TABLE OF CONTENTS

PREFACE 1

I. OVERVIEW 2

1. Typographical Conventions 2

2. Manual Organization 3

II. OVERVIEW OF PCYACC 5

1. History 5

2. What Does PCYACC Do ? 6

III. COMMAND LINE AND OPTIONS 8

1. Command Line Format 8

2. Command Line Options 8

IV. BASICS OF PROGRAMMING LANGUAGES 16

1. What Is A Computer Programming Language? 16

2. What Are Compilers 17

V. GETTING STARTED -- A SMALL EXAMPLE 19

1. Grammar Description File for SACALC 19

2. Building the Executable 24

3. Sample Session of SACALC at Work 25

VI. BASIC CONCEPTS REVISITED 26

1. General Concepts 26

2. BNF -- A Language for Writing CFG's 28

3. PCYACC Terminology's -- A Short Review 30

VII. INTOPOST -- A SECOND EXAMPLE 32

1. Problem Statement 33

2. Defining the infixel Language 35

3. Writing PCYACC Grammar Description For infixel 36

4. Converting Grammar Descriptions Into C Programs 42

5. Building the Executable **43**

VIII. PRINCIPLES BEHIND PCYACC **44**

1. Introduction to Formal Language Theories **44**

2. Context-free Grammars **45**

3. Context-free Languages **49**

4. Parse Trees **49**

5. Canonical Derivation and Canonical Reduction **51**

6. Top-down and Bottom-up Parsing **52**

7. Ambiguities of Context-free Grammars **53**

8. LR Parsers **54**

9. PCYACC -- From LALR Grammars to LALR Parsers **57**

IX. WRITING PCYACC GRAMMAR DESCRIPTIONS **58**

1. Structure of Grammar Description Programs **58**

2. The Declaration Section **60**

3. The Grammar Rule Section **62**

4. The Program Section **62**

5. Associating Actions with Grammar Rules **62**

X. MORE ON PCYACC PROGRAMMING **66**

1. Mandatory Supporting Functions **66**

2. The Role of the Drive Routine **67**

3. The Role of the Lexical Analyzer **68**

4. The Role of PCYACC Generated Token Definitions **71**

5. The Role of the Error Processing Routine **71**

6. Data Types of Grammar Symbols **71**

7. Defining YYSTYPE **72**

8. Associating Types with Grammar Symbols **73**

9. Manipulating Values of Grammar Symbols **74**

10. Ambiguity Resolution Mechanisms **75**

11. Resolving Shift/Reduce Conflicts 78

12. Resolving Reduce/Reduce Conflicts 80

13. Resolving Ambiguities -- A Summary 82

14. Error Recovery Utilities 83

15. Imbedded Actions 86

XI. DEBUGGING -- TOOLS AND TECHNIQUES 87

1. Correcting Syntax Errors 87

2. Correct Symbol Usage Errors 89

3. Correcting Grammar Rule Errors 91

XII. CONSTRUCTING COMPILERS -- REVISITED 99

1. Basic Architecture 99

2. Lexical Analysis 101

3. Syntax Analysis 103

4. Semantic Analysis 104

5. Code Generation 108

6. Symbol Table Management 110

7. Error Diagnostics 114

XIII. YAEC -- YET ANOTHER EXAMPLE COMPILER 116

1. Global Definition Head File 117

2. Lexical Token Definition Header File 118

3. Lexical Analysis Module 119

4. Syntactical Analysis Module 121

5. Semantical Checking 123

6. Main Routine and Error Handling Module 126

7. Execution Module 127

XIV. UNIQUE FEATURES OF PCYACC 130

1. Quick Syntax Check Option 130

2. Generating Parse Trees Using PCYACC 130

4. Lexical Analysis Caveats - Combining Lex & Yacc 133

XV. ERROR PROCESSING WITH PCYACC 135

1. Error Reporting 135

1.1. Integration with a Lexical Scanner 136

1.2. YYERROR Calling Conventions 136

2. Error Handling 137

2.1. Simple Recovery 138

2.2. Improved Recovery 138

2.3. Doing Your Own Parser Recovery 140

2.4. The ANSI C Parser 141

3. Error Recovery 142

3.1. Error Recovery in IMPROVED.Y 142

3.2. Error Recovery in PIC 142

4. Building Parsers 144

4.1. TOKENS program 144

4.2. Command Line Format 144

4.3. Command Line Options 145

4.4. Using Command Line Options 145

5. Wrapup 146

XVI. USING PCYACC WITH C++ AND MICROSOFT WINDOWS 147

XVII. PCYACC AND PCLEX RECURSION 148

XVIII. PCYACC CROSS REFERENCING WITH - YACC TOOL 150

1. Create GRAMMAR FOREST from YTOOL.EXE 151

XIX. Invoking pcYaccDeBugger - YDB 152

APPENDIX I. INSTALLING PCYACC 154

1. System Requirements 154

2. Unpack and Backup the PCYACC PROGRAM Disk 154

3. Installing PCYACC 156

APPENDIX II. ERROR MESSAGES 157

APPENDIX III. ANNOTATED BIBLIOGRAPHY 161

1. GENERAL REFERENCES 161

2. Error Recovery References 164

APPENDIX IV. GLOSSARIES 165

i. General Glossaries 165

ii. PCYACC Specific Glossaries 167

iii. PCYACC INPUT SYNTAX SUMMARY 170

PREFACE

The ABRAXAS PCYACC is a parser generator. A Parser Generator is a program that produces syntactical analyzers automatically for language translators from high level descriptions of the languages. Like any other program, PCYACC accepts inputs and produces outputs. The inputs to PCYACC are syntax specifications of the programming languages to be developed. Syntax specifications are written using a special purpose language which is referred to by PCYACC as the grammar description language (GDL). The outputs from PCYACC are C implementations of corresponding language recognizers.

PCYACC is a powerful and practical programming tool for software developers. It can dramatically reduce the amount of work in programming language development projects. As a result, it can reduce the production cycle time and hence the cost of software projects.

I. OVERVIEW

This manual describes ABRAXAS PCYACC. In addition to the most important theme of providing a reference resource to this software product, the manual includes material that explains the basic principles of the program. The contents are useful to readers with a wide range of expertise, from professional software developers to novice programmers.

1. Typographical Conventions

PCYACC is largely independent of machine hardware characteristics. It will run on most micro computer systems, and can easily be adapted to others. Therefore, our presentation will also maintain system independence, wherever possible. When we need to refer to a particular system configuration in order to make discussions concrete, we use a MSDOS command shell.

In the appendix, we will provide some tips on system dependent characteristics of PCYACC, such as installations on different machines, and how to take advantage of other application tools such as intelligent editors.

Throughout this manual, we will use the following typefaces and syntax conventions to add clarity:

1). Words or phrases that have important technical meaning or that have special interpretations in PCYACC will be underlined when they appear the first time in the manual. (Appropriate definitions and/or explanations are also given at this time, using normal text style.)

Example: PCYACC programs are written in the grammar description language, or GDL, which is a BNF like language suited to syntactic specifications of programming languages.

2). When illustrating user-machine interactions, we will print the text to be typed by the user in italics. Messages or prompts from the machine will be indicated with a double underline. File names and operating system commands will be shown in uppercase (except when case is sensitive). The enter key will be represented as <ENTER>.

Example: *PCYACC SACALC.Y* <ENTER>

3). Displays, such as diagrams or code listings, will be printed in mono spaced font. They will be indented to help you distinguish them from surrounding text.

Example: The following is a header file for one of the examples you will see later in this manual.

```
#include <stdio.h>
#include <ctype.h>

#define  LINE 001
#define  BOX  002
...
```

4). Keywords in GDL will be printed in **boldface** when they appear in text.

Example: **token**, **type**, ...

2. Manual Organization

This manual contains fourteen (14) chapters. The contents of these chapters are as follows:

Chapter 2 is an overview of PCYACC; its origin, history, and evolution.

Chapter 3 contains information about PCYACC invocation, command line format and options.

Chapter 4 presents the most basic concepts about computer programming languages and compilers.

Chapter 5 steps through a simple arithmetic calculator program example constructed using PCYACC.

Chapter 6 is a continuation and/or extension of Chapter 4. It summarizes existing basic concepts about languages and translations and introduces some new ones.

Chapter 7 is a slightly more complicated example program that translates infix arithmetic expressions into their postfix notations. A step by step procedure is provided for the entire program development process. This Chapter is intended to give you more hands-on experience using PCYACC.

Chapter 8 is devoted to discussing fundamental principles of PCYACC (or LALR parser generators in general). The discussion is brief and informal, so serious readers need to consult relevant literature for more information. (Compilers: Principles, Techniques, and Tools, the "Dragon" book, is highly recommended for those interested in detailed theory.)

Chapter 9 describes the macro structures of PCYACC grammar description files. These files are created for input to PCYACC specifying the intended language syntax. Formats for writing grammar descriptions and the parts required for such descriptions are outlined.

Chapter 10 details PCYACC internals from the programmer's viewpoint. It overviews and describes features of PCYACC, illustrating how these features work. Several examples are also included.

Chapter 11 describes debugging techniques for program development using PCYACC. It lists common errors that a programmer might encounter and suggests ways to fix them. It also explains the debugging facility provided by PCYACC: parsing tables revealing internal behaviors of parsers generated by PCYACC.

Chapter 12 looks at the general principles of compiler construction. It describes a typical architecture for compilers and outlines each component's function.

Chapter 13 is intended as another hands on project example. A simple picture specification language is defined and a compiler for it is presented.

Chapter 14 describes unique features of PCYACC. It also provides some tips on how to use these features more effectively with existing software tools.

Appendix I states configuration requirements and provides installation instructions.

Appendix II is a list of error messages from PCYACC and their explanations.

Appendix III suggests a list of related literature for further reference.

Appendix IV is a glossary defining the PCYACC vocabulary.

II. OVERVIEW OF PCYACC

This Chapter provides a brief historical overview of PCYACC and a description of its functionality at an abstract level.

1. History

PCYACC is a descendant of YACC, which is the acronym of "Yet Another Compiler-Compiler". YACC was originally designed and implemented by Stephen C. Johnson of AT&T Bell Laboratories over ten years ago. It is one of the standard language tools provided in almost all UNIX operating systems environments. PCYACC then, is a MSDOS implementation of the original UNIX YACC as defined by Johnson.

During the past decade or so, numerous software projects, large and small, have been developed using YACC. As such, YACC has historically been one of the most valuable tools available to programming language developers. YACC can dramatically reduce the time and complexity normally involved with compiler development. Compiler writing tools, such as YACC, provide developers with a practical means of writing the so called LR parsers. LR parsers have a number of advantages over more traditional technologies, such as recursive descent parsing.

Because YACC is such a useful tool, many developers have ported it to their own systems, or re-implemented it in their own software environments. PCYACC is an example of an implementation of YACC on micro computer systems.

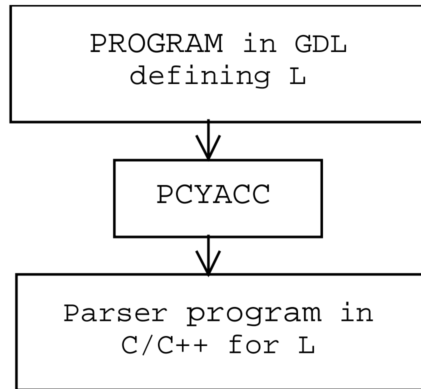
PCYACC is designed and implemented to be upward compatible with YACC. All the features of YACC are retained in PCYACC, some of which have been improved. In addition, a number of new functions have been incorporated into the design of PCYACC. Although PCYACC has an almost identical programming interface with its ancestor, there are a few differences. These will be discussed as we go along.

2. What Does PCYACC Do ?

PCYACC is no more than a computer program. In this respect, it is like any ordinary program; it takes inputs, computes values and produces results. There is a special class of programs, called compilers, which take computer programs as input, usually written in a high level programming language, and produces as results corresponding low level programs. (We will use the word compiler loosely to include other types of translators, unless otherwise specified. In the next Chapter we present a taxonomy.)

PCYACC is even more specialized, because it is a compiler generator, a program that writes compilers. PCYACC is a software tool that you will find useful when you wish to write your own compilers, or other similar programs such as command interpreters. (Or, for that matter, any program that needs to be able to read and understand one of the numerous high level languages. "Pretty printers", formatting utilities, cross reference generators, and specialized programs such as "LINT" are examples. Many of the GDL's necessary to parse high level languages are included in the Professional version of PCYACC.)

Based on what it does, PCYACC can still be classified as a compiler, a program that translates programs in one language into programs in another language. PCYACC uses a special programming language, which is referred to as a grammar description language (GDL). The object language of PCYACC is the C/C++ programming language, however the generation of Java, Basic, or Pascal is possible by using our Object Oriented Toolkit. Put simply, PCYACC takes programs written in GDL as input, and produces C programs as output by default. The resulting C/C++ programs are equivalent to the input GDL programs, in that a GDL program is a specification of the formal syntax for a programming language. The C program generated by PCYACC will be a parser for the corresponding programming language that satisfies the specification. The idea is illustrated by the following diagram:



NOTE: Although PCYACC (or YACC) is said to be a compiler generator, or compiler compiler as implied by its name, it is really a parser generator. PCYACC (YACC) can only generate a portion, often referred to as the front end, of a compiler.

III. COMMAND LINE AND OPTIONS

PCYACC is invoked by simply typing *PCYACC* while running under MSDOS command shell. This Chapter explains the command line format and possible options.

1. Command Line Format

PCYACC can be invoked by typing *PCYACC*, followed by zero or more command line options, followed by a file name. For example:

PCYACC [options] <gdf_name>

Where <gdf_name> is the name of the file containing a grammar description program (GDP), a program written in GDL, and [options] represents zero or more command line options. Files used to hold GDP's are called grammar description files, or GDF's for short.

Although there is no restriction on the format of input file names, it is a good practice to give PCYACC GDF's an extension field distinguishable from other kinds of files. Recommended extensions are ".y", ".Y", ".pcy" and ".PCY" and we will use ".y" throughout this document. PCYACC compiles the GDF <gdf_name> and produces a C program that is an LALR (look-ahead LR) parser for the language defined by the GDP. By default, the generated parser is kept in a file with an extension ".c" with the same basename as the GDF <gdf_name>.

If PCYACC is invoked without a grammar description file, it will display a short message advising you of the correct command line format.

2. Command Line Options

Command line options are used to override default actions or file name conventions, or to indicate actions you want PCYACC to perform in addition to what it does automatically. Available options are described below:

-c: This option overrides default C file name. Instead of using the basename of the grammar description file plus the ".c" extension, it uses "yytab.c". This option is provided to maintain compatibility with earlier versions.

-C<cf>: Like -c, this option overrides default C file name, but uses the name provided by the user, <cf>.

-d: This option tells PCYACC to produce a C header file, using the default file name "yytab.h", in addition to the C code file. This header file is used primarily by your lexical analysis routine yylex(). The definitions generated by PCYACC are used globally at parse time unless your yylex() routine is local to your grammar. PCYACC basically enumerates all of the tokens declared in the grammar, and these enumerated values are used as messages between yyparse() and yylex().

-D<hf>: Like -d, this option produces a C header file, but with a different file name convention. If no <hf> is provided, PCYACC will use the basename of the grammar description file with an extension ".h"; otherwise <hf> will be used instead.

-h: Print a help screen.

-n: Disable #line numbers from the .C output of PCYACC. This option is quite useful if your trying to use a source code debugger. In normal operation the output .C file uses #line to make the output relative to the original .y file, this normally causes source code debuggers like CodeView to generate strange results.

-p<pf>: Use the user provided parser skeleton contained in <pf> file instead of the system default (internal skeleton). A sample parser skeleton is supplied in the \src directory of the PROGRAM diskette. (yaccpar.c). The external parser skeleton is a commonly used to support multiple parsers.

-P<pf>: Same as -p<pf>.

-r: Report progress during execution. This is a good idea for huge grammars that take seemingly forever to compile.

-R: Report progress during execution.

-s: This option instructs PCYACC produces short integer internal arrays for the parser. The default type for the internal arrays is long integer.

-S: This option overrides PCYACC's default action. Instead of processing the grammar description file, it quits after the syntax analysis phase. This option is useful for doing syntax debugging

on large grammar description files, especially when coupled with an extensible text editor (see Chapter XIV).

-t: This option tells PCYACC to construct the parser in such a way that it will build a parse tree for the program being processed. The parse tree, by default, is saved to the file "yy.ast". (not compatible with the -p switch, requires internal skeleton parser). The parse tree is not actually generated until the parser is executed.

-T<tf>: Same as option -t, except with different file name conventions. If <tf> is not provided, the parse tree is saved to the file named by the basename of the grammar description file with an ".ast" extension; otherwise, it is saved to <tf>.

-v: This option produces a textual parsing table, in addition to the C parser, using the default file name "yy.lrt". The parsing table is a required tool in debugging PCYACC grammars.

-V<vf>: Same as option -v, except that the parsing table is saved to either <vf> or a file named by the basename of the grammar description file and the extension ".lrt".

NOTE: Command options may change over time. Consult the "readme.now" file in the root of the distribution disk for the latest information.

3. Using Command Line Options

This section shows you how to use command line options. The following example, HELLO.Y, is used throughout this section:

```
%token WORLD
%token HELLO
%start greetings

%%

greetings : HELLO ',' WORLD ;
```

At its default setting, PCYACC produces the C file HELLO.C for this example grammar.

3.1 Override Output C File Name Convention (-c and -C)

-c switch overwrites the output C file name convention of PCYACC. The C output is written to YYTAB.C.

Example: *PCYACC - c hello.y* <ENTER>

produces YYTAB.C in your current folder.

-C<cf> option lets you specify a name for the C output of PCYACC.

Example: *PCYACC -C greet.c hello.y* <ENTER>

produces GREET.C in your current folder.

3.2 Produce Token Definition File (-d and -D)

These two options let you have a separate file containing token definitions requested by the grammar file. The -d option instructs PCYACC to send the output to YYTAB.H.

Example: *PCYACC -d hello.y* <ENTER>

produces YYTAB.H in your current folder.

-D also gives you a header file. It allows you to specify a name for the file, or to use a different default.

Example: *PCYACC -D hello.y* <ENTER>

sends the header definitions to the file HELLO.H.

Example: *PCYACC -D hello.y* <ENTER>

produces the header definitions to the file GREET.H.

The contents of the three files, YYTAB.H, HELLO.H and GREET.H, are identical:

```
#define WORLD 257
#define HELLO 258
```

3.3 Ask PCYACC for Help (-h and -H)

The two options, -h and -H, do exactly the same thing. They ask PCYACC to print out a help message on the screen. This feature is especially useful to new users.

Example: *PCYACC -h* <ENTER>

displays the following message:

```
Abraxas Software (R) PCYACC version - 7.01.
Copyright (C) Abraxas Software, Inc. 1986 - 1997, All rights
reserved
```

```
Usage: PCYACC options gram.y
Available options:
-c : use file "yytab.c" for C output
-Cf: use file f for C output
-d : produce a header file in "yytab.h"
-Df: produce a header file in f or grm.h
-h : display this help message
-H : display this help message
-pf: use skeleton f as parser driver
-Pf: use skeleton f as parser driver
-s : produce short integer internal arrays
-S : quick syntax check on input file grm.y
-t : build parser with parse tree generation using "yy.ast"
-Tf: build parser with parse tree generation using f or grm.ast
-v : produce a parsing table for the parser using "yy.lrt"
-Vf: produce a parsing table for the parser using f or grm.lrt
```

3.4 Override System Default Parser Skeleton (-p and -P)

The -p option and the -P option have the same function. They let you write your own parser skeleton to drive the LALR table generated by PCYACC.

Example: *PCYACC -p myparser hello.y* <ENTER>

will use MYPARSER to produce HELLO.C.

3.5 Generate Short Integer Arrays (-s)

The option -s instructs PCYACC to generate short integer arrays for the parser. The default type for the internal arrays is integer. This feature is useful on systems that have severe memory constraints. (Note: "int" may mean different things to different C compilers; check your manual and change "int" to "short" if necessary. Apple MPW C thinks that "int" means a 32-bit value. Lightspeed™ C thinks that "int" means a 16-bit value.)

Example: *PCYACC -s hello.y* < ENTER>

will generate short integer arrays in HELLO.C. Most 32-bit compilers treat int as a long (32-bit) and short as 16-bit. In general this switch is used when saving space is essential and your working with small tables.

3.6 Quick Syntax Check (-S)

The -S option is useful for large grammar files. It instructs PCYACC to do a syntax check on the grammar file. No other processing is performed when this switch is used.

Example: *PCYACC -S hello.y* < ENTER>

will perform a quick syntax check on HELLO.Y. No HELLO.C is produced.

3.7 Build Parser with Parse Tree Generation Ability (-t, -T)

The -t option is useful for debugging grammar files and studying the language structures of the source language. With this option, the parser generated by PCYACC will produce textual form parse trees for source programs, using the default file YY.AST.

Example: *PCYACC -t hello.y* <ENTER>

will make HELLO.C have the ability to produce parse trees for source programs. The tree is saved to the file YY.AST in this example when HELLO.C is compiled an executed.

The -T options has the same function as the -t function. It uses a different naming convention for the file to dump the parse tree information.

Example: *PCYACC -T hello.y* < ENTER>

will make the parser use the file HELLO.AST to store the parse tree information when HELLO.EXE is executed.

Example: *PCYACC -T greet.ast hello.y* < ENTER>

will make the parser use the file GREET.AST to store the parse tree information.

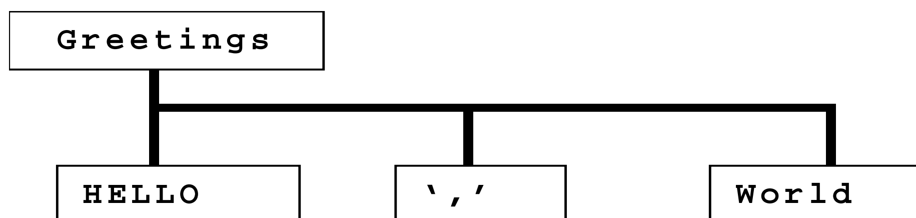
When the program generated from HELLO.Y in this example is applied to the following program:

hello, world

The file GREET.AST (YY.AST, or HELLO.AST) is created with the following information:

greetings HELLO , WORLD

which represents the parse tree for the input program:



Abstract Syntax Tree for Hello.Y

3.8 Use Verbose Mode (-v and -V)

The -v option produces verbose output in YY.LRT. The output resembles an LALR parsing table with additional information.

Example: *PCYACC -v hello.y* <ENTER>

generates a text file YY.LRT in your current folder.

The -V option has the same function as the -v option. -V lets you redirect the verbose output to either the file HELLO.LRT or a file of your choice.

Example: *PCYACC -V hello.y* <ENTER>

creates the verbose output in HELLO.LRT.

Example: *PCYACC -Vgreet.lrt hello.y* <ENTER>

produces the verbose output in GREET.LRT.
Partial contents of the file GREET.LRT (HELLO.LRT, or YY.LRT) is shown next:

```
-*-=-*-=-*-=-*-      LALR PARSING TABLE      -*-=-*-=-*-=-*-

+--------------------     STATE 0     ---------------------+

+ CONFLICTS:

+ RULES:
        $accept : ^greetings $end

+ ACTIONS AND GOTOS:
          HELLO : shift & new state 2
                : error

      greetings : goto state 1

...

+--------------------     STATE 4     ---------------------+

+ CONFLICTS:

+ RULES:
      greetings :  HELLO , WORLD^    (rule 1)

+ ACTIONS AND GOTOS:
                : reduce by rule 1

====================      SUMMARY      ====================

grammar description file = hello.y
number of terminals used =      5; limit =    500
number of nonterminals   =      1; limit =    500
number of grammar rules  =      2; limit =   1000
number of states         =      5; limit =   1000
number of s/r errors     =      0
number of r/r errors     =      0

...

-*-=-*-=-*-=-*-         END OF TABLE         -*-=-*-=-*-=-*-
```

IV. BASICS OF PROGRAMMING LANGUAGES

This Chapter introduces basic concepts related to computer programming languages. Some important terminology's to be used in later chapters are also explained.

1. What Is A Computer Programming Language?

The types of work that can be done and the speed at which these types of work are done by computers are quite overwhelming. Nevertheless, computers do not have intelligence -- they can do nothing without being given step by step instructions, called programs. These programs that make computers appear to be miracles are presented to the computer in the form of computer languages.

Computers can only understand instructions written in a special kind of language, called machine languages. Machine language sentences are phrased in terms of binary digits, (0's and 1's). Machine languages are so primitive that we as computer users find them difficult to understand. Consequently, writing instructions that tell computers what to do, or programming, had been considered a tedious task, before people started to explore the possibility of teaching computers secondary languages. These secondary languages, called high level languages, use sentences that make more sense to human minds.

2. What Are Compilers

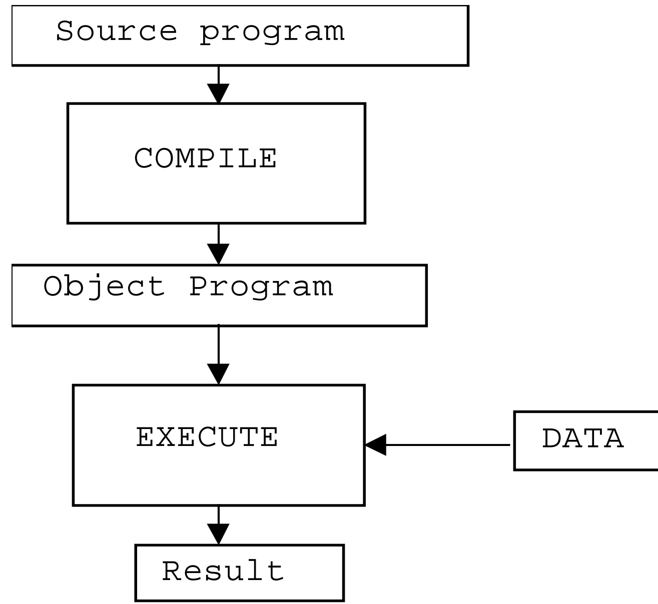
Computers can't really be taught to understand secondary languages. As a result we have developed translators for computers, which are capable of converting computer programs written in high level languages into instructions in a computer's native language -- machine language, or low level language. There is another kind of low level language, called assembly language. Assembly language is directly derived from machine language by assigning a mnemonic name to each machine instruction.

Translators don't have to be restricted to do the translations from high level languages to low level languages. There are different kinds of translators for computers, and they have been given special names. Translators that translate assembly language programs into machine language programs are called assemblers. Conversely, translators that translate machine language programs into assembly language programs are called disassemblers. Translators that translate high level language programs into low level language programs are called compilers. And translators that translate programs in one high level language into programs in another high language are called preprocessors (such as C++).

For a given translator, programs to be translated are called source programs, and the results of the translation are called object programs. The language in which source programs are written is referred to as the source language, and the language in which object programs are composed is referred to as the object language.

With compilers, human users no longer need to deal with machine languages in order to have computers do useful things, and programming has become a much more pleasant task. Carrying out the effects prescribed by programs written in high level languages is divided into two phases: a compilation phase and an execution phase. During the compilation phase, high level language programs are first translated into machine languages.

During the execution phase, the machine language programs are executed. Schematically, this process can be illustrated by the following diagram:


Source program
COMPILE
Object Program
EXECUTE
DATA
Result

V. GETTING STARTED -- A SMALL EXAMPLE

The purpose of this Chapter is to give you an idea of how to use PCYACC in a language development project. To achieve that, we assume you are familiar with the C programming language. We also assume you have an , ABRAXAS PCYACC, , and a C compiler.

This Chapter gives you an overview of the program development process using PCYACC. The example used in this Chapter is a simple calculator capable of doing ordinary arithmetic operations. The example will show you the PCYACC program listing for SACALC, the name given to the simple arithmetic calculator program, and illustrate how to build the executable SACALC using PCYACC and a C compiler. In a later Chapter we will detail the development procedure with a slightly more advanced example.

1. Grammar Description File for SACALC

The following is the listing of the PCYACC GDF for the calculator example, SACALC.Y. For reference, line numbers are added to the statements (line numbers should NOT be included in your GDL's).

```
01: %{
02:
03: #define YYSTYPE double /* stack data type */
04:
05: %}
06:
07: %token NUMBER
08: %left '+' '-'   /* left associative */
09: %left '*' '/'   /* left associative */
10: %left UNARYMINUS
11:
12: %%
13:
14: list:           /* nothing */
15:    | list '\n'
16:    | list expr '\n'
17:        { printf("\t%.8g\n", $2); }
18:    | list error '\n'
19:        { yyerrok; }
20:    ;
```

```
21:
22: expr: NUMBER
23:         { $$ = $1; }
24:   | '-' expr %prec UNARYMINUS
25:       { $$ = -$2; }
26:   | expr '+' expr
27:       { $$ = $1 + $3; }
28:   | expr '-' expr
29:       { $$ = $1 - $3; }
30:   | expr '*' expr
31:       { $$ = $1 * $3; }
32:   | expr '/' expr
33:       { $$ = $1 / $3; }
34:   | '(' expr ')'
35:       { $$ = $2; }
36:   ;
37:
38: %%
39:
40: #include <stdio.h>
41: #include <ctype.h>
42: char *progname;        /* for error messages */
43: int  lineno = 1;
44:
45: main(argc, argv)
46: char *argv[];
47: {
48:   progname = argv[0];
49:   yyparse();
50: }
51:
52: yylex()
53: {
54:   int c;
55:
56:   while ((c=getchar()) == ' ' || c == '\t' );
57:
58:   if (c == EOF)
59:     return 0;
60:   if (c == '.' || isdigit(c)) { /* number */
61:     ungetc(c, stdin);
62:     scanf("%lf", &yylval);
63:     return NUMBER;
64:   }
65:   if (c == '\n')
66:     lineno++;
67:   return c;
68: }
```

```
69:
70: yyerror(s) /* called on syntax error */
71: char *s;
72: {
73:   warning (s, (char *) 0);
74: }
75:
76: warning(s, t) /* print warning message */
77: char *s, *t;
78: {
79:   fprintf(stderr, "%s: %s", progname, s);
80:   if (t) fprintf(stderr, " %s", t);
81:   fprintf(stderr, " near line %d\n", lineno);
82: }
83:
```

This example, even though extremely small, exhibits the typical structure and components of a PCYACC grammar description program. Lines 1 through 11 form the so called declaration section, where token symbols, operator precedences, etc., are declared. Lines 12 through 37 make up the grammar rule section, where the grammar rules specifying the language being developed are placed. Lines 38 through 83 form the program section, where supporting C routines for the compiler are written. As illustrated by this example, a grammar description program is made up of three sections: a declaration section, a grammar rule section and a program section. Two of the three sections, the declaration section and the program section, can be empty. (If there is no declaration section, the delimiter "%%" will still be needed to tell PCYACC to start processing the grammar rule section.)

The symbol pair on line 1 and line 5 is a delimiter that is used in the declaration section to include C statements, such as preprocessor directives, global data structure definitions or variable declarations. (In this example, there is a preprocessor directive.) PCYACC will not look inside these delimiters. Everything inside the delimiters will be passed to the C program that is generated without any change.

Line 7 declares NUMBER to be a token, a grammar symbol that cannot be used as the left hand side of grammar rules. (When yylex is called, it is supposed to return the type of token that it found. NUMBER will be declared in a #define clause in the generated program. It can also be put into an include file if yylex is not part of the generated program. Token types like NUMBER and also 0, which signifies end of file, are what the generated parser is expecting to receive when yylex is called.)

Lines 8 through 10 define the associativity of the arithmetic operators involved. All the operators are declared to be left associative (i.e., if the statement is a+b+c, then a+b will be calculated first). These statements also convey the following information: addition (+) and subtraction (-) have the same precedence; multiplication (*) and division (/) have the same precedence and it is higher than addition or subtraction; the unary operator, namely the negation sign, has the highest precedence.

Line 12 is a delimiter separating the declaration section from the grammar rule section. Lines 14 through 20 are short hand for the following four grammar rules:

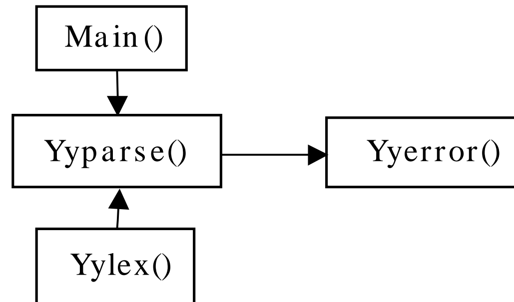
```
(1) list : ;
(2) list : list '\n' ;
(3) list : list expr '\n' ;
(4) list : list error '\n' ;
```

These rules say that a list can either be empty (1), be a list followed by a new line character (2), be a list followed by an expression and a new line character (3), or be a list followed by something which, in this case, is an error (4). In short hand notation, the colon (:) is used to delimit the left hand side of grammar rules. The vertical bar (|) is used to delimit the grammar rules that have a common left hand side nonterminal grammar symbol, or the alternatives of this common nonterminal symbol. The semicolon (;) is used to terminate grammar rules. Under the grammar rules for list, on lines 17 and 19, there are commands enclosed in braces. These are called <u>actions</u>. Similarly, lines 22 through 36 define grammar rules for expressions.

The second %% delimiter separates the grammar rule section from the program section. Everything in the program section is also copied to the output of PCYACC. This section defines three C functions: main(), yylex() and yyerror(). Note that these three functions are always required to provide support for the parser generated by PCYACC. (They can be in separate files and simply linked during program generation.) The following discussion will help you understand how to combine what PCYACC produces with the supporting functions written by the programmer to make an integrated C program.

What PCYACC produces is a C function, yyparse(), which is, technically, an LALR parser for the language defined by the grammar rules of in the GDF. The function of the user provided main function, main(), is to activate the parser, perform necessary initialization before activation, and to clean up after activation. The lexical analyzer, yylex(), is the front end of the parser. The lexical analyzer is responsible for decomposing raw text strings into

meaningful lexical units, called tokens, and passing this information to the parser. The error processing routine, yyerror(), is called by the parser when a syntax error is uncovered during parsing. The relationship among these four routines is depicted by the following diagram:



The three sections, the declaration section, the grammar rule section and the program section, will be discussed in detail in later chapters.

2. Building the Executable SACALC

To invoke PCYACC on the grammar description file SACALC.Y, issue the following command:

PCYACC SACALC.Y< ENTER>

The result of this operation is a C program. A file with the name SACALC.C will be created in the current folder, which is the C program for the calculator. To obtain the executable version of the SACALC, invoke the C compiler as follows (assuming the MICROSOFT C compiler here):

CL SACALC.C <ENTER>

This will generate the object file. Now use the LINK command to produce the actual MSDOS tool:

LINK SACALC.OBJ <ENTER>

(Note: there is a "make" file, SACALC.MAK, that can be invoked from MSDOS to automatically generate an tool. If you use a different C compiler, you will need to follow your compiler's procedures for building an application. This application will need some means, possibly a dialog, to get input and display results.)

3. Sample Session of SACALC at Work

After the executable version of the SACALC is built successfully (in our example, we would have a SACALC in the current folder), we can use it to perform simple arithmetic. The following is a sample session of SACALC at work:

SACALC <ENTER>

1 + 2 <ENTER>
3

*2.5 + 4 * 1.5* <ENTER>
8.5

*2 / 4 - 3 * 0.5 / 3 + 5* <ENTER>
5

VI. BASIC CONCEPTS REVISITED

This Chapter reviews and summarizes important concepts related to PCYACC that we have encountered so far, plus some new ones. We will list basic technical terms to prepare for more formal discussion of these subjects in a later Chapter.

1. General Concepts

A programming language is an artificial language used to write instructions for computers to perform certain tasks. Programs, which can have any number of instructions, are the basic form in which these instructions are presented to computers for execution. There are two classes of programming languages: machine languages and high level languages. Programs written in machine languages are directly understood by computers and can be executed immediately. Programs written in high level languages have two different modes of execution: interpreted execution and compiled execution. In an interpreted execution mode, instructions in a program are interpreted one at a time. Each step includes translation of one instruction followed by machine actions dictated by the instruction. This interpretive execution process is controlled and performed by another program, called an interpreter. A compiled execution mode is divided into two phases: a compilation phase and an execution phase. During the compilation phase, a compiler first translates the source program into the object program. In the execution phase, the object program is then executed.

A programming language can be defined in a number of ways. For example, you could list all legal programs if there are only a finite number of them. A better way is to use the so called context-free grammars (CFG's), which have been one the most versatile techniques for defining programming languages. There are several advantages to using context-free grammars as opposed to some other means. First, the formal theoretical aspects of CFG's have been studied intensively and are well understood. Second, CFG's are founded on a formal basis and are very precise, as far as defining syntax is concerned. Third, CFG's are powerful enough to encompass most programming languages in practice.

Each programming language has a particular set of predefined vocabularies with which programs can be written. A context-free grammar defining a programming language also uses the same set of vocabularies, which are referred to as terminal symbols, or tokens, of the CFG. In addition to tokens,

the CFG needs a separate set of vocabularies so that it can express intermediate constructs needed for defining the language. These additional vocabularies are called nonterminal symbols of the CFG. A grammar symbol can be used to mean either a terminal symbol or a nonterminal symbol. The key components of a CFG is a set of production rules, or grammar rules, constructed in the following form:

```
R: X --> X1 X2 ... Xn
```

With this form, a grammar rule R is defined. X is called the lefthand-side (LHS) of R and must be a nonterminal symbol. X1 through Xn form the righthand-side (RHS) of R. The RHS of a grammar rule can also be empty. When not empty, each component of the RHS can either be a terminal symbol or be a nonterminal symbol. The last element in a context-free grammar is a start symbol, which is a special nonterminal symbol used to indicate the most significant construct of the language being defined.

2. BNF -- A Language for Writing CFG's

Backus-Naur-Forms (BNF's) are often used to write context-free grammars. An extended version, referred to as Extended Backus-Naur-Forms (EBNF's), will be used to illustrate how to write a context-free grammar.

The basic conventions of using EBNF's to write a context-free grammar are the following:

1). terminal symbols are always quoted using the single quotes(');

Example: '+', '-', '*'

2). Non-terminal symbols are always enclosed using the angle brackets (< and >);

Example: <list>, <expr>

3). a space is used to separate two consecutive grammar symbols;

Example: <list> '\n'

4). a special symbol, ::=, is used to separate the LHS and the RHS of a grammar rule;

Example: <list> ::= <list> '\n'

5). two consecutive grammar symbols in a row have a meaning of syntactic concatenation;

Example: <list> ::= <list> <expr> '\n'

6). two grammar symbols separated by a vertical bar, |, have a meaning of syntactic alternative;

Example: <list> ::= <list> '\n' | <list> <expr> '\n'

7). parentheses can be used to group grammar symbols;

Example: <list> ::= <list> ('\n' | <expr> '\n')

8). square brackets ([and]) are used to enclose optional syntactic structures;

Example: <list> ::= [<list> ('\n' | <expr> '\n')]

9). braces ({ and }) are used to enclose repetitive syntactic structures (including zero repetition);

Example: <list> ::= {'\n' | <expr> '\n'}

To put the conventions together, the following example defines valid sentences (although they do not make much sense) using EBNF.

```
<sentences>  ::=  { <sentence> }
<sentence>   ::=  <subject> (<t_verb> <object>
             |    <i_verb>) '.'
<subject>    ::=  [ <article> ] <noun>
<article>    ::=  'a'
             |    'the'
<noun>       ::=  'dog'
             |    'cat'
<t_verb>     ::=  'chases'
<i_vers>     ::=  'runs'
<object>     ::=  [ <article> ] <noun>
```

3. PCYACC Terminology's -- A Short Review

PCYACC is a compiler-writer, or a parser generator, a program that assists in writing compilers. From the viewpoint that it translates a context-free grammar into a C function that recognizes programs written in the language defined by the grammar, PCYACC can be thought of as a compiler. The source language of PCYACC is the grammar description language (GDL). The object language of PCYACC is the C programming language. Source programs written in GDL are called grammar description programs (GDP's). Files used to hold GDP's are called grammar description files (GDF's). The function of PCYACC is therefore to translate a GDF into a C function. By convention, the name of the function is yyparse().

A GDP is made up of three sections, a declaration section (DS), a grammar rule section (GRS) and a program section (PS), in that order. Either DS or PS, or both can be empty. Two consecutive sections are separated using the special delimiter, %%.

In the declaration section, anything enclosed with %{ and %} are copied to the output without any alteration. Tokens are declared with the keyword **%token**. Associativities of operators are declared with the keywords **%left**, **%right** and **%nonassoc**. Precedence information is conveyed by the order in which the associativity declarations are made (later declarations have higher precedence). Data types of grammar symbols are declared with the keyword **%type**. The start symbol of the grammar is declared with the keyword **%start**.

The grammar rule section contains a number of grammar rules. Each grammar rule has a left hand side, which is a nonterminal symbol, and a right hand side, which is a sequence of zero or more grammar symbols. The LHS and the RHS of a grammar rule are separated by a colon (:). A grammar rule is terminated using a semicolon (;). An action is attached to a grammar rule using braces ({ and }). Actions should come before the grammar rule terminator (;). A collection of grammar rules with a common LHS are the syntactic alternatives of the nonterminal symbol and can be grouped together. In this case, the common nonterminal symbol appears on the left hand side of a colon, followed by a sequence of right hand sides separated by vertical bars (|), and terminated by a semicolon.

The following grammar rules taken from the calculator example illustrate the descriptions of the previous paragraph.

```
expr: NUMBER
  { $$ = $1; }
| '-' expr %prec UNARYMINUS
```

```
  { $$ = -$2; }
| expr '+' expr
  { $$ = $1 + $3; }
| expr '-' expr
  { $$ = $1 - $3; }
| expr '*' expr
  { $$ = $1 * $3; }
| expr '/' expr
  { $$ = $1 / $3; }
| '(' expr ')'
  { $$ = $2; }
;
```

Everything in the program section is copied to the output without any change. There are three functions that must be provided by the programmer: main(), yylex() and yyerror(). Additional functions may be required by the problem specification. However, C functions do not have to be included by GDF's, since the program section of a GDP can be empty. Instead, they can be contained in other C source files, then compiled and linked using facilities available in your C programming environment.

VII. INTOPOST -- A SECOND EXAMPLE

This Chapter is a continuation of Chapter V. This Chapter will help acquaint you with the procedure and style of program development using PCYACC, and provide you with some guidelines for project development.

A general guideline for project development using PCYACC is a six-step procedure, as follows:

1). Identify the problem
2). Define the source language
3). Write PCYACC grammar description program
4). Write auxiliary C code
5). Obtain yyparse() from the GDP using PCYACC
6). Build the executable program

(1.) The first step is concerned with the overall design of the system to be developed. It resolves major design decisions such as system architecture, system function decompositions, etc.

(2.) The objective of the second step is to specify the language to be translated. The tool that is most appropriate in this stage of the development is a context-free grammar. You may choose to write the grammar specification directly in the form of PCYACC grammar rules, or you can use the EBNF's.

(3.) The third step is to convert (if we used EBNF's) or incorporate (if we used PCYACC grammar rules) the language definition into a PCYACC grammar description file.

(4.) The fourth step is to augment the grammar rules with action codes. The three required support routines (main(), yylex() and yyerror()) should be coded to make it possible to debug the grammar description program. The grammar description program should then be debugged.

(5.) The fifth step translates the grammar description program into the C function, yyparse(), by invoking PCYACC. The rest of the C functions as required from the design phase can then be coded.

(6.) The sixth step relies on the existing C programming language environment to build the executable modules from C routines, which

normally involves intensive debugging efforts. It may be necessary to go back to a previous step one or more times before you arrive at a satisfactory result.

Note that the above guideline is only provided as a reference. Although in most cases it reflects the proper methodology, it may not be suited to all cases. Therefore, you may wish to develop your own development procedure as you become more familiar with PCYACC.

The example in this Chapter builds a compiler that translates infix arithmetic expressions into postfix notations. Let's call this compiler INTOPOST. This small example allows you to get a feel for the compiler writing process using PCYACC. You may notice that the two examples, SACALC and INTOPOST, are closely related. In particular, the approach in the SACALC example is an interpretive approach, since each expression is evaluated as it is entered and no object programs were produced. The approach taken in the INTOPOST example will be a more compiled approach. INTOPOST will generate object programs from source programs. Also, INTOPOST demonstrates strong programming language orientation, in that arithmetic expression translation is part of almost all conventional language compilers, and therefore is particularly suitable for our illustration purposes.

1. Problem Statement

A stack is a special list where elements can be entered and removed from only one end, the stack top. A stack machine is a computer that employs a hardware stack for executing its programs. (Postscript interpreters -- a Postscript GDL is included in the Professional version of PCYACC -- are stack based. FORTH is stack based.)

The following discussion will help you to understand how a stack machine works and why you would actually need such a program to translate arithmetic expressions from their infix notations to their postfix notations.

We are used to writing arithmetic expressions in their natural form, namely the infix notation. Infix notation is much easier to read and evaluate. However, if these expressions are to be evaluated by a machine, they are better written in a form that is most natural to the machine's architecture. Therefore, when evaluating arithmetic expressions such as this one:

    2 + 3 * 5,

on a stack machine, it is a common practice to first convert the infix notation to an intermediate form, called a postfix notation. This allows you to exploit the underlying architecture, simply because postfix notation is more natural to stack machines. The postfix notation for the previous expression is

2 3 5 * +

A stack machine typically has instructions such as PUSH, MUL and ADD. The following instruction sequence can easily get the expression evaluated:

```
PUSH 2
PUSH 3
PUSH 5
MUL
ADD
```

We wish to build a compiler, INTOPOST, to perform conversions on arithmetic expressions conversions from infix notation to postfix notation. Let's call the source language of INTOPOST infixel (infix expression language), and the object language of INTOPOST postfixel (postfix expression language). The task of INTOPOST is to translate programs written in infixel into programs written in postfixel.

2. Defining the infixel Language

The very first thing in constructing a compiler is to define its source language precisely (here we refer to the syntax of a language). For this example, we use EBNF's to define the syntax for infixel, just to show how they can be used. Later, we will use PCYACC grammar description language to directly accomplish the same task.

Since an infixel program specifies arithmetic expressions, the following descriptions intuitively match what we think of as infixel programs:

1). an infixel program is an infixel expression
2). an infixel expression is either an infixel term, an infixel term plus an infix expression, or an infixel term minus an infixel expression
3). an infixel term is either an infixel factor, an infixel factor multiplied by an infixel term, or an infixel factor divided by an infixel term
4). a factor is either a constant, a variable or a parenthesized infixel expression.

Translation of the informal description to EBNF's is fairly simple. One possible solution is provided below. Notice we did not try to write the EBNF definitions in their simplest form. Instead, we chose to do the translation in a straightforward manner. There are two reasons for taking this approach. First, the resulting EBNF's directly correspond to what an arithmetic expression should look like. Second, the resulting EBNF's have a one-one correspondence with their final forms -- PCYACC grammar rules. (We did not use either square brackets or curly brackets because PCYACC does not support these kind of notations.)

```
1). infixel_program ::= infix_expr
2). infix_expr      ::= infix_term
                     |  infix_expr '+' infix_term
                     |  infix_exir '-' infix_term
3). infix_term      ::= infix_factor
                     |  infix_term '*' infix_factor
                     |  infix_term '/' infix_factor
4). infix_factor    ::= constant

                     |  variable

                     |  '(' infix_expr ')'
```

Constants are a sequence of numerical digits, and variables are identifiers composed of alphanumerical character strings, with the first character being a letter.

3. Writing PCYACC Grammar Description For infixel

As discussed earlier, PCYACC requires the grammar description file in the form of a text file. A text editor (normally the builtin PWB editor, but anything that creates TEXT files can be used) is used to create the file INFIXEL.Y to hold the grammar description program for infixel.

The following is a list of the contents INFIXEL.Y, with comments and explanations. Line numbers are attached to statements for reference.

```
001: %{
002:
003: #include <stdio.h>
004: #include <ctype.h>
005:
006: char outfn[]="POSTFIX.EXP";
007: FILE *fopen(), *inf, *outf;
008:
009: %}
010:
011: %union {
012:   char *oprnd;
013: }
014:
015: %token CONSTANT
016: %token VARIABLE
017: %type  <oprnd> CONSTANT VARIABLE
018: %start infix_prog
019:
020: %%
021:
022: infix_prog : infix_expr ';'
023:    { fprintf(outf, " ;\n"); }
024:  | infix_prog infix_expr ';'
025:    { fprintf(outf, " ;\n"); }
026:  ;
027:
028: infix_expr : infix_term
029:  | infix_expr '+' infix_term
030:    { fprintf(outf, " +"); }
031:  | infix_expr '-' infix_term
032:    { fprintf(outf, " -"); }
033:  ;
034:
035: infix_term : infix_fact
036:  | infix_term '*' infix_fact
037:    { fprintf(outf, " *"); }
038:  | infix_term '/' infix_fact
039:    { fprintf(outf, " /"); }
040:  ;
041:
```

```
042: infix_fact : CONSTANT
043:     { fprintf(outf, " %s", $1); }
044:   | VARIABLE
045:     { fprintf(outf, " %s", $1); }
046:   | '(' infix_expr ')'
047:   ;
```

```
048:
049: %%
050:
051: int nxtch;
052:
053: main(argc, argv)
054: int argc;
055: char *argv[];
056: {
057:
058:   if (argc != 2) {
059:     fprintf(stderr,
060:         "Usage: intopost <infile> \n");
061:     exit(1);
062:   }
063:   if ((inf=fopen(argv[1], "r")) == NULL) {
064:     fprintf(stderr,
065:         "Can't open file: \"%s\"\n", argv[1]);
066:     exit(1);
067:   }
068:   if ((outf=fopen(outfn, "w")) == NULL) {
069:     fprintf(stderr,
070:         "Can't open file: \"%s\"\n", outfn);
071:     exit(1);
072:   }
073:
074:   nxtch = getc(inf);
075:   if (yyparse()) {
076:     fprintf(stderr,
077:         "Error in translation\n");
078:   }
079:
080:   fclose(inf);
081:   fclose(outf);
082: }
083:
084: yyerror(s)
085: char *s;
086: {
087:   fprintf(stderr, "%s\n", s);
088: }
089:
090: #define POOLSZ 2048  // POOLSZ >> avail
091: char chpool[POOLSZ];
092: int  avail = 0;
```

```
093:
094: yylex()
095: {
096:   int i, j, toktyp;
097:
098:   while ((nxtch==' ') || (nxtch=='\t')
099:                       || (nxtch=='\n'))
100:     nxtch = getc(inf);
101:   if (nxtch == EOF)
102:     return(0);
103:   if (isdigit(nxtch)) {
104:     toktyp = CONSTANT;
105:     yylval.oprnd = chpool + avail;
106:     chpool[avail++] = nxtch;
107:     while (isdigit(nxtch=getc(inf)))
108:       chpool[avail++] = nxtch;
109:     chpool[avail++] = '\0';
110:   } else if (isalpha(nxtch)) {
111:     toktyp = VARIABLE;
112:     yylval.oprnd = chpool + avail;
113:     chpool[avail++] = nxtch;
114:     while (isalnum(nxtch=getc(inf)))
115:       chpool[avail++] = nxtch;
116:     chpool[avail++] = '\0';
117:   } else {
118:     toktyp = nxtch;
119:     nxtch = getc(inf);
120:   }
121:
122:   return(toktyp);
123: }
124:
```

Note the similarities between the EBNF description for infixel, which came up during the language design phase, and the grammar rule portion enclosed by "%%" of this listing (ignoring the contents in curly braces). This is why PCYACC is such a good tool for writing fairly complex translation programs.

This example program is not a whole lot more complicated than the previous SACALC example. On the other hand, it is one step closer to illustrating compiler architectures. (Notice that SACALC took an interpretive approach towards program executions -- no object programs were produced.)

Notice that the program, again, exhibits a similar three-sectioned structure. The first section, the declaration section, starts at line 1 and ends with line 19. The second section, the grammar rule section, begins at line 20 and stops at line 48. The third section, the program section, starts at line 49 and runs to line 124.

The declaration section includes two system header files. The standard input/output declaration header, "stdio.h", is almost always needed by any program; in this particular case, this file is not optional since its declaration was needed for FILE type. The character type classification library header file "ctype.h" was also included, since those functions were used by the lexical analyzer. A fixed file, POSTFIX.EXP (line 6), was used to hold the results of translation, namely, object programs. Line 7 defines two file pointers; one for input files containing source infexel programs, and one for output files to hold object postfixel programs. Lines 11 through 13 define the data type for the internal value stack. The value stack can be of a union type, although in this case, a one-alternative union is used. The rest of the first section declares two terminal symbols, CONSTANT and VARIABLE, data types for the two terminal symbols (oprnd), and the start symbol (infix_prog).

The grammar rule section contains 11 grammar rules. The first two rules say an infixel program is made up of one or more infixel expressions, and that each infixel expression has a semicolon terminator ";". Note that the second rule,

```
infix_prog : infix_prog infix_expr ';'
```

uses the <u>left recursion</u> technique to describe a repetitive syntactic structure. Although <u>right recursio</u>n could do the job just as well, left recursion is the recommended style in LR parsers. (We will see why later.) The remaining grammar rules are directly mapped from the earlier EBNF descriptions and are fairly self-explanatory.

An action, which is a piece of C code enclosed by curly braces, can be associated with a grammar rule by appending the action code segment to the end of the grammar rule. (It is possible to insert action code in the middle of a grammar rule, which will be discussed later.) A grammar rule in PCYACC can take the following form:

```
LHS : RHS { C-code-segment } ;
```

This grammar rule is applicable during the parsing of a program. It can be read as: use the LHS to replace the RHS, and then execute the code segment. Note that the action code is executed after the grammar rule is applied.

Line 51 declares an integer variable, nxtch, which is used by the lexical analyzer to store the new character received from the input file.

The main routine, main(), runs from line 53 to line 82. It performs the following four simple tasks. First, it processes command line arguments. The program invocation uses the standard format; the program name is followed by a file name argument. The argument file is assumed to hold a source

program in infixel. This convention is enforced by checking the argument count, argc, which must be 2. Next, all files involved are opened and the status of the open operations are tested to detect exceptions. The preparation phase is completed when the first call to getc() is made to initialize the single character lookahead buffer used by the lexical analyzer. The environment at this point is ready for further operations.

The third task of this main routine is the most important. The LR parser generated by PCYACC, the yyparse() function, is called to work on the source program. This is the translation phase. yyparse() returns a zero (0) if the translation process is successful. A nonzero value is returned if an error situation occurrs during the translation. The last thing in this main routine is cleaning up; both the input and output file are closed and the program terminates.

The second function, yyerror(), prints an error message on the standard error device, which, under MSDOS, will be the working window.

Line 90 defines a symbolic constant. Line 91 uses the symbolic constant for declaring the size of a character array, chpool[]. chpool[] is the storage place for symbolic names appearing in the source programs being translated. The integer variable, avail, is an index into the character array, keeping track of the usage/availability of the storage space.

The third function, yylex(), is the most complicated in this example. Nevertheless, what it does is conceptually simple. yylex() is called each time yyparse() needs to look at the next token in the input stream. Each time yylex() is expected to return a single token, the next one in the input is returned to its caller. To do so, it first skips white spaces in the input stream. If this skipping process ends as an EOF shows up, yyparse() is notified of this fact. (Note that the character buffer, nxtch, was filled before yyparse() was invoked). If a meaningful symbol shows up to stop this skip loop, the symbol is saved in the storage pool, chpool. Two possibilities exist in this case. Depending on the first character of this next character string, the next symbol may represent either an integer number, if the first character is a digit, or a variable, when the first character is a letter. Finally, the tokens (their types) just captured are returned to yyparse().

4. Converting Grammar Descriptions Into C Programs

Recall your goal is to build a compiler for infixel, which you chose to call INTOPOST. You are not quite there yet, even though you have created INTOPOST.Y, a grammar description file for infixel language.

You need to convert the PCYACC grammar description program, INTOPOST.Y, into a C program, INTOPOST.C (PCYACC by default uses <filename>.C for its object programs, where <filename> is the name of corresponding source programs. See Chapter III for details on naming conventions of PCYACC). This INTOPOST.C is the compiler you would have to write using C in the first place, had you chosen not to use PCYACC to build INTOPOST.

Since we already have INTOPOST.Y, all you need to do to get INTOPOST.C is to issue the following command from MSDOS:

PCYACC INTOPOST.Y<ENTER>

Upon termination of PCYACC, INTOPOST.C will be written in the current folder.

5. Building the Executable infixel Compiler -- INTOPOST

You are still two steps away from having an executable compiler. In this stage, you need to rely on your programming environment, C compiler in particular, to achieve your final goal.

To obtain the executable infixel compiler, namely INTOPOST, issue the following command from MSDOS:

CL INTOPOST.C <ENTER>

Upon termination of the C compiler, the object module INTOPOST.obj is created in the current folder. INTOPOST.obj is a compiler for the infixel language. You now need to link the object module with the standard Microsoft C libraries to create the final program. (Note: there is a "make" file that automates all of the steps for you.)

To test the compiler, create a file, say TEST.INF, containing the following infixel program with three expressions:

```
1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9;
1 + 2 - 3 + 4 - 5 + 6 - 7 + 8 - 9;
1 + 2 - 3 * 4 / 5 + 6 - 7 * 8 / 9;
```

and issue the following:

INTOPOST TEST.INF <ENTER>

The result of the compilation is shown below. (A file called POSTFIX.TXT with the same contents will be created in your current folder.)

```
1 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 +;
1 2 + 3 - 4 + 5 - 6 + 7 - 8 + 9 -;
1 2 + 3 4 * 5 / - 6 + 7 8 * 9 / -;
```

VIII. PRINCIPLES BEHIND PCYACC

Having finished the exercise of building a complete (though small) compiler, INTOPOST, with little effort, you should begin to see that PCYACC is a wonderful tool for compiler writers. At the same time, you may start to wonder where PCYACC gains its power: the ability to turn grammar descriptions into C programs that recognize corresponding languages. This Chapter explores the fundamental principles of PCYACC, and its underlying theories. This is a short, informal discussion of relevant theory. There are outstanding text books dedicated to investigating this intriguing subject. A complete bibliography is included in the appendix section of this manual.

1. Introduction to Formal Language Theories

Computer programs are written in computer languages. These artificial languages are rather restricted compared to natural languages. There are precise rules for checking allowable individual sentences. Furthermore, the rules have to be simple so that the computer can perform checking according to a mechanical procedure. Rules for precise definition of computer languages make up what is called formal grammars.

All programs are written in a computer language of some kind. But this raises an interesting question. Given any particular programming language, be it C, PASCAL, FORTRAN or BASIC, how many different programs can be written in a language? It is possible to write an infinite number of them. How is it then, that language processors can cope with an infinite number of possibilities, since for each program, its language processor needs to check for both syntax and semantics?

It is exactly these kinds of questions that led computer scientists to the study of formal languages. Formal languages provide a solution to the problem of describing infinite languages in a concise manner, using only a finite number of symbols. The theory of formal languages has become established in computer science.

2. Context-free Grammars

According to Chomsky, who is well known to computer science community for his contributions to the study of formal languages, there are four kinds of grammars, each a super set of the other. These grammars, regular grammars, context-free grammars, context-sensitive grammars and phrase grammars, are studied in great detail by formal language theoreticians. These four grammars are listed according to increasing description power and complexity.

Regular grammars are the most simple kind of grammars, and can be used for a wide variety of applications. Although they are a bit too simple to describe practical programming languages, they are good for defining lexical rules for compilers. On the other end of the spectrum are the so called phrase grammars, which are most complicated. Phrase grammars are, in general, difficult for computers to deal with. Phrase grammars are so powerful that they can describe any task that can be done by a computer.

Context-free and context-sensitive grammars are most often used to effectively specify programming languages in a mechanical manner. In theory, for most applications,context-sensitive grammars are appropriate since most programming languages of practical use are context-sensitive, but not context-free. However, when considering computational complexities, context-free grammars are so much easier to handle. Context-free grammars can be most efficiently implemented as mechanical procedures. Therefore, they are used almost exclusively by computer science professionals.

Is the choice of context-free over context-sensitive a big compromise, since most of the practical programming languages are context-sensitive? The answer is no. It is possible to single out the context-sensitive components of programming languages and deal with them separately from pure syntax processing. The syntax part of any language is always context-free. Consequently, using context-free grammars does not pose serious restrictions to defining languages. A common practice is to shift the duty of processing context-sensitive components to a so called semantic-analysis phase. This is a standard process present in most language processors.

A grammar has the following four components:

1). a collection of terminal symbols,
2). a collection of nonterminal symbols,
3). a collection of productions, and
4). a start symbol.

Terminal symbols, also known as tokens, are the basic building elements of programs. They are constant symbols because each token represents itself, but no other symbol. In a programming language, terminal symbols are used to write programs.

Example: Using C language as an example, Key words such as **if**, **then**, **else**, **while**, **break**, **continue**, **return** etc., constants such as **10**, **2.5**, **'y'**, **"text-string"** etc., and identifiers such as **count**, **linebuffer** etc. are all terminal symbols.

Nonterminal symbols, on the other hand, are grammar variables that can take on different values of nonterminal sequences. Their presence in a grammar usually corresponds to important linguistic constructs of the language defined by the grammar. There are other reasons for introducing nonterminals into a grammar, such as to normalize the grammar. This allows you to avoid difficulties when constructing language accepters.

Example: In a typical programming language such as C, **data-declaration**, **function-declaration**, **statement** and **expression** are normally recognized as nonterminals.

In this manual, we will use the term grammar symbols to refer to either terminals or nonterminals.

Productions, or rewriting rules or grammar rules, play a very important role in context-free grammars for defining programming languages. For example, they are the mechanisms that allow claims such as: "this piece of text segment is indeed a syntactically correct program", and "this **if** statement is correctly written." A production of a grammar is written as:

```
U --> V
```

where both U and V are strings of grammar symbols. Various types of grammars are made by imposing restrictions on U and V. In particular, a grammar rule in a context-free grammar has the following form:

$$X \dashrightarrow X_1\ X_2\ \ldots\ X_n$$

where X has to be a nonterminal, and X_j can be either a terminal or a nonterminal. The meaning of such a production rule in a context-free grammar is that wherever X occurs, it can be rewritten by the sequence of the grammar symbols

$$X_1\ X_2\ \ldots\ X_n,$$

in that order. Also, whenever the sequence

$X_1 \; X_2 \; \ldots \; X_n$

occurs, it can be reduced to X. Note that the replacement of a sequence of grammar symbols by a nonterminal symbol, or vice versa, using grammar rules can be done without consulting the surrounding text of grammar symbols. This is why the phrase context-free is used.

In the production rule shown above, X is called the lefthand-side (LHS) of the grammar rule, and the sequence $X_1 \; X_2 \ldots X_n$ are collectively called the righthand-side (RHS) of the grammar rule. The process of replacing the LHS of a production by its RHS is called a (one step) derivation. The inverse of a derivation is called a reduction, which is the process of replacing the RHS of a grammar rule by its LHS.

Example: The following fragment for defining statements illustrates rewriting rules in defining programming languages (taken from the C programming language reference manual):

```
statement --> compound-statement
statement --> expression ';'
```

This example states that a C statement can either be a compound statement or a semicolon terminated expression.

The start symbol of a grammar is a distinguished nonterminal symbol that normally signifies the highest level syntax concept of the language being defined. This would be **program** in the case of programming languages, or **sentence** in the case of natural languages.

In a previous example, a PCYACC grammar description file was created for a simple arithmetic calculator. The grammar part of this calculator example will now be cast into the form of a context-free grammar.

Let's call the grammar SAG (simple arithmetic grammar). The SAG can be written as

```
SAG = (SAT, SAN, SAP, SAS),
```

where SAT represents the collection of terminals in SAG, SAN represents the collection of nonterminals in SAG, SAP represents the collection of production rules in SAG and SAS is the start symbol for in SAG. Each of these components is given by the following:

```
1). SAT: NUMBER, '+', '-', '*', '/', '(', ')', '\n'
2). SAN: list, expr, error
3). SAP:
    list -->
```

```
    list --> list '\n'
    list --> list expr '\n'
    list --> list error '\n'
    expr --> NUMBER
    expr --> '-' expr
    expr --> expr '+' expr
    expr --> expr '-' expr
    expr --> expr '*' expr
    expr --> expr '/' expr
    expr --> '(' expr ')'
3). SAS: list
```

The notation, '\n', is similar to C and represents the newline character. Depending on C/C++ compiler the actual internal value for newline may vary. Microsoft use's a <CR><LF> combination, Unix <CR>, and Macintosh <LF>. Verify that the lexical analyzer [yylex()] is compiled with the same compiler as the parser [yacc()], otherwise there may be a mismatch in actual token values. The use of '\n' is a much better practice for portability. Lastly, some compilers treat '\n' internally as a newline [0xa], and others carriage-return [0xd].

3. Context-free Languages

The phrase "languages defined by grammars" has been used on more than one occasion. What does it mean? Ideally it means that context-free languages are those defined by context-free grammars. Unfortunately, that definition is rather imprecise and provides little insights to the language recognition process, which is most important in developing a compiler.

The goal of this section is to refine the notion embedded in the phrase "language defined by grammars". Although this section will only examine context-free grammars and context-free languages, the discussion is applicable to other kinds of grammars and languages.

Let X and Y be strings of grammar symbols of a given grammar G. We write X ==> Y, read as X directly derives Y in G, if Y is obtained from X by an application of a grammar rule of G to X. That is, the LHS of the grammar rule also appears in X and it is replaced by the RHS of the grammar rule. We write X ==>* Y, read as X derives Y, if Y is obtained from X by zero or more such grammar rule applications. If Y is all terminal symbols and X is the start symbol, and X ==>*Y, then we say Y is a sentence of the grammar G. The collection of sentences of a context-free grammar G is the context-free language defined by G.

The idea of languages of grammars outlined in the previous paragraph is presented from the viewpoint of language generations. An alternative method of describing the same idea can be taken from the viewpoint of language recognition's. Recall the process of replacing the RHS of a grammar rule with its LHS is called a reduction. By starting with a string of terminal symbols, applying reductions repeatedly, and arriving at the start symbol, the terminal string is a sentence of the grammar. For building language translators, this view is more appropriate, since you always start with a program and work forward.

In the previous example grammar SAG, note that

```
1). expr '+' expr ==> expr '+' NUMBER
2). expr '+' expr ==>* NUMBER '+' NUMBER
3). 5 '+' 5 '*' 2 is a sentence of SAG, since
    list ==>* 5 '+' 5 '*' 2
```

4. Parse Trees

The process of derivation can be visually represented using tree-shaped structures, called parse trees or syntax trees. Each direct derivation step

adds a number of new nodes and a number of new edges to the parse tree under construction.

The following scenario illustrates this process. If the current parse tree has a leaf node labeled by a nonterminal symbol X, and the next step of the derivation is accomplished by applying the production rule

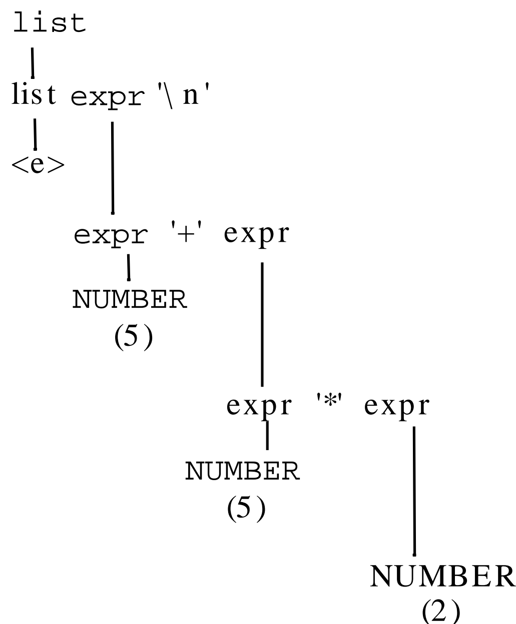
$$X \text{ --> } X_1\ X_2\ \ldots\ X_n,$$

then n new leaf nodes labeled X_1, X_2, ..., X_n respectively are created and the parse tree is augmented by attaching the newly created leaf nodes to their parent node X, which becomes an internal node. A derivation process stops when all leaf nodes of the parse tree are labeled by terminal symbols, and we have successfully derived a sentence of the language, or when there are no more production rules applicable to nonterminal leaf nodes, and the derivation fails. In the case that a derivation process does not terminate, the derivation fails. Note that in a successful derivation, the root node of the parse tree is labeled by the start symbol of the grammar, internal nodes are labeled by nonterminal symbols, leaf nodes are labeled by terminal symbols and the production rules used to obtain the derivation are captured by the parent-child relationship in the parse tree.

For example, a parse tree used to derive the sentence:

5 '+' 5 '*' 2 <ENTER>

in SAG is shown next:



In the parse tree illustrated above, the notation <e> is used to represent the empty string. Note that a parse tree representation of a derivation process is an abstraction of this derivation process. Specifically, the order of grammar rule applications is no longer present.

5. Canonical Derivation and Canonical Reduction

If there is more than one nonterminal leaf node for the current parse tree in a derivation process and more than one production rule is applicable at the same time, then you have a choice on which nonterminal node to extend to the parse tree for the next derivation step. A common strategy is to always extend the rightmost nonterminal node. The derivation process in which each step extends the rightmost non-terminals is called a rightmost derivation, or canonical derivation. Syntax trees obtained by rightmost derivations are called rightmost derivation trees. The inverse process is called a leftmost reduction, or canonical reduction.

Similarly derivation processes in which each step extends the leftmost nonterminal symbols are called leftmost derivation, leftmost derivation trees and rightmost reduction respectively.

Note that the syntax tree example is a leftmost derivation tree for the source expression.

6. Top-down and Bottom-up Parsing

Parsing a program refers to the activity of analyzing the program's syntax. Parsing is the process by which the program texts are reduced to the start symbol of the grammar that defines the language in which the program is written, or the process by which a syntax tree is constructed for the program. Equivalently, it is the process of constructing a derivation for the program from the start symbol of the grammar that defines the language in which the program is written.

Program parsing techniques fall into two broad categories, top-down parsing and bottom-up parsing. In both methods, program texts are scanned from left to right. These two methods differ in the way syntax trees are built. In top-down parsing, the construction of the syntax tree for the program starts from the root and proceeds top-down. Top-down parsing resembles a leftmost derivation process. At each step of the derivation, the left most nonterminal is chosen to expand first. If the next nonterminal to be expanded appears at the LHS of multiple productions, those productions are tried one at a time until a successful derivation is obtained. If none of the alternative production rules succeeds, the parser backtracks to the parent node of the leaf node to be expanded and tries another alternative from that point. Note that the parser can backtrack to higher ancestor nodes. As a result, the top-down parsing method may be expensive to implement. An improved top-down parsing strategy, called recursive decent parsing, can be used to avoid this problem. The limitation of recursive decent parsing is that it only recognizes a rather restricted class of context-free languages. In general, there is a serious drawback to the top-down method; it is very difficult to provide good error recovery and diagnostics.

In contrast, bottom-up parsing resembles a canonical reduction process. It builds a syntax tree for the program being parsed in a bottom-up fashion. It starts from the leaves and adds internal nodes as reductions are made. The process stops either when a well formed tree with the start symbol as the root node is obtained, (in this case the parsing succeeds and the program is said to be accepted,) or when the reduction is stuck, yet the desired syntax tree is not obtained, (in this case the parsing fails and the program is said to be in error.)

7. Ambiguities of Context-free Grammars

The task of parsing is to recognize sentences of a language. As discussed previously, this recognition can be done in two ways; either by instantiating grammar variables step by step (derivation), or by constructing grammar variables one at a time (reduction). Often it is possible to derive (or successfully reduce) sentences of a language in two or more distinct ways. This nonuniqueness phenomenon in derivability (or reducibility) is what is termed ambiguity in grammars.

A grammar is ambiguous if the language defined by the grammar contains a sentence that has two or more distinct canonical derivations (reductions). The language defined by an ambiguous grammar is also said to be ambiguous. While it is sometimes possible to rewrite a grammar to remove ambiguities, unfortunately, there are inherently ambiguous languages. Further, there are no known good algorithms available for testing for inherent language ambiguousness.

Because there are other semantic actions associated with the derivation (or reduction) processes, grammar ambiguities could cause two different derivations (or reductions) and two different sets of semantic actions to be performed. This can produce totally unexpected results.

For this reason, language developers prefer to work with nonambiguous grammars. Consider the following example grammar:

```
1). nonterminals: S, BS
2). terminal: ss, i, c, t, e
3). start symbol: S
4). productions:
    S  --> ss
    S  --> BS
    BS --> i c t S
    BS --> i c t S e S
```

The sentence "i c t i c t ss e ss" can be derived with two different rightmost derivations:

```
1). S ==> BS
      ==> i c t S
      ==> i c t BS
      ==> i c t i c t S e S
      ==> i c t i c t S e ss
      ==> i c t i c t ss e ss

2). S ==> BS
      ==> i c t S e S
      ==> i c t S e ss
```

```
==> i c t BS e ss
==> i c t i c t S e ss
==> i c t i c t ss e ss
```

The example grammar is therefore ambiguous. Note that this example also illustrates the difficulty in interpreting nested IF-THEN-ELSE statements, which are commonly used in programming languages.

8. LR Parsers

Bottom-up parsers are also called shift-reduce parsers because they use shift actions and reduce actions to recognize languages. When a stack, a data structure best suited for this purpose, is used to implement this kind of parser, a shift operation would correspond to taking an input symbol and pushing it onto the stack. At the same time, a reduce action replaces the top elements of the stack with a single grammar variable. An example will help to illustrate this parsing technique:

```
S --> a B c D e          (1)
B --> B b                (2)
B --> b                  (3)
D --> d                  (4)
```

With terminal string "a b b c d e", a stack-based bottom-up recognition process will proceed as follows:

ACTION	STACK	INPUT
initial		a b b c d e
shift	a	b b c d e
shift	a b	b c d e
reduce (3)	a B	b c d e
shift	a B b	c d e
reduce (2)	a B	c d e
shift	a B c	d e
shift	a B c d	e
reduce (4)	a B c D	e
shift	a B c D e	
reduce (1)	S	

Since the terminal string is reduced to S successfully, it is a legal sentence of the grammar.

You should be careful when performing reductions, otherwise you may fail to accept a terminal string that in fact is a legal sentence. For example, suppose you chose to use rule (3) instead of rule (2) in the second reduction step, you

would end up with the following configuration, where no more progress can be made:

ACTION	STACK	INPUT
...	a B b	c d e
reduce (3)	a B B	c d e

The concept of handles was introduced to deal with this problem associated with shift-reduce parsers. A handle of a sentential form in bottom-up parsing is a production rule that, when used to reduce the sentential form, will not block its successful reduction. If handles can always be identified and are used to perform reductions, shift-reduce parsers will be able to recognize all sentences of a grammar. LR parsers are a class of bottom-up parsers that possess this property.

An LR parser scans input from *L*eft to right, and recognizes sentences by *R*ightmost derivation in reverse. An LR parser consists of the following three components:

1). a stack holding parser states
2). a parsing table
3). a driver routine

At each parsing step, the driver uses the state on top of the stack and the first input symbol from the input stream, to determine one of the four possible actions to perform from the parsing table:

1). shift and change its state
2). reduce and change its state
3). accept and report success
4). error and invoke recovery

There are three common types of parsing tables. They are listed below in increasing power and complexity, but decreasing efficiency.

1). Simple LR parsing tables (SLR)
2). Look-Ahead LR parsing tables (LALR)
3). Canonical LR parsing tables (CLR)

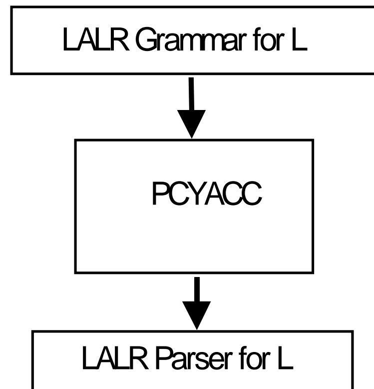
LR parsers constructed using SLR parsing tables are called simple LR parsers. Similar terminologies exist for look-ahead LR parsers and canonical LR parsers.

A context-free grammar is called an SLR (LALR, CLR) grammar, if it is possible to construct an SLR (LALR, CLR) parsing table for it. Not all

context-free grammars are SLR (LALR, CLR, or even LR). Nevertheless, most practical grammars fall into the class of LALR. Also, since LALR parser can be implemented quite efficiently, they are used for most of the programming languages in practice.

9. PCYACC -- From LALR Grammars to LALR Parsers

In previous chapters, it was established that PCYACC was a compiler generator, a program that writes compilers automatically. It is more accurate to say that PCYACC is an LALR parser generator. This concept is illustrated in the following diagram:



Note that PCYACC only provides partial support in a compiler development project. You, the programmer, are still responsible for the rest. The next few chapters will show you how to use PCYACC to generate LALR parsers, and how to combine an automatically generated parser and some hand-written code to form the compiler.

IX. WRITING PCYACC GRAMMAR DESCRIPTIONS

This Chapter examines how to write grammar description programs in PCYACC starting with a review of the typical structure of grammar description files; the three-sectioned-structure.

1. Structure of Grammar Description Programs

You have seen two examples of working PCYACC grammar description programs. To review, the general architecture of grammar description programs can be divided into three parts. These three parts are called the declaration section, grammar rule section and program section. The separator for sections is the special symbol %%.

The following structural template can be used to write a PCYACC grammar description program:

```
Declaration Section
%%
Grammar Rule Section
%%
Program Section
```

To PCYACC, the most important part in a grammar description program is the grammar rule section, which defines source languages for the parsers to be generated. Either the declaration section or the program section can be empty. If the program section is absent, the second %% delimiter can be omitted. The simplest grammar description looks like this:

```
%%
Grammar Rule Section
```

The following is a legal PCYACC grammar description that describes the decimal digits.

```
%%

digit :   '0' | '1' | '2' | '3' | '4'
          '5' | '6' | '7' | '8' | '9'
```

In this example, The grammar description program has only a single rule section, which is made up of ten alternative grammar rules for describing decimal digits. In general, white space characters (including new line, tab,

and blank characters) are not significant when writing grammar descriptions. Their only function is to separate words.

2. The Declaration Section

The declaration section is used to introduce names and/or their attributes. There are two kinds of names, those to be used by PCYACC in order to generate parsers and those for inclusion in the parsers generated by PCYACC.

In the declaration section, everything enclosed by %{ and %} is left unchanged and is directly copied to the PCYACC output. Typically, this mechanism is used to perform global header file inclusions and global variable definitions. For example:

```
 %{

#include <string.h>
#include "global.h"

#define STSZ 64
#define NMSZ 32

char namestack[STSZ][NMSZ];
int  sp;

%}
```

Since all lines enclosed by %{ and %} are copied to the parser, they must be in correct C syntax. Note that you may have more than one enclosed C segment in the declaration section, or even intermingled with YACC symbol declarations. To illustrate, the previous example can be rewritten as:

```
 %{

#include <stdio.h>
#include "global.h"
#define  STSZ 64
#define  NMSZ 32

%}

%{
char namestack[STSZ][NMSZ]
int  sp;
%}
```

Names that are useful only during PCYACC processing are introduced using PCYACC key words. Recall that a context-free grammar has four components: terminal symbols, nonterminal symbols, grammar rules and a start symbol. The most important function of the declaration section is to

make symbol classes explicitly known to PCYACC. The key word **tokens** is used to declare terminal symbols and **start** is used to declare the start symbol. Symbols appearing in a grammar rule but not declared as tokens are assumed to be nonterminals, with one exception. Single characters enclosed by single quotes are always treated as terminal symbols. The following fragment is taken from the declaration section of the SACALC example. This fragment declares the symbol **NUMBER** to be a terminal symbol, and the symbol **list** to be the start symbol for the SACALC grammar:

```
%token NUMBER
%start list
```

Note that keywords always begin with a percent sign (%). Several other key words used by PCYACC are listed below.

```
%type
%left
%right
%prec
%nonassoc
```

3. The Grammar Rule Section

The grammar rule section is used to specify grammar rules of a context-free grammar. One or more grammar rules can be specified in this section, each containing a lefthand side followed by a righthand side, separated using a colon, :, and terminated using a semicolon, ;. For example:

```
expr : expr '+' expr ;
```

Grammar rules with the same LHS can be combined using a vertical bar, '|', for example:

```
expr : NUMBER
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | '(' expr ')
     ;
```

The grammar rule section is the most important part of a PCYACC grammar description program. Keep the following guideline in mind when writing grammar rule sections:

1). terminal symbols cannot be the LHS of any grammar rule,
2). a non-terminal symbol must be the LHS of some grammar rule.

4. The Program Section

The last section of a PCYACC grammar description program is the program section. The contents of this section are not used by PCYACC but are copied to the parser program generated by PCYACC in their entirety.

Since PCYACC generates a parser in the form of a C function, you have a choice between putting other required hand-written C functions in the program section of the grammar description file, or writing them in separate C source files. For small projects it is often more convenient to include C functions in the grammar description file in order to obtain an integrated C program. For complex projects, it is better to use separate files to facilitate debugging and testing.

5. Associating Actions with Grammar Rules

So far PCYACC has been presented as a context-free grammar processor that can convert a context-free grammar into a language acceptor for the language defined by the grammar. This by itself is not particularly useful. For

example, suppose you have derived a language acceptor using PCYACC for the language defined by the following grammar description:

```
%token NUMBER
%start expr

%%

expr : expr '+' expr
     | NUMBER
     ;
```

Imagine you feed the program generated by PCYACC with the following two terminal strings:

1). 34 + 66
2). 34 + 66 +

You would expect the program to accept the first terminal string as a legal expression while rejecting the second. The next step is to be able to evaluate expressions after they are accepted. This is where actions come into play. Actions are associated with rules and are carried out immediately after reductions are made using the corresponding rules. To evaluate syntactically correct expressions you can rewrite the example PCYACC description as follows:

```
%token NUMBER
%start expr

%%

expr   :  expr '+' expr
          {
             $$ = $1 + $3;
          }
       |  NUMBER
          {
             $$ = $1;
          }
       ;
```

If the same expressions are fed to the program generated by PCYACC, you will obtain a value of 100 for the first expression and still reject the second expression.

Actions are C language statements enclosed in curly brackets. In the previous example, { $$ = $1 + $3; } and { $$ = $1; } are actions. Grammar symbols (both terminals and non-terminals) appearing in a grammar rule can possess values. The values of grammar symbols can be referenced from within action

statements associated with the rule. The convention $$ represents the value of the LHS non-terminal symbol of a grammar rule, $1 represents the value of the first grammar symbol of the RHS, $2 the value of the second symbol of the RHS, etc. In the example, the action { $$ = $1 + $3; } is attached to the grammar rule

```
expr --> expr '+' expr,
```

Thus, $$ represents the value of the first occurrence of the non-terminal symbol, expr, $1 represents the second occurrence and $3 the third occurrence. The meaning of this grammar rule when combined with its associated action is: If the current sentential form contains the pattern:

```
expr '+' expr,
```

then - replace it with the nonterminal symbol, expr. The value of the resulting nonterminal symbol is computed by summing the values of the two expr's occurring on the RHS of the grammar rule. Note that the values of the two RHS symbols must be computed prior to this reduction step.

Let's examine the process in which an expression such as "34 + 66" gets evaluated. We will use **val**(X), where X is a grammar symbol, to mean the value for X in the description.

step 1: in the initial configuration, the first terminal symbol, 34, is a NUMBER. It is reduced to an expr using the rule

```
expr : NUMBER
```

The action associated with this rule, { $$ = $1; } is executed, resulting in

```
val(expr) = val(NUMBER) = 34
```

step 2: with configuration

```
expr
```

The next symbol is a plus sign. No rule is applicable to '+' so the configuration is changed to

```
expr '+'
```

step 3: similar to the first step,

```
val(expr) = val(NUMBER) = 66
```

step 4: now the rule

```
expr : expr '+' expr
```

is applicable to

```
expr '+' expr
```

to produce a reduction to an expr and start the execution of the action { $$ = $1 + $3; }. The results of this are:

```
val(expr) = val(expr) + val(expr) = 100,
```

which completes the evaluation process.

Note that the last step, the execution of the assignment statement refers to different instances of the variable symbol expr.

X. MORE ON PCYACC PROGRAMMING

So far PCYACC has been described as a self-contained software tool that can function without dependence on any support elements, which is far from true. For example, additional support is needed for an input function that reads raw text input from files containing source programs and presents them as terminal symbols to PCYACC. This is called a lexical analyzer or scanner. PCYACC is not equipped with this kind of input mechanism. As a tool, PCYACC has a number of built-in handles, which give PCYACC programmers control over its behavior and/or the behavior of the generated parser.

The objective of this Chapter is to present in detail the important features of PCYACC. For novice users, this chapter will help you to develop some skills and techniques. For serious application developers, this chapter will help you to become masters of the tool, so that you can fully utilize the power of PCYACC.

This chapter is organized as follows:

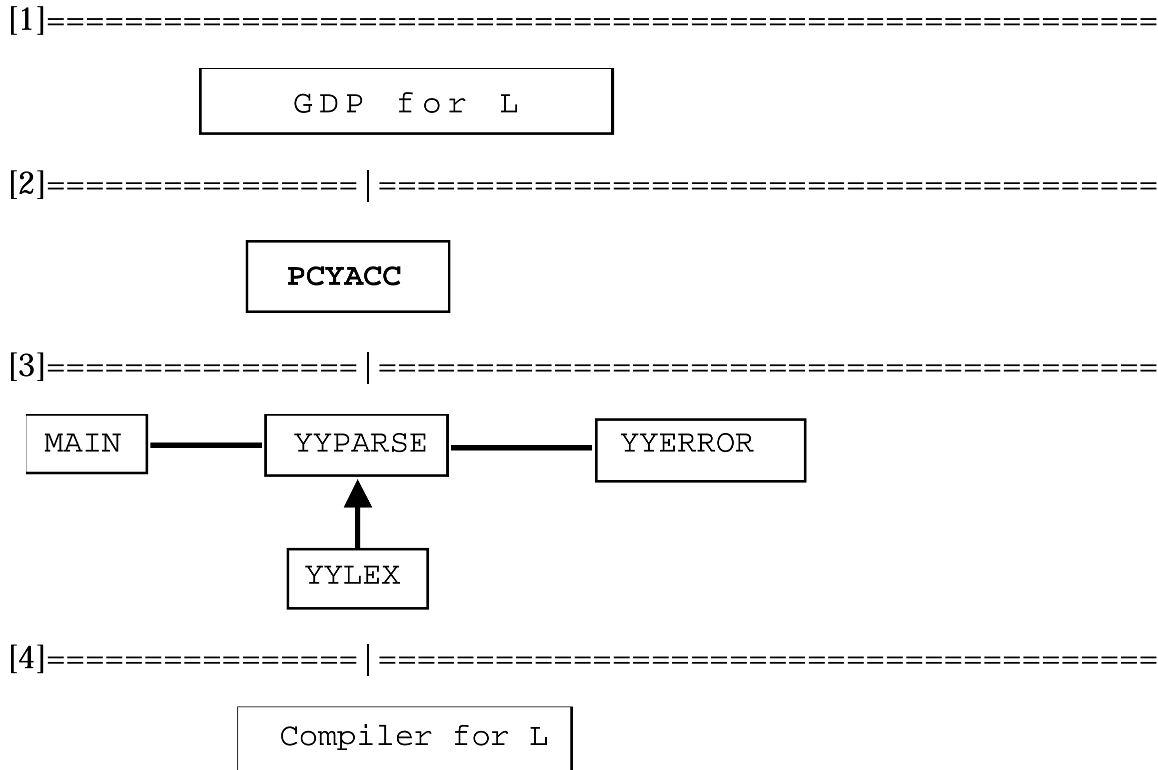
1). mandatory supporting functions
2). data types of grammar symbols
3). ambiguity resolution mechanisms
4). error recovery utilities

1. Mandatory Supporting Functions

As a PCYACC user, there are several things you have to provide to obtain a complete C program.

First, since the parser generated by PCYACC exists in the form of a C function, namely YYPARSE(), a driver routine, which normally is the main function, has to be supplied to activate the parser. Second, a lexical analyzer is required to digest raw text input and produce terminals. This lexical analyzer, YYLEX(), will be called by the generated parser when new terminal symbols are needed. Third, the parser assumes the existence of an error handling routine, YYERROR(), and delegates processing to it when the parser detects syntax errors in source programs.

The following diagram summarizes the relationship between PCYACC, the parsing function it generates, and the supporting functions it expects from the programmer.



The diagram agrees with the procedure for developing compilers with PCYACC, discussed in an earlier chapter. In the first block, a grammar description program is written for the language L. In the second block PCYACC translates the GDP into YYPARSE(), a parser for L. In the third block three C functions: MAIN(), YYLEX() and YYERROR() are added to the generated parsing function YYPARSE(). Their relationship, of caller-called, is depicted by the diagram. The fourth block represents the goal; a compiler for the language L is obtained.

2. The Role of the Drive Routine

In the most simple case, a main function (shown below) can be used to get the parser started.

```
main()
   {
     yyparse();
   }
```

The following skeleton main function is appropriate for most applications:

```
main(argc, argv)
   int argv; char *argv[];
   {
     preparse_preparation();
     yyparse();
     postparse_cleanup();
   }
```

The initialization function, PREPARSE_PREPARATION(), is involved with processing command line arguments, opening input files and setting up the lexical analyzer, YYLEX(). The wrap-up function, POSTPARSE_CLEANUP(), is usually very simple. It deals with things like reporting parsing status, success or failure, and closing files.

You can embed the invocation of the parsing function, YYPARSE() in other subfunctions instead of calling it from within the main routine directly. This method is most appropriate for more complicated applications.

3. The Role of the Lexical Analyzer

The parsing function does not work directly with raw program texts. Instead, it relies on the lexical analyzer to do the text scanning. The lexical analyzer converts source programs from their raw representations, namely, a sequence of characters, into a parser-understood intermediate form, sequence of terminals. For example, the expression

```
34 + 66
```

would be broken up into three terminal symbols as follows:

1). the NUMBER 34
2). the plus sign '+'
3). the NUMBER 66

What the parser actually sees is a three token sequence,

```
NUMBER '+' NUMBER,
```

rather than a seven character sequence (don't forget to count the two spaces). This text scanning and analysis capability relieves the parser from having to deal with these issues, achieving conceptual clarity of the source language specification. Consequently, the grammar description program in PCYACC is also simplified considerably. In the example, you can see that the three token sequence can be reduced by the parser to expr, because of the following grammar rules:

```
expr --> expr '+' expr
expr --> NUMBER
```

A clear distinction must be made between the value of a token and the type of the token. These are the most important concepts regarding information exchange between a parser and a scanner. The type of a token refers to the lexical category of the token, such as NUMBER. The value of a token refers to its actual value, such as 34. There are cases in which token values and token types correspond -- knowing one you can deduce the other. Examples include keywords such as **if**, and special symbols such as < >. While value and type distinction may not be crucial as far as information exchange is concerned, there are good reasons for keeping this conceptual separation. For example, tokens, such as NUMBER or **if**, are required to be declared in PCYACC grammar descriptions. It is infeasible to require numbers like 34 or 55 to always be predeclared in a similar fashion.

PCYACC is only interested in token types when it processes grammar description files for generating parsers. It is the scanner writer's responsibility to make sure that every token produced by the lexical analyzer has both a type and a value. The parser expects the lexical analyzer, YYLEX(), to return token types. Token values, on the other hand, are communicated via a global variable -- YYLVAL. A typical mode in which the lexical analyzer operates in responding to a parser request is:

1). read in the next token as a raw text stream
2). determine its type
3). determine its value and set the variable yylval
4). return the token type

To give you an idea how YYLEX() actually works, consider the following simple skeleton for YYLEX():

```
int yylex()
   {
     int c, i;

     c = getchar();          /* read next character */
     while (c == WHITESPACE)
       c = getchar();        /* skip white spaces */
     if (c == EOF)
       return (EOF);         /* end-of-file testing */

     if (isdigit(c)) {       /* a number */
       yylval.n = 0;
       while (isdigit(c)) { /* get an integer value */
         yylval.n += (c - '0');
         c = getchar();
       }
       ungetc(stdin);
       return (NUMBER);
     } else if (isletter(c)) {/* a keyword/identifier */
       i = 0;
       while (isalphnum(c)) {
         yylval.s[i++] = c;
         c = getchar();
       }
       yylval.s[i++] = '\0';
       ungetc(stdin);
       if (search(kwtable, yylval.s, &i))
         return(kwtable[i].type);
       else return (IDENTIFIER);
     } else ...                  /* other tokens */
       ...
     }
   }
```

First, the variable YYLVAL can be of some user defined union type, rather restricted to a predetermined default data type. Data types for grammar symbols will be discussed later in this Chapter. Second, in recognizing a NUMBER or an IDENTIFIER (a keyword), the only way to detect the NUMBER or IDENTIFIER in its entirety is by over-scanning. So the over-scanned character has to be put back to the input stream. Third, after finishing the scanning of a sequence of alphanumerical characters (started with a letter), a static keyword table must be searched to determine its real type. The rest of the routine is fairly self-explanatory.

4. The Role of PCYACC Generated Token Definitions

When PCYACC is invoked with "-d" option, it produces a header file called YYTAB.H, which contains all the definitions for tokens declared in the grammar description file.

This header file is most useful for the lexical analyzer, since its communication with the parser has to agree on these token (type) definitions. However, there are situations where other routines might also need access to definitions. This can be accomplished with the following include statement:

```
#include "yytab.h"
```

5. The Role of the Error Processing Routine

The third component required to build a working parser is an error handling routine, YYERROR(). When the parser encounters an error situation, it calls the error processing routine, YYERROR(). This is shown:

```
yyerror("Syntax error");
```

YYERROR() is expected to accept a string argument, which essentially is an error message.

An oversimplified error processing function is illustrated below. Note that in many cases more sophistication is necessary to produce comprehensive diagnostic messages and error recoveries.

```
void yyerror(s)
   char *s;
   {
     printf("%s\n", s);
   }
```

6. Data Types of Grammar Symbols

Types of grammar symbols were discussed in previous chapters. Previous examples assumed type int such as evaluating the expression

```
34 + 66
```

When discussing actions associated with grammar rules in the preceding Chapter, it was noted that you may associate arbitrary types with any grammar symbols. This subject will be discussed further in subsequent sections.

The most important data structure for an LR parser, such as the ones generated by PCYACC, is a stack. During parsing, a shift operation, more or less, refers to pushing a grammar symbol onto this stack, and a reduction operation, more or less, refers to replacing zero or more stack top elements with a single grammar symbol. When defining data types for grammar symbols, you are in effect defining a type for this stack. Since there are normally many grammar symbols, and they are not required to be of a uniform type, the only way to accommodate this nonuniformity is to define the stack as a union type.

If you do not change PCYACC's defaults, all grammar symbols are assumed to have the default type of **int**. Generally, there are two scenarios in which you may want to make use of this default type. First, when the default type coincides with what is required, such as the example of evaluating the expression

```
34 + 66.
```

The second case in which you would not change this default typing convention is when you decide to completely rewrite your own value passing mechanism using C data structures in the action code segments.

When dealing with types in PCYACC, follow this procedure:

1). define type YYSTYPE, which is used internally by PCYACC to specify the stack type;

2). associate different types with different grammar symbols using union tags and the keyword **type**;

3). manipulate symbol values properly within actions.

7. Defining YYSTYPE

There are two ways in which YYSTYPE can be defined. The first and easiest is to use PCYACC keyword **union**. For example, to use the value stack to handle two kinds of values, integer numbers and identifiers (sequence of alphanumerical characters starting with a letter), the following union definition can be added to the declaration section of the grammar description files:

```
%union {
     int  numval;
     char strval[IDSZ];
   }
```

The second way to accomplish the same thing is to define the union type directly in C syntax, and enclose the definition using the delimiters %{ and %}, as shown below.

```
%{

 typedef union {
   int numval;
   char strval[IDSZ];
 } YYSTYPE;

%}
```

Note that although the two methods for defining YYSTYPE seem different, they are closely related. PCYACC will translate the first declaration style into the second declaration style, which actually appears in the generated C-code parser. You can use the following C statements to handle simple declarations without union involved, as was shown in the SACALC example.

```
 %{

#define YYSTYPE double

%}
```

Or equivalently,

```
 %{

typedef double YYSTYPE;

%}
```

8. Associating Types with Grammar Symbols

The keyword **type** and union tags of YYSTYPE are used to associate types with grammar symbols. Consider the following grammar segment:

```
expr : expr '+' expr
     | NUMBER
     | IDENTIFIER
     ;
```

Suppose YYSTYPE is already defined as the union type in the preceding section. In this case, the following might well be part of the declaration section in the relevant grammar description file:

```
%token NUMBER
%token IDENTIFIER

%type <intval> NUMBER expr
%type <strval> IDENTIFIER
```

You can specify terminal symbol types within their token declarations. For example, the above declaration can be replaced by

```
%token <intval> NUMBER
%token <strval> IDENTIFIER

%type  <intval> expr
```

9. Manipulating Values of Grammar Symbols

Within actions of C-code segments, grammar symbols can be treated like ordinary variables (except they must be accessed using the dollar sign notation, like $$, $1, $2, etc.). They can be set to new values with assignment statements, can appear in expressions as operands, and can also be passed to subroutines for further processing.

However, since they are typed entities, it is the programmer's responsibility to make sure these manipulations on grammar symbols do not violate the typing rules stipulated by the C programming language. For example, assume the same type definition for grammar symbols in the previous section, and MAKENODE() is a C function that builds a new data structure of type nds:

```
struct nds {
     int op;
     struct nds *left;
     struct nds *right;
   };

struct nds *
   makenode(l, o, r)
   struct nds *l, *r;
   int o;
   {
     ...
   }
```

The following action clearly exhibits a type violation:

```
expr : expr '+' expr
         { $$ = makenode($1, '+', $3); }
        ;
```

Unfortunately, PCYACC is unable to detect this kind of type violation, since it will not look into actions. It is left to the C compiler to provide type checking and supply meaningful error messages.

There is a rudimentary check that PCYACC does perform. When a type other than the default is associated with a nonterminal symbol, PCYACC insists that relevant grammar rules have actions supplied by the programmer. For example, PCYACC will not accept the following grammar rule, provided the grammar symbol expr is of the type <intval>:

```
expr : expr '+' expr ;
```

This simple checking can only be used to help prevent nonintentional errors. Even the following rewrite can fool PCYACC:

```
expr : expr '+' expr { } ;
```

10. Ambiguity Resolution Mechanisms

This section will review the concept of ambiguity and examine the forms in which ambiguities occur in grammar description programs. If a sentence can be derived (reduced) using grammar rules in more than one canonical derivation (reduction), the grammar is ambiguous. For example, the following grammar (in PCYACC format) is ambiguous:

```
%token NUMBER
%start expr

%%

expr : expr '+' expr
         { $$ = $1 + $3; }
     | expr '*' expr
         { $$ = $1 * $3; }
     | NUMBER
         { $$ = $1; }
     ;
```

Given the expression:

5 + 5 * 2,

It has at least two distinct canonical derivations:

```
1). expr ==> expr '+' expr
         ==> expr '+' expr '*' expr
         ==> expr '+' expr '*' NUMBER
         ==> expr '+' NUMBER '*' NUMBER
         ==> NUMBER '+' NUMBER '*' NUMBER

2). expr ==> expr '*' expr
         ==> expr '*' NUMBER
         ==> expr '+' expr '*' NUMBER
         ==> expr '+' NUMBER '*' NUMBER
         ==> NUMBER '+' NUMBER '*' NUMBER
```

It appears that the two derivations should not make any difference, since the results of the derivations are the same in both case -- right? Wrong! There are semantic operations performed along with derivations (reductions). Different derivations (reductions), may lead to different operations being performed, or same operations being performed in different orders. To clarify, let's reexamine the previous example taking the reduction approach.

```
1). NUMBER(5) '+' NUMBER(5) '*' NUMBER(2) ==>
    expr  (5) '+' NUMBER(5) '*' NUMBER(2) ==>
    expr  (5) '+' expr  (5) '*' NUMBER(2) ==>
    expr  (5) '+' expr  (5) '*' expr  (2) ==>
    expr  (5) '+' expr  (10)              ==>
    expr  (15)

2). NUMBER(5) '+' NUMBER(5) '*' NUMBER(2) ==>
    expr  (5) '+' NUMBER(5) '*' NUMBER(2) ==>
    expr  (5) '+' expr  (5) '*' NUMBER(2) ==>
    expr (10) '*' NUMBER(2)               ==>
    expr (10) '*' expr  (2)               ==>
    expr (20)
```

The effects of semantic action are included in parenthesis. Note that only the result produced by the first derivation (reduction) is considered correct.

In an LR parser, there are four possible actions at each parsing step, shift, reduce, accept and error. An accept action is a special case of a reduction where the LHS of the reduction rule is the start symbol. An error action refers to a situation in which no further shift or reduce actions are applicable; when the parsing process is stuck. Accept actions and error actions can not interfere with any of the other remaining actions, nor can they interfere with each other.

There are three remaining conflicts to consider: shift/shift, shift/reduce and reduce/reduce. With further study we can eliminate shift/shift as a possible candidate for conflict. This is because a shift is essentially consuming the first terminal symbol in the input stream. If two shifts contend on the same terminal symbol, the two shifts might be merged to become a single shift operation.

Grammar ambiguities appear as two kinds of conflicts in LR parsers; shift/reduce conflicts and reduce/reduce conflicts. A shift/reduce conflict occurs when both a shift action and a reduce action are applicable in a parsing step. For example, during parsing of the expression

```
5 + 5 * 2,
```

Suppose you have executed the following:

```
        NUMBER(5) '+' NUMBER(5) '*' NUMBER(2) ==>
        expr  (5) '+' NUMBER(5) '*' NUMBER(2) ==>
        expr  (5) '+' expr  (5) '*' NUMBER(2) ==>
```

Now you have the expression in a parsing state. The parser stack contains the following grammar symbol sequence:

```
expr '+' expr,
```

and the input stream becomes

```
'*' NUMBER(2).
```

You have a choice between using a shift operation to consume the terminal symbol '*', or performing a reduction on the grammar symbols on the stack using

```
expr : expr '+' expr".
```

Thus, a shift/reduce conflict occurs at this stage.

Similarly, a reduce/reduce conflict occurs when two of more grammar rules are applicable simultaneously for a reduction operation in a parsing step. For example, suppose you have a small programming language described as follows:

```
%token NUMBER
%token IDENTIFIER
%token GOTO
%start program

%%

program   : statement
          | program statement
          ;

statement : assign_st
          | goto_st
          | label_st
          | expr
          ;

assign_st : IDENTIFIER '=' expr
          ;

goto_st   : GOTO label
          ;

label_st  : label
          ;

label     : IDENTIFIER               (*)
          ;

expr      : expr '+' expr
          | NUMBER
          | IDENTIFIER               (**)
          ;
```

This grammar exhibits a reduce/reduce conflict. The problem is caused by the two grammar rules marked by (*) and (**). When the parser encounters an IDENTIFIER, and it decides to do a reduction, it has difficulty deciding which grammar rule should be used.

11. Resolving Shift/Reduce Conflicts

PCYACC has built-in conflict-resolving rules for handling both shift/reduce and reduce/reduce conflicts.

In the case of a shift/reduce conflict, the default rule is in favor of the shift operation. The parser always prefers shift actions to reduce actions, whenever both are applicable.

Most of the time when shift/reduce conflicts occur, the shift operation is the correct action to perform. A typical example supporting this default rule is the **if-then-else** statement commonly seen in programming languages:

```
if_st : IF condition THEN statement
        | IF condition THEN statement ELSE statement
        ;
```

This grammar rule clearly gives rise to a shift/reduce conflict when **if** statements are parsed. However, the default rule says that if this kind of conflict occurs, to perform the shift to read in the next terminal symbol, ELSE. Note that this processing strategy coincides with the semantics of **if** statements used in a majority of programming languages, namely, that ELSE's should be matched with their closest IF's.

Unfortunately, there are whole classes of syntactical structures commonly used in programming languages that don't fit into this default rule. They are expressions. For example, when parsing the expression

```
2 * 5 + 5,
```

you would not want to use shifting. To use shifting would violate the elementary arithmetic law that multiplication has a higher precedence than addition. To get a round this problem, PCYACC allows you to specify precedence and association for grammar symbols.

Three keywords are provided for this purpose -- **left**, **right** and **nonassoc**, meaning left associative, right associative and nonassociative respectively. For example, the statement

```
%left '+' '-'
```

says that both '+' and '-' are left associative.

Precedences are implicitly defined using the order of associativity specification statements. Symbols listed in the same line have the same precedence, and they have higher precedence than the symbols listed in previous line. For example, the following declaration reflects a well known arithmetic principle:

```
%left '+' '-'
%left '*' '/'
```

12. Resolving Reduce/Reduce Conflicts

By default, PCYACC resolves reduce/reduce conflicts in favor of the grammar rule that appears first in a grammar description file.

This default rule is not as meaningful as the default rule for resolving shift/reduce conflicts. Therefore, it is important to realize that when a PCYACC grammar description contains reduce/reduce conflicts, it usually means the design of the language should be studied carefully, or the grammar needs rewriting to correctly represent the design.

If a reduce/reduce conflict is the result of incorrect language design, the language must be redesigned. On the other hand, if the conflict arises as a result of incorrect grammar writing, we do have some limited ways to remedy the situation.

A general technique for eliminating reduce/reduce conflicts is to use backward substitution. The idea of backward substitution is to directly use the right hand sides of conflicting rules where appropriate, instead of introducing extra grammar variables. For instance, in the earlier example of reduce/reduce conflicts, the following grammar rules were involved:

```
statement : label_st
          | expr
          ;

goto_st   : GOTO label
          ;

label_st  : label
          ;

label     : IDENTIFIER          (*)
          ;

expr      : IDENTIFIER          (**)
          ;
```

To eliminate the reduce/reduce conflict resulting from the two rules marked by (*) and (**), you can remove the rule marked by (*), and use the terminal symbol IDENTIFIER directly in the remaining rules involved. The result of this change is shown below:

```
statement : label_st
          | expr
          ;

goto_st   : GOTO IDENTIFIER
          ;

label_st  : IDENTIFIER    (*)
          ;

expr      : IDENTIFIER    (**)
          ;
```

But, there are still reduce/reduce conflicts. To correct this, remove the rule for label_st and do backward substitution one more time. Note that you can omit label_st (IDENTIFIER) alternative for the statement rule, since expr already has provision for IDENTIFIER's. Using this method you can effectively leave more processing work to the semantic analyzer. The final result is:

```
statement : expr
          ;

goto_st   : GOTO IDENTIFIER
          ;

expr      : IDENTIFIER
          ;
```

Eliminating conflicts from grammar rules is not always easy. There are no general rules for how to go about it. However, in general it is true that there is a tradeoff between the syntax processor and the semantic processor. The more work you leave to the semantic processor, the easier the syntax processor to build.

13. Resolving Ambiguities -- A Summary

In the same way a token symbol may have a precedence and an association, a grammar rule may also be associated with a precedence and an association. Normally, a grammar rule has the same precedence and association as the last terminal symbol of its right-hand side, if the terminal symbol has a precedence and an association. Otherwise, the grammar rule has no precedence or association. But, a grammar rule may also be explicitly given a precedence and an association with the keyword **prec**. This feature is particularly useful in grammar specifications where the same token symbol has different meanings in different contexts. For example, the minus sign "-" can be used as both a binary operator (meaning subtraction) and a unary operator (meaning negation) in arithmetic expressions. This is handled properly in the following grammar:

```
%token NUMBER
%left  '+' '-'
%left  '*' '/'
%left  UNARYMINUS
%start expr

%%

expr : expr '+' expr
       { $$ = $1 + $3; }
     | expr '-' expr
       { $$ = $1 - $3; }
     | expr '*' expr
       { $$ = $1 * $3; }
     | expr '/' expr
       { $$ = $1 / $3; }
     | '-' expr %prec UNARYMINUS
       { $$ = - $2; }
     | NUMBER
       { $$ = $1; }
     ;
```

Here is a summary of the rules and mechanisms available in PCYACC for resolving ambiguities.

1). Each terminal symbol may have a precedence and an association, declared using left, right, or nonassoc.

2). Each grammar rule is associated with the same precedence and association with that of the last terminal symbol of its righthand side. This can be overridden using %prec.

3). When a shift/reduce conflict or a reduce/reduce conflict occurs, and relevant input terminals or grammar rules have no precedence and associativity, the default rules will be used.

4). When a shift/reduce conflict occurs, and both involved input terminals and grammar rules have associated precedence and associativity, the conflict is resolved as follows:

A). if the precedence of the input terminal is higher than the precedence of the grammar rule, perform shift action;

B). if the precedence of the input terminal is lower than that precedence of the grammar rule, perform reduce action;

C). if both have the same precedence, then the associativity of the input symbol is used:

a). left associative implies reduce;

b). right associative implies shift;

c). nonassociative implies error.

14. Error Recovery Utilities

If you have written parsers before, you already know that good error handling is one of the most important features. It is also one of the issues that is most difficult to deal with. The situation is even more difficult for automatically generated parsers.

Normally, upon detecting a syntax error, parsers generated by PCYACC will first call YYERROR(), which is a programmer supplied function, then abort the processing completely. This means you can only uncover syntax errors one at a time, which is unsatisfactory.

Ideally, you would like to be able to detect all syntax errors in a program with a single compilation. PCYACC has a number of built-in mechanisms that allow programmers to have control over the error recovery process. PCYACC provides you with one predefined terminal symbol, **error**, and two predefined actions, **yyerrok** and **yyclearin**.

Actually, you have seen the predefined terminal symbol error before. The SACALC example contained the following grammar rules:

```
list : /* empty */
     | list '\n'
     | list expr '\n'
     | list error '\n'          (*)
     ;

expr : expr '+' expr
     | NUMBER
     ;
```

The fourth rule (marked with *) is added for the purpose of error recovery. If it was omitted from the grammar, trying to evaluate a syntactically incorrect expression, such as

```
2 + register '\n'
```

would cause the SACALC program to halt. On the other hand, the inclusion of the error recovery rule would restart SACALC, which is desirable in most cases.

Let us trace the process of evaluating

```
2 + register '\n'
```

and see how the recovery rule makes restarting of the parser possible. The process proceeds as follows (using the notation *(...S)::X Y Z* to represent the parser state: stack with top S and input stream is *X Y Z*):

1). start the parser with

```
() :: 2 + register '\n'
```

2). reduce using the first list rule

```
(list) :: 2 + register '\n'
```

3). shift

```
(list 2) :: + register '\n'
```

4). reduce using the second expr rule

```
(list expr) :: + register '\n'
```

5). shift

```
(list expr +) :: register '\n'
```

6). shift

```
(list expr + register) :: '\n'
```

At this point, if there are no error rules marked by (*), the only way for the parser to succeed is to be able to reduce the first three symbols of the stack expr. This is in turn dependent on reducing the symbol register to expr, which is impossible. In this case the parser fails and aborts processing. However, with the presence of the error rule, the parser can continue processing with:

7). reduce using the error token

```
(list error) :: '\n'
```

8). shift

```
(list error '\n') ::
```

9). reduce using the error rule

```
(list) ::
```

The result is a successful parse of the erroneous expression, which in effect restarts the parser for processing of the next expression.

Note that the idea of using a predefined symbol error to perform recovery is based on the fact that compiler writers are able to predict where errors are most likely to occur.

By default, when an error rule is used to do a reduce, the parser generated by PCYACC will quietly skip the next three input tokens and restart processing from that point. For example, this would be the case if you rewrite the error recovery rule as

```
list : list error ;
```

While the original rule

```
list : list error '\n' ;
```

instructs the parser to skip all input tokens preceding (and including) a carriage return, then restart the normal processing. The latter form of error recovery rules are in general easier to use since they make the error recovery process more visible to the programmer.

Actions may also be associated with error recovery rules. This is the place PCYACC allows programmers to write their own error recovery routines. The two predefined macros can be used in conjunction with user defined error recovery utilities. **yyerrok** clears the error flag set when the parser gets into an error state, and **yyclearin** erases the previous look-ahead input token. Both are needed when the error recovery mechanisms provided by the user are capable of bringing the parser to a safe state so that it can be restarted without any help from the environment.

15. Imbedded Actions

Actions are not actually part of the grammar they are only executed during a reduction. Sometimes it may be desired to imbed actions between terminals or in the middle of a production.

```
list : list { list_act(); } expr { expr_act(); }
     ;
```

However, if you compile this grammar it will generate shift-reduce errors, and sometimes the debugging can become quite difficult. So, the following can be used as a work around. A special note PCYACC will do this automatically, but the addition of the new empty set may imbalance your whole grammar leading to a debugging nightmare - depending on the complexity of your grammar.

```
list : list temp expr { expr_act(); }
     ;

temp : /* empty */ { list_act(); }
```

To avoid creating the empty set on the temp production the following solution can be used.

```
list : list_ expr { expr_act(); }
     ;

list_ : list { list_act(); }
```

The above solution is the preferred method of imbedding actions within productions. This method will always produce portable, and maintainable parsers. Note, we have accomplished the goal of imbedding actions without the debugging problems caused by the addition of the empty set.

XI. DEBUGGING -- TOOLS AND TECHNIQUES

Debugging in PCYACC programming refers to two different error correcting actions. One is correcting errors in grammar rules, and the other is debugging C programs. These C programs can be generated by PCYACC or hand-written supporting routines. However, this term is used exclusively to refer to the action of correcting errors in grammar rules. Debugging C programs is an important and extensively studied issue, and you should have no trouble finding good references if needed.

Though not necessarily obvious, errors that occur during a compiler development project can be classified into three types:

1). syntax error;
2). symbol usage error;
3). grammar rule specification error.

1. Correcting Syntax Errors

Syntax errors in grammar rules can be fixed with little effort, since most grammar description files are short in size. This makes syntax errors easy to locate and to fix.

Typical syntax errors result from missing delimiters, and/or improper use of punctuation’s. Examples of common syntax errors are illustrated below:

1). missing the delimiter "%%"

```
%token NUMBER
%start expr

expr : expr '+' expr
     | NUMBER
     ;
```

The problem with this grammar description program segment is that there should be a delimiter "%%" separating the declaration section from the program section.

2). missing the semicolon; ";"

```
%token NUMBER
%start expr

%%

expr : expr '+' expr
     | NUMBER
```

The problem here is that a semicolon ";" should always be used to terminate a sequence of alternative right-hand sides of grammar rules.

3). improper use of the colon ":"

```
%token NUMBER
%start expr

%%

expr : expr '+' expr
     : NUMBER
     ;
```

The problem in this grammar rule segment is that the vertical bar "|" instead of the colon should be used to separate alternative right-hand sides.

4). improper use of the vertical bar "|"

```
%token NUMBER
%start expr

%%

expr | expr '+' expr
     | NUMBER
     ;
```

The problem in this example is just the opposite of the previous one. The colon instead of the vertical bar should be used to separate the first alternative RHS from the LHS.

The examples definitely do not exhaust every possibility for syntax errors, but they are the most common and most easily fixed.

2. Correct Symbol Usage Errors

Errors in this category can still easily be fixed. Typically, improper use of symbols are caused by:

1). missing required token declarations,
2). missing required actions to enforce type consistency, or
3). uninstantiable nonterminal symbols.

Here is a list of symbol usage errors that most commonly occur:

1). undeclared token "<>"

```
%token NUMBER
%start expr

%%

expr : expr '+' expr
     | expr '<>' expr
     | NUMBER
     ;
```

A quick reminder. Although single character symbols, such as the plus sign (+), do not need explicit declarations and are treated as terminal symbols automatically, commonly used operators made up of two or more characters, such as the not equal comparator (<>), do not follow this rule. Extra terminals have to be introduced explicitly to represent these combined operators. The correct way to construct the grammar is:

```
%token NUMBER
%token NOT_EQUAL
%start expr

%%

expr : expr '+' expr
     | expr NOT_EQUAL expr
     | NUMBER
     ;
```

Note: there are some implementations of YACC that treat quoted strings as terminal symbols automatically, thus their declaration can be optional. But, to avoid unexpected results, it is best to declare them explicitly.

2). uninstantiable nonterminal "neq_expr"

```
%token NUMBER
%token NOT_EQUAL
%start expr

%%

expr : add_expr
     | neq_expr
     | num_expr
     ;
add_expr : expr '+' expr
         ;
num_expr : NUMBER
         ;
```

In this grammar description segment, the nonterminal symbol "neq_expr" can not derive any terminal string, which PCYACC will report as an error.

3). absence of actions for "expr" rules

```
%union {
   float floval;
}

%token <floval> NUMBER
%type  <floval> expr
%start expr

%%

expr : expr '+' expr
     | NUMBER
     ;
```

Another reminder: once a grammar symbol has been associated with a type other than the default type, grammar rules with the symbol on the left-hand side must have actions with a value of proper type assigned to the grammar symbol.

3. Correcting Grammar Rule Errors

Errors in this class go much deeper than the ones in the first two, and are in general hard to rectify. To a large extent, correcting errors of this kind involves resolving two kinds of conflicts, shift/reduce and reduce/reduce, which were discussed in previous Chapters. The remaining errors in this category are more serious. They are due to incorrect grammar specifications or language design.

There are no universally applicable techniques for fixing these types of errors. However, PCYACC does give you some help. PCYACC can generate parsing tables for grammar specifications. Parsing tables record states of the parser, and for each state what the next state would be upon seeing an input token, and what action (shift, reduce, accept, error) would be taken with this state transition. The information is helpful for debugging grammar rules with logical errors. Therefore, it is important for PCYACC programmers to know how to read parsing tables. Some small examples are provided in this section to explain entries in these tables.

3.1 How to Read Parsing Tables

The "-v" (verbose) command line switch tells PCYACC to produce a parsing table for the parser being generated. PCYACC will then produce the table and store it to the current folder a file named YY.LRT. (The extension LRT stands for LR parsing Table.)

First, let's look at a simple grammar, BINARY.Y:

```
%start binary

%%

binary : '0' | '1' ;
```

Invoking PCYACC with -v switch on BINARY.Y will produce the following YY.LRT file:

```
-*-=-*-=-*-=-*-      LALR PARSING TABLE      -*-=-*-=-*-=-*-

+--------------------     STATE 0     ---------------------+

+ CONFLICTS:

+ RULES:
        $accept : ^binary $end

+ ACTIONS AND GOTOS:
              0 : shift & new state 2
              1 : shift & new state 3
                : error

         binary : goto state 1

+--------------------     STATE 1     ---------------------+

+ CONFLICTS:

+ RULES:
        $accept :  binary^$end

+ ACTIONS AND GOTOS:
           $end : accept
                : error

+--------------------     STATE 2     ---------------------+

+ CONFLICTS:

+ RULES:
         binary :  0^    (rule 1)

+ ACTIONS AND GOTOS:
                : reduce by rule 1

+--------------------     STATE 3     ---------------------+

+ CONFLICTS:

+ RULES:
         binary :  1^    (rule 2)

+ ACTIONS AND GOTOS:
                : reduce by rule 2
```

```
====================      SUMMARY      ====================

grammar description file = binary.y
number of terminals used =      4; limit =    500
number of nonterminals   =      1; limit =    500
number of grammar rules  =      3; limit =   1000
number of states         =      4; limit =   1000
number of s/r errors     =      0
number of r/r errors     =      0
number of working set    =      3; limit =    600
memory for rules & stats =     30; limit =  20000
memory for actions       =      0; limit =  20000
number of lookahead sets =      3; limit =    600
number of extra closures =      0
number of shift entries  =      2
number of exception ents =      1
number of goto entries   =      1
number of goto defaults  =      0
optimization input       =      7; limit =  20000
optimization output      =      3; limit =  20000
number of table entries  =      3; zeros =      0
maximum spread           =     49; offst =     48

-*-=-*-=-*-=-*-         END OF TABLE         -*-=-*-=-*-=-*-
```

As you can see, in addition to state entries, there is also a summary section attached to the YY.LRT file. The statistics given in the summary section provides information about the size of the grammar, memory usage etc. This information is useful for dealing with very large grammars. You may also notice that the parsing table file is structured as follows:

```
entry for state 0
entry for state 1
  ...
summary
```

The structure for each entry is the following:

CONFLICTS
RULES
ACTIONS AND GOTOS

There are three subentries in each entry. The CONFLICTS subentry contains conflicts information. The RULES subentry lists the rules applicable in the state and ACTIONS AND GOTOS subentry provides information on parser's action in the state. For example, in the entry for state 0, you can see no conflicts occur in the state, since the CONFLICTS subentry is empty. There is only one applicable rule in the state:

$accept : ^binary $end

From the ACTIONS AND GOTOS subentry, you find the following three lines:

```
0 : shift & new state 2
1 : shift & new state 3
  : error
```

The meaning of these lines can be understood as follows: given that the parser is in state 0, the parser will do a shift operation and change its state to state 2 if it sees the token 0. Similarly, the parser will do a shift operation and change its state to state 3 if it sees the token 1. The parser is in error if it sees anything else.

The third subentries of state entries are usually combined and more concisely represented using a tabular form in literature on parsing techniques.

The tabular form for the example grammar is the following:

	Input Token			
State	0	1	$end	binary
0	shift/2	shift/3	error	1
1	error	error	accept	
2	reduce/1			
3		reduce/2		

Note that the rule

```
$accept : ^binary $end
```

is not in the grammar file BINARY.Y. PCYACC always adds a rule like this one to your grammar. The general rule for PCYACC to add the additional rule to your grammar is: suppose the start symbol of your grammar is START, then the rule

```
$accept : START $end
```

is added. $accept is an added non-terminal symbol, and $end is an added terminal symbol. PCYACC uses $accept as the internal start symbol. The purpose is to make sure that the start symbol of a grammar does not appear on the right-hand side of any grammar rule.

Note also that there is a caret (^) added to all grammar rules in the parsing table. The caret is used as a location mark. When inserted between two grammar symbols, the caret means the parser has seen the symbol to the marker's left, while expecting to see the symbol to the marker's right. A

marker following the very last symbol of a grammar rule means a possible reduce may occur at this state using this grammar rule.

From the summary section, you can find the name of the grammar file is BINARY.Y. The grammar has 4 terminal symbols ('0', '1', '$end' and 'error' you have seen in the previous Chapter), 1 non-terminal symbol (binary), 3 grammar rules (including the one added by PCYACC), 4 parsing states, and no conflicts, etc.

3.2 Locating Conflicts in the Grammar

To appreciate the -v switch of PCYACC, let's look at a grammar with errors:

```
%token NUMBER
%start expr

%%

expr : expr '+' expr
     | expr '*' expr
     | NUMBER
     ;
```

Two state entries and some summary information from the YY.LRT file for the grammar are included below.

```
-*-=-*-=-*-=-*-      LALR PARSING TABLE      -*-=-*-=-*-=-*-
...

+--------------------     STATE 5     ---------------------+

+ CONFLICTS:
? sft/red (shift & new state 3, rule 1) on +
? sft/red (shift & new state 4, rule 1) on *

+ RULES:
           expr :  expr^+ expr
           expr :  expr + expr^    (rule 1)
           expr :  expr^* expr

+ ACTIONS AND GOTOS:
              + : shift & new state 3
              * : shift & new state 4
                : reduce by rule 1

+---------------------     STATE 6     --------------------+

+ CONFLICTS:
? sft/red (shift & new state 3, rule 2) on +
? sft/red (shift & new state 4, rule 2) on *

+ RULES:
           expr :  expr^+ expr
           expr :  expr^* expr
           expr :  expr * expr^    (rule 2)

+ ACTIONS AND GOTOS:
              + : shift & new state 3
              * : shift & new state 4
                : reduce by rule 2

====================      SUMMARY      ====================

grammar description file = expr2.y
number of terminals used =      5; limit =    500
number of nonterminals   =      1; limit =    500
number of grammar rules  =      4; limit =   1000
number of states         =      7; limit =   1000
number of s/r errors     =      4
number of r/r errors     =      0

...

-*-=-*-=-*-=-*-         END OF TABLE         -*-=-*-=-*-=-*-
```

The information in the summary section indicates there are four shift/reduce conflicts in the grammar specification.

Two conflicts occur at state 5:

```
? sft/red (shift & new state 3, rule 1) on +
? sft/red (shift & new state 4, rule 1) on *
```

The problem is that after the parser has seen the grammar symbol sequence

```
expr + expr
```

and the next input token is a plus sign (+) or a multiplication sign (*), there are two possible actions for the parser to perform. The parser can perform a shift to consume the plus sign or multiplication sign, or it can perform a reduce by rule 1. The conflict situation at state 6 is similar. In the ACTIONS AND GOTOS subentry of the parsing table, it is also shown that the parser selects the shift operation to perform in shift/reduce conflicts.

To remove the conflicts in the previous grammar the grammar can be rewritten as follows:

```
%token NUMBER
%left  '+'
%left  '*'
%start expr

%%

expr : expr '+' expr
     | expr '*' expr
     | NUMBER
     ;
```

You have successfully removed the conflicts from the grammar:

```
-*-=-*-=-*-=-*-      LALR PARSING TABLE      -*-=-*-=-*-=-*-
...

+--------------------     STATE 5     ---------------------+

+ CONFLICTS:

+ RULES:
           expr :  expr^+ expr
           expr :  expr + expr^    (rule 1)
           expr :  expr^* expr

+ ACTIONS AND GOTOS:

              * : shift & new state 4
                : reduce by rule 1

+--------------------     STATE 6     ---------------------+

+ CONFLICTS:

+ RULES:
           expr :  expr^+ expr
           expr :  expr^* expr
           expr :  expr * expr^    (rule 2)

+ ACTIONS AND GOTOS:
                : reduce by rule 2

====================      SUMMARY      ====================

grammar description file = expr1.y
number of terminals used =      5; limit =    500
number of nonterminals   =      1; limit =    500
number of grammar rules  =      4; limit =   1000
number of states         =      7; limit =   1000
number of s/r errors     =      0
number of r/r errors     =      0

...

-*-=-*-=-*-=-*-         END OF TABLE         -*-=-*-=-*-=-*-
```

The two **%left** declarations eliminated four shift/reduce conflicts. It is also interesting to note, from the two state entries shown above, how the precedence of operators has changed the behavior of the parser.

XII. CONSTRUCTING COMPILERS -- REVISITED

This chapter discusses some methods of compiler construction, although a through review of this subject is beyond the scope of this manual. We will try to cover most of the important aspects of the discipline.

This Chapter will examine a typical architecture for compilers, then describe each part using a separate subsequent section. This section will provide you with a concrete and clear picture of what real compilers look like, their components and how they can be built. This will help you to understand what PCYACC adds to a compiler development project, and how it helps to reduce the programmer's coding burden.

1. Basic Architecture

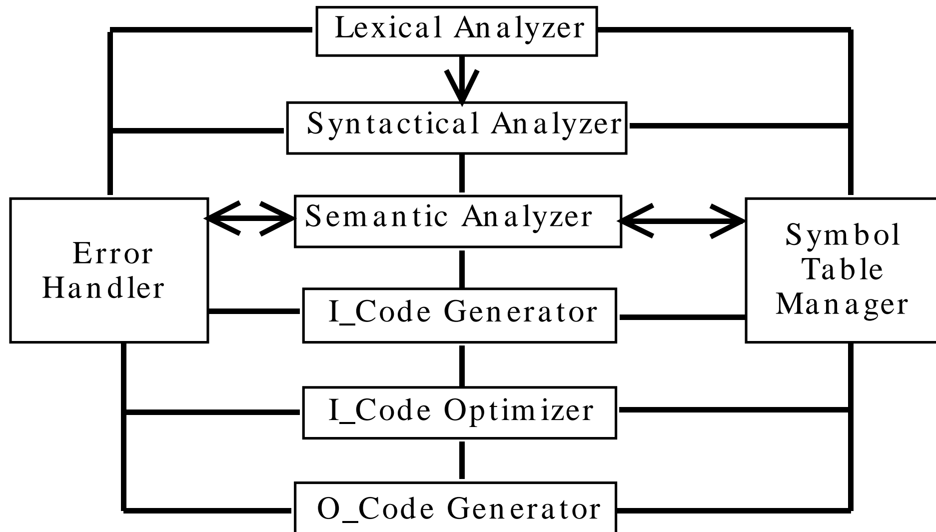
A compiler is a program that accepts a source program as input and produces an equivalent object program as output. This language translation process can be quite complex, though not impossible to do. For example, the first FORTRAN compiler took about fifteen man-years to build.

Based on careful studies and extensive experience, it is recognized that this complex language translation process can be broken up into several simpler, fairly independent, yet well-defined activities. The following language translation phases can be identified:

1). lexical analysis;
2). syntax analysis;
3). semantic analysis;
4). intermediate code;
5). intermediate code optimization;
6). code generation;

This multi-phase approach to compiler construction has been proven successful -- not only has it greatly reduced the complexity of the language translation process, it has also stimulated studies on each of the subprocesses. These studies produced many automated compiler development tools that are still in existence today. PCYACC is just one of the tools that is particularly useful in the syntax analysis phase of the language translation process.

The following picture illustrates a typical organizational strategy in which the various parts of a compiler are configured together to form an integrated translation program:



As shown in the diagram, there is a processing component for each translation phase. In addition, there are two more processing elements, a set of symbol table management routines and a set of error recovery routines. Although they do not correspond to any of the translation phases, these two components are extremely important for constructing compilers, since their services are needed by every other phase.

More detailed descriptions will be provided for each of the processing phases shown above, except for the intermediate code generation phase and the intermediate code optimization phase.

2. Lexical Analysis

The first processing phase of this multiphase architecture for compilers is the lexical analysis phase, which breaks up a source program into pieces, called tokens. Tokens are the smallest meaningful syntactical units in programming language constructs. As mentioned earlier, typical lexical units include keywords, numbers, identifiers and punctuations or delimiters.

The function of a lexical analyzer is best explained using a finite state machine, which is an abstract entity with a read-head and an input tape. The machine's movement is controlled by a set of internal states. The machine uses the read-head to scan the next symbol on the input tape, and consult its current state. It then makes the decision on what the next state should be. It changes its current state accordingly and moves its read-head one symbol to the right. To illustrate, let's take a simple example of lexical analysis for the small programming language INFIXEL, and construct a finite state machine.

INFIXEL contains following lexical units and definitions:

1). NUMBERs are sequences of one or more decimal digits;

2). IDs are sequences of one or more letters or digits, with the constraint that the first character must be a letter;

3). arithmetic operators '+', '-', '*' and '/';

4). delimiters '(' ')' and the newline character.

The finite state machine will have four states, START, IN_NUMBER, IN_ID and FINAL. In the pseudo-code description given below, a variable, state, will be used to hold the machine's current state. The function Read() will be used to read the next input symbol. The function UNREAD() will be used to give a symbol back to the input.

```
state = START
repeat
     next_symbol = read()
     case state of
          START:    case next_symbol of
               newline:  state = ACCEPT
               '+':      state = ACCEPT
               '-':      state = ACCEPT
               '*':      state = ACCEPT
               '/':      state = ACCEPT
               '(':      state = ACCEPT
               ')':      state = ACCEPT
               a digit:  state = IN_NUMBER
```

```
                a letter: state = IN_ID
                a space:  state = START
           IN_NUMBER: case next_symbol of
                newline:  state = ACCEPT
                '+':      state = ACCEPT; unread()
                '-':      state = ACCEPT; unread()
                '*':      state = ACCEPT; unread()
                '/':      state = ACCEPT; unread()
                '(':      state = ACCEPT; unread()
                ')':      state = ACCEPT; unread()
                a digit:  state = IN_NUMBER
                a letter: state = ACCEPT; unread()
                a space:  state = ACCEPT
           IN_ID:      case next_symbol of
                newline:  state = ACCEPT
                '+':      state = ACCEPT; unread()
                '-':      state = ACCEPT; unread()
                '*':      state = ACCEPT; unread()
                '/':      state = ACCEPT; unread()
                '(':      state = ACCEPT; unread()
                ')':      state = ACCEPT; unread()
                a digit:  state = IN_ID
                a letter: state = IN_ID
                a space:  state = ACCEPT
   until state == ACCEPT
```

Actually, this simple lexical analyzer can be implemented much more concisely than the description given in terms of the formalism of finite state machines (for example, see the function YYLEX() in the INTOPOST example).

3. Syntax Analysis

So far, we have discussed issues associated with the syntax processing phase, since PCYACC can be used to automate this process. A few other important concepts need to be reviewed at this point.

The formalisms useful at this stage are the context-free grammars and the family of languages they can be used to define. Syntactical aspects of contemporary programming languages are exclusively described using these formal tools.

Parser building techniques rely on grammatical specifications of programming languages. Top-down recursive decent parsing requires special treatment of grammar rules, such as eliminating left recursion and left factoring. It also requires that the first terminal symbol derived by each nonterminal must be unique. Not many context-free grammars can comply to this restriction. LR parsers, on the other hand, are more versatile because they cover a much wider a range of context-free grammars than recursive decent parsers.

4. Semantic Analysis

Semantic rules of programming languages are much harder to deal with than their syntactical counterparts. Formalisms for semantic processing that are comparable both in power and simplicity to context-free grammars for syntactical analysis are not available.

Semantic processing encompasses a range of guidelines that are incorporated into language translation facilities. Some of the most important guidelines are that identifiers cannot be used before they are defined, nor can they be multiply defined.

Research work done in the area of programming language semantics, seek ways of dealing with semantics rigorously and effectively. The three most important approaches are the operational-approach, the axiomatic-approach, and the denotational-approach.

This small example will illustrate how simple semantic constraints, like disallowing undefined or duplicate defined symbols, (which are also called static-semantics) can be checked and flagged by a language processor.

Consider the following language definition:

```
program      : decls stats
             ;
decls        :
             | decls decl
             ;
decl         : INTEGER IDENTIFIER ';'
             | REAL    IDENTIFIER ';'
             ;
stats        : stat
             | stats stat
             ;
stat         : IDENTIFIER '=' expr ';'
             ;
expr         : expr '+' expr
             | expr '-' expr
             | expr '*' expr•
             | expr '/' expr
             | '(' expr ')'
             | IDENTIFIER
             | FIXED_POINT_NUMBER
             | FLOAT_POINT_NUMBER
             ;
```

Programs in this languages consist of declarations and statements. Variables must be defined before they can be used. They can be defined as either type

INTEGER or type REAL, using the key words INTEGER and REAL respectively. Duplicate declarations are not allowed. The only kind of statements in this language is an assignment statement. Expressions are strongly typed, meaning all the operands appearing in an expression must be of the same type. The result of evaluating an expression is also the same type as each individual operand. Types of constants are determined from their representations. This means that a sequence of decimal digits representing a fixed-point number is type INTEGER, and a sequence of decimal digits with a decimal point representing a floating-point number is type REAL. Types of variables are determined from their declarations.

To deal with semantic issues, a symbol table is maintained to remember variable declarations. Each variable and its associated type is entered into the symbol table. To prevent duplicate declaration, a search is done on the symbol table before entering a new definition to see if the same symbol exists. The symbol table is also used to enforce the rule that every variable must be declared before it is used and that every symbol is properly implemented.

For example, consider the following C-like functions:

1). search(name) -- returns TRUE or FALSE depending on whether or not the name is found in the symbol table;

2). typeof(name) -- searched the symbol table, returns INTEGER, REAL, or UNDEFINED, depending on if the name has been declared as of type INTEGER, REAL or it is not found;

3). insert(name, type) -- insert a name and its type into the symbol table;

The necessary semantic checking can be incorporated into the language definition:

```
program      : decls stats
             ;

decls        :
             | decls decl
             ;

decl         : INTEGER IDENTIFIER ';'
               { if (search($2))
                   error("duplicate declaration");
                 else
                   insert($2, INTEGER);
               }
             | REAL    IDENTIFIER ';'
               { if (search($2))
```

```
                        error("duplicate declaration");
                      else
                        insert($2, REAL);
                    }
                  ;

stats             : stat
                     | stats stat
                      ;

stat              : IDENTIFIER '=' expr ';'
                    { if (typeof($1) == UNDEFINED)
                        error("undefined variable");
                      if (typeof($1) != expr.type)
                        error("type conflicts");
                    }
                  ;

expr         : expr '+' expr
                    { if (expr1.type != expr2.type)
                        error("type conflict");
                      else
                        expr0.type = expr1.type;
                    }
                  | expr '-' expr
                    { if (expr1.type != expr2.type)
                        error("type conflict");
                      else
                        expr0.type = expr1.type;
                    }
                  | expr '*' expr
                    { if (expr1.type != expr2.type)
                        error("type conflict");
                      else
                        expr0.type = expr1.type;
                    }
                  | expr '/' expr
                    { if (expr1.type != expr2.type)
                        error("type conflict");
                      else
                        expr0.type = expr1.type;
                    }
                  | '(' expr ')'
                    { expr0.type = expr1.type;
                    }
                  | IDENTIFIER
                    { if (typeof($1) == UNDEFINED)
                        error("undefined variable");
                      else
                        expr0.type = typeof($1);
                    }
                  | FIXED_POINT_NUMBER
                    { expr0.type = INTEGER;
```

```
  }
| FLOAT_POINT_NUMBER
  { expr0.type = REAL;
  }
;
```

Although the format of PCYACC grammar description programs was followed closely, expr0.type, expr1.type, etc. were used to refer to the types of nonterminal expr.

5. Code Generation

The code generation phase is typically machine dependent if low level object code is to be produced. To illustrate the basic ideas behind the code generation phase, assume that you have a stack machine with the following instruction set:

1). PUSHI <int> -- push an integer value onto the stack;

2). PUSHL <var> -- push the address of a variable onto the stack;

3). PUSHR <var> -- push the value of a variable onto the stack;

4). STORE -- store the contents of the top of stack to the address in the second to the top of the stack, and pop both off the stack;

5). ADD -- add the contents of the top of the stack to the contents of the second to the top of the stack, pop both off the stack, and push the result back onto the stack;

6). SUB -- subtract the content of the top of the stack from the content of the second tothe top of the stack, pop both off the stack, and push the result back onto the stack;

7). MUL -- multiply the contents of the second to the top of the stack by the contents of the top of the stack, pop both off the stack, and push the result back onto the stack;

8). DIV -- divide the content of the second to the top of the stack by the content of the top of the stack, pop both off the stack, and push the result back onto the stack.

Two functions are defined for writing object code to output like this object code file:

1). emit(code) -- write an opcode to the output;

2). emit2(code, oprnd) -- write an opcode and an operand to the output;

Now, a simplified skeleton for generating code for a typical assignment statement can be described as follows:

```
assign_stat : IDENTIFIER
              { emit2("PUSHL", $1); }
              '=' expr
              { emit("STORE"); }
            ;

expr        : expr '+' expr
              { emit("ADD"); }
            | expr '-' expr
              { emit("SUB"); }
            | expr '*' expr
              { emit("MUL"); }
            | expr '/' expr
              { emit("DIV"); }
            | '(' expr ')'
            | NUMBER
              { emit2("PUSHI", $1); }
            | IDENTIFIER
              { emit2("PUSHR", $1); }
            ;
```

Now, suppose you are given a two assignment statement program.

```
SUM = X1 + X2
PRO = X1 * X2
```

The following object code would be generated:

```
PUSHL SUM
PUSHR X1
PUSHR X2
ADD
STORE
PUSHL PRO
PUSHR X1
PUSHR X2
MUL
STORE
```

6. Symbol Table Management

Symbol table management is one of the most important tasks in building a compiler. This section will detail a set of routines that does symbol table management. This was discussed in a previous section which addressed semantic analysis issues.

Three routines were used in a previous discussion: INSERT(name, type), which inserts a symbol and its associated type into the symbol table, SEARCH(name), which checks if the symbol is in the symbol table, and TYPEOF(name), which finds the type of the symbol. However, before getting into details, a couple of preliminary decisions need to be made:

-- what kind of data structure to use
-- what search algorithm to use

1). Symbol table data structure

First, two constant definitions. NMSZ is the limit on the number of characters that are allowed in an identifier, and TNSZ is the limit on the number of identifiers allowed in a program.

```
#define NMSZ 32
#define TBSZ 257
```

You can use the following record structure to store a symbol:

```
struct sr {
     char name[NMSZ];
     int  type;
   };
```

The name field stores an identifier itself, and the type field stores the type of the identifier, encoded as an integer.

The symbol table is then a list of symbol records:

```
struct sr sytable[TBSZ];
int       sycount = 0;      // number of entries used
```

where sycount keeps track of the availability of symbol storage space.

2). Search algorithm:

Since searching is such a frequent activity in the language translation process, the hashing technique will be used in this example symbol table

management algorithm. The following two functions form the bases of the hashing algorithm:

```
001: int hash(name)
002: char *name;
003: { int hv, i;
004:
005:   hv = 0;
006:   for (i=0; i<NMSZ; i++) {
007:     hv = hv + name[i] << (i % 2);
008:   }
009:   return (hv % TBSZ);
010: }
011:
012: int probe(i)
013: int i;
014: {
015:   if (i >= TBSZ) return(0);
016:   else return (i++);
017: }
```

A folding accumulation strategy was used with this hashing algorithm and a simple linear probing method was used to resolve hashing conflicts.

Given an identifier, which is, a string of characters, the hashed value is found using following procedure. (The actual algorithm we used is slightly more efficient, though conceptually the same.) First, the character string is chopped into pieces, from left to right. Each piece has a uniform size of two characters (except the last one), or two bytes. Their sum is computed and used as the hashed value for the string.

The linear probing function returns the next index value of the hashing table (it cycles when it reaches TBSZ).

3). Symbol insertion:

The insertion routine takes an identifier and the type of the identifier and inserts them into the hashing table.

```
001: void
002: insert(name, type)
003: char *name;
004: int   type;
005: { int hv;
006:
007:   if (sycount >= TBSZ) {
008:     fprintf(stderr, "symbol table full\n");
009:     exit(1);
010:   }
011:   hv = hash(name);
```

```
012:   while (sytable[hv].type != UNDEFINED)
013:     hv = probe(hv);
014:   sytable[hv].type = type;
015:   strcpy(sytable[hv].name, name);
016:   sycount++;
017: }
```

It is assumed that the type entry of each symbol record in the symbol table has been initially set to UNDEFINED.

The algorithm first checks to see if the hashing table is full. If so, it complains and aborts. Otherwise, it computes the hashed value for the identifier and goes into a standard probing sequence to locate an available slot for the identifier. Then the identifier and its type is saved in the hashing table and the used counter is incremented.

4). Symbol lookup:

The symbol lookup routine is a Boolean valued function. It is given an identifier, and produces an answer of TRUE or FALSE, depending on whether the identifier is found in the hashing table.

```
001: int search(name)
002: char *name;
003: { int hv, firsthv;
004:
005:   hv = firsthv = hash(name);
006:   if (! strcmp(sytable[hv].name, name))
007:     return(TRUE);
008:   hv = probe(hv);
009:   while (hv != firsthv) {
010:     if (! strcmp(sytable[hv].name, name))
011:       return(TRUE);
012:     hv = probe(hv);
013:   }
014:   return(FALSE);
015: }
```

Note that in order for this search strategy to work properly, the probing sequence has to exhaust all other entries before it recycles. Its repetition cycle has to be the same as the hashing table size, which is certainly the case when using a linear probing strategy.

The algorithm first finds the hashed value for the identifier being searched, and checks to see if the identifier is in the hashing table. If not, it enters the probing loop until the identifier is located. In this case the search succeeds. Once the entire table is exhausted, the search fails.

5) Symbol type lookup:

This routine is basically the same as the one for pure symbol lookup, except it returns the type of symbols when they are found:

```
001: int typeof(name)
002: char *name;
003: { int hv, firsthv;
004:
005:   hv = firsthv = hash(name);
006:   if (! strcmp(sytable[hv].name, name))
007:     return(sytable[hv].type);
008:   hv = probe(hv);
009:   while (hv != firsthv) {
010:     if (! strcmp(sytable[hv].name, name))
011:       return(sytable[hv].type);
012:     hv = probe(hv);
013:   }
014:   return(UNDEFINED);
015: }
```

Note that the function returns a symbolic value UNDEFINED, in the case that the search fails.

7. Error Diagnostics

Providing meaningful and informative diagnostic messages when errors occur during language translation is one of the most valuable features of a compiler.

Basic error information should include the name of the source file and the line number where the error occurred.

In situations where there is only one source file, not much work is needed to maintain and display error information. For example, the lexical scanner can save the source file name and maintain a line count. The error processing routine will print out this information when producing error messages:

```
extern char *srcfile;
extern int   linecnt;

yyerror(s)
char *s;
{
   fprintf(stderr, "file \"%s\", line %d: %s\n",
     srcfile, linecnt, s);
}
```

Sometimes, it is desirable for the source file to be preprocessed before the parser sees it. Preprocessing can provide services like file inclusion, and/or macro expansion. In this case, there could be more than one source file involved in a compilation pass. Fortunately, most preprocessors supply source and location information in their output. For example, typical C preprocessors produce lines in the form:

 # line <linenumber> <filename>

In this case, it is reasonable to let the lexical analyzer detect the lines from its input stream and save it for the error handling routines to use. The following routine helps the scanner to do this:

```
001: extern char *locinfo;
002: extern char *srcfile;
003: extern int   linecnt;

004: mark()
005: {
006:   if (srcfile != NULL) free(srcfile);
007:   srcfile = (char *)
008:           malloc(strlen(locinfo) * sizeof(char));
009:   if (srcfile != NULL)
010:     sscanf(locinfo, "# line %d %s",
```

```
011:                        &linecnt, srcfile);
012: }
```

When the lexical analyzer detects that the next line is a line macro, it invokes the MARK() routine to save relevant information.

XIII. YAEC -- YET ANOTHER EXAMPLE COMPILER

In this Chapter, a slightly more complicated example of a simple picture specification language called PIC will be used to bring together the ideas discussed so far. PIC will be built as a simple load-and-go system, meaning object code programs are generated only in their internal forms. These internal programs are directly carried out by a built-in execution engine.

This Chapter is primarily made up of source code listings. It would be an ideal exercise for you to walk through the code listings. Also, there are many places that code can be improved.

1. Global Definition Head File

First, some global constant definitions and data type definitions are needed:

```
001: #define LISTSZ 127
002: #define PNTS 16
003: #define FATAL 1
004: #define NONFT 0
005: #define TRUE  1
006: #define FALSE 0
007:
008: typedef struct {
009:   int shape;
010:   int color;
011:   int style;
012:   int fill;
013:   int npoints;
014:   int x_coord[PNTS];
015:   int y_coord[PNTS];
016: } Object;
017:
018: typedef struct sc {
019:   char   *namep;
020:   Object *value;
021:   struct sc *next;
022: } Symbol;
023:
024: extern Object *objlst[], anObject;
025: extern Symbol *symlst[];
026: extern int ocount;
```

A geometric object (on the screen) is represented by a set of 2-dimensional points. The constant PNTS puts a limit on the number of points that can be used in specifying an object (16 in our case). Other characteristics of an object, in addition to the number of points it has, and the x-y coordinates of those points, are things like shape, color, line style and whether or not to fill a closed shape.

The symbol table is implemented using the hashing technique. Each symbol cell contains a name field, a pointer to its defined value, and a pointer to the next symbol cell, if any.

2. Lexical Token Definition Header File

This header file is actually generated using PCYACC with the -d option:

```
001: typedef union  {
002:   int   in;
003:   char *ch;
004: } YYSTYPE;
005: extern YYSTYPE yylval;
006: #define DRAW 257
007: #define DEFINE 258
008: #define LINE 259
009: #define BOX 260
010: #define POLYGON 261
011: #define CIRCLE 262
012: #define ELLIPSE 263
013: #define BLACK 264
014: #define WHITE 265
015: #define SOLID 266
016: #define DOTTED 267
017: #define FILL 268
018: #define IDENTIFIER 269
019: #define INTEGER 270
```

The type of the internal stack is defined to be a union of an int field and a pointer to a name.

3. Lexical Analysis Module

Since the lexical analyzer will use the symbol defined in YYTAB.H, generated by PCYACC with -d option, it should be included here:

```
001: #include <stdio.h>
002: #include <string.h>
003: #include <ctype.h>
004: #include "yytab.h"
005:
006: extern FILE *inf;
007: extern char *infn;
008: extern int   nxtch;
009:
010: static struct {
011:   char *kw;
012:   int   ltp;
013: } kwtable[] = {
014:   {"black",   BLACK},
015:   {"box",     BOX},
016:   {"circle",  CIRCLE},
017:   {"define",  DEFINE},
018:   {"dotted",  DOTTED},
019:   {"draw",    DRAW},
020:   {"ellipse", ELLIPSE},
021:   {"filled",  FILL},
022:   {"line",    LINE},
023:   {"polygon", POLYGON},
024:   {"solid",   SOLID},
025:   {"white",   WHITE},
026:   {"eot",     IDENTIFIER},
027: };
028:
029: kwsearch(s)
030: char *s;
031: { register i;
032:
033:   for (i=0; strcmp("eot", kwtable[i].kw); i++)
034:     if ( !strcmp(s, kwtable[i].kw) ) break;
035:   return(kwtable[i].ltp);
036: }
037:
038: #define POOLSZ 2048
039: char chpool[POOLSZ];
040: int  avail = 0;
041:
042: yylex() {
043: register int sign, tktyp;
044:
045:   while (nxtch==' ' || nxtch=='\t' || nxtch=='\n')
046:     nxtch = getc(inf);
```

```
047:   if (nxtch==EOF) return(0);
048:   if (isdigit(nxtch) || nxtch=='+' || nxtch=='-')
049:   {
050:     if (nxtch=='+') {
051:       sign = 1;
052:       yylval.in = 0;
053:     } else if (nxtch=='-') {
054:       sign = -1;
055:       yylval.in = 0;
056:     } else {
057:       sign = 1;
058:       yylval.in = nxtch - '0';
059:     }
060:     while (isdigit(nxtch=getc(inf)))
061:       yylval.in = (yylval.in * 10) + nxtch - '0';
062:     yylval.in = sign * yylval.in;
063:     tktyp = INTEGER;
064:   } else if (isalpha(nxtch)) {
065:     yylval.ch = chpool + avail;
066:     chpool[avail++] = nxtch;
067:     while (isalnum(nxtch=getc(inf)))
068:       chpool[avail++] = nxtch;
069:     chpool[avail++] = '\0';
070:     tktyp = kwsearch(yylval.ch);
071:   } else {
072:     tktyp = nxtch;
073:     nxtch = getc(inf);
074:   }
075:   return (tktyp);
076: }
```

The architecture of this lexical analyzer is typical of such programs. It first distinguishes number valued tokens from string valued tokens. When number valued tokens are found, their internal values are computed and passed to the parser. When string valued tokens are found, the keyword table is searched to distinguish keywords from user defined identifiers.

4. Syntactical Analysis Module

Here is the PCYACC grammar description file. From this grammar specification, you can see that a PIC program consists of zero or more statements, and a statement can be a define statement or a draw statement. A define statement associates a symbol with a simple graphical object, such as a line, a box, etc. A draw statement, on the other hand, puts a graphical object on the screen.

```
001: %{
002: #include "defs.h"
003: extern Object *new_object();
004: %}
005:
006: %union {
007:   int   in;
008:   char *ch;
009: }
010:
011: %token DRAW DEFINE
012:                      /* verbs */
013: %token LINE BOX POLYGON CIRCLE ELLIPSE
014:                      /* shapes */
015: %token BLACK WHITE SOLID DOTTED FILL
016:                      /* attributes */
017: %token <ch> IDENTIFIER
018: %token <in> INTEGER
019:
020: %start stats
021:
022: %%
023:
024: stats
025:   :
026:   | stats stat
027:   ;
028:
029: stat
030:   : draw_stat ';'
031:   | define_stat ';'
032:   ;
033:
034: draw_stat
035:   : DRAW IDENTIFIER
036:     { append_objlst(lookup($2)); }
037:   | DRAW object
038:     { append_objlst(new_object(&anObject)); }
039:   ;
040:
041: define_stat
042:   : DEFINE IDENTIFIER '=' object
043:     { install($2, new_object(&anObject)); }
```

```
044:   ;
045:
046: object
047:   : shape attrs '(' params ')'
048:   ;
049:
050: shape
051:   : LINE    { anObject.shape = LINE; }
052:   | BOX     { anObject.shape = BOX;  }
053:   | POLYGON { anObject.shape = POLYGON; }
054:   | CIRCLE  { anObject.shape = CIRCLE; }
055:   | ELLIPSE { anObject.shape = ELLIPSE; }
056:   ;
057:
058: attrs
059:   :
060:   | attrs attr
061:   ;
062:
063: attr
064:   : style
065:   | color
066:   | filling
067:   ;
068:
069: style
070:   : SOLID  { anObject.style = SOLID; }
071:   | DOTTED { anObject.style = DOTTED; }
072:   ;
073:
074: color
075:   : BLACK { anObject.color = BLACK; }
076:   | WHITE { anObject.color = WHITE; }
077:   ;
078:
079: filling
080:   : FILL BLACK { anObject.fill = BLACK; }
081:   | FILL WHITE { anObject.fill = WHITE; }
082:   ;
083:
084: params
085:   : point
086:   | params ',' point
087:   ;
088:
089: point
090:   : INTEGER INTEGER
091:     { anObject.x_coord[anObject.npoints] = $1;
092:       anObject.y_coord[anObject.npoints++] = $2;
093:     }
094:   ;
```

The grammar description program is certainly more complex than the previous examples. But, it is still small compared to an ordinary compiler project.

The program is fairly self-explanatory. With a little effort, you should be able to figure out what is going on.

5. Semantical Checking & Symbol Table Management Module

Symbol table management uses a hashing algorithm, similar with to the one discussed before, but not identical.

```
001: #include <stdio.h>
002: #include "defs.h"
003: #include "yytab.h"
004:
005: Object *objlst[LISTSZ], anObject;
006: Symbol *symlst[LISTSZ];
007: int     ocount=0;
008:
009: Object *
010: new_object(o)
011: Object *o;
012: { Object *p;
013:
014:   p = (Object *) malloc(sizeof(Object));
015:   if (p==NULL)
016:     exception(FATAL, "out of heap space");
017:   cpy_object(p, o);
018:   clr_object(o);
019:   return(p);
020: }
021:
022: cpy_object(t, f)
023: Object *t, *f;
024: { register int i;
025:
026:   t->shape = f->shape;
027:   t->color = f->color;
028:   t->style = f->style;
029:   t->fill  = f->fill;
030:   t->npoints = f->npoints;
031:   for (i=0; i<f->npoints; i++) {
032:     t->x_coord[i] = f->x_coord[i];
033:     t->y_coord[i] = f->y_coord[i];
034:   }
035: }
036:
```

```
037: clr_object(o)
038: Object *o;
039: { register int i;
040:
041:   o->shape = LINE;
042:   o->color = WHITE;
043:   o->style = SOLID;
044:   o->fill  = BLACK;
045:   o->npoints = 0;
046: }
047:
048: hash(s)
049: char *s;
050: { int hashval;
051:
052:   for (hashval=0; *s != '\0';) hashval += *s++;
053:   return(hashval % LISTSZ);
054: }
055:
056: Object *
057: lookup(s)
058: char *s;
059: { Symbol *sp;
060:
061:   for (sp=symlst[hash(s)]; sp!=NULL; sp=sp->next){
062:     if ( !strcmp(s, sp->namep) ) {
063:       return(sp->value);
064:     }
065:   }
066:   return(NULL);
067: }
068:
069: install(s, o)
070: char *s;
071: Object *o;
072: { Symbol *sp;
073:   Object *op;
074:   int hashval;
075:
076:   if ((op=lookup(s)) == NULL) {
077:     /* a new symbol definition */
078:     if ((sp=(Symbol *)
079:              malloc(sizeof(Symbol))) == NULL)
080:       exception(FATAL, "out of heap space");
081:     hashval = hash(s);
082:     sp->next = symlst[hashval];
083:     symlst[hashval] = sp;
084:   } else {
085:     /* symbol exists, override old definition */
086:     free(op);
087:   }
088:   sp->namep = s;
```

```
089:   sp->value = o;
090: }
091:
092: exception(f, m)
093: int f;
094: char *m;
095: {
096:
097:   fprintf(stderr, "Exception: %s\n", m);
098:   if (f==FATAL) exit(1);
099: }
100:
101: append_objlst(o)
102: Object *o;
103: {
104:   objlst[ocount++] = o;

105: }
```

6. Main Routine and Error Handling Module

This is a simple drive main and error processing routine. As an exercise, see if you can improve it.

```
001: #include <stdio.h>
002: #include "defs.h"
003:
004: FILE *fopen(), *inf;
005: char *infn;
006: int  nxtch;
007:
008: main(argc, argv)
009: int   argc;
010: char *argv[];
011: { int i;
012:
013:   if (argc != 2) {
014:     fprintf(stderr, "Usage: pic <file>\n");
015:     exit(1);
016:   }
017:   if ((inf=fopen((infn=argv[1]), "r")) == NULL) {
018:     fprintf(stderr,
019:       "Unable to open \"%s\"\n", infn);
020:     exit(1);
021:   }
022:   for (i=0; i<LISTSZ; symlst[i++]=NULL);
023:   clr_object(&anObject);
024:   nxtch = getc(inf);
025:   if (yyparse()) {
026:     fprintf(stderr,
027:       "Unsuccessful parsing of \"%s\"\n", infn);
028:     exit(1);
029:   }
030:   fclose(inf);
031:   picdraw();
032: }
033:
034: yyerror(s)
035: char *s;
036: {
037:   fprintf(stderr, "%s\n", s);
038: }
```

7. Execution Module

This is the execution engine of PIC. Note that it uses a lot of graphical routine calls that are provided in MicroSoft C++ version 1.5. This is for example purpose only the actual PCYACC disk set contains implementations for other compilers.

```
001: #include <stdio.h>
002: #include <graph.h> // MSVC 1.5 Graphics Library
003: #include "defs.h"
004: #include "yytab.h"
005:
006: #define GMODE _HRESBW
007: static char *gmname = "HRESBW";
008:
009: struct videoconfig vc;
010:
011: picdraw() {
012: int i;
013:
014:   if (! _setvideomode(GMODE))
015:     exception (FATAL,
016:       "nonsupported graphics mode");
017:   _getvideoconfig(&vc);
018:   _rectangle(_GBORDER, 0, 15,
019:       vc.numxpixels-1, vc.numypixels-1);
020:   _setlogorg(vc.numxpixels/2 - 1,
021:       vc.numypixels/2 - 1);
022:
023:   for (i=0; i<ocount; i++) {
024:     _setcolor((objlst[i]->color==BLACK) ?
025:         _BLACK : _WHITE);
026:     _setlinestyle((objlst[i]->style==SOLID) ?
027:         0xffff : 0xaaaa);
028:     switch (objlst[i]->shape) {
029:       case LINE: {
030:         draw_line(objlst[i]->npoints,
031:           objlst[i]->x_coord, objlst[i]->y_coord);
032:         break;
033:       }
034:       case POLYGON: {
035:         draw_polygon(objlst[i]->npoints,
036:           objlst[i]->x_coord, objlst[i]->y_coord);
037:         break;
038:       }
039:       case BOX: {
040:         draw_box(objlst[i]->npoints,
041:           objlst[i]->x_coord, objlst[i]->y_coord);
042:         break;
043:       }
044:       case CIRCLE: {
```

```
045:         draw_circle(objlst[i]->npoints,
046:           objlst[i]->x_coord, objlst[i]->y_coord);
047:         break;
048:       }
049:       case ELLIPSE: {
050:         draw_ellipse(objlst[i]->npoints,
051:           objlst[i]->x_coord, objlst[i]->y_coord);
052:         break;
053:       }
054:       default: {
055:       }
056:     }
057:   }
058:   fprintf(stdout,
059:     "press <ENTER> to clear screen and exit ...");
060:   getchar();
061:   _setvideomode(_DEFAULTMODE);
062: }
063:
064: draw_line(n, xs, ys)
065: int n;
066: int xs[], ys[];
067: { int i;
068:
069:   _moveto(xs[0], ys[0]);
070:   for (i=1; i<n; i++) _lineto(xs[i], ys[i]);
071: }
072:
073: draw_polygon(n, xs, ys)
074: int n;
075: int xs[], ys[];
076: { int i;
077:
078:   _moveto(xs[0], ys[0]);
079:   for (i=1; i<n; i++) _lineto(xs[i], ys[i]);
080:   _lineto(xs[0], ys[0]);
081: }
```

```
082:
083: draw_box(n, xs, ys)
084: int n;
085: int xs[], ys[];
086: {
087:   _rectangle(_GBORDER,
088:                  xs[0], ys[0], xs[1], ys[1]);
089: }
090:
091: draw_circle(n, xs, ys)
092: int n;
093: int xs[], ys[];
094: {
095:   _ellipse(_GBORDER, xs[0], ys[0], xs[1], ys[1]);
096: }
097:
098: draw_ellipse(n, xs, ys)
099: int n;
100: int xs[], ys[];
101: {
102:   _ellipse(_GBORDER, xs[0], ys[0], xs[1], ys[1]);
103: }
```

XIV. UNIQUE FEATURES OF PCYACC

PCYACC is a different implementation of the original UNIX YACC. There are several features considered very useful which PCYACC has incorporated that are not present in the original UNIX version. This Chapter discusses these unique features and provides suggestions on how to use them. Note: although a different implementation, PCYACC is completely upward compatible with UNIX YACC.

1. Quick Syntax Check Option

The command line option for doing a quick syntax check on grammar description programs is “-S”. This option is especially useful when working on very large grammars.

The syntax check option is invoked by the "-S" switch. For example, if you have just finished typing in a big grammar description file, BIGRAM.Y, and want to use this quick syntax check feature on it, you can use the following command:

PCYACC -S BIGRAM.Y<ENTER>

You don't even have to exit from your text editor to perform a syntax check if you are using (PWB - Microsoft Programmers WorkBench). Coupling an editor with the quick syntax check is an efficient and timesaving use of your resources. The initial debugging of the grammar file can be done without a lot of context switching.

2. Generating Parse Trees Using PCYACC

Another unique feature is that PCYACC can help you to generate parse trees.

Parse trees are representations only for the syntactic recognition processes. Still, they are very useful tools, especially for debugging grammar rules.

Consider the following scenario. You have completed the full cycle of a compiler (say, for language L) development project, and wrote a program in L. When you try to compile the L program with the compiler you developed, it complains about syntax errors in the L program. At this point, you can't tell which is faulty; the L compiler or the L program.

One way to isolate the problem is to look at the parse tree for the L program. If all the reduce actions are correct, then the problem is with the L program. Otherwise, you need to go back to the L compiler to do some more work.

Even if you are quite sure that the L compiler is faulty, you still need information about where the problem has occurred. In this case, the parse tree for the L program can be very helpful in locating the bug.

A -t command line switch is provided to help you out. The function of the -t switch is to add hooks to the parser (for language L) generated by PCYACC, so that when the parser is invoked on a program written in L, a syntax tree in textual form is produced to a file YY.AST. Use of the “-t” switch is not available when the “-p” switch is in operation, unless of course support has been added to the external skeleton parser..

The textual form of a syntax tree consists of lines. Each line is a grammar rule, corresponding to a reduce operation performed on the source program.

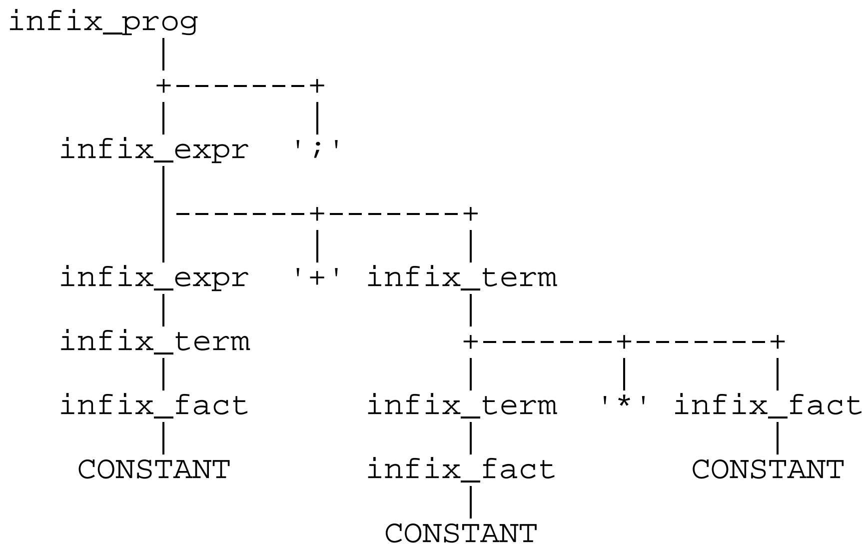
Consider the following infixel program with a single arithmetic expression:

```
2 + 3 * 5;
```

The following is the content of the YY.AST file generated for the infixel program:

```
infix_prog infix_expr  ;
infix_expr infix_expr  + infix_term
infix_term infix_term  * infix_fact
infix_fact CONSTANT
infix_term infix_fact
infix_fact CONSTANT
infix_expr infix_term
infix_term infix_fact
infix_fact CONSTANT
```

From the textual form of the parse tree, the following graphical representation can easily be constructed:



3. Supporting Multiple Parsers.

Multiple parsers are required where more than one yyparse() routine is required within one executable. This topic has been placed in the PCYACC specific section of the manual because this technique is generally not portable. The general solution to the problem is to generate a second parser where the yyparse() routine and tables do not conflict at link time with duplication messages. There are two solutions to this problem and the first involves using the alternate parser skeleton (-p option). This method is to simply generate a parser skeleton with an entry point other than yyparse(), and make the tables local to the new parser (if your compiler supports "static local"). A second method is to build parser executables and call them from your main executable, in the form system("parsera"); but again this solution is not portable. Many YACC implementations support external skeletons so this may be your best bet if portablility is your goal.

The multiple parser solution can also be solved at the grammar level where if the second language is not too complex both grammars can be combined into one. This is only mentioned because combining two large grammars is quite often a very difficult problem, and a debugging nightmare to boot.

See \MULPAR example on PCYACC PROGRAM DISK, call or Email Abraxas today and ask for a free PCYACC UPDATE DISK if you can't find MULPAR.

When using C++ and multiple yyparse() entries are required then the PCYACC Object Oriented Toolkit should be used.

4. Lexical Analysis Caveats - Combining Lex & Yacc

Lex by definition is a table driven state machine, e.g. pattern matching defined by regular expressions activate blocks of C or C++ code. Languages like C are not ambiguous, i.e. every token has an exact meaning - int, char, float. However C++ supports the concept of over-loading which means many tokens are actually context sensitive, in fact your lexical analyzer may even require multiple passes for languages like C++. These problems are not unique to PCLEX, but need to be mentioned.

Most PCYACC examples use yylex() subroutines written explicitly in C or C++. Parsing complex languages like C++ and Fortran90 is difficult when working with lexical analyzer generators like lex (because lex by its own nature has dumb lookahead).

Fortran Example:

```
DO I = 10      // This example is a DO assignment

DO I = 1, 10   // Here we have a DO loop
```

If the yylex() is written in C/C++, the Fortran solution is straight forward, if you see a "DO" look ahead and determine context and return a meta-token of either **DO_ASSIGN** or **DO_LOOP**. In the case of LEX, you must develop a recursive descent handler that triggers on "DO", looks ahead into the buffer and determines context, and then returns the LEX state machine to its original buffer pointer (*DO + 2), this is not always trivial because LEX uses multiple cyclic buffers. What we are trying to say is that LEX works best when treated as a "black box", when you have go into the internals of LEX to get your job done you may be using the wrong tool.

LEX has its place in rapid proto-typing or very simple languages like C or Pascal. Parsing languages is a very difficult problem, lexical analysis is a very simple problem we believe that source code generators should be used to solve the seemingly unsolvable problems and the simple problems be left to hand coding. This is an Abraxas philosophy. If you want to keep your project simple and get it done, write your own lexical analyzer and use PCYACC to generate your parser. It must be remembered that lex and yacc are not married, they were invented by two different people with different problems. Please don't assume that all your compiler problems need these two tools together. Always use the appropriate tool for the job. There are many successful products on the market that use PCLEX and PCYACC in a standalone manner.

XV. ERROR PROCESSING WITH PCYACC

Runtime examples for this chapter can be found on the PCYACC distribution disk entitled PCYACC ERROR PROCESSING DISK.

Syntax error processing breaks naturally into three parts: reporting (tell the user what went wrong), handling (how to fix up the input to continue parsing), and recovery (how to fix up the output or internal data structures). Completely automated error reporting is included in PCYACC. PCYACC has a good, general mechanism for error handling that allows tailoring for your specific application. Error recovery is specific to each application--some guidelines and examples will be given and some of the pitfalls pointed out.

Section 1 of this manual describes how to use the error reporting code in ERRORLIB.C and ERRORLIB.H. Section 1.2 is a reference manual for the error reporting functions in ERRORLIB.C. All parsers on this disk use the error reporting code. In \INTOPOST are three versions of the infix to postfix translator described in Chapter VII of the PCYACC manual. They illustrate how to use the supplied error reporting code and several ways to use the error handling mechanism. They are explained in detail in section 2 of this manual. In \ANSIC is an ANSI C syntax analyzer with error processing added. It ignores preprocessor lines and semantics, has a full lexical scanner, and has basic error handling. It is briefly explained in section 2.4 and is a more extensive example of the techniques outlined earlier in section 2. In \PIC is a version of the processor for the PIC graphic description language with extensive error handling and recovery. It is explained in section 3.

Section 4 describes how to build (yacc, compile, and link) parsers and is a reference manual for the TOKENS.EXE program. All programs on this disk have MAKEFILEs for additional examples on how to build a parser.

1. Error Reporting

PCYACC's error reporting mechanism reports the error number, type ("syntax error" or "stack overflow"), where the error was detected, the erroneous token's type and the expected/allowed token types. For example:

```
[error 1] file 'errors', line 1: syntax error
actual: '&' expecting: '+', '-', ';'
[error 2] file 'errors', line 3: syntax error
actual: ';' expecting: ')', '+', '-'
```

```
        [error 3] file 'errors', line 6 near "55": syntax
error
        actual: CONSTANT expecting: '+', '-', ';'
        [error 4] file 'errors', line 8 near "v": syntax error
        actual: VARIABLE expecting: '+', '-', ';'
```

The actual versus expected display in the second line of the error message prints the token that caused the error and the tokens that are acceptable at this point in the parse. These include "[end of file]", single characters (e.g., '+', '-', or ';'), or any of the terminals declared "%token" in the declaration section.

1.1. Integration with a Lexical Scanner

When you use the supplied error reporting code, "errorlib.c", in your program, your lexical scanner needs to pass information about the location of the current token to the "errorlib" routines. The integer variable "yylineno" should be set non-negative before calling "yyparse" and incremented as each new input line is read. This usually means setting it to 1 in the main program and incrementing it as each newline ('\n')
character is processed by "yylex". If your lexical analyzer uses a lookahead character, initialize it to '\n' and set "yylineno" to 0 in the main program. The input file name should be copied into the character array "yyerrsrc".
Error messages are written to the "yyerrfile" stream. The FILE pointer "yyerrfile" is initially set to "stderr". You may assign another FILE pointer to it to redirect error messages. The character pointer "yyerrtok" should be either "NULL" or point to a NUL-terminated character string containing the current input token text. Look at "yylex" in "SIMPLE.Y" for an example. All of these variables are initialized so that if your code doesn't set them, they won't show up in the error messages.

1.2. YYERROR Calling Conventions

If you wish to use the error reporting code in "errorlib.c" for other errors, read this section. Otherwise, skip it. The prototype for the "yyerror" routine called by the "yyparse" in "yaccpar.c" is:

```
extern void yyerror(char *msg, char *token);
```

When a syntax error is detected, "yyparse" calls "yyerror" with the error message ("syntax error") and a string containing the name of the actual token. It then repeatedly calls "yyerror" with NULL and a string containing the name of an expected token. Finally it calls "yyerror" with two NULL pointers to reset it.

For the simpler case of just an error message with no actual and expected tokens, like a stack overflow, "yyparse" calls "yyerror" with a message string and a NULL pointer. Note that a NULL pointer in the second argument always indicates the end of the error message.

The token number to string conversion routine, "yydisplay" is available for use. It takes a single integer argument, a token as returned by "yylex", and returns a pointer to the token's name in a NULL-terminated string. Do not depend on the string value surviving past another call to "yydisplay".

2. Error Handling

The simplest error handling mechanism is none, i.e., quit after reporting the first error. Except for simple, line-at-a-time interactive applications, this is unacceptable. More sophisticated error handling involves changing the input in the vicinity of the error. PCYACC uses a combination of deletion and insertion.

To control the error handling process, additional rules are inserted in the grammar. These error rules are ignored during normal parsing. Each error rule has the "error" keyword somewhere on the right side. When a syntax error occurs, the parser stack is popped until a state with a shift on "error" is found. The "error" token is inserted ahead of the token that caused the error in the input stream (the lookahead token) and parsing continues in the recovery mode. The parser will remain in the recovery mode
until 3 consecutive tokens are read and shifted without error. Syntax errors in recovery mode are not reported. Any tokens read between syntax errors in recovery mode are discarded.

Popping the parse stack effectively deletes tokens to left of the error detection point from the parse. Zero or more tokens to the right of the error are deleted from the parse during error handling.

If popping the parse stack never finds a state with a shift on "error", the parser aborts ("yyparse" returns with a non-zero value). Encountering EOF in the recovery mode always aborts the parser. If the parser is going to be robust and continue trying in the face of multiple errors, there must a state with a shift on "error" low on the stack. The initial state, State 0, is always the bottom state on the parse stack. For this reason, it is a good idea to have an "error" rule for the start symbol of the grammar.

2.1. Simple Recovery

For a unstructured, statement oriented language, skipping to the end of the statement on an error is a good strategy. The file "\INTOPOST\SIMPLE.Y" illustrates this technique. This program is the infix to postfix translator ("\INTOPOST\INTOPOST.Y" on the program disk) with two additional rules:

```
infix_prog : error ';'           { yyerrok; }
   | infix_prog error ';'        { yyerrok; }
            ;
```

These two rules ensure that there is always a state with a shift on "error" somewhere on the stack. Normally the parser would not get out of the recovery mode until parsing the ';' and the two tokens after that. The semantic action "yyerrok" says in effect: I know what's going on, it's okay to resume real parsing so leave the recovery mode and let me know if there are any syntax errors from here on. This technique is easy to apply and works well. The one syntax error detected per statement limit is reasonable for short statements and meshes well with most users' expectations.

The infix to postfix translator is simple enough to do error recovery by discarding output from a bad expression. At initialization and at the end of each complete expression, the position of the output file is saved in "start". If a syntax error is detected, the output file is backed up to the expression "start", discarding any partial postfix expression. Translation resumes with the next expression. Look at "\INTOPOST\SIMPLE.Y" for the details.

2.2. Improved Recovery

Detecting more than one syntax error per expression requires more error rules. For your language, determine which tokens are significant and are unlikely to be misspelled or mistyped. Most are either terminators, infix operators, or grouping markers. Terminators mark the end of significant pieces of the input. Examples are the ';' at the end of statements and the ',' in argument lists in C and Pascal. Add a rule or two with the error token just before the terminator. The error rules added to "SIMPLE.Y" are a good example of this case. Infix operators appear between their operands (like =, %, /, and || in C). The C binary operators and the tertiary conditional operator (e.g., "(a == 0) ? b : c") are infix operators. For binary operators, add one rule like the rule(s) for the lowest precedence operator with the operator replaced by "error". In the infix grammar, addition and subtraction are the lowest precedence operators and the following rule is added:

```
infix_expr : infix_expr error infix_term
           ;
```

Using the lowest precedence gives the widest coverage for error handling and gives the surrounding operators higher precedence, which is usually the users' understanding of the error correction.

For grouping markers, add an error rule to the construct that goes inside the markers. For example, in C:

```
compound_statement : '{' statement_list '}'
                   ;
statement_list : statement
          | statement_list statement
               ;
statement : labeled_statement
   | basic_statement
          | compound_statement
          | error ';'
          { yyerrok; } ;
```

This rule can be generalized to: add an error rule for the middle of any construct with three or more tokens or non-terminals, if the first symbol is unique to the context. The error rule can be down a level or two as in the example above or in the rule itself. For example:

```
declaration : declaration_specifiers error ';' { yyerrok; }
            ;
```

If the last symbol is a token and is unlikely to be mistyped, add the "yyerrok;" semantic action, otherwise let the 3 successfully shifted tokens rule apply. In the \ANSIC subdirectory is a grammar for ANSI C with error rules added according to these heuristics (see section 2.4 for more on the ANSI C syntax analyzer).

These strategies are adapted from strategies and tactics outlined by Axel Schreiner in his book listed at the end of this file. He outlines some techniques and notes that they don't work with several of the most common YACC implementations. PCYACC is one of them. The above rules do work with PCYACC and can be generalized to more complex constructs.

The exact effect of error handling is very tied to the parsing strategy. Three good rules of thumb are: 1) when "error" appears just before a terminal, input will be skipped up to that terminal, 2) when "error" appears just before a non-terminal, a dummy token with no meaning will be inserted, and 3) when "error" appears at the end of a rule, the token it substitutes for is inserted. The second and third rules have implications for error recovery that are outlined below. In the "improved" grammar, it is possible for several states with a shift on "error" to be on the stack at any given time. Remember that the top one counts. For example, if an error is detected at the beginning of a statement (i.e.,

at the beginning of the file or just after a ';'), rule 1 applies and input is skipped up to the next ';' (or EOF). If an error is detected within an expression and outside of parentheses, rule 2 applies and the inserted "error" token is effectively an addition operator. Inside parentheses, an "error" token is inserted and the lookahead token (the one that caused the error) will be examined, if it can be the start of an expression (a CONSTANT, IDENTIFIER, or left parenthesis) then the inserted "error" token is again a pseudo addition operator. Otherwise, it is treated like a closing parenthesis.

2.3. Doing Your Own Parser Recovery

If you would rather do your own error handling, hooks are provided in PCYACC to do so. In the simplest usage, just add a rule that says the start symbol of the grammar can be an "error". This adds a shift on "error" to the state on the bottom of the parse stack so the action that does your recovery will always be called. For example:

```
infix_prog : error     { recover();  yyerrok;  yyclearin; }
           ;
```

You write the "recover" function. It should advance the input stream to a point where parsing can continue (for example, the start of an infix expression). The "yyerrok" action kicks the parser out of the recovery mode as in previous parsers. The "yyclearin" action empties the lookahead token. The lookahead token is the last one read by the parser. In the case above, it is the token that caused the error. For a rule with the "error" in the final position, it is either the last terminal of the rule or the first terminal after the rule, if lookahead is needed to determine which parsing action to take. Examine the ".lrt" file if the difference is critical.

The obvious way to write "recover" for the infix to postfix translator is:

```
recover()
{
     int t;

     while ((t = yylex()) != ';' && t != 0) // EOF == 0
          ;
     fseek(outf, start, SEEK_SET);
}
```

This works but occasionally discards too much input. For an infix expression like "(1 + 1;" with an error detected at the ';', the above code will start discarding input with the token after the ';' that the parser has already read into the lookahead token and the entire next expression will be discarded. The parser's

lookahead token is in "int token;". Rewriting "recover" to use this information produces:

```
        recover()
        {
    while (token != ';' && token != 0)       /* EOF == 0 */
                  token = yylex();
             fseek(outf, start, SEEK_SET);
        }
```

This change makes the ``Do It Yourself'' error processing version behave the same as the "SIMPLE.Y" translator. Error recovery is also the same. The complete source code is in "DIY.Y".

2.4. The ANSI C Parser

The error rules in the ANSI C syntax analyzer in \ANSIC are a straight forward implementation of the heuristics given in section 2.2. The start symbol is "translation_unit" and an error rule has been added to it to guarantee that a shift on "error" state is always somewhere on the parse stack. Two rules are added to "function_definition" to handle missing or incorrect text in the middle of a function header. Error rules for "declaration" handle an incorrect "declaration_specifiers" and an incorrect "init_declarator_list". As a catch-all for incorrect structure declarations, an error rule is added to "struct_declaration" to discard all text through the terminating semicolon and then exit recovery mode. The same is true for "statement".

Semicolons are presumed to be correct and in the examples above rocovery mode is terminated and normal parsing begins again. For the other error rules, the three correct tokens rule is used to quit recovery mode.

The lists within grouping markers are: "enumerated_list" and "identifier_list". The former is enclosed in braces and the error rule is added to "enumerator". The latter is enclosed in parentheses and the error rule is added to "identifier_list" directly.

In C, the lowest precedence operator in expressions is the comma operator. The error rule for "expression" catches all missing or incorrect operators in expressions. The rule of three is used to exit recovery mode. Badly garbled expressions can lead any error handling strategy astray. The rule of three reduces the number of cascading error messages (multiple messages from one error).

The ANSI C grammar has examples of all of the error rule heuristics given previously.

3. Error Recovery

Error recovery (repairing the output of the parser) is similar to error handling. The simplest is none, i.e., don't generate any output if a syntax error occurs. This is often acceptable. Continued type checking in strongly typed languages and other areas where syntax and semantics overlap requires some continued semantic processing. Other solutions for error recovery are: back out the incorrect actions to some safe state, or correct the semantics, making safe or neutral assumptions about the user's intent (e.g., C compilers assume that an undeclared variable is either an "int" or some probable type based on the context of its first use). The simple and do-it-yourself programs in the previous section illustrated a little error recovery. The error recovery in the improved program is described in section 3.1. The PIC program in the \PIC directory has more extensive error recovery. It extensively illustrates the principles of good recovery. It is described in section 3.2.

3.1. Error Recovery in IMPROVED.Y

The "improved" grammar attempts to always output a valid postfix expression. For errors detected at the start of an expression, nothing has been output for the expression so no fix up is needed. For the error rule

```
infix_expr : infix_expr error infix_term
           ;
```

two operands have been recognized and something needs to be done to combine both values into one; addition was arbitrarily picked and a '+' is output. Without the addition, the end of the expression would be reached with two values still on the stack of the postfix evaluator. Parentheses in infix expressions have no semantics and so no fix up is done for the error rules of "infix_fact". Error recovery needs to track error handling. It is still possible for the "improved" infix to postfix translator to output incorrect postfix but for widely spaced errors, the output will at least be valid even if not what the user intended.

3.2. Error Recovery in PIC

PIC is a simple graphics description language. A program to run PIC on a CGA video system is included in the \PIC directory on the program disk and is described in Chapter XIII of the manual. In the \PIC directory on this disk is a version with extensive error recovery added. Read Chapter XIII and then this section. The differences between the two versions are limited to the error recovery changes so FC or any other file comparison utility will pinpoint the changes.

An error rule is added to the grammar for "stats" to guarantee a shift on "error" in the state on the bottom of the stack (State 1 in this case). This last chance error recovery simply discards very badly formed statements.

Missing DRAW and DEFINE keywords, missing equals signs, and missing semicolons are repaired by the rules added to "draw_stat" and "define_stat". The keywords and terminals are redundant, some can be missing without losing any important information. The rules show how the statements can be garbled and the necessary information still retrieved. The 6th and 7th rules for "define_stat" cover a missing IDENTIFIER. Parsing of the Object can continue, but there is no name for the object and it is discarded.

After each Object is used, the values of "anObject" are re-initialized to default values (by "clr_object()" in YYSUB.C). These values are used when a shape or attribute is not specified, is incorrectly specified, or its value discarded during error recovery. The default values are the most likely values (e.g., white borders, black fill), the most general (e.g., polygon), or some other safe value. Error rules are added for missing or incorrect shapes, attributes, fill colors, and commas to permit error handling.

Incorrect points are handled slightly differently. Completely wrong points are discarded. A single INTEGER is assumed to be the X coordinate and -32768 is used for the Y coordinate. When an Object is copied from anObject, the unspecified points are set to (-32768,-32768). No check is made that the number of points for a shape is correct. Due to the way signed integers are represented, there is no +32768 and so the value -32768 is impossible to get through the lexical scanner. This impossible value is used to signal a missing value.

The error rules in PIC.Y are extensive enough that all states with shifts have shifts on "error", except those only accessible in recovery mode. The stack never needs to be popped to uncover a state with a shift on "error", so correctly parsed input is never discarded by error handling. If your semantic actions use the semantic stack to pass dynamically allocated memory pointers about, this behavior is very useful to allow de-allocating the memory when errors render the information it carries useless. To get this behavior, you may need to add error rules in certain areas of the grammar that are not otherwise needed for good error handling.

Good error handling is a requirement for good error recovery. Global variables used by the semantic actions must be initialized before parsing begins and re-initialized to safe (e.g., 0, white borders, black fill) or significant values (e.g., -32768, or other "impossible" or unlikely values) at good breaking points (e.g., end of statement). The intention is that all variables have well defined values at all times so error do not produce erratic behavior. Dynamically

allocated memory must be tracked carefully to prevent repeated loss of significant pieces, especially ones that are expected to have only limited scope or lifetime (e.g., C local variables). This may necessitate additional error rules that parsing alone does not require. Sufficient redundancy in a construct may allow it's semantics to survive missing or incorrect parts. Good error recovery involves extensive defaulting or initialization, good repair of missing information, and robust post-parse handling of possibly incomplete information.

4. Building Parsers

Translating parser source code to an executable image involves PCYACC, TOKENS.EXE, a C compiler, and the linker. TOKENS converts the C header file output by PCYACC into a list of token names in quotes for the error reporting in "ERRORLIB.C". For the simplest case of a parser entirely in one file, for example, "HELLO.Y": 1) copy YACCPAR.C from \INTOPOST on this disk to your working directory, 2) copy TOKENS.EXE from \TOKENS to a directory in the PATH list (e.g., \BIN), 3) copy ERRORLIB.* from \INTOPOST to your working directory, and 4) YACC, tokenize, compile, and link like this:

```
pcyacc -d -pyaccpar.c hello.y
tokens
cl hello.c errorlib.c
```

This example assumes that "cl" is your C compiler. A MAKE utility makes builds less tedious. The MAKEFILEs on this disk assume Microsoft's C compiler and have been tested with Microsoft's NMAKE utility. The rest of this section describes TOKENS in more detail.

4.1. TOKENS program

The TOKENS program reads the C header file (also called the Token Definition file) produced by PCYACC and writes a file to be included in "errorlib.c", the advanced error reporting code. The C header file is created when PCYACC is run with either the -d or -D option (see Chapter III, sections 2 and 3.2). The C header file name is "yytab.h" with the -d option, the base name of the grammar description file with an extension ".h" with the -D option, and any name you choose with the -D<hf> option.

TOKENS is invoked by simply typing TOKENS from the MSDOS command prompt. This chapter explains the command line format and the possible options.

4.2. Command Line Format

TOKENS can be invoked by typing TOKENS, followed by zero or more command line options, followed by an optional file name. For example:

TOKENS [options] [<filename>]

<filename> is the name of the C header file produced by PCYACC. If no extension is given and <filename> cannot be found, then <filename> with the extension ".h" is tried. If no <filename> is specified, "yytab.h" is used.

4.3. Command Line Options

Command line options are used to override default actions or file name conventions. Available options are described below:

-b: Change all underscores in token names to blanks. For example, "arith_exp" becomes "arith exp".

-c: Force the first letter of each token to upper case, the rest to

lower case. For example, "aNycaSe" becomes "Anycase".

-l: Force all letters of each token to lower case.

-o<tf>: Output is to <tf> instead of the default "yytok.h".

-u: Force all letters of each token to upper case.
The option letter's case does not matter, i.e., -b and -B mean the same.

4.4. Using Command Line Options

This section shows you how to use the command line options. The following example, HELLO.Y, is used throughout this section:

```
%{
#include "errorlib.h"
%}

%token WORLD
%token HELLO

%%

greetings : HELLO ',' WORLD ;
```

To build a parser from HELLO.Y using the default actions of PCYACC and TOKENS as much as possible, do the following actions (assuming CL is the C compiler):

```
pcyacc -d -pyaccpar.c hello.y
tokens
cl hello.c errorlib.c
```

Another possibility is using the grammar base name for the C header file. In this example, the token names used in error messages are capitalized:

```
pcyacc -D -pyaccpar.c hello.y
tokens -c hello
cl hello.c errorlib.c
```

Normally, TOKENS writes to "yytok.h". The supplied "errorlib.c" code includes "yytok.h". If you change this file name to something else, for example "hello.tok", run the following procedure to build your parser:

```
pcyacc -pyaccpar.c -D hello.y
tokens -ohello.tok hello
cl hello.c errorlib.c
```

The TOKENS program converts the C header file that defines the tokens that the parser expects from the lexical scanner to a printable form for the improved error reporting.

5. Wrapup

Advanced error processing involves first: reporting the error and its location in the input with the errorlib routines. Syntax error information is automatically passed by the parser as generated by PCYACC. Any location information is passed by the lexical scanner. Second: error handling depends on error rules you put in the grammar. The heuristics given cover the common situations and can be extended to almost any grammar. Where error rules and the built-in error recovery mechanism are not adequate, hooks are provided for you to roll your own error handling. Third: error recovery involves keeping all internal data structures in a useable state even when the information in the input is incomplete or invalid.

XVI. USING PCYACC WITH C++ AND MICROSOFT WINDOWS

Windows/C++ PIC notes and comments: Examples for this chapter can be found on PCYACC PROGRAM disk in the directory entitled \WIN_CPP, there are Borland OWL and Microsoft MFC / Visual C++ examples. (In this case we will use the BORLAND OWL example). This example is useful for building Windows Application and Components using PCYACC.

1. The Borland C++ project demonstrates the use of a TParser class with PCYACC. The provided parser includes syntax error reporting methods that substantially improve upon normal PCYACC error messages.

2. The Windows PIC application program runs under MS Windows 3.x or later. The actual PCYACC distribution set contains Windows 95/NT examples of this problem. PIC for Windows cannot be executed from the MSDOS command line.

3. Required for successful compilation: Borland C++ 4.0 with OWL 2.0 (the Object Windows Library) and the Windows Resource Compiler.

4. The central element of this package is the file TPARSER.H, which contains the C++ interface for the parser class. The file PARSER.H is designed for use with this interface. To integrate this parser into your grammar, use a command line like this:

```
pcyacc -pParser.h -Dpic.h -Cgrammar.cpp -v grammar.y
```

5. The error reporting methods for TParser and the lexical analyzer for PIC files is found in the file SCAN.CPP. For your own application you will need to write your own version of TParser::yylex().

6. TCANVAS.CPP and PICAPP.CPP are OWL specific files (interface and display methods). Additional methods for PIC processing are in the parser subdirectory.

7. To play with this project using the Borland IDE (integrated development environment), start up the IDE from within Windows. To create PIC.EXE from MSDOS command prompt, use this command line: make -fpic.mak -B
8. The file ERROR.PIC demonstrates the behavior of the parser when a syntax error is encountered.

XVII. PCYACC AND PCLEX RECURSION AND RE-ENTRANCY

We assume that the second call to yyparse() is not interaction with the first call, if it is refer to the MULPAR example in this manuals index. The problem with recursion is because the input buffer for the lexical analyzer has not been cleaned up after the first yyparse() call. Here what you need to do is to take care of the buffer your self as shown below;

1. Do this modification on the code generated by PCLEX, change the code as follows:

```
case YY_END_TOK:
  return (YY_END_TOK);
```

```
to
```

```
case YY_END_TOK:
  YY_INIT;
  return (YY_END_TOK);
```

this makes sure that once the lexical analyzer has reached the EOF, it cleans up the buffer. However, this only works for the case yyparse() exit normally. In case a syntax error occurs, and the EOF will never reach yylex(). So to deal this case we do the following:

2. declare a global variable "int reinit" in the lexer file generated by PCLEX, and at the very beginning of function body of yylex(), add the lines

```
if (reinit)
{
  yy_init=1;
  reinit=0;
}
```

3. Now we do the work on parser code, declare reinit as external in the file generated by PCYACC, and in the body of function yyparse(), find every statement "return 1", this statement marks the error exit, so you just make it to:

```
reinit=1;
return 1;
```

to make sure once the error exit of yyparse() happens, it will remind yylex() to do the initializaion.

With these changes, your parser should be able to be called as many times as you like. What we have described above is do the change in the generated code. However, this will be tedious each time you recompile .l and .y files. You can choose to make the changes on skeleton files for both PCLEX and PCYACC, which we you can find on distribution disk (see \src directory on pcyacc and pclex program disks). And once the changes are done, you use option -P on both PCLEX and PCYACC with your new skeleton files. So you can avoid the hand work each time you are generating code.

XVIII. PCYACC CROSS REFERENCING WITH - YACC TOOL

This chapter describes how to use YACC TOOL briefly. This tool is written in C. The project file (makefile) is provided for Microsoft C 1.5 and later.

YACC TOOL is provided as executable program called ytool.exe, and C source is included. The program can be found in the \YTOOL directory on the PCYACC PROGRAM DISK. The source files used to build ytool.exe include the following files:

ytool.c
token.c
param.h

YTOOL GRAM.Y [A-H], choose a letter after the grammar filename to select option, no '-' is necessary. YACC TOOL offers you the following functions:

a. Create GRAMMAR FOREST from ytool source file
b. Create GRAMMAR FOREST with rule & line number (1)
c. Create GRAMMAR FOREST with rule & line number (2)
d. Create GRAMMAR FOREST with rule & line number (3)
e. Cross Reference Table of the rule in the rule appearing order
f. Cross Reference Table of the rule in the order of alpha
g. Cross Reference Table of the rule in nonterminal appearing order
h. Grammar Description File with line number

The results of these functions is shown as follows:

1. Create GRAMMAR FOREST from YTOOL.EXE

Command line: **ytool cpp.y a** >**temp**, this example strips out the C source and outputs a pure yacc grammar to stdout, which is piped to temp.

The content of temp is:

```
------------------------------------------------------------
                       Grammar Forest
------------------------------------------------------------
program          : .program
                 | .program extdefs
;
.program         :
;
extdefs          : extdef
                 | extdefs extdef
;
extdef           : fndef
                 | datadef
                 | overloaddef
                 | ASM '(' string ')' ';'
                 | extern_lang_string '{' extdefs '}'
                 | extern_lang_string '{' '}'
                 | extern_lang_string fndef
                 | extern_lang_string datadef
;
extern_lang_string: EXTERN_LANG_STRING
;
overloaddef      : OVERLOAD ov_identifiers ';'
;
```

XIX. Invoking pcYaccDeBugger - YDB

This program is a visual debugger for debugging pcyacc grammars. This visual debugger can be used to learn how PCYACC actually works, or can be used to debug grammars that just can't be debugged any other way.

Sources for YDB are included in the ydb_src.zip file, and can be found on the PCYACC PROGRAM DISK in the directory entitled \YDB. The YDB makefile is designed for Microsoft C 1.5 in case you want to rebuild or modify the debugger, otherwise just use ydb.exe as provided by abraxas. YDB32 is available for the Microsoft Windows 95/NT environments.

This debugger allows post analysis of a runtime session, the results of the session are written to ydb.err and this information with yy.lrt provides on-line visual debugging. YDB allows a user to step through an entire parsing operation.

For example type the following to build YDB support into your PCYACC SACALC example and recompile calc.c:

```
pcyacc -pydbpar.c -v -d calc.y     // build calc for YDB
```

In the above case -pydbpar.c includes the YDB skeleton parser for generating ydb.err at runtime, -v generates yy.lrt for state information, and -d generates yytab.h for converting enumerated tokens into token strings.

cl calc.c   // compile and link calc

At this point you must run the calculator to generate the ydb.err file. Just run CALC type "2+2" followed by <return>, and then type "quit".

calc   // execute the calculator

Now we are ready to enter the debugging session, ydb loads all the appropriate files and allows the user to visually examine all parser states.

```
ydb ydb.err yy.lrt yytab.h         // invoke the debugger
```

We hope this simple example gives you the background to apply this knowledge to your own project. Lastly, we suggest you modify the CALC example, and

rebuild and run CALC many time in conjunction with YDB to truly get the most out of this visual debugging system.

A complete gui development environment is available from Abraxas Software called PCYDB.

APPENDIX I. INSTALLING PCYACC

1. System Requirements

PCYACC will work on all IBMPC or compatible computer system configurations. Specifically, any configuration that will support MSDOS or OS/2 will also support PCYACC.

The following software programs are needed for software development using PCYACC:

1). A text editor (the built-in one in PWB will do quite nicely), the Programmers Work-Bench may also be called MSDEV.
2). A C programming language compiler (Microsoft C, or any C or C++ compiler should work without modifications; the generated ANSI C/C++ code was built to be very generic in nature)
3.) An experienced user may install PCYACC directly into MSDEV by using the "options" pull down in Microsoft Visual Developer C++.

2. Unpack and Backup the PCYACC PROGRAM Disk

The distribution medium for PCYACC PROGRAM is 3.5" diskettes. The structure and contents of the PROGRAM diskette are as follows:

```
readme.txt            // general overview of entire package


PCYACC.EXE

\sacalc directory     readme.txt
makefile
sacalc.y

\intopost directory   readme.txt
makefile
                      intopost.y

\pic directory        readme.txt
makefile
pic.h
pic.y
```

scan.c

NOTE: Both the structure and the contents of the PROGRAM diskette may change over time. Consult the "readme.txt" file in the root directory of the distribution disk for the latest information.

It is always a good practice to make a backup copy of your original diskettes to protect yourself from unexpected information loss. PCYACC is not copy protected. Lastly, PCYACC is actually five diskettes. The PCYACC program diskette and the PCLEX diskette contain executables, all other diskettes contain example software.

Updates are available by request via email at support@abxsoft.com and down-loadable via FTP from www.abxsoft.com. If you can’t find what you need please send email.

3. Installing PCYACC

PCYACC eliminates the need for a separate library code file, which was required by earlier UNIX versions and is still required by many other implementations. This change makes it transparent to you as user that there is a library routine that supports PCYACC's operation. This change also simplifies the installation process.

To perform the standard installation, follow these steps (there are many alternate ways of installing PCYACC):

1). Copy the PCYACC program to the MSDOS command prompt C:\BIN directory (or any other directory that MSDOS knows to look in for tools), also make sure your environmental PATH points to the \BIN directory.

2). Create a new directory inside of the \ETC directory and copy the PCYACC examples into it. (Files in subdirectories of the distribution diskettes contain several interesting examples, which you may want to copy onto your hard disk at this time; or you may choose to copy them later, when you actually need them.). To copy the contents of PROGRAM Diskette to your hard-disk for instance from the \etc directory: type, "**md pcyacc_program**<CR>", and "**xcopy a:*.* pcyacc_program /s**<CR>", this will copy all files and directories from the PROGRAM diskette to the new sub-directory on your hard-disk.

At this point, the installation process is complete and PCYACC is ready to go. However, it is recommended that you create a separate directory for your PCYACC projects. Creating a separate directory makes it easier to organize your PCYACC related files.

APPENDIX II. ERROR MESSAGES

Error Code: Error Message and Explanation

Y5000 *oversized production rule*
a grammar rule exceeds the limit of a pre-allocated static array.

Y5001 *nonterminal X not rewritten*
the nonterminal X should appear on the left hand side of a grammar rule at least once

Y5002 *pyield N error*
the internal array has a problem at index N

Y5003 *state/nolook error*
error at an internal state with no lookahead

Y5004 *state space overflow*
too many states

Y5005 *useless nonterminal symbol X*
the nonterminal symbol X cannot derive any terminal string

Y5006 *internal space overflow*
not enough internal memory

Y5007 *working set overflow*
not enough work memory

Y5008 *look-ahead set overflow*
not enough work memory

Y5009 *bad %start declaration*
syntax error in %start construction

Y5010 *bad %type declaration*
syntax error in %type construction

Y5011 *type redeclaration of terminal X*
type for terminal X is redefined

Y5012 *type redeclaration of nonterminal X*

type for nonterminal X is redefined

Y5013 *prec redeclaration of symbol X*
precedence for grammar symbol X is redefined

Y5014 *type redeclaration of symbol X*
type for grammar symbol X is redefined

Y5015 *misplaced type declaration of symbol X*
type declaration for grammar symbol X too late

Y5016 *syntax error*
syntax error, specific information unavailable

Y5017 *EOF before %*
EOF (end of file) before a %

Y5018 *bad first rule*
syntax error in the first grammar rule

Y5019 *token on LHS of grammar rule*
terminal used on the lefthand side of a rule

Y5020 *missing ; or |*
syntax error caused by missing ; or |

Y5021 *bad %prec declaration*
syntax error in %prec construction

Y5022 *nonterminal X after %prec*
only terminal symbol allowed in %prec construct

Y5023 *too many grammar rules, limit N*
the number of grammar rules exceeds N

Y5024 *LHS expects a value*

A type was defined for the non-terminal symbol on the left-hand side, the symbol expects to be assigned a value

Y5025 *potential type clash*
default type may cause a conflict

Y5026 *too many nonterminals, limit N*
the number of nonterminals exceeds N (Contact Abraxas Software)

Y5027 *too many terminals, limit N*

the number of terminals exceeds N

Y5028 *bad escape*
syntax error in character escaping sequenceY5029 *bad \nnn*
error in \nnn construction

Y5030 *\000 is illegal*
\000 is not allowed

Y5031 *symbol pool overflow*
too many symbols in the grammar

Y5032 *bad < ... >*
syntax error in <...> construction

Y5033 *missing ' or "*
syntax error in quote construction

Y5034 *bad reserved word X*
X is not a reserved word

Y5035 *X needs a type*
the symbol X has to have a type

Y5036 *misplaced definition of X*
the definition for X should be done earlier

Y5037 *EOF in %union definition*"
EOF encountered in %union definition

Y5038 *EOF before %}*
incomplete grammar file

Y5039 *bad comment syntax*
syntax error in comment

Y5040 *EOF in comment*
EOF encountered in comment

Y5041 *bad $<...>*
syntax error in $<...> construction

Y5042 *bad usage of %N*
illegal usage of %N construction

Y5043 *%N needs a type*

a type definition has to be provided for the grammar symbol referenced by %N

Y5044 *\n in string or character*
new line character occurred in string or character

Y5045 *EOF in string or character*
EOF found in string or character

Y5046 *non-terminating action*
action not terminated by }

Y5047 *action table overflow*
not enough space in internal action table

Y5048 *cannot open tempfile*
system failed to open intermediate file

Y5049 *bad tempfile*
corrupted intermediate file

Y5050 *internal array overflow*
certain internal array overflow

Y5051 *fail to place goto N*
unable to place a goto entry for state N

Y5052 *clobber of internal array (N, M)*
internal array fault

Y5053 *fail to place state N*
unable to place state entry N

Y5054 *internal space overflow*
internal memory overflow

Y5055 *cannot reopen tempfile*
system failed to reopen intermediate file

Y5056 *EOF inside action code*
EOF encountered inside action

Y5057 *quick check syntax error*
a syntax error uncovered by quick syntax checker

Y5058 *optimizer out of space*
overflow during optimization phase

APPENDIX III. ANNOTATED BIBLIOGRAPHY

1. GENERAL REFERENCES\

The following books were used to develop PCYACC we hope you have these excellent reference sources available in your office or library.

[Aho72a] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and compiling.* vol.I, *Parsing*, Prentice-Hall, Englewood Cliffs, NJ, 1972. For those who didn't get enough theory from Aho77.

[Aho72b] A.V. Aho and J.D. Ullman, *The Theory of Parsing, Translation and compiling.* vol.II, *Compiling,* Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Aho74] A.V. Aho and S.C. Johnson, "LR Parsing", *Computing Surveys*, 6:2, 99-124, June 1974.

[Aho77] A.V. Aho and J.D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading MA, 1977. Original classic - best foundation for those who like theory.

[Aho86] A.V. Aho, J.D. Ullman and R. Sethi, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1986. Where first edition [Aho77] was a graduate text this second edition is an undergraduate text.

[Allen95] Allen, James, *Natural language understanding*, 2nd ed. Benjamin/Cummins Publishing, Redwood City, CA 1995. An in-depth description of the natural language problem.

[App97] Appel, Andrew W., *Modern Compiler Implementation in Java*, Cambridge University Press, Cambridge, United Kingdom, 1997. Contains sample code for developing compilers in Java.

[Bac60] J.W. Backus, Report on the algorithmic language ALGOL 60, Communications of the ACM, Vol. 3, no. 5, p. 299-314, 1960. Chomsky's work on english language is applied to ALGOL. First application of BNF (Backus Normal Form).

[Cal79] P. Calingaert, *Assemblers, Compilers, and Program Translation.* Computer Science Press, Potomac, MD, 1979. If you ever need to know how to build a linker this is the book.

[Cho59] N.Chomsky, *On certain formal properties of grammars*, Inform. Control, Vol. 2, p.137-167, 1959. The definition and foundation of what we call context sensitive grammar. A general notation for programming languages was later developed for the language ALGOL by Backus. (BNF)

[DeR71] F.L. DeRemer, *Simple LR(k) Grammars*, Comm. of the ACM 14, 7 (1971).

[Fis91] C.N. Fischer, *Crafting a compiler with C,* Benjamin/Cummings Publishing, Redwood City, CA, 1991. A good undergraduate text for building a C compiler using YACC.

[Fri97] Friedl, Jeffery, Mastering Regular Expressions, O'Reilly & Associates, Sebastopol, CA 1997. A good reference source for regular expressions used in PCLEX and Extended BNF [PCYPP].

[Gri71] D. Gries, *Compiler Construction for Digital Computers*. Wiley, NY, 1971.

[Gru90] D. Grune, *Parsing techniques: a practical guide*., Ellis Horwood, West Sussex, England, 1990. This is good review of all common parsing techniques available as of 1990. Graduate Level - Great bibliography.

[Hol90] A.I. Holub, *Compiler Design in C*, Prentice-Hall, Englewood Cliffs, NJ, 1990. An excellent book for people building their own YACC or C compiler. Our highest recommendation

[Holmes95] Holmes, Jim. Object-Oriented compiler construction. Prenctice-Hall, Englewood Cliffs, NJ 1995. This is the best book on the design of your compiler using Object Oriented techniques.

[Joh78] S.C. Johnson, "YACC: Yet Another Compiler-Compiler", *Unix Programmer's Manual.* Bell Laboratories, 1978. The Original "YACC documentation", written by the creator of YACC.

[Joh78] S.C. Johnson and M.E. Lesk, "Language Development Tools", *The Bell System Technical Journal*, 1978.

[Ker78] B.W. Kernighan and D.M. Ritchie, *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, - 1978. The Bible of K&R C.

[Ker84] B.W. Kernighan and R. Pike, *The Unix Programming Environment.* Prentice-Hall, Englewood Cliffs, NJ, - 1984.

[Knu65] D.E. Knuth, O*n the Translation of Languages from Left to Right*, Information and Control 8, 6 (1965). Defines a parsing machine (PM) which can process context free grammars.

[Kru91] G.K. Krulee, *Computer processing of natural language*, Prentice-Hall, Englewood Cliffs, NJ - 1991, The natural language processing problems are discussed at a Graduate Level.

[Lev92] Levine, John / Mason, Tony / Brown, Doug, Lex & Yacc, O'Reilly & Associates, Sebastopol, CA -1992. This book is the basic tutorial for beginners, calculator and simple subset of SQL is discussed.

[Pys80] A.B. Pyster, *Compiler Design and Construction.* Van Nostrand Reinhold, New York, NY, 1980. Good Examples on building a PASCAL compiler.

[Sch85] A.T. Schreiner and H.G. Friedman, Jr. *Introduction to Compiler Construction with UNIX.* Prentice-Hall, Englewood Cliffs, NJ, 1985. Everything you need to know to build a small C compiler.

[Wil95] Wilhelm, Reinhard & Maurer, Dieter. Compiler Design Addison-Wesley Publishing, England - 1995. This book has valuable information on writing object oriented compilers.

Error Recovery References

Error handling is the real 'black-art' of compiler design and implementation, we think these references are useful.

[Bou84] P.Boullier, *Syntax Analysis and Error Recovery, in Methods and Tools for Compiler Construction*, B. Lorho, Cambridge Univ. press, 1984.

[Bur87] M. G. Burke, *A practical method for LL and LR syntactic error diagnosis and recovery*, ACM trans. Prog. Lang. Syst., vol. 9, no. 2, p. 164-197, 1987. Parsing is done with two simultaneous parsers, one with actions, one without. Estimates are that errors can be corrected with almost the same reliability as humans. The parser without the actions is used to maintain a heuristic database.

[Cie79] J. Ciesinger, *A bibliography of error-handling*, ACM SIGPLAN Notices, Vol. 14, no. 1, p. 16-26, 1979.

[Gra79] S.L. Graham, P*ractical LR Error Recovery*, Proc. SIGPLAN - Symposium. on Compiler Construction, 168, 1979

[Mau88] J. Mauney, *Determining the extent of lookahead in syntactic error repair*, ACM Trans. Prog. Lang. Syst., vol. 10, no. 3, p. 456-469, 1988.

[Pen78] T.J. Pennello, A *forward move algorithm for LR error recovery*, Fifth Annual ACM sym. on Prin. of Prog. Lang., p. 241-254, 1978.

[Poo77] G. Poonen, *Error Recovery for LR(k) parsers*, Info. Process. 77, p. 529-533, IFIP, North Holland Pub., Amsterdam, 1977

[Smi70] W.B. Smith, *Error Dectection in formal languages*, Journal Computer Science, Vol. 4, p. 385-405, 1970.

APPENDIX IV. GLOSSARIES

i. General Glossaries

Action: In syntax-directed parsing, when right hand sides of grammar rules are recognized (that is, when grammar rules are reduced), manipulative activities, or actions are triggered, to set flags, update global storages, allocate tree nodes, and whatever else is appropriate.

Ambiguous Context Free Grammars: A context free grammar that defines an ambiguous context free language.

Ambiguous Context Free Language: A context free language containing a sentence that has more than one parse tree.

Backus-Naur-Forms (or BNF): A notation for describing context free grammars, first used in Algol-60 report for describing the syntax of the language.

Bottom-up Parsing: A parsing technique that builds parse tree from leaves to the root.

Canonical Derivation: One way for deriving sentences defined by a grammar, in which each step always replaces the right most nonterminal symbol. Also called right most derivation.

Compiler: A computer program that translate programs written in a high level language into programs in low level languages.

Compiler Generator: A program that generates compilers from meta-level specifications.

Context Free Grammars: A formal system for specifying languages, in which the rewriting rules have the special format that their left hand sides are always a single nonterminal symbol.

Context Free Languages: Languages defined using context free grammars.

Derivations: Processes in which the left hand sides of grammar rules are replaced by their right hand sides.

Extended Backus-Naur-Form (EBNF): BNF extended with iterative and optional constructs.

Grammar Rules: In a grammar, grammar rules are used to define how each nonterminal should be rewritten. Also called rewriting rules or productions.

Grammar Symbols: Terminal symbols and nonterminal symbols in grammars are collectively called grammar symbols.

Interpreter: A computer program that translates and execute programs written in certain source language.

LALR: A particular instance of LR parsing in which at most one symbol lookahead is needed.

Left Hand Side (LHS): First part of a grammar rule that can only be a nonterminal symbol in the case of context-free grammars.

LR: A parsing technique that scans input from left to right and uses canonical reduction (that is, right-most-derivation in reverse).

Nonterminal Symbols: Grammar symbols that can be replaced by other grammar symbols or symbol sequences. Also called variables.

Object Languages: Languages into which translators translate source programs.

Object Programs: Output of program translators (including compilers), written in object language.

Parse Trees: A graphical representation of a derivation in which order information (that is, which variable was replaced first, which second, etc.) is no longer present.

Parser: A program that analyzes syntax of programs.

Parser Generator: A program that is capable of generating parsers automatically for compiler constructions.

Productions: see Grammar Rules.

Recursive Decent Parsing: A special top-down parsing technique in which no backtracking is needed.

Reduce: Reverse of derivation, that is, the process the right hand sides of grammar rules are replaced by their left hand sides.

Rewriting Rules: see Grammar Rules.

Right Hand Side (RHS): The second part of a grammar rule, which can be used to rewrite the LHS of the rule.

Right Most Derivation: see Canonical Derivation.

Scanner: Front end of a parser, which digests raw text inputs and partition them into meaningful lexical units, or tokens.

Source Languages: Languages with which programs are written are the subjects for translators to work on.

Source Programs: Programs written in source languages.

Stack: A special list whose access is restricted to one end of the list, a very useful data structure.

Syntax Trees: see Parse Trees.

Terminal Symbols: Grammar symbols that can not be rewritten. These are the symbols that make up sentences of languages.

Token: Smallest syntax unit recognized by scanners or lexical analysis.

ii. PCYACC Specific Glossaries

Declaration Section: The first part of a GDL program, in which one defines terminal symbols for grammars, declares types for grammar symbols, precedence and associativity for grammar symbols, etc.

Grammar Description File (GDF): Input file to PCYACC, with which one defines the syntax for source languages.

Grammar Description Language (GDL): The language one uses to write grammar description programs.

Grammar Description Programs (GDP): Programs written using GDL, and contained in GDF's.

INFIXEL.Y: Name of the file containing a simple GDP the defines the syntax of a infix expression notation.

INTOPOST: A program for translating infix expression notation to postfix expression notation.

Left: PCYACC keyword for defining left associativities of grammar symbols.

Nonassoc: PCYACC keyword for defining non-associativity of grammar symbols.

Prec: PCYACC keyword for defining precedence of grammar symbols.

Program Section: The last part of a GDP, where C routines can be included.

Reduce/Reduce Conflicts: In an ambiguous GDP, more than one grammar rule can be applicable at the same time to perform a reduce. They usually mean errors in the GDP.

Right: PCYACC keyword for defining right associativity of grammar symbol.

Rule Section: The second part of a GDP, where grammar rules and their associated actions are defined.

SACALC.Y: Name of the 'yacc' file containing an expression definition of a Simple Arithmetic Calculator.

Shift: A semantic action in which the next token is retrieved.

Shift/Reduce Conflicts: A shift/reduce conflict occurs if applicable processing activities can be either a reduce or a shift.

Start: PCYACC keyword for defining root grammar symbol.

Token: PCYACC keyword for defining non-terminal symbols.

Type: PCYACC keyword for defining types of grammar symbols.

Union: PCYACC keyword for defining stack types.

*yyclear*in: A macro for error recovery.

yyerrok: A macro for clearing the error flag.

yyerror: A routine that generated parsers call to handle errors during parsing.

yylex: A lexical analyzer used by generated parsers to recognize tokens.

yyparse: Main parsing routine, generated by PCYACC.

YYSTYPE: Type of the parser stack (see %union).

'%%' : Delimiters for separating different sections of a GDP.

'$$' : Symbol for referencing values of LHS of grammar rules.

'$1' : Symbol for referencing values of the first grammar symbol of the RHS of grammar rules.

'$2' : Symbol for referencing values of the second grammar symbol of the RHS of grammar rules.

...

'$N' : Symbol for referencing values of the N-th grammar symbol of the RHS of grammar rules.

'%' : Introduces keywords.

':' : Separates the LHS and the RHS of grammar rules.

'|' : Separates alternative RHS's for an LHS.

';' : Terminates grammar rules.

'%{ ... %}' : Delimit's C definitions in declaration section.

'< ... >' : Delimit's type tags in %type declarations.

'{ ... }' : Delimit's C code action in rule section.

III. PCYACC INPUT SYNTAX SUMMARY

The following is meant to be a typical input file template, which should not be treated as complete syntax definitions.

```
%{
  // C/C++-includes or C/C++-initial definitions
%}

%union {
type1 typetag1;
type2 typetag2;
...
}

%token TERMINAL_1
%token TERMINAL_2
%token OPERATOR_1
%token OPERATOR_2
%token OPERATOR_3
...
%type <typetag_1> TERMINAL_1
%type <typetag_1> nonterminal_1
%type <typetag_2> TERMINAL_2
%type <typetag_2> nonterminal_2
%left     OPERATOR_1
%right    OPERATOR_2
%nonassoc OPERATOR_3
%start start_symbol

%%     // grammar definition section and actions

start_symbol
  : nonterminal_1 OPERATOR_1 nonterminal_2
    { // C/C++ -code-action_1 }
  | nonterminal 1 OPERATOR_2 nonterminal_2
    { // C/C++ -code-action_2 }
  | nonterminal_1 OPERATOR_3 nonterminal_2
    { // C/C++ -code-action_3 }
  ;

nonterminal_1
  : TERMINAL_1 { // C/C++ -code-action_4 }
  | TERMINAL_2 { // C/C++ -code-action_5 }
  ;

nonterminal_2
  : TERMINAL_2 { // C/C++ -code-action_6 }
```

```
  ;

%%      // local code definition section

   // local C/C++ -code ( passed through to parser )
```

Index

%PREC, 80
ABRAXAS, 2
ABRAXAS PHILOSOPHY, 132
ACCEPT, 74
ACTION, 40
ACTION, 164
ACTIONS, 22, 61
ACTIONS, 61
AMBIGUITY, 53, 73
AMBIGUOUS, 53, 54
ARTIFICIAL LANGUAGES, 44
ASSEMBLERS, 17
ASSEMBLY LANGUAGE, 17
ASSOCIATION, 77
ASSOCIATIVITY, 77, 81
AT&T BELL, 5
AXEL SCHREINER, 138
AXIOMATIC-APPROACH, 103
BACKUS-NAUR-FORMS, 28
BACKWARD SUBSTITUTION, 78
BASIC, 44
BNF, 28, 164
BORLAND C++, 146
BORLAND IDE, 146
BOTTOM-UP PARSING, 52
BUILDING THE EXECUTABLE, 24, 43
C, 44
C++, 6, 132, 146
CANONICAL DERIVATION, 51
CANONICAL DERIVATIONS, 74
CANONICAL LR PARSERS, 55
CANONICAL REDUCTION, 51
CFG, 26
CHOMSKY, 45
CLR, 55
CODE GENERATION, 107
CODEVIEW, 9
COMMAND LINE OPTIONS, 8
COMMAND LINE OPTIONS, 8
COMPILATION PHASE, 17, 26
COMPILED EXECUTION, 26
COMPILER, 26
COMPILER GENERATOR, 6
COMPILERS, 6, 17
COMPILER-WRITER, 30
COMPONENTS, 146
COMPUTER LANGUAGES, 16
CONFLICT-RESOLVING, 76
CONFLICTS, 89
CONSTANT, 35
CONTEXT-FREE GRAMMARS, 26, 45
CONTEXT-SENSITIVE GRAMMARS, 45
CROSS REFERENCE, 149
DEBUGGING, 85
DEBUGGING PCYACC GRAMMARS, 151
DECLARATION SECTION, 21, 30, 57, 58
DENOTATIONAL-APPROACH, 103
DERIVATION, 47
DERIVES, 49
DIAGNOSTIC MESSAGES, 113
DIRECTLY DERIVES, 49
DISASSEMBLERS, 17
DRAGON, 3
DS, 30
EBNF, 28
ERROR, 74, 81
ERROR HANDLING, 81, 136, 139, 142
ERROR MESSAGE, 156
ERROR PROCESSING DISK, 134
ERROR RECOVERY, 84, 140
ERROR RECOVERY, 139
ERROR RULE, 83
ERRORS, 113
EXECUTION PHASE, 17, 26
EXTENDED BACKUS-NAUR-FORMS, 28
FORMAL GRAMMARS, 44
FORMAL LANGUAGE THEORIES, 44
FORMAL LANGUAGES, 44
FORTRAN, 44
FORTRAN90, 132
GDF, 8, 30
GDL, 2, 6, 30, 166
GDP, 8, 30, 166
GRAMMAR DESCRIPTION, 57
GRAMMAR DESCRIPTION FILES, 8, 30
GRAMMAR DESCRIPTION LANGUAGE, 6, 30
GRAMMAR DESCRIPTION PROGRAM, 8
GRAMMAR DESCRIPTION PROGRAMS, 30
GRAMMAR FOREST, 149
GRAMMAR RULE SECTION, 21, 30, 57, 60
GRAMMAR RULES, 26, 46
GRAMMAR SYMBOL, 26
GRAMMAR SYMBOLS, 46
HANDLE, 55
HASHING, 109
HIGH LEVEL LANGUAGES, 16, 26
HIGHER PRECEDENCE, 137
IBM PC, 19
IBMPC, 153
IMBED ACTIONS, 84
INFIX, 33
INFIX, 33
INFIXEL, 34
INFIXEL EXPRESSION, 35
INFIXEL FACTOR, 35
INFIXEL PROGRAM, 35
INFIXEL TERM, 35
INFIXEL.Y, 36
INHERENTLY AMBIGUOUS, 53
INSERT, 104

INSTALLATION PROCESS, 155
INSTRUCTION SET, 107
INSTRUCTIONS, 16
INTERNAL NODES, 50
INTERNAL SKELETON PARSER, 9
INTERPRETED EXECUTION, 26
INTERPRETER, 26
INTOPOST, 33
INTOPOST.C, 42
JAVA, 6, 160
JOHNSON, 5
KEYWORD, 30
KEYWORD **PREC**, 80
KEYWORD **UNION**, 70
LALR, 8, 55
LEAF NODES, 50
LEFT, 30, 77
LEFT ASSOCIATIVE, 77
LEFT HAND SIDE, 165
LEFT RECURSION, 40
LEFTHAND-SIDE, 27, 47
LEFTMOST DERIVATION, 51
LEFTMOST DERIVATION TREE, 51
LEFTMOST REDUCTION, 51
LEX AND YACC, 133
LEXICAL ANALYSIS CAVEATS - COMBINING LEX & YACC, 132
LEXICAL ANALYZER, 64, 100, 119
LHS, 27, 47, 167
LINE NUMBER, 149
LINEAR PROBING STRATEGY, 111
LINT, 6
LOAD-AND-GO, 115
LOCATING CONFLICTS, 93
LOOK-AHEAD LR, 8
LOOK-AHEAD LR PARSERS, 55
LOOKAHEAD TOKEN, 136
LOW LEVEL LANGUAGE, 17
LOWEST PRECEDENCE, 137
LR PARSERS, 55
LRT, 89
MACHINE DEPENDENT, 107
MACHINE LANGUAGES, 16, 26
MAKE, 24
MARK(), 114
MICROSOFT MFC, 146
MICROSOFT WINDOWS 95/NT, 151
MS WINDOWS 3.X, 146
MSDOS, 2, 19, 153, 155
MSDOS TOOL, 24
MULPAR, 132, 147
MULTIPLE PARSERS, 9
NATURAL LANGUAGES, 44
NONASSOC, 30, 77
NONASSOCIATIVE, 77
NONTERMINAL SYMBOLS, 26, 45
OBJECT CODE, 107, 108
OBJECT LANGUAGE, 17
OBJECT PROGRAM, 26
OBJECT PROGRAMS, 17
OPERATIONAL-APPROACH, 103
OPTIONS, 8
OS/2, 153
OWL, 146
PARSE STACK, 136
PARSE TREE, 9, 13, 51
PARSE TREES, 49
PARSE TREES, 129
PARSER CLASS, 146
PARSER GENERATOR, 7, 30
PARSER SKELETON, 9, 12
PARSER.H, 146
PASCAL, 44
PCYACC, 2, 76
PCYACC OBJECT ORIENTED TOOLKIT, 132
PHRASE GRAMMARS, 45
PIC, 115
POSTFIX, 33
POSTFIXEL, 34
PREC, 166
PRECEDENCE, 22, 77, 166
PREPROCESSORS, 17, 113
PRODUCTION RULES, 26
PRODUCTIONS, 45
PRODUCTIONS, 46
PROGRAM, 47
PROGRAM SECTION, 21, 30, 57
PROGRAMMING, 16
PROGRAMMING LANGUAGE, 26
PROGRAMS, 16
PROGRAMS, 26
PS, 30
PWB, 36, 129, 153
QUICK SYNTAX CHECK, 12
README.NOW, 10
RECURSION, 147
RECURSIVE DECENT PARSING, 52
REDUCE, 74
REDUCE ACTION, 54
REDUCE/REDUCE, 75
REDUCE/REDUCE CONFLICTS, 75, 78
REDUCTION, 47
RE-ENTRANCY, 147
REGULAR GRAMMARS, 45
REWRITING RULES, 46
RHS, 27, 47, 167
RIGHT, 30, 77
RIGHT ASSOCIATIVE, 77
RIGHT HAND SIDE, 165
RIGHT RECURSION, 40
RIGHTHAND-SIDE, 27, 47
RIGHTMOST DERIVATION, 51
RIGHTMOST DERIVATION TREES, 51
RIGHTMOST REDUCTION, 51
ROOT NODE, 50
SACALC, 19, 25
SACALC, 25
SACALC.C, 24
SACALC.Y, 19
SAG, 47
SCANNER, 64
SEARCH, 104
SEARCH ALGORITHM, 109
SEMANTICAL CHECKING, 122
SENTENCE, 47, 49
SHIFT, 74

SHIFT OPERATION, 54
SHIFT/REDUCE, 75
SHIFT/REDUCE CONFLICT, 76
SHIFT/REDUCE CONFLICTS, 75, 96
SHIFT/SHIFT, 75
SHIFT-REDUCE PARSERS, 54
SIMPLE LR PARSERS, 55
SLR, 55
SOURCE LANGUAGE, 17
SOURCE PROGRAM, 26
SOURCE PROGRAMS, 17
STACK, 33, 70
STACK MACHINE, 33
STACK TOP, 33
STACK TYPES, 167
START, 30, 59
START SYMBOL, 27, 45
STEPHEN C. JOHNSON, 5
SYMBOL INSERTION, 110
SYMBOL LOOKUP, 111
SYMBOL TABLE, 104
SYMBOL TABLE MANAGEMENT, 99
SYMBOL TABLE MANAGEMENT, 109
SYNTAX CHECK, 12, 129
SYNTAX DEBUGGING, 9
SYNTAX ERROR, 81, 137
SYNTAX TREE, 130
SYNTAX TREES, 49
TEMPLATE, 57
TERMINAL SYMBOLS, 26, 45
TOKEN, 21, 30
TOKEN VALUES, 67
TOKENS, 26, 45, 58
TOKENS.EXE, 134, 142
TOP-DOWN PARSING, 52
TPARSER CLASS, 146
TPARSER.H, 146
TRANSLATORS, 17
TUTORIAL, 161
TYPE, 30, 67, 70, 71
TYPE OF A TOKEN, 67
TYPEOF, 104
UNION, 70
UNION TAGS, 71
UNION TYPE, 68, 70, 71, 72
UNIX, 5
UNIX YACC, 5
VALUE, 67
VALUES, 61
VARIABLE, 35
VISUAL DEBUGGER, 151
WINDOWS, 146
WINDOWS 95/NT, 146
YACC, 5
YACC TOOL, 149
YDB, 151
YDB SKELETON PARSER, 151
YDB.ERR, 151
YDB32, 151
YTOOL, 149
YY.AST, 9
YY.AST, 12, 130
YY.LRT, 10, 151
YY.LRT, 13, 89, 91
YYCLEARIN, 81, 84, 167
YYERROK, 81, 84, 137, 167
YYERROR, 135, 167
YYERROR(), 22
YYERROR(), 69
YYLEX, 167
YYLEX(), 9, 22
YYLEX(), 64, 101
YYLVAL, 67, 68
YYPARSE(), 22, 41, 147
YYSTYPE, 70, 71, 72
YYTAB.C, 8
YYTAB.C, 10
YYTAB.H, 8, 151
YYTAB.H, 11, 69

COMPILER CONSTRUCTION ON PERSONAL COMPUTERS

(WITH PCYACC™)

PCYACC® is a software product of ABRAXAS SOFTWARE INC.

For more information, contact

ABRAXAS SOFTWARE INC.
Post Office Box 19586
PORTLAND, OR 97280 USA

TEL: 503-232-0540
FAX: 503-232-0543
Internet: support@pcyacc.com
www.pcyacc.com

Copyright © 1984-2000 by ABRAXAS SOFTWARE INC.

PCYACC MANUAL PRINTED - 4 JANUARY 2000