Jhanvi Parekh

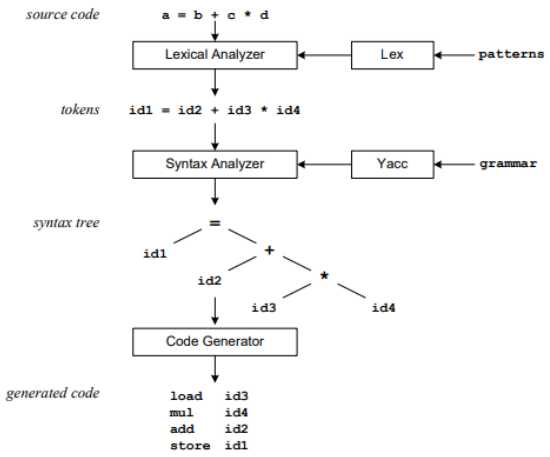
60009210033

CSE(Data Science)

1. Write a program to implement Symbol table
2. Write a program to identify tokens from the given Program

Link for the above two programs:
https://colab.research.google.com/drive/1gTX6wbuVBKQijcjZmhxwVFM8asTxTcY-?usp=sharing

3. case Study on LEX and YACC



The file you produce with a text editor has the patterns shown in the diagram above. Lex will read your patterns and produce C code for a scanner or lexical analyser. Based on your patterns, the lexical analyser compares strings in the input and turns the strings into tokens. A token is strings represented numerically to streamline processing. Identifiers are entered into a symbol table by the lexical analyser when they are discovered in the input stream. Other details like the data type (integer or real) and location of each variable in memory may also be included in the symbol table. All identifier references after that refer to the relevant symbol table index. The text file representing the grammar in the diagram above was produced using a text editor. Yacc

LEX

- Lex is officially known as a "Lexical Analyser".
- Its main job is to break up an input stream into more usable elements. Or in, other words, to identify the "interesting bits" in a text file.
- For example, if you are writing a compiler for the C programming language, the symbols { } ( ); all have significance on their own.
- The letter a usually appears as part of a keyword or variable name, and is not interesting on its own.
- Instead, we are interested in the whole word. Spaces and newlines are completely uninteresting, and we want to ignore them completely, unless they appear within quotes "like this"

- All of these things are handled by the Lexical Analyser.
- A tool widely used to specify lexical analysers for a variety of languages
- We refer to the tool as Lex compiler, and to its input specification as the Lex language.

Lex specifications:

A Lex program (the .l file) consists of three parts:

declarations

%%

translation rules

%%

auxiliary procedures

1. The declarations section includes declarations of variables, manifest constants (A manifest constant is an identifier that is declared to represent a constant e.g. # define PIE 3.14).
2. and regular definitions.
3. The translation rules of a Lex program are statements of the form :

   p1 {action 1}

   p2 {action 2}

   p3 {action 3}

   … …

   … …

   Where each p is a regular expression and each action is a program fragment describing what action the lexical analyser should take when a pattern p matches a lexeme. In Lex the actions are written in C.

   1. The third section holds whatever auxiliary procedures are needed by the actions. Alternatively these procedures can be compiled separately and loaded with the lexical analyser.

How does this Lexical analyser work?

- The lexical analyser created by Lex behaves in concert with a parser in the following manner.
- When activated by the parser, the lexical analyser begins reading its remaining input, one character at a time, until it has found the longest prefix of the input that is matched by one of the regular expressions p.
- Then it executes the corresponding action. Typically the action will return control to the parser.
- However, if it does not, then the lexical analyser proceeds to find more lexemes, until an action causes control to return to the parser.

- The repeated search for lexemes until an explicit return allows the lexical analyser to process white space and comments conveniently.
- The lexical analyser returns a single quantity, the token, to the parser. To pass an attribute value with information about the lexeme, we can set the global variable yylval.
- e.g. Suppose the lexical analyser returns a single token for all the relational operators, in which case the parser won't be able to distinguish between "<=","=>","<",">","==" etc. We can set yylval appropriately to specify the nature of the operator.
- Note: To know the exact syntax and the various symbols that you can use to write the regular expressions visit the manual page of FLEX in LINUX :

  $man flex

The two variables yytext and yyleng

- Lex makes the lexeme available to the routines appearing in the third section through two variables yytext and yyleng
  1. yytext is a variable that is a pointer to the first character of the lexeme.
  2. yyleng is an integer telling how long the lexeme is.
- A lexeme may match more than one patterns. How is this problem resolved?
- Take for example the lexeme if. It matches the patterns for both keyword if and identifier.
- If the pattern for keyword if precedes the pattern for identifier in the declaration list of the lex program the conflict is resolved in favour of the keyword.
- In general this ambiguity-resolving strategy makes it easy to reserve keywords by listing them ahead of the pattern for identifiers.
- The Lex's strategy of selecting the longest prefix matched by a pattern makes it easy to resolve other conflicts like the one between "<" and "<=".

  In the lex program, a main () function is generally included as:

```
main (){
yyin = fopen (filename,"r");
while (yylex ());
 }
```

- Here filename corresponds to input file and the yylex routine is called which returns the tokens.

YACC

- Yacc is officially known as a "parser".
- It's job is to analyse the structure of the input stream, and operate of the "big picture".
- In the course of it's normal work, the parser also verifies that the input is syntactically sound.

- Consider again the example of a C-compiler. In the C-language, a word can be a function name or a variable, depending on whether it is followed by a (or a = There should be exactly one } for each { in the program.
- YACC stands for "Yet another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers.

How does this yacc works?

- yacc is designed for use with C code and generates a parser written in C.
- The parser is configured for use in conjunction with a lex-generated scanner and relies on standard shared features (token types, yylval, etc.) and calls the function yylex as a scanner coroutine.
- You provide a grammar specification file, which is traditionally named using a .y extension.
- You invoke yacc on the .y file and it creates the y.tab.h and y.tab.c files containing a thousand or so lines of intense C code that implements an efficient LALR (1) parser for your grammar, including the code for the actions you specified.
- The file provides an extern function yyparse.y that will attempt to successfully parse a valid sentence.
- You compile that C file normally, link with the rest of your code, and you have a parser! By default, the parser reads from stdin and writes to stdout, just like a lex-generated scanner does.

Difference between LEX and YACC

- Lex is used to split the text into a list of tokens, what text become token can be specified using regular expression in lex file.
- Yacc is used to give some structure to those tokens. For example in Programming languages, we have assignment statements like int a = 1 + 2; and i want to make sure that the left hand side of '=' be an identifier and the right side be an expression [it could be more complex than this]. This can be coded using a CFG rule and this is what you specify in yacc file and this you cannot do using lex (lexcannot handle recursive languages).
- A typical application of lex and yacc is for implementing programming languages.
- Lex tokenizes the input, breaking it up into keywords, constants, punctuation, etc.
- Yacc then implements the actual computer language; recognizing a for statement, for instance, or a function definition.
- Lex and yacc are normally used together. This is how you usually construct an application using both:
- Input Stream (characters) -> Lex (tokens) -> Yacc (Abstract Syntax Tree) -> Your Application