Table of Contents

I. OVERVIEW 1

II. INTRODUCTION 3

1. FEATURES 3
2. CONVENTIONS 3
3. READING THIS MANUAL 3

III. OPERATING PROCEDURE 5

1. WRITING GRAMMAR DESCRIPTION FILES FOR PCYACC 5
2. GENERATING THE OBJECT-ORIENTED PARSERS 5
3. WRITING SCANNER DESCRIPTION FILES 7
4. GENERATING THE OBJECT-ORIENTED LEXERS 8
5. INTEGRATION OF ALL SOURCE FILES 9

IV. PCLEX 10

1. C CODE STRUCTURE GENERATED BY PCLEX 10
2. CODE GENERATED BY PCLEX IN C++ 11
3. STRUCTURE OF GENERATED C++ CODE 12
4. SYNOPSIS FOR ABXLEX CLASS 17

a. Description 17
b. Example 17
c. Public Constructor and Destructor 18
d. Public Member Functions 18

V. PCYACC 21

1. C++ CODE GENERATED WITH PCYACC C++ SKELETON 22
2. GENERATING C++ CODE BY USING PCYTOOL 26
3. SYNOPSIS FOR ABXYACC CLASS 26

a. Description 27

b. Example 27
c. Public Constructor and Destructor 29
d. Public Member Functions 30

VI. SYMBOL TABLE 31

1. INTRODUCTION 31
2. SYNOPSIS FOR ABXSYMBOLTABLE CLASS 32
a. Description 32
b. Symbol Table Entry Definition 34
c. Private Class Member 34
d. Public Constructor and Destructor 35
e. Public Member Functions 35

VII. ERROR HANDLER 38

1. INTRODUCTION 38
a. Error Reporting 38
b. Error Recovery 39
2. FUNCTIONS FOR ERROR REPORTING 39
3. FUNCTIONS FOR ERROR RECOVERY 40
4. SYNOPSIS FOR ABXERROR CLASS 41
a. Description 42
b. ABXError Class Definition 42
c. Public Constructor and Destructor 43
d. Public Member Functions 43

VIII. PARSE TREE NODE 45

1. ANALYZE PARSE TREE NODE CLASS ABXPARSETREENODE 45
2. STRUCTURE FOR ABXPARSETREENODE CLASS 47
3. STRUCTURE FOR ABXLEAF CLASS 48
4. EXPRESSION CLASSES ABXEXPRNODE 49
5. STRUCTURE FOR PARSE TREE CLASS ABXPARSETREE 51

IX. JAVA PARSER AND LEXER 53

1. INTRODUCTION 53
2. JAVA CLASS LIBRARY 56
a. JavaLex Class 56
b. JavaYacc Class 59
c. JavaError Class 61
d. JavaParseTree Class 63
(i). JavaParseTreeNode Class 63
(ii). JavaLeaf Class 64
(iii). JavaLeafList Class 64
(iv). JavaExprNode Class 65
(v). JavaExprNodeList Class 66
(vi). JavaParseTree Class 66
e. JavaSymbolTable Class 67
3. EXAMPLE 71

X. DELPHI PARSER AND LEXER 77

1. INTRODUCTION 77
2. DELPHI UNIT LIBRARY 79
a. DelphiLex Unit 79
b. DelphiYacc Unit 84
3. EXAMPLE 86

XI. VBSCRIPT PARSER AND LEXER 90

1. INTRODUCTION 90
2. STRUCTURE OF VBSCRIPT PARSER AND LEXER 92
a. VBScript Lex Modules 94
b. VBScript Yacc Modules 96
c. VBScript Error Report Modules 96
3. EXAMPLE 97

XII. PASCAL PARSER AND LEXER 113

1. INTRODUCTION 113
2. PASCAL LIBRARY 114
a. Pascal Lexer 114

b. Pascal Parser 118

XIII. BASIC PARSER AND LEXER 120

1. INTRODUCTION 120
2. STRUCTURE OF VBASIC PARSER AND LEXER 123
a. VBasic Lex Modules 124
b. VBasic Yacc Modules 126
c. VBasic Error Report Modules 126

XIV DESIGN REQUIREMENT FOR YACC 128

1. OBJECTIVE 128
2. SCOPE 128
3. COMMAND LINE OPTIONS 128

XV DESIGN REQUIREMENT FOR LEX 132

1. OBJECTIVE 132
2. SCOPE 132
3. COMMAND LINE OPTIONS 132

APPENDIX I. HOW TO CREATE C PARSER AND LEXER 136

1. COMMAND LINE FORMAT FOR PCYACC 136
2. COMMAND LINE OPTIONS FOR PCYACC 136
3. COMMAND LINE FORMAT FOR PCLEX 138
4. COMMAND LINE OPTIONS FOR PCLEX 138
5. DEFAULT SKELETON FILE 139

APPENDIX II. HOW TO CREATE C++ PARSER AND LEXER 140

1. COMMAND LINE FORMAT FOR PCYTOOL 140
2. COMMAND LINE OPTIONS FOR PCYTOOL 140

3. COMMAND LINE FORMAT FOR PCLTOOL 140
4. COMMAND LINE OPTIONS FOR PCLTOOL 141
5. DEFAULT SKELETON FILE 141

APPENDIX III. HOW TO CREATE JAVA PARSER AND LEXER 142

1. COMMAND LINE FORMAT FOR PCYTOOL 142
2. COMMAND LINE OPTIONS FOR PCYTOOL 142
3. COMMAND LINE FORMAT FOR PCLTOOL 142
4. COMMAND LINE OPTIONS FOR PCLTOOL 143
5. DEFAULT SKELETON FILE 143

APPENDIX IV. HOW TO CREATE DELPHI PARSER AND LEXER 144

1. COMMAND LINE FORMAT FOR PCYTOOL 144
2. COMMAND LINE OPTIONS FOR PCYTOOL 144
3. COMMAND LINE FORMAT FOR PCLTOOL 144
4. COMMAND LINE OPTIONS FOR PCLTOOL 145
5. DEFAULT SKELETON FILE 145

APPENDIX V. HOW TO CREATE PASCAL PARSER AND LEXER 146

1. COMMAND LINE FORMAT FOR PCYTOOL 146
2. COMMAND LINE OPTIONS FOR PCYTOOL 146
3. COMMAND LINE FORMAT FOR PCLTOOL 146
4. COMMAND LINE OPTIONS FOR PCLTOOL 147
5. DEFAULT SKELETON FILE 147

APPENDIX VI. HOW TO CREATE VISUAL BASIC SCRIPT PARSER AND LEXER 148

1. COMMAND LINE FORMAT FOR PCYTOOL 148
2. COMMAND LINE OPTIONS FOR PCYTOOL 148

3. Command Line Format for PCLTOOL 148
4. Command Line Options for PCLTOOL 149
5. Default Skeleton File 149

APPENDIX VII. HOW TO CREATE BASIC PARSER AND LEXER 150

1. Command Line Format for PCYTOOL 150
2. Command Line Options for PCYTOOL 150
3. Command Line Format for PCLTOOL 150
4. Command Line Options for PCLTOOL 151
5. Default Skeleton File 151

APPENDIX VIII. ERROR MESSAGES FOR PCYTOOL 152

Error Code: Error Message and Explanation 152

APPENDIX IX. ERROR MESSAGES FOR PCLTOOL 154

Error Code: Error Message and Explanation 154

APPENDIX X. BIBLIOGRAPHY 156

I. OVERVIEW

PCLEX and **PCYACC** are widely used software tools for developing compilers. Currently, almost all **PCYACC** generate output codes in C. It misses the well known advantages of object oriented programming, *e.g.*, data abstraction, encapsulation and inheritance. **PCYACC OBJECT ORIENTED TOOLKIT Library** offers an object-oriented version of lexical analyzers, syntax parsers, and error handling facilities. In the construction of compilers, symbol table management is a commonly used technique. In this library, parse tree and symbol table classes are also provided as the tools of compiler construction. **PCYACC OO TOOLKIT Library** provides five basic classes:

1). **Lexical Analyzer Class**: This class serves as a code skeleton for **PCLEX**.

2). **Syntax Parser Class**: This class supports syntactic parser **PCYACC**.

3). **Symbol Table Class**: This class is used for symbol table management.

4). **Error Handling Class**: This class is responsible for error reporting.

5). **Parse Tree Class**: This class is available when user wants to construct parse trees.

PCYACC OO TOOLKIT Library takes advantage of data abstraction, encapsulation and inheritance in object oriented software design. The structure of our **PCYACC OO TOOLKIT** Library is shown below:

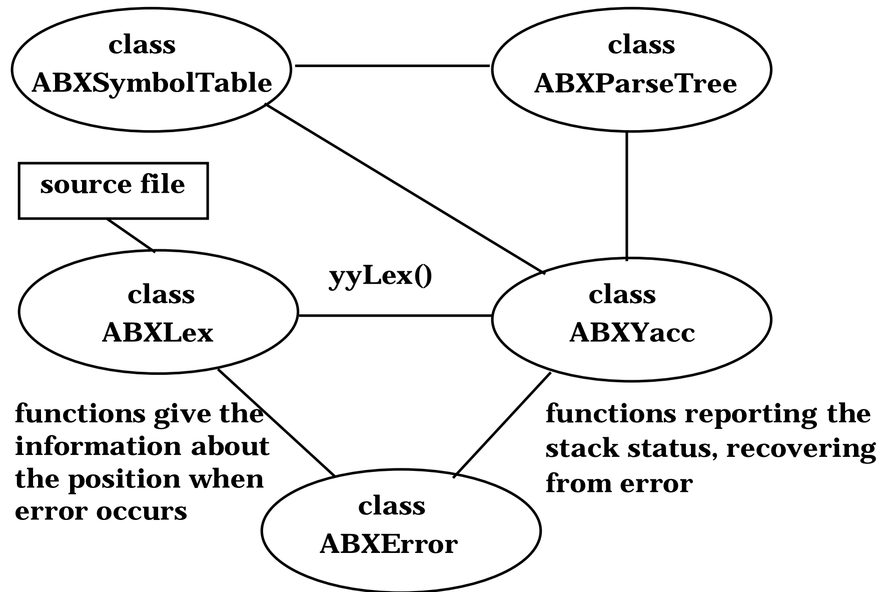

Figure 1-1. Structure of PCYACC OO TOOLKIT

More detailed explanation about our **PCYACC OO TOOLKIT** Library will be provided from section IV through section VIII from both theoretical and practical points of view.

II. INTRODUCTION

1. Features

PCLEX and **PCYACC** have proven to be very efficient and elegant tools for constructing compilers and interpreters for decades. At the time it was born, C was the only target language. With the emergence of object oriented programming languages, especially C++, the programming world quickly switched gears to take advantage of the new programming method. To incorporate C++ features into lexical analyzers and syntactic parsers, C++ classes can be inserted into user defined part in the processing. The resulting lexical analyzers and syntactic parsers are in C++ and they take the advantages of the object oriented programming features. Here we provide a package called **PCYACC OO TOOLKIT Library** which provides five classes,

- **ABXLex** (a class for lexical analyzer)
- **ABXYacc** (a class for syntactic parser)
- **ABXSymbolTable** (a class for symbol table management)
- **ABXError** (a class for error reporting and error recovery)
- **ABXParseTree** (a class for parse tree)

However, due to the specialty of **PCYACC** and **PCLEX**, the classes that will be utilized by the users in their C++ code are generated from a special form of source files, scanner description files and grammar description files. It will be impossible to define a general class for all the objects in the same category as other class libraries usually do. We will see the reasons in detail later.

2. Conventions

All **Abraxas Software** class names start with the letters "**ABX**". All function names start with a lower case letter, followed by uppercase letters and underscores. For example, YACC class is named as **ABXYacc** and parser function is **yyParse()**. To make it easy to remember and to understand, abbreviations are not used.

3. Reading This Manual

This manual is intended to serve as tutorial book for **PCYACC** Object Oriented Toolkit Library. It describes all the classes and member functions embedded in this library and how to use **PCYTOOL** and **PCLTOOL** utility

programs to create parsers and lexers in different languages. The discussion about different classes is generally more detailed than what is necessary for actual use. The reason is that this manual will help the user get familiar about not only the usage of **PCYACC OO Toolkit Library** itself but also understanding its internal design for further upgrading later.

III. OPERATING PROCEDURE

Abraxas Software provides the **PCYACC** and **PCLEX** tools to create C parsers and lexers. With the appearance of C++ and object oriented programming language it is a trend in the industry to upgrade products from non-object oriented design to object oriented design. Thus taking advantage of the fact that C++ programming language provides data encapsulation, inheritance, etc. C++ **PCYACC** classes provided by **Abraxas Software** are available for the user to create C++ lexers and parsers. The following describes the operation of how to use our new **PCYACC OO TOOLKIT Library** to create C++ version of lexer and parser in more detail.

1. Writing Grammar Description Files for PCYACC

If you only need a single parser in your main program there is no change in the requirements for the grammar description file.

If you plan to generate multiple parsers and use them in the same main program one grammar description file should be generated for each parser you wish to have. The main function should not be included in any one of the grammar description files. Instead, a separate **.cpp** file should be generated for the main function. In order for lexers to support multiple parsers, it is **required** that tokens defined in the grammar files should be different for each parser. Otherwise redefinition of tokens will occur which makes it impossible to support multiple parsers. Referencing external variables or functions defined in a parser or lexer description file in any user function is also discouraged.

2. Generating the Object-Oriented Parsers

There are two methods to create C++ parser:

- Use class skeleton file with -p on the **PCYACC**command line.
- Use utility program **PCYTOOL** to translate C parser into C++ parser with default class skeleton file.

In the first approach, simply make use of the -p command line option of **PCYACC** (assuming DOS **PCYACC**) to include the skeleton C++ code “pcy_sk.cpp”. Everything else is the same as the procedure for generating C code. Users can insert their own skeleton class definition files in place of "pcy_sk.cpp".

The file translation is illustrated as follows, assuming the grammar description filename is “myparser.y”,

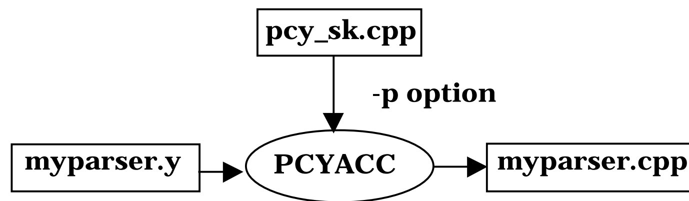

Figure 3-1. Generate C++ parser by using PCYACC directly with -p option on

The default skeleton file provided is "pcy_sk.cpp". If you would like to use your own class skeleton file, simply follow -p option with your own skeleton filename. Everything else is the same as the usual procedure of using the **PCYACC** tool.

The second approach provided by **Abraxas Software** is focused on separation of the procedure for creating C++ parser. This can make it easy for the user to understand how the utility program **PCYTOOL** works with C parser. Also the new -k option can be used to create some other language parsers like JAVA, Borland Delphi, Basic, Pascal, etc. For a detailed description of -k options, check later sections. The default -k option is -k1 for creating a C++ parser.

There are two separate procedures for creating a C++ parser. In the first phase, based on the availability of grammar description file, simply invoke **PCYACC** tool on the command line to create a C parser. Once the C parser is generated by **PCYACC** a utility program named **PCYTOOL** is needed for generating a C++ parser in the second phase. The **PCYTOOL** uses the default class skeleton file “**pcy_sk.cpp**” that is actually the parser class declaration file. The corresponding class header file name is “**pcy_sk.hpp**”.

If the grammar description file is named “myparser.y”, the following diagram shows how the second approach creates the C++ parser.

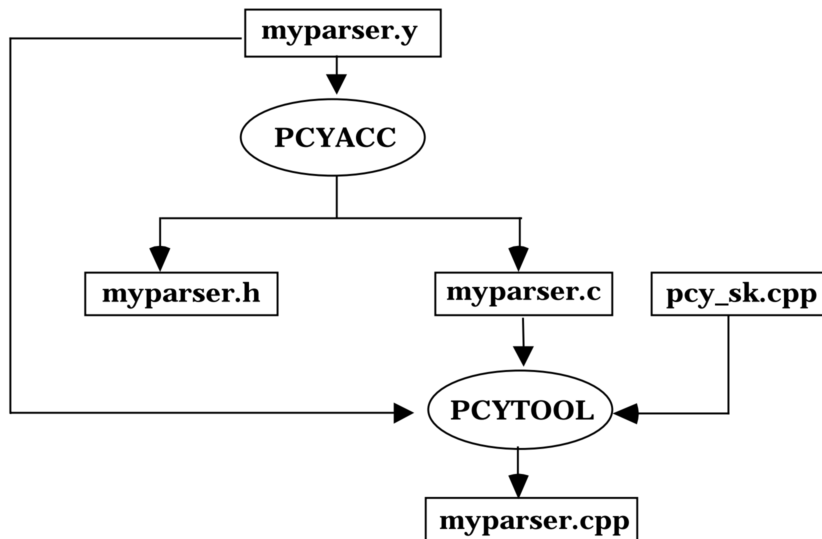

Figure 3-2. Diagram of C++ parser generated by PCYTOOL

The second approach for creating a C++ parser looks inconvenient for the user. However, separation of the procedure of generating a C parser and a C++ parser makes it easier to understand the process of generating a C++ parser. It will also be convenient to implement other language parsers by simply providing -k option that can tell **PCYTOOL** in which language the parser will be generated.

3. Writing Scanner Description Files

If you only need a single lexer in your main program there is no change in the requirements for the scanner description file. Include the corresponding header file (e.g., y1.h) for the tokens expected by the supported parser (e.g., y1.y).

If you plan to generate multiple lexers and use them in the same main program, one scanner description file should be created for each lexer you wish to have. Each lexer file should include the corresponding token definition header file for the supported parser. A lexer can support only one set of tokens. No lexer externals should be referenced in user's main function.

4. Generating the Object-Oriented Lexers

There are two methods for creating a C++ lexer:

- Use a class skeleton file with -**p** on in the **PCLEX** command line.
- Use the utility program **PCLTOOL** to translate a C lexer into a C++ lexer with a default class skeleton file.

In the first approach simply make use of the -**p** command line option of the **PCLEX** (assuming **DOS PCLEX**) to include the skeleton C++ code “pcl_sk.cpp”. Everything else is the same as when generating C code.

The file translation is illustrated below, assuming the scanner description filename is “mylexer.l”,

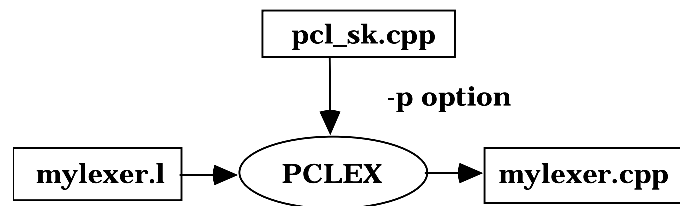

Figure 3-3. Generate C++ lexer by using directly with -p option on

The default skeleton file provided is "pcl_sk.cpp". If you would like to use your own class skeleton file simply follow -p option with your skeleton filename. Everything else is the same as the usual procedure of using **PCLEX** tool.

The second approach provided by **Abraxas Software** is focused on the separation of the procedure for creating a C++ lexer. This can make it easy for the user to understand how the utility program **PCLTOOL** works with the C lexer. Also the new -k option can be used to create some other language lexers like JAVA, Borland Delphi, Basic, Pascal, etc. For details about the -k option, please check later sections. The default -k option is -k1 for creating a C++ lexer.

There are two separate procedures to create a C++ lexer. In the first phase, based on the availability of scanner description file, simply invoke the **PCLEX** tool on the command line to create the C lexer. Once the C lexer is generated by **PCLEX**, a utility program named **PCLTOOL** is needed for generating a C++ lexer in the second phase. The **PCLTOOL** uses the default

class skeleton file "**pcl_sk.cpp**" that is actually the lexer class declaration file. The corresponding class header file name is "**pcl_sk.hpp**".

If the scanner description file is named "mylexer.l", the following diagram shows how the second approach creates a C++ lexer.

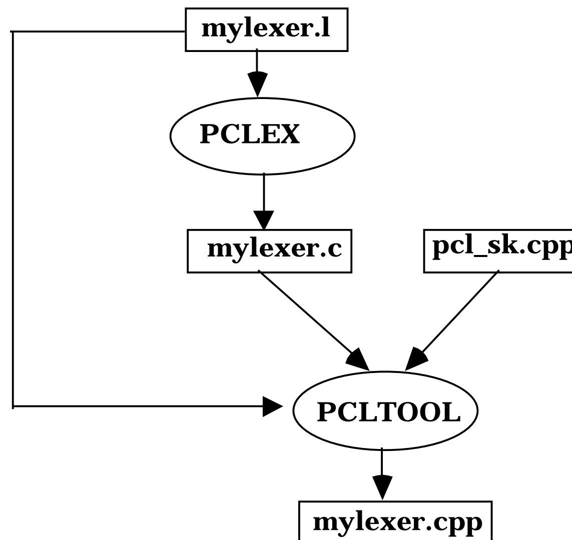

Figure 3-4. Diagram of C++ lexer generated by PCLTOOL

The second approach creating a C++ lexer looks inconvenient for the user. However, separation of the procedure of generating a C lexer and a C++ lexer makes it easier to understand the process of generating C++ lexer. It will also be convenient for you to implement other language lexers by simply providing the -k option that can tell **PCLTOOL** which language the lexer will be generated in.

5. Integration of All Source Files

All the source files generated in step 2 and step 4 plus the user's main function file form the complete set of source code for generating a parser application.

IV. PCLEX

PCLEX™ is a program generator for writing lexical analyzers. A lexical analyzer reads a stream of characters and separates the stream into symbols of a target language, also known as tokens. **PCLEX** translates a lexical analyzer (lex(.l) file) written in the **Scanner Description Language** (**SDL**) into C language. SDL is a special high level language oriented toward string matching. Scanner descriptions can be extended with code sections written in C or C++ to accommodate the different needs of different languages. SDL allows software developers to concentrate on what the scanner recognizes instead of dragging in the details of how. It can reduce the work necessary to bring a project to completion. Almost all the LEXes use C as the host language. To explore the advantages of an object oriented programming language like C++ we provide a lexical scanner class as a skeleton that can be inserted to the output file by specifying the **-p** option on the **PCLEX** command line. After running **PCLEX** on the SDL file (lex(.l) file) with the option **-p** on, the generated code will be in C++.

1. C Code Structure Generated by PCLEX

The C code of a lexical analyzer generated by **PCLEX** has the following layout,

1. Macro definitions
2. Code from section 1 of .l file
3. Data tables
4. Global variables
5. Auxilary functions
6. Function yylex()
7. Code from section 3 of .l file

Figure 4-1. Structure of Code Generated by PCLEX

1). <u>Macro definitions</u>: This part defines some macros that are actions of the lexical analyzer. It defines symbolic names and gives users a chance to redefine them to meet their special requirements.

2). <u>Code segment copied directly from the declaration section of lexer file</u>: This part contains the macros defined by user, the declarations of variables,

functions and types to be used in embedded actions. It varies with different lexical analyzers and it is optional.

3). Data tables: This part consists of the data tables for driving the **Deterministic Finite Automaton** (DFA) simulator. They are different for different lexical analyzers.

4). Global variables: This part consists of a variable representing the input stream buffer and pointers that indicate the status of input being scanned. They should be almost the same for different lexical analyzers.

5). Auxiliary functions: This part defines the functions that are only called by lexer function **yylex()**.

6). Function **yylex()**: This part defines the function yylex().

7). Code segment copied directly from the function section of lexer file: This part contains the possible function definitions by the user. It is also optional.

2. Code Generated by PCLEX in C++

There are two approaches for getting the C++ lexical analyzer. 1). Explore the convenience of option **-p** of **PCLEX**, which creates a C++ version of lexical analyzer skeleton by using **pcl_sk.cpp** skeleton file. 2). Modify the C lexer generated by **PCLEX** to convert it to a C++ lexer.

The first approach is quick and its operation is more familiar to users. The second approach allows new features to be added to **PCLEX**. For example, we can create a utility program **PCLTOOL** that will convert the C lexer code generated by PCLEX to be C++ lexer code by using default option -k1. And lexers in other languages can be generated by supporting the -k option.

The ideal situation is to create a general class **ABXLex** that fits all lexical analyzers. Just like all the sets of objects are instances of the same class **SET**. However, there are major obstacles for us to reach the goal. A lexical analyzer is a simulator of **DFA**, it depends on the data tables to drive its transition. For each specific lexical analyzer, it has its own data tables. In other words, data tables vary with the individual analyzers. These data tables have the values determined after **PCLEX** scanned the lex(.l) file. They are referenced by function **yylex()**, and they are always **read-only**. If we have a general class for all lexical analyzers every instance of this class will have the same copies of these tables. But each class instance should be different because the initial values of these tables are decided after running **PCLEX**. This is one reason why each lexical analyzer should have one class. Another reason is the embedded user actions. The user actions also vary with

different lexical analyzers. These two reasons make it almost impossible to have a general class for all lexical analyzers.

Besides the data tables, the other important parts of a lexical analyzer are the input buffer and the pointers indicating the scanning position. To separate the input stream into tokens we need to give both the value representing the token and the token itself. The value representing the token will be returned by function **yylex()**. The token will be a variable of type **YYSTYPE**.

3. Structure of Generated C++ Code

Like the C code generated by **PCLEX**, the C++ code also has a certain structure. In the C code some functions and variables used as global variables and functions will be moved into the class to become member variables and functions in the C++ code. The C++ lexer has the structure listed below, assuming that the class definition resides in file **pcl_sk.hpp**, which is the default header file.

```
001:    #include “pcl_sk.hpp”
002:    Code from section 1 of .l file
003:    Data tables
004:    Definition of the lexical analyzer class’ public
           member functions
```

Header file **pcl_sk.hpp** contains the definition of the default lexical analyzer class **ABXLex.** All these data tables are **read-only** for the lexical analyzer. This makes it possible to have all instances of that class share the same copies of these large tables without each needing to have a copy.

The structure of class **ABXLex** is shown below:

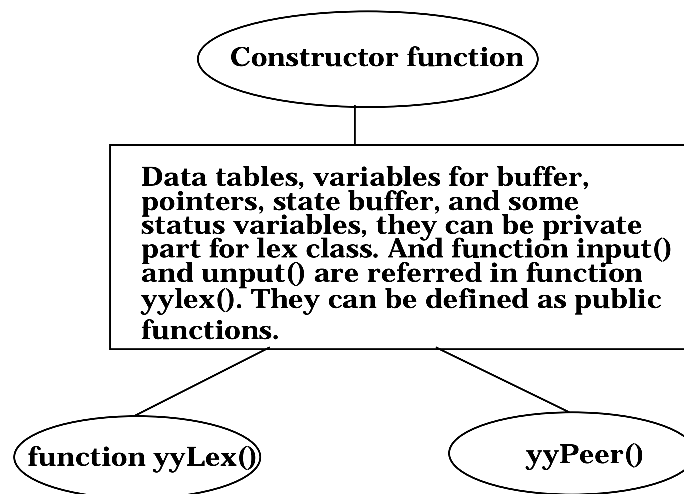

Figure 4-2. Class Structure of Generated C++ Code

```
001:    Private part:
002:        Declare variables YYlval and YYval
003:        Declare data tables
004:        Declare YY_JAM and YY_JAM_BASE variables used
               for lexer
005:        Define macros in C lexer that will be used for
               C++ lexer
006:        Declare input streams
007:        Declare buffer and buffer pointers
008:        Declare a variable yyLineNo counting the line
               number of input
009:        Declare a variable yyText containing the content
               of lexer matching pattern
```

YYlval and **YYval** variables are internal variables for **ABXLex** class. These represent **yylval** and **yyval** variables in a C lexer respectively.

Data tables are declared as **read-only** arrays. These arrays are **const** for the lifetime of particular **ABXLex** object, but not for the **ABXLex** class as a whole.

YY_JAM and **YY_JAM_BASE** variables are private members of the **ABXLex** class which are used by lexing function. These two variables are defined as macros in a C lexer by **PCLEX** tool.

Macros defined in a C lexer will be put into the **ABXLex** class as its private data members. Putting C lexer macros into a class scope is due to the characteristics of data encapsulation from the Object Oriented Programming technique.

Four types of input streams are provided in **ABXLex** class. These are input from **stdin**, file, character string or **istream** respectively.

The buffer is the memory segment where input stream stays for scanning. It can only be accessed by the lexer function **yyLex**. The size of the buffer is decided by a constant, which is defined by a macro. To change the size of the buffer we just need to redefine the macro to a new constant. The pointer indicates the position of the character that has been scanned most recently.

The line number **yyLineNo** is used to keep track of the actual input line number the scanner has processed.

Variable **yyText** contains the content of the lexer matching pattern. The corresponding variable in a C lexer is **yytext**.

```
001: Public part:
002:      Declare constructor function
003:      Declare destructor function
004:      Declare function get_yyLineNo()
005:      Declare function get_yyText()
006:      Declare function get_yyBufferPtrC()
007:      Declare function input()
008:      Declare function unput()
009:      Declare function set_YYSTYPEInstance()
010:      Declare function yyCheck()
011:      Declare function yyInit()
012:      Declare function yyLex()
013:      Declare function yyPeer()
014:      Declare function yySetBuffer()
015:      Declare function yySetInput()
016:      Declare function yySearch()
017: Protected part:
018:      Declare virtual function yyWrap()
```

Currently, we consider four types of input stream to the **ABXLex** objects. Each type of input stream has its own corresponding constructor function. All the private data members for **ABXLex** class are defined inside the constructor.

The destructor function simply frees all the memory that is allocated in the constructor.

Function **get_yyLineNo** returns the current line number in the input stream. This information is very useful especially when you want to report an error message.

Function **get_yyText** returns the lex token buffer which contains the current token processed by the lexer.

Function **get_yyBufferPtrC** returns the current token buffer index processed by the lexer.

Function **input** fills the input buffer of the lexer which gets the next character from input.

Function **unput** puts a character back in the logical input stream.

Function **set_YYSTYPEInstance** will set two instances of **YYSTYPE** that is shared between **ABXLex/ABXYacc** instances.

Function **yyCheck** will output the content of the lexeme presently being examined by the lexer. It maintains its own line number count and counts line number, whenever a new line character '\n' is encountered. It also should have the ability to manage the output formatting.

Function **yyInit** resets the position of the buffer pointers and clears the buffer. This function is useful when the lexer needs to switch the input from one file to another.

Function **yyLex** is actually a lexer that will return a token to a parser whenever it needs a token from the input stream.

Function **yyPeer** allows users to get the next n characters from the input stream. The number of characters that will be fetched is specified by the first parameter of this function. The fetched characters will be stored in a string that is specified by the second parameter of the function.

Function **yySetBuffer** allows the user to reset the input buffer size according to project requirement. The buffer size can be modified to shrink or enlarge. This makes it convenient for the user to simply call the member function to change it instead of doing it by "hand".

Function **yySetInput** enables a lexer to switch input from one file to another even in the middle of processing a file. To do this, users must take care of the backup and restoration of the buffer and function. **yySetInput** will do this work.

Function **yySearch** is provided by the user. This will search the token list based on the current token from the lexical analyzer.

Function **yyWrap** always returns 1. This indicates the program is done and there is no more input.

We can also take the second approach to create a C++ lexer by using **PCLTOOL** with the default k1 option on. However, you should create a C lexer with **PCLEX** first. **PCLTOOL** will hook the C lexer and insert the default C++ lexer skeleton into it. A C++ version lexer is generated in the way described above. The following gives an example of the command line to create a C++ lexer, however you have to make sure that the C++ lexer skeleton files (**pcl_sk.cpp** and **pcl_sk.hpp**) are residing the current working directory.

```
PCLTOOL –k1 yacc.h lex.l lex.c
Or
PCLTOOL –k1 lex.c
```

yacc.h is a yacc header file including tokens and **YYSTYPE** union definition. **lex.l** is a scanner description file. **lex.c** is a lexical analyzer generated by **PCLEX**.

As a result, users get the source code file **lex.cpp**. To declare any instance of class **ABXLex**, you just need to include the default header file **pcl_sk.hpp** in the source code to make the class defined.

The structure of this generated C++ code in file lex.cpp is listed as follows:

```
001: #include “pcl_sk.hpp”
002: Code copied from section 1 of file lex.l (if there is any)
003: YY_JAM and YY_JAM_BASE constants
004: Data tables
005: ABXLex::ABXLex(FILE *, FILE *) { }
006: ABXLex::ABXLex(istream *, FILE *) { }
007: ABXLex::ABXLex(char *, FILE *) { }
008: ABXLex::~ABXLex() { }
009: ABXLex::get_yyLineNo(void) { }
010: ABXLex::get_yyText(void) { }
011: ABXLex::input(void) { }
012: ABXLex::unput(int) { }
013: ABXLex::set_YYSTYPEInstance(YYSTYPE *, YYSTYPE *) { }
014: ABXLex::yyCheck(char *, int) { }
015: ABXLex::yyInit(void) { }
016: ABXLex::yyLex(void) { }
017: ABXLex::yyPeer(int, char *) { }
018: ABXLex::yySearch(char *) { }
```

```
019: ABXLex::yySetBuffer(int) { }
020: ABXLex::yySetInput(FILE *, FILE *) { }
021: ABXLex::yyWrap(void) { }
022: Code copied from section 3 of file lex.l (if there is any)
```

The C++ header file **pcl_sk.hpp** contains the definition of the class **ABXLex**.

4. Synopsis for ABXLex Class

```
#include "pcl_sk.hpp"
ABXLex* mylex;
```

Assume that the lexical analyzer skeleton provided by **Abraxas Software** declares the lexical analyzer class as **ABXLex** and the lexer file has the name **lex.l**. **PCLTOOL** will generate a C++ file **lex.cpp** that contains the definitions of member functions of the class **ABXLex**. And the header file **pcl_sk.hpp** contains the definition of the class **ABXLex**.

a. Description

Although we have two approaches to generate a C++ lexer, the second one is recommended by **Abraxas Software**. By using the second method users have a choice to generate lexers in other languages.

The library has a class **ABXLex** that provides a C++ version skeleton of the lexical analyzer. After scanning a C code lexer file successfully, **PCLTOOL** generates a C++ file. This defines a lexical analyzer class. Besides the class definition, the data tables will be declared as static arrays that can only be accessed by the instance of lexical analyzer class.

b. Example

Assume that the **SDL** file that describes the lexer has the file name **lex.l** and that you intend to name the lexical scanner class **ABXLex**. Then you just need to use the default skeleton file **pcl_sk.cpp**. After the C lexer was processed by **PCLTOOL**, file **lex.cpp** is generated. In this file the member functions of class **ABXLex** are defined, such as **yyLex()**. The lexical analyzer class can be used as follows:

```
001:    #include "pcl_sk.hpp"
002:    #include <stdlib.h>
003:
004:    FILE *yyin;
005:    main()
006:    {
007:        ABXLex* mylex = new ABXLex(yyin, stdout);
```

```
008:        mylex->yyLex();
009:    }
```

c. Public Constructor and Destructor

```
ABXLex(FILE* inputFile, FILE *outputFile);
ABXLex(istream* inputStream, FILE* outputFile);
ABXLex(char* inputString, FILE* outputFile);
```

Initialize all **ABXLex** class data members. The first parameter for this function is a file pointer to an input file, a pointer to istream, or a character string. A file pointer to an output file is its second parameter.

```
~ABXLex();
```

This function frees the allocated memory used by **ABXLex** class data members.

d. Public Member Functions

```
int get_yyLineNo(void);
```

This function returns the current line number that is very useful in error reporting.

```
char* get_yyText(void);
```

This function returns the current token in the lex token buffer.

```
int get_BufferPtrC(void);
```

This function returns the current index of token buffer.

```
int input(void);
```

This function returns the next character from input.

```
void unput(int);
```

This function puts a character back in the logical input stream.

```
void set_YYSTYPEInstance(YYSTYPE *, YYSTYPE *);
```

This function will set two instances of **YYSTYPE** (**yylval** and **yyval**) shared between **ABXLex/ABXYacc** instances.

```
void yyCheck(char *, int);
```

This function examines the current lexeme token, and returns its content and location information in input stream.

```
void yyInit(void);
```

Initialize the input buffer. Reset the input buffer pointer to indicate the buffer is empty.

```
int yyLex(void);
```

Separate an input character stream into tokens following the patterns specified in the lex(.l) file. This function varies for each different lexical analyzer because the embedded user actions defined in the lex(.l) file are different. This is the reason why we cannot have a general class for all the lexical analyzers. This function only returns an integer that represents a token. However, the user still needs to take care of the value of the token. The value of a token is passed to the syntactic parser via variable **yylval**. The type of this variable is **YYSTYPE** that is defined by a **typedef** or **union** declaration at the beginning of a grammar file.

```
void yyPeer(int n, char *s);
```

Look ahead to the n*th* character from the current scanning position. The number of characters to get from input stream is defined as the first parameter of this function. The actual fetched characters are stored in a string that is defined as the second parameter.

```
void yySetBuffer(int);
```

This function takes an integer representing the buffer size defined by the user as input. The input buffer size is set based on this parameter.

```
void yySetInput(FILE *fp);
```

Allows the lexer to switch scanning from one input file to another. This enables the scanner to be interrupted to process another input instead of the current one.

```
int yySearch(char* str);
```

This function is provided by the user, which will return a token value according to the input string. Since every lexer is associated with a different set of tokens determined by the parser, we can not provide a general function for the customer to get a current token value.

```
int yywrap(void);
```

This function can be modified by the user. The default **yyWrap()** always returns 1 which indicates the program is done and there is no more input.

V. PCYACC

Similar to **PCLEX**, **PCYACC** accepts a grammar file as input and generates a syntactic parser for a specific language. After processing the grammar file, it generates a C file for the syntactic parser. The generated C file has the layout as shown below.

1. Code copied from the first section of .y file
2. Macro definitions
3. Code copied from the third section of .y file
4. Data tables
5. Macro definitions and data declarations
6. Function: yyparse

Figure 5-1. Structure of the code generated by PCYACC

1). Code copied directly from the first section of yacc (.y) file: This part declares the global variables to be used in user actions, includes the necessary header files for the library functions to be used and defines the macros.

2). The macro definitions for tokens: This part contains the list of all the constants representing the terminal symbols declared in lines by using the %token.

3). Code copied directly from the third section of yacc (.y) file: This part contains user defined functions.

4). Data tables: This part consists of two parts, a parsing action function *action* and a goto function *goto*. They are definitely different for each individual syntactic parser.

5). Macro definitions and data declarations: This part contains macro definitions and data declarations used by yyparse function.

6). Function **yyparse()**: This is the actual syntactic parser that will translate a Grammar Description File into a C file. It works exactly the same way as a translator or interpreter that can translate a special grammar language into the C programming language source file.

To generate C++ code for the syntactic parser there are two approaches:

- Supply a C++ skeleton to **PCYACC**.
- Generate a C parser by **PCYACC**. Convert it into a C++ parser by using **PCYTOOL**.

The first approach simply explores the advantage of the option **-p** of **PCYACC**. By modifying the skeleton file, users can get syntactic parser classes for their special purpose. The second approach needs to use a utility program **PCYTOOL**, which will hook the C parser generated by **PCYACC** with the skeleton file and translate a C parser into a C++ parser (assume using -k1 default option on **PCYTOOL** command line). **PCYTOOL** will deal with everything regarding this translation according to the -k option setting.

1. C++ Code Generated with PCYACC C++ Skeleton

PCYACC provides a command option **-p** that enables us to generate a parser by using a user-provided skeleton. With this option, we can make **PCYACC** generate a parser class. We provided such a skeleton file that contains the definition of the member functions of the parser class and also the macro definitions that will be used in those member functions. The standard skeleton file name is **pcy_sk.cpp**. We also provide a corresponding header file named **pcy_sk.hpp** that contains the definition of the parser class with standard name **ABXYacc**.

The layout of generated C++ code is shown as below.

#include "pcy_sk.hpp"
Code copied directly from section 1 of .y file (if there is any)
Code copied directly from section 3 of .y file (if there is any)
Macro definitions representing tokens
Data tables
PCYACC macros
Class ABXYacc

Figure 5-2 Layout of Generated C++ code

Because data tables are only read by the parser function they can be shared by all instances of the syntactic parser class. So it is not necessary for each instance to keep its own copy of these tables. This will make the class more efficient.

The structure of the syntactic parser class:

```
001: Private part:
002:      State stack
003:      Value stack
004:      Stack pointers
005:      Middle storage
006:      Parsing table
007:      YACC constants
008: Public part:
009:      Constructor function
010:      Destructor function
011:      Function yyParse()
012:      Member functions
```

The constructor function of the parser class establishes the stacks, initializes the pointers of the stacks. The constructor's name is decided by the class name. The standard syntactic parser class name provided by the **PCYTOOL** is **ABXYacc**. It is defined in the header file **pcy_sk.hpp**. An **ABXLex** class instance will be passed to the **ABXYacc** constructor as its parameter, which will provide tokens from the input stream for the **ABXYacc** class instance. The class member functions are defined within the file **pcy_sk.cpp** that is used as a **PCYACC** skeleton. If you want to have a syntactic parser class

with a different name, you need to make a copy of the header file **pcy_sk.hpp**. Then edit this header file, change the class name from **ABXYacc** to the name you want and change the names of constructor and destructor accordingly. You also need to make a copy of the skeleton file **pcy_sk.cpp** and change the class name, constructor function and destructor function's names to make them consistent with the names in header file.

Traditionally, the state stack is designed to be interior to the parser function. This is mainly because the state stack is not as significant as the value stack. In case the parser spots an error, it may call some error handling mechanism. To enable the error handler to peer and change the state stack, it is better to make them accessible to error handler. For example, it will be very helpful to report the current state of the stacks.

The destructor function does the reverse of the constructor function. It cleans the stacks and destroys the memory allocated for the stacks.

One thing that needs to be considered is which lexical analyzer feeds the stream of tokens to the syntactic parser. Traditionally, the parser function **yyparse()** calls the lexer function **yylex()** directly whenever it needs a token for future parsing. Then the lexer function will go to scan the input stream and get the next token. Since the lexer function is called by the parser function we may make the lexer function **yylex()** a global function so that **yyparse**() could call **yylex()** directly. However, it will be difficult for a parser to handle multiple lexers because we have to modify the parser function to switch among several lexer functions. To make this easy, we set a pointer to an integer function as a part of the class, and define a class method **yySetLexer** to switch between lexers. The class declares a pointer to an integer function that is called by **yyParse**. To set a lexical scanner using function **yySetLexer**, we need to specify the parameter for this function as a new lexer object reference for the parser.

The other side of interaction with a lexical analyzer is to transfer the value of the scanned token, which is carried by variable **yylval**, with type **YYSTYPE**. **YYSTYPE** is defined at the beginning of the **GDL** file. It is desirable that each instance of the parser class has its own copy of **yylval**. As mentioned before, variable **yylval** and **yyval** have to be shared between **ABXLex/ABXYacc** class instances. Since the user is already familiar with these two global variables we need to find a way to make these two variables shared by lexer and parser objects. **ABXLex** and **ABXYacc** classes should have correspondent data members that can be accessed by using member functions. The detailed implementation is as follows:

Add two macro definitions in pcy_sk.hpp:

```
#define yylval *YYlval
#define yyval *YYval
```

Declare two private data members for **ABXYacc** class:

```
YYSTYPE *YYlval;
YYSTYPE *YYval;
```

The same scheme is applied to **ABXLex** class. Define two macros and declare two data members for **ABXLex** class in **pcl_sk.hpp**.

```
#define yylval *YYlval
#define yyval *YYval
YYSTYPE *YYlval;
YYSTYPE *YYval;
```

The **ABXLex** class needs a method called:

```
void ABXLex::set_YYSTYPEInstnace(YYSTYPE *yys, YYSTYPE
*yyss)
{
     YYlval = yys;
     YYval = yyss;
}
```

The **ABXYacc** constructors need to call:

```
LexObject->set_YYSTYPEInstance(YYlval, YYval);
```

Thus, global variable **yylval** and **yyval** can be shared between **ABXLex/ABXYacc** instances. If a lexical analyzer is chosen by a parser to get the token from the input stream, **ABXYacc** class should include the following component:

```
Public:
     void yySetLexer(ABXLex *);
```

To get the corresponding lexer that the parser requested, the following steps should be included in the place where the parser is invoked (assuming a string buffer will provide token input):

```
ABXLex* mylex = new ABXLex(buf, stdout);
myyacc->yySetLexer(mylex);
```

In this way, a lexer function can be chosen for the parser explicitly. It is quite convenient to have a parser to switch the lexer that it will get a token from.

Function **yyParse** is the core part of the syntactic parser class. It is a **LALR** parser for the designed language that is defined by the grammar rules in the grammar description file (yacc(.y) file). It takes tokens provided by lexical scanner as input. Its pattern is fixed. However, since the embedded user actions regarding the specific grammar rules will be inserted into function body, this means that for each lexical scanner class, function **yyParse** is different from one of another lexical scanner class. This is one reason why we could not provide a general class for all lexical scanners.

Function **yyParse** produced with the skeleton works in exactly the same way as **yyparse()** function automatically generated by **PCYACC**, except that all the data members used for the parser have been stored as a private part of **ABXYacc** class. The only way to access these data members is to use member functions of **ABXYacc** class or user supplied methods in **ABXYacc** class.

2. Generating C++ Code by Using PCYTOOL

Above is stated the scenario of the syntactic parser class generated by applying option -**p** on a skeleton file on **PCYACC** command line. We can also take a second approach to make **PCYTOOL** generate C++ code. The parser class definition file defaults to name "**pcy_sk.hpp**". Its class member function definition file name will be "**pcy_sk.cpp**". The user can change these file names or use their own parser class based on their needs. For more detailed option settings in **PCYTOOL** command line, please check later sections.

Since we have a very reliable C parser, it really makes sense to create a utility program **PCYTOOL** to translate a C parser into a C++ parser. In this way, parser global data tables and variables can be moved from global scope into class scope. This makes it possible to support multiple parsers easily.

The procedure that **PCYTOOL** creates a C++ parser will be described in detail in later sections.

3. Synopsis for ABXYacc Class

```
#include "pcy_sk.hpp"
ABXYacc* parser;
```

Assume that the grammar file has name **myparser.y**. After running **PCYACC** on this file with the skeleton option -**p** on, we get a source code file named **myparser.cpp**. The file **myparser.cpp** contains the data tables and member functions of the syntactic parser class.

a. Description

Unlike other classes, the syntactic parser class is generated by inserting a skeleton file into **PCYACC** generated output file. The class is specific to a language and actions the user intends to specify for the language. For example, we are developing parsers for two different languages L1 and L2. Suppose corresponding grammar files for them are **l1.y** and **l2.y**. After applying **PCYACC** on these grammar files, we will get the declaration of two different classes for each language. This is necessary because the classes access different data tables and have different member functions even though the data tables and the functions share the same names. The class definition of parser class is in header file **"pcy_sk.hpp"**. The class name is **ABXYacc** in this file. And the skeleton is in the file **pcy_sk.cpp**. If you go along with the name **ABXYacc**, what you need next is to run **PCYACC** with option -**ppcy_sk.cpp**, and include header file **"pcy_sk.hpp"** in the source file where class **ABXYacc** is referred. Sometimes, you would like to choose some other names for the syntactic parser class. For example, your program may require multiple parsers. To create a parser class with name different from **ABXYacc**, what you need to do is to make a copy of the file **pcy_sk.cpp** with a different file name, make a copy of the header file **pcy_sk.hpp** and rename it as well. Modify the class name from **ABXYacc** to the one you will choose. Change the included file from **pcy_sk.hpp** to the one copied from **pcy_sk.hpp**. And you also need to change the class name in the header file from **ABXYacc** to the same as the one in the skeleton file. Then run **PCYACC** with option -**p** specified with the skeleton. At the place where you want to use this class, you just need to include the header file to make syntactic parser class declared.

b. Example

Suppose that we are writing a parser for a language. We have the grammar file for this language named **myparser.y**. We run **PCYACC** on this grammar file with option -**p.** As a result, we get a C++ output file **myparser.cpp**. Assume that we declared the syntactic parser class with name **ABXYacc** in the skeleton used. File **myparser.cpp** contains the definition of function **ABXYacc::yyParse()** and the declaration of the data tables and the code moved from section 1 and section 3 of file **myparser.y** if they do exist. What we need to do is to declare an instance of the syntactic parser class to create a parser. And the other thing we need to pay attention to is that we also need to have a lexical analyzer that provides tokens for this parser. Suppose we have already had such a lexical analyzer class **ABXLex**. (For details, please see lexical analyzer class section.) To create a lexical analyzer, we just need to declare an instance of class **ABXLex**. It is necessary to choose a lexer function for the parser. The syntactic parser class provides such a method to enable the parser to choose a lexer function. The call

```
myyacc->yysetLexer(mylex);
```

makes object **mylex** the chosen lexer of parser object **myyacc,** assuming **myyacc** and **mylex** are declared as **ABXYacc** and **ABXLex** object pointers respectively. A lexical analyzer is built around an input file, which means when you are creating a lexical analyzer, you need to specify an input stream as the arguments of the constructor.

Once the class definition is in place, we can write a simple program as below:

```
001:    #include <stdlib.h>
002:    #include <string.h>
003:    #include "pcy_sk.hpp" // header file for YACC class
004:    #include "pcl_sk.hpp" // header file for LEX class
005:
006:    int yylineno = 1;
007:    int yyerrcnt;              // count of errors
008:    char yyerrsrc[80];         // input file name
009:    FILE   *yyin;              // pointer to input file
010:
011:    void main(int argc, char** argv)
012:    {
013:        yyin = fopen(argv[1], "r");
014:        if (yyin == NULL)
015:            exit(1);
016:        strcpy(yyerrsrc, argv[1]);
017:
018:        ABXLex* mylex = new ABXLex(yyin, stdout);
019:        ABXYacc* myyacc = new ABXYacc(mylex, stdout, stderr,
                                                yyerrsrc);
020:        myyacc->yySetLexer(mylex);  // Set lexer
021:        myyacc->yyParse();          // Activate parsing function
022:
023:        yyerrcnt = myyacc->get_yyErrorCount();
024:        fclose(yyin);
025:        if (yyerrcnt != 0)
026:            fprintf(stderr, "There are errors\n");
027:        else
028:            fprintf(stdout, "No error.\n");
029:        exit(1);
030:    }
```

In this example, only a single lexical analyzer is used. The scheme makes a syntactic parser easier to take tokens from a lexical analyzer specified by the **yySetLexer** function. All the data tables and global information generated by a parser are hidden inside the parser object. In this way, each instance of **ABXYacc** object will have its own data tables, macro definitions, etc.

Since it is very important for **PCYACC** to be able to generate a parser that can take tokens from multiple lexers, the following example demonstrates how to write a parser that does this. Multiple lexical analyzer class definitions have to be generated by **PCLEX** beforehand. The key function to call is **yySetLexer** member function of a parser class. This takes a pointer to a lexer object as its argument. To use a specific lexical analyzer to supply token to the parser, make a function call to **yySetLexer** with a pointer to a lexer object as argument. Note that lexical analyzer objects are dynamically created by supplying an input file name as an argument. Each instance of the lexical analyzer class is different because the data tables for a lexer object are created according to the input description file.

Assume that we have all the required header files generated by **PCYACC** and **PCLEX** that define the lexical analyzer and parser classes. The main routine shown below illustrates how to switch between two different lexers in a parser:

```
001:    #include <stdlib.h>

002:    #include “pcy_sk.hpp”
003:    #include “pcl_sk.hpp”
004:
005:    void main()
006:    {
007:        FILE *input1, *input2;
008:
009:        input1=fopen(“input1”, “r”);
010:        input2=fopen(“input2”, “r”);
011:
012:        ABXLex* mylexer1 = new ABXLex(input1, stdout);
013:        ABXLex* mylexer2 = new ABXLex(input2, stdout);
014:        ABXYacc* myparser = new ABXYacc(mylexer1, stdout,
               stderr, NULL);
015:
016:        myparser->yyParse(); // Activate parsing function
017:        myparser->yySetLexer(mylexer2); // Set another lexer
018:        myparser->yyParse(); // Activate parsing function
019:    }
```

c. Public Constructor and Destructor

Assume the class name is **ABXYacc**, the constructor will be

```
ABXYacc();
```

This constructor establishes the stacks with default stack depth defined by **ABXYacc** class constant **YYMAXDEPTH** and initializes the pointers of the

stacks. An **ABXLex** class instance will be passed to an **ABXYacc** constructor as its parameter, which will provide tokens from input stream for an **ABXYacc** class instance

The public destructor for this class will be:

```
~ABXYacc();
```

It will clear the stacks and free memory used by an **ABXYacc** class instance.

d. Public Member Functions

```
int get_yyErrorCount(void);
```

This function will return the count of errors for a parser that is generated by **yyParse ()**.

```
void yySetLexer(ABXLex *)
```

This function's purpose is to allow a parser to choose a lexer function that provides tokens. The lexer function is specified as a pointer to a lexer object. The function will be called by the parser function.

```
int yyParse(void)
```

It is a **LALR** parser for the language defined by the grammar rules in the grammar description file. It is called somewhere by other functions, like **main()** function.

```
void yySetTokStack(int)
```

This function will modify the token stack size based on the input size provided by the user. It is a very important feature so that the user can modify the **PCYACC** buffer size to meet the project requirement.

VI. SYMBOL TABLE

1. Introduction

In the practice of writing a compiler or an interpreter, it is almost inevitable to manage symbol tables. There are several reasons to build a symbol table. First, we can build a symbol table as a lexer is running so that symbols can be entered in the lexical analysis phase and symbol properties can be entered in later phases. Second, without consulting the lexer, symbol tables can be checked to find out information about a symbol. Third, we can access a permanent copy of information referring to an appropriate symbol table entry even after lexer is ready to process the next token since each subsequent token overwrites **yytext**. Symbol table keeps track of symbols encountered by a compiler and services all phases of compiler. A general data structure is required to build a symbol table that can be shared by both lexer and parser.

A symbol table is a collection of symbols. A symbol itself is represented by a string of characters. In the construction of a compiler, there are some essential elements associated with a symbol.

Typically, they are:

1). Symbol name: Each symbol is a character string itself, it has a specific identification name to distinguish itself from others.

2). The attribute of a symbol: Basically, a symbol could be a variable name, a function name or a type name, the attribute identifies what kind of symbol it is.

3). The type of a symbol: This defines the specific domain associated with this symbol.

4). The value of a symbol: Each symbol could have a value associated with it. For example, a symbol could have an initial value before being modified in a program.

5). The scope of a symbol: A symbol may have a lifetime in the program. This scope identifies the symbol's living range in the program.

Since symbol table is a very important facility in compiler construction, it has been widely used to keep track of information in compiler or interpreter design field, the following sections will cover the details about building symbol table class.

2. Synopsis for ABXSymbolTable Class

```
#include "abxsym.hpp"
ABXSymbolTable symtbl;
```

The file "**abxsym.hpp**" contains all the macro definitions and the definition of **ABXSymbolTable** class. All the data declaration and class declaration will be inside file **abxsym.cpp**. The implementation of member functions for class **ABXSymbolTable** lives in the file **abxsym.cpp** as well.

a. Description

Class **ABXSymbolTable** provides the basic facilities of managing symbols in compiler construction. Due to the strong dependence of symbol information on the particular language, users should use this class as a starting point, define information specific to the language of concern and customize the class as necessary.

A symbol is a character string itself. This symbol table class is usually used for compiler construction, there are five parts associated with a symbol:

1). Name: It identifies a symbol and is represented by a character string.

2). Attribute: Each symbol must fall into a category. For example, a compiler may divide symbols into 3 categories, constants, variables and functions. An integer is used to represent the attribute of a symbol. In this example, constant=1(#define ATTR_CONST 1), variable=2(#define ATTR_VAR 2), function=3(#define ATTR_FUNCT 3). Of course, in writing a real compiler, the attribute is far more complicated.

3). Type: Every variable has an associated type that constrains its value to a specific domain. By defining its data type, the evaluation of a variable will become meaningful. The type of a variable normally will be defined by the variable declaration and the type of an expression is decided by the definition of the expression operators. The base types are defined in "**abxsym.hpp**" as following:

```
001:    #define TYPE_VOID               1
002:    #define TYPE_CHAR               2
003:    #define TYPE_SHORT              3
004:    #define TYPE_INT                4
005:    #define TYPE_LONG               5
006:    #define TYPE_LONG_LONG          6
007:    #define TYPE_EXTRA_INT          7
008:    #define TYPE_UCHAR              8     // unsigned char
009:    #define TYPE_USHORT             9     // unsigned short
```

```
010:    #define TYPE_UINT               10     // unsigned int
011:    #define TYPE_ULONG              11     // unsigned long
012:    #define TYPE_EXTRA_UINT         12     // Nonstandard
013:    #define TYPE_FLOAT              13
014:    #define TYPE_SHORT_DOUBLE       14
015:    #define TYPE_DOUBLE             15
016:    #define TYPE_LONG_DOUBLE        16
017:    #define TYPE_EXTRA_FLOAT        17     // Nonstandard float
018:    #define TYPE_ENUM               18
019:    #define TYPE_UNION              19
020:    #define TYPE_STRUCT             20
021:    #define TYPE_CLASS              21     // C++ only
022:    #define TYPE_DEFINED            22     // Typedef name
023:    #define TYPE_EXTRA_PTR          23     // Nonstandard
024:    #define TYPE_CONSTRUCTOR_TYPE   24     // C++ only
025:    #define TYPE_DESTRUCTOR         25     // C++ only
```

4). Value: Besides the attribute, a symbol may also have a value. For example, a constant normally has a value to represent a specific quantity number. It is useful to maintain this value during compiler construction.

5). Scope: In most programming languages, a symbol has its own lifetime. It is decided by its scope. Basically, if a symbol is global, it is in the scope 0. Scope of inner layers has bigger scope number than the outer layers.

According to the preceding description, we come up with following symbol table entry structure:

NAME	ATTRIBUTE	TYPE	VALUE	SCOPE

Figure 6-1. Structure for symbol table

Possible values that can be taken by each field should be defined by the user according to the specific needs of language.

A symbol table consists of a list of the symbol table entries. The symbol name and its scope uniquely identify a symbol table entry. The symbol table organizational structure is illustrated below.

scope 0	**sym 00**	**sym 01**	**... ...**	**sym 0(m-1)**
scope 1	**sym 10**	**sym 11**	**... ...**	**sym 1(m-1)**
scope n-1	**sym (n-1)0**	**sym (n-1)1**	**... ...**	**sym (n-1)(m-1)**

Figure 6-2. Symbol table format

For each scope, a sub-table can be maintained with the same table entry structure. This will make the scenario clear and make the table easier to access. A variable can be set as a pointer to the current scope. To make symbol table class independent of its client, the representation of symbol table should be completely hidden from its client.

b. Symbol Table Entry Definition

Based on the discussion about symbol table structure above, a simple symbol table entry can be defined as follows. Users should define the possible values for attribute and type field according to the language of concern. Also the **ABXVALUE** field should be augmented to support the possible types in the language. The definition is shown as below.

```
001: union ABXVALUE {
002:      int       i;
003:      double    db;
004:      char ch[80];
005:      ...
006: };

007: struct ABXsymtabentry {
008:      char      name[80];
009:      int       attribute; // e.g., constant=1, variable=2,
010:                           // function=3
011:      int       type;
012:      ABXVALUE  value;
013:      int       scope;
014:      struct    ABXsymtabentry *next;
015: };
```

c. Private Class Member

```
001: private:
```

```
002:      int scope_indicator; // current scope indicator
003:      struct ABXsymtab
004:      {
005:           int scope;
006:           struct ABXsymtabentry *symbol_list; // subtable
007:           struct ABXsymtab *next;
008:      };
009:
010:      struct ABXsymtab *symtab_head; // symbol table head
011:      struct ABXsymtab *symtab_ptr; // current symbol table
```

d. Public Constructor and Destructor

```
ABXSymbolTable();
```

Set head and current symbol table pointer to NULL.

```
~ABXSymbolTable();
```

Releases all dynamically allocated memory for symbol tables. Destroys the created symbol tables.

e. Public Member Functions

```
int addSymbol(char* name, int attribute, int type,
ABXVALUE value, int scp);
```

Add a symbol into symbol table according to input parameters. If the new symbol is added successfully, this function returns 1, if the specified symbol already exists in symbol table, returns 2, otherwise returns 0. The current scope indicator and symbol table pointer have been modified accordingly.

```
int cleanScope(int scp);
```

Remove all symbols in symbol table with scope **scp**, set current scope indicator and current symbol table pointer accordingly. If the specified scope does not exist, function returns 0, otherwise returns 1.

```
int createScope(int scp);
```

Create a new scope, set current scope indicator and symbol table pointer. If successful, function returns 1, if the scope requested already exists returns 2, otherwise returns 0.

```
int deleteSymbol(char *name, int scp);
```

Delete a symbol table entry according to its name and scope. Return 1 if the symbol is deleted successfully. If the symbol does not exist, return 0. The current scope indicator and symbol table pointer have been modified accordingly.

```
int getScope(void);
```

Get the value of current scope indicator.

```
int getSymbolAttribute(char *name, int scp);
```

Return a symbol's attribute if the symbol was found, otherwise return -1. The current scope indicator and symbol table pointer have been modified accordingly.

```
int getSymbolType(char * name, int scp, int type);
```

Return a symbol’s type if the symbol was found, otherwise return -1. The current scope indicator and symbol table pointer have been modified accordingly.

```
union ABXVALUE getSymbolValue(char *name, int scp,
ABXVALUE value);
```

Return a symbol's value if the symbol was found, otherwise returns -1. The current scope indicator and symbol table pointer have been modified accordingly.

```
void initSymbolTable(void);
```

Construct a symbol table with only scope 0 (global layer).

```
int insertSymbol(ABXsymtabentry *symbol)
```

Insert symbol based on the current scope indicator and symbol table pointer. If successful, return 1. Otherwise, return 0 (symbol already exists).

```
struct ABXsymtabentry *lookupSymbol (char *s, int
scp);
```

Look up a symbol that is already inserted in the symbol table. If the symbol is found, return the symbol's symbol table entry pointer, otherwise return NULL. The current scope indicator and symbol table pointer have been modified accordingly.

```
int maxScope(void);
```

Get the biggest scope number of the symbol table.

```
int setScope(int scp);
```

Set the current scope indicator to the **scp** and current symbol table pointer has been changed accordingly. If the specified scope does not exist, the function returns -1. Otherwise, it returns the scope number just set.

```
int setSymbolAttribute(char *name, int scp, int attribute);
```

Set symbol attribute value (defined by user) according to its name and scope. This function will return 1 if successful, otherwise return 0. The current scope indicator and symbol table pointer have been modified accordingly.

```
int setSymbolType(char * name, int scp, int type);
```

Set symbol data type according to its name and scope. Customers can define their own types by using macro definitions according to their specific compiler requirements. This function will return 1 if successful, otherwise, return 0. The current scope indicator and symbol table pointer have been modified accordingly.

```
int setSymbolValue(char *name, int scp, ABXVALUE value);
```

Set symbol value according to its name and scope. Return 1 if proper entry has been found, otherwise return -1. The current scope indicator and symbol table pointer have been modified accordingly.

VII. ERROR HANDLER

1. Introduction

There are few errors that a lexical analyzer can detect because it lacks the context in which the input stream appears. The parser, on the other hand, has a much richer view of what is "correct" and can detect errors relating to the order of tokens. It also will be harder for a parser to recover from errors because even a single erroneous character can badly botch the overall program structure. So when we are talking about error processing, we are pointing to the error processing for syntactic parsers.

The errors can be divided into several categories according to their severity. They deserve different treatments due to their severity. Some errors are recoverable and some others just are not.

Basically, errors can be divided into 3 categories based on their severity.

1). Minor: A syntactic violation for which the parser believes it has a correction that is likely to be what programmer intended. It fixes the program and continues processing with a very high probability that it 'guessed' right.

2). Major: A violation for which the parser has no reliable correction. It will attempt to continue the parsing, but will probably have to skip over part of the input or take some other exceptional action to do so. There is a significant risk it "guessed" wrong.

3). Panic: Things are so fouled up that the parser can not continue its job. It terminates execution after issuing some error messages.

Therefore, we provide an error handling class in the library for these purposes above. It includes the facilities for error reporting and error recovering. The error reporting will give the information about the type of the error detected, the snapshot when the error occurs.

a. Error Reporting

We need a function to issue the warning message. The warning message should be informative enough to let caller know where the error occurs and some internal information such as that of stacks. Considering that this class is for the purpose of error handling of syntax parser, the target for the parser to process is the tokens that the lexical scanner separates from the input stream. So the error reporting should be able to point out at which line of the input stream the error occurred, and ideally, also at which character of the

line. The function also should be able to give the information about what kind of error it is.

b. Error Recovery

Besides the error reporting facilities, we also need to have some facilities to recover from errors that are not fatal to make the program halt. It is possible for the parser to recover from an error and continue processing for additional errors. By using **error** token, the parser is capable of discovering the synchronization point in the Grammar Description File. From that point, the parser can discard some undesirable tokens until it reaches one that follows the error token in the GDF to continue its processing procedure.

To get the information like line number, token string, and stack states, it is necessary for the error processing class to be able to access some data of the lexical analyzer class and the syntactic parser class. It is unwise to make the data member accessible from public. So, error class requires one **ABXLex** and one **ABXYacc** object representing the current lexer and parser as its private data members. By setting these two private data members in error class constructor, an error object can refer its own member function to get information from the current lexer and parser.

2. Functions for Error Reporting

Function **yyError**

This function simply invokes the function **yyErrPrefix** to implement the error message display. Some proper message display formats should be considered as input parameter to make the error message straight to the customers.

Function **yyErrPrefix**

This function is used by the parser (a **ABXYacc** class instance) which is called by function **yyError** to display the error message based on the information provided by the lexical scanner (a **ABXLex** class instance). In order to make the error message clear to customers, some information such as: total number of detected errors, the current input stream, the current line number and the token context, etc, will be displayed. The information related to current token that the lexical scanner just returns to the parser should be provided as input parameters.

Function **yyGetCharNumber**

Get the beginning position of current token at the current line. Combine this function and **yyGetLineNumber**, we can get the exact coordinate of the current token.

Function **yyGetErrFileName**

This function will take input parameter to set error file name.

Function **yyGetErrorCount**

This function returns current error number that the lexical scanner is currently processing.

Function **yyGetExpectedTokens**

Get the expected tokens. It is common that multiple tokens can fit in the position. The function will take an integer array as parameter. After this function is called, the array will be filled with the internal values of those expected tokens. The returned value of this function will be the exact number of the possible tokens. We can call function **yyGetToken** to convert these tokens into character strings one by one for display.

Function **yyGetLineNumber**

Gets the current line number of the source file that is being scanned by the lexical analyzer.

Function **yyGetToken**

yyGetToken takes the token value defined by the user as input and returns a corresponding token context by searching the user defined token list. More detailed information about the user defined token list structure is described in the following **Synopsis** section.

Function **yySetErrText**

This function will use input parameter to set **yyErrText**(error message).

3. Functions for Error Recovery

Function **yyInsertToken**

Push an additional token in front of the erroneous one. The current symbol remains unchanged.

Function **yyMatchToken**

Pretend the current token matches the current symbol if it is a terminal.

Function **yyReplaceToken**

Replace the current token with one that legitimately might have appeared there. The current symbol in the sentential form is unchanged.

Function **yySkipSymbol**

Skip the current symbol in the sentential form if it is a terminal and pretend that it was matched. The current token is left unchanged.

Function **yySkipToken**

Ignore the current token and pretend it was never there. The current symbol in the sentential form is unchanged.

The above functions are provided for error processing. The C++ skeleton provides the basic facilities for error handling. It is up to the user to apply the functions to satisfy the specific needs of the language.

If a parser does not provide any error recovery measure, it will stop immediately. To make the parser robust, we need to insert some extra rules with errors in it to make the parser recoverable when the errors occur at the places where the extra rules are introduced. The strategies about where the best places are to put these extra rules have been explained quite clearly in some articles and the on-line manual on error processing. This is the part decided by the user about how robust he/she wants to make his/her parser. The error processing class will provide the standard procedure for error recovery when those extra rules are inserted in the grammar description file.

The error recovery mechanisms are scattered in the body of function **yyParse()**. Once the parser detects an error, it takes the code branch with the error processing. In C++, it is possible to take advantage of the exception handling facilities. The error recovering work could be done in the exception handler.

4. Synopsis for ABXError Class

```
#include "abxerr.hpp"
ABXError* errObject;
```

The **ABXError** class definition and macro definitions are included inside file "**abxerr.hpp**". We provide the **ABXError** member function definitions in a separate file named "abxerr.cpp".

a. Description

The library provides a class for error processing. This class contains functions for error reporting, error handling and error recovering. Actually this class is the expansion of the syntactic parser class. The skeleton contains the standard error recovery mechanism in function **yyParse()**.

Whenever a parser detects a syntax error, it calls **yyError** to report the error to the user. However, in the error display phase, a token list that stores the token values and the corresponding token context should be provided by the users. The following token list structure is recommended:

```
001: static struct
002: {
003:      char keyname[ABX_TEXT];
004:      int       tokenvalue;
005: }keywords[];
```

According to different requirements, the user can define the array size by using macro definition. Any token that will appear in compiler will be put inside this list by the users.

b. ABXError Class Definition

Just like the description in the introduction section, **ABXError** object has to hold both **ABXLex** and **ABXYacc** objects as its private data members so that an **ABXError** instance is granted rights to access the private part of **ABXLex** and **ABXYacc**'s instances by requiring membership. Since all the error handling functions will be called by the parser whenever it detects a syntax error in the user program, some data members related to the error display from the lexical scanner and syntax analyzer should be defined as the private data of class **ABXError** .

```
001: class ABXError
002: {
003:      Private:
004:           ABXLex*        lexObject;
005:           ABXYacc*       yaccObject;
006:           int            yyErrCnt;
007:           char*          yyErrSrc;
008:           FILE*          yyErrFile;
009:           int            yyErrLineNo;
010:           char*          yyErrText;
011:           ... ...
012: };
```

c. Public Constructor and Destructor

Assume the error processing class is named **ABXError**, its class constructor will be

```
ABXError(ABXLex*, ABXYacc*, FILE*, char*);
```

This constructor will get **ABXLex** and **ABXYacc** instances with initialization of its private data member. Since **ABXError** class has both **ABXLex** and **ABXYacc** instances, the error processing can detect errors and display the error message properly as long as the lexer and parser are working.

```
~ABXError();
```

Release all allocated memory.

d. Public Member Functions

```
void yyDisplay(int);
```

Display token context based on input token value. Normally this function is provided by the user.

```
void yyError(char *, char *, char *);
```

This function simply calls **yyErrPrefix** to finish error reporting by reformatting the error message display. The two input strings can represent two different error messages to be displayed.

```
void yyErrPrefix(char *, char *);
```

This function takes a syntax error message as input and displays it according to the current token information. The current token information including line number, char position number, ..., etc, is provided by the lexer.

```
int yyGetCharNumber(void);
```

This function returns the beginning position of current token on the current line the lexical scanner is processing.

```
int yyGetErrorCount(void);
```

This function returns current error number that the lexical scanner is currently processing.

```
int yyGetExpectedTokens(int *);
```

This function takes an integer array of internal values for expected tokens as input and returns the exact number of those expected tokens.

```
void yyGetFileName(char *s);
```

This function will take input parameter to set error file name.

```
int yyGetLineNumber(void);
```

This function returns the current line number that the lexical scanner is currently processing.

```
char* yyGetToken(int);
```

This function takes a token value (integer) as input and returns an actual token context expressed in a character string. A user-defined token list should be provided in order for this function to work.

```
void yyInsertToken(char *);
```

This function takes a token to be inserted as input parameter. Its purpose is to insert a token in front of the one that will be processed by the lexer.

```
int yyMatchToken(char *);
```

This function takes a matching token as input parameter. It will return 1 if the current token matches the actual input token value, otherwise returns 0 instead.

```
void yyReplaceToken(char *);
```

This function takes a replacing token as input and replaces the token that will be processed by the lexer.

```
void yySetErrText(char *s);
```

This function will use input parameter to set **yyErrText**(error message).

```
void yySkipSymbol(void);
```

This function simply skips the current symbol in the sentential form if it is a terminal and keeps the current token unchanged.

```
void yySkipToken(void);
```

This function makes file indicator forward one unit to skip the current token as if the current token being processed by the lexer does not exist in the input stream.

VIII. PARSE TREE NODE

In compiler design, a data structure called intermediate representation or **IR** is used for the compiler to communicate with the code generator. This **IR** is a tree structure, so called parse tree. It has the important purpose of making explicit hierarchical syntactic structure of sentences that is implied by the grammar.

As a tree structure, parse tree has one or many nodes, which represent specific syntactic meaning in the grammar. Each node has its own data called semantic attributes. These are determined by one or more of its children. Some actions can be imposed on this information to allow it to be synthesized through tree or stored as field or members of the appropriate syntax tree node. There are some necessities to implement this action.

- It is necessary for semantic information to move down from a parent node to a child node.

- Some semantic information inherited from the parent node is passed down to a child node during traversal of the parse tree.

In our **PCYACC OO TOOLKIT Library**, we build a parse tree node class named **ABXParseTreeNode**. Each parse tree node represents an object that not only has data member where semantic information lives, but also has its own member functions that are capable of imposing language specification, which are effective both at compilation and execution.

As we choose the object oriented approach to design our **PCYACC OO TOOLKIT Library**, it makes it possible for the customer to take several advantages during compiler construction. The semantic information can be obtained by accessing **ABXParseTreeNode** object's data member. The additional dynamic semantic information that the customer would like to impose on the language specification can be inserted into the parse tree by using **ABXParseTreeNode** class' member functions at run time.

1. Analyze Parse Tree Node Class ABXParseTreeNode

Parse tree is widely used for customers to evaluate the semantic rules. The normal procedure would be: 1). Parse the input token stream, 2). Build the parse tree and 3). Traverse the tree as needed.

Since each symbol has an associated set of attributes, it is necessary to partition them into two subsets: synthesized and inherited attributes of symbol. Each node in a parse tree contains a record with fields for holding information and an attribute corresponding to the name of a field.

An attribute can be a string, a number, a type, a memory location, etc. Its value at a parse tree node is determined by production of semantic rule applied at that node. For a synthesized attribute at a node, its value is computed from the values of attributes at the children of that node in the parse tree. For an inherited attribute of a node, its value is computed from all the attribute values at the siblings and parent of that node.

The parse tree itself is a useful intermediate language representation for a source program, especially in optimizing compilers where the intermediate code needs to be extensively restructured. However, a parse tree often contains redundant information that can be eliminated. In this way, a more efficient representation of the source program is produced.

In the design phase we are responsible for generating specification of class communication and membership. A general procedure is to discover the parse tree objects and then arrange them into clusters that ultimately become class.

The parse tree nodes have various attributes. Some have attributes just pointing to other parse tree nodes. Several classifications of parse tree node are necessary for customers to construct their compiler.

```
001: Program node: this actually could serve as root of
     parse tree.
002: Declaration nodes:
003:      * Constant definition nodes.
004:      * Type definition nodes.
005:      * Variable declaration nodes.
006:      * Procedure-function declaration nodes.
007: Statement nodes.
008: Expression nodes.
009: Leaf nodes.
```

Each of these nodes will be a subclass of the general class of parse tree nodes. By analyzing these nodes attributes we need to place all the data members and member functions that are common for all parse tree nodes into the general class. Some specific data member and member functions for the appropriate parse tree node will be put in the subclass as its members.

By using **PCYTOOL**, the parser will recognize the input program first, followed by building a parse tree from the bottom up. The first recognized grammar productions are those specifying or containing reference to the lexeme token. These tokens are mostly responsible for a parse tree node, which normally is a leaf object of the tree. The subsequent grammar production to be completely recognized by the parser is the one that calls the lexeme token production. The proper parse tree node will be built for this

grammar production based on all the lowest level token productions that have been recognized. By following these procedures until the highest grammar productions have been processed, we can simulate the parse recognition of process to build up the parse tree. The scenario to mimic the parser recognizing the grammar is described as follows:

1). Start with lowest level grammar production.

2). Insert the corresponding leaf to the list of objects.

3). Move down and up the grammar.

* Find a production up one level that calls one of the present production.

* Trace down the grammar. Find all leaves referenced by this production and add the objects for parse tree nodes and leaves using the order in which they would be recognized by the parser.

The discussion above is an analysis for building parse tree node class. In the following sections, we will focus on describing data members and member functions provided in our **ABXParseTreeNode** class.

2. Structure for ABXParseTreeNode Class

A general parse tree node class that encapsulates the common property for all the other tree nodes should be built as the base class. Additional information corresponding to a particular class will be added in the derived node class. So some member functions in this base class should be defined as virtual functions to allow the subclass to modify them according to their needs.

```
001: class ABXParseTreeNode
002: {
003:      public:
004:           ABXParseTreeNode();
005:           virtual ABXParseTreeNode*     build_tree();
006:           virtual ABXParseTreeNode*     append_node();
007:           virtual ABXParseTreeNode*     delete_node();
008:           virtual ABXParseTreeNode*     lookup_tree();
009:           virtual ABXParseTreeNode*     optimize_tree();
010:           virtual void   print_tree();
011:           virtual int    get_token();
012:      private:
013:           int  tok;
```

```
014: };
```

```
ABXParseTreeNode();
```

This constructor initializes private data member for parse tree base class.

```
~ABXParseTreeNode();
```

Destroy all dynamically allocated memory for parse tree base class.

3. Structure for ABXLeaf Class

Leaf object should take lexeme value and store it as class member for future access. Also the corresponding type for this lexeme will be considered as data member for **ABXLeaf** class. In order to encapsulate the lexemes and corresponding types, the **ABXLeaf** class will be defined like:

```
001:
002: {
003:      public:
004:           ABXLeaf();
005:      private:
006:           char *tok_name;
007:           int  type;
008: };
```

```
ABXLeaf();
```

The task for this constructor is to initialize the data members for **ABXLeaf** class. All the preparation work for **ABXLeaf** object is done here.

```
~ABXLeaf();
```

Destroy all dynamically allocated memory for **ABXLeaf** class.

In order to manipulate the parse tree with the statement list, function parameter list, etc., an **ABXLeafList** class is necessary to take the whole list as single input for this purpose. The **ABXLeafList** definition is shown as below:

```
001: class ABXLeafList : public ABXParseTreeNode
002: {
003:      public:
004:           ABXLeafList();
005:           ABXLeafList(ABXParseTreeNode *);
006:           ABXParseTreeNode* append_node(ABXParseTreeNode *);
007:           ABXParseTreeNode* delete_node(ABXParseTreeNode *);
008:           ABXLeaf*  create_leaf_list();
```

```
009:      private:
010:           ABXLeaf*  leaf_list_head;     // head of leaf list
011:           ABXLeaf*  leaf_list_tail;     // tail of leaf list
012: };
```

```
ABXLeafList(); or ABXLeafList(ABXParseTreeNode *);
```

ABXLeafList class will provide two constructors. The first one will simply initialize the data members. The second one will take a pointer to **ABXParseTreeNode** as input, set corresponding information to the data members in **ABXLeafList** class.

```
~ABXLeafList();
```

Destroy all dynamically allocated memory for **ABXLeafList** class.

```
ABXParseTreeNode* append_node(ABXParseTreeNode *);
```

This function will take a pointer to parse tree as input. After appending the current leaf nodes to the parse tree, the corresponding pointer to parse tree node will be returned.

```
ABXParseTreeNode* delete_node(ABXParseTreeNode *);
```

This function will take a pointer to parse tree as input. After deleting the current leaf nodes to the parse tree, the corresponding pointer to parse tree node will be returned.

```
ABXLeaf*   create_leaf_list();
```

This function creates a leaf list according to the current leaf node's information hidden in the **ABXLeafList** object. A corresponding **ABXLeaf** pointer to the created leaf list will be returned.

4. Expression Classes ABXExprNode

Since there are several different expression objects for any given parse tree, we can build **ABXExprNode** class to represent their instances. For lengthy strings, we can always expect the inheritance based on some common data member and functions. As every expression has a basic type and a function to be used in execution, these members must be considered as expression base class data member. Any additional data members will be put into derived expression class for specific expression objects. The base expression class **ABXExprNode** is defined as following:

```
001: class ABXExprNode
002: {
```

```
003:      public:
004:           ABXExprNode();
005:           ABXExprNode(int type);
006:           virtual ABXSymbolTable*  get_attr();
007:      private:
008:           int type;
009:           ABXVALUE value;
010: };
```

```
ABXExprNode(); or ABXExprNode(int type);
```

The initialization of data member for class **ABXExprNode** will be done in these two constructors.

```
~ABXExprNode();
```

Destroy all dynamically allocated memory for **ABXExprNode** class.

```
virtual ABXSymbolTable* get_attr();
```

This function will search the symbol table to find a corresponding table entry for this current particular expression.

During compiler design, we need to handle the information hidden in multiple expressions, a class representing this kind of linked list expressions that derived from the base class **ABXParseTreeNode** should be generated. We name this class as **ABXExprNodeList**, the class definition is like the following:

```
001: class ABXExprNodeList : public ABXParseTreeNode
002: {
003:      public:
004:           ABXExprNodeList();
005:           ABXExprNodeList(ABXParseTreeNode*  expr);
006:           ABXParseTreeNode* append_node(ABXParseTreeNode*);
007:      private:
008:           ABXExprNode*   expr_head;
009:           ABXExprNode*   expr_tail;
010: };
```

```
ABXExprNodeList(); or
ABXExprNodeList(ABXParseTreeNode*  expr);
```

The initialization of data member for class **ABXExprNodeList** is done here.

```
~ABXExprNodeList();
```

Destroy all dynamically allocated memory for **ABXExprNodeList** class.

```
ABXParseTreeNode* append_node(ABXParseTreeNode*);
```

This function takes a pointer to a parse tree node as input, appends the current expression list to the parse tree node and returns a pointer to the decorated parse tree.

5. Structure for Parse Tree Class ABXParseTree

This class will be used to create instances of a parse tree by using instance of class **ABXParseTreeNode**. Some member functions are provided for the user to view the parse tree generated by **PCYACC**. The class definition for this class is presented below:

```
001: class ABXParseTree
002: {
003:      public:
004:           ABXParseTree(ABXParseTreeNode *root);
005:           ABXParseTreeNode* optimize(ABXParseTreeNode *);
006:           void print_tree(ABXParseTreeNode *);
007:           void show_tree(ABXParseTreeNode *);
008:           ABXParseTreeNode*   execute_tree(ABXParseTreeNode *);
009:           ABXParseTreeNode*   decorate_tree(ABXParseTreeNode
               *, ...);
010:      public:
011:           ABXParseTreeNode*   root;
012: };
```

```
ABXParseTree(ABXParseTreeNode *root)
```

This **ABXParseTree** class constructor will take the parse tree generated by **PCYACC** machine or parse tree root as input information, and then set the corresponding **ABXParseTree** data member to create a parse tree instance.

```
~ABXParseTree();
```

Destroy all dynamically allocated memory for **ABXParseTree** class.

```
ABXParseTreeNode* optimize(ABXParseTreeNode *);
```

This function will optimize the generated parse tree to make it more efficient.

```
void print_tree(ABXParseTreeNode *);
```

This function will create a hard copy for a specific parse tree.

```
void show_tree(ABXParseTreeNode *);
```

This function can display a parse tree based on the pointer to a user-specified parse tree.

```
ABXParseTreeNode* execute_tree(ABXParseTreeNode *);
```

This function can traverse the whole parse tree from the bottom and return a pointer to the top parse tree node based on a pointer to the user-specified parse tree node.

```
ABXParseTreeNode * decorate_tree(ABXParseTreeNode *, ...);
```

This function can add some information to a specific parse tree, or insert a parse tree node, etc.

IX. Java Parser and Lexer

1. Introduction

Although Java is a pretty new programming language, it brings the advantage of building applications that will run cross-platform and through network. It also provides object-oriented capability like C++. With the syntax similar to C/C++ and extensive extensions, the Java programming language has been widely received by the world community of software developers and internet providers as the new generation language after C++. Because of Java's popularity, supporting Java Parser has become one of our important tasks as a **PCYACC** tool provider.

Java is fully object-oriented, even more than C++. C++ parser is widely used in today's programming world and there is a stable set of classes used for it. Our design will focus on establishing a Java class equivalent to every C++ class used for C++ parser.

Symbol table management is a commonly used technique in compiler construction, and parse tree plays an important role of making explicit hierarchical syntactic structure of sentences that is implied by the grammar. In addition to Java Yacc, Lex and error classes, we also provide symbol table and parse tree classes.

The structure of Java class library is shown as following:

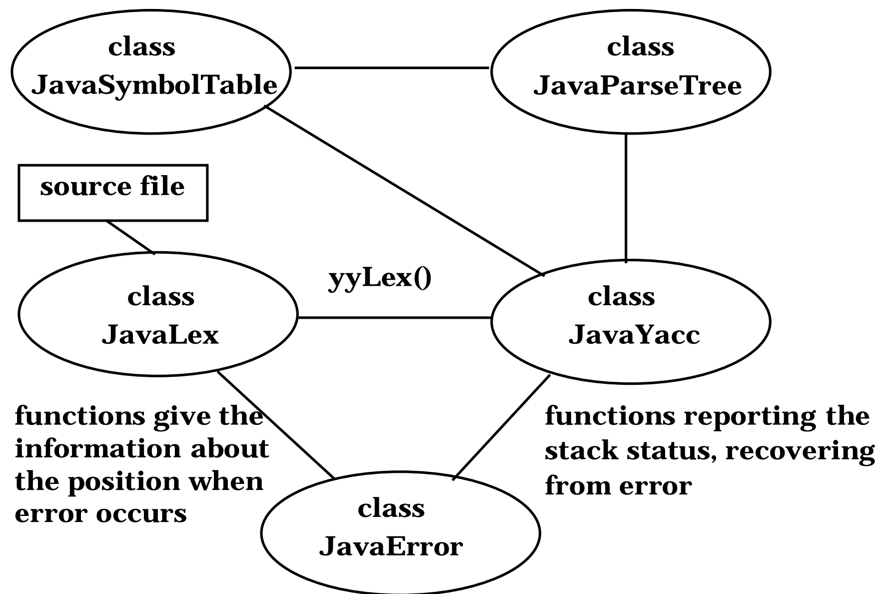

Figure 9-1. Structure of Java Class Library

Where functions for five basic classes are:

1). **JavaLex Lexical Analyzer Class**: this class serves as a code skeleton for PCLEX.

2). **JavaYacc Syntax Parser Class**: this class supports syntactic parser PCYACC.

3). **JavaSymbolTable Symbol Table Class**: this class is used for symbol table management.

4). **JavaError Error Handling Class**: this class is responsible for error reporting.

5). **JavaParseTree Parse Tree Class**: this class is available when the user wants to construct parse trees.

There are two phases to create a Java parser. The following diagram shows these procedures:

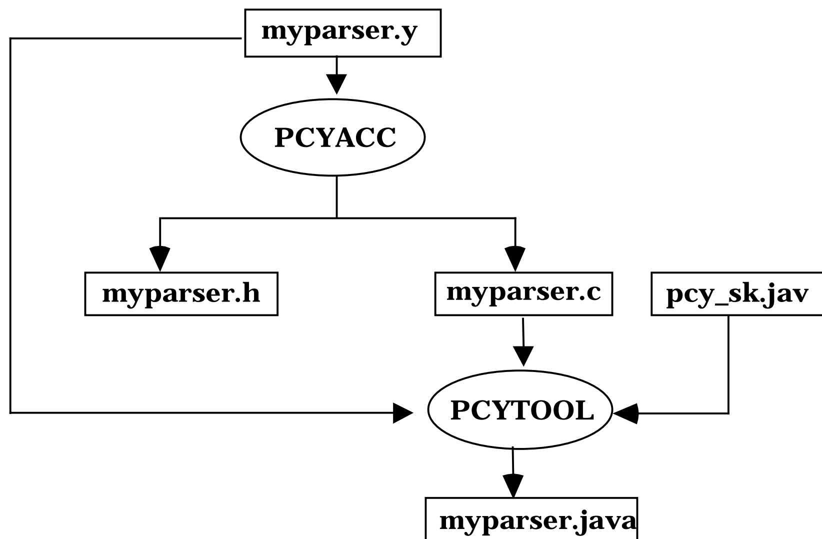

Figure 9-2. Diagram of Java parser generated by PCYTOOL

In the first phase, based on the availability of grammar description file, simply invoke the **PCYACC** tool on the command line to create a C parser. Once the C parser is generated by **PCYACC**, a utility program named **PCYTOOL** is needed for generating a Java parser in the second phase.

There are two phases to create a Java Lexer. The following diagram shows these procedures:

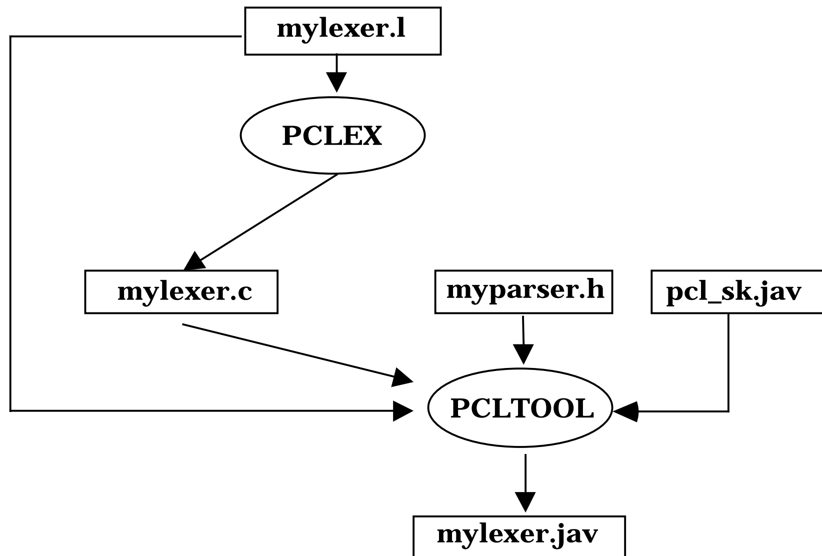

Figure 9-3. Diagram of Java lexer generated by PCLTOOL

In the first phase, based on the availability of scanner description file, simply invoke the **PCLEX** tool on the command line to create a C lexer. Once the C lexer is generated by **PCLEX**, a utility program named **PCLTOOL** is needed for generating a Java lexer in the second phase.

2. Java Class Library

We will introduce several Java classes that are equivalent to the C++ class library. These classes will be used for generating Java parser, lexer and for other auxiliary functionality. Detailed definition and member function description will be given for each class.

a. JavaLex Class

Class Data Members:

```
private static FileInputStream lex_fis;
```

lex_fis is defined as lexer input file stream. It will be initialized in **JavaLex** constructor.

```
001: private static final int YY_END_TOK = 0;
```

```
002: private static final int YY_NEW_FILE = -1;
003: private static final int YY_DO_DEFAULT = -2;
004: private static final int YY_NULL = 0;
005: private static final int BUFSIZ = 4096;
006: private static final int F_BUFSIZ = 4096;
007: private static final int YY_BUF_SIZE = 8192;
008: private static final int YY_BUF_MAX = 8191;
009: private static final int YY_MAX_LINE = 4096;
010: private static final int YY_BUF_LIM = 4095;
```

Java language does not support macro definition. However, in traditional C lexer generated by **PCLEX** tool, there are a lot of macros involved, which provides a lot of convenience to the users. The previously listed constants like **F_BUFSIZ, YY_BUF_SIZE, YY_BUF_MAX, YY_MAX_LINE** and **YY_BUF_LIM**, have been defined based on **BUFSIZ** with a default value of 4096. For details about how to assign the values of these constants, please refer to a C lexer source code generated by **PCLEX**.

```
001: private static int yy_start;
002: private static int yy_b_buf_p;
003: private static int yy_c_buf_p;
004: private static int yy_e_buf_p;
005: private static int yy_saw_eof = 1;
006: private static int yy_init = 1;
007: private static char yy_ch_buf[] = new char[YY_BUF_SIZE+1];
008: private static int yy_st_buf[] = new int[YY_BUF_SIZE];
009: private static char yy_hold_char;
010: private static char yytext[] = new char[BUFSIZ];
011: private static int yyleng;
012: private static int yy_lp;
013: private static int yy_curst
014: private static int reject_flag = 0;
015: private int yylineno = 1;
```

These variables are originally defined as globals in a C lexer. Since we have defined **JavaLex** class, all these variables used for the lexer will be put inside the **JavaLex** class scope and act as its private members.

Class member functions:

```
public JavaLex(FileInputStream finput) { }
```

This function is used to initialize the objects of a **JavaLex** class – giving the instance variables the initial state you want them to have.

```
int get_yylineno() { }
```

yylineno is a private data member of the **JavaLex** class. This function is used to fetch **yylineno** information, which normally is used for error reporting.

```
int input() { }
```

This function returns the next character from input stream.

```
void unput(char c) { }
```

This function puts a character back to the logical input stream. You can call several times in a row to put several characters back into the input stream.

```
private static void YY_DEFAULT_ACTION() { }
```

This function is defined as a macro in a C lexer. It will emit information of **yytext** buffer to the standard output.

```
private static int YY_INPUT (char buf[], int buf_offset, int max_size) { }
```

This function deals with the storage of input stream into a lexer. It gets input from the input stream and stores it into a buffer, which the user can specify as the function call parameter.

```
private static void YY_OUTPUT(char c) { }
```

This function outputs one character to the standard output.

```
private static void YY_FATAL_ERROR(String s) { }
```

This function prints out the string specified in as the function call parameter to the standard error.

```
private static boolean yywrap() { }
```

This function simply returns boolean value **true**.

```
private static void YY_SET_EOL(char array[], int pos) { }
```

This function sets a new line character on a special position of an array.

```
private static void yyless(int n) { }
```

This function tells the lex to “push back” part of the token that was just read. The argument to **yyless()** is the number of token characters to push back into the input stream.

```
private static void YY_INIT() { }
```

This function is used to initialize the scanner's state.

```
private static int YY_LENG() { }
```

This function returns the length for the text of the token stored in **yytext**.

```
private static void YY_DO_BEFORE_SCAN() { }
```

This function puts the character that the scanner holds at the end of **yytext**.

```
private static void YY_DO_BEFORE_ACTION() { }
```

This function does some preparation job before the scanner takes actions.

```
private static void REJECT(int yy_full_match) { }
```

This function puts back the text matched by the pattern and finds the next best match for it.

```
int yylex() { }
```

This function is used to start or resume scanning. It separates an input character stream into tokens following the patterns specified in lex (*.l) file.

```
char [] get_yytext() { }
```

This function returns the current token in the lex token buffer.

b. JavaYacc Class

Class Data Member:

```
private static String pcyyerrsrc;
private static FileInputStream fis;
```

pcyyerrsrc defines the input source file name that the parser is processing. If error occurs, the error reporting mechanism will be able to indicate which input source file processing has errors. **fis** is defined as an input stream to a parser, which will be used to pass a lexer object's internal information to the parser.

```
private static imp_union yylval = new imp_union();
private static imp_union yyval = new imp_union();
```

These two variables are used to pass the value of the token to a syntax analyzer. So they are defined as ***union*** type in a C equivalent lexers, which is represented by **imp_union** class in Java.

```
private static int pcyyerrfl=0;
private static int pcyyerrcnt=0;
private static int pcyytoken=-1;
private static int pcyylineno;
int list = 0;
```

These variables are defined as globals in a C parser. In Java, they are defined as **JavaYacc** class data members.

```
JavaLex lexobject;
```

A lex object is defined as a **JavaYacc** member so that the **JavaYacc** object can access the **JavaLex** information.

Class Member Functions:

```
public JavaYacc(JavaLex lex_object, String
  input_file_name) { }
public JavaYacc(String input_file_name,
  FileInputStream f) { }
public JavaYacc() { }
```

These three **JavaYacc** constructors deal with three different parser situations, integrated case with both lex and yacc objects, stand-alone yacc with hand-coded lexer whose input is either from standard input or file input stream.

```
private static void set_input_file_name(String fname) { }
```

This function is used to set **pcyyerrsrc** so that error report could show which input source file the parser is processing.

```
int yyparse() { }
```

This function is the entry point to a yacc-generated parser. When your program calls the member function **yyparse()**, the parser attempts to parse an input stream. The parser returns a value of zero if the parse succeeds and non-zero if not.

```
private static void yyerrok() { }
```

This function is implemented to replace a macro in a C parser. It tells the parser to return to the normal state, which can avoid the problem of multiple error messages resulting from a single mistake as the parser gets resynchronized.

```
int get_yyerrcnt() { }
```

This function simply returns **pcyyerrcnt** value in case the other class would like to access this internal variable of **JavaYacc** class.

c. JavaError Class

This class provides functions for error processing. It contains functions for error reporting, error handling and error recovering. Actually this class is the expansion of the syntactic parser class.

Class Data Member:

```
001: JavaLex lexObject;
002: JavaYacc yaccObject;
```

JavaError object has both **JavaLex** and **JavaYacc** objects as its private data members so that a **JavaError** instance is granted rights to access the private part of **JavaLex** and **JavaYacc**'s instances by requiring membership.

```
003: private static int yyErrCnt;
004: private static String yyErrSrc;
005: private static FileInputStream yyErrFile;
006: private static int yyErrLineNo;
007: private static String yyErrText;
```

These variables are defined as the private static members of **JavaError** class so that the other class' instance can access them by requiring membership.

Class Member Function:

```
JavaError(JavaLex LexObject, JavaYacc YaccObject,
FileInputStream yyerrfile, String yyerrsrc) { }
```

This constructor will construct **JavaLex** and **JavaYacc** instances and initialize their private data members. Since **JavaError** class has both **JavaLex** and **JavaYacc** instances, the error processing can detect errors and display the error message properly as long as the lexer and parser are working.

```
private static String yyGetToken(int tokenvalue) { }
```

This function takes a token value (integer) as input and returns an actual token context expressed in a character string. The user defined token list should be provided in order for this function to work.

```
private static int yyGetLineNumber() { }
```

This function returns the current line number that the lexical scanner is currently processing.

```
private static int yyGetErrorCount() { }
```

This function returns the current error number that the lexical scanner is currently processing.

```
private static void yyGetErrFileName(String s) { }
```

This function copies **yyErrSrc** to string **s**, which will provide the source file name the parser is processing.

```
private static void yySetErrText(String s) { }
```

This function will use a input parameter to set **yyErrText** (error message).

```
private static int yyGetCharNumber() { }
```

This function returns the beginning position of the current token on the current line, which the lexical scanner is processing.

```
private static int yyGetExpectedTokens(int token_list[]){ }
```

This function takes an integer array of the internal values for expected tokens as input and returns the exact number of those expected tokens.

```
private static void yyErrPrefix(String msg, String
YYTEXT) { }
```

This function takes a syntax error message as an input and displays it according to the current token information. The current token information including a line number, a character position number, …, etc, is provided by the lexer.

```
private static void yyError(String s, String t,
StringYYTEXT) { }
```

This function simply calls **yyErrPrefix** to finish error reporting by reformatting the error message display. The two input strings can represent two different error messages to be displayed.

```
private static void yySkipToken() { }
```

This function makes file indicator forward one unit to skip the current token as if the current token being processed by the lexer does not exist in the input stream.

```
private static void yySkipSymbol() { }
```

This function simply skips the current symbol in the sentential form if it is a terminal and keeps the current token unchanged.

```
private static void yyReplaceToken(String token) { }
```

This function takes a replacing token as an input and replaces the token that will be processed by the lexer.

```
private static int yyMatchToken(String token) { }
```

This function takes a matching token as an input parameter. It will return 1 if the current token matched the actual input token value, otherwise returns 0 instead.

```
private static void yyInsertToken(String token) { }
```

This function takes a token to be inserted as an input parameter. Its purpose is to insert a token in front of the one that will be processed by the lexer.

```
private static String yyDisplay(int tokenvalue) { }
```

Display token context based on the input token value. Normally, this function is provided by the user.

d. JavaParseTree Class

A general parse tree node class that encapsulates the common property for all the other tree nodes should be built as the base class. Additional information corresponding to a particular class will be added in the derived node class. Since Java language provides us with methods to redefine subclasses, some member functions in this base class should only have basic functionality to allow subclasses to modify them according to their needs.

(i). JavaParseTreeNode Class

Class Data Member:

```
JavaLex lex_tok;
```

Class Member Function:

```
JavaParseTreeNode() { }
```

This constructor initializes the private data member for the parse tree base class.

```
001: public JavaParseTree build_tree() { }
002: public JavaParseTree append_node() { }
003: public JavaParseTree delete_node() { }
004: public int execute() { }
005: public JavaParseTree lookup_tree() { }
006: public JavaParseTree optimize_tree() { }
007: public void print_tree() { }
008: public int get_token() { }
```

All these functions only provide prototype functionality. The actual functionality will be implemented in the member functions of the derived subclasses.

(ii). JavaLeaf Class

Class Data Member:

```
private static String tok_name;
private static int type;
```

These two members will allow **JavaLeaf** class to encapsulate the lexemes and the corresponding types.

Class Member Function:

```
JavaLeaf() { }
```

The task for this constructor is to initialize the data members for **JavaLeaf** class. All the preparation work for **JavaLeaf** object is done here.

(iii). JavaLeafList Class

Class Data Member:

```
JavaLeaf leaf_list_head;
JavaLeaf leaf_list_tail;
```

These two variables can be used to retrieve **JavaLeafList** information so that the user can use them to manipulate parse trees for statement list, function parameter list, etc.

Class Member Function:

```
JavaLeafList() { }
JavaLeafList(JavaParseTreeNode Leaf) { }
```

The first constructor will simply initialize the data members. The second one

will take a reference to **JavaParseTreeNode** as input, set corresponding information to the data members in **JavaLeafList** class.

```
public JavaParseTreeNode append_node(JavaParseTreeNode
Leaf) { }
```

This function will take a reference to a parse tree as an input. After appending the current leaf nodes to the parse tree, the corresponding reference to the parse tree node will be returned.

```
public JavaParseTreeNode delete_node(JavaParseTreeNode
Leaf) { }
```

This function will take a reference to a parse tree as an input. After deleting the current leaf nodes to the parse tree, the corresponding reference to the parse tree node will be returned.

```
public JavaLeaf create_leaf_list() { }
```

This function creates a leaf list according to the current leaf node's information hidden in the **JavaLeafList** object. A corresponding **JavaLeaf** reference to the created leaf list will be returned.

(iv). JavaExprNode Class

Class Data Member:

```
private static int type;
private static ABXVALUE value;
```

Every expression is associated with one type and one value. So these two variables will be designed as **JavaExprNode** class data members.

Class Member Function:

```
JavaExprNode() { }
JavaExprNode(int Type) { }
```

The initialization of data members for class **JavaExprNode** will be done in these two constructors.

```
JavaSymbolTable get_attr() { }
```

This function will search the symbol table to find a corresponding table entry for this current particular expression.

(v). JavaExprNodeList Class

Class Data Member:

```
JavaExprNode expr_head;
JavaExprNode expr_tail;
```

These two variables are used to keep track information for the linked list of expression nodes.

Class Member Function:

```
JavaExprNodeList() { }
JavaExprNodeList(JavaParseTreeNode Expr) { }
```

The initialization of data member for class **JavaExprNodeList** is done here.

```
JavaParseTreeNode append_node(JavaParseTreeNode Expr) { }
```

This function takes a reference to a parse tree node as an input, appends the current expression list to the parse tree node, and returns a reference to the decorated parse tree.

(vi). JavaParseTree Class

Class Data Member:

```
private JavaParseTreeNode root;
```

JavaParseTree class will store a reference to the root of a parse tree as its private data.

Class Member Function:

```
JavaParseTree(JavaParseTreeNode Root) { }
```

This constructor will take the parse tree generated by **PCYACC** machine or a parse tree root as input information, and then set the corresponding **JavaParseTree** data member to create a parse tree instance.

```
JavaParseTreeNode optimize(JavaParseTreeNode Tree) { }
```

This function will optimize the generated parse tree to make it more efficient.

```
void print_tree(JavaParseTreeNode Tree) { }
```

This function will create a hard copy for a specific parse tree.

```
void show_tree(JavaParseTreeNode Tree) { }
```

This function can display a parse tree based on the reference to a user specified parse tree.

```
JavaParseTreeNode execuate_tree(JavaParseTreeNode Tree) { }
```

This function can traverse the whole parse tree from the bottom, and return a reference to the top parse tree node based on a pointer to the user specified parse tree node.

```
JavaParseTreeNode decorate_tree(JavaParseTreeNode Tree) { }
```

This function can add some information to a specific parse tree, or insert a parse tree node, etc.

e. JavaSymbolTable Class

In Java, there are no equivalent structures to C++ **union**s and **struct**s. Generally **struct**s are easy to convert -- just change them to classes. Remember that in C++ the default access control in **struct**s is public (while in classes it is private), so mark members **public** accordingly.

```
001: union ABXVALUE {
002:      int i;
003:      double db;
004:      char ch[80];
005: };
```

Corresponds to

```
001: class ABXVALUE
002: {
003:      public int i;
004:      public double db;
005:      public char ch[80];
006: }
```

And

```
007: struct ABXsymtabentry
008: {
009:      char name[80];
010:      int attribute;
011:      int type;
012:      ABXVALUE value;
013:      int scope;
014:      struct ABXsymtabentry *next;
```

```
015: };
```

Corresponds to

```
001: class ABXsymtabentry
002: {
003:      public String name;
004:      public int attribute;
005:      public int type;
006:      public ABXVALUE value;
007:      public int scope;
008:      public ABXsymtabentry next;
009: }
010: struct ABXsymtab
011: {
012:      int scope;
013:      struct ABXsymtabentry *symbol_list;
014:      struct ABXsymtab *next;
015: };
```

Corresponds to

```
001: class ABXsymtab
002: {
003:      public int scope;
004:      public ABXsymtabentry symbol_list;
005:      public ABXsymtab next;
006: }
```

Class Data Member:

```
int scope_indicator;
```

This variable can be used to indicate the current scope number.

```
ABXsymtab symtab_head;
ABXsymtab symtab_tail;
```

These two variables are used to keep track of a symbol table list.

Class Member Function:

```
JavaSymbolTable() { }
```

Set head and current symbol table reference to NULL.

```
int addSymbol(String Name, int Attribute, int Type,
ABXVALUE Value, int Scope) { }
```

Add a symbol into a symbol table according to input parameters. If the new symbol is added successfully, this function returns 1, if the specified symbol already exists in symbol table, returns 2, otherwise returns 0. The current scope indicator and symbol table reference have been modified accordingly.

```
int cleanScope(int Scope) { }
```

Remove all symbols in a symbol table with scope **Scope**, set the current scope indicator and current symbol table reference accordingly. If the specified scope does not exist, function returns 0, otherwise returns 1.

```
int createscope(int Scope) { }
```

Create a new scope, set the current scope indicator and symbol table reference. If successful, function returns 1, if the scope requested already exists returns 2, otherwise returns 0.

```
int deleteSymbol(String Name, int Scope) { }
```

Delete a symbol table entry according to its name and scope. Return 1 if the symbol is deleted successfully. If the symbol does not exist, return 0. The current scope indicator and symbol table reference have been modified accordingly.

```
int getScope() { }
```

Get the value of the current scope indicator.

```
int getSymbolAttribute(String Name, int Scope) { }
```

Return a symbol's attribute if the symbol was found, otherwise return –1. The current scope indicator and symbol table reference have been modified accordingly.

```
int getSymbolType(String Name, int Scope, int Type) { }
```

Return a symbol's type if the symbol was found, otherwise return –1. The current scope indicator and symbol table reference have been modified accordingly.

```
ABXVALUE getSymbolValue(String Name, int Scope,
ABXVALUE Value) { }
```

Return a symbol's value if the symbol was found, otherwise return –1. The current scope indicator and symbol table reference have been modified accordingly.

```
void initSymbolTable() { }
```

Construct a symbol table with only scope 0 (global layer).

```
int insertSymbol(ABXsymtabentry Symbol) { }
```

Insert a symbol based on the current scope indicator and symbol table reference. If successful, return 1. Otherwise, return 0 (symbol already exists).

```
ABXsymtabentry lookupSymbol(String Name, int Scope) { }
```

Look up a symbol that is already inserted in the symbol table. If the symbol is found, return the symbol's symbol table entry reference, otherwise return NULL. The current scope indicator and symbol table reference have been modified accordingly.

```
int maxScope() { }
```

Get the biggest scope number of the symbol table.

```
int setScope(int Scope) { }
```

Set the current scope indicator to the **Scope** and current symbol table reference will be changed accordingly. If the specified scope does not exist, the function returns –1, Otherwise, it returns the scope number just set.

```
int setSymbolAttribute(String Name, int Scope, int
Attribute) { }
```

Set a symbol attribute value (defined by user) according to its name and scope. This function will return 1 if successful, otherwise return 0. The current scope indicator and symbol table reference have been modified accordingly.

```
int setSymbolType(String Name, int Scope, int Type) { }
```

Set a symbol data type according to its name and scope. Customers can define their own type by using macro definitions according to their specific compiler requirements. This function will return 1 if successful, otherwise, return 0. The current scope indicator and symbol table reference will be modified accordingly.

```
int setSymbolValue(String Name, int Scope, ABXVALUE
Value) { }
```

Set a symbol value according to its name and scope. Return 1 if proper entry has been found, otherwise return –1. The current scope indicator and symbol table reference will be modified accordingly.

3. Example

For flexibility reasons and cross platform compatibility, Java programming language is not as powerful as the C programming language. Translating a C parser into a Java parser is thus nontrivial and it is important to follow the correct procedures so that you can create a runnable Java parser when you use our **PCYTOOL** and **PCLTOOL** to translate C parser and lexer into Java parser and lexer.

If you want to create a standalone Java parser, you need to provide **yylex** function as a **JavaYacc** class member function in the user function section of GDF. **PCYTOOL** will check **yylex** function to identify if it is processing a standalone Java parser. All the content in the user function section of GDF will be treated as **JavaYacc** class members and put into **JavaYacc** class scope except user provided classes. Please check CALC or SQL example in our JAVA SDK package for details.

If you want to create a standalone Java lexer, you also need to provide **yyparse** function as a **JavaLex** class member function in the user function section of SDF. **PCLTOOL** will check **yyparse** function to identify if it is processing a standalone Java lexer. All the content in the user function section of SDF will be treated as **JavaLex** class members and put into **JavaLex** class scope except user provided classes. Please check the WC example in our JAVA SDK package for details.

If you want to create an integrated Java parser and lexer, there are at least three Java classes that have been created (main driver class, **JavaYacc** class and **JavaLex** class, which are generated from three different files). Everything inside the user function section will be treated as **JavaYacc** or **JavaLex** class members except the user provided classes. However, the user has to make sure that no **yylex** function is inside GDF and no **yyparse** function is inside SDF. Otherwise, **PCYTOOL** or **PCLTOOL** will generate a standalone Java parser or lexer. Since lexer will use token value information, so "*.h" files generated by **PCYACC** (-D option) should be included on the **PCLTOOL** command line. Please check JAVA grammar in the Java example for detail.

We will introduce a simple standalone calculator parser in Java here to demonstrate how to use our Java classes.

```
001:  %union{
002:      double db;
003:  }
004:
005:  %token NEWLINE
006:  %token NUMBER
```

```
007:  %left '+' '-'            /* left associative */
008:  %left '*' '/'            /* left associative */
009:  %left UNARYMINUS         /* left associative */
010:
```

Lines 1 through 6 form the so-called declaration section, where token symbols, operator precedences, etc., are declared.

Lines 001 through 003 define **YYSTYPE**. **PCYTOOL** will put the content of this union declaration into **imp_union** class since Java would not support **union** type by now.

Lines 005 through 006 declare two tokens, which can not be used on the left-hand side of grammar rules.

Lines 007 through 009 define the associativity of the arithmetic operators involved. All the operators are declared to be left associative (i.e., if the statement is a+b+c, the a+b will be calculated first). These statements also convey the following information: addition (+) and subtraction (-) have the same precedence; multiplication (*) and division (/) have the same precedence and it is higher than addition or subtraction; the unary operator, namely the negation sign, has the highest precedence.

```
011:  %%
012:
```

The delimiter **%%** on **line 011** separates grammar rule section from user declaration section..

Lines 013 through 043 form grammar rule section, which is the main body of GDF.

```
013:  list:    /* nothing */
014:         { prompt(); }
015:       | list NEWLINE /* change ALL '\n' to token newline */
016:         { prompt(); }
017:       | list expr NEWLINE
018:         {
019:             if ( $2.db == QUIT)
020:             {
021:                 return(0);
022:             }
023:             else
024:             {
025:                 System.out.println("  RESULT ====>" + $2.db);
026:                 System.out.flush();
027:                 prompt();
```

```
028:             }
029:         }
030:       | list error NEWLINE
031:         {
032:             yyerror;
033:             prompt();
034:         }
035:       ;
```

Lines 013 through 035 state that a list can either be empty, be a list followed by a new line character, be a list followed by an expression and a new line character, or be a list followed by something which is an error. Everything enclosed in braces is called **actions**. Any function used here except from **JavaYacc** class has to be provided by the user in the user function section of GDF and will be incorporated as **JavaYacc** class member functions.

```
036:  expr: NUMBER                  { $$.db = $1.db; }
037:      | '-' expr %prec UNARYMINUS { $$.db = -$2.db; }
038:      | expr '+' expr           { $$.db = $1.db + $3.db; }
039:      | expr '-' expr           { $$.db = $1.db - $3.db; }
040:      | expr '*' expr           { $$.db = $1.db * $3.db; }
041:      | expr '/' expr           { $$.db = $1.db / $3.db; }
042:      | '(' expr ')'            { $$.db = $2.db; }
043:      ;
```

Lines 036 through 043 contain the rules on how to form **expr** nonterminal. Since $x is a variable which will be used for communication with parser, so in these rules the correspondent content associated with every terminal or nonterminal has to be assigned value.

```
044:  %%
045:
046:
047:  private static final int QUIT = 101010;
048:  private static final int LF = 0x0a;
049:  private static final int CR = 0xd;
050:
051:  private static PushbackInputStream in =
052:      new PushbackInputStream(System.in);
053:
```

%% on **Line 044** is a delimiter that separates the grammar rule section from the program section. Everything in the program section must be written in Java, and it will be copied to the output of **PCYACC**. Then it will be moved into **JavaYacc** class body by **PCYTOOL** if the code portion is not the user provided class. This section also defines two major Java member functions:

yylex() and **yyerror()**. Since **yylex** function is provided, so **PCYTOOL** will treat this application as a standalone parser. These two functions are always required to provide support for the parser.

Lines 051 through 052 define a **PushbackInputStream** object as input source to the parser.

```
054:  private static int yylex()
055:  {
056:      /* user's own lexer written in Java */
057:  }
058:
```

Lines 054 through 057, the lexical analyzer, **yylex()** is responsible for decomposing raw text strings into meaningful lexical units, called tokens, and passing this information to the parser. Since this example is a standalone Java parser, the user has to write his/her Java lexer to return tokens to the syntactic parser.

```
059:  /* invoke prompt line */
060:  private static void prompt()
061:  {
062:      System.out.print("READY> ");
063:      System.out.flush();
064:  }
065:  /* display error message */
066:  private static void yyerror(String s, String t)
067:  {
068:      System.out.println(s+" near line " + pcyylineno);
069:  }
070:  /* auxiliary function for error reporting */
071:  private static String yydisplay(int ch)
072:  {
073:      int i=0, tok_index = 0;
074:      int tok_num = 2;
075:      String[] tok = new String[tok_num];
076:
077:      tok[tok_index++] = "NEWLINE";
078:      tok[tok_index++] = "NUMBER";
079:
080:      switch (ch)
081:      {
082:          case           0:   return ("[end of file]");
083:          case        '\b':   return (    "'\\b'"    );
084:          case        '\f':   return (    "'\\f'"    );
085:          case        '\n':   return (    "'\\n'"    );
086:          case        '\r':   return (    "'\\r'"    );
```

```
087:            case           '\t':   return (    "'\\t'"    );
088:            case     YYERRCODE:   return (   "[error]"   );
089:        }
090:
091:        if(ch > YYERRCODE && ch <= YYERRCODE + tok_num)
092:        {
093:            return (tok[ch - (YYERRCODE + 1)]);
094:        }
095:        return "char \""+(char)ch+"\"";
096:    }
097:
```

Lines 066 through 069, the error processing routine, **yyerror()** is called by the parser when a syntax error is uncovered during parsing.

Two functions called **yydisplay(int)** and **prompt()** are used for this special example. Function **yydisplay(int)** will return correspondent content based on the input token value. Function **prompt()** is used to invoke prompt line.

```
098:    private static int get_char()
099:    {
100:        try
101:        {
102:            return in.read();
103:        }
104:        catch (IOException e)
105:        {
106:            System.out.println(e);
107:            System.exit(-1);
108:        }
109:        return 0;
110:    }
111:
112:    private static void put_char(int c)
113:    {
114:        try
115:        {
116:            in.unread(c);
117:        }
118:        catch (IOException e)
119:        {
120:            System.out.println(e);
121:            System.exit(-1);
122:        }
123:    }
```

Lines 098 through 123 contain two auxiliary functions for **yylex**,

get_char() and **put_char(int)**. Their presence is to shorten **yylex** function's length.

```
124:  /* main driver class */
125:  class calc_jav
126:  {
127:      public static void main(String args[])
128:      {
129:          if (args.length>0)
130:              System.out.println("nonmeaningful arguments");
131:
132:          pcy_sk myparser = new pcy_sk();
133:          myparser.yyparse();
134:      }
135:  }
136:
```

Lines 125 through 136 contain user's own class calc_jav. This class calc_jav has main function, main(), which is to activate a parser, perform necessary initialization before activation and clean up after activation.

In general, all action code should be written in Java and you can still use **$x** variable to refer to the rule content. Our **PCYTOOL** will translate $x variables to the corresponding variables in **JavaYacc** class. All code in the user function section except the user-provided class will be put inside a single Java class whose name is **JavaYacc**. In order to make your parser runnable, you have to make sure that all the code you have written will work with **JavaYacc** class.

X. Delphi Parser and Lexer

1. Introduction

Delphi is a completely object oriented system. It is a viable alternative to C or C++ developer tools. Delphi's programming language, ObjectPascal, gives access to assembler and low-level windows event handling to satisfy any commercial software application needs. As the **PCYACC** compiler tool provider, **Abraxas Software** is obliged to develop Delphi parser and lexer to satisfy the current programming needs.

All the code you write in Delphi is stored in units. A unit is a separate file from your main program file that contains code and that may also contain variable declarations, constant declarations, objects, and anything else that a Pascal program can contain. Whenever you create a new form, Delphi automatically creates a unit just for that form.

Because Delphi creates a unit for each form, units serve to group like procedures together in one package. Based on this fact, we will create five basic units as following:

1). **DelphiLex Lexical Analyzer Unit**: this unit serves as a code skeleton for PCLEX.

2). **DelphiYacc Syntax Parser Unit**: this unit supports syntactic parser PCYACC.

3). **DelphiSymbolTable Symbol Table Unit**: this unit is used for symbol table management.

4). **DelphiError Error Handling Unit**: this unit is responsible for error reporting.

5). **DelphiParseTree Parse Tree Unit**: this unit is available when the user wants to construct parse trees.

There are two separate phases for generating a Delphi Parser. The following diagram shows the phases:

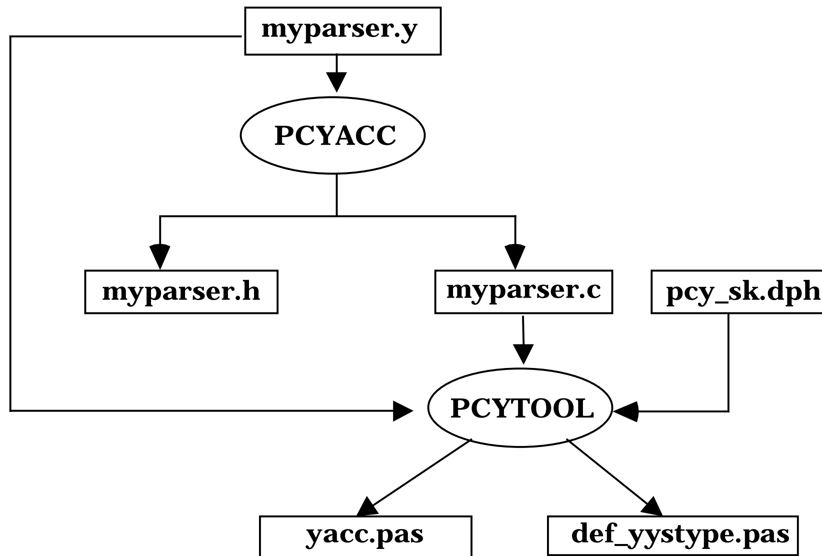

Figure 10-1. Diagram of Delphi parser generated by PCYTOOL

In the first phase, based on the availability of grammar description file, simply invoke **PCYACC** tool on the command line to create a C parser. Once the C parser is generated by **PCYACC**, a utility program named **PCYTOOL** is needed for generating a Delphi parser in the second phase, in which yacc.pas represents **DelphiYacc** unit and def_yystype.pas represents the unit for **YYSTYPE** in **PCYACC**.

There are two separate phases for generating a Delphi lexer. The following diagram shows the phases:

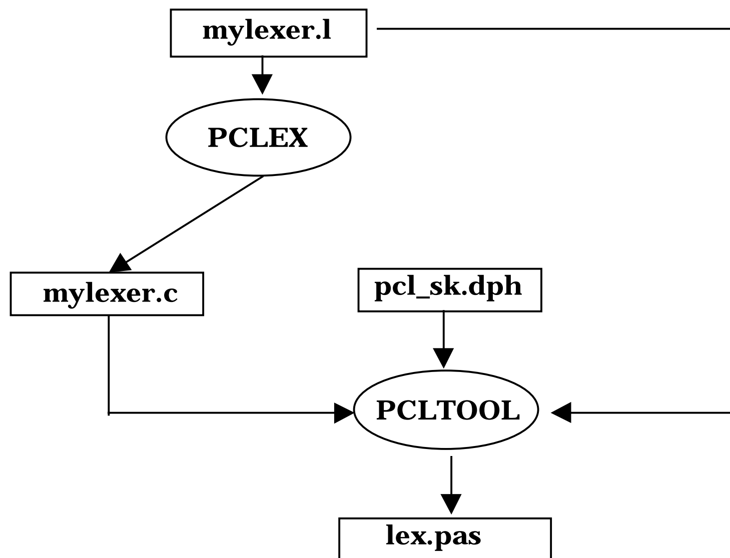

Figure 10-2. Diagram of Delphi lexer generated by PCLTOOL

In the first phase, based on the availability of a scanner description file, simply invoke **PCLEX** tool on the command line to create a C lexer. Once the C lexer is generated by **PCLEX**, a utility program named **PCLTOOL** is needed for generating a Delphi lexer in the second phase.

2. Delphi Unit Library

Although Delphi is completely object-oriented, it uses unit instead of class in C++ and Java language. So in order to provide the equivalent functionality, we will introduce several units to realize the full equivalent functionality of our C++ and Java class libraries. These units will be used for generating Delphi parser, lexer and for other auxiliary functionality. Detailed definitions and member function descriptions will be given for each unit.

a. DelphiLex Unit

The Constant Declaration Part:

Constant declaration part declares constants within the block containing the declaration.

```
001:    const
```

```
002:         YYERRCODE = 256;
003:         YY_END_TOK = 0;
004:         YY_NEW_FILE = -1;
005:         YY_DO_DEFAULT = -2;
006:         BUFSIZ = 4096;
007:         F_BUFSIZ = 4096;
008:         YY_BUF_SIZE = 8192;
009:         YY_BUF_MAX = 8191;
010:         YY_MAX_LINE = 4096;
011:         YY_BUF_LIM = 4095;
012:         YY_NULL = 0;
```

Delphi language does not support macro definition. However, in a traditional C lexer generated by **PCLEX** tool, there are a lot of macros involved, which provides a lot of convenience to users. The previously listed constants like **F_BUFSIZ, YY_BUF_SIZE, YY_BUF_MAX, YY_MAX_LINE** and **YY_BUF_LIM**, have been defined based on **BUFSIZ** with a default value of 4096. For details about how to assign the values of these constants, please refer to a C lexer source code generated by **PCLEX**.

The Type Declaration Part:

Programs, procedures, functions, and methods have a type declaration part to declare types.

```
013:     type
014:         charstr = Array[0..YY_BUF_SIZE] of Char;
015:         TLex = class(TObject)   { lex class }
016:         private
017:         protected
018:             YYlval : yystype;
019:             YYval : yystype;
020:             yylineno : Integer;
021:             filvar:Text;
022:             input_file_name:String[64];
023:             yy_start:Integer;
024:             yy_b_buf_p:Integer;
025:             yy_c_buf_p:Integer;
026:             yy_e_buf_p:Integer;
027:             yy_saw_eof:Integer;
028:             yy_init:Integer;
029:             yy_ch_buf:charstr;
030:             yy_st_buf:Array[0..YY_BUF_SIZE-1] of Integer;
031:             yy_hold_char:Char;
032:             yytext:Array[0..BUFSIZ-1] of Char;
033:             yyErrorCount : Integer;
034:             yyErrSrc : String;
```

```
035:            yyleng:Integer;
036:            yy_lp:Integer;
037:            yy_curst:Integer;
038:            reject_flag:Integer;
039:            list: Integer;
040:            column: Integer;
041:        public
042:            constructor create;
043:            destructor destroy;
044:            Procedure set_input_file(name:String);
045:            Function yylex:Integer;
046:            Function get_yylineno:Integer;
047:            Function input:Integer;
048:            Procedure unput(c:Char);
049:            Procedure YY_DEFAULT_ACTION;
050:            Function YY_INPUT(var buf:charstr;
051:            index:Integer; max_size:Integer):Integer;
052:            Procedure YY_OUTPUT(c:Char);
053:            Procedure YY_FATAL_ERROR(s:String);
054:            Function yywrap:Boolean;
055:            Procedure yyless(n:Integer);
056:            Procedure YY_INIT_PROC;
057:            Function YY_LENG:Integer;
058:            Procedure YY_DO_BEFORE_SCAN;
059:            Procedure YY_DO_BEFORE_ACTION;
060:            Procedure REJECT(yy_full_match:Integer);
061:            Procedure yyerror(s:String; t:String);
062:            Procedure errprefix(msg:String);
063:        end;
```

Nearly everything that you will deal with while programming in Delphi is an object. Although Delphi directly implements the following OOP concepts: **Encapsulation**, **Inheritance** and **Polymorphism**, Delphi does not support multiple inheritance and parametric polymorphism, known as **overloading**. Multiple inheritance allows a single object inherit traits from two non-related ancestor classes. Delphi is highly typed. Variable assignments must meet strict assignment compatibility rules. Due to this strict type assignment philosophy, overloading was left out of the Delphi language.

In **DelphiLex** unit, we declare an object type **TLex,** which uses the **class** keyword. This will ensure that your new class will inherit all the properties and behavior included in the Delphi **TObject** type. The data fields contain values pertinent to that object.

The variables declared in data fields of **TLex** object are originally defined as globals in a C lexer since all these variables are used for the lexer.

Following data fields is procedure and function heading part, which is in the public section of **TLex** object. Unless a procedure or function is inline, the interface part only lists the procedure or function heading.

The Implementation Part:

This part defines the block of all public procedures and functions. Function declaration defines a block that computes and returns a value, and procedure declaration associates an identifier with a block as procedure.

```
constructor TLex.create;
destructor TLex.destroy;
```

The constructor not only calls the inherited **Create** method, but also gives the instance variables the initial state you want them to have. The destructor calls the inherited Destroy method, which is invoked when an object disappears.

```
Procedure TLex.set_input_file(name:String);
```

Allow the lexer to switch scanning from one input file to another. This enables scanner to be interrupted to process another input instead of the current one.

```
Function TLex.yylex:Integer;
```

Separate an input character stream into tokens following the patterns specified in the lex (*.l) file. This function varies for each different lexical analyzer because the embedded user actions defined in the lex (*.l) file are different.

```
Function TLex.get_yylineno:Integer;
```

This function returns the current line number that is very useful in error reporting.

```
Function TLex.input:Integer;
```

This function returns the next character from input.

```
Procedure TLex.unput(c:Char);
```

This procedure puts a character back in the logical input stream.

```
Procedure TLex.YY_DEFAULT_ACTION;
```

This procedure is defined as a macro in a C lexer. It will emit information of **yytext** buffer to the standard output.

```
Function TLex.YY_INPUT(var buf:charstr; index:Integer;
max_size:Integer):Integer;
```

This function deals with storage of input stream into a lexer. It gets an input from the input stream and stores it into a buffer, which the user can specify as the function call parameter.

```
Procedure TLex.YY_OUTPUT(c:Char);
```

This procedure outputs one character to the standard output.

```
Procedure TLex.YY_FATAL_ERROR(s:String);
```

This procedure prints out the string specified in as the function call parameter to the standard error.

```
Function TLex.yywrap:Boolean;
```

This function simply returns boolean value **true**.

```
Procedure TLex.yyless(n:Integer);
```

This procedure tells lex to “push back” part of the token that was just read. The argument to **yyless()** is the number of token characters to push back into the input stream.

```
Procedure TLex.YY_INIT_PROC;
```

This procedure is used to initialize the scanner’s state.

```
Function TLex.YY_LENG:Integer;
```

This function returns the length for the text of the token stored in **yytext**.

```
Procedure TLex.YY_DO_BEFORE_SCAN;
```

This procedure puts the character that the scanner holds at the end of **yytext**.

```
Procedure TLex.YY_DO_BEFORE_ACTION;
```

This procedure does some preparation job before the scanner takes actions.

```
Procedure TLex.REJECT(yy_full_match:Integer);
```

This procedure puts back the text matched by the pattern and finds the next best match for it.

```
Procedure TLex.yyerror(s:String; t:String);
```

This procedure simply calls **errprefix** to finish error reporting by reformatting the error message display. The two input strings can represent two different error messages to be displayed.

```
Procedure TLex.errprefix(msg:String);
```

This procedure takes a syntax error message as an input and displays it according to the current token information. The current token information including the line number, the character position number, ..., etc, is provided by the lexer.

b. DelphiYacc Unit

The Constant Declaration Part:

```
001:    const
002:        YYERRCODE = 256;
003:        YYMAXDEPTH = 200;
004:        YYREDMAX = 1000;
005:        PCYYFLAG = -4096;
006:        WAS0ERR = 0;
007:        WAS1ERR = 1;
008:        WAS2ERR = 2;
009:        WAS3ERR = 3;
```

These variables are defined as globals in a C parser. In Delphi, they are defined as **DelphiYacc** unit's constants.

The Type Declaration Part:

```
010:    type
011:        valuestack = Array [0..YYMAXDEPTH-1] of yystype;
012:        TYacc = class (TLex) { yacc class }
013:        private
014:        protected
015:            yyOutFile : File of Char;
016:            yyErrFile : File of Char;
017:            yyErrorFlag : Integer;
018:            yyToken : Integer;
019:            yyValueStack : ^valuestack;
020:            redSequence : ^Integer;
021:            redCount : Integer;
022:        public
023:            constructor create;
024:            destructor destroy;
```

```
025:            function yyParse: Integer;
026:            function get_yyErrorCount : Integer;
027:            procedure yyerrok;
028:        end;
```

In **DelphiYacc** unit, we declare an object type **TYacc,** which uses the **class** keyword. This will ensure that your new class will inherit all the properties and behavior included in the **TLex** type. The data fields contain values pertinent to that object.

The variables declared in the data fields of **TYacc** object are originally defined as globals in a C parser since all these variables are used for a parser.

Following data fields is procedure and function heading part, which is in the public section of **TYacc** object. Unless a procedure or function is inline, the interface part only lists the procedure or function heading.

The Implementation Part:

This part defines the block of all public procedures and functions. In which, function declaration defines a block that computes and returns a value, and procedure declaration associates an identifier with a block as a procedure.

```
constructor TYacc.create;
desctructor TYacc.destroy;
```

The constructor not only calls the inherited **Create** method, but also gives the instance variables the initial state you want them to have. The destructor calls the inherited **Destroy** method, which is invoked when an object disappears. Also deallocate memory allocated in the constructor.

```
function TYacc.yyParse: Integer;
```

This function is the entry point to a yacc-generated parser. When your program call the member function **yyParse()**, the parser attempts to parse an input stream. The parser returns a value of zero if the parse succeeds and non-zero if not.

```
function TYacc.get_yyErrorCount: Integer;
```

This function simply returns **pcyyerrcnt** value in case the other unit would like to access this internal variable of **DelphiYacc** unit.

```
procedure TYacc.yyerrok;
```

This procedure is implemented to replace a macro in a C parser. It tells the parser to return to the normal state, which can avoid the problem of multiple

error messages resulting from a single mistake as the parser gets resynchronized.

3. Example

Since **PCYTOOL** will search for **yyparse** function in user function section of GDF, so **yyparse** function appearance in GDF will decide if **PCYTOOL** is processing a standalone parser. **PCLTOOL** will follow the same scenario. **yylex** function appearance in SDF will inform **PCLTOOL** that it is dealing with a standalone lexer.

If you want to create standalone Delphi parser, please check the CALC example in our DELPHI SDK package for detail.

If you want to create both Delphi parser and lexer as a single application, please check out the ODL example in our DELPHI SDK package for detail.

Before you create Delphi parser or lexer, please make sure that every use function inside GDF or SDF you write will be written in Delphi so that you can create a runnable Delphi parser or lexer.

Here, we will introduce a simple standalone calculator parser in Delphi to demonstrate how to use our **DelphiYacc** unit.

```
001:  %{
002:  const
003:      QUIT = 101010;
004:      STRINGLEN = 1023;
005:
006:  %}
007:  %union {
008:      i: Integer;
009:      db: Double;
010:      str: String;
011:  }
012:
013:  %token NUMBER
014:  %left '+' '-'     /* left associative */
015:  %left '*' '/'     /* left associative */
016:  %left UNARYMINUS  /* left associative */
017:
```

Lines 001 through 013 form the so-called declaration section, where token symbols, operator precedences, etc., are declared.

Lines 001 through 006 are user declaration section. This section declares some constants, which will be used in **DelphiYacc** unit.

Lines 007 through 011 define **YYSTYPE**. **PCYTOOL** will put the content of this union declaration into the **def_yystype** unit since Delphi would not support a **union** type.

Line 013 declares one token, which will be used as terminal in the grammar rule.

Lines 014 through 016 define the associativity of the arithmetic operators involved. All the operators are declared to be left associative (i.e., if the statement is a+b+c, the a+b will be calculated first). These statements also convey the following information: addition (+) and subtraction (-) have the same precedence; multiplication (*) and division (/) have the same precedence and it is higher than addition or subtraction; the unary operator, namely the negation sign, has the highest precedence.

```
018:  %%
019:
020:  list:    /* nothing */
021:        {   prompt();    }
022:       | list '\n'
023:        {   prompt();    }
024:       | list expr '\n'
025:        {
026:            If ( $2.i = QUIT) Then
027:            Begin
028:                yyparse := 0;
029:                Goto endyyparse;
030:            End
031:            Else
032:            Begin
033:                WriteLn(Output, 'Result =====>', $2.db:10:2);
034:                prompt();
035:            End;
036:        }
037:       | list error '\n'
038:        {
039:            yyerrok;
040:            prompt();
041:        }
042:       ;
043:  expr: NUMBER              { $$.db := $1.db; }
044:       | '-' expr %prec UNARYMINUS { $$.db := -$2.db; }
045:       | expr '+' expr       { $$.db := $1.db + $3.db; }
046:       | expr '-' expr       { $$.db := $1.db - $3.db; }
047:       | expr '*' expr       { $$.db := $1.db * $3.db; }
048:       | expr '/' expr       { $$.db := $1.db / $3.db; }
049:       | '(' expr ')'        { $$.db := $2.db; }
```

```
050:      ;
```

Lines 018 through 050 form the grammar rule section, which is the main body of GDF.

Lines 020 through 042 say that a list can either be empty, be a list followed by a new line character, be a list followed by an expression and a new line character, or be a list followed by something, which is an error. Everything enclosed in braces is called **actions**. Any function used here except from **DelphiYacc** unit's own member functions has to reside in the user function section of GDF. The user needs to make sure this usage can be successfully integrated into **yyparse** function of **DelphiYacc** unit.

Lines 043 through 050 contain the rule on how to form **expr** nonterminal. Since **$x** is a variable which will be used for communication with parser, in these rules the corresponding content associated with every terminal or nonterminal has been assigned a value.

```
051:  %%
052:  (* main driver *)
053:  Program calc_pas(Input, Output);
054:
055:  Uses yacc;
056:  Var
057:      myyacc : TYacc;
058:  Begin
059:      myyacc := TYacc.create;
060:      myyacc.yyparse;
061:      myyacc.destroy;
062:  End.     (* end of program calc_pas *)
063:
064:  Function TLex.yylex:Integer;
065:  (* user's own lexer written in Delphi *)
066:  Begin
067:  End; (* end of Function yylex *)
068:
069:  Procedure TLex.yyerror(s:String);
070:  Begin
071:      WriteLn(Output, s, ' near line ', yylineno);
072:  End; (* end of procedure yyerror *)
073:
074:  Procedure prompt;
075:  Begin
076:      Write(Output, 'READY> ');
077:  End; (* end of prompt *)
078:
```

%% on **Line 051** is a delimiter that separates the grammar rule section from the program section. Everything in the program section must be written in Delphi, and it will be copied to output of **PCYACC**. Then it will be moved into **DelphiYacc** unit body by **PCYTOOL** except the user provided program body. This section defines one major Delphi function: **yylex** and one major Delphi procedure: **yyerror**. These functions and procedures are always required to provide support for the parser.

Program calc_pas is to activate the parser, perform necessary initialization before activation and to clean up after activation.

The lexical analyzer, **yylex()** is responsible for decomposing raw text strings into meaningful lexical units, called tokens, and passing this information to the parser. Since this example is a standalone Delphi parser, the user has to write his/her Delphi Lexer to return the token to the syntactic parser.

The error processing routine, **yyerror()**, is called by the parser when syntax error is uncovered during parsing.

Procedure **prompt()** is used to invoke prompt line.

Generally, all the action code should be written in Delphi except that you can still use **$x** variable to refer rule content. Our **PCYTOOL** will translate $x variables to corresponding variables in **DelphiYacc** unit. All the code in the user function section except the user's own program body will be put inside **DelphiYacc** unit. In order to make your parser runnable, you have to make sure that all the code you have written will work with **DelphiYacc** unit.

XI. VBScript Parser and Lexer

1. Introduction

Microsoft Visual Basic, Scripting Edition, most commonly referred to as VBScript, is one of the most valuable tools available for Web page development. VBScript is a subset of the popular Visual Basic development language. It has been designed to be lightweight, fast, and safe. Because of its close link to Visual Basic, VBScript can be easily learned by the current Visual Basic programmers. Also VBScript allows you to add Visual Basic code directly to HTML document. It is the tool you will use to bring client-side processing to your web pages and applications. As a major YACC tool provider, it becomes very important for us to provide tools to generate VBScript parser and lexer.

VBScript is a text-based, interpreted language that is downloaded to the browser within the HTML stream. Once at the browser, the VBScript program is compiled by the VBScript engine within the browser and placed into memory, where it then waits to be executed. All VBScript code has to be included between <**SCRIPT**>, the begin script tag, and </**SCRIPT**>, the end script tag.

Code in VBScript is stored in modules. Each module can contain:

• Declarations. You can place constant, type, variable, and dynamic-link library (DLL) procedure declarations at the module level of form, class or standard modules.

• Procedures. A Sub, Function, or Property procedure contains pieces of code that can be executed as a unit.

Modules (.bas file name extension) are containers for procedures and declarations commonly accessed by other modules within the application. They can contain global (available to the whole application) or module-level declarations of variables, constants, types, external procedures, and global procedures.

There are two separate phases for generating a VBScript Parser. The following diagram shows the phase:

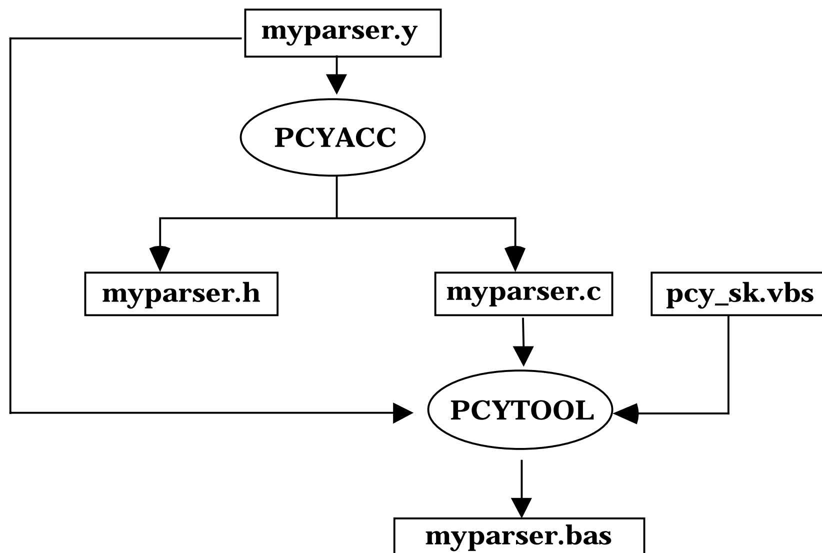

Figure 11-1. Diagram of VBScript parser generated by PCYTOOL

In the first phase, based on the availability of a grammar description file, simply invoke **PCYACC** tool on the command line to create a C parser. Once the C parser is generated by **PCYACC**, a utility program named **PCYTOOL** is needed for generating a VBScript parser in the second phase, in which myparser.bas is final VBScript parser.

There are two separate phases for generating a VBScript Lexer. The following diagram shows the phase:

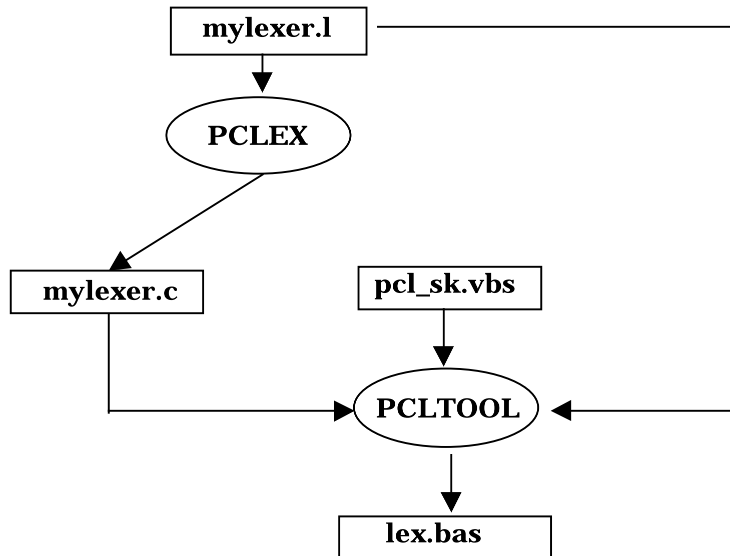

Figure 11-2. Diagram of VBScript lexer generated by PCLTOOL

In the first phase, based on the availability of a scanner description file, simply invoke **PCLEX** tool on the command line to create a C lexer. Once the C lexer is generated by **PCLEX**, a utility program named **PCLTOOL** is needed for generating a VBScript lexer in the second phase.

2. Structure of VBScript Parser and Lexer

Since all the VBScript code will be put between <SCRIPT>, the begin tag, and </SCRIPT>, the end tag, we have to integrate parser and lexer files into one single VBScript file. All the procedures are identical to their counterparts of the C parser and lexer. The macros in C parser and lexer will be translated into the corresponding VBScript procedures.

The VBScript code of a lexical analyzer generated by **PCLTOOL** has the following layout,

1. Code from the section 1 of .l file
2. Data tables
3. Module-level variables
4. Auxilary procedures
5. Function yylex()
6. Code from the section 3 of .l

Figure 11-3. Structure of VBScript Code Generated by PCLTOOL

1). Code segment copied directly from the declaration section of lexer file: This part contains the declarations of variables and functions to be used in embedded actions. It varies with different lexical analyzers and it is optional. The code in this section should be written in VBScript language.

2). Data tables: This part consists of the data tables for driving the **Deterministic Finite Automaton** (DFA) simulator. They are different for different lexical analyzers.

3). Module-level variables: This part consists of the variables representing the input stream buffer and pointers that indicate the status of the input being scanned. They should be almost the same for different lexical analyzers.

4). Auxiliary procedures: This part defines the procedures that are called by the lexer function **yylex()**.

5). Function **yylex()**: This part defines the function **yylex()**.

6). Code segment copied directly from the function section of the lexer file: This part contains the possible function definitions by the user. It is also optional. The User is responsible for writing the valid VBScript code.

The layout of generated VBScript YACC code is shown below.

Code copied directly from the section 1 of .y file (if there is any)
Data tables
Constants representing tokens
Code copied directly from the section 3 of .y file (if there is any)
Auxiliary Yacc procedures
Function yyparse()

Figure 11-4 Layout of Generated VBScript YACC code

1). Code copied directly from the section 1 of .y file: This part contains the declarations of variables and functions to be used in embedded actions. It varies with different parsers and it is optional. The code in this section should be written in VBScript language.

2). Data tables: This part consists of the parsing tables for driving yacc machine. They are different for different syntax analyzers.

3). Constants representing tokens: This part consists of all tokens defined in grammar file.

4). Code copied directly from the section 3 of .y file: This part contains the possible function definitions by user. It is also optional. User is responsible for writing valid VBScript code.

5). Auxiliary yacc procedures: This part defines the procedures that are called by **yyparse** function.

6). Function yyparse(): This part contains **yyparse()** function definition.

a. VBScript Lex Modules

All the auxiliary procedures related to lexer are listed below:

```
Sub YY_DO_BEFORE_ACTION()
```

This procedure does some preparation job before scanner takes actions.

```
Sub YY_DEFAULT_ACTION()
```

This procedure is defined as a macro in a C lexer. It will emit information of **yytext** buffer to the output the user defined.

```
Sub YY_DO_BEFORE_SCAN()
```

This procedure puts the character that the scanner holds at the end of **yytext**.

```
Function yywrap
```

This procedure simply returns 1.

```
Sub YY_FATAL_ERROR(s)
```

This procedure prints out the string specified in as the function call parameter to the output the user defines.

```
Sub YY_INIT_PROC()
```

This procedure is used to initialize the scanner's state.

```
Function YY_INPUT(buf, index, maxsize)
```

This procedure deals with storage of an input stream into a lexer. It gets an input from the input stream and stores it into a buffer, which the user can specify as the function call parameter.

```
Sub YY_OUTPUT(c)
```

This procedure outputs one character to the output the user defines.

```
Function input()
```

This procedure returns the next character from the input stream.

```
Sub unput(c)
```

This procedure puts a character back to the logical input stream. The user can call several times in a row to put several characters back into the input stream.

```
Function yylex()
```

This procedure is used to start or resume scanning. It separates an input character stream into tokens following the patterns specified in lex (*.l) file.

b. VBScript Yacc Modules

All the auxiliary procedures related to a parser are listed below:

```
Sub yyerrok()
```

This procedure is implemented to replace a macro in C parser. It tells the parser to return to the normal state, which can avoid the problem of multiple error messages resulting from a single mistake as the parser gets resynchronized.

```
Function yyparse()
```

This procedure is the entry point to a yacc-generated parser. When your program calls the member function **yyparse()**, the parser attempts to parse an input stream. The parser returns a value of zero if the parse succeeds and non-zero if not.

c. VBScript Error Report Modules

All the auxiliary procedures related to error reporting are listed below:

```
Sub set_input_file_name(fname)
```

This procedure is used to set **pcyyerrsrc** so that error report could show which input source file the parser is processing.

```
Sub errprefix(msg)
```

This procedure takes a syntax error message as an input and displays it according to the current token information. The current token information including the line number, the character position number, etc, is provided by the lexer.

```
Sub yyerror(s, t)
```

This procedure simply calls **errprefix** to finish error reporting by reformatting the error message display. The two input strings can represent two different error messages to be displayed.

```
Function yydisplay(token)
```

This procedure displays token context based on the input token value. Normally, this function is provided by the user.

```
Sub yyskiptoken()
```

This procedure makes file indicator forward one unit to skip the current token as if the current token being processed by the lexer does not exist in the input stream.

```
Sub yyskipsymbol()
```

This procedure simply skips the current symbol in the sentential form if it is a terminal and keeps the current token unchanged.

```
Sub yyreplacetoken(token)
```

This procedure takes a replacing token as an input and replaces the token that will be processed by the lexer.

```
Function yymatchtoken(token)
```

This procedure takes a matching token as an input parameter. It will return 1 if the current token matched the actual input token value, otherwise returns 0 instead.

```
Sub yyinserttoken(token)
```

This procedure takes a token to be inserted as an input parameter. Its purpose is to insert a token in front of the one that will be processed by the lexer.

3. Example

HTML allows the user to include a script directly by using the HTML comment tags around the user's VBScript code. One comment tag is placed after the <SCRIPT> tag and the other before the </SCRIPT> tag. The initial <SCRIPT> tag must include the **LANGUAGE** property identifies the scripting language that the browser should use to interpret the code.

```
001: <SCRIPT LANGUAGE = "VBSCRIPT">
002: <!——
003: •
004: •
005: •
006: -->
007: </SCRIPT>
```

Since our **PCYTOOL** and **PCLTOOL** are not language translators, everything involving the user defines has to be written in the VBScript language. In order to make it easier for the customer to get information from or to a parser, the customer can still use $x variable to refer to the parser internal information. However, there will be restrictions on the GDF and

SDF files. To generate a runnable VBScript parser, the specific procedures have to be followed.

Here, we will introduce one example that can illustrate how to write a corresponding grammar description file and scanner description file in order to get a VBScript parser and lexer generated by our **PCYTOOL** and **PCLTOOL**.

The following is a SDF for a Simple Calculator, which exhibits the typical structure of a scanner description file used to generate a VBScript lexer by **PCLTOOL**. For reference, line numbers are added to the listing.

Lines 001 through 014 form the **definition** section, where needed global variables and several regular expression macros are declared. **Lines 008 through 013** define several regular expression macros. **Lines 017 through 064** make up the **rule** section where the input patterns to match and their corresponding actions are defined. **Lines 067 through 190** are the user function section with the necessary support functions written in **VBScript**.

```
001:
002:  %{' declare variables for error reporting
003:  Dim yyerror_column, yyerror_list
004:  Dim yyerror_msg       ' String type
005:  Dim lex_index
006:  %}
007:
008:  letter                        [a-zA-Z_]
009:  alphanum                      [a-zA-Z_0-9]
010:  digit                         [0-9]
011:  blank                         [ \t]
012:  sign                          [+-*/]
013:  other                         .
014:
```

Lines 003 through 005 define some global variables, which are necessary for error reporting.

Lines 008 through 013 define several regular expression macros. Macro names are substitutions of the corresponding pattern. The patterns can be referred in the rule section with the macro names in braces, for example, **"{letter}"**.

```
015:  %%
```

The "**%%**" delimiter standing alone on **line 015** separates the definition section from the rule section.

The rule section spans from **lines 017 through 064**. In this section, the user has to assign **yytext** content to **yylval**, which is a variable shared between the parser and lexer.

```
016:  /* assign yytext content to yylval in each rule */
017:  {letter}{alphanum}*   {
018:                        For lex_index=0 To UBound(yytext)
019:                            yylval(lex_index) =
                                             yytext(lex_index)
020:                        Next
021:                        yylex = yysearch(yytext)
022:                        Exit Function
023:                        }
```

The pattern on **lines 017 through 023** matches both **identifiers** and **keywords**. The function **yysearch(yytext)** defined in the user function section determines which one it is.

```
024:  {digit}+              {
025:                        For lex_index=0 To UBound(yytext)
026:                            yylval(lex_index) =
                                             yytext(lex_index)
027:                        Next
028:                        yylex = NUMBER
029:                        Exit Function
030:                        }
031:
032:  {digit}+\.{digit}*    {
033:                        For lex_index=0 To UBound(yytext)
034:                            yylval(lex_index) =
                                             yytext(lex_index)
035:                        Next
036:                        yylex = NUMBER
037:                        Exit Function
038:                        }
039:
040:  \.{digit}+            {
041:                        For lex_index=0 To UBound(yytext)
042:                            yylval(lex_index) =
                                             yytext(lex_index)
043:                        Next
044:                        yylex = NUMBER
045:                        Exit Function
046:                        }
047:
```

The three rules above on **lines 024 through 046** handles **numeric**

constants. All integers and floating point numbers are returned as NUMBER.

```
048:  sign                      {
049:                            yylex = yytext(0)
050:                            Exit Function
051:                            }
052:
```

The pattern on **lines 048 through 051** handles **signed** character.

```
053:  \n                        {
054:                            yylineno = yylineno + 1
055:                            yylex = Asc("\n")
056:                            Exit Function
057:                            }
058:
```

Lines 053 through 057 say when an end of line is reached ('\n' is shorthand for new line), add one to the line counter (**yylineno**).

```
059:  {blank}+                  ;
060:
```

Line 059 handles white spaces. The scanner discards blanks and tabs.

```
061:  {other}                   {
062:                            yylex = yytext(0)
063:                            Exit Function
064:                            }
```

Lines 061 through 064 handle the situation other than those discussed above.

```
065:  %%
```

The "%%" delimiter standing alone on **line 065** separates the rule section from the user function section.

The user function section spans **lines 067 through 190**.

```
066:  ' decide whether a token is an identifier or a reserved
     keyword
067:  Function yysearch(arr) ' arr is Array type
068:  Dim i, tmp_str
069:      tmp_str = ""
070:      For i=0 To UBound(arr)
071:          If ( arr(i)=0 ) Then
```

```
072:              If ( tmp_str="SQRT" ) Then
073:                  yysearch = SQRT       ' SQRT token
074:              Else
075:                  yysearch = IDENTIFIER ' IDENTIFIER token
076:              End If
077:              Exit Function
078:          End If
079:          tmp_str = tmp_str & String(1, Chr(arr(i)))
080:      Next
081:  End Function
```

The function **yysearch()**, determines whether a name is an **identifier** or a **reserved keyword**. Other functions in this section are the error reporting routines for this Simple Calculator.

```
082:  ' translate the content in an array variable into a
     corresponding string variable
083:  Function array_to_string(arr) ' arr is array type
084:  Dim i, temp_str
085:      temp_str = ""
086:      For i=0 To UBound(arr)
087:          If ( arr(i)=0 ) Then
088:              array_to_string = temp_str
089:              Exit Function
090:          End If
091:          If ( Asc(String(i+1, Chr(arr(i))))=10 ) Then
092:              temp_str = temp_str & "\n"
093:          Else
094:              temp_str = temp_str & String(i+1, Chr(arr(i)))
095:          End If
096:      Next
097:      array_to_string = temp_str
098:  End Function
```

Function **array_to_string()** spans from **lines 083 through 098**, which will translate the content in an array variable into a corresponding string variable. It is not necessary if the function for this kind of data conversion is available in the future improved VBScript language.

The error reporting functions are independent of the lexical scanner defined above. They are for the parser part and could be put into another file or the program part of the grammar description file. The reason for putting them here 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.

```
099:  ' provide line number and error count number for error
```

```
     report
100:  Sub errprefix(msg) 'msg must be string type
101:  Dim punct
102:
103:      punct = 1
104:      yyerror_msg = yyerror_msg & "[error " &
                        Chr(Asc(pcyyerrcnt+1)) & "] "
105:
106:      If (yylineno>=0) Then
107:          If (punct<>0) Then
108:              yyerror_msg = yyerror_msg & ", "
109:          End If
110:          yyerror_msg = yyerror_msg & "line " &
                            Chr(Asc(yylineno))
111:          punct = 1
112:      End If
113:
114:      If (yytext(0)<>0) Then
115:          If (punct<>0) Then
116:              yyerror_msg = yyerror_msg & " "
117:          End If
118:          yyerror_msg = yyerror_msg & "near '" &
                            array_to_string(yytext) &  "'"
119:          punct = 1
120:      End If
121:
122:      If (punct<>0) Then
123:          yyerror_msg = yyerror_msg & ": "
124:      End If
125:      yyerror_msg = yyerror_msg & msg & Chr(13) & Chr(10)
126:  End Sub
```

An auxiliary sub, **errprefix()** spanning from **lines 100 through 126**, is called by **yyerror()** to report the location of the error occurred and the current error number.

```
127:  ' display token context
128:  Function yydisplay(token)  ' must return a string
129:
130:      If ( token>=0 And token<=255) Then
131:          yydisplay = "'" & Chr(token) & "'"
132:      Else
133:          yydisplay = "char " & token
134:      End If
135:      Exit Function
136:
137:  End Function
```

```
138:
```

An auxiliary sub, **yydisplay(token)** spanning from **lines 128 through 137**, is used to display token context based on an input token value. Normally, this function is provided by the user.

```
139:  ' display error message, s and t must be string type
140:  Sub yyerror(ByVal s, ByVal t)
141:  Dim expecting
142:      expecting = "expecting: "
143:
144:      yyerror_column = 0
145:
146:      If (Len(s)<>0) Then
147:          If (yyerror_column<>0) Then
148:              yyerror_msg = yyerror_msg & ""
149:          End If
150:          errprefix(s)
151:
152:          If (Len(t)=0) Then
153:              yyerror_column = 0
154:          Else
155:              yyerror_msg = yyerror_msg & "actual: " & t
156:              yyerror_column = 8 + Len(t)
157:          End If
158:          yyerror_list = 0
159:      Else
160:          If (Len(t)<>0) Then
161:              If (yyerror_list=0) Then
162:                  If ( ( yyerror_column + Len(t) +
                           Len(expecting) + 1 ) < 78 ) Then
163:                      yyerror_msg = yyerror_msg & " " &
                                        expecting & t
164:                      yyerror_column = yyerror_column +
                                           Len(expecting) +
                                           Len(t) + 1
165:                  Else
166:                      yyerror_msg = yyerror_msg &
                                        Chr(13) & Chr(10)
167:                      yyerror_msg = yyerror_msg &
                                        expecting & t
168:                      yyerror_column = Len(expecting) +
                                           Len(t) - 1
169:                  End If
170:              Else
171:                  If ( yyerror_column + Len(t) < 78) Then
172:                      yyerror_msg = yyerror_msg & ", " & t
```

```
173:                        yyerror_column = yyerror_column +
                                              Len(t) + 2
174:                    Else
175:                        yyerror_msg = yyerror_msg & "," &
                                          Chr(13) & Chr(10)
176:                        yyerror_msg = yyerror_msg&"     "&t
177:                        yyerror_column = 4 + Len(t)
178:                    End If
179:                End If
180:                yyerror_list = yyerror_list + 1
181:            Else
182:                yyerror_msg = yyerror_msg & Chr(13) & Chr(10)
183:                yyerror_column = 0
184:                yyerror_list = 0
185:            End If
186:        End If
187:    End Sub
188:
189:
190:
```

The error reporting sub, **yyerror()** spanning from **lines 140 through 190**, 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.

The following is the listing of the **PCYACC** GDF for the Simple Calculator example. For reference, line numbers are added to the statements (line numbers should NOT be included in the user's GDF).

```
001:  %{
002:  Dim result
003:  Dim index, tmp(8191), tmp1(8191), tmp2(8191), num
004:  %}
005:
006:  %token          NUMBER
007:  %token          IDENTIFIER
008:  %token          SQRT
009:  %token          Shift
010:
011:  %left           '-' '+'
012:  %left           '*' '/'
013:  %left           '(' ')'
014:  %left           '='
015:  %left           Shift
016:  %nonassoc       UMINUS
017:
```

Lines 001 through 016 form the so-called declaration section, where token symbols, operator precedences, etc, are declared.

Lines 001 through 004 are the user declaration section, where some global variables used by the user actions are declared.

Lines 006 through 009 declare several tokens, which cannot be used as the left-hand side of grammar rules.

Lines 011 through 016 define the associativity of the arithmetic operators involved. All the operators are declared to be left associative (i.e., if the statement is a+b+c, the a+b will be calculated first). These statements also convey the following information: addition (+) and subtraction (-) have the same precedence; multiplication (*) and division (/) have the same precedence and it is higher than addition or subtraction; the unary operator, namely the negation sign, has the highest precedence.

```
018:  %start list
019:
020:  %%
```

start on **line 018** defines a start symbol. Rules with the start symbol on the LHS are called start rules.

The delimiter **%%** on **line 020** marks the beginning of the rule section.

Lines 022 through 133 form the grammar rule section, which is the main body of a GDF.

```
021:
022:  list
023:    : expression
024:      {
025:          For index=0 To UBound($$)
026:              tmp1(index) = $1(index)
027:          Next
028:          result = array_to_double(tmp1)
029:      }
030:    ;
031:
```

Lines 022 through 030 say that a list can be an expression. Everything enclosed in braces is called **actions**. Any function used here except from VBScript skeleton file has to be provided by the user themselves, which will reside in the user function section of a GDF.

Lines 032 through 133 contains the rules on how to form **expression**

nonterminal. Since **$x** is a variable which will communicate with the parser, in these rules the corresponding content associated with every terminal or nonterminal has to be assigned a value. Based on the fact that the information residing $$ and $x variables is stored in an array and VBScript language does not support **structur**e and **union** like C language, in the user action, we have to use the different data conversion functions to translate data between the different data types. This could make the user action code look more complicated.

```
032:  expression
033:     : expression '+' expression
034:       {
035:            For index=0 To UBound($$)
036:                 tmp1(index) = $1(index)
037:            Next
038:            For index=0 To UBound($$)
039:                 tmp2(index) = $3(index)
040:            Next
041:            For index=0 To UBound($$)
042:                 If ( Len( CStr( array_to_double(tmp1) +
                         array_to_double( tmp2 ) ) ) > index ) Then
043:                     $$(index)=Asc(Mid(CStr(array_to_double(tmp1)
                         + array_to_double(tmp2) ), index+1, 1))
044:                 Else
045:                      $$(index) = 0
046:                      Exit For
047:                 End If
048:            Next
049:       }
```

Lines 033 through 049 state that an expression can be an expression '+' an expression. The user action inside this rule will add values of $1 and $3, then assign the result to the LHS nonterminal – expression.

```
050:     | '(' expression ')'
051:       {
052:            For index=0 To UBound($$)
053:                 $$(index) = $2(index)
054:            Next
055:       }
```

Lines 050 through 055 state that an expression can be '(', followed by an expression, followed by ')'. The user action inside this rule will assign the value of $2 to the LHS nonterminal – expression.

```
056:     | expression '-' expression
057:       {
```

```
058:            For index=0 To UBound($$)
059:                tmp1(index) = $1(index)
060:            Next
061:            For index=0 To UBound($$)
062:                tmp2(index) = $3(index)
063:            Next
064:            For index=0 To UBound($$)
065:                If ( Len( CStr( array_to_double( tmp1 ) -
                       array_to_double( tmp2 ) ) ) > index ) Then
066:                  $$(index)=Asc(Mid(CStr(array_to_double(tmp1)
                           - array_to_double(tmp2)),index+1,1))
067:                Else
068:                    $$(index) = 0
069:                    Exit For
070:                End If
071:            Next
072:        }
```

Lines 056 through 072 state that an expression can be an expression '-' an expression. The user action inside this rule will subtract $3 value from $1 value, then assign the result to the LHS nonterminal – expression.

```
073:    | expression '*' expression
074:        {
075:            For index=0 To UBound($$)
076:                tmp1(index) = $1(index)
077:            Next
078:            For index=0 To UBound($$)
079:                tmp2(index) = $3(index)
080:            Next
081:            For index=0 To UBound($$)
082:                If ( Len( CStr( array_to_double( tmp1 ) *
                       array_to_double( tmp2 ) ) ) > index ) Then
083:                  $$(index)=Asc(Mid(CStr(array_to_double(tmp1)
                           *array_to_double(tmp2),index+1,1))
084:                Else
085:                    $$(index) = 0
086:                    Exit For
087:                End If
088:            Next
089:        }
```

Lines 073 through 089 state that an expression can be an expression '*' an expression. The user action inside this rule will multiply the values of $1 and $3, then assign the result to the LHS nonterminal – expression.

```
090:    | expression '/' expression
```

```
091:        {
092:            For index=0 To UBound($$)
093:                tmp1(index) = $1(index)
094:            Next
095:            For index=0 To UBound($$)
096:                tmp2(index) = $3(index)
097:            Next
098:
099:            If ( array_to_double(tmp2)=0 ) Then
100:                Call yyerror("divided by zero", "");
101:            Else
102:                num = array_to_double(tmp1) /
                            array_to_double(tmp2)
103:                For index=0 To UBound($$)
104:                    If (Len(CStr(num))>index) Then
105:                        $$(index) = Asc(Mid(CStr(num),
                                          index+1, 1))
106:                    Else
107:                        $$(index) = 0
108:                        Exit For
109:                    End If
110:                Next
111:            End If
112:        }
```

Lines 090 through 112 state that an expression can be an expression '/' an expression. The user action inside this rule will divide the values of $1 by the value of $3, then assign the result to the LHS nonterminal – expression.

```
113:      | '-' expression                %prec UMINUS
114:        {
115:            For index=0 To UBound($$)
116:                tmp1(index) = $2(index)
117:            Next
118:            For index=0 To UBound($$)
119:                If (Len( CStr( array_to_double( tmp1 ) *
                       (-1 ) ) ) > index ) Then
120:                  $$(index)=Asc(Mid(CStr(array_to_double(tmp1)
                                   *(-1)),index+1,1))
121:                Else
122:                    $$(index) = 0
123:                    Exit For
124:                End If
125:            Next
126:        }
```

Lines 113 through 126 state that an expression can be negation of an

expression. The user action inside this rule will negate the value of $2, then assign the result to the LHS nonterminal – expression.

```
127:    | NUMBER
128:      {
129:          For index=0 To UBound($$)
130:              $$(index) = $1(index)
131:          Next
132:      }
133:    ;
134:
```

Lines 127 through 134 state that an expression can be a NUMBER. The user action inside this rule will assign $1 value to the LHS nonterminal – expression.

```
135:  %%
```

The %% on **line 135** is delimiter separates the grammar rule section from the program section. Everything in the program section must be written in VBScript, and it will be copied to the output of **PCYACC**. Then it will be moved into the VBScript parser by **PCYTOOL**.

Lines 136 through 219 define several auxiliary procedures used to convert data from different data types and provide the user interface for input stream.

```
136:  ' main driver module
137:  Sub ParseFile_OnClick()
138:      str = CStr(InputString.Value)
139:
140:      yyparse
141:      If ( pcyyerrcnt=0 ) Then
142:          MsgBox "RESULT ========> " & CStr(result) &
                     Chr(10) & "There is no syntax error",
                     0, "Parsing Result"
143:      Else
144:          MsgBox yyerror_msg, 0, "PCYACC Error Message"
145:      End If
146:  End Sub
```

Lines 136 through 146 define an auxiliary procedure, which is used to invoke parser through function call **yyparse**. After parsing process is over, the result will be displayed in a message box.

```
147:  ' translate an array to an integer
148:  Function array_to_integer(arr) ' return a integer,
```

```
                arr is Array type with ASCII code element
149:  Dim i
150:      array_to_integer = 0
151:      For i=0 To UBound(arr)
152:          If ( arr(i)=0 ) Then
153:              Exit Function
154:          End If
155:          If ( arr(i)>=Asc("0") And arr(i)<=Asc("9") ) Then
156:              array_to_integer = (arr(i) - Asc("0")) +
                                       array_to_integer * 10
157:          End If
158:      Next
159:  End Function
```

Lines 147 through 159 define an auxiliary procedure, which translate an array to an integer. This procedure is very important for retrieving information from VBScript lexer and parser since **yylval** or **yyval** variable is an array type.

```
160:  ' translate an array to a double
161:  Function array_to_double(arr) ' return a double, arr
                     is Array type with ASCII code element
162:  Dim i, j, tmp, k, E_flag, sign_flag, e_value
163:      E_flag = 0
164:      sign_flag = 1   ' default to positive
165:      e_value = 0
166:
167:      j = 0
168:      array_to_double = 0
169:      For i=0 To UBound(arr)
170:          tmp = 1
171:          If ( arr(i)=0 ) Then
172:              Exit For
173:          End If
174:          If ( (arr(i)>=Asc("0") And arr(i)<=Asc("9"))
                 Or arr(i)=Asc(".") or arr(i)=Asc("E") or
                 arr(i)=Asc("+") or arr(i)=Asc("-") ) Then
175:              If ( arr(i)>=Asc("0") And
                    arr(i)<=Asc("9") And E_flag=0 ) Then
176:                  If ( j>0 ) Then
177:                      j = j + 1
178:                      For k=1 to j-1
179:                          tmp = 10 * tmp
180:                      Next
181:
182:                      array_to_double = (arr(i)-Asc("0")) /
                              tmp + array_to_double
```

```
183:                    Else
184:                        array_to_double = (arr(i)-Asc("0")) +
                                array_to_double * 10
185:                    End If
186:                End If
187:                If ( arr(i)>=Asc("0") And
                       arr(i)<=Asc("9") And E_flag=1 ) Then
188:                    e_value = e_value * 10 + arr(i)-Asc("0")
189:                End If
190:                If ( arr(i)=Asc(".") ) Then
191:                    j = 1
192:                End If
193:                If ( arr(i)=Asc("E") ) Then
194:                    E_flag = 1
195:                End If
196:                If ( arr(i)=Asc("+") ) Then
197:                    sign_flag = 1   ' positive
198:                End If
199:                If ( arr(i)=Asc("-") ) Then
200:                    sign_flag = 2   ' negative
201:                End If
202:            End If
203:        Next
204:
205:        ' handle Scientific Expression
206:        If ( E_flag=1 ) Then
207:            tmp = 1
208:            For k=1 to e_value
209:                tmp = tmp * 10
210:            Next
211:            If ( sign_flag=1 ) Then ' positive
212:                array_to_double = array_to_double * tmp
213:            Else
214:                array_to_double = array_to_double / tmp
215:            End If
216:        End If
217:
218:    End Function
219:
```

Lines 160 through 219 define an auxiliary procedure, which translate an array to a double. This procedure adds some statements to handle scientific number expression, **Abraxas Software** highly encourages the user to improve this part.

Generally, all the action code should be written in VBScript except that you can still use $x variable to refer the rule content. Our **PCYTOOL** will

translate $x variables to the corresponding variables in VBScript. All the code in the user function section will be put inside the VBScript parser. In order to make your parser runnable, you have to make sure that all the code you have written will work with the VBScript parser created by our **PCYTOOL**.

XII. Pascal Parser and Lexer

1. Introduction

Pascal is a general-purpose, high level programming language. It is established as one of the foremost high-level languages; whether the application is education or professional programming.

There are two separate phases in generating a Pascal Parser. The following diagram shows the phases:

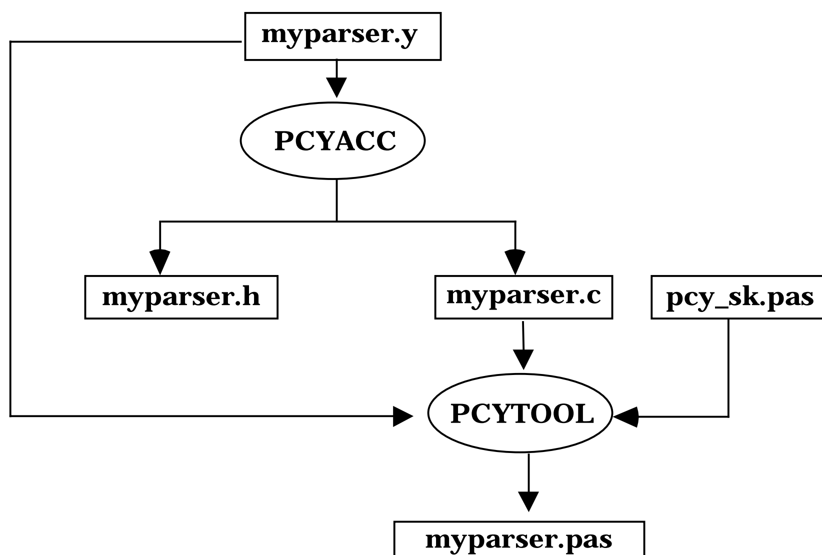

Figure 12-1. Diagram of Pascal parser generated by PCYTOOL

In the first phase, based on the availability of a grammar description file, simply invoke **PCYACC** tool on the command line to create a C parser. Once the C parser is generated by **PCYACC**, a utility program named **PCYTOOL** is needed for generating a Pascal parser in the second phase, in which myparser.pas represents a Pascal parser generated by **PCYTOOL**.

There are two separate phases for generating a Pascal lexer. The following diagram shows the phases:

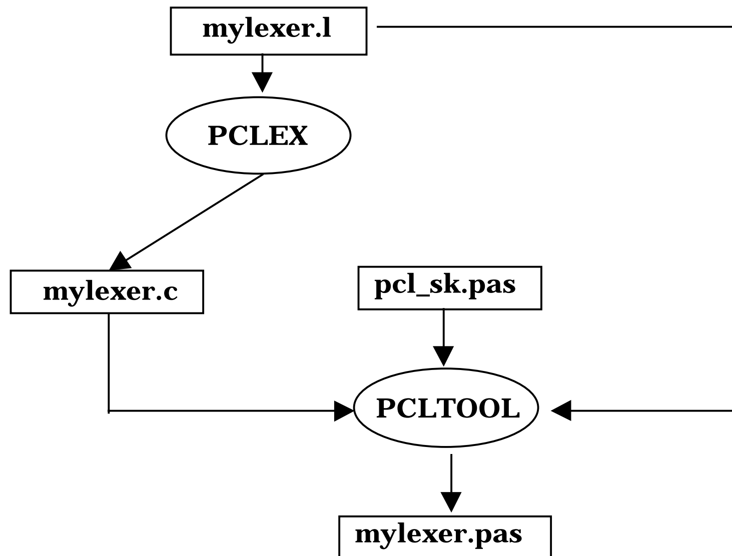

Figure 12-2. Diagram of Pascal lexer generated by PCLTOOL

In the first phase, based on the availability of a scanner description file, simply invoke **PCLEX** tool on the command line to create a C lexer. Once the C lexer is generated by **PCLEX**, a utility program named **PCLTOOL** is needed for generating a Pascal lexer in the second phase.

2. Pascal Library

a. Pascal Lexer

The Constant Declaration Part:

Constant declaration part declares constants within the block containing the declaration.

```
001:    const
002:        YYERRCODE = 256;
003:        YY_END_TOK = 0;
004:        YY_NEW_FILE = -1;
005:        YY_DO_DEFAULT = -2;
006:        BUFSIZ = 4096;
007:        F_BUFSIZ = 4096;
008:        YY_BUF_SIZE = 8192;
```

```
009:         YY_BUF_MAX = 8191;
010:         YY_MAX_LINE = 4096;
011:         YY_BUF_LIM = 4095;
012:         YY_NULL = 0;
```

Pascal language does not support macro definition. However, in a traditional C lexer generated by PCLEX tool, there are a lot of macros involved, which provides a lot of convenience to the users. The previously listed constants like **F_BUFSIZ, YY_BUF_SIZE, YY_BUF_MAX, YY_MAX_LINE** and **YY_BUF_LIM**, have been defined based on **BUFSIZ** with a default value of 4096. For details about how to assign the values of these constants, please refer to a C lexer source code generated by **PCLEX**.

The Type Declaration Part:

Programs, procedures, functions, and methods have a type declaration part to declare types.

```
013:     type
014:         charstr = Array[0..YY_BUF_SIZE] of Char;
```

The Var Declaration Part:

Every variable occurring in a program must be declared before use. The declaration must textually precede any use of the variable. A variable declaration part consists of the reserved word **var** followed by one or more identifier(s), separated by commas, each followed by a colon and a **type**.

```
015:
016:
017:     var
018:             YYlval : yystype;
019:             YYval : yystype;
020:             yylineno : Integer;
021:             filvar:Text;
022:             input_file_name:String[64];
023:             yy_start:Integer;
024:             yy_b_buf_p:Integer;
025:             yy_c_buf_p:Integer;
026:             yy_e_buf_p:Integer;
027:             yy_saw_eof:Integer;
028:             yy_init:Integer;
029:             yy_ch_buf:charstr;
030:             yy_st_buf:Array[0..YY_BUF_SIZE-1] of Integer;
031:             yy_hold_char:Char;
032:             yytext:Array[0..BUFSIZ-1] of Char;
033:             yyErrorCount : Integer;
```

```
034:            yyErrSrc : String;
035:            yyleng:Integer;
036:            yy_lp:Integer;
037:            yy_curst:Integer;
038:            reject_flag:Integer;
039:            list: Integer;
040:            column: Integer;
```

Pascal is highly typed language. Variable assignments must meet strict assignment compatibility rules.

The variables declared in var declaration part are originally defined as globals in a C lexer since all these variables are used for the lexer.

The Implementation Part:

```
set_input_file(name:String);
```

Allow the lexer to switch scanning from one input file to another. This enables a scanner to be interrupted to process another input instead of the current one.

```
yylex:Integer;
```

Separate an input character stream into tokens following the patterns specified in the lex (*.l) file. This function varies for each different lexical analyzer because the embedded user actions defined in the lex (*.l) file are different.

```
get_yylineno:Integer;
```

This function returns the current line number that is very useful in error reporting.

```
input:Integer;
```

This function returns the next character from an input.

```
unput(c:Char);
```

This procedure puts a character back in the logical input stream.

```
YY_DEFAULT_ACTION;
```

This procedure is defined as a macro in a C lexer. It will emit information of **yytext** buffer to the standard output.

```
YY_INPUT(var buf:charstr; index:Integer;
```

```
max_size:Integer):Integer;
```

This function deals with the storage of an input stream into a lexer. It gets an input from the input stream and stores it into a buffer, which the user can specify as the function call parameter.

```
YY_OUTPUT(c:Char);
```

This procedure outputs one character to the standard output.

```
YY_FATAL_ERROR(s:String);
```

This procedure prints out the string specified in as the function call parameter to the standard error.

```
yywrap:Boolean;
```

This function simply returns boolean value **true**.

```
yyless(n:Integer);
```

This procedure tells lex to “push back” part of the token that was just read. The argument to **yyless()** is the number of token characters to push back into the input stream.

```
YY_INIT_PROC;
```

This procedure is used to initialize the scanner’s state.

```
YY_LENG:Integer;
```

This function returns the length for the text of the token stored in **yytext**.

```
YY_DO_BEFORE_SCAN;
```

This procedure puts the character that the scanner holds at the end of **yytext**.

```
YY_DO_BEFORE_ACTION;
```

This procedure does some preparation job before the scanner takes actions.

```
REJECT(yy_full_match:Integer);
```

This procedure puts back the text matched by the pattern and finds the next best match for it.

```
yyerror(s:String; t:String);
```

This procedure simply calls **errprefix** to finish error reporting by reformatting the error message display. The two input strings can represent two different error messages to be displayed.

```
errprefix(msg:String);
```

This procedure takes a syntax error message as an input and displays it according to the current token information. The current token information including the line number, the character position number, etc, is provided by the lexer.

b. Pascal Parser

The Constant Declaration Part:

```
001:    const
002:        YYERRCODE = 256;
003:        YYMAXDEPTH = 200;
004:        YYREDMAX = 1000;
005:        PCYYFLAG = -4096;
006:        WAS0ERR = 0;
007:        WAS1ERR = 1;
008:        WAS2ERR = 2;
009:        WAS3ERR = 3;
```

These variables are defined as globals in a C parser. In Pascal, they are defined as constants.

The Type Declaration Part:

```
010:    type
011:        valuestack = Array [0..YYMAXDEPTH-1] of yystype;
```

The reserved word **type** heads the type declaration part, and it is followed by one or more type assignments separated by semicolons. Each type assignment consists of a type identifier followed by an equal sign and a type. Here, we define a **valuestack** type.

The Var Declaration Part:

```
012:
013:
014:    var
015:            yyOutFile : File of Char;
016:            yyErrFile : File of Char;
017:            yyErrorFlag : Integer;
018:            yyToken : Integer;
```

```
019:            yyValueStack : ^valuestack;
020:            redSequence : ^Integer;
021:            redCount : Integer;
```

The variables declared in var declaration part are originally defined as globals in a C parser since all these variables are used for the parser.

The Implementation Part:

```
yyParse: Integer;
```

This function is the entry point to a yacc-generated parser. When your program call function **yyParse()**, the parser attempts to parse an input stream. The parser returns a value of zero if the parse succeeds and non-zero if not.

```
yyerrok;
```

This procedure is implemented to replace a macro in a C parser. It tells the parser to return to the normal state, which can avoid the problem of multiple error messages resulting from a single mistake as the parser gets resynchronized.

XIII. Basic Parser and Lexer

1. Introduction

Visual basic, as the fastest and easiest way to create applications, provides the user with a complete set of tools to simplify rapid application development. The BASIC refers to “Beginners All-Purpose Symbolic Instruction Code” language. Visual Basic has evolved from the original BASIC language and adds in several keywords, statements and functions to deal with Window’s graphical user interface. Since VBScript for Internet programming is a subset of the Visual Basic language, so **Abraxas Software** will provide identical VBasic parser and lexer to VBScript except some implementation details.

The Code in VBasic is stored in modules. Each module can contain:

• Declarations. You can place constant, type, variable, and dynamic-link library (DLL) procedure declarations at the module level of form, class or standard modules.

• Procedures. A Sub, Function, or Property procedure contains pieces of code that can be executed as a unit.

Modules (.bas file name extension) are containers for procedures and declarations commonly accessed by other modules within the application. They can contain global (available to the whole application) or module-level declarations of variables, constants, types, external procedures, and global procedures.

There are two separate phases for generating a VBasic Parser. The following diagram shows the phase:

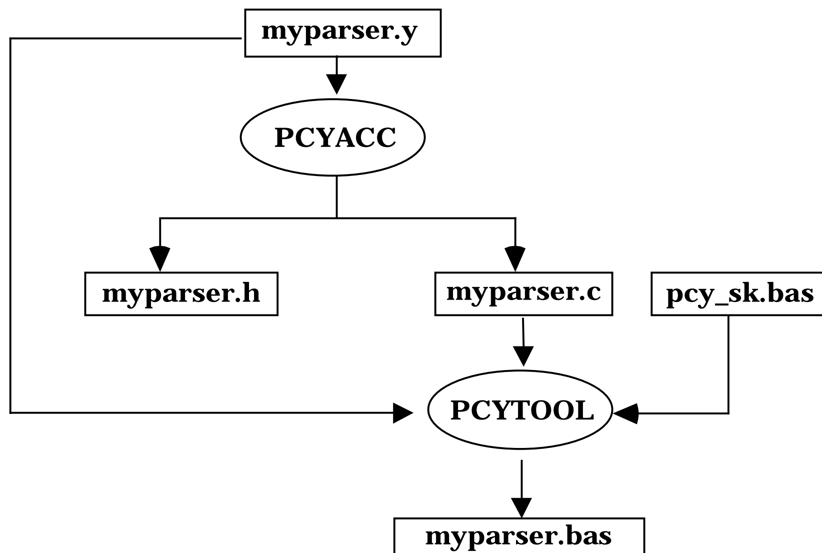

Figure 13-1. Diagram of VBasic parser generated by PCYTOOL

In the first phase, based on the availability of a grammar description file, simply invoke **PCYACC** tool on the command line to create a C parser. Once the C parser is generated by **PCYACC**, a utility program named **PCYTOOL** is needed for generating a VBasic parser in the second phase, in which myparser.bas is final VBasic parser.

There are two separate phases for generating a VBasic Lexer. The following diagram shows the phase:

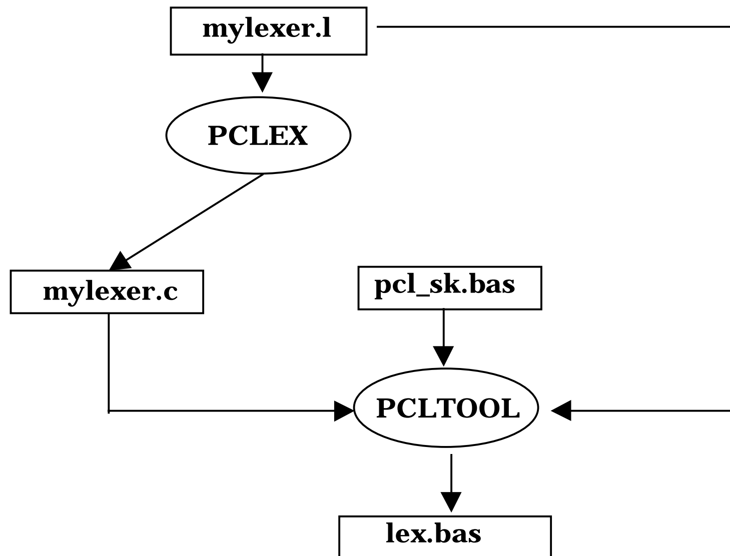

Figure 13-2. Diagram of VBasic lexer generated by PCLTOOL

In the first phase, based on the availability of a scanner description file, simply invoke **PCLEX** tool on the command line to create a C lexer. Once the C lexer is generated by **PCLEX**, a utility program named **PCLTOOL** is needed for generating a VBasic lexer in the second phase.

2. Structure of VBasic Parser and Lexer

The VBasic code of a lexical analyzer generated by **PCLTOOL** has the following layout,

1. Code from the section 1 of .l file
2. Data tables
3. Module-level variables
4. Auxilary procedures
5. Function yylex()
6. Code from the section 3 of .l

Figure 13-3. Structure of VBasic Code Generated by PCLTOOL

1). Code segment copied directly from the declaration section of lexer file: This part contains the declarations of variables and functions to be used in embedded actions. It varies with different lexical analyzers and it is optional. The code in this section should be written in VBasic language.

2). Data tables: This part consists of the data tables for driving the **Deterministic Finite Automaton** (DFA) simulator. They are different for different lexical analyzers.

3). Module-level variables: This part consists of the variables representing the input stream buffer and pointers that indicate the status of the input being scanned. They should be almost the same for different lexical analyzers.

4). Auxiliary procedures: This part defines the procedures that are called by the lexer function **yylex()**.

5). Function yylex(): This part defines the function **yylex()**.

6). Code segment copied directly from the function section of the lexer file: This part contains the possible function definitions by the user. It is also optional. The User is responsible for writing the valid VBasic code.

The layout of generated VBasic YACC code is shown below.

Code copied directly from the section 1 of .y file (if there is any)
Data tables
Constants representing tokens
Code copied directly from the section 3 of .y file (if there is any)
Auxiliary Yacc procedures
Function yyparse()

Figure 13-4 Layout of Generated VBasic YACC code

1). Code copied directly from the section 1 of .y file: This part contains the declarations of variables and functions to be used in embedded actions. It varies with different parsers and it is optional. The code in this section should be written in VBasic language.

2). Data tables: This part consists of the parsing tables for driving yacc machine. They are different for different syntax analyzers.

3). Constants representing tokens: This part consists of all tokens defined in grammar file.

4). Code copied directly from the section 3 of .y file: This part contains the possible function definitions by user. It is also optional. User is responsible for writing valid VBasic code.

5). Auxiliary yacc procedures: This part defines the procedures that are called by **yyparse** function.

6). Function yyparse(): This part contains **yyparse()** function definition.

a. VBasic Lex Modules

All the auxiliary procedures related to lexer are listed below:

```
Sub YY_DO_BEFORE_ACTION()
```

This procedure does some preparation job before scanner takes actions.

```
Sub YY_DEFAULT_ACTION()
```

This procedure is defined as a macro in a C lexer. It will emit information of **yytext** buffer to the output the user defined.

```
Sub YY_DO_BEFORE_SCAN()
```

This procedure puts the character that the scanner holds at the end of **yytext**.

```
Function yywrap
```

This procedure simply returns 1.

```
Sub YY_FATAL_ERROR(s)
```

This procedure prints out the string specified in as the function call parameter to the output the user defines.

```
Sub YY_INIT_PROC()
```

This procedure is used to initialize the scanner's state.

```
Function YY_INPUT(buf, index, maxsize)
```

This procedure deals with storage of input stream into a lexer. It gets input from the input stream and stores it into a buffer, which the user can specify as the function call parameter.

```
Sub YY_OUTPUT(c)
```

This procedure outputs one character to the output the user defines.

```
Function input()
```

This procedure returns the next character from input stream.

```
Sub unput(c)
```

This procedure puts a character back to the logical input stream. The user can call several times in a row to put several characters back into the input stream.

```
Function yylex()
```

This procedure is used to start or resume scanning. It separates an input character stream into token following the patterns specified in lex (*.l) file.

b. VBasic Yacc Modules

All the auxiliary procedures related to parser are listed below:

```
Sub yyerrok()
```

This procedure is implemented to replace a macro in C parser. It tells the parser to return to the normal state, which can avoid the problem of multiple error messages resulting from a single mistake as the parser gets resynchronized.

```
Function yyparse()
```

This procedure is the entry point to a yacc-generated parser. When your program calls the member function **yyparse()**, the parser attempts to parse an input stream. The parser returns a value of zero if the parse succeeds and non-zero if not.

c. VBasic Error Report Modules

All the auxiliary procedures related to error reporting are listed below:

```
Sub set_input_file_name(fname)
```

This procedure is used to set **pcyyerrsrc** so that error report could show which input source file the parser is processing.

```
Sub errprefix(msg)
```

This procedure takes a syntax error message as an input and displays it according to the current token information. The current token information including the line number, the character position number, etc, is provided by the lexer.

```
Sub yyerror(s, t)
```

This procedure simply calls **errprefix** to finish error reporting by reformatting the error message display. The two input strings can represent two different error messages to be displayed.

```
Function yydisplay(token)
```

This procedure displays token context based on the input token value. Normally, this function is provided by the user.

```
Sub yyskiptoken()
```

This procedure makes file indicator forward one unit to skip the current token as if the current token being processed by the lexer does not exist in the input stream.

```
Sub yyskipsymbol()
```

This procedure simply skips the current symbol in the sentential form if it is a terminal and keeps the current token unchanged.

```
Sub yyreplacetoken(token)
```

This procedure takes a replacing token as an input and replaces the token that will be processed by the lexer.

```
Function yymatchtoken(token)
```

This procedure takes a matching token as an input parameter. It will return 1 if the current token matched the actual input token value, otherwise returns 0 instead.

```
Sub yyinserttoken(token)
```

This procedure takes a token to be inserted as an input parameter. Its purpose is to insert a token in front of the one that will be processed by the lexer.

XIV DESIGN REQUIREMENT FOR YACC

1. Objective

• Extend to support generation of parser in multiple programming languages.

• Extend to support generation of multiple parsers in the same program.

2. Scope

• Support generation of C parser.

• Support generation of C++ parser.

• Support generation of JAVA parser.

• Support generation of Borland Delphi parser.

• Support generation of PASCAL parser.

• Support generation of Visual Basic Script parser.

• Support generation of BASIC parser.

3. Command Line Options

To use **PCYTOOL**, there are two procedures to follow:

A. Create a C parser by using **PCYACC**.

B. Create a target language parser other than C by using **PCYTOOL**.

The diagram is shown as below:

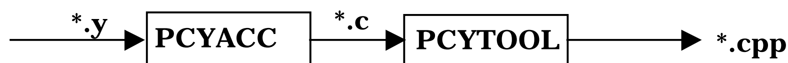

Figure 14-1. Diagram for Creating C++ Parser by Using PCYTOOL

A. In the first phase, apply the following on the command line.

```
pcyacc [options] <gdf_name>
```

<gdf_name> is the grammar description file name.

Options supported by **PCYACC** are listed as follows.

-c	Generate yytab.c .
-C<fn>	Generate the output source file <fn>.
-d	Generate token definition file yytab.h.
-D<hf>	Generate token definition file <hf>. If <hf> is not specified the filename defaults to <gdf_base>.h.
-h	Print a help screen.
-n	Disable the #line directive.
-p<pf>	Override default skeleton file with a user-provided parser skeleton file <pf>.
-P<pf>	Same as -p<pf>.
-r	Report progress during execution.
-R	Same as [-r].
-s	Use short integer internal arrays instead of long integer array.
-S	Syntax check only.
-t	Build a parse tree file "yy.ast".
-T<tf>	Build a parse tree file <tf>.ast.
-v	Produce a textual parsing table "yy.lrt".
-V<vf>	Produce a textual parsing table <vf>.lrt.

B. In the second phase, apply the following on the command line.

```
pcytool [options] yacc.y yacc.c
```

"yacc.y" is the grammar description file.

“yacc.c” is the actual parser code in C.

Options supported by **PCYTOOL** are as follows.

-D<fn> Override global definition and macro definition file with <fn> (The default file name is yypcy.h).

-K<n> Identify the dialect of parser assumed for the source files. A digit should follow immediately, corresponding to the dialect. The dialects of parser that are supported include:

0=> C

1 => C++

2 => JAVA

3 => DELPHI

4 => PASCAL

5.=> VISUAL BASIC SCRIPT

6 => BASIC

- THE DEFAULT IS K1 (C++)*

-L<fn> Append lexer file with file name <fn>.

-N” “ Change prefix class name ABX.

-O<fn> Override the default parser output file name with <fn> (the default name is the basename of C parser plus the program extension name which depends on “K” option).

-P<pf> Override internal skeleton with file <pf> for parser.

-T” “ Change default table type for C parser generated by PCYACC.

All these options are **case-insensitive**.

Files exist before running **PCYTOOL** for default C++ parser:

pcy_sk.hpp	Parser class definition.
pcy_sk.cpp	Parser class member function definition.
yacc.c	Syntactic parser generated by **PCYACC**.

Example command line to generate a C++ parser:

```
pcytool yacc.y  yacc.c
```

Files generated after running the command line:

yypcy.h	Global definitions and macro definitions(Optional).
yacc.h	Token definition file for lexer.
yacc.cpp	Actual parser code.

These files combined with the lexer source code files and the source code files containing user functions form a complete set of source code for a user project.

XV DESIGN REQUIREMENT FOR LEX

1. Objective

- Extend to support generation of lexer in multiple programming languages.

- Extend to support generation of multiple lexers in the same program.

2. Scope

- Support generation of C lexer.

- Support generation of C++ lexer.

- Support generation of JAVA lexer.

- Support generation of Borland Delphi lexer.

- Support generation of PASCAL lexer.

- Support generation of Visual Basic Script lexer.

- Support generation of BASIC lexer.

Each parser has a set of tokens associated with it. Usually, each parser requires a different lexer to identify the tokens from input source files. Several parsers implementing different grammars on the same set of tokens may use the same lexer. In this case, the lexer SDF will only have to include one of the token definition header files generated from parsers with the same set of tokens.

3. Command Line Options

To use **PCLTOOL**, there are two procedures to follow:

A. Create a C lexer by using **PCLEX**.

B. Create a target language lexer other than C by using **PCLTOOL**.

The diagram is shown as below:

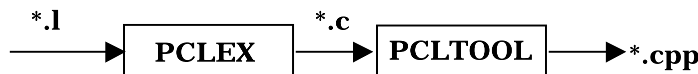

Figure 15-1. Diagram for Creating C++ Lexer by Using PCLTOOL

A. In the first phase, apply the following on the command line.

```
pclex [options] <sdf_name>
```

<sdf_name> is the scanner description file name.

Options supported by **PCLEX** are listed as follows.

-c	Override the default output **C** file name.
-C<fn>	Generate the output source file <fn>.
-h	Print a help screen.
-i	Generate a case-insensitive lexer.
-n	Disable the #line directive.
-p<pf>	Override default skeleton file with a user provided scanner skeleton file <pf>.
-s	Only check the input, if illegal token is found, exit.

These options are **case-insensitive** except for the [-c] and [-C] options.

B. In the second phase, apply the following on the command line.

```
pcltool [options] lex.l lex.c
```

“lex.l” is the scanner description file name.

“lex.c” is the actual lexer code in C.

All the options supported by **PCLTOOL** are as follows.

-D<fn>	Override global definition and macro definition file with <fn>

(the default file name is yypcl.h).

-K<n> Identify the dialect of lexer assumed for the source files. A digit should follow immediately, corresponding to the dialect. The dialects of lexer that are supported include:

0=> C

1 => C++

2 => JAVA

3 => DELPHI

4 => PASCAL

5.=> VISUAL BASIC SCRIPT

6 => BASIC

- THE DEFAULT IS K1 (C++)*

-N" " Change prefix class name ABX.

-O<fn> Override the default lexer output file name with <fn> (the default name is the basename of C lexer plus program extension name which depends on "K" option).

-P<pf> Override internal skeleton with file <pf> for lexer.

-Y<fn> Append parser file with file name <fn>.

All these options are **case-insensitive**.

Files exist before running **PCLTOOL** for default C++ lexer:

pcl_sk.hpp	Lexer class definition.
pcl_sk.cpp	Lexer class member function definition.
lex.c	Lexer generated by **PCLEX**.

Example command line to generate a C++ lexer:

```
pcltool lex.l lex.c
```

Files generated after running the command line:

yypcl.h	Global definitions and macro definitions (Optional).
lex.cpp	Actual lexer code.

This file combined with the parser source code files and the source code file containing user functions form a complete set of source code for a user project.

APPENDIX I. HOW TO CREATE C PARSER AND LEXER

1. Command Line Format for PCYACC

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

```
PCYACC [options] <gdf_name>
```

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

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

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

2. Command Line Options for PCYACC

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

-c	This option overrides default C file name. Instead of using the basename of the grammar description file plus the ".c" extension, it uses "yytab.c". This option is provided to maintain compatibility with earlier versions.
-C<cf>	Like -c, this option overrides default C file name, but uses the name provided by the user, <cf>.
-d	This option tells PCYACC to produce a C header file, using the default file name "yytab.h", in addition to the C code file. This header file is used primarily by your lexical analysis routine

yylex(). The definitions generated by PCYACC are used globally at parse time unless your yylex() routine is local to your grammar. PCYACC basically enumerates all of the tokens declared in the grammar, and these enumerated values are used as messages between yyparse() and yylex().

-D<hf>	Like -d, this option produces a C header file, but with a different file name convention. If no <hf> is provided, PCYACC will use the basename of the grammar description file with an extension ".h"; otherwise <hf> will be used instead.
-h	Print a help screen.
-n	Disable #line numbers from the .C output of PCYACC. This option is quite useful if you are trying to use a source code debugger. In normal operation the output .C file uses #line to make the output relative to the original .y file, this normally causes source code debuggers like CodeView to generate strange results.
-p<pf>	Use the user provided parser skeleton contained in <pf> file instead of the system default (internal skeleton). A sample parser skeleton is supplied in the \src directory of the PROGRAM diskette. (yaccpar.c). The external parser skeleton is a commonly used to support multiple parsers.
-P<pf>	Same as -p<pf>.
-r	Report progress during execution. This is a good idea for huge grammars that take seemingly forever to compile.
-R	Report progress during execution.
-s	This option instructs PCYACC produces short integer internal arrays for the parser. The default type for the internal arrays is long integer.
-S	This option overrides PCYACC's default action. Instead of processing the grammar description file, it quits after the syntax analysis phase. This option is useful for doing syntax debugging on large grammar description files, especially when coupled with an extensible text editor.
-t	This option tells PCYACC to construct the parser in such a way that it will build a parse tree for the program being processed. The parse tree, by default, is saved to the file "yy.ast". (not

	compatible with the -p switch, requires internal skeleton parser). The parse tree is not actually generated until the parser is executed.
-T<tf>	Same as option -t, except with different file name conventions. If <tf> is not provided, the parse tree is saved to the file named by the basename of the grammar description file with an ".ast" extension; otherwise, it is saved to <tf>.
-v	This option produces a textual parsing table, in addition to the C parser, using the default file name "yy.lrt". The parsing table is a required tool in debugging PCYACC grammars.
-V<vf>	Same as option -v, except that the parsing table is saved to either <vf> or a file named by the basename of the grammar description file and the extension ".lrt".

3. Command Line Format for PCLEX

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

```
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.

4. Command Line Options for PCLEX

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, but uses the file name provided by the user in <cf>.
-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 .

-s This option suppress the default rule (the 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.

5. Default Skeleton File

The default internal lexer skeleton file is the same as "**pcl_sk.c**" included in PCYACC OO SDK.

The default internal parser skeleton file is the same as "**pcy_sk.c**" included in PCYACC OO SDK.

APPENDIX II. HOW TO CREATE C++ PARSER AND LEXER

To create C++ parser and lexer is based on the assumption that the user has already gotten C parser and lexer generated by PCYACC and PCLEX respectively (APPENDIX I has a detailed description on how to use PCYACC and PCLEX to create C parser and lexer.). A utility program is involved to translate C code into C++ code with availability of C++ skeleton files.

1. Command Line Format for PCYTOOL

```
pcytool [options]  yacc.y yacc.c
```

Where “yacc.y” is the grammar description file and “yacc.c” is the actual parser code in C. If PCYTOOL is invoked without a grammar description file or actual C parser file, it will display a short message advising you of the correct command line format.

2. Command Line Options for PCYTOOL

Options supported by PCYTOOL are as follows.

-D<fn>	Override global definition and macro definition file with <fn> (the default file name is yypcy.h).
-K1	K option can identify the dialect of parser assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k1 will tell PCYTOOL the target parser will be created in C++ code.
-N” “	Change prefix class name ABX.
-O<fn>	Override the default parser output file name with <fn> (the default name is the basename of C parser plus the “.CPP” extension if no K option is on or K1 is set).
-P<pf>	Override internal skeleton with file <pf> for parser.

All these options are **case-insensitive**.

3. Command Line Format for PCLTOOL

```
pcltool [options] lex.l lex.c
```

Where “lex.l” is scanner description file, “lex.c” is the actual lexer code in C and [options] represents zero or more command line options. If **PCLTOOL** is invoked with no arguments, it outputs a short message advising you of the correct command line format.

4. Command Line Options for PCLTOOL

All the options supported by PCLTOOL are as follows.

-D<fn>	Override global definition and macro definition file with <fn> (the default file name is yypcl.h).
-K1	K option can identify the dialect of lexer assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k1 will tell PCLTOOL the target lexer will be created in C++ code.
-N” “	Change prefix class name ABX.
-O<fn>	Override the default lexer output file name with <fn> (the default name is the basename of C lexer plus the “.cpp” extension if no K option is on or K1 option is set).
-P<pf>	Override internal skeleton with file <pf> for lexer.

All these options are **case-insensitive**.

5. Default Skeleton File

The default internal lexer skeleton file is the same as “**pcl_sk.cpp**” included in PCYACC OO SDK. The corresponding header file is “**pcl_sk.hpp**”.

The default internal parser skeleton file is the same as “**pcy_sk.cpp**” included in PCYACC OO SDK. The corresponding header file is “**pcy_sk.hpp**”.

APPENDIX III. HOW TO CREATE JAVA PARSER AND LEXER

To create JAVA parser and lexer is based on the assumption that user has already gotten C parser and lexer generated by PCYACC and PCLEX respectively (APPENDIX I has detailed description on how to use PCYACC and PCLEX to create C parser and lexer.). A utility program is involved to translate C code into JAVA code with availability of JAVA skeleton files.

1. Command Line Format for PCYTOOL

```
pcytool [options] yacc.y yacc.c
```

Where "yacc.y" is the grammar description file and "yacc.c" is the actual parser code in C. If PCYTOOL is invoked without a grammar description file or actual C parser, it will display a short message advising you of the correct command line format. Be sure you have to put "#define STAND_ALONE 1" in the user declaration section in *.y file when you expect a standalone Java parser.

2. Command Line Options for PCYTOOL

Options supported by PCYTOOL are as follows.

-K2	K option can identify the dialect of parser assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k2 will tell PCYTOOL the target parser will be created in JAVA code.
-O<fn>	Override the default parser output file name with <fn> (the default name is the basename of C parser plus the ".JAVA" extension if K2 option is on).
-P<pf>	Override internal skeleton with file <pf> for parser.

All these options are **case-insensitive**.

3. Command Line Format for PCLTOOL

If you expect a standalone Java lexer, the following command should be appeared on the PCLTOOL command line.

```
pcltool [options] lex.l lex.c
```

Where “lex.l” is scanner description file, “lex.c” is the actual lexer code in C and [options] represents zero or more command line options. If **PCLTOOL** is invoked with no arguments, it outputs a short message advising you of the correct command line format. Be sure if you expect standalone Java lexer, you have to put “#define STAND_ALONE 1” in user declaration section in *.l file.

If you expect a single application with both parser and lexer, the following command should be appeared on the PCLTOOL command line.

```
pcltool [options] yacc.h lex.l lex.c
```

Where “yacc.h” is token definition file generated by PCYACC(-D option), “lex.l” is scanner description file, “lex.c” is the actual lexer code in C and [options] represents zero or more command line options. If PCLTOOL is invoked with no arguments, it outputs a short message advising you of the correct command line format.

4. Command Line Options for PCLTOOL

All the options supported by PCLTOOL are as follows.

-K2 K option can identify the dialect of lexer assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k2 will tell PCLTOOL the target lexer will be created in JAVA code.

-O<fn> Override the default lexer output file name with <fn> (the default name is the basename of C lexer plus the “.java” extension if K2 option is on).

-P<pf> Override internal skeleton with file <pf> for lexer.

All these options are **case-insensitive**.

5. Default Skeleton File

The default internal lexer skeleton file is the same as “**pcl_sk.jav**” included in PCYACC OO SDK that should be kept in your current working directory.

The default internal parser skeleton file is the same as “**pcy_sk.jav**” included in PCYACC OO SDK that should be kept in your current working directory.

APPENDIX IV. HOW TO CREATE DELPHI PARSER AND LEXER

To create Delphi parser and lexer is based on the assumption that user has already gotten C parser and lexer generated by PCYACC and PCLEX respectively (APPENDIX I has detailed description on how to use PCYACC and PCLEX to create C parser and lexer.). A utility program is involved to translate C code into DELPHI code with internal skeleton or availability of user provided DELPHI skeleton files.

1. Command Line Format for PCYTOOL

```
pcytool [options] yacc.y yacc.c
```

Where “yacc.y” is grammar description file and “yacc.c” is the actual parser code in C. If PCYTOOL is invoked without a grammar description file, it will display a short message advising you of the correct command line format.

2. Command Line Options for PCYTOOL

Options supported by PCYTOOL are as follows.

-K3	K option can identify the dialect of parser assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k3 will tell PCYTOOL the target parser will be created in DELPHI code.
-O<fn>	Override the default parser output file name with <fn> (the default name is the basename of C parser plus the “.pas” extension if K3 option is on. The YACC unit program is named as “yacc.pas”).
-P<pf>	Override internal skeleton with file <pf> for parser.

All these options are **case-insensitive**.

3. Command Line Format for PCLTOOL

```
pcltool [options] lex.l lex.c
```

Where “lex.l” is scanner description file, “lex.c” is the actual lexer code in C and [options] represents zero or more command line options. If **PCLTOOL** is invoked with no arguments, it outputs a short message advising you of the correct command line format.

4. Command Line Options for PCLTOOL

All the options supported by PCLTOOL are as follows.

-K3 K option can identify the dialect of lexer assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k3 will tell PCLTOOL the target lexer will be created in DELPHI code.

-O<fn> Override the default lexer output file name with <fn> (the default name is the basename of C lexer plus the ".pas" extension if K3 option is on. The LEX unit program is named as "lex.pas").

-P<pf> Override internal skeleton with file <pf> for lexer.

All these options are **case-insensitive**.

5. Default Skeleton File

The default internal lexer skeleton file is the same as "**pcl_sk.dph**" included in PCYACC OO SDK, which should be kept in your current working directory.

The default internal parser skeleton file is the same as "**pcy_sk.dph**" included in PCYACC OO SDK, which should be kept in your current working directory.

APPENDIX V. HOW TO CREATE PASCAL PARSER AND LEXER

To create PASCAL parser and lexer is based on the assumption that the user has already gotten C parser and lexer generated by PCYACC and PCLEX respectively (APPENDIX I has detailed description on how to use PCYACC and PCLEX to create C parser and lexer.). A utility program is involved to translate C code into PASCAL code with availability of PASCAL skeleton files.

1. Command Line Format for PCYTOOL

```
pcytool [options] yacc.y yacc.c
```

Where “yacc.y” is grammar description file and “yacc.c” is the actual parser code in C. If PCYTOOL is invoked without a grammar description file, it will display a short message advising you of the correct command line format.

2. Command Line Options for PCYTOOL

Options supported by PCYTOOL are as follows.

-K4	K option can identify the dialect of parser assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k4 will tell PCYTOOL the target parser will be created in PASCAL code.
-O<fn>	Override the default parser output file name with <fn> (the default name is the basename of C parser plus the “.pas” extension if K4 option is on).
-P<pf>	Override internal skeleton with file <pf> for parser.

All these options are **case-insensitive**.

3. Command Line Format for PCLTOOL

```
pcltool [options] lex.l lex.c
```

Where “lex.l” is scanner description file, “lex.c” is the actual lexer code in C and [options] represents zero or more command line options. If **PCLTOOL** is invoked with no arguments, it outputs a short message advising you of the correct command line format.

4. Command Line Options for PCLTOOL

All the options supported by PCLTOOL are as follows.

-K4 K option can identify the dialect of lexer assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k4 will tell PCLTOOL the target lexer will be created in PASCAL code.

-O<fn> Override the default lexer output file name with <fn> (the default name is the basename of C lexer plus the ".pas" extension if K4 option is on).

-P<pf> Override internal skeleton with file <pf> for lexer.

All these options are **case-insensitive**.

5. Default Skeleton File

The default internal lexer skeleton file is the same as "**pcl_sk.pas**" included in PCYACC OO SDK, which should be kept in your current working directory.

The default internal parser skeleton file is the same as "**pcy_sk.pas**" included in PCYACC OO SDK, which should be kept in your current working directory.

APPENDIX VI. HOW TO CREATE VISUAL BASIC SCRIPT PARSER AND LEXER

To create VISUAL BASIC SCRIPT parser and lexer is based on the assumption that the user has already gotten C parser and lexer generated by PCYACC and PCLEX respectively (APPENDIX I has detailed description on how to use PCYACC and PCLEX to create C parser and lexer.). A utility program is involved to translate C code into VISUAL BASIC SCRIPT code with availability of VISUAL BASIC SCRIPT skeleton files.

1. Command Line Format for PCYTOOL

```
pcytool [options] yacc.y yacc.c
```

Where “yacc.y” is grammar description file and “yacc.c” is the actual parser code in C. If PCYTOOL is invoked without a grammar description file, it will display a short message advising you of the correct command line format.

2. Command Line Options for PCYTOOL

Options supported by PCYTOOL are as follows.

-K5	K option can identify the dialect of parser assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k5 will tell PCYTOOL the target parser will be created in VISUAL BASIC SCRIPT code.
-L<fn>	Append lexer file with file name <fn>.
-O<fn>	Override the default parser output file name with <fn> (the default name is the basename of C parser plus the “.bas” extension if K5 option is on).
-P<pf>	Override internal skeleton with file <pf> for parser.

All these options are **case-insensitive**.

3. Command Line Format for PCLTOOL

```
pcltool [options] lex.l lex.c
```

Where “lex.l” is scanner description file, “lex.c” is the actual lexer code in C and [options] represents zero or more command line options. If **PCLTOOL** is invoked with no arguments, it outputs a short message advising you of the correct command line format.

4. Command Line Options for PCLTOOL

All the options supported by PCLTOOL are as follows.

-K5 K option can identify the dialect of lexer assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k5 will tell PCLTOOL the target lexer will be created in VISUAL BASIC SCRIPT code.

-O<fn> Override the default lexer output file name with <fn> (the default name is the basename of C lexer plus the ".bas" extension if K5 option is on).

-P<pf> Override internal skeleton with file <pf> for lexer.

-Y<fn> Append parser file with file name <fn>.

All these options are **case-insensitive**.

5. Default Skeleton File

The default internal lexer skeleton file is the same as "**pcl_sk.vbs**" included in PCYACC OO SDK, which should be kept in your current working directory.

The default internal parser skeleton file is the same as "**pcy_sk.vbs**" included in PCYACC OO SDK, which should be kept in your current working directory.

APPENDIX VII. HOW TO CREATE BASIC PARSER AND LEXER

To create BASIC parser and lexer is based on the assumption that the user has already gotten C parser and lexer generated by PCYACC and PCLEX respectively (APPENDIX I has detailed description on how to use PCYACC and PCLEX to create C parser and lexer.). A utility program is involved to translate C code into BASIC code with availability of BASIC skeleton files.

1. Command Line Format for PCYTOOL

```
pcytool [options] yacc.y yacc.c
```

Where “yacc.y” is grammar description file and “yacc.c” is the actual parser code in C. If PCYTOOL is invoked without a grammar description file, it will display a short message advising you of the correct command line format.

2. Command Line Options for PCYTOOL

Options supported by PCYTOOL are as follows.

-O<fn>	Override the default parser output file name with <fn> (the default name is the basename of C parser plus the “.bas” extension if K6 option is on).
-K6	K option can identify the dialect of parser assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k6 will tell PCYTOOL the target parser will be created in BASIC code.
-P<pf>	Override internal skeleton with file <pf> for parser.

All these options are **case-insensitive**.

3. Command Line Format for PCLTOOL

```
pcltool [options] lex.l lex.c
```

Where “lex.l” is scanner description file, “lex.c” is the actual lexer code in C and [options] represents zero or more command line options. If **PCLTOOL** is invoked with no arguments, it outputs a short message advising you of the correct command line format.

4. Command Line Options for PCLTOOL

All the options supported by PCLTOOL are as follows.

-K6 K option can identify the dialect of lexer assumed for the source files. A digit should follow immediately, corresponding to the dialect. Option k6 will tell PCLTOOL the target lexer will be created in BASIC code.

-O<fn> Override the default lexer output file name with <fn> (the default name is the basename of C lexer plus the ".bas" extension if K6 option is on).

-P<pf> Override internal skeleton with file <pf> for lexer.

All these options are **case-insensitive**.

5. Default Skeleton File

The default internal lexer skeleton file is the same as "**pcl_sk.bas**" included in PCYACC OO SDK, which should be kept in your current working directory.

The default internal parser skeleton file is the same as "**pcy_sk.bas**" included in PCYACC OO SDK, which should be kept in your current working directory.

APPENDIX VIII. ERROR MESSAGES FOR PCYTOOL

Error Code: Error Message and Explanation

TY1000 *can not open source file*

Opening a source file has failed.

TY1001 *the input source file name *.* is not correct*

The input source file is not expected.

TY1002 *option -* usage is not correct*

Option is behind source files on the pcytool command line.

TY1003 *there is no number following the option*

Some options require corresponding number following like -k option.

TY1004 *option is not supported*

The option the user invokes on the command line is not supported by PCYTOOL.

TY1005 *there is no filename following the option*

There is no source file on the command line.

TY1006 *the C++ skeleton parser file name is not correct*

The extension of C++ skeleton parser is not ".cpp".

TY1007 *fseek failed*

fseek failure.

TY1008 *searching file provided by user failed*

PCYTOOL can not find the file provided by the user.

TY1009 *syntax in grammar file is not correct*

Syntax error is found in grammar description file

TY1010 ** is not supported*

Some information in grammar file can not be handled correctly by PCYTOOL.

TY1011 *End of file is reached*

End of file.

TY1012 *missing ‘ or “*

Missing ‘ or “ in grammar file.

TY1013 *bad comment syntax*

The syntax for comment is not correct.

TY1014 *syntax error in Grammar file*

PCYTOOL finds syntax error in Grammar file.

TY1015 *READ and WRITE are not allowed in the file*

The file can not be accessed.

TY1016 *no grammar file is provided*

User does not provide GDF on the pcytool command line.

APPENDIX IX. ERROR MESSAGES FOR PCLTOOL

Error Code: Error Message and Explanation

TL1000 *can not open source file*

Opening a source file has failed.

TL1001 *the input source file name *.* is not correct*

The input source file is not expected.

TL1002 *option -* usage is not correct*

Option is behind source files on the pcltool command line.

TL1003 *there is no number following the option*

Some options require corresponding number following like -k option.

TL1004 *option is not supported*

The option the user invokes on the command line is not supported by PCLTOOL.

TL1005 *there is no filename following the option*

There is no source file on the command line.

TL1006 *the C++ skeleton lexer file name is not correct*

The extension of C++ skeleton lexer is not ".cpp".

TL1007 *fseek failed*

fseek failure.

TL1008 *searching file provided by user failed*

PCLTOOL can not find the file provided by user.

TL1009 *syntax in scanner description file is not correct*

Syntax error is found in scanner description file

TL1010 ** is not supported*

Some information in scanner description file can not be handled correctly by PCLTOOL.

TL1011 *end of file is reached*

End of file.

TL1012 *missing ‘ or “*

Missing ‘ or “ in scanner description file.

TL1013 *bad comment syntax*

The syntax for comment is not correct.

TL1014 *syntax error in Scanner Description file*

PCLTOOL finds syntax error in Scanner Description file.

TL1015 *READ and WRITE are not allowed in the file*

The file can not be accessed.

TL1016 *no scanner description file is provided*

User does not provide SDF on the pcltool command line.

APPENDIX X. BIBLIOGRAPHY

Aho, A.V., Sethi, R., and Ullman, J.D. “Compilers Principles, Techniques, and Tools”, Addison Weslay, Reading, Massachusetts, 1985.

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

Allen, J. “NATURAL LANGUAGE UNDERSTANDING”, Second Edition, The Benjamin/Cummings Publishing Company, Inc., Reading, Massachusetts, 1995.

Appel, A. W. “Modern Compiler Implementation in Java”, First Edition, Cambridge University Press, 1997

Arnold, K. and Gosling, J. “The Java Programming Language ”, The Java Series, Addison-Wesley, Reading, Massachusetts, 1996.

Cline, M.P. and Lomow, G.A. “C++ FAQs Frequently Asked Questions”, Addison-Wesley, Reading, Massachusetts, 1994.

Coplien, J. “Advanced C++ Programming Styles and Idioms”, Addison-Wesley, Reading, Massachusetts, 1992.

Cornell, G. and Horstmann, C.S. “Core JAVA”, SunSoft Press, Prentice-Hall, Mountain View, CA, 1996.

Eckel, B. “THINKING in C++”, Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

Ellis, M.A. and Stroustrup, B “The Annotated C++ Reference Manual”, ANSI Base Document, Addison-Wesley, Reading, Massachusetts, 1990.

Flanagan, D. “Java in a Nutshell”, First Edition, O’Reilly & Associates, CA, 1996.

Friedl, J.E.F. “Mastering Regular Expressions”, A Nutshell Handbook, First Edition, O’Reilly & Associates, CA, 1997.

Gosling, J., Joy, B. and Steele, G. “The Java Language Specification”, The Java Series, Addison-Wesley, Reading, Massachusetts, 1996.

Grand, M. “JAVA Language Reference”, First Edition, O’Reilly & Associates, CA, 1997.

Grune, D. and Jacobs, C.J.H. “Parsing Techniques”, Ellis Horwood,

Chichester, England, 1990.

Holmes, J. “OBJECT-ORIENTED COMPILER CONSTRUCTION”, Prentice-Hall, Englewood Cliffs, New Jersey, 1995.

Linden, P.V.D. “Just JAVA”, ”, SunSoft Press, Prentice-Hall, Mountain View, CA, 1996.

Lippman, S.B. “C++ Primer”, Addison-Wesley, Reading, Massachusetts, 1989.

Lomax, P. “Learning VBSCRIPT”, O’Reilly & Associates, First Edition, 1997.

Mason, T., Brown D., and Levine, J. “Lex & Yacc”, O’Reilly & Associates, Second Edition, 1992.

Microsoft Corp, “Microsoft Visual Basic 5.0 Programmer’s Guide”, Microsoft Press, 1997.

Musciano, C. and Kennedy, B. “HTML The Definitive Guide”, O’Reilly & Associates, First Edition, 1996.

Pittman, T. and Peters, J. “The Art of Compiler Design THEORY AND PRACTICE”, Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

Simon, J., “VBScript SUPERBIBLE”, Waite Group Press, 1996.

Stroustrup, B. “The C++ Programming Language”, Second Edition, Addison-Wesley, Reading, Massachusetts, 1991.

Stroustrup, B. “The C++ Programming Language”, Third Edition, Addison-Wesley, Reading, Massachusetts, 1997.

Wilhelm, R. and Maurer, D. “COMPILER DESIGN”, Addison-Wesley, Reading, Massachusetts, 1995

Index

A

ABXError class
- constructor · 42
- destructor · 42
- members · 41
- methods · 42

ABXExprNode class
- constructor · 50
- definition · 50
- destructor · 50
- methods · 50

ABXExprNodeList class
- constructor · 51
- definition · 50
- destructor · 51
- methods · 51

ABXLeaf class
- constructor · 48
- definition · 48
- destructor · 48

ABXLeafList class
- constructor · 49
- definition · 48
- destructor · 49
- methods · 49

ABXLex class
- constructor · 14
- destructor · 14
- members · 13
- methods · 15

ABXParseTree class
- constructor · 51
- definition · 51
- destructor · 51
- methods · 52

ABXParseTreeNode class
- constructor · 48
- definition · 47
- destructor · 48

ABXSymbolTable class
- constructor · 34
- destructor · 34
- members · 33
- methods · 34

ABXYacc class
- constructor · 29
- destructor · 29
- members · 22
- methods · 29

addSymbol method
- of class ABXSymbolTable · 34
- of class JavaSymbolTable · 69

append_node method
- of class ABXExprNodeList · 51
- of class ABXLeafList · 49
- of class JavaExprNodeList · 66
- of class JavaLeafList · 65

B

BUFSIZ
- of class JavaLex · 57
- of DelphiLex unit · 80
- of Pascal Lexer · 115

C

C Lexer
- layout · 10

C Parser
- layout · 20

C++ Lexer
- code structure · 12, 16
- constructor · 14
- destructor · 14
- methods · 15
- methods to create · 8
- usage of yyLex method · 17

C++ Parser
- code structure · 22
- constructor · 22
- destructor · 23
- methods to create · 5, 21
- switch between two different lexers · 28

cleanScope method
- of class ABXSymbolTable · 34
- of class JavaSymbolTable · 69

constructor
- DelphiLex unit
  - of TLex object · 82
- DelphiYacc unit
  - of TYacc object · 85
- of class ABXError · 42
- of class ABXExprNode · 50
- of class ABXExprNodeList · 51
- of class ABXLeaf · 48
- of class ABXLeafList · 49
- of class ABXLex · 18

of class ABXParseTree · 51
of class ABXParseTreeNode · 48
of class ABXSymbolTable · 34
of class ABXYacc · 29
of class JavaError · 61
of class JavaExprNode · 65
of class JavaExprNodeList · 66
of class JavaLeaf · 64
of class JavaLeafList · 65
of class JavaLex · 57
of class JavaParseTree · 66
of class JavaParseTreeNode · 64
of class JavaSymbolTable · 68
of class JavaYacc · 60
create_leaf_list method
of class ABXLeafList · 49
of class JavaLeafList · 65
createscope method
of class JavaSymbolTable · 69
createScope method
of class ABXSymbolTable · 34

D

decorate_tree method
of class ABXParseTree · 52
of class JavaParseTree · 67
definition
of class ABXExprNode · 50
of class ABXExprNodeList · 50
of class ABXLeaf · 48
of class ABXLeafList · 48
of class ABXParseTree · 51
of class ABXParseTreeNode · 47
delete_node method
of class ABXLeafList · 49
of class JavaLeafList · 65
deleteSymbol method
of class ABXSymbolTable · 35
of class JavaSymbolTable · 69
Delphi Lexer
constant declaration · 79
procedures to create · 79
type declaration · 80
Delphi library
basic units · 77
Delphi Parser
constant declaration · 84
example · 86
procedures to create · 78
type declaration · 84
DelphiLex unit
TLex object
methods · 82
DelphiYacc unit
TYacc object
methods · 85
destructor
DelphiLex unit
of TLex object · 82
DelphiYacc unit
of TYacc object · 85
of class ABXError · 42
of class ABXExprNode · 50
of class ABXExprNodeList · 51
of class ABXLeaf · 48
of class ABXLeafList · 49
of class ABXLex · 18
of class ABXParseTree · 51
of class ABXParseTreeNode · 48
of class ABXSymbolTable · 34
of class ABXYacc · 29
Deterministic Finite Automaton · 11

E

error
categories · 37
recovery · 38
reporting · 37
errprefix method
DelphiLex unit
of TLex object · 84
errprefix procedure
of VBasic Error Report · 126
of VBScript Error Report · 96
execute_tree method
of class ABXParseTree · 52
of class JavaParseTree · 67

F

F_BUFSIZ
of class JavaLex · 57
of DelphiLex unit · 80
of Pascal Lexer · 115

G

get_attr method
of class ABXExprNode · 50
of class JavaExprNode · 65
get_BufferPtrC method
of class ABXLex · 18
get_yyBufferPtrC method
of class ABXLex · 15
get_yyerrcnt method
of class JavaYacc · 61
get_yyErrorCount method
DelphiYacc unit
of TYacc object · 85
of class ABXYacc · 29
get_yylineno method
DelphiLex unit

of TLex object · 82
of class JavaLex · 57
get_yyLineNo method
of class ABXLex · 15, 18
get_yytext method
of class JavaLex · 59
get_yyText method
of class ABXLex · 15, 18
getScope method
of class ABXSymbolTable · 35
of class JavaSymbolTable · 69
getSymbolAttribute method
of class ABXSymbolTable · 35
of class JavaSymbolTable · 69
getSymbolType method
of class ABXSymbolTable · 35
of class JavaSymbolTable · 69
getSymbolValue method
of class ABXSymbolTable · 35
of class JavaSymbolTable · 70

I

initSymbolTable method
of class ABXSymbolTable · 35
of class JavaSymbolTable · 70
input method
DelphiLex unit
of TLex object · 82
of class ABXLex · 15, 18
of class JavaLex · 58
input procedure
of VBasic Lex · 125
of VBScript Lex · 95
insertSymbol method
of class ABXSymbolTable · 35
of class JavaSymbolTable · 70
interaction
C++ parser and lexer · 23

J

Java class library
basic classes · 54
JavaError
constructor · 61
members · 61
methods · 61
JavaExprNode
constructor · 65
members · 65
methods · 65
JavaExprNodeList
constructor · 66
members · 66
methods · 66
JavaLeaf
constructor · 64
members · 64
methods · 64
JavaLeafList
constructor · 65
memebers · 64
methods · 64
JavaLex
constructor · 57
members · 56
methods · 57
JavaParseTree
constructor · 66
members · 66
methods · 66
JavaParseTreeNode
constructor · 64
members · 63
methods · 63
JavaSymbolTable
constructor · 68
members · 68
methods · 68
JavaYacc
constructor · 60
members · 59
methods · 60
imp_union class · 60
structure · 54
symbol table · 67
class ABXsymtab · 68
class ABXsymtabentry · 68
class ABXVALUE · 67
Java Lexer
procedures to create · 56
Java Parser
example · 72
procedures to create · 55

L

lookupSymbol method
of class ABXSymbolTable · 35
of class JavaSymbolTable · 70

M

maxScope method
of class ABXSymbolTable · 36
of class JavaSymbolTable · 70
members
of class ABXError · 41
of class ABXLex · 13
of class ABXSymbolTable · 33
of class ABXYacc · 22
of class JavaError · 61
of class JavaExprNode · 65

of class JavaExprNodeList · 66
of class JavaLeaf · 64
of class JavaLeafList · 64
of class JavaLex · 56
of class JavaParseTree · 66
of class JavaParseTreeNode · 63
of class JavaSymbolTable · 68
of class JavaYacc · 59

methods
DelphiLex unit
TLex object · 82
DelphiYacc unit
TYacc object · 85
of class ABXError · 42
of class ABXExprNode · 50
of class ABXExprNodeList · 51
of class ABXLeafList · 49
of class ABXLex · 15
of class ABXParseTree · 52
of class ABXSymbolTable · 34
of class ABXYacc · 29
of class JavaError · 61
of class JavaExprNode · 65
of class JavaExprNodeList · 66
of class JavaLeaf · 64
of class JavaLeafList · 64
of class JavaLex · 57
of class JavaParseTree · 66
of class JavaParseTreeNode · 63, 64
of class JavaSymbolTable · 68
of class JavaYacc · 60

O

optimize method
of class ABXParseTree · 52
of class JavaParseTree · 66

P

parse tree node
analysis for building · 47
classifications · 46

Pascal Lexer
constant declaration · 114
functions · 116
errprefix · 118
get_yylineno · 116
input · 116
REJECT · 117
set_input_file · 116
unput · 116
YY_DEFAULT_ACTION · 116
YY_DO_BEFORE_ACTION · 117
YY_DO_BEFORE_SCAN · 117
YY_FATAL_ERROR · 117
YY_INIT_PROC · 117
YY_INPUT · 117
YY_LENG · 117
YY_OUTPUT · 117
yyerror · 118
yyless · 117
yylex · 116
yywrap · 117
procedures to create · 114
type declaration · 115
var declaration · 115

Pascal Parser
constant declaration · 118
functions
yyerrok · 119
yyParse · 119
procedures to create · 113
type declaration · 118
var declaration · 118

PCYACC OO TOOLKIT
basic classes · 1
C++ basic classes · 3
Delphi basic units · 77
general structure · 2
Java basic classes · 54
Pascal functions · 116
VBasic modules · 124
VBScript modules · 94

print_tree method
of class ABXParseTree · 52
of class JavaParseTree · 67

R

REJECT method
DelphiLex unit
of TLex object · 84
of class JavaLex · 59

S

Scanner Description Language · 10

set_input_file method
DelphiLex unit
of TLex object · 82

set_input_file_name method
of class JavaYacc · 60

set_input_file_name procedure
of VBasic Error Report · 126
of VBScript Error Report · 96

set_YYSTYPEInstance method
of class ABXLex · 15, 18

setScope method
of class ABXSymbolTable · 36
of class JavaSymbolTable · 70

setSymbolAttribute method
of class ABXSymbolTable · 36
of class JavaSymbolTable · 70

setSymbolType method
of class ABXSymbolTable · 36
of class JavaSymbolTable · 70
setSymbolValue method
of class ABXSymbolTable · 36
of class JavaSymbolTable · 71
show_tree method
of class ABXParseTree · 52
of class JavaParseTree · 67
state stack
of C++ Parser · 23
symbol table
entry definition · 33
entry structure · 32
representation · 30, 31

T

type
object type
TLex · 81
TYacc · 85
Pascal Parser
valuestack · 118
symbol's · 31
TObject type · 81
YYSTYPE · 12, 18
yylval and yyval
share · 18
sharing implementation · 23

U

unput method
DelphiLex unit
of TLex object · 82
of class ABXLex · 15, 18
of class JavaLex · 58
unput procedure
of VBasic Lex · 125
of VBScript Lex · 95

V

value stack
of C++ Parser · 23
VBasic Error Report
module · 126
VBasic Lexer
code structure · 123
module · 124
procedures to create · 122
VBasic Parser
code structure · 124
module · 126
procedures to create · 121
VBScript Error Report
module · 96
VBScript Lexer
code structure · 93
example · 98
module · 94
procedures to create · 92
VBScript Parser
code structure · 94
example · 104
module · 96
procedures to create · 91

Y

YY_BUF_LIM
of class JavaLex · 57
of DelphiLex unit · 80
of Pascal Lexer · 115
YY_BUF_MAX
of class JavaLex · 57
of DelphiLex unit · 80
of Pascal Lexer · 115
YY_BUF_SIZE
of class JavaLex · 57
of DelphiLex unit · 80
of Pascal Lexer · 115
YY_DEFAULT_ACTION method
DelphiLex unit
of TLex object · 83
of class JavaLex · 58
YY_DEFAULT_ACTION procedure
of VBasic Lex · 125
of VBScript Lex · 95
YY_DO_BEFORE_ACTION method
DelphiLex unit
of TLex object · 83
of class JavaLex · 59
YY_DO_BEFORE_ACTION procedure
of VBasic Lex · 124
of VBScript Lex · 94
YY_DO_BEFORE_SCAN method
DelphiLex unit
of TLex object · 83
of class JavaLex · 59
YY_DO_BEFORE_SCAN procedure
of VBasic Lex · 125
of VBScript Lex · 95
YY_FATAL_ERROR method
DelphiLex unit
of TLex object · 83
of class JavaLex · 58
YY_FATAL_ERROR procedure
of VBasic Lex · 125
of VBScript Lex · 95
YY_INIT method
of class JavaLex · 58
YY_INIT_PROC method

DelphiLex unit
of Tlex object · 83
YY_INIT_PROC procedure
of VBasic Lex · 125
of VBScript Lex · 95
YY_INPUT method
DelphiLex unit
of TLex object · 83
of class JavaLex · 58
YY_INPUT procedure
of VBasic Lex · 125
of VBScript Lex · 95
YY_LENG method
DelphiLex unit
of TLex object · 83
of class JavaLex · 59
YY_MAX_LINE
of class JavaLex · 57
of DelphiLex unit · 80
of Pascal Lexer · 115
YY_OUTPUT method
DelphiLex unit
of TLex object · 83
of class JavaLex · 58
YY_OUTPUT procedure
of VBasic Lex · 125
of VBScript Lex · 95
YY_SET_EOL method
of class JavaLex · 58
yyCheck method
of class ABXLex · 15, 18
yyDisplay method
of class ABXError · 42
of class JavaError · 63
yydisplay procedure
of VBasic Error Report · 126
of VBScript Error Report · 96
yyerrok method
DelphiYacc unit
of TYacc object · 86
of class JavaYacc · 60
yyerrok procedure
of VBasic Yacc · 126
of VBScript Yacc · 96
yyerror method
DelphiLex unit
of Tlex object · 84
yyError method
of class ABXError · 38, 42
of class JavaError · 62
yyerror procedure
of VBasic Error Report · 126
of VBScript Error Report · 96
yyErrPrefix method
of class ABXError · 38, 42
of class JavaError · 62
yyGetCharNumber method
of class ABXError · 39, 42
of class JavaError · 62
yyGetErrFileName method
of class ABXError · 39
of class JavaError · 62
yyGetErrorCount method
of class ABXError · 39, 42
of class JavaError · 62
yyGetExpectedTokens method
of class ABXError · 39, 43
of class JavaError · 62
yyGetFileName method
of class ABXError · 43
yyGetLineNumber method
of class ABXError · 39, 43
of class JavaError · 62
yyGetToken method
of class ABXError · 39, 43
of class JavaError · 61
yyInit method
of class ABXLex · 15, 19
yyInsertToken method
of class ABXError · 39, 43
of class JavaError · 63
yyinserttoken procedure
of VBasic Error Report · 127
of VBScript Error Report · 97
yyless method
DelphiLex unit
of TLex object · 83
of class JavaLex · 58
yylex method
DelphiLex unit
of TLex object · 82
of class JavaLex · 59
yyLex method
of class ABXLex · 15, 19
yylex procedure
of VBasic Lex · 125
of VBScript Lex · 95
yylval · 13, 18, 23, 99
yyMatchToken method
of class ABXError · 39, 43
of class JavaError · 63
yymatchtoken procedure
of VBasic Error Report · 127
of VBScript Error Report · 97
YYMAXDEPTH
of class ABXYacc · 29
yyparse method
of class JavaYacc · 60
yyParse method
DelphiYacc unit
of TYacc object · 85
of class ABXYacc · 29
yyparse procedure
of VBasic Yacc · 126
of VBScript Yacc · 96
yyPeer method
of class ABXLex · 15, 19
yyReplaceToken method

of class ABXError · 40, 43
of class JavaError · 63
yyreplacetoken procedure
of VBasic Error Report · 127
of VBScript Error Report · 97
yySearch method
of class ABXLex · 15, 19
yySetBuffer method
of class ABXLex · 15, 19
yySetErrText method
of class ABXError · 39, 43
of class JavaError · 62
yySetInput method
of class ABXLex · 15, 19
yySetLexer · 28
yySetLexer method
of class ABXYacc · 29
yySetTokStack method
of class ABXYacc · 29
yySkipSymbol method
of class ABXError · 40, 43
of class JavaError · 63
yyskipsymbol procedure
of VBasic Error Report · 127
of VBScript Error Report · 97
yySkipToken method
of class ABXError · 40, 44
of class JavaError · 62
yyskiptoken procedure
of VBasic Error Report · 127
of VBScript Error Report · 97
YYSTYPE · 12, 15, 23, 72, 78
yytext · 14, 30, 58, 83, 95, 116, 125
yyval · 13, 23, 110
yywrap method
DelphiLex unit
of TLex object · 83
of class ABXLex · 19
of class JavaLex · 58
yyWrap method
of class ABXLex · 16
yywrap procedure
of VBasic Lex · 125
of VBScript Lex · 95

PCYACC
OBJECT ORIENTED
TOOLKIT

PCYACC® is a software product of ABRAXAS SOFTWARE INC.

For more information, contact

ABRAXAS SOFTWARE INC.
Post Office Box 19586
PORTLAND, OR 97280 USA

TEL: 503-232-0540
FAX: 503-232-0543
support@pcyacc.com
www.pcyacc.com

Copyright © 1984-2000 by ABRAXAS SOFTWARE INC.

PCYACC OO TOOLKIT • Printed - December 11, 2000