PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Teaching Assistant : Ananya Angadi

Compiler Design

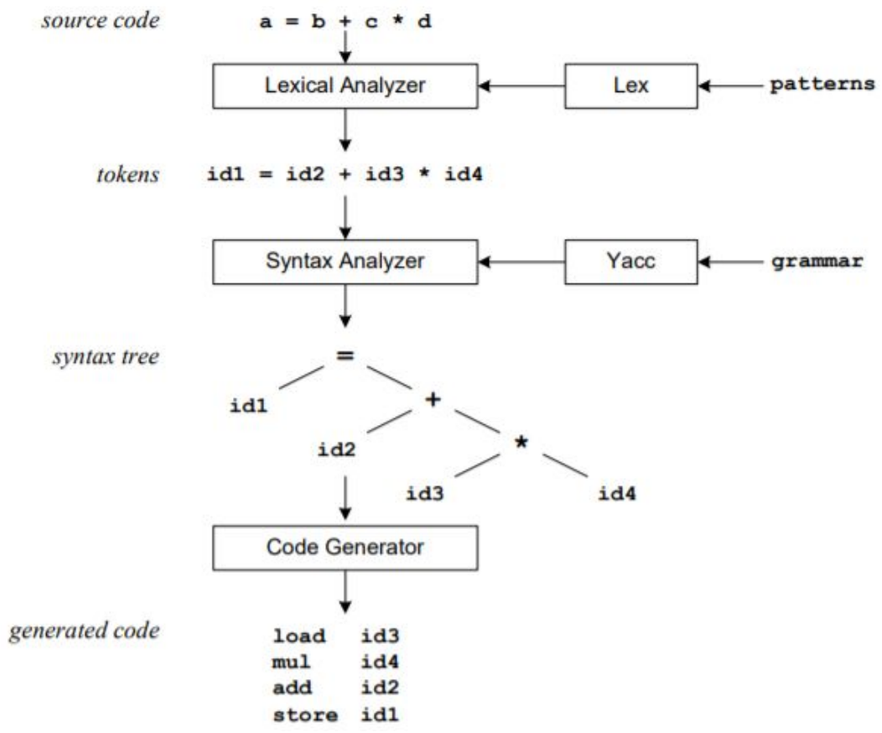
Building a Mini Compiler - Intro to Lex and Yacc

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

Introduction



- Lex: tool used to write the lexical analyzer
- Yacc: tool used to generate the parsing program (for context free grammars)
- Lex and Yacc are designed to complement each other
- Tokens identified by lexer are passed on to parser which fits them into the grammar of the language

Compiler Design

Installation

Linux

$ sudo apt-get update

$ sudo apt-get install flex
(for lex)

$ sudo apt-get install bison
(for yacc)

Windows

1. Download Flex 2.5.4a
2. Download Bison 2.4.1
3. Download DevC++
4. Install Flex at “C:\GnuWin32“
5. Install Bison at “C:\GnuWin32“
6. Install DevC++ at “C:\Dev-Cpp“
7. Open Environment Variables.
8.Add “C:\GnuWin32\bin;C:\Dev-Cpp\bin;” to the path

MacOS

1. Open the terminal.
2. Install homebrew:
ruby -e "$(curl -fsSL $ https://raw.githubusercontent.com/Homebrew/install/master/install)"
3. Install Lex: brew install flex
4. Install Yacc: brew install bison

Compiler Design

Lex

General outline of a lex program:

... definitions ...

%%

... rules ...

%%

... subroutines ...

To execute:

$ lex hello.l // creates lex.yy.c

$ gcc lex.yy.c

$ ./a.out < input

lex.yy.c is the generated C file containing the definition for yylex() which drives the lexical analysis.

Sample Program

```
/*lex program to match identifiers*/

%{
#include<stdio.h>
int i = 0;
%}

/* Rules Section*/
%%

([a-zA-Z0-9])*    {printf("Identifier\n");}
.                 {printf("%s\n",yytext);}

%%

/* Subroutines section */
int main()
{
    // The function that starts the analysis
    yylex();
    return 0;
}
```

Lex

Definitions

- Specify the global declarations in the generated C file.
- Have scope throughout the program
- Enclosed inside %{ ... %}

Rules

- The regex for each token is specified, followed by the action to be performed when there is a match
- ‘yytext’ is a variable which holds the currently matched lexeme.

Subroutines

- Define all the necessary functions in this section
- Main() function defines the main function of the generated C file.
- Main makes a call to the yylex() function, which performs the lexical analysis.

Note: We can also specify regular definitions, which are commonly occurring regex patterns. This saves the effort of rewriting them everywhere.

Compiler Design

Regex symbols

- (.) =>The dot symbol will match any other symbol.
- (*) => The asterisk symbol will tell the computer to match the preceding character for 0 or more times.
- (+) => The asterisk symbol will tell the computer to match the preceding character for 1 or more times.
- (?) => The question mark symbol tells the computer that the preceding character may or may not be present.
- ([seof_characters]) => Matches any single character present in seof_characters].
- (\) => The escape symbol enables us to match the functional symbols as a regular character.
- ((regex)) => Parenthesis groups a regex to act as a single block
- (|) => Vertical bar separates out 2 regex and matches any one of them.

Compiler Design

Regex symbols

Regex	Matched text
ab.c	ac, abc, abbc, ….
ab+c	abc, abbbc, ….
yacc(lex)+	yacclex, yacclexlex….
yacc(lex)*	yacc, yacclex, yacclexlex….
a\+b	a+b
ab?c	abc, ac
(abc|def)	abc, def
[a-z]	one letter from a-z
[A-Z_]	one letter from A-Z or underscore
.*	Matches everything

Compiler Design

Yacc

- The general outline of yacc remains the same as lex
- The rules section contains production rules for the grammar
- %token is used to declare token
- %start is used to indicate the start symbol

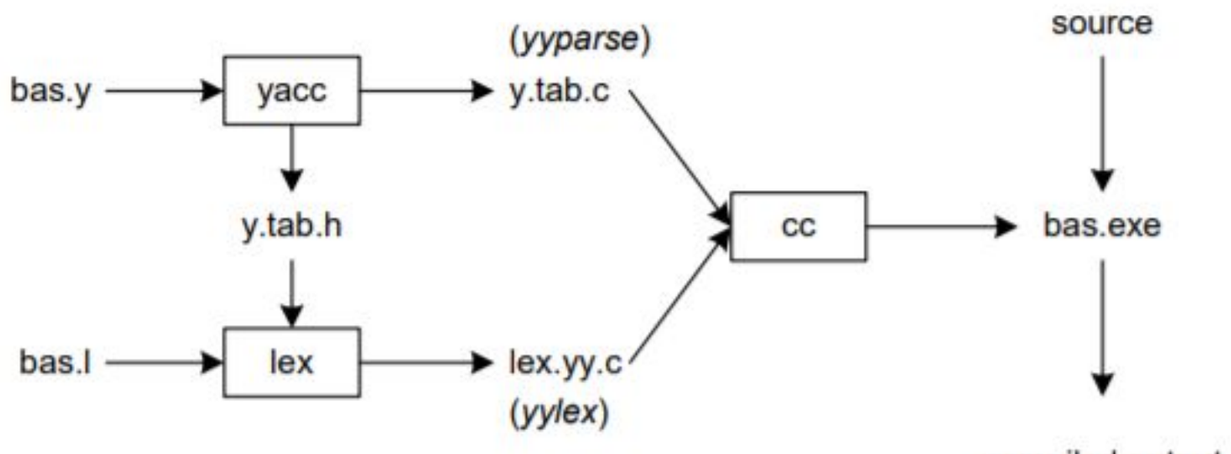
Execution

```
$ yacc -d prog.y // creates y.tab.h, y.tab.c
$ gcc y.tab.c
$ ./a.out < input
```

- y.tab.h contains the token definitions
- y.tab.c contains the definition for yyparse(), which drives the parsing

Compiler Design

Interaction between Lex and Yacc




```
$ lex lexer.l  // generates lex.yy.c - contains definition of yylex()
$ yacc parser.y  // generates (1) y.tab.c - contains definition of yyparse()
                              (2) y.tab.h - contains token definitions
```

- Parser drives the lexical analysis - it must know the function which performs lexical analysis
- Hence, we must declare yylex() function in definitions part of the yacc file
- Similarly, since we expect the lex file to generate tokens, it must know their definitions
- Hence, we must include the y.tab.h file in definitions part of the lex file

Execution

Linux/MacOS

```
$ lex lexer.l // generates lex.yy.c
$ yacc parser.y // generates y.tab.c, y.tab.h
$ gcc y.tab.c lex.yy.c -ll -ly // linking lex and yacc
$ ./a.out < input // run the executable
```

Windows

```
$ bison -dy prog.y
$ flex hello.l
$ gcc y.tab.c lex.yy.c
$ a.exe < input
```

Compiler Design

Lab task: Implementing lexical and syntax analyser

PES UNIVERSITY ONLINE

A: Implement the lexer

Step 1: Include all the declarations in the definitions part - within %{ … %}

- Remember to include the y.tab.h file, which contains definitions of the token, otherwise an error will occur.
- Also include the standard stdio.h and declare the yyerror() function.

Step 2: Write the regular definitions

When specifying the regex for identifiers and numbers, regular definitions help eliminate redundancies and improve readability.

Example:

```
digit  [0-9]
letter [a-zA-Z]
```

Compiler Design

Lab task: Implementing lexical and syntax analyser

PES UNIVERSITY ONLINE

A: Implement the lexer

Step 3: List the rules after the %%

- Specify the regex followed by the action
- Action usually involves returning the corresponding token
- **Important:** The order in which you specify the regex for the tokens matters. Remember the first match and maximal munch principles.
- **Important:** For single character tokens like ';' , the character itself is the token, there is no need to define one separately.
  Instead, we can simply return *yytext. yytext contains the most recently matched lexeme, which will be the character in this case.

Refer lex programs covered in class for an idea of how the program is supposed to look.

Compiler Design

Lab task: Implementing lexical and syntax analyser

A: Implement the lexer

The following tokens must be recognised by the lexer

INT	(integer type)
CHAR	(character)
FLOAT	(float type)
DOUBLE	(double type)
WHILE	(while keyword)
FOR	(for keyword)
DO	(do keyword)
IF	(if keyword)
ELSE	(else keyword)
INCLUDE	(#include)
MAIN	(main keyword)
ID	(identifier)
NUM	(number)

STRLITERAL	(string literal)
HEADER	(header file name)
EQCOMP	(= =)
GREATEREQ	(> =)
LESSEREQ	(< =)
NOTEQ	(! =)
INC	(++)
DEC	(--)
OROR	(Logical OR - | |)
ANDAND	(Logical AND - &&)

!	(Logical not)
(	(Opening bracket)
)	(Closing bracket)
[	(array indexing - opening)
]	(array indexing - closing)
{	(Opening brace)
}	(Closing brace)
;	(Semicolon)
,	(Comma)

Compiler Design

Lab task: Implementing lexical and syntax analyser

PES UNIVERSITY ONLINE

A: Implement the lexer

Following are the operators we are handling (arithmetic, logical, relational)

Precedence	Operator	Associativity	Description
1	()	Left-to-right	Function call or parentheses
	[]		Accessing elements of an array
	.		Accessing fields of a structure
2	-	Right-to-left	Unary minus
	!		Logical NOT
	++		Increment
	--		Decrement
3	*	Left-to-right	Multiplication
	/		Division
4	+	Left-to-right	Plus
	-		Binary minus

Compiler Design

Lab task: Implementing lexical and syntax analyser

PES UNIVERSITY ONLINE

A: Implement the lexer

Following are the operators we are handling (arithmetic, logical, relational)

Precedence	Operator	Associativity	Description
5	>	left-to-right	Greater than
	>=		Not less than
	<		Less than
	<=		Not greater than
6	= =		Equal to
	! =		Not equal to
7	&&		Logical AND
8	||		Logical OR
9	=	Right-to-left	Assignment

Lab task: Implementing lexical and syntax analyser

B: Implement the parser

Step 1: Include all the declarations in the definitions part - within %{ … %}

- Ensure that you declare the yylex() function (defined in lex.yy.c, drives lexical analysis)
- If you have defined a main function in lexer for testing, remove it now - lexer will be called from within the parser
- Declare the yyerror() function

Step 2: Declare the tokens

- Any token returned in lexer must be defined in parser
- Declare tokens using %token
- After the parser code is compiled, these token definitions will be part of y.tab.h

Compiler Design

Lab task: Implementing lexical and syntax analyser

B: Implement the parser

Step 3: Define the grammar

Grammar: Start, program, variable declaration, type and assignment

Start → Prog

Prog → Include < Header > Prog
|MainF Prog
|Declr ; Prog
| Assgn ; Prog
| λ

Declr → Type ListVar

Listvar → ListVar , ID| ID

Type →int | float | double | char

Assgn → ID = Expr

Compiler Design

Lab task: Implementing lexical and syntax analyser

PES UNIVERSITY ONLINE

B: Implement the parser

Step 3: Define the grammar

Grammar: Expression, arithmetic expression, main function

Expr → Expr Relop E | E

Relop → < | > | <= | >= | == | !=

E → E + T | E - T | T

T → T * F | T / F | F

F → (Expr) | ID | NUM

MainF → Type Main (Empty_ListVar) { Stmt}

Empty_ListVar → ListVar | λ

Compiler Design

Lab task: Implementing lexical and syntax analyser

B: Implement the parser

Step 3: Define the grammar

Grammar: Statement, single statement, block, if, if-else, while

Stmt → SingleStmt Stmt | Block Stmt | λ

SingleStmt → Declr | Assgn ; | if Cond Stmt | if Cond Stmt Else Stmt | WhileL

Block → { Stmt }

WhileL → While (Cond) While_2

COND → Expr | Assgn

While_2 → { Stmt } | λ

Compiler Design

Lab task: Implementing lexical and syntax analyser

PES UNIVERSITY ONLINE

B: Implement the parser

Step 3: Define the grammar

- Terminate each rule with a semicolon
- Enclose single character tokens within quotes, such as ';'
- There should be no reduce-reduce conflicts
- Shift-reduce conflicts are acceptable to a certain degree - they can be solved by removing redundancies in RHS of the rule
- In case of a shift-reduce conflict, shift is prioritised over reduce
- Specify the %start which indicates the start symbol
- There might rise a conflict in the case of if and if-else, in which case you can use %non-assoc and %pred to prioritise if over if-else.

If the syntax is valid, the output should be "Valid syntax"
Hint: This should be printed when the program is completely parsed.
If syntax is invalid, print 'Error: <error>, line number: <line no>,token: <token>'. You will have to define your own yyerror function.

Compiler Design

Lab task: Implementing lexical and syntax analyser

PES UNIVERSITY ONLINE

B: Implement the parser

Step 3: Subroutines section

Define the main function, and call yyparse(), which will perform parsing. yyparse() will in turn call yylex() internally.

Step 4: Execute

Test your code against a sample input, steps for execution have already been discussed

A note on error handling:

Sample error message: Error: syntax error, line number: 5,token: abc

When an error is found, we would like to continue parsing in order to find more errors instead of exiting. For this, we can have error productions.

These look like X : Y

| error

Compiler Design

Lab task: Implementing lexical and syntax analyser

PES UNIVERSITY ONLINE

C: Submission

Instructions:

1. Ensure the analyser outputs 'Valid syntax' if the file has no lexical or syntax errors, and ‘Error: <error>, line number: <line no>,token: <token>’ if the file has an error.
2. Please make sure the makefile is named 'makefile.mk'
3. Zip the 3 files into a .zip file named <SRN>.zip
4. For additional information on how to create a makefile, refer here

Deliverables:

1. lex file (containing the lexical analysis)
2. yacc file (containing the grammar)
3. make file named 'makefile.mk'(to generate the executable of the analyser)

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu