Compiler Lab Cheat sheet

March 8, 2025

Contents

1 Flex

1.1 Word Counter

1.1.1 lexer.l

```
%{
    #include <stdio.h>
    #include <string.h>
    int chars = 0;
    int words = 0;
    int lines = 0;
%}
%%
[a-zA-Z]+[a-zA-Z0-9]* {words++; chars += strlen(yytext);}
[0-9]+[a-zA-Z0-9]*  { words++; chars += strlen(yytext); }
\n         { lines++; chars++; }
.          { chars++; }
%%
int main() {
   // yyin = fopen("input.txt", "r");
   // yyout=freopen("output.txt","w",stdout);
    yylex();
    printf("# of words: %d\n", words);
    printf("# of lines: %d\n", lines);
    printf("# of characters: %d\n", chars);
    return 0;
}
int yywrap() {
    return 1;
}
```

1.2 Keyword Count

```
%{
    #include<stdio.h>
    int keyword=0;
    int identifier=0;
    int others=0;
%}
%%
"do"|"while"|"for"|"return"|"int"|"float"|"main"|"include"|
"stdio.h"|"if" {
    keyword++;
    printf("\'%s\' is a keyword\n",yytext);
}
[a-zA-Z_][a-zA-Z0-9]* {
    identifier++;
    printf("\'%s\' is an identifier\n",yytext);
}
. {others++;printf("\'%s\' is an others\n",yytext);}
%%
int main(){
    //yyin=fopen("input.txt","r");
   //freopen("output.txt","w",stdout);
    yylex();
    printf("# of keyword is %d\n",keyword);
    return 0;
}
int yywrap(){
    return 1;
}
```

1.3 Recognize octal

```
%{
#include <stdio.h>
#include <stdlib.h>
int count = 0;
%}
%%
0[0-7]+ {
// Convert octal string to decimal
long decimal = strtol(yytext, NULL, 8); // Base 8 for octal
if (decimal > 250) {
printf("Found %s\n", yytext);
count++;
}
}
[ \t\n] ; // Ignore whitespace and newlines
. ; // Ignore any other characters
%%
int yywrap() {
return 1;
}
int main() {
printf("Enter octal integers (end with Ctrl+D or Ctrl+Z):\n");
yylex();
printf("Total %d\n", count);
return 0;
}
```

1.4 Recognizes hex, binary, octal (Previous year)

//previous year was hex only
For instance, from input:
JJJ0x44a88C33K1b
0x110B
K02x9CB671011B WP
KKJJG
Your full-featured lexer should output something like the following:
0x44A88C33
0x110B
0x9CB671011B
Not recognized

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

```
#include <string.h>

void print_number(const char *type, const char *lexeme);
int found = 0; // Flag to check if a hex number was recognized
//in the current line
%}
%%
0[xX][0-9a-fA-F]+ {
    print_number("Hexadecimal", yytext);
    found = 1;  // Set flag since we found a hex number
}
0[oO][0-7]+ {
    print_number("Octal", yytext);
    found=1;
}
0[bB][01]+ {
    print_number("Binary", yytext);
    found=1;
}
[ \t]+   { /* Skip spaces and tabs */ }
\n  {
    if (!found) {
        printf("Not Recognized\n");
    }
    found = 0; // Reset flag for next line
}
. { /* Ignore other characters */ }
%%
void print_number(const char *type, const char *lexeme) {
    printf("%s: %s\n", type, lexeme);
}
int yywrap() {
    return 1;  // Signals end of input
}
int main() {
    printf("Enter numbers (Press Enter after each line,
    Ctrl+C to stop):\n");

    yylex(); // Start scanning input

    return 0;
}
```

2 Flex bison

2.1 Assaignment's Calculator

Write a simple calculator using Flex and Bison for the given grammar.

E-->E + E | E

- E | E * E | E / E | (E) | num

1. Use precedence and associativity to resolve conflicts

```
% left   '+' '-'
```

- '
  % left   '*' '/'
  ```



Figure 1: Calculator Question

### 2.1.1 lexer.l

```
%{
#include "parser.tab.h"
#include <stdlib.h>
%}

/* Token definitions */
%%

[0-9]+ { yylval = atoi(yytext); return NUM; }//positive numbers
"-"               { return SUB;}//Handle subtraction separately
"+"               { return ADD; }
"*"               { return MUL; }
"/"               { return DIV; }
"("               { return LPAREN; }
")"               { return RPAREN; }
[ \t]             { /* Ignore whitespace */ }
\n                { return '\n'; }
.                 { printf("Unknown character: %s\n", yytext);}

%%

int yywrap() {
    return 1;
}
```

2.1.2 parser.y

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

int yylex(void);
void yyerror(const char *s);
%}

/* Token declarations */
%token NUM
%token ADD SUB MUL DIV LPAREN RPAREN

/* Precedence and associativity */
%left ADD SUB
%left MUL DIV
%right UMINUS  /* Unary minus should have higher precedence */

%%

calclist:
      /* empty */
    | calclist expr '\n' { printf("= %d\n", $2); }
    ;/* Evaluate expression immediately */

expr:
      expr ADD expr { $$ = $1 + $3; }
    | expr SUB expr { $$ = $1 - $3; }
    | expr MUL expr { $$ = $1 * $3; }
    | expr DIV expr {
        if ($3 == 0) {
            printf("Error: Division by zero\n");
            exit(1);
        }
        $$ = $1 / $3;
    }
    | SUB expr %prec UMINUS { $$ = -$2; }/* Handle unary minus*/
    | LPAREN expr RPAREN { $$ = $2; }
    | NUM { $$ = $1; }
    ;

%%

int main() {
    printf("Enter expressions (press Enter after each expression,
    Ctrl+D to exit):\n");
    return yyparse();
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

2.2 Book's calculator

2.2.1 lexer.l

```
%{
#include "parser.tab.h"
extern int yylval;
%}
```
```

```
%%
[0-9]+  { yylval = atoi(yytext); return NUMBER; }//positive num
"-" {return SUB;}
"+" {return ADD;}
"*" {return MUL;}
"/" {return DIV;}
"|" {return ABS;}
[ \t] {/*ignore whitespace*/}
\n {return EOL;}
. { printf("Unrecognized character: %c\n", *yytext);}
%%
int yywrap(){
    return 1;
}
```

2.2.2 parser.y

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

void yyerror(const char *s);
int yylex();
%}

%token NUMBER
%token ADD SUB MUL DIV
%token ABS
%token EOL

%left ADD SUB
%left MUL DIV
%right ABS
%right UMINUS  /* Unary minus should have higher precedence */

%%
calclist:
        /* empty */
      | calclist exp EOL { printf("= %d\n", $2); }
      ;
exp:
    factor
    | exp ADD factor {$$=$1+$3;}
    | exp SUB factor {$$=$1-$3;}
    | SUB exp %prec UMINUS { $$ = -$2; }/* Handle unary minus */
    ;
factor:
    term
    |factor MUL term {$$=$1*$3;}
    |factor DIV term{
        if($3==0){
            yyerror("Divivsion by zero");
            $$=0;
        }else{
            $$=$1/$3;
        }
    }
    ;
term:
    NUMBER {$$=$1;}
    | ABS exp ABS {$$= ($2>=0)? $2: -$2;}
    ;
%%
int main(){
    printf("Eype expression\n");
    yyparse();
    return 0;
}
void yyerror(const char *s)
{
    fprintf(stderr,"Error %s\n",s);
}
```

2.3 Hex Calculator

2.3.1 lexer.l

```
%{
#include <stdio.h>
#include <stdlib.h> // for strtol
#include "parser.tab.h"  /
%}

%%

[ \t]   ; // Skip whitespace

\n      { return '\n'; } // End of line

"+"     return ADD;
"-"     return SUB;
"*"     return MUL;
"/"     return DIV;
"("     {return LPAREN;}
")"     {return RPAREN;}

0[xX][0-9a-fA-F]+ { // Hexadecimal number
    yylval = (int)strtol(yytext + 2, NULL, 16);
    return NUM;
}

[0-9]+  { // Decimal number
    yylval = atoi(yytext); // Convert decimal string to integer
    return NUM;
}

.       { fprintf(stderr, "Invalid character: %s\n", yytext); }

%%


int yywrap() {
    return 1;
}
```

2.3.2 parser.y

//same as assaignment's calculator

2.4 String Calculator (Prv year)

Use yacc/bison specifications to write a calculator capable of doing string algebra. Strings will be alphabetic (with letters of both upper and lower case). There are two operators: the '+' and '*' operators indicate concatenation and repetition respectively; there will be single or nested following the operator. Sample input-output are as follows:
Input –> a + b
Output –> Result = ab
Input –> a * C'3' * q
Output –> Result = aCCCq

2.4.1 lexer.l

```
%{
#include <stdio.h>
#include <string.h>
#include "parser.tab.h"

%}

%option noyywrap

%%

[a-zA-Z]+          { yylval.str = strdup(yytext); return STRING; }  /
"+"                { return PLUS; }
"*"                { return STAR; }
\'[0-9]+\'          { yylval.num = atoi(yytext+1); return NUMBER; }
\n                 { return EOL; }
[ \t]              { /* ignore whitespace */ }
.                  { printf("Unrecognized character: %c\n", *yytext);

%%
```

2.4.2 parser.y

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

```
void yyerror(const char *s);
int yylex();
%}

%union {
    char* str;
    int num;
}

%token <str> STRING
%token <num> NUMBER
%token PLUS STAR EOL

%type <str> calclist exp factor term rep
%type <num> num


%%

calclist:
    /* empty */
    | calclist exp EOL {printf("Result = %s\n", $2); free($2);}
;

exp:
    factor
    | exp PLUS factor {
          char* result=(char*)malloc(strlen($1) + strlen($3)+1);
          strcpy(result, $1);
          strcat(result, $3);
          free($1); free($3);
          $$ = result;
      }
    ;
factor:
    term
    |factor STAR term {
          char* result=(char*)malloc(strlen($1)+strlen($3)+1);
          strcpy(result, $1);
          strcat(result, $3);
          free($1); free($3);
          $$ = result;
    };

term:
    STRING {$$=$1;}
    | rep {$$=$1;}
    ;
rep:
    STRING num{
          int count = $2;  /* Use the int value directly */
          char* str = $1;
          int len = strlen(str);
          char* result = (char*)malloc(len * count + 1);
          result[0] = '\0';
          for (int i = 0; i < count; i++) {
              strcat(result, str);
          }
          free($1);
          $$ = result;
    };
num:
    NUMBER {$$=$1;}
    ;

%%

int main() {
    printf("String Algebra Calculator(Type and press Enter)\n");
    yyparse();
    return 0;
}

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

3 Sybol table

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

#define TABLE_SIZE 100

// Symbol Table Entry
typedef struct Symbol {
    char name[50];
    char type[20];
    int scope;
    struct Symbol* next; // For collision handling (linked list)
} Symbol;

// Symbol Table (hash table)
Symbol* symbolTable[TABLE_SIZE];

// Hash function to calculate index
unsigned int hash(char* name) {
    unsigned int hashValue = 0;
    for (int i = 0; name[i] != '\0'; i++) {
        hashValue = 31 * hashValue + name[i];
    }
    return hashValue % TABLE_SIZE;
}

// Insert into the symbol table
void insert(char* name, char* type, int scope) {
    unsigned int index = hash(name);
    Symbol* newSymbol = (Symbol*) malloc(sizeof(Symbol));
    strcpy(newSymbol->name, name);
    strcpy(newSymbol->type, type);
    newSymbol->scope = scope;
    newSymbol->next = symbolTable[index];
    symbolTable[index] = newSymbol;
}

// Lookup a symbol in the table
Symbol* lookup(char* name) {
    unsigned int index = hash(name);
    Symbol* current = symbolTable[index];
    while (current != NULL) {
        if (strcmp(current->name, name) == 0) {
            return current;
        }
        current = current->next;
    }
    return NULL;
}

int main() {
    insert("x", "int", 0);
    insert("y", "int", 1);

    Symbol* s = lookup("x");
    if (s != NULL) {
        printf("Found %s of type %s in scope %d\n", s->name,
        s->type, s->scope);
    } else {
        printf("Symbol not found\n");
    }

      s = lookup("y");
    if (s != NULL) {
        printf("Found %s of type %s in scope %d\n", s->name,
        s->type, s->scope);
    } else {
        printf("Symbol not found\n");
    }

    return 0;
}
```

4 Recursive Descent

```
/*
Recursive Descent Parsing Functions for the following CFG:
```

```
E -> T E'
E' -> + T E' | ε
T -> F T'
T' -> * F T' | ε
F -> (E) | id
*/

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>


const char *input;
char lookahead;

// Function declarations for recursive parsing
void E();
void E_prime();
void T();
void T_prime();
void F();

// Helper function to match and move forward in the input
void match(char expected)
{
    if (lookahead == expected)
    {
        lookahead = *++input; // Move to the next character
    }
    else
    {
        printf("Syntax Error: Expected '%c', found '%c'\n",
        expected, lookahead);
        exit(1);
    }
}

// Function to parse Expression: E -> T E'
void E()
{
    printf("E -> T E'\n");
    T();
    E_prime();
}

// Function to parse E' -> + T E' | ε
void E_prime()
{
    if (lookahead == '+')
    {
        printf("E' -> + T E'\n");
        match('+');
        T();
        E_prime();
    }
    else
    {
        printf("E' -> epsilon\n"); // ε production
    }
}

// Function to parse Term: T -> F T'
void T()
{
    printf("T -> F T'\n");
    F();
    T_prime();
}

// Function to parse T' -> * F T' | ε
void T_prime()
{
    if (lookahead == '*')
    {
        printf("T' -> * F T'\n");
        match('*');
        F();
        T_prime();
    }
    else
    {
        printf("T' -> epsilon\n"); // ε production
    }
}

// Function to parse Factor: F -> (E) | id
void F()
{
    if (lookahead == '(')
    {
        printf("F -> (E)\n");
        match('(');
        E();
        match(')');
    }
    else if (isalnum(lookahead))
    {
        printf("F -> id\n");
        match(lookahead); // Match identifier/number
    }
    else
    {
        printf("Syntax Error: Unexpected character '%c'\n",
        lookahead);
        exit(1);
    }
}

// Main function to start parsing
int main()
{
    // Input arithmetic expression
    // input = "(2++3)*5";
     input = (char *)malloc(100 * sizeof(char));

    printf("Input an arithmetic expression\n");
    while (scanf("%s", input) != EOF)
    {

        lookahead = *input; // Initialize lookahead

        printf("Parsing input: %s\n", input);

        E(); // Start parsing from the start symbol E

        if (lookahead == '\0')
        {
            printf("Parsing successful!\n");
        }
        else
        {
            printf("Syntax Error: Unexpected input after parsing.\n")
        }
        printf("Input an arithmetic expression\n");
    }

    return 0;
}
```

4.1 Recursive Descent II

```
Write a parser for the following grammar.

S → if C then S
| while C do S
| id = num ;
| id ++ ;
C → id == num | id != num

where S represents a statement and C represents a condition.

#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"

const char *input;
char lookahead;

void match(char expected) {
    if (lookahead == expected) {
```

```
        lookahead = *++input;
    } else {
        printf("Syntax error: expected %c, found %c", expected, lookahead);
        exit(1);
    }
}

void match_v2(char *s) {
    for (int i = 0; s[i] != '\0'; i++) {
        match(s[i]);
    }
}

void syntax_error() {
    printf("Sytax Error\n");
    exit(1);
}

void absorb_whitespace() {
    while (lookahead == ' ') lookahead = *++input;
}

void C() {
    if (isalnum(lookahead)) {
        printf("C -> id");
        match(lookahead);
        absorb_whitespace();
        if (lookahead == '=') {
            printf("==");
            match_v2("==");
            absorb_whitespace();
        } else if (lookahead == '!') {
            printf("!=");
            match_v2("!=");
            absorb_whitespace();
        } else syntax_error();

        if (isalnum(lookahead)){
            printf("num\n");
            match(lookahead);
            absorb_whitespace();
        }
        else syntax_error();
    } else syntax_error();
}

void S() {
    if (lookahead == 'i') {
        printf ("S -> if C then S\n");
        match_v2("if");
        absorb_whitespace();
        C();
        absorb_whitespace();
        match_v2("then");
        absorb_whitespace();
        S();
        absorb_whitespace();

    } else if (lookahead == 'w') {
        printf("S -> while C do S\n");
        match_v2("while");
        absorb_whitespace();
        C();
        absorb_whitespace();
        match_v2("do");
        absorb_whitespace();
        S();
        absorb_whitespace();
    } else if (isalnum(lookahead)) {
        printf("S -> id");
        match(lookahead);
        absorb_whitespace();
        if (lookahead == '=') {
            printf("=");
            absorb_whitespace();
            match('=');
            absorb_whitespace();
            if (isalnum(lookahead)) {
                printf("num");
                match(lookahead);
                absorb_whitespace();
            } else {
                syntax_error();
            }
        } else if (lookahead == '+') {
            printf("++");
            match_v2("++");
            absorb_whitespace();
        } else {
            syntax_error();
        }

        if (lookahead == ';') {
            printf(";\n");
            match(lookahead);
            absorb_whitespace();
        }
        else syntax_error();
    }
}

int main() {
    input = (char *)malloc(100 * sizeof(char));

    input = "if 2 == 3 then 3 = 2;";
    //printf("Input an arithmetic expression\n");

    //while (scanf("%s", input) != EOF) {
    lookahead = *input;
    printf("Parsing input: %s\n", input);
    S();
    if (lookahead == '\0') {
        printf("Parsing successful!\n");
    } else {
        syntax_error();
    }
    //}

}
```

5 Makefile

```
all:
    bison -d parser.y
    flex lexer.l
    gcc -o cal.exe parser.tab.c lex.yy.c
    .\cal
```