthand for the `$0` of the enclosing rule and can be referred to from an action anywhere within the rule.

$*label*

$*rule* is identical to `$$` where *rule* is the name of the enclosing rule.

$*parameter* refers to one of the user-defined parameters given in the enclosing rule definition.

$*return_value* refers to one of the user-defined return values given in the enclosing rule definition.

`$[`*token, text*`]`

This represents an attribute constructor. Its value is computed using `zzcr_attr` just like any other attribute with *token* and *text* as parameters. This is an explicit method of attribute creation whereas normally attributes are created implicitly — one for each input terminal. This mechanism can be useful when creating AST nodes which are functions of attributes or for passing values to subrules. For example, if attributes were integers and integer tokens were labeled `NUM`, `$[NUM,"34"]` would create an attribute whose value was 34. `$[]` returns an empty attribute node.

4.1.4.1. $*i* references

As a simple example of `$i` numbering in rules with alternatives, consider the following.

```
a: B | C ;
```

Rule `a` has 2 alternatives. `$i` refers to the *i*th rule element in the current block and within the same alternative. So, in rule `a`, both `B` and `C` are `$1`.

Below, rule `a` has a subrule as the second rule element and is referenced as `$2` after the entire subrule has been matched. Assume that `charptr.h` attributes are in effect.

```
a : "ick" ("A" <<$0 = $1;>> | "B" <<$0 = $1;>>) "ugh"
    <<printf("$1,$2,$3 are %s,%s,%s\n", $1,$2,$3);>>
  ;
```

In this case, one of the following will be printed depending on whether `"A"` or `"B"` was found

```
$1,$2,$3 are ick,A,ugh
$1,$2,$3 are ick,B,ugh
```

`$2` is the attribute of the subrule `("A"|"B")` and is the second rule element of rule `a` (outermost level). However, within the subrule, `"A"` and `"B"` are both `$1`. The initial value of `$0` will be undefined unless it either inherits a value or is assigned a value by defining the macro `zzdef0`. This value can, of course, be altered within the subrule by using `$0` on the left side of an assignment.

`$$` always refers to the `$0` of the outermost scope — the rule scope (i.e `$1.0`). `$$` can be referenced from any user action within any subrule.

4.1.4.2. $*i.j* references

The EBNF subrules of PCCTS introduce scoping as in programming languages. To access attributes at scopes above (in an enclosing scope) from within a user action, `$`*i.j* variables are used where *i* is the scope and *j* follows the normal attribute numbering. Levels are numbered from 1 starting at the rule level. Each subrule increments the level. Actions at level *i* cannot access attributes at level *i*+1 or below just like in a programming language. Attributes of the form `$`*i* are accepted as a shorthand. When a level is not indicated, the current scope will be assumed. For example,

```
a   :    A B (C (D E) F) G ;
```

The following table summarizes levels and attribute numbers:

Token	$level.attr#	$level	Availability
A	$1.1	$1	rule a after A
B	$1.2	$2	rule a after B
C	$2.1	$1	(C (D E) F)
D	$3.1	$1	(D E)
E	$3.2	$2	(D E)
F	$2.3	$3	in subrule after F
G	$1.4	$4	rule a after G

In this example, actions at level 1 (rule level) cannot access attributes within any of the subrules. Actions before the subrules cannot access these lower attributes because they have not been created yet (the terminals have not been recognized). Actions following the subrules cannot access the lower attributes because they have all been collapsed into one attribute--namely, `$3`.

4.1.4.3. $*label* references

Inheritance (parameters and return values) and rule-level `$0` attributes are referred to by name. $*rule* is identical to `$$`, but is more informative. The following two examples perform the same function, but the first uses upward-inheritance and and the second uses $*rule* notation.

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

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

start :     <<int r;>>
            expr > [r] "\n" <<printf("result %d\n", r);>>
      ;

expr > [int result]
      :    "[0-9]+"        <<$result = $1;>>
           ( "\+" "[0-9]+" <<$result += $2;>> )*
      ;
```

Note that the next version does not require a parameter definition since all rules implicitly return an attribute ($$, $*rule*).

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

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

start :     expr "\n" <<printf("result %d\n", $1);>>
      ;

expr  :    "[0-9]+"        <<$expr = $1;>>
           ( "\+" "[0-9]+" <<$expr += $2;>> )*
      ;
```

Similarly, $*parameter* refers to one of the user-defined parameters.

4.1.5. $[*token, text*] attribute constructor

This attribute is not automatically destroyed via `zzd_attr` if `zzd_attr` is defined. If necessary, the user must explicitly destroy these attributes. For example,

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

<<
#include "charptr.c"
main() {ANTLR(a(), stdin);}
>>

a   :    <<Attrib b = $[0, "a string"];>>
         <<printf("b is %s\n", b);
           zzd_attr(&b); >>
    ;
```

which would yield

```
b is a string
```

Note that the token number is irrelevant for `charptr` type attributes since its `zzcr_attr` is a function of the text only.

4.1.6. Inheritance

Inheritance refers to the ability of a rule to obtain information about the context in which it was invoked. Because ANTLR generates top-down parsers, information can be carried down through each rule invocation as well as returned. This ability, dubbed downward-inheritance, provides significant power because a rule can decide which rule invoked it or gain some other context altering information. LR parser-generators create parsers of the bottom-up variety and will never be able to pass information to a subrule because recognition begins at the leaves of the parse-tree and works it way upwards to the root (start symbol).

4.1.6.1. `$0` inheritance

`$0` is one mechanism by which attributes can be inherited from and returned to an invoking block. The inheritance mechanism uses the attribute stack to initialize the `$0` value seen by a subrule. Subrules can only pass one parameter which is implicitly typed as an attribute (`$0`). The `$0` of a subrule is undefined unless set by a user action or by passing an attribute to that subrule. For example,

```
decl :    TypeID
          Var       <<$2.type = $1;>>
          ( "," Var <<$2.type = $0;>> )*[$1]
     ;
```

where the `[ ]` is an `InheritAction` mentioned above in the grammar for a PCCTS rule. The value inside is passed to the `$0` of the subrule. This syntax is also used to pass parameters to rules. See below for more details. In this case, we are passing the type of a declaration (represented by the attribute associated with `TypeID` — `$1` in the outermost scope) to a subrule which accepts a list of variables separated by commas. This could alternatively be accomplished via

```
decl :    TypeID
          Var       <<$2.type = $1;>>
          ( "," Var <<$2.type = $1.1;>> )*
     ;
```

or with a local variable.

```
decl :    <<Attrib type;>>
          TypeID    <<type = $1;>>
          Var       <<$2.type = type;>>
          ( "," Var <<$2.type = type;>> )*
     ;
```

Unlike subrules, rules employ the C hardware stack and `$0` is *never* automatically set by inheritance. Despite this, the inheritance of a value for a rule's `$0` can be accomplished by:

`rule[Attrib dummy]: <<$0=$dummy;>>` *remainder_of_rule* `;`

This uses the more powerful rule communication mechanism and an assignment to `$0` to simulate the inheritance mechanism used for subrules. Rules have a much more flexible inheritance scheme which, combined with rule-local variables for subrule inheritance, make the YACC-like attributes (`Attrib`'s) useful primarily for communicating with the lexical analyzer.

If a macro called `zzdef0` is `#define`'d, it is executed upon entry to any block (rule or subrule) via `zzdef0(&($0))`. This macro can be used to ensure that all attributes have a certain property. For example, if `$0` is not set in a rule, then the `$`*i* in the invoking rule will be undefined (some previous attribute typically). Setting the `$0` initially to `NULL` or some other meaningful value can be convenient. If attributes are string pointers, not assigning `$0` to `NULL`

before rule completion could cause an invoking rule to assume a valid pointer had been returned (as `$`*i*).

Note that use of `zzdef0` is incompatible with the use of `$0` in subrules to inherit values, i.e., `zzdef0` will destroy the inherited value.

4.1.6.2. Generalized rule communication

PCCTS rules may pass parameters and return values just as C functions do (though, PCCTS rules can return multiple values). This mechanism can be viewed as generalized attribute handling where each rule defines its downward- and upward-inheritance attribute types.

4.1.6.2.1. Downward-inheritance

The syntax for PCCTS parameter passing (downward-inheritance) is identical to that of the ANSI C Standard [ANS90] except that the `( )` is replaced by `[ ]`. Specifically, parameter declarations are comma-separated lists of single declarations:

`rule[`$type_1\ var_1, ..., type_n\ var_n$`] : ... ;`

The parameter declarations are broken down into old style C declarations when parsers are generated:

```
rule(var1, ..., varn)
type1 var1;
     ...;
typen varn;
{
    /* Stuff to recognize rule */
}
```

which is recognized by both ANSI compliant and old-style C compilers.

The parameters mentioned in the parameter declaration list may be used in user actions like any other variable just as in C except that they must be preceded by a `$`. The `$` is consistent with references to the attributes associated with terminals and rule references. For example,

```
a[int i, int j] : <<printf("%d\n", $i+$j);>> ;
```

adds the two inherited attributes `i` and `j` and prints the result to `stdout`. It can be invoked in the following manner:

```
rule : a[3,4]
     ;
```

4.1.6.2.2. Upward-inheritance

Return values are similarly declared via a comma-separated list of types and names enclosed in `[ ]` (following the parameter definition if any):

`rule[`*parameters*`] > [`*type*$_1$ *var*$_1$, ..., *type*$_n$ *var*$_n$`] : ... ;`

The names given in the return value definition may be used to explicitly set the return values. These may be treated like `$`-prefixed variables and can referenced as well as set. The names in the return value definition may not conflict with any parameter or local variable defined elsewhere in that rule.

```
a[int i, int j] > [int x, int y] : <<$x = $i+$j; $y = 0;>> ;
```

Rule `a` can be invoked in the following manner:

```
rule :  <<int i,j;>>
        a[3,4] > [i,j]
     ;
```

which passes 3 and 4 to rule `a` which returns 7 and 0 placing them into `i` and `j` respectively.

4.2. Abstract-syntax-tree construction and manipulation

Many translation problems can be accomplished with actions embedded in a grammar that simply compute an output phrase and send it to the output stream. Unfortunately, most translation problems cannot be implemented in such a straightforward manner. For this reason, PCCTS supports an abstract-syntax-tree (AST) mechanism that provides a framework for constructing and manipulating intermediate representations of the input stream.

Without touching the grammar description, the user may direct PCCTS (by using the `-gt` command-line option) to automatically generate an AST of the input stream. By default, this AST is a flat tree (a list) with one node for each input token. Generally, the organization of an AST reflects the language structure used to recognize an input phrase. A flat AST does not contain information about the language structure and, in effect, would only be a copy of the input. The default automatic tree construction can be modified by annotating the grammar with a few appropriately placed characters and is sufficient to handle many situations (such as arithmetic expressions) well.

Some intermediate-forms cannot be adequately described by simple grammar annotations (e.g. type-trees for the C programming language). A more general tree construction mechanism is provided by PCCTS to allow explicit intermediate-form tree description and is described in section 4.2.4.

AST nodes and subtrees, both automatically and explicitly constructed, are referenced via #-variables. The notation and behavior of these variables is similar to that of the attribute `$`-variables. They are always implicitly typed as pointers to AST nodes and are generally

associated with rule and token references; rules yield subtrees and tokens yield nodes.

PCCTS parsers manage the actual creation of nodes and the construction of the trees, but the user must describe what information an AST node is to carry and how to fill in that information.

The command-line option `-gt` must be used to access any of the features is this section. Also note that the PCCTS automatically defines `#define GENAST` so that the user may write code that depends on whether or not trees are being constructed.

4.2.1. AST node creation

PCCTS provides a default and an explicit node constructor. The default constructor is applied for each token in a rule defined without the `!` modifier (which indicates that the rule should disable automatic tree construction) and is a function of the associated attribute, token and lexical text. The explicit constructor is user defined and can take a variable number of arguments.

A macro called `zzcr_ast()` can be defined to give a default node definition or transformation. Also, user actions embedded in the grammar can modify AST fields. AST nodes have the structure

```
struct _ast {
    struct _ast *right, *down;
    user_defined_fields
};
```

where *user_defined_fields* is filled in via the `#define` constant: `AST_FIELDS`. Only the user-defined fields should be modified by the user as `right` and `down` are handled by the code inserted by PCCTS and could be subject to change. `zzcr_ast()` is of the form

```
#define zzcr_ast(ast,attr,tok,text) *ast = F(attr, tok, text);
```

with

```
AST *ast;
Attrib *attr;
int tok;
char *text;
```

For example, a simple, but useful attribute and AST definition is

```
#header <<
typedef int Attrib;
#define AST_FIELDS int i;
#define zzcr_attr(attr, token, text) *(attr)=atoi(text)
#define zzcr_ast(ast, attr, tok, text) (ast)->i = *(attr)
>>
```

which defines both attributes and AST node data to be integers. This default AST node construction merely copies the integer found on the input stream into the data field `i` of the new AST node.

AST nodes may be explicitly constructed via a user-defined function called `zzmk_ast()` which is defined as follows:

```
#include <varargs.h>

/* fill in an AST node (which is passed in) and return a ptr to it */
AST *zzmk_ast(va_alist)
va_dcl
{
    va_list ap;
    AST *node;

    va_start(ap);
    node = va_arg(ap, AST *);  /* 1st arg is always a new AST node */
    ...
    /* fill in node */

    va_end(ap);
    return node;
}
```

Note that the K&R-style variable argument method is not required if you are not interested in portability or a constant number of parameters is passed (if only one node type is used, for example). A portable ANSI C version would look like the following:

```
#include <stdarg.h>

/* fill in an AST node (which is passed in) and return a ptr to it */
AST *zzmk_ast(AST *node, ...)
{
    va_list ap;

    va_start(ap, node);
    ...
    /* fill in node */

    va_end(ap);
    return node;
}
```

To make an explicit node constructor that duplicates the simple integer AST node definition given above, the following would suffice (ANSI C version):

```
#header <<
typedef int Attrib;
#define AST_FIELDS int i;
#define zzcr_attr(attr, token, text) *(attr)=atoi(text)
#define zzcr_ast(ast, attr, tok, text) (ast)->i = *(attr)
#include <stdarg.h>
>>

AST *zzmk_ast(AST *node, ...)
{
    va_list ap;

    va_start(ap, node);
    node->i = va_arg(ap, int);
    va_end(ap);
    return node;
}
```

Now, an action may construct tree nodes via `#[`*integer*`]`. For example,

```
a > [AST *node]
    :    <<$node = #[34];>>  /* return a node with 34 in it */
    ;
```

The `#[ ]` node constructor is detailed in section 4.2.6.

4.2.2. AST node destruction

A routine called `zzfree_ast(AST *t)` is used to free any abstract-syntax-tree created by an PCCTS parser. If defined, a macro called `zzd_ast` is applied to each AST node before its memory is `free()`'d. This can be used to `free` any additional memory that may have been allocated by `zzcr_ast` and attached to the AST nodes. The `zzd_ast` macro is passed the address of the node it is to destroy and is of the form

```
#define zzd_ast( treeNodePtr )  /* perform operation on tree node */
```

`zzd_ast` must be defined in or included into the `#header` action. `zzfree_ast()` must be called explicitly by the user.

4.2.3. Automatic AST construction

For small or uniform grammars (like C language expressions), PCCTS provides a concise tree construction mechanism. Token references are annotated to indicate

- which tokens are to be subtree roots in the AST
- which tokens are to be excluded from the AST

Any non-modified (non-root and non-excluded) token is considered a leaf node. Rule names are never included in the automatically generated AST's. The resulting tree is composed entirely of

nodes derived from grammar terminals. Each rule yields one subtree. Subrules do not create subrules; elements enclosed within subrules modify the subtree for the current rule. Subtrees and nodes created from terminal references are accessed from user actions via #-variables which are discussed in section 4.2.6.

4.2.3.1. Grammar annotation

PCCTS generates an AST by placing C code into the parser that, at parser run-time, builds a tree and makes it available to the user's program. AST's are stored in a child-sibling list form similar to that used by LISP. This allows each level in the AST to have one root and arbitrarily many children:

(root $child_1$ $child_2$ ... $child_n$)

Suffixing a token with ^ identifies it as a subtree root. Each root-token encountered increases the depth of the current subtree by 1. All further leaf tokens are considered siblings to the previous root node. A ! suffix on a terminal reference excludes the terminal from the AST (no AST node is created or linked in). Suffixing a rule reference with a ! indicates that the subtree created by that rule reference is not to be linked into the current subtree, but is still to be made available via *#i* variables. Rule references yield a pointer to the root of the subtree created by that rule invocation. Therefore, the subtree roots returned by rules become siblings in the current rule. Rules defined with a ! immediately following the rule name (and before the :) do not automatically construct trees. A ! in the rule definition is like placing a ! after each grammar element within that rule. #-variables do not exist for tokens suffixed with !. Hence, tree nodes are not automatically constructed within these rules although references to other rules can still create subtrees that can be accessed via #-variables.

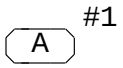
To demonstrate the actual construction of AST's, consider the following grammar fragment.

```
x   :   A B^ C! D y
    ;

y   :   E^ F
    ;
```

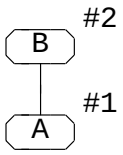
The grammar matches

```
A B C D E F
```

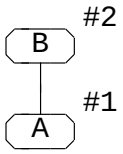
and considers B to be a subtree root. Token C is to be ignored. Rule y matches E F and returns a subtree of depth 2 to rule x where the root of rule y's subtree becomes available as #5. The following figure sequence shows the state of the AST after each token has been recognized. The various AST nodes are labeled with the #-variable that could be used to access that node. The LISP-like child-sibling notation is given after the input state.



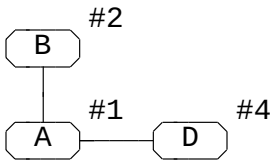
A • B C D E F, (A)



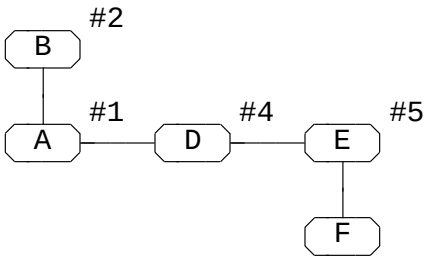
A B • C D E F, (B A)



A B C • D E F, (B A)



A B C D • E F, (B A D)



A B C D E F •, (B A D (E F))

Because of the ! suffix, terminal C does not appear in the AST. Terminal B is a root-token because of the ^ modifier and is therefore the root of the subtree created for rule x. The root of the subtree created by rule y is a sibling of the leaf nodes of rule x. The way in which rules describe a grammar effects AST construction. e.g. rules x and y could be rewritten to recognize the same input, but to construct a different tree.

```
x   :   A Bˆ C! D Eˆ y
    ;

y   :   F
    ;
```

In this case the root-token E will become the root of the tree matched up to that point (i.e. `( B A D )`) yielding

```
( E ( B A D ) )
```

The final AST would be

```
( E ( B A D ) F )
```

4.2.3.2. Operator precedence

Operator associativity in the AST's is directly related to the precedence implicitly defined among the rules. For example, to specify that the exponentiation operator, `**`, groups right-to-left, a grammar similar to the following would be required.

```
expr :   exp0 { "\*\*"ˆ expr }
     ;
exp0 :   "[0-9]+"
     ;
```

The input

```
3**4**5
```

would yield

```
( \*\* 3 ( \*\* 4 5 ) )
```

indicating that the `4**5` would be performed first.

For a full explanation of arbitrary tree construction see the section on explicit AST construction. However, the following example is beneficial in this section. Give a grammar fragment that recognizes a simple assignment, a tree of the form: `( := expr lvalue )` can be constructed via:

```
assign! : lvalue ":=" expr ";" <<#0 = #(#[$2], #3, #1);>>
        ;
```

The `!` after the rule definition tells PCCTS to turn off automatic tree construction for this rule. `#[$2]` is an AST node constructed from an attribute.

4.2.3.3. Example

The following simple grammar illustrates the automatic construction of AST's, the definition/creation of AST nodes, and the use of *#i* variables

```
#header <<
    typedef int Attrib;
    #define AST_FIELDS int i, token;
    #define zzcr_attr(attr, token, text) *(attr)=atoi(text)
    #define zzcr_ast(ast, attr, tok, text) {(ast)->i = *(attr); (ast)->token=0;}
>>



<<
AST *root = NULL;   /* define a root ptr for AST; must be NULL */

void show(tree)     /* define function to show a tree node */
AST *tree;
{
    if ( tree->token != 0 ) printf(" %s", zztokens[tree->token]);
    else printf(" %d", tree->i);
}
void before() { printf(" ("); }
void after() { printf(" )"); }

main()
{
    ANTLR(expr(&root), stdin);       /* get an expr AST by passing &root */
    zzpre_ast(root, show, before, after);/* print out in LISP notation */
}
>>



expr  :  exp0 ( "\+"^ <<#1->token=LA(1);>> exp0 )* "\n"! ;
exp0  :  exp1 ( "\*"^ <<#1->token=LA(1);>> exp1 )* ;
exp1  :  "[0-9]+"
      ;
```

The actions in rules `expr` and `exp0` set the `token` field (defined in `AST_FIELDS`) to the current token of look-ahead (`LA(1)`). This tells the predefined `astpreorder()` function to print the token value and not the number value (field `i`). Note that we must pass the address of a root pointer to `expr` so that `expr` can make it point to the subtree constructed for that rule. The newline character is not included in the trees because of the `!` suffix. The `+` and the `*` characters are considered operators in our language and so they are modified by `^`. The job performed by the default AST node creation macro `zzcr_ast` could have just as easily been accomplished by removing `zzcr_ast` and changing rule `exp1` to be

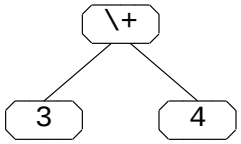
```
exp1  :  "[0-9]+" <<#1->i = $1; #1->token = 0;>>
      ;
```

which is somewhat more explicit, but would be inconvenient if many such actions were required.

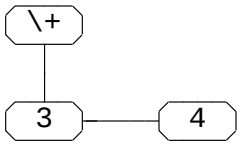
If the input to this PCCTS parser example were `3+4`, the program would print out

```
( \+ 3 4 )
```

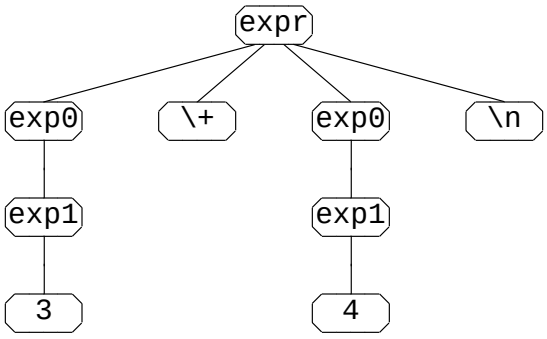
which represents



or, in child-sibling form



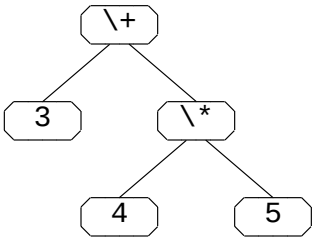
Note that the AST is significantly different than the parse-tree which can be represented by



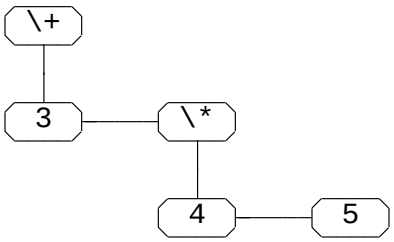
An input of `3+4*5` yields

```
( \+ 3 ( \* 4 5 ) )
```

which represents



or, in child-sibling form



The `4*5` would be performed before the addition because one of the addition's operands is itself an operation (i.e. a multiply).

4.2.4. Explicit AST construction

The automatic tree construction scheme available with PCCTS is useful for building AST's whose structure is similar to that of the grammar parse trees. It is not suited to applications where the structure of the grammar bears no resemblance to the structure required in the intermediate-form. C declarations are a good example of this and a sample "C declaration to English" translator is developed in this section.

Reasonable tree-construction and tree-rewriting systems, such as Purtilo and Callahan's [PuC89] NewYacc, have been developed but most rely on bottom-up parser generators like YACC to build a parse-tree in memory which is later traversed to simulate inherited attributes. PCCTS parsers construct AST's while parsing the input. Rules can pass subtrees as arguments to other rules and construct AST's without first having to build a parse-tree. PCCTS supports general AST construction via two simple tree description constructs — AST node constructors and AST tree definitions.

The AST node constructor is similar to the constructor for attributes (`$[`*token,text*`]`) except that AST node constructors yield pointers to AST nodes whereas attribute constructors yield attributes (of type `Attrib`). `#[`*args*`]` creates an AST node and passes a pointer to it to `zzmk_ast()` with *args* as arguments. Constructors may be nested to create AST nodes from attributes; e.g. `#[ $[`*token,text*`] ]`.

Trees are defined using a LISP-like notation because of its brevity.

`#( root, child`$_1$`, child`$_2$`, ..., child`$_n$ `)`

where `root` and `child`$_i$ are pointers to AST nodes which can themselves be trees or node constructors. The return tree for any rule is `#0` which can appear on both the left- and right-hand side of an assignment statement. Automatic construction of trees will proceed unless the `!` is included in the rule definition to override the normal AST mechanism. The same grammar may have some rules that use the automatic tree mechanism and other rules that explicitly construct trees. However, the methods must not be mixed within the same rule. In other words, do not assign a tree to `#0` unless you have turned off the automatic tree construction. When default AST construction is turned off, #-variables do not exist for terminals or rule references except

for references to rules that construct trees (explicitly or automatically). Nodes for terminals must be explicitly created just as the tree structure must be explicitly defined.

4.2.4.1. Simple example

We present a trivial example in this section to illustrate the AST node and tree constructors.

```
#header <<
    #include "int.h"
    #define AST_FIELDS int i;
>>


<<
AST *root = NULL;   /* define a root ptr for AST; must be NULL */

void show(tree)     /* define function to show a tree node */
AST *tree;
{
    printf(" %d", tree->i);
}
void before() { printf(" ("); }
void after() { printf(" )"); }

AST *zzmk_ast(node, v)
AST *node;
int v;
{
    node->i = v;
    return node;
}

main()
{
    ANTLR(tree(&root), stdin);
    zzpre_ast(root, show, before, after);
}
>>


tree! :  NUM NUM NUM "\n"
         <<#0 = #( #[$[NUM,"101"]], #[$1], #[$2], #[$3] );>>
      ;

#token "\ "   <<zzskip();>>
#token NUM "[0-9]+"
```

Given the input `3 4 5`, rule `tree` would return

```
( 101 3 4 5 )
```

The rule itself matches three integers followed by a newline. As usual, the four attributes `$1-$4` exist, but `#1-#4` do not because rule `tree` was defined with a `!` modifier. To create AST nodes for the terminals we use the node construct `#[$`*i*`]` which returns a new node constructed

through `zzmk_ast`. The newline is not to be included in the tree so no AST node was constructed for it. To illustrate the attribute constructor, a root node was created as a function of an attribute with a value of 101, though `#[101]` would have been easier. There is no corresponding terminal in the grammar for this AST node and therefore one was created. `$[NUM,"101"]` creates an attribute through `zzcr_attr` which is passed to `zzmk_ast` when enclosed in `#[]`. The subtree created in rule `tree` is explicitly returned by making `#0` point to the root of the subtree.

4.2.4.2. C declaration to English translator

The type declaration syntax of C is somewhat bizarre and can be troublesome to even experienced programmers. The syntax of declaration mirrors that of use. In other words, variables are defined the way they would be used in an expression. We give a loose description of how to correctly "parse" a C declaration and then we present a complete translator to describe, in English, what the declaration means.

Kernighan and Ritchie [KeR88] give an excellent description of how to understand C declarations and they present a program to convert from C to a word description. To parse a declaration as a Human, we follow a few simple rules.

(i) Start parsing at the variable name.

(ii) Scan to the right. For each `[`*expr*`]` or `()` encountered say "*n*-element array of" or "function returning" respectively where *n* is the (optional) expression found inside the brackets. Continue scanning until a right-parenthesis or a semicolon.

(iii) Now, scan left. For each `*` say "pointer to" until you see a type name or a left-parenthesis.

(iv) Move to the next outer nesting level if one exists and begin again at rule *(ii)*. If no more declaration symbols exist, look to the left and say the type name (ignoring storage classes here).

For example, let us parse

```
int *(*a[10])();
```

By *(i)*, we begin at `a`. Looking to the right in *(ii)*, we see brackets and so we say "10-element array of." The right-parenthesis makes us jump to rule *(iii)* and scan left. We see a `*` and say "pointer to." The left-parenthesis finishes off *(iii)* and rule *(iv)* tells us to exit this level of parenthesis (nesting) and start again at *(ii)*. Scanning to the right we see `()` and say "function returning." The semicolon forces us to *(iii)* where we see a `*` and say "pointer to." *(iv)* tells us to say "integer." Packing all of the "sayings" together yields: "10-element array of pointer to function returning pointer to integer" or, in a flat tree representation,

```
( [10] ( * ( () ( * int ) ) ) )."
```

Our example parser will construct trees of a similar form but with result in a little better English.

To build a type tree for C we "unroll" the nesting of the declaration so that it can be read left-to-right. Our trees will have the variable name at the root and the type as the leaf. Each node in the tree will be a type-qualifier. The type-qualifiers are one of `[`*expr*`]`, `()`, `*` and are linked together in a parent/child relationship. e.g. `( i ( * int ) )` means that `i` is a pointer to an integer.

The translator uses simple character buffers as attributes and AST node data. The trees are constructed at parse-time and later traversed to print out the description. A function to find the bottom of a parent/child list is required to augment the normal tree construction features of PCCTS.

```
#header <<
    #include "charbuf.h"
    #define AST_FIELDS  Attrib attr;
>>

<<
AST *root = NULL;

AST *zzmk_ast(node,attr)
AST *node;
Attrib attr;
{
    node->attr = attr;
    return node;
}

main() { ANTLR(start(&root), stdin); }


AST *bottom(t)
AST *t;
{
    if ( t==NULL ) return NULL;
    if ( t->down != NULL ) return bottom(t->down);
    return t;
}

english(tree)
AST *tree;
{
    printf("%s is", tree->attr.text);
    engl(tree->down, 0);
}
```

```
engl(tree, plural)
AST *tree;
int plural;
{
    if ( tree == NULL ) return;
    if ( strcmp(tree->attr.text, "*") == 0 )
        printf(plural?" pointers to":" a pointer to");
    else if ( strcmp(tree->attr.text, "nodim") == 0 ) {
        printf(plural?" arrays of":" an array of");
        plural = 1;
    }
    else if ( strcmp(tree->attr.text, "int") == 0 )
        printf(plural?" integers\n":" an integer\n");
    else if ( strcmp(tree->attr.text, "ret") == 0 )
        printf(plural?" functions returning": " a function returning");
    else {
        if ( plural ) printf(" %s-element arrays of", tree->attr.text);
        else printf(" a %s-element array of", tree->attr.text);
        plural = 1;
    }
    engl(zzchild(tree), plural);
    engl(zzsibling(tree), plural);
}
>>


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

start!  :   ( d <<english(#1); zzfree_ast(#1);>> )*
        ;

d!      :   <<AST *word;>>
            "int" d1[#[$1]] > [word] ";"        <<#0 = #(word, #2);>>
        ;

d1![AST *type] > [AST *word]
        :   "\*" d1[#(#[$1], $type)] > [$word]  <<#0 = #2;>>
        |   d2[$type] > [$word]                 <<#0 = #1;>>
        ;

d2![AST *type] > [AST *word]
        :   WORD d3[$type]                      <<$word = #[$1]; #0 = #2;>>
        |   "\(" d1[NULL] > [$word] "\)" d3[$type]
            <<#( bottom(#2), #4 ); #0 = (#2==NULL)? #4 : #2;>>
        ;

d3![AST *type]
        :   "\[" NUM "\]" d3[type]  <<#0 = #( #[$2], #4 );>>
        |   "\[" "\]"     d3[type]  <<#0 = #( #[$[WORD,"nodim"]], #3 );>>
        |   "\(" "\)"               <<#0 = #( #[$[WORD,"ret"]], type );>>
        |                           <<#0 = type;>>
        ;

#token NUM "[0-9]+"
#token WORD "[a-z]+"
```

The rules of this grammar perform the following functions

`start`
: Match multiple declarations translating each one to English and freeing each tree.

`d`
: Only integers are recognized for simplicity. Declarations begin with `"int"` and are terminated by a `";"`. Rule `d1` returns a pointer to the AST containing the symbol name found in the declaration (which has not yet been added to the tree). Rule `d` returns the declaration found in rule `d1` with the symbol name as the new root. The type node is always the leaf and is passed to rule `d1` which builds everything on top of the type node.

`d1`
: The `*` is the lowest priority symbol in a declaration and hence we "look" to the right first. The `*` is appended to whatever is found to the right. This is accomplished by passing the `*` and the currently built tree to rule `d2`. Since we are scanning from the left, but must translate as if starting from the most deeply nested structure, we must keep adding type-qualifiers on top of the root. The current tree is always passed down. Rule d1 returns a pointer to the AST node containing the variable name.

`d2`
: Recognize a variable name followed by an array/function descriptor or recognize a parenthesized declarator. Rule `d2` creates an AST node for the variable name and returns it. The tree constructed by rule `d3` is also returned. We need to attach the array/function descriptor to the bottom of the parenthesized declarator which could be arbitrarily large. For this reason, we use the `bottom()` function to find the end of the declarator returned by the reference to `d1`.

`d3`
: Match an array or function descriptor. Note that two tokens of look-ahead are required to decide between alternatives one and two since both begin with a left-bracket. Tail-recursion is used to get a sequence of `[]` or `()` descriptors. Array descriptors like `[34]` are converted to tree nodes containing just the number of elements. If no size is present, `"nodim"` is substituted for the number in the AST node. If nothing is seen by rule `d3`, it returns the type that was passed in since nothing was appended. Function descriptors form type-qualifiers labeled `"ret"`.

It is interesting to note that the grammar actions required to construct the type trees were small and mostly assignments. The explicit tree constructors allow complicated trees to be built without a great deal of C code.

The translator can be made using the following makefile.

```
# Makefile for C -> English translator
CFLAGS= -I../h
AFLAGS= -gt -k 2
GRM=decl
SRC=scan.c $(GRM).c err.c
OBJ=scan.o $(GRM).o err.o

test: $(OBJ) $(SRC)
    cc -o $(GRM) $(OBJ)

$(GRM).c parser.dlg : $(GRM).g
    antlr $(AFLAGS) $(GRM).g

scan.c : parser.dlg
    dlg -C2 parser.dlg scan.c
```

Our translator knows how and when to make things plural so it generates correct English. For example,

```
int *(*i[5])();
```

yields

```
i is a 5-element array of pointers to functions returning pointers to integers
```

4.2.5. Tree manipulation routines

PCCTS generates code that calls the following set of tree manipulation routines to create AST's. The user may modify the routines to suit their own purposes. The routines and `AST` node definitions are located in `ast.c` and `ast.h` respectively and reside in the PCCTS include directory. They are automatically included when the ANTLR `-gt` command-line option is specified. Routines prefixed with underscore are not normally called by the user.

`void _link(_root, _sibling, _tail)`

Called to ensure tree manipulation variables for current rule are valid after a rule reference.

`AST *_astnew()`

Create a new `AST` node and return it.

`void _child(_root, _sibling, _tail)`

Add a leaf node to the current AST. Creates an `AST` node, applies the default node transformation (`zzcr_ast`) if it exists, and appends the node to the current sibling list. After this function, *#i* variables are available for user actions.

`void _subroot(_root, _sibling, _tail)`

Perform the same operations as _child(), but make the newly-created node the root for the current sibling list. If a root node exists, make the newly-created node the root of the current root.

`void zzpre_ast(tree, apply, before, after)`

Perform a preorder traversal of `tree` applying `apply` to each node. Apply function `before` before the start of each new subtree and function `after` after you complete each subtree.

A common use of this function is to print out a tree. For example, if AST nodes had a field called token, the tree of tokens could be printed by defining the following

```
void show(tree)
AST *tree;
{
    if ( tree == NULL ) return;
    printf(" %s", zztokens[tree->token]);
}

void before() { printf(" ("); }
void after()  { printf(" )"); }
```

and then calling `zzpre_ast()` as

```
zzpre_ast(some_tree, show, before, after);
```

`AST *zzdup_ast(tree)`

Return a duplicate of the tree passed in.

`void zzfree_ast(tree)`

Free the AST pointed to by `tree` applying `zzd_ast` to each node before freeing if it is defined.

`zzrm_ast`

This macro is used to free the subtree for the current rule (using `zzfree_ast()`) and remove all references to it. PCCTS maintains local pointers to the current sibling list etc... which must be `NULL`'ed in order to begin a new subtree. Translators that perform an operation on a list of items using `()*` or `()+` can add an action to free and remove references to the AST associated with the different items upon each iteration of the loop. e.g. `(item <<zzrm_ast;>>)+`.

`zzchild(tree)`

Return the first child of the node pointed to by `tree`. Returns `NULL` if `tree` is `NULL`.

`zzsibling(tree)`

Return the immediate sibling of the node pointed to by `tree`. Returns `NULL` if `tree` is `NULL`.

The parameters `tree`, `_root`, `_sibling` and `_tail` are defined as

```
AST *tree, **_root, **_sibling, **_tail;
```

A useful modification to `_astnew()` would be to use custom allocation of `AST` nodes. Since they are all the same size, it could be done quickly and efficiently.

4.2.6. AST references

References to AST objects come in three basic forms.

#i This attribute refers to the tree constructed for the i^{th} rule element within the current alternative. *#i* variables are scoped/numbered just like *$i* variables and their association dissolves after leaving the current attribute scope. *#i* is unique until the current block is exited or another iteration of a `( )*` or a `( )+` loop has begun.

#0 is special and always refers to the root of the subtree associated with the rule in which it was referenced. `#0` is scoped like `$$`, not like `$0`; there is only one `#0` for the entire rule. If `#0` is ever set to `NULL`, the current subtree disappears (without freeing the nodes). Normally, `#0` is only set when the automatic AST tree creation is turned off with a `!` after the rule name in a definition.

Since subrules do not have their own `#0` values. The *#i* value of a subrule is undefined. However, for compatibility with *$i* numbering, subrules are counted as though they were assigned *#i* values:

An example #-variable number is as follows.

```
a   :   B
        ( C D /* #1 is C, #2 is D */ )*
        E
        /* #1 is B, #2 does not exist, #3 is E */
    ;
```

When the automatic tree construction is turned off for a rule, #-variables for terminals are undefined since nodes are not created for them. #-variables always exist for rule references since the referenced rule may construct a tree (implicitly or explicitly).

`#[`*args*`]`

This form constructs an AST node and returns a pointer to it (of type `AST *`). The arguments and the newly-constructed node are passed to the `zzmk_ast()` function defined by the user.

`#(root, child`$_1$`, child`$_2$`, ..., child`$_n$`)`

This form represents a tree descriptor from a set of AST node pointers. It is translated to a function call that links all the `child` nodes together as siblings, with `root` as their parent. A pointer to the root is returned. If the root is `NULL`, a pointer to the sibling list is returned. `#( NULL )` and `#()` both return `NULL`. `#(` *nodeptr* `)` yields *nodeptr*. All `child`'ren become siblings of each other and any siblings of the `child`'ren that existed previously become siblings also. For example,

```
#( NULL, #( NULL, #[a], #[b]), #[c] )
```

results in *a*, *b* and *c* all being siblings.

4.2.7. Invoking parsers that construct trees

PCCTS parsers pass an address of a root pointer to other rules while constructing AST's. This root pointer is always the first parameter and will point to the subtree built for that rule upon rule completion. Any parameters defined for a rule are not affected by this additional rule argument. All root pointers must be set to NULL initially. Normally, PCCTS generates code to handle all pointer parameter passing and tree manipulations. However, the starting rule (i.e. the one called using the `ANTLR` macro) may try to reference the root pointer it presumes was passed in. Therefore, the address of a `NULL`-initialized root pointer must be manually passed in. e.g.

```
AST *root = NULL;

main()
{
    ANTLR(start(&root), stdin);
}

start : stuff ;
```

This root parameter will be not be defined in rule `start` if the `-gt` command-line option is not used.

Note that if `main()` is located in a file other than a grammar file, the user must explicitly include `ast.h`.

4.2.8. Error recovery and AST's

When a syntax error is discovered by an ANTLR-generated parser, no further additions will be made to the subtree of the current rule. Additions will resume after ''successful'' error recovery. The AST is always left in a stable state (i.e. a valid child-sibling tree) so that user actions may examine and/or modify the tree. This also ensures that syntax errors will not result in ill-formed trees.

5. Error detection and repair

Reasonable error recovery is a notoriously difficult task which is handled very poorly, or not at all, by most parser-generator systems. In part, this may be due to the fact that most systems construct LR parsers, and these are theoretically less able to recover from errors than LL parsers [FiL88]. It can also be justified on the basis that a parser can be used to quickly find the first error, so that an interactive user can repair the error and recompile without having to read through a multitude of additional error messages which might represent the parser's confusion rather than actual errors.

In contrast, PCCTS attempts to provide a simple, automatic, error reporting and recovery mechanism. Currently, ANTLR-generated parsers recover from syntax errors by consuming tokens until a token in the FOLLOW set of ξ is discovered where ξ is the rule in which the error was discovered. Errors are reported in terms of the set of tokens which could legally have appeared at that point.

A syntax error occurs when a PCCTS parser discovers a token on the input stream that cannot be matched to any currently recognizable rule element. Because PCCTS performs grammar analysis on the user's grammar description, it is aware of all possible tokens that can be recognized at a given instant. Sets representing these tokens are passed to an error reporting function when a syntax error is discovered. Optionally, a fail-action can be executed.

[Wir76] provides a simple mechanism for recovering from syntax errors in LL(1) parsers by consuming tokens until the current token is found to be a member of a ''stopping set.'' This stopping set is comprised of the FOLLOW set for that rule and any user-defined marking (''resynch'') tokens that are never to be skipped (e.g. `BEGIN`). PCCTS currently employs a simple variant of the [Wir76] mechanism. When a syntax error occurs in any production of rule ξ, the parser will consume input until the current token is a member of the stopping set for ξ.

The syntax error recovery mechanism employed by PCCTS does not effect parser recognition speed unless an error is discovered. PCCTS error recovery is attractive because it is completely automatic, but it tends to delete groups of tokens which may lead to erroneous resynchronizations. This ''glutton'' scheme could be refined by allowing the user to specify a set of resynch tokens to augment the FOLLOW set [Wir76]. These resynch tokens would be determined by experience.

Sophisticated error recovery is possible using k tokens of look-ahead. Future versions of PCCTS might employ this large look-ahead for intelligent error recovery. Hence, it can be expected that automatic error handling will improve without requiring users to change the input to PCCTS.

5.1. Modifying grammars to recognize common syntax errors

The user may modify a given grammar to recognize common, but erroneous sentences. For example, if rule `expr8` below were part of an expression recognizer for a programming language,

```
expr8   :    VAR
        |    INT
        |    FUNC "\(" elist "\)"
        ;
```

it could be modified to print an error message whenever an undefined variable was seen. In other words,

```
expr8   :    VAR
        |    INT
        |    FUNC "\(" elist "\)"
        |    WORD <<error("Undefined variable: %s", $1);>>
        ;
```

Without this additional production, the parser would print something like (assuming `test` was the undefined variable found):

```
syntax error at "test" missing { VAR INT FUNC }
```

The extra production would allow a more informative error message to be printed, such as:

```
Undefined variable: "test"
```

5.2. Default error reporting function

The default error reporting function, `zzsyn()`, is defined as follows.

```
void
zzsyn(char *text,     /* text of current token */
      int tok,        /* current token */
      char *egroup,   /* label attached to rule with error if present */
      unsigned *eset, /* etok==0 -> >1 token missing; set is eset */
      int etok)       /* etok>0 -> 1 token missing; token is etok */
{
    fprintf(stderr, "syntax error at \"%s\"", text);
    fprintf(stderr, " missing ");
    if ( !etok ) zzedecode(eset);
    else fprintf(stderr, "%s", zztokens[etok]);
    if ( strlen(egroup) > 0 ) fprintf(stderr, " in %s", egroup);
    fprintf(stderr, "\n");
}
```

A list of the regular expression or label associated with each token value is stored in the `char *zztokens[]` array in `err.c`. `zzedecode()` is a predefined function that decodes the set of tokens stored in the bit-set `eset`.

`zzsyn()` can be modified according to the user's needs. For example, it can be modified to print out the file and line number (`zzline`).

5.3. Error groups

The user may attach a string to a non-terminal that is passed on to the error reporting macro/function `zzsyn()` by placing it just before the `:` in a rule definition. e.g.

```
declarator "declaration or definition" : ... ;
```

Although `zzsyn()` can be redefined by the user, the default error reporting facility appends the specified string to the error message. More than one rule may share the same string. This mechanism can generate decent error messages. For example, one could specify that all rules dealing with expressions (e.g. `factor`, `term`, `conditional`) are to be labeled as `"expression"`. The default `zzsyn()` would then yield something like:

```
syntax error at "NUM" missing { "\+" "\*" } in expression
```

as opposed to

```
syntax error at "NUM" missing { "\+" "\*" }
```

Another example might be that all rules used to define variables, such as rule `declarator` above, should be labeled `"declaration or definition"`.

5.4. Error token classes

The default syntax error reporting facility generates a list of tokens that could be possibility matched when the erroneous token was encountered. Often, this list is large and means little to the user for anything but small grammars. For example, an expression recognizer might generate the following error message for an invalid expression, `"a . b"`:

```
syntax error at "." missing { "\+" "\-" "\*" "\/" "\&" "\|" "\<" "\>" "\=" ";" }
```

A better error message would be

```
syntax error at "." missing { operator ";" }
```

This modification can be accomplished by defining the error class:

```
#errclass "operator" { "\+" "\-" "\*" "\/" "\&" "\|" "\<" "\>" "\=" }
```

5.4.1. Error class definitions

The general syntax for `#errclass` is as follows:

#errclass *label* { e_1 e_2 ... e_n }

where *label* is either a quoted string or a label (capitalized just like token labels). The quoted string must not conflict with any rule name or regular expression. Groups of expressions will be replaced with this string and it must not appear to be a simple token. Similarly, an error class label may not share the same name as a labeled token. The error class elements, e_i, can be a labeled token, a regular expression or a non-terminal. Tokens (labeled or regular expressions) referenced within an error class must at some point in the grammar be referenced in a rule or explicitly defined with `#token`. The definition need not appear before the `#errclass`

definition. If a non-terminal is referenced, the FIRST set (set of all tokens that can be recognized first upon entering a rule) for that rule is instantiated into the error class. The `-ge` command-line option can be used to have PCCTS generate an error class for each non-terminal of the form:

```
#errclass Rule { rule }
```

which implies that each non-terminal will have an error class composed of the first set for that rule. The error class name is the same as the non-terminal except that the first character is converted to upper case.

Error class names may also be used as error class elements (e_i) since error class names are treated somewhat like tokens. This capability allows error class hierarchies. For example,

```
#errclass Fruit { CHERRY APPLE }
#errclass Meat { COW PIG }
#errclass "stuff you can eat" { Fruit Meat }

yum :  (CHERRY | APPLE) PIE
    |  (COW | PIG) FARM
    |  THE (CHERRY | APPLE) TREE
    ;
```

Different error messages will result depending upon where in rule `a` a syntax error is detected. If the input were

```
THE PIG TREE
```

the following error message would result:

```
syntax error at "PIG" missing { Fruit }
```

However, if the input were

```
FARM COW
```

the decent error message

```
syntax error at "FARM" missing { "stuff you can eat" THE }
```

would result. Note that without the error class definitions, the error message would have been:

```
syntax error at "FARM" missing { CHERRY APPLE COW PIG THE }
```

which conveys the same information, but at a much lower level.

5.4.2. Error class utilization

PCCTS attempts to construct sets of tokens for error reporting--error sets. The sets are created wherever a parsing decision will be made in the generated parser. At every point in the parsing process there exists a set of currently recognizable/acceptable terminals. This set can be decoded and printed out when a syntax error is detected — when the current token is not currently recognizable. PCCTS attempts to replace subsets of all error sets with error classes defined by the user. For example, rule `a` below contains a subrule with more than one alternative which implies that a parsing decision will be required at run-time to determine which alternative to choose.

```
a   :  (Happy | Sad | Funny | Carefree) Person ;
```

If, upon entering rule `a`, the current token is not one of the four terminals found in the alternatives, a syntax error will have occurred and the following message would be generated (if `"huh"` were the current token):

```
syntax error at "huh" missing { Happy Sad Funny Carefree }
```

Let us define an error class called `Adjective` that groups those same four tokens together.

```
#errclass Adjective { Happy Sad Funny Carefree }
```

Now, the error message would be:

```
syntax error at "huh" missing { Adjective }
```

PCCTS repeatedly trys to replace subsets of the error set until no more substitutions can be made. At each replacement iteration, the **largest** error class that is completely contained within the error set is substituted for that group of tokens. One replacement iteration may perform some substitution that makes another, previously inviable, substitution possible. This allows the hierarchy mechanism described above in the error class description section. The sequence of substitutions for the `yum` example would be:

```
[i]     { CHERRY APPLE COW PIG THE }
[ii]    { Fruit COW PIG THE }
[iii]   { Fruit Meat THE }
[iv]    { "stuff you can eat" THE }
```

The error class mechanism leads to smaller error sets and can be used to provide more informative error messages.

6. Things you should know about

This section is a collection of important things that did not properly belong in any other section.

6.1. Available program symbols

This section describes all symbols visible to the user that may be of interest.

6.1.1. Macros, functions, `#define`'s

```
ZZCOL                 /* define so that DLG tracks column # of tokens */
ZZLEXBUFSIZE          /* if not set, default size of lexical bufs */
ZZA_STACKSIZE         /* if not set, default size of attrib stack */
ZZAST_STACKSIZE       /* if not set, default size of AST stack */
zzTRACE(rule)         /* macro called at start of rule when -gd set */
LL_K                  /* == max # tokens of look-ahead used in parser */
ANTLR(start,stream)   /* ANTLR startup macro; get input from stream */
ANTLRf(start,funcPtr) /* ANTLR startup macro; get input from func */
zzcr_attr()           /* create attribute from token, text on input */
zzd_attr()            /* call zzd_attr(&($i)) for each attrib created */
zzcr_ast()            /* how to create an AST node from attribute */
zzmk_ast()            /* how to create an AST node (user-defined) */
zzd_ast()             /* how to destroy an AST node */
LA(i)                 /* ith token of look-ahead */
LATEXT(i)             /* text for ith token of look-ahead */
zzlextext             /* buffer holding text of current token */
zzsyn()               /* error reporting macro */
zzpre_ast(tree, func1, func2, func3) /* see section 4.2.5 */
zzfree_ast(tree)      /* see section 4.2.5 */
zzdup_ast(tree)       /* see section 4.2.5 */
zzedecode(unsigned *error_set); /* decode a bit-set */
zzskip()              /* Tell DLG to skip this token */
zzmore()              /* Tell DLG to try for more chars */
zzmode(mode)          /* switch DLG to another lex mode */
zzadvance()           /* Get next char allow only valid chars */
zzrdstream(s)         /* Reset lex_line, use stream s for input */
zzrdfunc(f)           /* Reset lex_line, call function f to read input */
zzreplchar(c)         /* replace current lex buffer with c */
zzreplstr(s)          /* replace current lex buffer with s */
```

6.1.2. Global variables

```
char    *zztokens[],  /* zztokens[t] is rexpr or label for token t */
        *zzbegexpr,   /* start of text for currently recognized expr */
        *zzendexpr;   /* end of text for currently recognized expr */
int     zzline,       /* set by user to current line # */
        zzbegcol,     /* column # of 1st char in token */
        zzendcol,     /* column # of last char in token */
        zzchar;       /* current character of look-ahead */
void    (*zzerr)();   /* ptr to func to exec upon lexical error */
```

One of the most common actions found among PCCTS descriptions is a statement similar to

```
#token "\n"    << zzline++; >>    /* Move to next line */
```

6.2. ANSI versus K&R

K&R, actually Kernighan and Ritchie, ‘‘The C Programming Language,’’ Prentice-Hall, 1978, was effectively the standard for the C language until the ANSI X3-J11 committee put forth a definition of ANSI C [ANS90]. Unfortunately, ANSI C is not compatible with K&R C, yet both dialects are in common use. For this reason, PCCTS supports generation of either ANSI C or K&R C.

By default, PCCTS generates K&R C code with parameterless `extern` declarations automatically placed in `tokens.h` for each parsing function that has a return value. If the `-ga` command-line option is used (on both ANTLR and DLG), PCCTS generates ANSI C compatible code and function prototype declarations; all parsing functions with return values or parameter definitions have appropriate prototypes placed in `tokens.h`.

According to the ANSI C standard, a compliant compiler must define the macro name `__STDC__`, which would not be defined by a K&R C compiler. Hence, code can be written to work correctly with both ANSI C and K&R C by using `#ifdef` to test if `__STDC__` has been defined. Although the output of PCCTS would have been excessively large if `#ifdef` had been used to make the code be acceptable to both ANSI C and K&R C compilers, `#include` files and other support code may test for the macro `__STDC__`.

6.3. Bugs

Occasionally, PCCTS will find a grammar that it cannot analyze and you will receive an error similar to the following.

```
file.c, line 43: out of memory while analyzing alts 2 and 5 of (..)+
```

This is due to LL(k) grammar analysis requiring exponential space in some cases. Normally, this problem only will occur with k > 2 for large, ambiguous, grammars. A technique which avoids the worst-case exponential space requirement is being considered for a future release.

Although not technically a ‘‘bug,’’ the differences between β release PCCTS and version 1.00 will cause numerous errors to be reported if old grammars are submitted to the new system. Most of these incompatibilities require only minor changes; see section 7.1. This was expected — why do you think we called it β?

The code generated using the `-ga` option has not been tested for compliance with the ANSI C standard.

The source code for ANTLR itself is not ANSI compliant, although it is fairly close.

Error messages reporting ambiguities involving nullable subrules (i.e., `{}` or `()*`) sometime blame the ambiguity on an alternative rather than the nullable construct.

The names of PCCTS variables and/or functions are not necessarily consistent, mnemonic or convenient. PCCTS symbols are, therefore, subject to change.

ANTLR will not compile on any system that does not have `_iobuf` as the standard `FILE *` structure name. This is because `struct _iobuf *` is used in a header file instead of `FILE *`. The header file is automatically generated via `proto`--the C example distributed with PCCTS. You can circumvent this problem by changing the declaration of `struct _iobuf *` in `proto.h` to `FILE *`.

ANTLR normally checks to see that you do not reference an attribute (parameter, return value, normal attribute) that is not defined. However, if you use a `$`*name* where *name* is a substring of a valid parameter or return value, it thinks that it is ok.

Invoking an ANTLR parser multiple times or invoking multiple parsers does not work very well; tokens tend to be lost.

7. History

PCCTS Version 1.00 was written using β release PCCTS — 1.0B. 1.0B was, in turn, written in alpha release PCCTS, 1.0A, which was written in YUCC (a graduate class project gone wild). YUCC was written in PG; both of which were simple BNF-style recursive descent parser generators. PG was written in RD [JuD84].

7.1. Differences from β release PCCTS

PCCTS Version 1.00 is significantly different from β release PCCTS. However, Version 1.00 is mostly a superset — most old grammars require only minor changes for version 1.00. This section delineates all of the changes and a few possible migration paths.

- `#attrib` is now called `#header` to reflect the more general use of the construct in version 1.00.
- The `ANTLR` initiation macro now requires that the starting rule have `()` appended just as you would call the function by hand. For example, `ANTLR(start(), stdin)`.
- A number of internal functions and variables have been renamed so that conflicts with user-defined names are less likely. All the new names begin with the prefix `zz`. For example:
  - `aCreate()` is now `zzcr_attr()`.
  - `cur_char` is now called `zzchar`, but the user should not need to reference it.
  - `LexSkip` and `LexMore` are now `zzskip` and `zzmore`.
- `Token` is now called `LA(1)` (first token of look-ahead). `LA(`*i*`)` is the i^{th} token.
- `LexText` is now `LATEXT(1)` (text for first token of look-ahead). `LATEXT(`*i*`)` is the text for the i^{th} token.
- `ANTLRi` no longer exists.

- Rules no longer automatically define an attribute parameter. Also, parameters of the form `{...}` (i.e. `rule[{...}]`) cannot be used. Basically, parameter passing to a rule is exactly like in a programming language. All parameters must be defined and can be any valid type. Also, return values may be defined for rules rather than using old-style upward attributes. Rules may simulate the old-style `$0` inheritance via

```
rule[Attrib inh] : <<$0 = inh;>> ... ;
```

- A new file called `mode.h` is created by DLG and is included into your ANTLR parser. This implies that DLG must be run before you can compile the C files created by ANTLR.
- New command-line options have appeared, old ones have been changed and/or disappeared. To be precise,

  `-T` is now `-gd`.

  `-z` is gone.

  `-n` is now `-gx`.

  `-l` is now `-fl`.

  `-c` is now `-gc`.

  `-I` is gone.

  `-s` is gone.

  `-fi` is gone.

  `-fo` is gone.

  `-R` is now `-cr`.

  `-e` is gone.

  `-a` is gone.

  `-w` is now one of `-e1`, `-e2`, `-e3`.

  `-d` is gone.

  `-E` is gone since the user can redefine `zzsyn()`.
- `zzreplchar()` and `zzreplstr()` should be used instead of modifying the lexical buffer manually.
- A pointer to a function called `zzerr` is set to a function to call upon lexical error (invalid token etc...).

7.2. Future directions

Neither we nor Purdue University guarantee that PCCTS works or accept liability if it does not do what you want it to. However, we do try to maintain and improve the system. So, we value user feedback that might help us improve PCCTS.

Within Purdue University's School of Electrical Engineering, we are constantly updating PCCTS. The 1.00 release does not include all the latest goodies — it contains only those features whose implementation we believe to be useful, correct, and reasonably stable. The following is a partial list of improvements currently being considered:

Parser construction

Currently, ANTLR parsers use a sequence of `if` statements to find the correct alternative to match. This is inefficient if only one token of look-ahead is required to uniquely determine which alternative to choose. Depending on the C compiler, a `switch` statement would generate much faster code.

Attribute and AST variables are maintained on software stacks in 1.00. Grammar analysis could easily create a collection of local hardware-stack-based variables to use in place of the software stack. This would eliminate the need to specify an attribute or AST stack size and produce faster code.

Inline rules

For the most part, subrules are like macro-expansions of rules. Rather than referencing a rule, the user simply instantiates the rule in place of the rule reference. Grammatically the two methods are the same, but they differ in two important ways. First, subrules cannot employ the sophisticated parameter passing mechanism available to rules. Second, subrules are much faster than rule references since subrules avoid the function call overhead. A future version of PCCTS might introduce **inline** rules. Inline rules would be defined as before (with the addition of the `inline` keyword) but would be instantiated into rules that reference them. This would cut down on grammar clutter by breaking up long rules while maintaining the speed advantage of subrules.

Abstract-syntax-trees

Through experience with PCCTS AST's, a new group of tree manipulation routines should appear. Some of them could relate to code-generation.

Example grammars

More and more grammars are being built with PCCTS. Some of them might be made available through future PCCTS releases and through users' network postings.

Recognition

Currently, PCCTS parsers cannot recognize as large a class of languages as bottom-up parsers. Research is underway to create hybrid parsers that would retain top-down's programmer-interface features while gaining bottom-up's recognition strength.

Code generation

Research continues at Purdue toward a code-generator generator for parallel machines (PIG).

A simple uni-processor code-generator that has been in use at Purdue may be made available in the next release of PCCTS.

\$*token*PCCTS only recognizes \$*i*, \$*i.j*, \$*rule* variables presently. Future versions might allow \$*token* variables which will refer to attributes for tokens by the name of the token not the position number.

7.3. Acknowledgements

Thanks is due to the users of β release PCCTS for their excellent suggestions. In particular, Mark Nichols (PhD candidate in Electrical Engineering at Purdue) discovered many of the initial quirks and bugs. We acknowledge Dana Hoggatt (Micro Data Base Systems Inc) for his idea of error grouping (strings attached to non-terminals) and his significant software testing efforts. Thanks is due Professor Matthew O'Keefe and Peter Dahl (both at University of Minnesota) for their review of the user's manual and their bug-finding escapades into PCCTS.

8. References

[ANS90]

American National Standard for Information Systems, "Programming Language C," Document X3J11/90-013, February 1990.

[AhU79]

A.V. Aho, J.D. Ullman, Principles of Compiler Design, Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

[DoP90]

H. Dobler and K. Pirklbauer, "Coco-2 A New Compiler Compiler," ACM SIGPLAN Notices, Vol. 25, No. 5, May 1990.

[Joh78]

Stephen C. Johnson, "Yacc: Yet Another Compiler-Compiler," Bell Laboratories, Murray Hill, NJ, 1978.

[JuD84]

R. Juels, H. Dietz, S. Arbeeny, J. Patane, E. Tepe, "Automatic Translation Systems," Study Report, Center For Digital Systems, Department of Electrical Engineering and Computer Science, The Polytechnic Institute of New York, New York, 1984.

[KeR88]

Brian W. Kernighan and Dennis M. Ritchie, "The C programming Language," Prentice Hall Inc., Englewood Cliffs, New Jersey, 1988.

[Les75]

M. E. Lesk, "LEX--a Lexical Analyzer Generator," CSTR 39 Bell Laboratories, Murray Hill, NJ, 1975.

[LeP81]

Harry R. Lewis, Christos H. Papadimitriou, Elements of the Theory of Computation, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Nau63]

P. Naur (ed.), "Revised report on the algorithmic language ALGOL 60," Communications of the ACM 6:1, 1-17.

[PaD90]

Terence Parr, Henry Dietz, Will Cohen, "Purdue Compiler-Construction Tool Set," Technical Report TR-EE 90-14, School Of Electrical Engineering, Purdue University, West Lafayette, IN, February 1990.

[Par90]

Terence Parr, "The Analysis of Extended BNF Grammars and the Construction of LL(1) Parsers," Technical Report TR-EE 90-30, School Of Electrical Engineering, Purdue University, West Lafayette, IN, May 1990.

[PuC89]

James J. Purtilo and John R. Callahan, "Parse-Tree Annotations", Communications of the ACM, Vol. 32, No. 12, December 1989.

[Str87]

Bjarne Stroustrup, "The C++ Programming Language," Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[Wir76]

Niklaus Wirth, Algorithms + Data Structures = Programs, Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1976.

Notes:[LeP81] credits P. Naur [Nau63] with Backus-Naur Form (BNF) and is a reasonable text on language theory. [AhU79] gave more complete references to Johnson's YACC and Lesk's LEX — i.e. the CSTR numbers from Bell Laboratories.

Table of Contents