TABLE OF CONTENTS

PREFACE

ABRAXAS PCLEX is a program generator for writing lexical scanners. Lexical scanners read a stream of characters and divide it into identifiers, keywords, and other symbols of the target language. **PCLEX** translates a scanner written in the Scanner Description Language (SDL) into the host language **C**. SDL is a high level language oriented towards string matching. It is a special purpose language; where the generality of a programming language is needed, scanner descriptions can be extended with code sections written in **C**. SDL allows software developers to concentrate on what the scanner recognizes instead of getting bogged down in the details of how. It reduces the work necessary to bring a project to completion. Programs are done sooner with fewer errors and updates are simplified.

PCLEX is source file compatible with ‘classic’ AT&T UNIX LEX, uses the faster FLEX algorithms, and has a command line format similar to **PCYACC**. It is self-contained and does not require additional source, object, or library files. **PCLEX** generated scanners integrate easily with **PCYACC** generated parsers. PCLEX used with Abraxas Object Oriented Toolkit is capable of generating C++, Java, Pascal, Delphi, and Visual Basic.

I. INTRODUCTION

This manual describes the **ABRAXAS PCLEX** features and gives examples of both standalone text processing programs and lexical scanners. It is both a tutorial on LEX use and the reference manual for **ABRAXAS PCLEX**. It is useful for all users from novices to experienced software professionals.

1. Typographic Conventions

PCLEX is largely independent of the characteristics of the hardware it is running on. It has been adapted for most microcomputer systems and can be easily adapted for others. Therefore, the presentation also is largely system independent. Where a specific system is used to make the discussion concrete, an IBM-PC/XT running MS-DOS is used. In the appendix, there are tips on system dependent characteristics of **PCLEX**, including installation and hardware and software requirements.

Throughout this manual, the following typefaces and syntax conventions are used to aid clarity:

1). Words or phrases that have important technical meanings or that have special interpretations in **PCLEX** will be underlined when they first appear. Definitions and/or explanations are given at this time in the normal text style.

Example: **PCLEX** programs are written in the scanner description language, or SDL, which is a high level language for lexical specifications of computer input.

2). When illustrating user-machine interactions, user entered text is in italics. Messages or prompts from the computer are in normal text. A carriage return is represented as *<CR>*.

Example: C>*pclex example.l<CR>*

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

Example: The following is a small scanner description file that you will see again later in the tutorials:

```
        letter              [a-zA-Z]
        %%
        {letter}+           ECHO;
        .                   ;
```

4). Special meaning characters or strings will be bolded to help you distinguish them from surrounding text.

Example: Regular expression **[a-zA-Z]** matches any letter.

2. Examples

This manual contains three examples: **Word Count Program (Chapter V)**, **Date Program (Chapter VII)**, and **ANSI C Syntax Analyzer Program (Chapter VIII)**. The source code of these examples are in the **PCLEX [DOS-OS/2] Toolkit Disk** under **\WC**, **\DATES**, and **\ANSIC** directories respectably. Each directory is self-contained, including all source files, a **MAKEFILE** for **UNIX** style **MAKE** programs, and **makefile** files for **Microsoft C 7.0's MAKE**. The disk contains one more example called **DIGRAPH** under **\DIGRAPH** directory. The purpose of **DIGRAPH** is explained in **Chapter X**, however, the completed example is not included in this manual.

II. OVERVIEW OF PCLEX

This chapter gives a brief overview of **PCLEX**, its origin, history, and evolution, and a description of its operation at an abstract level.

1. History

PCLEX is an descendent of FLEX (Fast LEX), which is an improved version of LEX. LEX was first publicly described in an article by **Michael Lesk** and **Eric Schmidt**. The article is one of the most widely cited in computer literature. That version of LEX was designed, coded, and debugged by **Eric Schmidt**, based on ideas from **Stephen C. Johnson** and **Alfred Aho**. FLEX was programmed by **Vern Paxson** and **Kevin Gong**, based on ideas developed by **Van Jacobson.** FLEX added some useful additional features and significantly speeded up both the generated scanners and scanner generation. LEX (and more recently FLEX) has been part of the standard language tools provided in the UNIX operating system environments.

During the past decade or so, numerous software projects, both large and small, have been developed with LEX. As such, LEX has historically been one of the most valuable tools available to application and programming language developers. LEX can drastically reduce the time and complexity normally involved in program and compiler development. Compiler writing tools like LEX provide developers with a powerful aid to writing compilers, command interpreters, and other programs with text input.

Because LEX is such a useful tool, many software developers have ported it to their own systems, or re-implemented it in their own software environments. **PCLEX** is an example of an implementation of FLEX on micro computers (IBM PC's and Macintosh's).

PCLEX is designed and implemented to be upward compatible with LEX and FLEX. All of the features of LEX are retained in **PCLEX**, including the improvements from FLEX. Although **PCLEX** has an almost identical programming interface with LEX, there are quite a few differences, which will become clear as we go along.

2. What PCLEX Does

PCLEX is a computer program. Like most other programs, it takes input, computes values, and produces output. It is a translator, i.e., its input is in one language (SDL) and its output is in another (**C**). Program generators are programs that translate from a very high level language that is tailored to one area of programming or computing to a lower level language, usually a general purpose programming language like **C**. LEX and YACC, which is the acronym of "Yet Another Compiler-Compiler", are program generators for writing compilers. Compilers are translators from a programming language to a very low level language like assembly language or machine language (i.e., object code). **PCLEX** is written using **PCYACC** and itself.

SDL is designed for describing the text input of a program at the lexical level, that is the basic words and symbols of the input language. SDL is a language for describing languages, a meta-language. A written language is expressed as a string of characters. Characters fall into several broad groups: for example, letters, digits, and punctuation. Identifiers and key or reserved words are built out of characters according to the convention decided on by the language's designer. For example, identifiers in **C** can contain the underscore character ('**_**'). **FORTRAN** and **Pascal** identifiers are limited to an initial letter and additional letters and digits.

III. COMMAND LINE AND OPTIONS

This chapter tells how to run **PCLEX**, its filename conventions, and the command line options. **PCLEX** is command line driven and can be run either from batch procedures or from the DOS prompt of other programs.

1. Command Line Format

PCLEX retains the flavor of **PCYACC**'s command line interface.

PCLEX [options] <sdf_name>

Where <**sdf_name**> is the name of a scanner description file (SDF) and [options] represents zero or more command line options. If **PCLEX** is invoked with no arguments, it outputs a short message advising you of the correct command line format.

1.1 File Name Conventions

The <**sdf_name**> can be any legal MS-DOS filename. For consistency and clarity, it is recommended that you pick one extension and stick to it. All examples in this manual use the extension "**.L**". The default name of the output file is the basename (the filename without any extensions) plus the extension "**.C**". For example, if the input file name is "**EXAMPLE.L**", the output file name is "**EXAMPLE.C**". The output file name can be changed with the "**-c**" and "**-C**" options (see the next section).

1.2 Command Line Options

Command line options are used to override default actions or change the file name conventions. The available options are:

-c: This option overrides the default output **C** file name. Instead of using the basename of the input scanner description file plus the "**.C**" extension, **PCLEX** uses "**YYLEX.C**" as the output scanner source file name.

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

-h: Show a help screen.

-i: Build a case-insensitive scanner. The case of letters in the patterns is ignored and patterns are matched regardless of case. The matched text in "**yytext**", the internal defined character pointer pointing to the matched input token, is not altered, the original case of the scanner input is preserved.

-n: Suppress **#line** directives in the output scanner source file. This option is useful if you are trying to use a source code debugger. In normal operation the output scanner source file uses **#line** to make the reference to the original scanner description file, this normally causes source code debuggers like **CodeView** to generate strange results.

-p<pf>: Use the user provided scanner skeleton in <pf> instead of the default (see **Appendix C**). This option can be used to generate C++ scanners.

-s: This option suppresses the default rule (that unmatched input be written to "**stdout**"). With this option, if the scanner finds input that is not matched by any rules, the scanner program quits with a "**pclex scanner jammed**" message.

Only the "**-c**" and "**-C**" options are case-sensitive. The rest are the same for both upper and lower case. For example, "**pclex -h**" and "**pclex -H**" are equivalent.

2. Using Command Line Options

This section shows how to use the command line options. The following example, "**EXAMPLE.L**", is used throughout this section:

```
letter              [a-zA-Z]
%%
{letter}+           ECHO;
.                   ;
```

With no options, **PCLEX** writes the **C** output file to "**EXAMPLE.C**".

2.1 Override Output C File Conventions (-c and -C)

The "**-c**" and "**-C**" options override the output **C** filename convention of **PCLEX**. With the "**-c**" option, the output is written to "**YYLEX.C**". For example:

C>*pclex -c example.l*

creates "**YYLEX.C**" in your current directory.

The "**-C**" option lets you specify the name of the output file. For example:

C>*pclex -Canything.c example.l*

creates "**ANYTHING.C**" in the current directory.

2.2 Ask PCLEX for Help (-h or -H)

The two help options, "**-h**" and "**-H**", do the same thing. They ask **PCLEX** to show a help message on the screen. This display is helpful when you are first learning how to use **PCLEX** and later on it is a helpful reminder for infrequently used options.

```
C>pclex -h
```

shows the following message:

```
Abraxas Software (R) PCLEX Version - 8.01 [NT].
Copyright (C) Abraxas Software, Inc. 1986-98, all rights reserved

Usage: pclex options scanner.l
Available options:
-c : use file "yylex.c" for C output
-Cf: use file f for C output
-h : display this help message
-i : case insensitive
-n : do not generate '#line' directive in output
-pf: use skeleton f as scanner driver
-s : suppress scanner output
```

and,

```
C>pclex -h example.l
```

shows the help message and generates the scanner source file **EXAMPLE.C.**

2.3 Generate Case-insensitive Scanner (-i or -I)

Normally, **PCLEX** generates scanner that treat upper and lower case letters as distinct. For example, the pattern "**abc**" will not match "**ABC**". This option tells **PCLEX** to build a case-insensitive scanner. The case of letters in **PCLEX** input patterns and the scanner will be ignored and the rules will be matched regardless of case. The scanner simply converts all lower case letters in the **.L** scanner description file into upper case. The matched text in "**yytext**" will have case preserved.

The following statement

```
letter          [a-zA-Z]
```

can be rewritten as

```
letter          [a-z]
```

or

```
letter          [A-Z]
```

with the same scanner behavior if one uses the command line:

C>*pclex -i example.l*

2.4 Suppress #line Directives in Scanner (-n or -N)

Normally, **#line** directives are put in the **.C** output scanner source file to provide a way of correlating line numbers in compiler error messages with the original line in the input scanner definition file. This option causes the **#line** directives to be omitted. Some source level debuggers handle **#line** directives incorrectly.

2.5 Override Default Scanner Skeleton (-p or -P)

Normally, the generated scanner is built around the default scanner skeleton. The **C** source code for the default skeleton of a scanner program is internal to **PCLEX**. The "**-p**" option allows a custom scanner skeleton to be used. The form of the scanner skeleton is explained in **Appendix C**. The default skeleton **C** source file "**LEXSCAN.C**" used to built the internal skeleton is in the **\PCLEX** directory, available for customization. For example:

C>*pclex example.l*

and

C>*pclex -plexscan.c example.l*

are equivalent. And,

C>*pclex -pmyscan.c example.l*

will use **MYSCAN.C** to produce **EXAMPLE.C**.

2.6 Suppress Default Action (-s or -S)

Each piece of the scanner's input text that is matched by a pattern specified in the scanner description file has an <u>**action**</u> associated with it. For scanner input text not matched by any pattern, the <u>**default action**</u> is taken. Normally, the default action is to copy the unmatched text to the output. The "**-s**" and "**-S**" options suppress the default action and treat unmatched input text as an error: the scanner outputs the message "**pclex scanner jammed**" on the standard error device (usually the screen) and exits.

2.7 Foreign Support for the 8-bit ASCII Character Set (-8)

By default **PCLEX** generates 8-bit character scanner which reads input files written in symbols in the 8-bit ASCII character set. With the **-8** option, **PCLEX** will generate 7-bit character scanner which can read input files written in symbols in the 7-bit ASCII character set. If you define hi-bit set characters in your Lexical Definition File (.L), you must define the rules with octal characters, i.e.,characters above octal `177(8)` must be defined as `\0200 [200(8)].`

IV. BASIC CONCEPTS REVISITED

This chapter introduces the basic concepts related to **PCLEX** such as computer programming languages, translators, interpreters compilers, lexical analyzers, grammar of languages, and regular expressions. Basic technical terms used in later chapters are also explained. Some terms are listed to prepare for a more formal discussion of those subjects in later chapters

1. What is a Programming Language

Computers do an overwhelming and constantly expanding variety of tasks. Nevertheless, computers are not intelligent. They must be told how to do each task with a set of step by step instructions, called programs. To do these seeming miracles, someone has to work out the sequence of steps to accomplish the task in minute detail and present this step by step program to the computer in a computer language. A programming language is an artificial language used to write the detailed instructions necessary for the performance of any task by the computer. Programs of any length are the basic forms in which these instructions are presented to the computer for execution.

There are two broad classes of programming languages: low level languages like machine language and assembly language that are machine instruction oriented, and high level languages like **C**, **Basic**, and **Pascal** that are largely machine independent and are oriented towards arithmetic and logical operations. Machine language is directly executable by the computer. Assembly language maps directly into machine language and translation is fast and easy.

2. What is a Programming Language Translator

A computer can only directly understand programs written in its own language, called machine language, one kind of low level language. Machine language programs are written in binary digits. Machine languages are very primitive. Writing programs in machine language is tedious and very error-prone. Soon after the invention of the computer, symbolic languages were developed to shift some of the burden of programming to the computer itself. Instead of writing the program in binary digits, meaningful names (mnemonics) were given to each of the machine's instructions and the program can then be written in an easier to read symbolic form. A translation then has to be done to translate the programs written in symbolic languages to the programs written in machine languages. At first the translation was done by hand. Later a program, called translator, was written in machine language to translate symbolic language programs to machine language programs. After the first such translator was written, programming no longer needed to be done in machine language. The first symbolic languages were still very oriented toward the machine's own instruction set (the computer's repertoire of basic actions). They are called assembly languages, another kind of low level language, and the translator is called an assembler. Conversely, translators that translate machine language programs into assembly language programs are called disassemblers.

3. What are Compilers and Interpreters

Translation to machine language is not limited to just low-level languages. A high-level language describes a program in terms that are better matched to the task at hand and human thought processes. Programs written in high level languages have two different modes of execution: interpreted execution and compiled execution.

Compilers are translators between high-level languages and low-level languages. A compiler translates the source program from a high level language to an object program in a low level language (either assembly language or machine language). 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. A preprocessor translates from a high level language to a similar language. Usually, the preprocessor's object language is a subset of its source language. Compiled execution mode is divided into two phases: a compilation phase and an execution phase. During compilation, a compiler first recognizes the input source program written in the source language then composes an equivalent object program. written in the object language. In the execution phase, the object program (in machine language) is executed.

Programs can be written that carry out the source language program actions directly instead of first translating them to machine language. Such a program is called an interpreter. Interpreters skip the compilation step. The computer runs the interpreter machine language program and the interpreter runs the source language program. This extra layer of program slows down running the source program. Interpreters remove the compile step from the usual edit-compile-debug cycle of development. In interpreted execution mode, program statements are decoded and executed one at a time. Each step includes translation of a statement followed the appropriate machine actions dictated by the statement.

4. Grammar of a Language

A programming language can be defined in a number of ways, some formal and rigorous, others casual and illustrative. A very small language may have a finite number of programs and the language could be defined by listing them all. A more general method is to describe the grammar of the language, the set of rules that outline what the valid structures and constructs are and how they are combined. Formal languages, languages defined with mathematical rigor, provide a solution to the problem of describing infinite languages in a concise manner, using only a finite number of symbols. The rules for precise definition of computer languages make up what is called formal grammars.

According to **Noam Chomsky**, who is well known to computer science community for his contributions to the study of formal languages, there are four classes of grammar to generate languages in four levels. The grammars are listed according to increasing description power and complexity as following: regular grammars, context-free grammars, context-sensitive grammars and phrase grammars. Subsequent research had identified four corresponding abstract machine types, which can recognize those strings written in the languages generated by their respective grammars. Corresponding to the grammars listed above, these abstract machines or automata are: finite-state automaton, push-down automaton, linear-bounded automaton, and Turing machine.

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 and powerful. Phrase grammars 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. For most applications, context-sensitive grammars are appropriate since most programming languages of practical use are context-sensitive. However, when considering computational complexities, context-free grammars are so much easier to handle. They can be most efficiently implemented as mechanical procedures. Therefore, they are used almost exclusively by computer science professionals. Most of the real compilers are written first as context-free grammars which are then mechanically translated into codes.

At the first look it seems that the choice of context-free over context-sensitive a big compromise, since most of the practical programming languages are context-sensitive. However, 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. A common practice is to shift the duty of processing context-sensitive components to a so called semantic-analysis phase.

A grammar defines a language by explaining which sentences may be formed. It consists of four components, they are the terminals, the non-terminals, the rewriting rules, and the start symbol

Terminals, also known as tokens, are the basic building elements of programs. They are constant symbols because each token represents itself. In a programming language, terminal symbols are used to write programs. Using **C** language as an example, key words such as **if**, **then**, **else**, **which**, **break**, **continue**, **return** etc., constants such as **10**, **2.5**, **'y'**, **"text-string"** etc., and identifiers such as **line_count**, **line_buffer** etc. are all terminal symbols. The terminal symbols are not further explained in the grammar; they are the alphabet and words in which the sentences of the language, which the grammar describes, are written. An alphabet is a finite set of symbols; a word is a sequence of alphabetic symbols; a sentence is a sequence of words comprising a language; and a language is a set of words formed from the alphabet.

Non terminals are syntactic variables that can take on different values of strings made up of non terminals and terminals. The presence of non terminals in a grammar usually corresponds to important linguistic constructs of the language defined by the grammar. In a typical programming language such as **C**, **data-declaration**, **function-declaration**, **statement** and **expression** are normally recognized as non terminals.

Productions, or rewriting rules or grammar rules, specify the manner in which the terminals and non terminals can be combined to form strings. For each non terminal, a production must exist, and any one of the formulations from this production can be substituted for the non terminal. A production rule of a grammar consists of a left-hand-side (LHS) and a right-hand-side (RHS), separated by an arrow:

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 --> X1  X2  ...  Xn
```

where the left-hand-side of the grammar rule, **X**, has to be a single, unique non terminal symbol, and any of the components of the right-hand-side of the grammar rule, **Xj** , can be either a terminal or a non terminal. It could even be the left-hand-side, **X**. 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

```
X1  X2  ...  Xn
```

in that order. Also, whenever the sequence

```
X1  X2  ...  Xn
```

occurs, it can be reduced to **X**. The process of replacing the LHS of a production by its RHS is called a (one step) derivation. The inverse of a derivation, the process of replacing the RHS of a grammar rule by its LHS, is called a reduction. The replacement of a sequence of grammar symbols by a non terminal 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. Again using **C** language as an example (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 non terminal 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.

5. Regular Grammar and Regular Language

Regular grammars (RE grammars), also called finite-state grammars (FS grammars), are used to defined regular languages describing the structure of text on the character level. That is, the terminal symbols in a regular language are single characters. Hence, a right-hand side in a regular grammar contains at most one non-terminal and regular grammars tent to be quite big in size depending on the regular language to be defined. However, all regular grammars can be reduced considerably in size by using the regular expression notations. In the other hand, a regular expression can be converted into a regular grammar by expanding it according to the meaning of the operators. Therefore, regular languages are often described by regular expressions rather than by regular grammars.

6. Regular Expressions

Regular expressions are string specifiers or pattern descriptions using a pattern description language, which is a very convenient specification language for the finite state automaton actually constructed. The characters set used in the language is a subset of the ASCII characters set. Pattern descriptions are specified by giving special meaning to certain characters called metacharacters. The following rules are used by **PCLEX** to form a regular expressions. The rules are the same as the rules used by UNIX LEX and FLEX.

1. A single character that is not a metacharacter is a regular expression matching the single character. Thus, letters, digits, and some special characters represent themselves.

For example, regular expression **A** matches the single character **A**.

2. Two regular expressions concatenated form a regular expression matching the pattern that a match of the first expression is immediately followed by a match of the second.

For example, regular expression **P** and regular expression **C** form a regular expression **PC** which matches the string **PC**.

3. Two regular expressions separated by a vertical bar ('|') form a regular expression matches either the preceding expression or the following regular expression.

For example, regular expression **RED|GREEN|BLUE** matches any of the three words..

4. Two regular expressions separated by a slash ('/') form a regular expression matches the preceding expression but only if followed by the following regular expression.

For example, regular expression **PC/LEX** matches the substring **PC** in the string **PCLEX.** It would not match string **PC** or **PCYACC**.

5. A series of regular expressions can be grouped together in a pair of parentheses ('(' and ')') form a new regular expression.

For example, regular expression **(Lind|Brian)a** matches both **Linda**, and **Briana**.

6. A period ('.') matches any single character except the **C** new line metacharacter ('**\n**').

For example, regular expression **M.** matches strings **ME** and **MY**.

7. An up arrow circumflex accent ('^') as the first character of a regular expression matches the beginning of a line.

For example, regular expression **^ME** matches the string **ME** only if it is the first two characters on the line.

8. A dollar sign ('$') as the last character of a regular expression matches the end of a line; but not the new line character itself.

For example, regular expression **ME$** matches the string **ME** only if it is the last two characters on the line.

9. A pair of square brackets ('[' and ']') enclosing a sequence of characters forms a regular expression called a <u>character class</u>, which matches any of the characters enclosed in the brackets. Metacharacter '**[**' marks the start of a character class and metacharacter '**]**' marks the end of the character class.

Inside a character class only metacharacters '**]**', '^', '-', '"', '\', '**{**', and '**}**' have special meaning. Other metacharacters lose their special meaning inside a character class except **C** escape characters starting with '\'. Use **{**, **}**, **]**, and so forth to put these characters into a character class.

For Example, regular expression **[0123456789]** matches any single decimal digit; and, regular expression **[.$]** matches a period or a dollar sign.

10. An up arrow circumflex accent ('^') as the first character of a character class makes a <u>negative character class</u>, which matches any character except the ones within the brackets.

For example, regular expression **[^0123456789]** matches any character except a digit character. And regular expression **[^0123456789]** matches any digit character or an circumflex accent.

11. A dash or hyphen ('-') inside a character class or a negative character class indicates a character range. **A '-' as the first character after the '['** matches the character '-' itself. This provides another why to put the character into the character class.

For example, regular expression **[0-9]** matches any single decimal digit; it means the same thing as regular expression **[0123456789]**. Regular expression **[-^]** matches a dash or an up arrow.

12. A quotation mark ('"') regardless inside a character class or not takes away special meaning of characters up to next quotation marks. Everything within the quotation marks is interpreted literally; metacharacters other than <u>C character escapes</u> lose their meaning within the quotation marks.

For example, regular expression "**1.5/3.0**" matches the string **1.5/3.0**.

13. A backslash ('\') regardless inside a character class or not takes away special meaning of next character. The metacharacter is used to escape metacharacters, and as part of the usual **C** character escapes.

For example, regular expression **[\t\n\]** matches a whitespace character.

14. A pair of curly braces ('{' and '}') regardless within a character class or not marks the start and the end of a macro name.

For example,

```
letter                  [a-zA-Z]
%%
{letter}      ECHO;
.                       ;
```

In the example we first define a macro name **letter** which represents a regular expression **[a-zA-Z]** in the **definition section** of the **scanner description file**. Later, in the **rule section** we refer to the regular expression by calling its macro name using **{letter}**.

15. **Numbers in a pair of braces, {number1, number2}**, form an operator which indicates how many times the previous pattern is allowed to match.

For example, regular expression **[0-9]{1,3}** matches numbers **0** to **999**.

16. **A regular expression followed by an asterisk ('*')** matches that expression repeated zero or more times. The metacharacter '*****' is a <u>closure operator</u>. A <u>closure operation</u> has higher precedence than concatenation.

For example, regular expression **[0-9][0-9]*** matches numbers consist of one or more digits.

17. **A regular expression followed by a plus ('+')**, a closure operator, matches that expression repeated one or more times.

For example, regular expression **[0-9]+** matches numbers consist of one or more digits.

18. **A regular expression followed by a question mark ('?')**, a closure operator, matches that expression repeated zero or one times.

For example, regular expression **Brian?a** matches **Briana** and **Brianna**.

The precedence of operators in a regular expression is listed from highest to lowest as following:

operator	**description**
()	parentheses grouping
[]	character class
*** + ? {}**	times the pattern is allow to match
ee	concatenation
|	either pattern
^ **$**	beginning and end of line

7. PCLEX Terminology -- A Short Review

PCLEX is a <u>scanner generator</u>, a program that assists in writing lexical scanners. The scanner is a <u>finite-state automaton</u> or <u>finite state machine</u> (FSM). **PCLEX** takes an input <u>scanner description file</u> comprised of regular expressions and associated **C** codes (actions), then builds a recognizer program which executes the **C** code when a certain string is recognized. **PCLEX** builds the lexical scanner by translating the target language's lexical syntax description in regular expressions into a **C** program that recognizes the symbols or tokens of the target language. The source language of **PCLEX** is the scanner description language (SDL). The object language of **PCLEX** is the programming language **C**. Source programs written in SDL are called scanner description programs (SDP's). Files of SDP are called scanner description files (SDF's). The function of **PCLEX** is to translate a GDF into a **C** file that defines a function. By convention, the name of the function is **yylex()**. A target language in a lexical analysis process is the language described by the scanner description program.

A SDF is made up of three sections, a **definition section**, a **rule section**, and a **user subroutine section**, in that order. Any or all of the three sections can be empty. If the user subroutine section is empty, the second "**%%**" **delimiter** can be omitted. The first "**%%**" delimiter can not be omitted.

V. GETTING STARTED -- OUR FIRST EXAMPLE

This chapter gives you an idea of how to use **PCLEX** with a standalone scanner program. Many simple text processing and statistics gathering programs can be quickly written this way with the aid of **PCLEX**. It is assumed that you are already familiar with MS-DOS and the **C** programming language. You need an MS-DOS personal computer with **ABRAXAS PCLEX** and a **C** compiler installed to build and run this example. A programming editor is helpful for making program modifications and experiments.

The basics of the program development process is similar for all standalone **PCLEX** programs. This chapter gives an overview of that process. The example program used in this chapter counts the bytes, words, and lines in a text file. How to build and run the program will be shown and explained. Later chapters show more complicated examples with more involved build procedures.

1. Scanner Description File for Word Count Program

The following is the listing of the SDF for the word count program, **WC.L**. For reference, line numbers are added to the listing.

```
001: /*
002:  * WC.L - simple standalone PCLEX application.
003: */
004:
005: %{
006: long nchar = 0;            /* # of characters */
007: long nword = 0;            /* # of words      */
008: long nline = 0;            /* # of lines      */
009: %}
010:
011: %%
012:
013: \n             nchar += 2;   ++nline;
014:
015: [^ \t\n]+      ++nword;      nchar += yyleng;
016:
017: .              ++nchar;
018:
019: %%
020:
021: main()
022: {
023:    yylex();
024:    printf("%d\t%d\t%d\n", nchar, nword, nline);
025:    exit(0);
026: }
```

This example, though small, exhibits the typical structure of a **PCLEX** scanner description. **Lines 001 through 010** form the **definition section**, where needed header files are included, global variables declared, and names defined. **Lines 012 through 018** make up the **rule section** where the input patterns to match and their corresponding actions are defined. **Lines 020 through 026** are the **user subroutine section** with the necessary support functions written in **C**. As illustrated by this example, a scanner description file is made up of three section: a definition section, a rule section, and a user subroutine section. Any or all of the three sections can be empty.

Lines 001 through 003 are a comment. SDL comments look the same as **C** comments and are passed through to the output file intact. The rules for SDL comment placement are not quite the same as in **C**. They are explained in detail in **Chapter X, section 5.3**.

The symbol pairs, "**%{**" and "**%}**", on **lines 005 and 009** are delimiters used in the definition section to bracket **C** code, such as preprocessor directives, global type and structure definitions, and global variable declarations. In this example, there are three variable declarations. **PCLEX** does not look at the code inside these delimiters, it is passed through to the output scanner source file intact. The code is placed on the top of the output file so that other parts of the scanner can refer to the data definitions contained in it.

Line 011 is the delimiter ("**%%**"), on a line by itself, separating the definition section from the rule section.

Line 013 says: when an end of line is reached ('**\n**' is shorthand for new line), add two to the character counter ("**nchar**") and add one to the line counter ("**nline**"). The end of line in MS-DOS text files is marked by two characters, a carriage return and a line feed character. The pattern in a rule is everything before the first whitespace (blanks and tabs), in this case the '****' and '**n**' characters. Everything from the whitespace to the end of the line is the action part of the rule. Each **action** is a section of **C** code executed when its pattern is recognized in the scanner input. Actions can be empty (a null statement in **C** language, called an **empty action**.) Rules normally are just one line long. How to extend the action over several lines is explained in a later chapter (**Chapter VII**, **section 4**).

Line 015 says: when a string of non-whitespace characters are found, increment the word counter ("**nword**") and add the length of the matched text ("**yyleng**") to the character counter. The pattern matches one or more ('**+**' operator) occurrences of the class of characters (square brackets, '**[**' and '**]**', enclose character classes) that includes everything except (an initial '**^**' operator complements or reverses the contents of a character class) a space, a tab ('**\t**' is shorthand for tabs), and ends of lines.

Line 017 is the final rule in the rule section and says: for any character not otherwise matched ('**.**' operator matches any single character), increment the character counter. The blank lines between rules are for readability, they are not required.

The second "**%%**" delimiter on **line 019** separates the rule section from the user subroutine section. Everything in the user subroutine section is passed intact to the output file of **PCLEX**. In this example, the main function calls "**yylex()**", the scanner function generated by **PCLEX**, prints the character, word, and line counts, and exits. The following discussion will help you understand how to combine what **PCLEX** produces with the user subroutines written by the programmer to make a complete **C** program.

PCLEX generates **C** code for a function, "**yylex()**", and data tables that together read the input, divide it into matches of the patterns, and execute the corresponding actions. Technically, the "**yylex()**" function is a table-driven interpreter that simulates a Deterministic Finite State Machine (DFSM). DFSM are discussed in more detail in **Chapter IX**. The rest of the program must include a "**main()**" function, do any program setup and cleanup, and do any additional processing needed.

2. Building the Executable File

To invoke **PCLEX** on the scanner description file **WC.L**, issue the following command:

```
C>pclex wc.l
```

The result of this operation is a **C** program. A file with the name **WC.C** will be created in the current directory, which is a **C** program for the **Word Count** program. To build the executable version of **WC**, invoke the **C** compiler as follows (the example assumes the Microsoft **C** compilers):

```
C>cl wc.c
```

3. Sample Session

After the executable version of **WC** is built successfully (in our example, we have a **WC.EXE** in the current directory), we can use it to check the size of **WC.L**. The following is a sample session of **WC** at work:

```
C>wc <wc.l
398     64      26
```

This example, though simple, contains almost all of the parts of building a program with **PCLEX**.

Note, if you take **WC.EXE** and type

```
C>wc < wc.exe
```

This will lock up the machine or scanner because the executable file is made up by **8**-bit ASCII characters. To build an eight bit scanner use the command

```
C>pclex -8 wc.l > wc.c
```

then type

```
C>cl wc.c
```

now type

```
C>wc < wc.exe
```

and note the machine will not jam.

VI. INTEGRATING PCYACC AND PCLEX

1. PCYACC -- A Parser Generator

PCYACC is a compiler-writer, or a parser generator, a program that assists in writing compilers. A parser is often the front end portion of a compiler. It is a push-down automaton, or a stack machine, consisting of a **stack** holding current states, a **transition matrix** determining next state according to the current state and next input symbol, a **table of user defined actions** executed at certain points in the grammar analyzing, and finally an **interpreter** managing the execution.

PCYACC translates a context-free grammar into a **C** function that recognizes programs written in the target language defined by the grammar. 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) which describes the grammar syntax of the target language.

2. Parser and Scanner

The function of **PCYACC** is to translate a GDF into a **C** file that defines a function. By convention, the name of the function is **yyparse()**; which calls repeatedly on a lexical analyzer function **yylex()** to read input and returns zero or one indicating whether a sentence was presented in the target language according to the grammar syntax of the language.

A **PCYACC** generated parser calls the **PCLEX** generated scanner "**yylex()**" for each token it parses. The parser passes no arguments to "**yylex()**" and expects an integer return value. The return value is either a character widened to an integer or one of the **%token** values **#define**d in the **C** header file generated by **PCYACC** (see next section). Note that while **PCYACC** allows periods in **%token** names, the **C** compiler will not. Periods are okay in non-terminal names since the **C** compiler will never see them.

A typical scanner action ends with either

```
return yytext[0]
```

or

```
return TOKEN;
```

where "**yytext**" is a internal defined character pointer pointing to the matched input text, and "**TOKEN**" , representing a specific input stream, is **#define**d in the **C** header file generated by **PCYACC**.

Further examples of **PCLEX/PCYACC** interaction are the **DATES** and **ANSI C Syntax Analyzer** programs explained in the next section.

3. C Header File Generated by PCYACC

Using command line option **-d** or **-D**, **PCYACC** will generated a header file as well as the **C** parser file. The header file is used primarily by the lexical analysis routine **yylex()**. **PCYACC** will enumerates all of the tokens declared in the grammar, and these enumerated values are used as messages between **yyparse()** and **yylex()**.

The header file declares and defines all the global variables and macros shared by the syntax parser and the lexical scanner. Usually it will define the parser stack type **YYSTYPE**, declare the intercommunication variable **yylval**, and define the terminal symbols for the scanner and the parser. The definitions generated by **PCYACC** are used globally at parse time unless your **yylex()** routine is local to the grammar.

The **-d** switch tells **PCYACC** to produce the header file using the default file name **yytab.h**. The **-D<hf>** switch produces the header file using "**hf**" as the name. If no **<hf>** is provided, **PCYACC** will use the basename of the grammar description file with an extension "**.h**". For more information regarding to **PCYACC** command line options, see **PCYACC Users' Manual**.

4. "yylval" and "YYSTYPE"

The conventional way to pass additional information from the scanner actions to the parser actions is through the variable "**yylval**". Its type is "**YYSTYPE**", the same as the semantic stack maintained by the parser and accessed through the "**$$**", "**$1**", "**$2**", etc. variables in the parser actions. For more information regarding to **PCYACC** symbols, **"$$", "$1", "$2"**, etc. see **PCYACC Users' Manual**.

The default type of **PCYACC** stack is an **int**. It can be changed by the user in two ways. The first and easiest is to use **PCYACC** keyword **<u>%union</u>**. For example, to use the value stack to handle three kinds of values, integer numbers, floating point numbers, and identifiers, the following union definition can be added to the declaration section of the grammar description file:

```
%union {
        int   i;
        float r;
        char *s;
}
%token DDD
%%
```

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   i;
        float r;
        char *s;
} YYSTYPE;
%}
%token DDD
```

PCYACC will translate the first declaration style into the second declaration style, which actually appears at the very begin of the generated **C** code parser:

```
typedef union {
        int   i;
        float r;
        char *s;
} YYSTYPE;
extern YYSTYPE yylval;
#define DDD 257
```

Note, **PCYACC** **typedef**s the parser stack type, **YYSTYPE**, according to the union definition in the declaration section of the grammar description file, declares the intercommunication variable **yyval** to be of type **YYSTYPE**, and enumerates the token, **DDD**, declared in the grammar description file.

With the **-d** or **-D** switch, **PCYACC** will make a copy of this code section from the generated **C** code parser to the header file specified by the switch (see last section).

VII. DATES -- A SECOND EXAMPLE

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

A general guideline for project development using **PCLEX** and **PCYACC** is a seven step process, as follows:

1) define the problem
2) develop a language to express the problem and/or solution
3) separate the source language into lexical and syntactic parts
4) write the **PCLEX** scanner description
5) write the **PCYACC** parser description
6) write the auxiliary **C** code
7) build the executable program

The **DATES** program is a simple example of **PCLEX** and **PCYACC** working together. The advanced error processing from **PCYACC** is included in this example. The sources are in **\DATES** directory on the **EXAMPLES** disk.

1. Problem Statement

The problem is recognizing dates after January 1, 0000, and converting them to an internal form. There are a variety of calendars in use throughout the world. To avoid complicating the problem too much, only the **Gregorian Calendar** will be handled. The **Gregorian Calendar** (named after **Pope Gregory**, who oversaw its development) is currently in use in North America, South America, and Europe. To allow future extensions to other calendars, the number of days since January 1, 0000, is used for the internal form.

2. Developing the Source Language

While everyone who uses the **Gregorian Calendar** agrees on the number of months and the number of days in each month, there is not agreement on how to write a date. The U.S. writes dates with the month first, then the day and year. Europe writes the day first, then the month and year. When the month is spelled out, the spelling differs from language to language, though they are often recognizable. For this program, only English months are recognized, though both orders are handled properly.

In the U.S., the short form is punctuated with slashes, for example: **12/31/93**, the last day of 1993. The long form is the month, the day, a comma, and the year, for example: **December 31, 1993**. In the short form, the century is almost always left off the year. Two digit years will be assumed to be in this century (this convention makes short form clumsy for dates in the first century B.C). The year will always be assumed to be complete in the long form.

In Europe, the short form is punctuated by periods, for example, **31.12.93** is the last day of 1993. The same date in European long form is: **31 December 1993**.

In both forms, the months can be abbreviated to the first three letters. The month is always capitalized. This restricted set of date conventions allows one program to handle dates without explicitly specifying the form used. There is sufficient difference between the forms that a computer program can tell which is used.

3. Separate the Lexical and Syntactic Parts

In isolation, the line between lexical and syntax analysis can be drawn in a number of places. For consistency with the usual practice in compilers, the **scanner** will handle whitespace, full numbers (not just digits), punctuation, and spelled-out months. The **parser** will handle form and correctness recognition. **Auxiliary C code** will handle two kinds of things. The first task of the auxiliary **C** code is handling date correctness checking, month lookup, ASCII to binary conversion, and output. This work is tightly associated with the parser. The second task of the auxiliary **C** code is some how independent from the parser. It handles the real calculation and is therefore more project dependent. Usually the auxiliary **C** code that strongly associated with the parser is put into the last section of the grammar description file. And the **C** functions that are parser independent are coded in other files.

4. Write the PCLEX Scanner Description

In the first example shown in **Chapter V**, the scanner description contained the entire program. In this example, the SDF contains just part of the program. The parser with some auxiliary **C** functions is in the grammar description file (GDF). And the auxiliary **C** function doing calculation is in another **C** file.

```
/*
 * LEX.L - lexical analyzer for DATES program
*/
%{
#include   <stdlib.h>               /* atoi() */
#include   "dates.h"     /* token definitions */
extern int  yylval;      /* defined by PCYACC */
int         yylineno; /* inputed line counter */
#define MON(x) { yylval = x;  return MONTH; }
%}

%%

Jan("."|uary)?     MON(1);          /* months */
Feb("."|ruary)?    MON(2);
Mar("."|ch)?       MON(3);
Apr("."|il)?       MON(4);
May                MON(5);
Jun("."|e)?        MON(6);
```

```
020: Jul("."|y)?        MON(7);
021: Aug("."|ust)?      MON(8);
022: Sep("."|tember)?   MON(9);
023: Oct("."|ober)?     MON(10);
024: Nov("."|ember)?    MON(11);
025: Dec("."|ember)?    MON(12);
026:
027: [0-9]+     {
028:                    yylval = atoi(yytext);
029:                    return NUMBER;
030:            }
031: [ \t]              ;    /* discard whitespace */
032: \n         {
033:                    ++yylineno;
034:                    return '\n';
035:            }
036: .                  return yytext[0];
```

Lines 005 and 006 include the necessary header files: "**stdlib.h**" for the prototype of "**atoi()**", and "**dates.h**" for the **#define**s for the token labels, "**MONTH**" and "**NUMBER**". Header file "**dates.h**" can be produced by invoking **PCYACC** with **-d** switch.

Line 007 is for "**yylval**", a variable used to pass additional information to the parser. The variable is for communication between the syntax parser and the lexical analyzer. It is predefined as a global variable by **PCYACC** in **yyparse()**. Variable **yylval** has the same type as the **PCYACC** stack, **YYSTYPE**. **Line 007** listed here is for better explanation. It could be omitted since in the header file generated by **PCYACC, yylval** is always declared to be:

    extern YYSTYPE yylval;

right after the definition of **YYSTYPE**. In our example only integer value stack is used. Therefore, we don't need to redefine the parser stack type **YYSTEPE**.

Line 008 defines the variable **yylineno** used to count the lines scanned of the input.

Line 009 defines a macro used in the action for all the months. The definition saves typing (nice!) and ensures that all months behave the same (important!).

The "**%%**" delimiter standing along on **line 012** separates the definition section from the rule section.

Lines 014 through 025 handle the months and their abbreviations, both with and without a period. The period has a special meaning in patterns and must be enclosed in double quotes to match itself. The question mark after the right parenthesis indicates that the part of the pattern inside the parentheses is optional. The parentheses enclose two alternatives separated by the vertical bar ('|'). The pattern "**Jan("."|uary)?**" is equivalent to the three patterns: "**Jan**", "**Jan"."**", and "**January**".

The pattern on **line 027** matches integers. The action spans **lines 027 through 030**. Multi-line actions are enclosed in braces to form a **C** compound statement. The current matched text is pointed by the **PCLEX** predefined character pointer "**yytext**". The binary equivalent is assigned to "**yylval**", a **PCYACC** predefined global variable, for use by the parser.

Line 031 matches and discards blanks and tabs. Every action so far ends with a "**return**" statement. "**yylex()**" returns a token with each call by the parser. In the **WC** example, all processing was controlled by the scanner. In this example, the parser is the center of control. If the action for a pattern does not return to the parser, scanning continues and the next pattern match is found. The **empty action** for the "**[\t]**" pattern does nothing and the parser does not see whitespace. This idiom is a common one. Whitespace, comments, and ends of line are ignored in the grammar of most modern programming languages.

Unlike modern programming languages, this example program expects one date per line. Ends of lines are not ignored: the line counter, "**yylineno**", is updated and the scanner returns a token to the parser.

Line 036 is a catchall. The pattern matches any other single character and returns it to the parser. This is also a common idiom to pass off handling invalid characters and single character operators and punctuation. The parser will handle all of these directly.

5. Write the PCYACC Parser Description

The parser for the **DATES** program is built with **ABRAXAS PCYACC**. With minor changes, it should work with YACC or any of its clones and work-alikes. The **PCYACC** specific parts are noted in the explanation below.

```
001: /*
002:  * DATES.Y:  grammar for U.S. and European dates
003: */
004: %{
005: #include <stdio.h>          /* for fprintf() */
006: %}
007:
008: %token    MONTH  NUMBER
                    /* will be defined in dates.h */

009: %start    input
010:
011: %%
012:
013: input
014:  :                     /* empty file is legal */
015:  | input  date   '\n'
016:  | input  error  '\n'           { yyerrok; }
017:  ;
018:
```

```
date
/*
 * U.S. text form
*/
 : MONTH  day  ','  year   { date($1, $2, $4); }

/*
 * European text form
*/
 | day  MONTH  year        { date($2, $1, $3); }

/*
 * U.S. short form
*/
 | month '/' day '/' year  { date($1, $3, $5); }

/*
 * European short form
*/
 | day '.' month '.' year  { date($3, $1, $5); }
 ;

day
 : NUMBER
 ;

month
 : NUMBER
 ;

year
 : NUMBER
 ;

%%

extern int  yylineno;     /* defined in lex.l */
extern int  yyparse(void);
extern int  days(int, int, int);

/*
 * main(): main routine for the program
*/
main()
{
   yylineno = 1;
   printf("\nInput a date:         ");
   yyparse();
}

/*
 * date(): generic date action routine
```

```
071: */
072: static void date(int month, int day, int year)
073: {
074:    printf("%d/%d/%d\n", month, day, year);
075:    days(month, day, year);
076: }
077:
078: /*
079:  * yyerror(): error reporting routine
080: */
081: void yyerror(char *s)
082: {
083:    fprintf( stderr, "%s\n", s);
084: }
```

Grammar Definition Files (GDF), like scanner definition files, are divided into three sections by "**%%**" lines. For GDF, the sections are: the declaration section (**lines 001 through 010**), the grammar rule section (**lines 012 through 052**), and the program section (**lines 054 through 084**). The declaration section contains comments, in-line **C** code within "**%{**" and "**%}**" lines, token declarations, and the goal symbol declaration. More complex grammars can have additional types of declarations.

The file "**stdio.h**" **#include**d on **line 005** defines the prototype for "**printf()**" which is used in "**main()**", "**date()**", and "**fprintf()**" used in "**yyerror()**".

The "**%token**" declaration on **line 008** introduces the two tokens (terminal symbols), **MONTH** and **NUMBER**, used in the parser that are not character literals. **PCYACC** will sequentially assign values, starting from 257, to these tokens and **#define** them in the **C** header file, "**dates.h**". The **C** header file is **#include**d in the scanner description file for these constants. The lexical scanner defined by the scanner description file, **LEX.L**, will return the corresponding token values to the syntax parser upon the lexical analyzing (see **lines 014 through 025** and **line 029** in the **LEX.L** file of last section.). Integer values 1 to 256 are reserved for the ASCII characters. In the case that the lexical scanner returns a single symbol to the parser, it just returns the ASCII value of the symbol.

The goal symbol, "**input**", for the grammar is declared on **line 009**. The declaration on **line 009** is strictly speaking unnecessary. Token declarations are required by **PCYACC** to allow it to check the grammar for consistency, similar to requiring a variable to be declared before use. The goal symbol defaults to the first non-terminal that does not appear on the right hand side of a production. It is explicitly declared for safety.

The grammar rule section consists of a number of grammar rules. Each grammar rule has a **left hand side (LHS)**, which is a nonterminal symbol, a '**:**' operator separating the left hand side and the right hand side, a **right hand side (RHS)**, which is a sequence of zero or more grammar symbols, and a semicolon indicating the end of the rule. An action coding in **C** language 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.

Actions are **C** language statements enclosed in curly brackets. Grammar symbols (both terminals and nonterminals) 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 conventions **$$** represents the value of the **LHS** nonterminal 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.

The "**error**" on **line 016** is a special terminal symbol predefined by **PCYACC**. It can be used in a GDF like a terminal symbol. The "**error**" symbol is generated internally. The parser will take "**error**" to be the next terminal symbol if the actual next terminal symbol, produced by a call to the lexical scanner, leads to an error operation for the current state.

The **PCYACC** predefined macro "**yyerrok**" on **line 016** tells the parser to return to the normal state when it take "**error**" to be the current terminal symbol.

The main routine, **main()** runs from **line 062** to **line 067**. It first initiates the line number counter **yylineno** to be **1** and guides the user input a date. The most important task of main routine is calling **yyparse()**, the LR parser generated by **PCYACC**. **yyparse()** is called to work on the inputted date, it intern calls **yylex()** asking for the input data. **yyparse()** returns a zero if the parsing process is successful. A non zero value is returned if an error situation occurs during the parsing.

The second function, **date()**, prints the inputted data and invokes **days()** to do the computation. The parser will call **date()** when it recognizes the inputted stream to be a date. See the action parts on **lines 023, 028, 033, and 038** of **DATES.Y** file.

The third function, **yyerror()**, prints an error message on the standard error device, which, under MSDOS, will be the working window. **yyerror()** is the standard **PCYACC** error routine. **yyparse()** will call **yyerror()** whenever it detects a syntax error. In our example, **yyerror()** is oversimplified. In some cases a more sophisticated error handling routine is necessary to produce comprehensive diagnostic messages and error recoveries.

6. Write the Auxiliary C Code

The auxiliary function **days()** will actually perform the computation.

```
001: /*
002:  * DAYS.C  -- routines do the calculation
003: */
004: #include <stdio.h>            /* for printf() */
005:
006: extern int yylineno;      /* defined in lex.l */
007:                   /* initiated by main(), in  */
008:                   /* dates.y, program section */
009:
010: extern int yyparse(void);
```

```
                      /* generated by PCYACC  */
                      /* according to dates.y */

int regular_year[]
                 { 0, 31, 28, 31, 30, 31, 30,
                      31, 31, 30, 31, 30, 31, };

int leap_year[]
                 { 0, 31, 29, 31, 30, 31, 30,
                      31, 31, 30, 31, 30, 31, };

/*
 *  days(): calculating routine
*/
days( int month, int day, int year )
{
    int  *yearis;

    /*
     * check the year
    */
    yearis = check_year(year) ?
                    leap_year : regular_year;

    /*
     * check the month
    */
    if ( month < 1 || month > 12 )
    {
       printf("month should be in the range of
               1-12\n");
       return 0;
   }

   /*
    * check the day
   */
   if ( day < 1 || day > yearis[month])
   {
       printf("day of the month should be in the
               range of 1-%d\n", yearis[month]);
       return 0;
   }

   /*
    * calculate the number of days from
    * January 1, 0000
   */
   for (month; month > 1; month--)
       day += yearis[month];

   if ( year > 0)
   {
       day += year * 365;
       day += year/4 - year/100 + year/400;
   }

   printf("The date is %d days form
            Jan. 1, 0000\n\n", day-1);

   printf("Input another date or CTRL-C to exit
            the program\n\n");

   return 1;
}

/*
 * check_year(): leap year checking routine
*/
check_year(int year)
{
   if (year % 4 != 0)
       return 0;
   if (year % 100 != 0)
       return 0;
   if (year % 400 != 0)
       return 0;
   return 1;
}
```

7. Build the Program

To obtain the executable file of our example we need to do the following:

a) generating the parser **C** source file, **DATES.C**, and the header file **DATES.H**, which is created by the **-D** switch, using **PCYACC**:

C>*pcyacc -D dates.y*

b) generating the lexical scanner **C** source file, **LEX.C**, using **PCLEX**

C>*pclex lex.l*

c) compiling and linking the parser **C** source file **DATES.C**, the scanner **C** source file **LEX.C**, and the auxiliary **C** code file **DAYS.C** assuming that the Microsoft Visual **C++** compiler is used

C>*cl -Fedays dates.c lex.c days.c*

The C compiler switch -*Fedays* gives the name *days.exe* to the executable file.

Now we get the executable file called **DAYS.EXE**.

8. Build the Program with MAKE

Most of the programming environment provide the **MAKE** utility which provides a convenient way for project development relating recompilation process. When you invoke the **MAKE** program, by determining which files depend on others in a project, **MAKE** automatically execute the commands needed to update the project when any project file has changed. The **makefile** of our example is listed as follows.

```
001: # DOS makefile for DAYS
002:
003: LD = cl  -Fedays
004: LDFLAGS =
005: CFLAGS = -c
006: YFLAGS = -D
007: SRCS =    days.c dates.c lex.c
008: OBJS =    days.obj dates.obj lex.obj
009:
010: days.exe: $(OBJS)
011:      $(LD) $(LDFLAGS) $(OBJS)
012:
013: .c.obj:
014:      cl $(CFLAGS) $*.c
015:
016: date.c: date.y
017:      pcyacc $(YFLAGS) date.y
018:
019: lex.c: lex.l
020:      pclex lex.l
```

Line 001 is a comment line. Preceding a line with a number sign ('**#**') makes the line to be a comment line.

Lines 003 through 008 define couple macros which allow us to do text replacements throughout the makefile. One can define a macro with

macroname = string

The ***string*** can be any string, including a null string. In our makefile, **line 004** defines a macro, **LDFLAGS**, to be a null string.

A macro can be invoked by enclosing its name in parentheses preceded by a dollar sign ('**$**'). When **MAKE** runs, it replaces every invoked macroname with its corresponding string.

The Microsoft Visual **C++** compiler option ***-c*** tells the compiler to compile the **C** source files listed on the line, creating object files, but not to link the object files.

The **PCYACC** switch ***-D*** tells **PCYACC** to produce a **C** header file using the basename of the grammar description file, ***dates***. The generated header file **DATES.H** will be used by **yylex()**, see **line 006** in the file **LEX.L** in **section 4**.

VIII. ANSI C SYNTAX ANALYZER--A THIRD EXAMPLE

In this example we are trying to show the users how to build a **language engine** using **PCLEX** and **PCYACC**. We will discuss both programming languages, the utilities and the cooperation's in more detail.

The **ANSI C** lexical scanner, **LEX.L**, which is in the **\ANSIC** directory of the **PCLEX [DOS-OS/2] Toolkit Disk**, is a "plug compatible" replacement for the hand coded lexical scanner **LEX.C** in **\ANSIC** directory of the **PCYACC Professional Upgrade Disk**. The rest of the code is the same.

1. Problem Statement

The problem is building an **ANSI C** syntax analyzer which reads input **C** source files, checks the syntax of the statements, and gives a report of the checking.

2. Developing the Source Language

In this example the source language is **ANSI C**. We need to support all features of the language.

The precise definition of escape sequences is different for **PCLEX**, the older **Kernighan and Ritchie C** standard **(K&R)**, and the **ANSI C** standard. The **PCLEX** escape sequence definition is:

```
\\(.|0[0-7]{1,3})
```

Most of the older **(K&R) C** compilers use:

```
\\(.|\n|0[0-7]{0,3})
```

This definition allows escaped ends of lines. They are used for strings that span line boundaries. **PCLEX** does not allow this.

The **ANSI C** escape sequence definition is a backslash followed by either

1) one of the letters "**abfnrtv**",
   a single quote
   a double quote
   a question mark, or
   another backslash, or

2) one to three octal digits, or

3) a lower case "**x**" followed by one or more hexadecimal digits.

The above definition can be described by a regular expression:

```
\\([abfnrtv'"?\\]|[0-7]{1,3}|x[0-9a-fA-F]+)
```

PCLEX does not allow "**\0**". The string "**\x007**" is two bytes long for **ANSI C** (equivalent to "**\a**", the **BELL** character and a terminating **NUL**) and five bytes long for **PCLEX** and **K&R C** (equivalent to "**x007**").

3. Separate the Lexical and Syntactic Parts

The implementation of the **ANSI C** syntax analyzer divides into four files: **LEX.L**, **ANSIC.Y**, **MAIN.C** and **ERR_SKEL.C**. The lexical scanner generated by **PCLEX** from **LEX.L** will handle keywords, identifiers, full numbers, one- and two-symbol operators, letters, whitespace, and punctuation. The syntactic parser generated by **PCYACC** from **ANSIC.Y** will handle statement and correctness recognition. Auxiliary **C** function **main()** in **MAIN.C** file will handle Input/Output task. Auxiliary error handling routines in **ERR_SKEL.C** file provide a general way for a language engine to handle syntax errors in the input source file.

Two levels of error handling are involved in a language engine: the external level is the error reporting for the input source file for the language engine; and the internal level involves what action need to be performed when syntax errors are occurred and how far the parsing should go. The external level will be discussed in detail in this example. The internal error handling routines are in the **ERR_SKEL.C** file provided in the diskette. **ERR_SKEL.C** is quite general to any language engine. However the discussion of **ERR_SKEL.C** file requires a deep understanding of **PCYACC**. Readers please see the **PCYACC** user's manual for more information regarding to the file.

4. Write the PCLEX Scanner Description

The scanner description file **LEX.L** is listed as following :

```
%{

/*
=====================================================

  lex.l: lexical analyzer for ANSI C parser
 Version 2.0
 By Yan Luo

 PCYACC(R) is a software product of
           ABRAXAS SOFTWARE INC.
 Copyright(C) 1986-1997 by ABRAXAS SOFTWARE INC.

=====================================================
*/

#include <stdio.h>
/* FILE, fprintf(), fputc(),sprintf(), stderr */

#include <string.h>          /* strcmp(), strlen() */
021: #include <ctype.h>         /* isascii(), isprint() */
022: #include "ansic.h"                 /* token values */
023:
024: #define DIM(a) (sizeof(a)/sizeof((a)[0]))
025:
026: extern char *yytext;/* pclex's predefined pointer  */
027:                     /* to the current token's text */
028:
029: extern FILE *yyin;   /* pointer to the input file, */
030:                               /* defined in main.c */
031:
032: extern int error_count;          /* error counter, */
033:                           /* defined in err_skel.c */
034:
035: int  yylineno  = -1;        /* current line number */
036:
037: %}
038:
039: letter    [a-zA-Z_]
040: digit     [0-9]
041: esc       \\([abfnrtv"'?\\]|[0-7]{1,3}|x[0-9a-fA-F]+)
042: alphanum  [a-zA-Z_0-9]
043: blank     [ \t]
044: other     .
045:
046: %x        COMMENT
047:
```

The definition section spans from **line 001** to **line 047**. **Lines 017 to 022 #include**s the needed header files. Utilities defined in these headers can be used in the action parts of the rule section and the user subroutine section.

Line 024 declares a macro, "**DIM**", used for calculating the number of items in a table.

Line 029 declares a FILE pointer to the input source file of the language engine. In the last example, **DATES**, the program can only take inputs from the standard input device. The pointer allows the language engine to parse **C** source files.

Line 032 declares the syntax error counter of the language engine counting the syntax errors in the input **C** source file of the language engine.

Line 035 declares a counter counting the input source file line number for error reporting routines. Before the parsing process it is assigned to be -1.

Lines 039 through 044 define several regular expression macros. Macro names are substitutions of the corresponding patterns. The patterns can be referred in the rule section with the macro names in braces, for example, "**{letter}**".

Line 039 defines "**letter**" to be any letter in English. **Line 040** defines "**digit**" to be any decimal digit.

Line 041 defines "**esc**" to be a backslash followed by either

1) one of the letters "**abfnrtv**",
   a single quote
   a double quote
   a question mark, or
   another backslash, or

2) one to three octal digits, or

3) a lower case "**x**" followed by one or more hexadecimal digits.

Line 042 defines "**alphanum**" to be any letter, the underscore character ('_'), or any digit.

Line 043 defines "**blank**" to be a space or a tab.

The regular expression operator "**.**" on **line 044**, the definition of "**other**", matches any character except the end of a line.

Line 046 declares an exclusive start condition, "**COMMENT**". A detailed explanation of exclusive start condition is in **Chapter X**, **section 5.3**.

The "**%%**" delimiter standing along on **line 048** separates the definition section from the rule section.

The rule section spans from **line 049** to **096**:

```
048: %%
049:
050: ^{blank}*"#".*$          ;
051:                 /* ignore preprocessor directives */
052:
053: "||"                     return OROR;
054: "&&"                     return ANDAND;
055: "=="                     return EQU;
056: "!="                     return NEQ;
057: "<="                     return LEQ;
058: ">="                     return GEQ;
059: "<<"                     return SHL;
060: ">>"                     return SHR;
061: "++"                     return ADDADD;
062: "--"                     return SUBSUB;
063: "->"                     return PTR;
064: "+="                     return ADDEQ;
065: "-="                     return SUBEQ;
066: "*="                     return MULEQ;
067: "/="                     return DIVEQ;
068: "%="                     return MODEQ;
069: "<<="                    return SHLEQ;
070: ">>="                    return SHREQ;
```

```
071: "&&="                      return ANDEQ;
072: "^="                       return XOREQ;
073: "|="                       return IOREQ;
074:
075: {letter}{alphanum}*        return binary_search();
076:
077: {digit}+[uUlL]*            return INTEGER_CONSTANT;
078:
079: {digit}+\.{digit}*((e|E)("+"|"-"){digit}+)?
                                return FLOAT_CONSTANT;

080: \.{digit}+((e|E)("+"|"-"){digit}+)?
                                return FLOAT_CONSTANT;
081:
082: L?\'([^'\\\n]|{esc})*\'   return CHARACTER_CONSTANT;
083: L?\"([^"\\\n]|{esc})*\"   return STRING;
084:
085: {blank}+                   ;
086: \n                         ++yylineno;
087: "/*"                       BEGIN(COMMENT);
088: <COMMENT>"*/"              BEGIN(0);
089: <COMMENT>[^*\n]+           ;
090:                    /* breaks comments into lines    */
091:                    /* so they won't overflow buffer */
092:
093: <COMMENT>\n                ++yylineno;
094: <COMMENT>"*"               ;
095: {other}                    return yytext[0];
096:
```

The first pattern in the rule section, **line 050**, matches **preprocessor directives**. They are discarded by the scanner using an **empty action**. Regular expression macro "**blank**" was defined to be a space or a tab in the definition section of the file (**line 043**). Whenever a regular expression macro name appears within braces in the rule section, **PCLEX** substitutes it (and the braces) with its definition.

The next block of patterns, **line 053 through line 073**, match the **C multi-character operators**. The returned token macros are defined in the **ANSIC.H** file which is generated by **PCYACC** according to the **ANSIC.Y** file.

The pattern on the **line 075**, "**{letter}{alphanum}***" matches both **identifiers and keywords in C**. The function **binary_search()** defined in the user subroutine section determines which.

The next three rules on **lines 077 through 080** handles **numeric constants in C**. Integers are simple. Floating point numbers are fairly simple. The need to quote or escape all the non-text characters obscures the underlying simplicity. Basically, a floating point number is any number with a decimal point in it and a possible exponent after it. Two rules are needed to make sure that a decimal point all by itself is not a number.

The patterns for **character literals** and **quoted strings** on **line 082** and **line 083** follow. The **ANSI C** standard makes a nod toward the multi-national nature of computer use with locale specific characters. Preceding a character literal or string with a capital "**L**" allows locale specific characters. Chinese, Japanese, Korean, and some other countries use **ideographic characters**. Currently, **double byte character set** (**DBCS**) system or the **Unicode Standard** is used to create coded character sets for such languages. Two bytes (16-bit) are used to represent each character in both systems. The hexadecimal escape sequences have no limit on the number of digits and can be more than a byte long. In our example, the semantics of multi-byte characters is not handled, but the syntax is checked.

The obvious pattern for a quoted string is:

```
\".*\"
```

This is short, elegant, clear, and unfortunately, doesn't work. If there is more than one string on a line, this pattern matches both of the strings and everything in between (remember, the longest match is used). The correct pattern is:

```
\"([^"\\\n]|{esc})*\"
```

This pattern excludes quotes and line boundaries from strings and handles escape sequences properly. The escape sequence macro is overkill. A simpler regular expression would allow all valid strings without trying to bring out the semantics. A simpler pattern that is adequate is:

```
\"([^"\\\n]|\\\")*\"
```

Lines 084 and 085 handle white spaces. The scanner discards blanks and tabs. Ends of lines are added to the line number, "**yylineno**" and otherwise ignored. Comments are also discarded. Ends of lines within comments are counted. Any otherwise unmatched character is passed directly to the parser.

A detailed explanation of the comment rules, **lines 088 to 094**, is in the section on exclusive start conditions (**Chapter X**, **section 5.3**).

Line 095 handles the situation other than those discussed above.

The **LEX.L** scanner illustrates the techniques of recognizing the kinds of tokens you will encounter in most programming languages: quoted literals, numeric literals, comments, single and multi-character operators, identifiers, and keywords.

The "**%%**" delimiter standing along on **line 097** separates the rule section from the user subroutine section.

The user subroutine section spans from **line 098** to **362**. The first function in this section, **binary_search()**, determines whether a name is an identifier in the **C** source file or a reserved keyword in the **C** language. Other functions in this section are error reporting routines for the language engine.

```
097: %%
098:
```

```
/*
 * reserved word table
*/
static const struct
{
    char *name;
    int yylex;
} keywords[] =
{
     {"auto",       Auto},
     {"break",      Break},
     {"case",       Case},
     {"char",       Char},
     {"const",      Const},
     {"continue",   Continue},
     {"default",    Default},
     {"do",         Do},
     {"double",     Double},
     {"else",       Else},
     {"enum",       Enum},
     {"extern",     Extern},
     {"float",      Float},
     {"for",        For},
     {"goto",       Goto},
     {"if",         If},
     {"int",        Int},
     {"long",       Long},
     {"register",   Register},
     {"return",     Return},
     {"short",      Short},
     {"signed",     Signed},
     {"sizeof",     Sizeof},
     {"static",     Static},
     {"struct",     Struct},
     {"switch",     Switch},
     {"typedef",    Typedef},
     {"union",      Union},
     {"unsigned",   Unsigned},
     {"void",       Void},
     {"volatile",   Volatile},
     {"while",      While}
};

/*
 *  binary_search():
 *      reserved word table look up routine
*/
int binary_search(void)
{
    register int mid;
    int cc, hi, lo;

    lo = 0;
    hi = DIM(keywords) - 1;

    while (lo <= hi)
    {
        mid = (lo + hi) / 2;

       if((cc=strcmp(yytext,keywords[mid].name))==0)
            return keywords[mid].yylex;
        if (cc < 0)
            hi = mid - 1;
        else
            lo = mid + 1;
   }
   return IDENTIFIER;
}

```

The reserved words in the array "**keywords[]**" must be in alphabetical order for the binary search in "**binary_search()**" to work. Each keyword is paired with its token value that is passed to the parser. Token values are defined in the **PCYACC** generated header file **ANSIC.H** according to the grammar description file **ANSIC.Y**. Any name that is not a keyword is an "**IDENTIFIER**".

The following error reporting routines are independent of the **C** lexical scanner defined above. They are for the parser part of the language engine and could be put into another file or the program part of the grammar description file. The reason for putting them hear is for better discussion. Furthermore, usually a grammar description file is considerably bigger than the scanner description file and requires more computer memory for compilation.

```
/*
 *  error reporting code for C language engine
*/

#define WIDTH        80  /* width of stderr device */
#define YYERRCODE   256     /* characters in ASCII */

FILE *yyerrfile = stderr;
                  /* file to write error report to */

char yyerrsrc[64] = ""; /* current input file name */

/*
 *  yyerror(): improved error reporting routine
*/
void yyerror(char *s, char *t)
{
    static const char expecting[] = "expecting: ";
    static int list = 0;
    static int column = 0;

    if (s != NULL)
    {
        if (column != 0)
            fputc('\n', yyerrfile);

        errprefix(s);

        if (t == NULL)
            column = 0;
        else
           column = fprintf(yyerrfile,"actual:%s",t);

       list = 0;
    }
    else if (t != NULL)
    {
       if (list == 0)
       {
            if( column+strlen(t)+sizeof(expecting)+1
                   < WIDTH - 2 )
               column += fprintf( yyerrfile,
                            "  %s%s", expecting, t );
           else
               column = fprintf( yyerrfile, "\n%s%s",
                                   expecting, t) - 1;
       }
       else
       {
           if( column + strlen(t) < WIDTH - 2 )
               column += fprintf( yyerrfile,
                                         ", %s", t );
           else
               column = fprintf( yyerrfile,
                                ",\n    %s", t ) - 1;
       }
       ++list;
    }
    else
    {
       fprintf(yyerrfile, "\n");
       column = list = 0;
    }
}

/*
 *  errprefix():
 *      print where the error occured on yyerrfile
*/
static void errprefix(char *msg)
```

```
{
    int punct = 0;

    fprintf(yyerrfile, "[error %d] ", error_count+1);

    if(yyerrsrc[0] != '\0')/* any input file name? */
    {
        fprintf(yyerrfile, "file '%s'", yyerrsrc);
       punct = 1;
    }
    if (yylineno >= 0)       /* valid line number? */
    {
        if (punct)
            fprintf(yyerrfile, ", ");
        fprintf(yyerrfile, "line %d", yylineno);
        punct = 1;
    }
    if( yytext != NULL && *yytext != '\0' )
                                      /* real token? */
    {
        if (punct)
            fprintf(yyerrfile, " ");
        fprintf(yyerrfile, "near \"%s\"", yytext);
        punct = 1;
    }
    if (punct)
        fprintf(yyerrfile, ": ");
    fprintf(yyerrfile, "%s\n", msg);
}

```

The error reporting routine, **yyerror()**, is the standard **PCYACC** error routine. **yyparse()** will call **yyerror()** whenever it detects a syntax error. The lexical scanner could also use the same routine to report any lexical error occurred in the input source file.

yyerror() in the **ANSIC** project is quite different from the one in the **DATES** project. Two parameters are passed to this version of **yyerror()**. Character pointer ***s** will point to the type of the error. For example, "Syntax Error" or "Illegal Character". The second character pointer, ***t**, will point to the actual error token or the token expected. FILE pointer ***yyerrfile** is used to guide the language engine reporting error message to a specific file. In our example **stderr** is used.

An auxiliary function, **errprefix()**, is called by **yyerror()** to report the location of the error occurred and the current error number. Character array **yyerrsrc[]** holds the current input **C** source file name. Function **main()** performs the assignment when the language engine is invoked.

Software developers can write their own version of **yyerror()** according to their needs.

The following function, **yydisplay()**, takes a token value as the passed parameter, "**ch**" , and returns a pointer to the print out form of the token. In our example the function is passed to function **yyerror()** as the second parameter.

```
/*
 *  yydisplay():
 *      returns pointer to the printable form
 *     for token value of  "ch"
*/
char *  yydisplay(int ch)
{
    static char *tok[] =
    {
          "DDD",
          "CHARACTER_CONSTANT",
          "FLOAT_CONSTANT",
          "INTEGER_CONSTANT",
          "STRING",
          "IDENTIFIER",
          "TYPENAME",
          "ENUMERATION_CONSTANT",
          "Auto",
          "Break",
          "Case",
          "Char",
          "Const",
          "Continue",
          "Default",
          "Do",
          "Double",
          "Else",
          "Enum",
          "Extern",
          "Float",
          "For",
          "Goto",
          "If",
          "Int",
          "Long",
          "Register",
          "Return",
          "Short",
          "Signed",
          "Sizeof",
          "Static",
          "Struct",
          "Switch",
          "Typedef",
          "Union",
          "Unsigned",
          "Void",
          "Volatile",
          "While",
          "OROR",
          "ANDAND",
            "EQU",
            "NEQ",
            "LEQ",
            "GEQ",
            "SHL",
            "SHR",
            "ADDADD",
            "SUBSUB",
            "PTR",
            "ADDEQ",
            "SUBEQ",
            "MULEQ",
            "DIVEQ",
            "MODEQ",
            "SHLEQ",
            "SHREQ",
            "ANDEQ",
            "XOREQ",
            "IOREQ",
            0
    };

    static char buf[16];

    switch (ch)
    {
            case          0: return ("[end of file]");
            case YYERRCODE: return ("[error]");
            case      '\a': return ("'\\a'");
            case      '\b': return ("'\\b'");
            case      '\f': return ("'\\f'");
            case      '\n': return ("'\\n'");
            case      '\r': return ("'\\r'");
            case      '\t': return ("'\\t'");
            case      '\v': return ("'\\v'");
    }

    if (ch > YYERRCODE && ch < YYERRCODE + DIM(tok))
            return(tok[ch-(YYERRCODE + 1)]);
                                        /* is %token */

    if (isascii(ch) && isprint(ch))
            sprintf(buf, "'%c'", ch);   /* printable */
    else
            sprintf(buf,"char %d",ch);/* unprintable */

    return(buf);
}

```

The tokens in the token table, "**tok**", should be in the order according to their corresponding token values. Token values are defined in the **PCYACC** generated header file, **ANSIC.H**, according to the grammar description file, **ANSIC.Y**.

5. Write the PCYACC Parser Description

The **PCYACC** parser description file **ANSIC.Y** is listed as following:

```
/*
===================================================

ANSIC.Y : PCYACC grammar description file for ANSI C
version 2.0

by Yan Luo


PCYACC (R) is a software product of
        ABRAXAS SOFTWARE INC.
Copyright (C) 1986-1997 by ABRAXAS SOFTWARE INC.

Reference:  The C Programming Language
            Second Edition
            By B.W. Kernighan and D.M. Ritchie
====================================================
*/
%{
#include <stdio.h>

extern void  yyerror(char *, char *);
extern char *yydisplay(int);
extern int   yylex(void);
%}

%union {
     int       i;
     float     r;
     char      *s;
}

/*
=================
special symbols
=================
*/
%token    DDD                /* three dots ... */

/*
=================
constants
=================
*/
%token    CHARACTER_CONSTANT
%token    FLOAT_CONSTANT
%token    INTEGER_CONSTANT
%token    STRING

/*
==================
identifiers
==================
*/
%token    IDENTIFIER
%token    TYPENAME
%token    ENUMERATION_CONSTANT

/*
==================
key words
==================
*/
%token    Auto      Break     Case      Char
%token    Const     Continue  Default   Do
%token    Double    Else      Enum      Extern
%token    Float     For       Goto      If
%token    Int       Long      Register  Return
%token    Short     Signed    Sizeof    Static
%token    Struct    Switch    Typedef   Void
%token    Volatile  Union     Unsigned  While

/*
==================
combined operators
==================
*/
/*
 * binary logicals and comparators
*/
%token    OROR           /* || */
%token    ANDAND         /* && */
%token    EQU            /* == */
%token    NEQ            /* != */
%token    LEQ            /* <= */
%token    GEQ            /* >= */

/*
 * shift operators
*/
%token    SHL            /* << */
%token    SHR            /* >> */

```

```
/*
 * unary increments
*/
%token    ADDADD         /* ++ */
%token    SUBSUB         /* -- */

/*
 * pointer
*/
%token    PTR            /* -> */

/*
 * assignments
*/
%token    ADDEQ          /* += */
%token    SUBEQ          /* -= */
%token    MULEQ          /* *= */
%token    DIVEQ          /* /= */
%token    MODEQ          /* %= */
%token    SHLEQ          /* <<= */
%token    SHREQ          /* >>= */
%token    ANDEQ          /* &= */
%token    XOREQ          /* ^= */
%token    IOREQ          /* |= */

/*
====================
operator precedence
====================
*/
%nonassoc Shift
%nonassoc error
%nonassoc IDENTIFIER
%nonassoc Else

/*
 * comma operator
*/
%left     ','

/*
 * assignment operators
*/
%right    ADDEQ     SUBEQ     MULEQ     DIVEQ
          MODEQ     SHLEQ     SHREQ

/*
 * binary operators
*/
%right    '?'       ':'
%left     OROR
%left     ANDAND
```

```
%left     EQU
%left     NEQ
%left     LEQ       GEQ
%left     SHL       SHR

/*
 * unary operators
*/
%right    ADDADD
          SUBSUB
          '~'
          '!'
          Sizeof
          '*'

/*
 * special operators
*/
%left     '('       '['       '.'  PTR

/*
=================
 start symbol
=================
*/
%start    translation_unit

%%

translation_unit
 :                    external_declaration
 | translation_unit   external_declaration
 | error              /* last ditch error recovery */
 ;

external_declaration
 : function_definition
 | declaration
 ;

function_definition
 : declaration_specifiers   declarator
   declaration_list         compound_statement

 | declaration_specifiers   declarator
                            compound_statement

 |                         declarator
   declaration_list         compound_statement

 |                         declarator
                            compound_statement
```

```

/*
 * insert wrong or missing parts of function headers
*/
 | declaration_specifiers   error
                            compound_statement

 |                         declarator
   error                    compound_statement
 ;

declaration
 : declaration_specifiers   init_declarator_list  ';'
 | declaration_specifiers                         ';'
 | identifier               identifier_list       ';'
 | identifier  identifier  '['  identifier  ']'   ';'
 | identifier              '*'  identifier        ';'

/*
 * fixed incorrect initializers list
*/
 | declaration_specifiers   error ';'    { yyerrok; }
 ;

declaration_list
 :                          declaration
 | declaration_list         declaration
 ;

declaration_specifiers
 : storage_class_specifier
 | storage_class_specifier  declaration_specifier
 | type_specifier
 | type_specifier           declaration_specifier
 | type_qualifier
 | type_qualifier           declaration_specifier
 ;

storage_class_specifier
 : Auto
 | Extern
 | Register
 | Static
 | Typedef
 ;

type_specifier
 : Char
 | Double
 | Float
 | Int
 | Long
 | Short
 | Signed
 | Unsigned
 | Void
 | typedef_name
 | enum_specifier
 | struct_or_union_specifier
 ;

type_qualifier
 : Const
 | Volatile
 ;

struct_or_union_specifier
 : struct_or_union identifier
                      '{' struct_declaration_list '}'
 | struct_or_union    '{' struct_declaration_list '}'
 | struct_or_union identifier
 ;

struct_or_union
 : Struct
 | Union
 ;

struct_declaration_list
 :                                struct_declaration
 | struct_declaration_list        struct_declaration
 ;

init_declarator_list
 :                                init_declarator
 | init_declarator_list    ','    init_declarator
 ;

init_declarator
 : declarator
 | declarator   '='   initializer
 ;

struct_declaration
 : declaration_specifiers  struct_declarator_list ';'

/*
 * fixed incorrect structures
*/
 | error                        ';'   { yyerrok; }
 ;

struct_declarator_list
 :                                struct_declarator
 | struct_declarator_list   ','   struct_declarator
 ;

struct_declarator
 : declarator
 | declarator               ':'   constant_expression
 |                          ':'   constant_expression
 ;

enum_specifier
 : Enum  identifier  '{'  enumerator_list  '}'
 | Enum              '{'  enumerator_list  '}'
 | Enum  identifier
 ;

enumerator_list
 :                         enumerator
 | enumerator_list   ','   enumerator
 ;

enumerator
 : identifier
 | identifier   '='    constant_expression
/*
 * fixed incorrect enumeration tags
*/
 | error
 ;

declarator
 : pointer            direct_declarator
 |                    direct_declarator
 ;

direct_declarator
 : identifier                            %prec Shift
 |                    '('  declarator             ')'
 | direct_declarator  '['  constant_expression    ']'
 | direct_declarator  '['                         ']'
 | direct_declarator  '('  parameter_type_list    ')'
 | direct_declarator  '('  identifier_list        ')'
 | direct_declarator  '('                         ')'
 ;

pointer
 : '*'   type_qualifier_list
 | '*'
 | '*'   type_qualifier_list    pointer
 | '*'                          pointer
 ;

type_qualifier_list
```

```
 :                              type_qualifier
 | type_qualifier_list          type_qualifier
 ;

parameter_type_list
 : parameter_list
 | parameter_list     ','       DDD
 ;

parameter_list
 :                              parameter_declaration
 | parameter_list     ','       parameter_declaration
 ;

parameter_declaration
 : declaration_specifiers       declarator
 | declaration_specifiers       abstract_declarator
 | declaration_specifiers

/*
 * fixed missing parameter
*/
 | declaration_specifiers       error
 ;

identifier_list
 :                            identifier
 | identifier_list   ','      identifier
 | error              /* insert missing identifier */
 ;

initializer
 : assignment_expression
 | '{'   initializer_list          '}'
 | '{'   initializer_list   ','    '}'
 ;

initializer_list
 :                            initializer
 | initializer_list    ','    initializer
 | error                 /* fixed missing constant */
 ;

type_name
 : declaration_specifiers   abstract_declarator
 | declaration_specifiers
 ;

abstract_declarator
 : pointer
 | pointer   direct_abstract_declarator
 |           direct_abstract_declarator
 ;

direct_abstract_declarator
 :                       '(' abstract_declarator ')'
 |                       '[' constant_expression ']'
 |                       '['                     ']'
 |                       '(' parameter_type_list ')'
 |                       '('                     ')'

 | direct_abstract_declarator
                         '[' constant_expression ']'

 | direct_abstract_declarator
                         '['                     ']'

 | direct_abstract_declarator
                         '(' parameter_type_list ')'

 | direct_abstract_declarator
                         '('                     ')'
 ;

typedef_name
 : TYPENAME
 ;

statement
 : labeled_statement
 | expression_statement
 | compound_statement
 | selection_statement
 | iteration_statement
 | jump_statement

/*
 * last ditch statement recovery
*/
 | error ';' { yyerrok; }
 ;

labeled_statement
 : identifier                       ':'  statement
 | Case        constant_expression  ':'  statement
 | Default                          ':'   statement
 ;

expression_statement
 : expression   ';'
 |              ';'
 ;

compound_statement
```

```
 : '{'  declaration_list   statement_list  '}'
 | '{'  declaration_list                   '}'
 | '{'                     statement_list  '}'
 | '{'                                     '}'
 ;

statement_list
 :                 statement
 | statement_list  statement
 ;

selection_statement
 : If     '(' expression ')' statement    %prec Shift
 | If     '(' expression ')' statement Else statement
 | Switch '(' expression ')' statement
 ;

iteration_statement
 :                While '(' expression ')'  statement
 | Do  statement  While '(' expression ')'  ';'

 | For   '(' expression ';' expression ';'
                            expression ')'  statement

 | For   '(' expression ';' expression ';'
                                       ')'  statement

 | For   '(' expression ';'            ';'
                            expression ')'  statement

 | For   '(' expression ';'            ';'
                                       ')'  statement

 | For   '('            ';' expression ';'
                            expression ')'  statement

 | For   '('            ';' expression ';'
                                       ')'  statement

 | For   '('            ';'            ';'
                            expression ')'  statement

 | For   '('            ';'            ';'
                                       ')'  statement

/*
 * fixed error in "for expression list"
*/
 | For   '(' error                     ')'  statement
 ;

jump_statement
```

```
 : Goto             IDENTIFIER   ';'
 | Continue                      ';'
 | Break                         ';'
 | Return           expression   ';'
 | Return                        ';'
 ;

expression
 :                           assignment_expression
 | expression   ','          assignment_expression

/*
 * lowest precedence infix op
*/
 | expression   error        assignment_expression
 ;

assignment_expression
 : conditional_expression
 | unary_expression          assignment_operator
                             assignment_expression
 ;

assignment_operator
 : '='
 | MULEQ
 | DIVEQ
 | MODEQ
 | ADDEQ
 | SUBEQ
 | SHLEQ
 | SHREQ
 | ANDEQ
 | IOREQ
 | XOREQ
 ;

conditional_expression
 : logical_or_expression
 | logical_or_expression  '?'  expression
                          ':'  conditional_expression
 ;

constant_expression
 : conditional_expression
 ;

logical_or_expression
 :                            logical_and_expression
 | logical_or_expression OROR logical_and_expression
 ;

```

```
logical_and_expression
 :                            inclusive_or_expression
 | logical_and_expression     ANDAND
                              inclusive_or_expression
 ;

inclusive_or_expression
 :                            exclusive_or_expression
 | inclusive_or_expression    '|'
                              exclusive_or_expression
 ;

exclusive_or_expression
 :                                   and_expression
 | exclusive_or_expression    '^'    and_expression
 ;

and_expression
 :                             equality_expression
 | and_expression     '&'      equality_expression
 ;

equality_expression
 :                             relational_expression
 | equality_expression   EQU   relational_expression
 | equality_expression   NEQ   relational_expression
 ;

relational_expression
 :                                 shift_expression
 | relational_expression    '<'    shift_expression
 | relational_expression    '>'    shift_expression
 | relational_expression    LEQ    shift_expression
 | relational_expression    GEQ    shift_expression
 ;

shift_expression
 :                             additive_expression
 | shift_expression     SHL    additive_expression
 | shift_expression     SHR    additive_expression
 ;

additive_expression
 :                         multiplicative_expression
 | additive_expression '+' multiplicative_expression
 | additive_expression '-' multiplicative_expression
 ;

multiplicative_expression
 :                                   cast_expression
 | multiplicative_expression   '*'   cast_expression
 | multiplicative_expression   '/'   cast_expression
 | multiplicative_expression   '%'   cast_expression
 ;

cast_expression
 :                                   unary_expression
 | '('  type_name  ')'               cast_expression
 ;

unary_expression
 : postfix_            expression
 | ADDADD              unary_expression
 | SUBSUB              unary_expression
 | unary_operator      cast_expression
 | Sizeof              unary_expression
 | Sizeof         '('  type_name  ')'
 ;

unary_operator
 : '&'
 | '*'
 | '+'
 | '-'
 | '~'
 | '!'
 ;

postfix_expression
 : primary_expression
 | postfix_expression
                  '['  expression                ']'

 | postfix_expression
                  '('  argument_expression_list  ')'

 | postfix_expression
                  '('                            ')'

 | postfix_expression    '.'       IDENTIFIER
 | postfix_expression    PTR       IDENTIFIER
 | postfix_expression    ADDADD
 | postfix_expression    SUBSUB
 ;

primary_expression
 :      identifier
 |      constant
 |      string
 | '('  expression  ')'
 ;

argument_expression_list
 :                              assignment_expression
 | argument_expression_list ',' assignment_expression
 ;

constant
 : INTEGER_CONSTANT
 | CHARACTER_CONSTANT
 | FLOAT_CONSTANT
 | ENUMERATION_CONSTANT
 ;

identifier
 : IDENTIFIER
 ;

string
 : STRING
 ;
```

6. Write the Auxiliary C Code

The auxiliary **C** file **MAIN.C** is listed as following:

```
/*
=====================================================
MAIN.C: main routine for ANSIC parser
Version 2.0
by Yan Luo

PCYACC (R) is a software product of
           ABRAXAS SOFTWARE INC.
Copyright (C) 1986-1997 by ABRAXAS SOFTWARE INC.
=====================================================
*/

#include <stdio.h> /* fopen(),fclose(),fprintf() */

#include <stdlib.h> /* EXIT_FAILURE/SUCCESS, exit() */

#include <string.h> /* strcpy() */

extern int yylineno;
       /* line # of current line, defined in lex.l */

extern int error_count;
         /* count of errors, defined in err_skel.c */

extern char yyerrsrc[64];
              /* input file name, defined in lex.l */

```

```
extern int yyparse(void); /* defined in err_skel.c */
FILE      *yyin;          /* pointer to input file */

main(int argc, char *argv[])
{
    if (argc < 2)
    {
        fprintf(stderr,
         "\nUsage:\n\tansic8 <program>\n\n\tor\n\n");

        fprintf(stderr, "\tansic7 <program>\n\n");
        exit(EXIT_FAILURE);    /* EXIT_FAILURE = 1 */
    }

    yyin = fopen(argv[1], "r");

    if (yyin == NULL)
    {
        fprintf(stderr,
         "Can't open source program file %s\n",
          argv[1]);
        exit(EXIT_FAILURE);    /* EXIT_FAILURE = 1 */
    }

    strcpy(yyerrsrc, argv[1]);
    yylineno = 1;
    (void) yyparse();
    fclose(yyin);

    if (error_count != 0)
    {
        fprintf(stderr,
    "\n<==== %d error%s found by the parser ====>\n",
    error_count,  (error_count == 1) ? "" : "s");

        exit(EXIT_FAILURE);
    }
    else
    {
        fprintf(stdout,
      "\nNo syntax error was found by the parser\n");
    }

    exit(EXIT_SUCCESS);        /* EXIT_SUCCESS = 0 */
}
```

7. Build the Program

The makefile of our **ANSI C** syntax analyzer is listed as follows.

```
#
#    UNIX style makefile for ANSI C syntax analyzer
#
CC=cl
CFLAGS=-c -qc
OBJS=ansic.obj  main.obj  lex.obj

ansic.exe : $(OBJS)
          $(CC) $(OBJS)

.c.obj :
          $(CC) $(CFLAGS) $*.c

lex.c : lex.l
          pclex lex.l

ansic.c : ansic.y
          pcyacc -r -n -D   ansic.y

ansic.obj : ansic.c

main.obj :  global.h main.c
```

IX. PRINCIPLES BEHIND PCLEX

1. Introduction to Formal Languages

Noam Chomsky, a linguist, in the mid-1950s defined a taxonomy of formal languages that is still in use. He defined four broad classes of languages in terms of the grammars, which are 4-tuples

$$G = (V, T, P, S)$$

where:

G is a grammar;
V = *{N, T}* is an alphabet contains non terminal symbols *N* and terminal symbols *T*;
T in *V* is an alphabet of terminal symbols;
P is a finite set of rewriting rules; and
S is a single non terminal , a member of *N*, which serves as an initial symbol to initiate each derivation sequence.

The language of the grammar is the set of terminal string that can be generated from ***S***. The difference in the four types of grammars is the allowed forms of the rewriting rules in ***P***. A grammar ***G*** is **Chomsky type 0** if the rules in ***P*** have the form

u ::= U with ***u*** in ***V+*** and ***U*** in ***V*.***

That is the left part ***u*** is a sequence of symbols and the right part ***U*** can be empty. **Chomsky type 0** grammars are also known as phrase structure grammars or phrase grammars. Little work has been done on **type 0** grammars.

For the **type 1** grammars, the context sensitive grammars, the rewriting rules are restricted to the form:

xUy ::= xuy with ***U*** in ***N***; ***x, y*** in ***V****, and ***u*** in ***V+***.

The context sensitive part in the name comes from the fact that ***U*** can be rewritten as ***u*** only in the context of ***x...y***. Context sensitive grammars have received quite a bit of attention from the theoreticians in linguistics, mathematics, and computer science.

In the **type 2** or context free grammars, the rewriting rules are further restricted to the form:

U ::= u with ***U*** in ***N*** and ***u*** in ***V****.

This class of grammars are called context free because ***U*** can be rewritten as ***u***, regardless of the context it appears in. The context free grammars are restricted enough to be amenable to analysis and general enough to be useful. Almost all programming languages have context free grammars.

The rewriting rules for the **Chomsky type 3** or regular grammars are even more restrictive. The form must be:

U ::= u or ***U ::= wu***, where ***u*** is in ***T*** and ***U*** and ***w*** are in ***N***.

Regular grammars have a fundamental role in both formal language theory and automata theory. The set of strings generated by a regular grammar is also the set "accepted" by a simple program (or machine) called a finite state automata (which is more precisely defined in **Chapter IX**), and vice versa. Thus we have a characterization of this set of languages in terms of the program complexity required to parse them.

Phrase structure grammars are the most general and include the languages generated by the other three types of grammar. Each of the classes is completely included in its predecessor and completely includes its successor class. That is all regular languages have a regular grammar, a context free grammar, a context sensitive grammar, and a phrase structure grammar. There are phrase structure languages that do have a context sensitive grammar.

2. Regular Expressions

Regular expressions are a notation for defining a regular language. The rules of the notation comprise a pattern description language. Regular expressions can be described by a context-free grammar:

RegularExpression = (T, N, P, S)

where

a) ***T*** = { ?, *, +, |, (,), /, ***symbol*** }

where ***symbol*** is any symbol in the alphabet of the target language the regular expression defines.

b) ***N = { RegularExpression,PrimaryList, Primary, Element }***

P is the set of grammar rules or productions. It should take the form of ***U := u*** with ***U*** in ***N*** and ***u*** in ***V****. ***P*** is listed as following:

c)
RegularExpression -> RegularExpression '|' PrimaryList
RegularExpression -> RegularExpression '/' PrimaryList
RegularExpression -> PrimaryList
PrimaryList -> PrimaryList Primary
PrimaryList -> Primary
Primary -> Element '*'
Primary -> Element '+'
Primary -> Element '?'
Primary -> Element
Element -> symbol
Element -> '(' RegularExpression ')'

And , ***S*** is the starting non terminal:

d) ***S = { RegularExpression }***

3. Regular Languages

A regular language, or called a regular set, is a language can be defined by either a regular grammar or a regular expression. More often in the practice we use a regular expression to define a regular language. Thus a language is regular if there exists a regular expression that represents the strings in the language. Some of the conclusions of further research regarding to regular language are listed as following:

Every finite set of strings is a regular language or, every regular language is finite;

If **L1** and **L2** are regular languages, then **L1*** or **L2***, **L1|L2**, and **L1.L2** are also regular languages;

If **L** is a regular language, then **L{0,n}** with **n>=0** is also a regular language;

If **L** is a regular language, then **L{1,1} = L{0,0}L**.

4. Non deterministic Finite State Machines (NDFSM)

A finite-state machine is defined as a 5-tuple:

M = (T, Q, P, q, F)

where:

M is a finite-state machine;
T is an alphabet contains terminal symbols;
Q is a finite set of states;
q in Q is one specific state called the start state;
F in Q is a set of final states or halting states;
P is a finite set of transition rules defining how the automaton advances from one state to the next according to the current state and the input symbol.

Strings in a regular language defined by a regular grammar or a regular expression can be recognized by a finite-state machine which processes the input string one character at a time. When the last character has been processed, if the machine is in one of a final states, the string is accepted; otherwise, it is rejected. If two automata accept the same language, they are said to be equivalent. If two automata are equivalent and have the same states and transitions except for the names of the states, they are call isomorphic. If an automaton has no equivalent automata with fewer states, it is called reduced.

A non deterministic finite-state machine (NFSM) can have more than one possible transition for any given state and input symbol. It can arbitrarily choose any available transition. If there exists at least one sequence of transitions form the start state to a final state that reads the whole string, the non deterministic finite-state machine is said to accept the input string.

A state transition without reading an input token is call an empty transition. A non deterministic finite-state machine allows to make an empty transition.

5. Deterministic Finite State Machines (DFSM)

A deterministic finite-state machine (DFSM) has at most one possible transition for any given state and input symbol and does not allow to make an empty transition.

6. PCLEX -- From Regular Expressions to DFSM

Many strategies can be used for building a finite state machine, or called a lexical analyzer, from a regular expression. **PCLEX** uses the strategy that first construct an **NFSM** from a regular expression and then convert the **NFSM** into a **DFSM**.

Thompson's construction is one of the algorithms building an **NFSM** from a regular expression. The input of the construction is a regular expression **r** over an alphabet **T**. The output of the construction is an **NFSM** accepting language **L(r)**.

The methods of the algorithm is described as follows:

i) For a regular expression **e** which denotes the set containing the empty string, **{e}**, construct the **NFSM**

```
start             e
-------->  i  ---------> f
```

where **i** is a new start state and **f** is a new accepting state.

ii) For a regular expression **a**, where **a** is a symbol in **T**, that denotes the set containing string **a**, **{a}**, construct the **NFSM**

```
start             a
-------->  i  ---------> f
```

where **i** is a new start state and **f** is a new accepting state.

Suppose **N(a)** and **N(b)** are **NFSM**'s for regular expressions **a** and **b**.

iii) For the regular expression **a|b**, construct the **NFSM**

```
                e            e
               ----->  N(a) ----->
start         |                   |
------> i --                       ------> f
              |  e            e   |
               ----->  N(b) ----->
```

ix) For the regular expression **ab**, construct the **NFSM**

```
start
--------> i ------->  N(a)  N(b)  -------> f
```

x) For the regular expression **a***, construct the **NFSM**

```
                       e
                 <--------------
start        e |               |      e
------>  i  ------>   N(a)  ------->-------> f
         |                              |
         |             e                |
         ------------------------------->
```

And the algorithm is as following :

a) parse **r** into its constituent sub expressions;

b) using rules **(i)** and **(ii)** described above to construct **NFSM**'s for each of the basic symbols in **r**;

c) combining these **NFSM**'s inductively using rule (**iii)**, **(ix)**, and **(x)** until obtaining the **NFSM** for the entire expression.

An algorithm called <u>subset construction</u> is then used to construct a **DFSM**, denoted by **D**, from the **NFSM**, denoted by **N**. By definition, **D** should be able to accept the same language **L(r)**. Each **DFSM** state is a set of **NFSM** states. The subset construction algorithm constructs a transition table, **Dtran**, for **D** so that **D** will simulate all possible moves **N** can make on a given input string.

Let **c** represents a possible input symbol, **s** represents an **NFSM** state, **s0** the start state of **N**, **S** a set of **NFSM** states that defines a **DFSM** state, and **Dstates** the set of states of **D**. Three operations are used to keep track of sets of **NFSM** states. They are:

e-closure(s) : the set of **NFSM** states reachable from **s** on e-transition;

e-closure(S): the set of **NFSM** states reachable from some **s** in **S** on e-transition;

move(S, c): the set of **NFSM** states to which there is a transition on input symbol **c** from some **s** in **S**.

The algorithm then can be written as:

```
Initiate the Dtran to all failure transitions.
while (there is an unmarked S in Dstates)
{
       mark S
       for(each input symbol c)
       {
            U = e-closure(move(S,c));
            if(U is not 0)
            {
                 if (U is not in Dstates)
                      add U as a new unmarked state
                      to Dstates
                 Dtran[S,c] = U
            }
       }
}
```

X. WRITING PCLEX SYNTAX DESCRIPTIONS

A simple scanner program is:

```
%%
\t              output(' ');
```

This program converts all tabs to blanks. The "**%%**" marks the beginning of the rules. The rule contains a pattern to match tabs ("**\t**") and an action **("output(' ');")**. Every time the pattern is matched, the action is executed. The function "**output()**" writes a character to the output (usually "**stdout**", the standard output) and is included in every generated scanner.

In general, the structure of **PCLEX** input files is:

```
definitions
%%
rules
%%
user subroutines
```

The **definition section** contains pattern macro definitions (see **section 4**), start condition declarations (see **sections 5.1 and 5.2**), and any preliminary in-line **C** code. The **rule section** is patterns to search for and actions to execute when the pattern is found. The user subroutines are **C** support routines used by the actions. The definition and **user subroutine sections** are optional. The second "**%%**" is optional, but the first is required to mark the beginning of the rules.

Rules start in the first column and are a pattern and an optional action. Patterns are written with regular expressions (**section 1**). Actions (**section 2**) are **C** program fragments to be executed when the pattern is found. Whitespace (blanks or tabs) separate patterns and actions. Unmatched input is handled by the default action (see "**-s**" option in **Chapter III**, **section 1.2**).

The rule section can be empty. The default action applies to all unmatched input. If the "**-s**" is not specified, the default action is to copy all input to the output. The minimum **PCLEX** program is one line:

```
%%
```

It copies the input to the output.

C code needed in the scanner can be included in one of three ways. Lines that start with a "**#**" in the first column are passed through intact. This allows preprocessor directives. **C** comments and lines that start with a tab or a space are also passed through intact. Arbitrary lines between a "**%{**" line and a "**%}**" line are also passed through. The bracketing lines are not copied to the **C** scanner file. The brackets must start in the first column. For example:

```
#include <stdio.h>
#define MAX_WIDGET      100
#define LONG_MACRO \

        rest of the macro

        /* C comments */
%{
int word_count = 0;
static void do_nothing(void);
%}
```

C code in the definition section is copied to near the start of the scanner. Header file **#include**s, global variables, **C** macro definitions, and redefinition of the pre-defined macros go here. **C** code in the rule section (other than actions) is copied to the start of the scanner function, just after the scanner function's own local variables. Local variables used by the actions and initial code to be executed each time the scanner is called goes here.

1. Regular Expressions

A regular expression specifies a set of strings to be matched. It contains text characters (which match themselves in the text to be compared) and operator characters (which specify repetitions, choices, and other features). The letters of the alphabet and the digits are always text characters.

1.1 Operators

The operator characters are:

" \ [] ^ - ? . * + | () $ / {} % <>

and if they are to be used as text characters, they must be put in quotes or preceded by a backslash. In addition, "**#**" and "/" are not text characters in the first column.

Whatever is between a pair of double quotes (**"**) is to be taken as text characters. An operator character can also be turned into a text character by preceding it by a "\", the escape character. The following patterns are all equivalent:

```
xyz\+\+
xyz"++"
"xyz++"
```

Another use of the quoting mechanisms is to get a blank into an expression. Normally, blanks and tabs end a rule. Any blank not within a character class (see **section 1.2**) must be quoted or escaped. **PCLEX** recognizes the older Kernighan and Ritchie **(K&R) C** escape sequences: "**\n**" is a new line, "**\t**" is a tab and "**\040**" is the blank character (the syntax of **ANSI C** escape sequences is slightly different). The **NUL** character ("**\0**") is not allowed in patterns.

1.2 Character Classes

Two slightly different patterns (for example, "**sit**" and "**sat**") can be combined into the single pattern ("**s[ai]t**"). The "**ai**" enclosed in square brackets, "**[**" and "**]**", matches a single character, either a single "**a**" or a single "**i**". The brackets enclose a character class, a list of alternatives to match. Character classes can include escape sequences (for example, "**[\t\n]**" matches a whitespace character). A character class is negated or complemented by a caret at the beginning. For example, "**[^ \t\n]**" is any character except a blank, tab, or end of line. Note that this is not the same as the printable characters, it includes the control characters.

The other operator in character classes is the range operator. The digits form a continuous range and can be abbreviated to "**[0-9]**" instead of "**[0123456789]**". The octal digits are "**[0-7]**". The lower and upper case letters both form ranges and "**[a-zA-Z]**" matches any letter. The order of the range limits is unimportant, "**[a-z]**" and "**[z-a]**" are equivalent. Ranges of other characters are allowed, but **PCLEX** gives a warning message. The actual contents of the character class are machine-dependent. To use the hyphen as itself in a character class, put it at the end or beginning of the character class or escape it with a backslash, e.g., "**[-a+]**", "**[+a-]**", and "**[a\-+]**" are all equivalent. The character class "**[\001-\0177]**" is all allowed characters and "**[-~]**", "**[^\01-\037\0177]**", and "**[\040-\0176]**" are the 96 ASCII printable characters.

1.3 Repetition

Repetitions of elements are indicated by the "*****" and "**+**" operators. The regular expression "**A***" matches any number of **A**'s, including none, while "**A+**" matches one or more **A**'s. The pattern for **C** identifiers is:

```
[_a-zA-Z][_a-zA-Z0-9]*
```

A specific number of repetitions is specified by a number inside braces,"**{**" and "**}**". For example, "**AA**" and "**A{2}**" both match exactly two **A**'s. Two numbers separated by a comma inside the braces specify a range of repetitions. For example, "**[a-z]{1,5}**" matches one to five lower-case letters.

1.4 Arbitrary Character

The "**.**" operator matches any character except the end of a line. For example, "**a.b**" matches "**aab**", "**a0b**", "**a\b**", etc. To avoid letting unmatched input fall through to the default action, the last rule in the section is typically:

```
.          /* action for all other input */
```

Do not use the patterns "**.***" or "**.+**" to replace the default action, hoping to get unmatched input text in chunks, instead of a character at a time. The "**.+**" pattern matches entire lines and overrides almost any other pattern. Use the single character pattern in the example to replace the default action, to start with. (When you are proficient with **PCLEX**, consider redefining the **ECHO** macro as described in **Appendix III**.)

1.5 Alternation and Grouping

The "|" operator indicates alternation (either this or that) and the "**()**" operator pair indicates grouping. The expression "**(ab|cd)**" matches either "**ab**" or "**cd**". Alternation is the lowest precedence operator. As a complete pattern, "**ab|cd**" also matches "**ab**" or "**cd**". The patterns in each column below are equivalent to each other:

```
ab|cd          [a-c]
(ab|cd)        (a|b|c)
(ab)|(cd)
```

Multi-character patterns can be repeated by enclosing them in parentheses and following them with either "**+**" or "*****". For example, "**(abc)+**" matches any number of repetitions of "**abc**", i.e., "**abc**", "**abcabc**", "**abcabcabc**", etc. The patterns "**(a|e|i|o|u)+**" and "**[aeiou]+**" are equivalent and match any number of consecutive vowels.

1.6 Optional Expressions

The "**?**" operator indicates that the preceding element is optional. For example, "**ab?c**" matches either "**ac**" or "**abc**". Groups and character classes can also be optional, "**a(b|c)?d**" and "**a[b-c]?d**" both match "**ad**", "**abd**", and "**acd**".

1.7 Context Sensitivity

Sometimes it is desirable to match a pattern only within a specific context. Context before the pattern (left context) is handled with start conditions, exclusive start conditions, actions, and the "^" operator. Context after the pattern (trailing context) is handled with the "/" and "**$**" operators, and the "**yyless()**" action.

The "^" character is only an operator in the first column of a pattern (or as the first character in a character class). Elsewhere, it is a text character. The "^" operator anchors a pattern to the start of a line. The pattern must start at the beginning of the line to match. For example:

```
^[ \t]*#.*
```

matches **C** preprocessor directives. Other left context methods are explained in **section 5**.

The "**$**" character is an operator only when it's the last character in a pattern. Elsewhere, it is a text character. The "**$**" operator anchors the pattern to the end of a line. It does not match the end of line character(s), it restricts matches of the pattern to immediately before the end of a line. For example, the scanner program:

```
^[ \t]+$       ;
```

matches and discards trailing whitespace on lines. The ends of lines are not matched and remain in the output.

The "/" operator is a more general way of indicating trailing context. The pattern precedes the "/" and the trailing context follows. For example, the patterns "**abc$**" and "**abc/\n**" are equivalent. Both match three characters and leave the end of line for another pattern to match.

Another way to handle trailing context is with the "**yyless()**" action. It is described in the next section on actions.

2. Actions

When a pattern is matched, the scanner executes the corresponding action, the **C** code associated with the pattern. If **PCLEX** finds a match of a pattern without associated **C** code, it will write a copy of the matched input to the output; that is, **PCLEX** does the default **ECHO** action.

PCLEX allows multiple statements in an action. If they will fit after the pattern on the same line, no braces are necessary. Longer actions are enclosed in braces, "**{**" and "**}**", and can span several lines. The braces must be balanced. **PCLEX**'s brace counting can be thrown off by braces in comments and quoted strings. If an action has either of these, use "**%{**" "**%}**" brackets. These brackets cannot be nested or substituted for braces. Do not put either bracket in a comment or quoted literal. If you need one in a string, use escape sequences to break it up. For example, instead of **"%{...%}"**, use **"%\{...%\}"** on older **K&R C** compilers and **"%" "{...%" "}"** on **ANSI C** compilers.

3. Ambiguous Rules

PCLEX generated scanners compare the patterns against the input stream. The longest string that matches a pattern is read into "**yytext[]**", the pattern's action executed, and scanning resumes on the remaining input. If more than one pattern matches this input string, the order of the patterns in the scanner description file determines which matches. The first pattern is the first choice. For example, given the patterns:

```
abc
[abc]+
```

Both patterns match the first three characters of "**abcd**". The action associated with the first pattern will be taken.

A frequent shortcut for scanners used with parsers is to have no explicit rules for single character operators and delimiters, and to add a final rule like:

```
.           return yytext[0];
```

This matches any character not otherwise matched and passes it to the parser. Since the parser has to deal with invalid input anyway, this leaves the problem of reporting invalid characters as well as invalid syntax up to the parser. This centralizes both lexical and syntactic error reporting in one place.

Sometimes the above rules are not sufficient. One may need an action determining that another match is better. The "**REJECT;**" action forces the scanner to put back the text matched and take the next best match. The **REJECT** allows **PCLEX** use even in situations where the basic pattern matching mechanism is not quite general enough.

Sometimes, matches should not be laid end to end. Instead, all possible matches, even ones that overlap are wanted. For example, consider the following scanner program to count the frequency of pairs of letters (digraphs) in a file:

```
%%
[a-zA-Z][a-zA-Z]        ++digraph[yytext[0]][yytext[1]];
```

This scanner counts consecutive pairs. For example, in the word "**pair**" it would count two digraphs: "**pa**" and "**ir**". To count overlapping pairs, i.e., "**pa**", "**ai**", and "**ir**", the following changes are needed:

```
%%
[a-zA-Z][a-zA-Z] {
                        ++digraph[yytext[0]][yytext[1]];
                        REJECT;
                 }
```

Note that this can also be done another way:

```
%%
[a-zA-Z][a-zA-Z] {
                        ++digraph[yytext[0]][yytext[1]];
                        yyless(1);
                 }
```

For clarity, some details necessary for a working program have been omitted. The complete programs are in the **\DIGRAPH** directory.

In general, the **REJECT** action is useful for matches that overlap and for instances where semantics and lexical analysis interact.

4. Definitions

Definitions (macros for regular expressions) are declared in the first section of the scanner description (before the first "**%%**"). They are of the form:

```
name        translation
```

and must start in the first column. The translation is substituted in the rule section wherever "**name**" appears within braces. The following scanner program matches **C** identifiers:

```
alpha       [a-zA-Z]
digit       [0-9]
%%
({alpha}|_)({alpha}|{digit}|_)*
```

In **PCLEX**, the replaced text is enclosed in parentheses. For example, given the definition and regular expression:

```
        NAME      [A-Z][A-Z0-9]*
        %%
        foo{NAME}?
        %%
```

PCLEX (and FLEX) will match "foo" because they expand the rule to:

```
        foo([A-Z][A-Z0-9]*)?
```

Both of the character classes are included in the optional part. UNIX LEX will not match "foo" because it expands the rule to:

```
        foo[A-Z][A-Z0-9]*?
```

Here, the "**?**" only applies to the second character class. The first character class is in the required part.

5. Context Sensitivity

Not all programs have the same syntax throughout the entire input. For example, the three different sections in both **PCLEX** and **PCYACC** have quite different syntax. **PCLEX** provides several facilities to make a pattern sensitive to left and right context. This section describes fore ways to handle this situation: with the actions and user subroutines (**section 5.1**), with start conditions (**section 5.2**), with exclusive start conditions (**section 5.3**) and with special position anchors (**section 5.4**).

5.1 Actions and User Subroutines

Actions can explicitly set and test variables to change the output depending on some prior condition. This method is suitable when the output is context dependent and the input syntax is constant regardless of context.

In an action, the matched text is pointed by the character pointer "**yytext**", and its length is in the integer variable "**yyleng**". The amount of matched text can be changed in the action. "**yytext**" should not be changed directly.

The built-in action "**yyless(n)**" indicates that only the first "**n**" characters of the current match are to be retained and the rest are to be pushed back onto the input for rescanning. The "**yyless()**" function provides the same sort of lookahead offered by the "/" operator (see **section 1.7**) in a different form. The following two rules are equivalent:

```
        abcd/efg    ;
        abcdefg                 yyless(4);
```

The difference is that **yytext** holds "**abcdefg**" in the second rule; **yytext** holds "**abcd**" in the first rule.

5.2 Start Conditions

Start conditions allow turning some patterns on in some contexts and off in others. Which is useful in the case one requires that one token precede another. The scanner starts off with all start conditions off, the default start condition. Actions put the scanner in a start condition and return it to the default start condition. Each start condition must first be declared in the definition section with a line like:

```
%Start name1 name2
```

The start conditions may be named in any order. The keyword "**%Start**" can be shortened to either "**%s**" or "**%S**". The start condition is referenced at the start of the applicable rule(s) in pointed brackets, "<" and ">": **<name1>** or **<name2>**.

To enter start condition "**name1**", execute the action statement:

```
BEGIN(name1);
```

To reset the start condition to the default, execute the action:

```
BEGIN(0);
```

For example:

```
%Start name1
%%
pattern1      { BEGIN(name1);}
<name1>pattern2          { BEGIN(0); action; }
```

In our example, **pattern2** is only recognized when the scanner is in the start condition "**name1**", that is, **pattern1** had been matched. After **pattern2** being matched the scanner reset the start condition to the default.

The scanner can be in only one start condition a at time. A rule may be active in several start conditions. For example:

```
<name1,name2,name3>pattern        ;
```

Except when the scanner is in an exclusive start condition, any rule not beginning with a "<>" prefix is always active. For example, in processing a SDF, one may code something like:

```
%Start section2
...
%%
...
^"%%"{ BEGIN(section2);}
...
```

In the example all rules not beginning with a "<>" will be active no mater it processes **section1** or **section2** of the input SDF.

The current definition of the **BEGIN** macro is:

```
#define BEGIN yy_start = 1 +
```

This maintains compatibility with UNIX LEX. However, there is a trap for the unwary here. The naive expectation for the action "**BEGIN 2 << x;**" is "**yy_start = 1 + (2 << x);**", the reality is "**yy_start = (1 + 2) << x;**" (in **C**, "+" is higher precedence than "<<"). It is strongly suggested that you always enclose the "**BEGIN**" expression in parentheses. This will give the expected results, even if the **BEGIN** macro definition changes in future releases.

5.3 Exclusive Start Conditions

A regular start condition turns on additional rules when active. An exclusive start condition turns off all normal rules and turns on only those rules prefixed with the exclusive start condition name. Exclusive start conditions are declared in the definition section with "**%x**" instead of "**%s**".

Matches can theoretically be of any length (for example, "**(.|\n)***" matches the entire input file). The practical limit is set by the size of an internal buffer. If a match can span several lines, it should be broken into several matches, each not exceeding a line in length. A good way to break up the matches is with start, middle, and end patterns. The start patterns match the beginning, (e.g., the "**/***" in a **C** comment) and **BEGIN** the exclusive start condition. The middle patterns are prefixed by the exclusive start name in "<>" and match partial or whole lines in the middle (e.g., anything up to the end of the line or a "***/**"). The end patterns match the character(s) that close the whole thing (e.g., "***/**") and reset the start condition with "**BEGIN(0);**".

C comments can extend over several lines. Whatever the scanner's buffer size, some program will exceed it. The **C** comments portion of the **ANSI C** scanner in **\ANSIC** looks like this:

```
%x                              COMMENT
%%
"/*"                            BEGIN(COMMENT);
<COMMENT>"*/"                   BEGIN(0);
<COMMENT>[^*\n]+  ;
<COMMENT>\n                     ++yylineno;
<COMMENT>"*"                    ;
%%
```

The **COMMENT** exclusive start condition is declared in the definition section. The first rule recognizes the start of a **C** comment. The second rule recognizes the end. To avoid overflowing the scanner's internal buffer, the comment contents are recognized a line at a time. The third and forth rules do this. The third and fifth rules fix a subtle problem. If the third rule read "**[^\n]+**", it would match every line after the start of a comment including the asterisk of the closing "***/**" (except in the case of a "***/**" at the start of a line because the second and third rules would both match the same length text and the earlier rule takes precedence). The "*****" in the third rule prevents it from matching the comment's close. The fifth rule matches a solitary "*****" without a following "/".

The above code would not work if **COMMENT** had been declared as a regular start condition name since all regular patterns would still be active in the **COMMENT** state. For extensive examples of exclusive start conditions, see **SCAN.L** in **\PCLEX** on the **PCLEX** disk.

5.4 Special Position Anchors

The "^" operator anchors a pattern to the start of a line. The "**$**" operator anchors the pattern to the end of a line. And the "/" operator is used for indicating trailing context.

APPENDIX A. INSTALLATION

Installation of **PCLEX** is simple and straight forward. Its self-contained nature makes it much easier to install than comparable products.

1. System Requirements

PCLEX will work on most MS-DOS and Microsoft Windows 95/NT computers. Specifically, all IBM PCs, XTs, ATs, and compatibles, as well as IBM PS2s, i86, and Pentium based computers. In fact PCLEX is available for all computer operating systems and architectures.

The following minimum configuration is sufficient to run **PCLEX**:

640KB memory
3.5 inch floppy drive
20 MB hard disk

The following programs are needed for software development using **PCLEX**:

1) A text editor for programming (like **BRIEF**, **EPSILON**, **EMACS**, or **EDLIN**). Many word processing programs (like Word star) will work in non-document mode. The scanner description file must be straight ASCII with no IBM extended characters, hidden characters, or other characters with the high bit set.

2) A **C/C++** compiler (like **Microsoft Visual C++**).

2. Making Working Copies

It is always a good practice to make copies of your original diskettes to protect against accidental damage. Installation should be done from the copies and the original diskettes stored safely away from the computer and other source of heat or strong magnetic fields like music speakers, motors, and transformers.

3. Installing PCLEX

PCLEX eliminates the need for a separate library of source code, 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 **PCLEX**'s operation. This change also simplifies the installation process.

To perform the standard installation, follow these steps (these are many alternate ways of installing **PCLEX**):

1) create a directory for **PCLEX** to reside (e.g., **\PCLEX**):

```
C>cd \
C>mkdir pclex
C>cd pclex
```

2) insert the diskette containing **PCLEX** in **drive A**:

3) copy files from the diskette to the hard disk, the /s switch allows you to copy all files in the diskette including sub directories:

```
C>copy a:*.* /s
```

4) modify "**AUTOEXEC.BAT**" file to add "**\PCLEX**" to the **PATH** environment variable and reboot.

At this point, the installation process is complete and **PCLEX** is ready to go. (Files in sub directories of the distribution diskette contain several interesting examples, which you may also want to copy onto your hard disk at this time; or you may choose to copy them later as you need them.)

NOTE: If you already have a directory (like "**\BIN**") set up for executable programs, "**PCLEX.EXE**" may be directly copied to that directory. No changes need to be made to the "**AUTOEXEC.BAT**" file. However, it is recommended that you create a separate directory for **PCLEX**. Creating a separate directory makes it easier to organize your **PCLEX** related files and example **PCLEX** programs.

APPENDIX B. ERROR MESSAGES

The error messages produced by PCLEX have the following format:

Lxxxx: Error Message

L0001: illegal character
the inputted character is out of ASCII character set

L0002: incomplete name definition
in the definition section of a SDF, a name following only
white spaces on a line

L0003: indented code found outside of action
in the rule section, only comment and action lines can be indented

L0004: undefined {name}
a name which has not been defined in the definition section
is used in the rule section

L0005: bad start condition name
a name can contain only letters, digits, underscores,
and must not start with a digit

L0006: missing quote
in the rule section only odd number of quotes being in a pattern

L0007: bad character inside {}'s
in the rule section only a defined name or a number list can be inside {}'s

L0008: missing }
in the rule section more {'s than }'s in a pattern

L0009: bad name in {}'s
a name can contain only letters, digits, underscores,
and must not start with a digit

L0010: read error in section 3 of the SDF
system function read() failed to read the contents of section 3
in a SDF into a buffer

L0011: error in processing section 1 of the SDF
unknown syntax or system error found in processing start
condition name declaration or exclusive start condition name
declaration in the definition section of a SDF

L0012: bad start condition list in section 1 of the SDF
syntax error in a start condition name list or exclusive start condition

name list.   usually caused by some illegal characters being in the names or by their self in the name list

L0013: unrecognized rule in section 2 of the SDF
syntax error in a rule

L0014: undeclared start condition
a start condition name which has not been declared in the definition section is used in the rule section

L0015: bad start condition list in section 2 of the SDF
error in a start condition name list or exclusive start condition name list.   usually caused by some bad names or undeclared start condition names in the list

L0016: trailing context used twice
end of line anchor '$' used twice in a pattern

L0017: illegal trailing context
both head and trail are variable-length; the trailing context had better be fixed-length

L0018: bad iteration values
inside an iteration operator, {n1, n2}, n2 must be bigger then n1 and n1 must be bigger then zero

L0019: iteration value must be positive
inside an iteration operator, {n1,   }, n1 must be bigger then zero

L0020: null in rule
null character '\0' found in a rule

L0021: negative range in character class
in a character class [c1-c2], the ASCII value of c2 must bigger then that of c1

L0022: symbol table memory allocation failed
system error, can't allocate memory

L0023: name defined twice
name defined twice in the definition section of a SDF

L0024: start condition declared twice
start condition name declared twice in the definition section of a SDF

L0025: input rules are too complicated
the current maximum on number of NFA states plus the amount to bump above by is bigger then the maximum number of NFA states

L0026: found too many transitions
too many possibilities in making transitions from one state to others

L0027: PCLEX scanner push-back overflow
unputing more characters then inputted

L0028: fatal scanner internal error
PCLEX scanner can not find what action should be done upon a token inputted

L0029: PCLEX input buffer overflowed
the inputted pattern string is more than 128 characters

L0030: PCLEX scanner saw EOF twice
system error, the scanner should not see EOF twice for one run

L0031: memory allocation of an internal integer array failed
system error, can't allocate memory

L0032: dynamic memory failure in copying string
system error, can't allocate memory

L0033: escape sequence for null not allowed
\\0 is not allowed

L0034: illegal \^ escape sequence
\\^ is not allowed

L0035: memory reallocation of a dynamic array failed
system error, can't allocate memory

L0036: consistency check failed in the epsilon closure
system error, the state should be marked if we've already pushed it onto the stack

L0037: dynamic memory failure in converting a set of NDFA states into a DFA state
system error, can't allocate memory

L0038: consistency check failed in symbol transitions
system error, input character put in wrong place

L0039: bad transition character detected
system error, input character put in wrong place

L0040: -p or -C flag must be given separately
command line message, other flags could be grouped together

L0041: unknown flag
command line message

L0042: could not create scanner output file
system function freopen() does not work well

L0043: extraneous arguments given
command line message

L0044: can not open input file
system function fopen() does not work well

L0045: can not open skeleton file
system function fopen() does not work well

L0046: can not open temporary action file
system function fopen() does not work well

L0047: fatal parse error at the input SDF
unrecoverable grammar syntax error found in the SDF

L0048: fread() in PCLEX scanner failed
system function fread() does not work well

APPENDIX C. EXTENDING AND CUSTOMIZING SCANNERS

PCLEX has a number of macros that can be redefined to customize or extend the generated scanner. The input stream functions ("**input()**" and "**unput()**") can be called by actions, user subroutines, and other parts of the program.

1. Macros

The following macros are those most likely to redefined. For portability, they should be undefined before being redefined. "**yywrap()**" can be redefined as a macro or as a function.

"**yywrap()**" - called when the scanner reaches the end of the file. If it evaluates to non-zero, the scanner finishes up processing and returns a zero to the caller. If "**yywrap()**" evaluates to zero, the scanner continues, expecting new input. This is useful for doing file inclusion and other multiple input file scanning. To do this, **yyin** also needs to be adjusted. A common way is using system function **freopen()** to make **yyin** pointing to a new file. The predefined "**yywrap()**" evaluates to 1.

"**ECHO**" - is the default action if the "**-s**" option is not given. The predefined "**ECHO**" macro copies the matched input to the output stream. It is equivalent to:

```
fputs(yytext, yyout);
```

Redefine it to change the default action.

"**YY_DECL**" - is a macro that declares the scanning function generated by **PCLEX**. It can be redefined to change the function name, type, or argument list. For example:

```
#undef YY_DECL
#define YY_DECL float lexscan(float a, float b)
```

gives the scanner function the name "**lexscan**". It takes two floats as arguments and returns a float. **YY_DECL** is predefined as:

```
#define YY_DECL int yylex()
```

"**YY_INPUT**" - macro called by "**input()**" to read more input text into a buffer. Its three arguments are: the buffer to read the input into, an integer variable that receives the number of characters actually read, and the size of the buffer. The default value is:

```
#define YY_INPUT(buf,result,max_size) \
      if (fgets(buf, max_size, yyin) != NULL) \
            result = strlen(buf); \
      else if (!ferror(yyin)) \
            result = YY_NULL; \
      else \
            YY_FATAL_ERROR \
                  ("fgets() in flex scanned failed");
```

A faster, though not as portable version is:

```
#define YY_INPUT(buffer,result,max_size) \
      if ((result=read(fileno(yyin),buf,max_size)) < 0) \
            YY_FATAL_ERROR \
                  ("fgets() in flex scanned failed");
```

On some compilers, The "**read()**" function does not do end of line translation.

Because of the buffering within "**input()**", "**YY_INPUT**" should not be called, except by "**input()**". If your code needs to read the input stream directly, call "**input()**".

"**YY_BUF_SIZE**" - is the size of the input buffer used "**input()**" and "**unput()**". The longest allowed match is "**YY_BUF_SIZE - 1**" , which evaluates to **254**.

"**YY_NULL**" - is the value of the "**result**" in "**YY_INPUT**" macro at an end of file (**EOF**) on the input stream.

2. Variables

"**FILE *yyin**" - input stream read by "**input()**", initialized to "**stdin**".

"**FILE *yyout**" - output stream written to by "**ECHO**", initialized to "**stdout**".

3. Functions

"**input()**" - returns the next input character.

"**unput(c)**" - pushes the character "**c**" back onto the input stream to be later read by "**input()**".

4. Scanner Skeleton Format

PCLEX combines the user's **C** code, tables generated from the patterns, and the scanner code that uses the tables (the scanner skeleton). The default scanner skeleton code is built into **PCLEX**. With the **-P** option, you can use your own skeleton code. **LEXSCAN.C** in **\PCLEX** is equivalent to the built-in skeleton. A skeleton is in four sections separated by lines beginning with "**%%**". The skeleton is copied to the scanner **C** file with the "**%%**" lines replaced by code copied from the scanner description file or tables generated by **PCLEX**.

The first section of the skeleton is header file **#include**s, **C** macro definitions, and function prototypes. **The first "%%" line** is replaced by **C** code from the definition section and the scanner tables generated by **PCLEX**. The second section is the "**unput()**" and "**input()**" functions that buffer the input stream and handle the reading ahead and backing up done for trailing context, "**yyless()**", and other methods of lookahead. The scanner function header ("**YY_DECL**") and its local variable declarations finish up the second section. **The second "%%" line** is replaced by **C** code from the rule section of the scanner description file. This code is declarations local to the scanner function used by the actions and executable code to be run each time the scanner function is called. The third section is the bulk of the code that interprets the generated tables and does the actual pattern matching. **The third "%%" line** is replaced by the actions' code. This code is in a "**switch**" statement. Each action's code is preceded by a "**case**" label and followed by a "**YY_BREAK**" macro call. The "**YY_BREAK**" macro normally is a "**break**" statement. The fourth section is the rest of the scanner function. The entire users' subroutine section is copied verbatim to the scanner **C** file, after the fourth section.

Study the **LEXSCAN.C** code carefully and look at several generated scanner **C** files before writing your own scanner skeleton.

APPENDIX D. BIBLIOGRAPHY

Aho, A.V. and Ullman, J.D. "Principle of Compiler Design", Addison-Wesley, Reading, Massachusetts, 1977.

This first edition is much better than the current ‘second edition’.

Aho, A.V. and Ullman, J.D. "The Theory of Parsing, Translation, and Compiling", Vol. 1, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

The original ‘theory’ of yacc table generation.

Barrett, William A. "Compiler Construction", 2nd Edition, Science Research Associates, Chicago.

Bickel, M. A. "Automatic Correction to Misspelled Names: A Fourth Generation Language Approach", CACM, 30:224-228, 1987.

Chapman, N. "*Regular Attribute Grammars and Finite State Machines*" SIGPLAN Notices, 24(6):97-104, 1989

Chomsky, Noam, "*Three models for the description of languages*," IEEE Transactions on Information Theory, Vol. 2 (1956), pg. 113-124.

Farmer, Mick. "Compiler Physiology for Beginners", Chartwell-Brat.

Fischer, Charles N. and LeBlanc, Richard J. "Crafting a Compiler", Addison-Wesley, 1988.

Friedl, Jeffery E.F. **“Mastering Regular Expressions”** O’Reilly & Associates 1997

This book is an excellent reference to the general field of regular expressions.

Genillard, C. and A. Srohmeier. "*A Grammar Description Language for Lexical and Syntactic Parsers*" SIGPLAN Notices, 23(10):103-122, 1988

Grune, D. and Jacobs, C.J.H. "Parsing Techniques", Ellis Horwood, Chichester, England, 1990.

Holmes, Jim. “Object-oriented compiler construction” Prentice Hall 1995

Our new Pclex Object Oriented Toolkit is based on the philosophy of this book.

Johnson, S.C. and Lesk, M.E. "Language Development Tools", The Bell System Technical Journal, Vol. 57, No. 6, Part 2 (July-August 1978).

Johnson, W. L.,   Porter, J. H., Ackley, S. I., and Ross, D. T. "Automatic Generation of Efficient Lexical Processors using Finite-State Machines", CACM, 11(12):805-813, 1968.

Lemone, K. A. "Design of Compilers, Techniques of Programming Language Translation", CRC Press, 1992.

Lesk, M.E. and Schmidt, E. "Lex - A Lexical Analyzer", Unix Programmer's Manual, Bell Laboratories, 1978.

Kernighan, B.W. and Ritchie, D.M. "The C Programming Language", Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

Mason, T., Brown D., and Levine, J. "Lex & Yacc", O'Reilly & Associates, Second Edition, 1992.

This is an excellent book for beginners. The standard free PCYACC DEMO will do all examples in this book.

Pyster, A.B." Compiler Design and Construction", Van Nostrand Rheinhold, New York, New York, 1980.

How to write a pascal compiler using lex&yacc.

Schreiner, A.T. and Friedman, Jr. H.G. "Introduction to Compiler Construction with UNIX", Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

How to write a C compiler using lex&yacc.

Szafron, P. and R. Ng. "LexAGen: An Interactive Incremental Scanner Generator", 20(5), 1990.

APPENDIX E. GLOSSARY

Action: Fragment of C code executed when its corresponding pattern is matched.

Algorithm: step by step instructions on how to do some task, that guarantees success. For example: in cooking, recipes are algorithms.

Backus-Naur Form (BNF): a notation used for describing context free grammars. It was first used in the report on the ALGOL-60 programming language for describing the syntax.

Backus-Normal Form (BNF): see Backus-Naur Form

Declaration Section: The first part of a grammar description program, in which one defines terminal symbols for grammars, declares types for grammar symbols, precedence and associativity for grammar symbols.

Default: the action or value used if no action or value is explicitly given.

Default action: The default action of PCLEX is to copy the unmatched text to the output.

Definition Section: The first part of a scanner description program, in which one defines names, declares global variables, C macros, and #includes needed header files.

Exclusive Start Condition: a state used to turn off all normal rules and turn only those rules prefixed with the exclusive start condition name

Equivalent: having identical effects. Two patterns are equivalent if every input that one matches the other also matches.

Grammar: the structures of a language and the rules for combining them.

Grammar Description File (GDF): input file to PCYACC describing the syntax of the target language.

Grammar Description Language (GDL): the language used to describe the syntax and parsing of the target language, a combination of BNF and C.

Grammar Description Program (GDP): programs written in GDL for parsing programs in the target language.

Grammar Rule Section: the second part of a grammar description program, where grammar rules and their associated actions are defined.

Keyword: word or identifier reserved by the language for special use. Examples in C are: "if", "else", "void".

Lexical Scanner: Front end of a parser, which reads the raw text input and partition them into meaningful lexical units, or tokens, of the target language.

Macro: an expression that resembles a constant or a function call. The C preprocessor replaces every macro expression with its appropriate C expansion before compilation.

Options: command line arguments to change the action of a program. Also known as switches and flags.

Parser: A program that analyzes the syntax of program input.

Parser Generator: A program that is capable of automatically generating parsers from a language description.

Port: (verb) short for transport. Means to move a program to another hardware or software environment and make any necessary changes

PCYACC: Abraxas Software's implementation of YACC (*see*), a parser generator
.
Program Section: third and last section of a grammar description program, where needed C functions can be included.

Regular Expression: a notation for defining a regular language

Reserved Word: see *Keyword*.

Rule Section: the second part of a scanner description program, where the input patterns to match and their corresponding actions are defined.

Scanner: see *Lexical Scanner*.

Scanner Definition File (SDF): input file to PCLEX that defines the tokens of the input language and any auxiliary processing.

Scanner Definition Language (SDL): the language used to write scanner description programs; a combination of regular expressions and C.

Scanner Definition Program (SDP): programs written in SDL that describe the lexical analysis of a target language.

Start Condition: a state used to turn some patterns on in some contexts and off in others.

Target Language: the language described by the scanner description program

Token: Smallest syntactic unit, usually recognized by scanners or lexical analysis. Examples are: identifiers, keywords, operators, terminators, and delimiters.

User Subroutine Section: third and last section of a scanner description program, where needed C functions can be included.

YACC: Yet Another Compiler-Compiler, a widely available parser generator.

%Start: PCLEX keyword for declaring start condition

%S: PCLEX keyword for declaring start condition

%s: PCLEX keyword for declaring start condition

%x: PCLEX keyword for declaring exclusive start condition

%%: Delimiters for separating different sections of a scanner description program

%{ ... %}: Delimits C definitions in definition section

< ... >: Delimits start conditions or exclusive start conditions in the rule section

{ ... }: Delimits macro names in the pattern part of a rule.

APPENDIX F. DIFFERENCES BETWEEN LEX AND PCLEX

- o The FLEX engine is used.
- o Command line format has a PCYACC flavor (see section 2).
- o Exclusive start conditions ("%x") have been added (see 3.5.2).
- o Ratfor scanners ("%r") are not supported.
- o Translation tables ("%t") are not supported.
- o Internal array sizes are dynamically resized. The "%p", "%n", "%e", "%a", "%k", and "%o" lines are ignored.
- o There is no run-time library to link to.
- o Definitions are enclosed in parentheses when expanded (see 3.4).
- o Only one input file is supported. LEX concatenates all the files in the command line (see section 2).
- o multiple actions per line are allowed (see 3.2).
- o the undocumented LEX variable "yylineno" is not supported (see the examples in section 8 for how to explicitly support "yylineno".
- o the "yymore()" and "output()" functions are not supported.

APPENDIX G. DIFFERENCES BETWEEN FLEX AND PCLEX

o Command line format has a PCYACC flavor (see section 2).
o no fast (FLEX "-f" option) or full (FLEX "-F" option) tables.
o no equivalence classes (FLEX "-ce" option) or meta-equivalence classes (FLEX "-cm" option).
o default scanner skeleton is stored internally.
o the FLEX header files ("fastsdef.h", "flexsdef.h", and "flexscom.h") are stored internal to PCLEX and output at scanner generation time.
o support -c and -C options.
o support -h options.
o interactive scanners (FLEX "-I" option) are always generated.
o the REJECT action is always supported.

INDEX

Generating Lexical Scanners with PCLEX

PCLEX® is a software product of ABRAXAS SOFTWARE®, Inc.

For more information, contact:

ABRAXAS SOFTWARE, INC.
Post Office Box 19586
Portland, Oregon 97280, USA
Phone: 503-232-0540
Fax: 503-232-0543

Email: support@pcyacc.com

www.pcyacc.com