Project1 Report

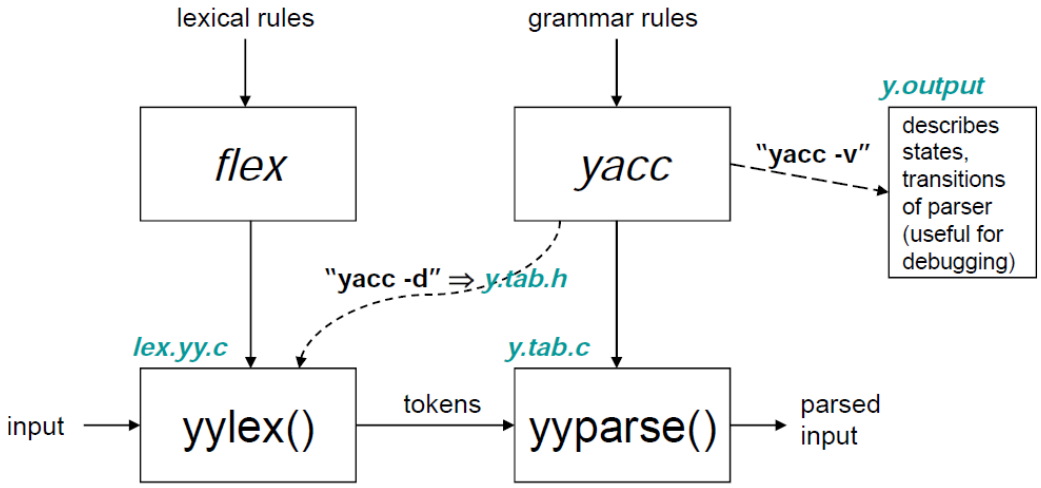
Weichen Li 5120309662
eizo.lee@sjtu.edu.cn

1. Introduction

In project1 I achieved a lexical analyzer and a parser, using Lex and Yacc.

The Lex reads the source file named smallc.l, and generates a C file – lex.yy.c, then this C file will be included in smallc.y, which is used by Yacc to generates a file called y.tab.c.

The process is displayed the graph below.



The parser is robust but still has some imperfection, which will be more precisely debugged in project2.

2. Smallc.l

```
/*This is smallc.l
This program will be compiled by lex,
which then output a file named "lex.yy.c",
lex.yy.c will be linked to y.tab.c in gcc and then produce a new compiler program.

In this program, strings of the input will be analyzed and splitted into tokens, by the yylex() program.

Then when Yacc needs a token, the yylex() will return one, if exists, to Yacc.

*/
%{
#include <stdio.h>
#include <string.h>
#include "y.tab.h"

/*linecount is used to sum up the total number of lines*/
```

```
int linecount = 0;

/*iniSize is used to store the size of a integer*/
int intSize = 0;
%}

/*Definition of the sets of classes*/
digit         [0-9]
letter        [a-zA-Z]
spedigit    (0x|0X|0)({digit})+
formaldigit       ({digit})+

%%
";"           {return SEMI;}
","           {return COMMA;}
"("           {return LP;}
")"           {return RP;}
"["           {return LB;}
"]"           {return RB;}
"{"           {return LC;}
"}"           {return RC;}
"."           {return DOT;}
"!"           {return LOGICNOT;}
"++"  {return PREINCRE;}
"--"   {return PREDEC;}
"~"           {return BITNOT;}
"*"           {return PRODUCT;}
"/"           {return DIVISION;}
"%"           {return MODULUS;}
"+"           {return PLUS;}
"-"           {return MINUS;}
"<<"  {return SHIFTLEFT;}
">>"  {return SHIFTRIGHT;}
">"           {return GREATERT;}
"<"           {return LESST;}
">="  {return NOTLESST;}
"<="  {return NOTGREATERT;}
"=="  {return EQUAL;}
"!="  {return NOTEQUAL;}
"&"           {return BITAND;}
"^"           {return BITXOR;}
"|"           {return BITOR;}
"&&"{return LOGICAND;}
"||"  {return LOGICOR;}
```

```
"="             {return ASSIGN;}
"+="  {return PLUSASSIGN;}
"-="   {return MINUSASSIGN;}
"*="  {return PRODUCTASSIGN;}
"/="  {return DIVISIONASSIGN;}
"&=" {return ANDASSIGN;}
"^="  {return NORASSIGN;}
"|="  {return ORASSIGN;}
"<<="       {return SLASSIGN;}
">>="       {return SRASSIGN;}
"int"  {return TYPE;}
"struct"    {return STRUCT;}
"return"   {return RETURN;}
"if"    {return IF;}
"else"       {return ELSE;}
"break"    {return BREAK;}
"continue"       {return CONT;}
"for" {return FOR;}
\n     {linecount++;}
("_"|{letter})({letter}|{digit}|"_")*  { return ID;}
{spedigit}|{formaldigit}    {intSize = atoi(yytext);
                                        /*Only decimal integers envolved here*/
                                        if((intSize>>30)==1)
                                              printf("integer size overflow!\n");
                                        else
                                              return INT;}
[ \t\r]                     /* skip whitespace */
.                                 {return UNKNOWN;       }
%%
/*When parsing completes*/
int yywrap(){
 printf("\nParsing complete.\n");
 return 1;
}
```

3. Smallc.y

```
/*
This is smallc.y

This source file defines the grammar of SmallC language.

Its structure unit is named token, which comes from the yylex() program in lex.yy.c .

This source file will generates a program called yyparse(), which interacts with yylex(),
```

and then produce a parsed input for the next stages of compiling.

*/

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern FILE *yyin;
extern int linecount;
extern int yychar;
%}
%token SEMI COMMA LC RC TYPE STRUCT RETURN IF ELSE BREAK CONT FOR ID INT DOT UNKNOWN

%start PROGRAM

%right SRASSIGN SLASSIGN ORASSIGN NORASSIGN ANDASSIGN DIVISIONASSIGN PRODUCTASSIGN MINUSASSIGN PLUSASSIGN ASSIGN
%left LOGICOR
%left LOGICAND
%left BITOR
%left BITXOR
%left BITAND
%left EQUAL NOTEQUAL
%left GREATERT LESST NOTGREATERT NOTLESST
%left SHIFTLEFT SHIFTRIGHT
%left PLUS MINUS
%left PRODUCT DIVISION MODULUS
%right LOGICNOT PREINCRE PREDEC BITNOT
%left LP RP LB RB

%%
PROGRAM       : EXTDEFS;
EXTDEFS   : EXTDEF EXTDEFS
 |
 ;
EXTDEF    : TYPE EXTVARS SEMI
 | STSPEC SEXTVARS SEMI
 | TYPE FUNC STMTBLOCK
 ;
SEXTVARS: ID
 | ID COMMA SEXTVARS
 |
```

```
 ;
EXTVARS   : VAR
 | VAR ASSIGN INIT
 | VAR COMMA EXTVARS
 | VAR ASSIGN INIT COMMA EXTVARS
 |
 ;
STSPEC     : STRUCT ID LC SDEFS RC
 | STRUCT LC SDEFS RC
 | STRUCT ID
 ;
FUNC: ID LP PARAS RP
 ;
PARAS      : TYPE ID COMMA PARAS
 | TYPE ID
 |
 ;
STMTBLOCK: LC DEFS STMTS RC
 ;
STMTS      : STMT STMTS
 |
 ;

STMT: IF LP EXP RP STMT1 ELSE STMT
 | IF LP EXP RP STMT
 | EXP SEMI
 | STMTBLOCK
 | RETURN EXP SEMI
 | FOR LP EXP SEMI EXP SEMI EXP RP STMT
 | CONT SEMI
 | BREAK SEMI
 ;
STMT1      : IF LP EXP RP STMT1 ELSE STMT1
 | EXP SEMI
 | STMTBLOCK
 | RETURN EXP SEMI
 | FOR LP EXP SEMI EXP SEMI EXP RP STMT1
 | CONT SEMI
 | BREAK SEMI
 ;

DEFS : TYPE DECS SEMI DEFS
 | STSPEC SDECS SEMI DEFS
 |
```

```
 ;
SDEFS       : TYPE SDECS SEMI SDEFS
 |
 ;
SDECS       : ID COMMA SDECS
 | ID
 ;
DECS : VAR
 | VAR COMMA DECS
 | VAR ASSIGN INIT COMMA DECS
 | VAR ASSIGN INIT
 ;
VAR   : ID
 | VAR LB INT RB
 ;
INIT   : EXP
 | LC ARGS RC
 ;
EXP   : EXPS
 |
 ;
EXPS : MINUS    EXPS %prec PRODUCT
 | LOGICNOT EXPS
 | PREINCRE EXPS
 | PREDEC EXPS
 | BITNOT EXPS
 | EXPS PRODUCT EXPS
 | EXPS DIVISION EXPS
 | EXPS MODULUS EXPS
 | EXPS PLUS EXPS
 | EXPS MINUS EXPS
 | EXPS SHIFTLEFT EXPS
 | EXPS SHIFTRIGHT EXPS
 | EXPS GREATERT EXPS
 | EXPS LESST EXPS
 | EXPS NOTLESST EXPS
 | EXPS NOTGREATERT EXPS
 | EXPS EQUAL EXPS
 | EXPS NOTEQUAL EXPS
 | EXPS BITAND EXPS
 | EXPS BITXOR EXPS
 | EXPS BITOR EXPS
 | EXPS LOGICAND EXPS
 | EXPS LOGICOR EXPS
```

```
 | EXPS ASSIGN EXPS
 | EXPS PLUSASSIGN EXPS
 | EXPS MINUSASSIGN EXPS
 | EXPS PRODUCTASSIGN EXPS
 | EXPS DIVISIONASSIGN EXPS
 | EXPS ANDASSIGN EXPS
 | EXPS NORASSIGN EXPS
 | EXPS ORASSIGN EXPS
 | EXPS SLASSIGN EXPS
 | EXPS SRASSIGN EXPS
 | LP EXPS RP
 | ID LP ARGS RP
 | ID ARRS
 | ID DOT ID
 | INT
 ;
ARRS : LB EXP RB ARRS
 |
 ;
ARGS: EXP COMMA ARGS
 | EXP
 ;

%%
#include "lex.yy.c"
int main(int argc, char *argv[]){
 yyin = fopen(argv[1], "r");
            if (yyin == 0)
            {
                  fprintf(stderr, "failed to open %s for reading\n", argv[1]);
                  exit(1);
            }
            yyparse();
            fclose(yyin);
 return 0;
}

/*Print the error*/
static void print_tok(){
 if(yychar<255){
       fprintf(stderr, "%c", yychar);
 }
 else{
       switch (yychar){
```

```
case SEMI:    {
        fprintf(stderr," ;\n");
        break;
}
case COMMA:    {
        fprintf(stderr," ,\n");
        break;
}
case LP:    {
        fprintf(stderr," (\n");
        break;
}
case RP:    {
        fprintf(stderr," )\n");
        break;
}
case LB:    {
        fprintf(stderr," [\n");
        break;
}
case RB:    {
        fprintf(stderr," ]\n");
        break;
}
case LC:    {
        fprintf(stderr," {\n");
        break;
}
case RC:    {
        fprintf(stderr," }\n");
        break;
}
case DOT:    {
        fprintf(stderr," .\n");
        break;
}
case LOGICNOT:    {
        fprintf(stderr," !\n");
        break;
}
case PREINCRE:    {
        fprintf(stderr," ++\n");
        break;
}
```

```
case PREDEC:    {
        fprintf(stderr," --\n");
        break;
}
case BITNOT:    {
        fprintf(stderr," ~\n");
        break;
}
case PRODUCT:    {
        fprintf(stderr," *\n");
        break;
}
case DIVISION:    {
        fprintf(stderr," /\n");
        break;
}
case MODULUS:    {
        fprintf(stderr," %%\n");
        break;
}
case PLUS:    {
        fprintf(stderr," +\n");
        break;
}
case MINUS:    {
        fprintf(stderr," -\n");
        break;
}
case SHIFTLEFT:    {
        fprintf(stderr," <<\n");
        break;
}
case SHIFTRIGHT:    {
        fprintf(stderr," >>\n");
        break;
}
case GREATERT:    {
        fprintf(stderr," >\n");
        break;
}
case LESST:    {
        fprintf(stderr," <\n");
        break;
}
```

```
case NOTLESST:    {
        fprintf(stderr," >=\n");
        break;
}
case NOTGREATERT:    {
        fprintf(stderr," <=\n");
        break;
}
case EQUAL:    {
        fprintf(stderr," ==\n");
        break;
}
case NOTEQUAL:    {
        fprintf(stderr," !=\n");
        break;
}
case BITAND:    {
        fprintf(stderr," &\n");
        break;
}
case BITXOR:    {
        fprintf(stderr," ^\n");
        break;
}
case BITOR:    {
        fprintf(stderr," |\n");
        break;
}
case LOGICAND:    {
        fprintf(stderr," &&\n");
        break;
}
case LOGICOR:    {
        fprintf(stderr," ||\n");
        break;
}
case ASSIGN:    {
        fprintf(stderr," =\n");
        break;
}
case PLUSASSIGN:    {
        fprintf(stderr," +=\n");
        break;
}
```

```
case MINUSASSIGN:    {
        fprintf(stderr," -=\n");
        break;
}
case PRODUCTASSIGN:    {
        fprintf(stderr," *=\n");
        break;
}
case DIVISIONASSIGN:    {
        fprintf(stderr," /=\n");
        break;
}
case ANDASSIGN:    {
        fprintf(stderr," &=\n");
        break;
}
case NORASSIGN:    {
        fprintf(stderr," ^=\n");
        break;
}
case ORASSIGN:    {
        fprintf(stderr," |=\n");
        break;
}
case SLASSIGN:    {
        fprintf(stderr," <<=\n");
        break;
}
case SRASSIGN:    {
        fprintf(stderr," >>=\n");
        break;
}
case TYPE:    {
        fprintf(stderr," int\n");
        break;
}
case STRUCT:    {
        fprintf(stderr," struct\n");
        break;
}
case RETURN:    {
        fprintf(stderr," return\n");
        break;
}
```

```
            case IF:    {
                fprintf(stderr," if\n");
                break;
            }
            case ELSE:    {
                fprintf(stderr," else\n");
                break;
            }
            case BREAK:    {
                fprintf(stderr," break\n");
                break;
            }
            case CONT:    {
                fprintf(stderr," continue\n");
                break;
            }
            case FOR:    {
                fprintf(stderr," for\n");
                break;
            }
            case ID:    {
                fprintf(stderr," %s\n", yytext);
                break;
            }
            case INT:    {
                fprintf(stderr," %s\n", yytext);
                break;
            }
            case UNKNOWN:    {
                fprintf(stderr," %s\n", yytext);
                break;
            }
            default:{

            }
        }
 }
}

/*Error handling*/
int yyerror(char* s){
 fprintf(stderr,"[line %d]:%s",linecount+1,s);
 print_tok();
```

```
}
```

4. **makefile**

```
LEX=lex

YACC=yacc

CC=gcc

program: y.tab.c
 $(CC) y.tab.c -ly -ll -o program

smallc.yy.o:smallc.yy.c y.tab.h
 $(CC) -c smallc.yy.c

y.tab.c y.tab.h:smallc.y smallc.yy.c
 $(YACC) -v -d smallc.y

smallc.yy.c:smallc.l
 $(LEX) smallc.l

clean:
	rm -f *.o *.c *.h
```

5. **test file**

```
int main()
{
	i=0;
	j=1;
	for(;i<10;++i)
	{
		j=j*i;
	}
	return 0;
 }
```