PES UNIVERSITY ONLINE

Compiler Design

Preet Kanwal

Department of Computer Science & Engineering

Compiler Design

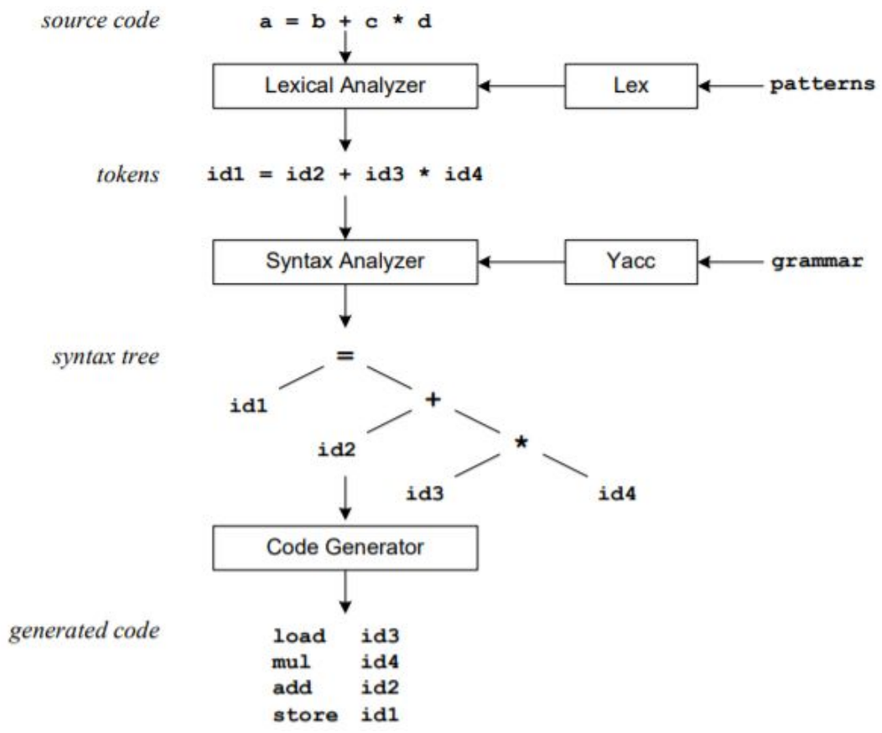
Building a Mini Compiler - Intro to Lex and Yacc

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

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

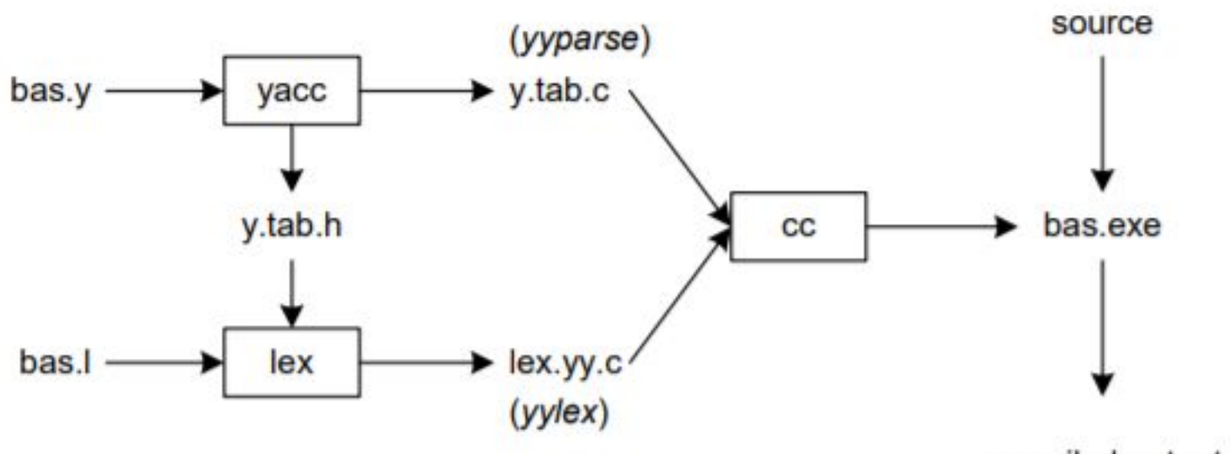
Execution

```
$ 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

PES UNIVERSITY ONLINE

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 // run the executable
```

Windows

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

Compiler Design

Mini-Compiler: lexer.l

```
%{
    /* DEFINITIONS */
     #define YYSTYPE char* // specify the data type for variable yylval
     #include "y.tab.h" // y.tab.h contains token definitions
     #include <stdio.h>
     extern void yyerror(const char *); // declare the error handling function
%}

/* Regular definitions */
digit  [0-9]
letter [a-zA-Z]
id  {letter}({letter}|{digit})*
digits {digit}+
opFraction    (\.{digits})?
opExponent    ([Ee][+-]?{digits})?
number {digits}{opFraction}{opExponent}

%%   // Separates definitions section from rules section
```

(Code continued on next slide)

Compiler Design

Mini-Compiler: lexer.l

```
\/\/(.*) ; // ignore comments
[\t\n] ; // ignore whitespaces
"int"     {return T_INT;}
"char"    {return T_CHAR;}
"double"  {return T_DOUBLE;}
"float"   {return T_FLOAT;}
"while"   {return T_WHILE;}
"if"      {return T_IF;}
"else"    {return T_ELSE;}
"do"      {return T_DO;}
"#include"   {return T_INCLUDE;}
"main"    {return T_MAIN;}
\".*\"    {return T_STRLITERAL;}
"=="      {return T_EQCOMP;}
"!="      {return T_NOTEQUAL;}
">="        {return T_GREATEREQ;}
“<="       {return T_LESSEREQ;}
```

(Code continued on next slide)

Compiler Design

Mini-Compiler: lexer.l

```
"("     {return *yytext;}
")"     {return *yytext;}
"."             {return *yytext;}
","             {return *yytext;} // yytext contains the currently matched lexeme
"{"             {return *yytext;} // single char tokens can be passed directly
"}"             {return *yytext;} // (they are tokens in themselves)
"*"             {return *yytext;}
"+"             {return *yytext;}
";"             {return *yytext;}
"-"             {return *yytext;}
"/"             {return *yytext;}
"="             {return *yytext;}
">"             {return *yytext;}
"<"             {return *yytext;}
{number}  {return T_NUM;}
{id}\.h   {return T_HEADER;}  // ending in .h => header file name
{id}            {return T_ID;}
.       {} // anything else => ignore

%%   // Separates rules section from subroutines section (there are no subroutines in this case)
```

Note:
There is no main function in this lex file.
This is because yylex() is called by the parser: lexer is not run independently

Compiler Design

Mini-compiler - parser.y

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    void yyerror(char* s); // error handling function
    int yylex(); // declare the function performing lexical analysis
    extern int yylineno; // track the line number

%}
/* declare tokens */
%token T_INT T_CHAR T_DOUBLE T_WHILE  T_INC T_DEC   T_OROR T_ANDAND
T_EQCOMP T_NOTEQUAL T_GREATEREQ T_LESSEREQ T_LEFTSHIFT T_RIGHTSHIFT
T_NUM T_ID T_PRINTLN T_STRING  T_FLOAT T_BOOLEAN T_IF T_ELSE
T_STRLITERAL T_DO T_INCLUDE T_HEADER T_MAIN


/* specify start symbol */
%start START

%%
```

(Code continued on next slide)

Compiler Design

Mini-compiler - parser.y : Grammar for program and variable declaration

```
START : PROG { printf("Valid syntax\n"); YYACCEPT; }   /* If program fits the grammar,
                                                          syntax is valid */
      ;     /* Anything within {} is C code, it is the action corresponding to the production rule */

PROG :  T_INCLUDE '<' T_HEADER '>' PROG /* include header */
    |MAIN PROG              /* main function  */
    |DECLR ';' PROG           /* declarations   */
    | ASSGN ';' PROG          /* assignments     */
    |                   /* end of program */
    ;
/* Grammar for variable declaration */
DECLR : TYPE LISTVAR
    ;   /* always terminate with a ; */

LISTVAR : LISTVAR ',' T_ID
      | T_ID
      ;

TYPE : T_INT
       | T_FLOAT
       | T_DOUBLE
       | T_CHAR
       ;
```

(Code continued on next slide)

Mini-compiler - parser.y : Grammar for assignment and expressions

```
/* Grammar for assignment */
ASSGN : T_ID '=' EXPR
    ;

EXPR : EXPR REL_OP E
       | E
       ;
       ;

REL_OP :   T_LESSEREQ
       | T_GREATEREQ
       | '<'
       | '>'
       | T_EQCOMP
       | T_NOTEQUAL
       ;
```

```
/* Expression Grammar */
E : E '+' T
    | E '-' T
    | T
    ;

T : T '*' F
    | T '/' F
    | F
    ;

F : '(' EXPR ')'
    | T_ID
    | T_NUM
    ;
```

(Code continued on next slide)

Compiler Design
Mini-compiler - parser.y : Main, Statement, If-Else

```
/* Grammar for main function */
MAIN : TYPE T_MAIN '(' EMPTY_LISTVAR ')' '{' STMT '}';

/* argument list can be empty, or have a list of variables */
EMPTY_LISTVAR : LISTVAR
       |   /* similar to lambda */
       ;

/* statements can be standalone, or parts of blocks */
STMT : STMT_NO_BLOCK STMT
       | BLOCK STMT
       |
       ;

/* to give IF precedence over IF-ELSE */
%nonassoc T_IFX
%nonassoc T_ELSE

STMT_NO_BLOCK : DECLR ';'
       | ASSGN ';'
       | T_IF COND STMT %prec T_IFX  /* if loop */
       | T_IF COND STMT T_ELSE STMT  /* if else loop */
       | WHILE
       ;
BLOCK : '{' STMT '}';
```

(Code continued on next slide)

Compiler Design

Mini-compiler - parser.y : Grammar for while, subroutine section


```
/* Grammar for while loop */
WHILE : T_WHILE '(' COND ')' WHILE_2;

/* Condition can be an expression or an assignment */
COND : EXPR
       | ASSGN
       ;
// while loop may or may not have block of statements
WHILE_2 : '{' STMT '}'
      | ';'
      ;
%%

/* error handling function */
void yyerror(char* s)
{
    printf("Error :%s at %d \n",s,yylineno);
}

/* main function - calls the yyparse() function which will in turn drive yylex() as well */
int main(int argc, char* argv[])
{
    yyparse();
    return 0;
}
```

THANK YOU

Preet Kanwal

Department of Computer Science & Engineering

preetkanwal@pes.edu