Indian Institute of Information Technology, Vadodara

Instructor : Dr. Ashish Phophalia

Core#

Compiler Design Project Report

(CS- 307)

Team Members

Neetu Shah	201451015
Ritika Nath	201451020
Shreeja Injapuri	201451023
Shweta Tiwari	201451050
K Naveen Kumar	201451074

Table Of Contents

1. **Abstract** ……3

2. **Keywords** ……4

3. **Special symbols** ……6

4. **Grammar** ……7

5. **Parsing using LEX+YACC** …..12

6. **Symbol Table and Semantic Analysis** …..14

7. **Three address code** …..23

8. **Conclusion** …..32

9. **References** …..33

1. Abstract

Our compiler “Core#” has been designed based on C language and as the name denotes, it consists of the core qualities and functionalities of C and its structure is inspired by the LEX and YACC programs. Features like the parenthesis(brackets) have been eliminated with new features or keywords in order to minimize the errors and make the coding easier. The necessary keywords and constraints in our design are clearly specified. All the required grammars including statements, loops, arithmetic expressions, arrays, functions are defined in the report as the first stage for our designing.
Further parsing using LEX+YACC using symbol tables, semantics and three address code are also mentioned in this as a part of the final stage.

2. Keywords

---

show	:	To print or display any statement on the terminal.
take	:	To read or scan the required input.
stop	:	To end a loop, conditional statement and function.
if	:	Conditional statement.
elif	:	“else if” in conditional statement.
else	:	Conditional statement.
cut	:	To denote ‘break’ or break the statement.
int	:	Integer data type.
float	:	Float data type.
double	:	Long float.
char	:	character data type.
bool	:	Boolean data type.
return	:	To return value or expression.
for	:	For loop.

while : While loop.

do : do-while loop.

switch : switch condition.

case : cases in the switch.

default : Default case.

3. Special Symbols

; - **Single line comment.**

;;.....;; - **Multiline comment.**

: : - **Used in function declaration.**

- **Separator.**

\n - **Next line.**

\t - **Tab.**

4. Grammar

Part 1

Declaration :

<program> ----> <declaration> \n# <body> \n # <functions> | <comment>

<declaration> ----> <vardec> , <declaration> | <arrdec>,<declaration>

| Null

<vardec> ----> <datatype><s><id>

<s> -----> [ ] | \t

<datatype> ----> int | double | float | char | bool

<id> ----> <id><digit> | <nondigit><id> | <nondigit>

<digit> ----> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<nondigit> ----> a | b | c | ..... | z | A | B | C | ..... | Z |

(In Array declaration everything before ' , ' is same as variable declaration. A after ' , ' indicates that its an array declaration . Input is taken from user for defining size of the array.)

<arrdec> ----> <datatype><id>,<A>

<A> ----> A take<constant>

<constant> ----> <digit><constant> | <digit>

Part 2 :

Body :

<body> ----> <s>
<stmts> ----> <condstmts>\n | <loopstmts>\n | <io> \n | <expr> \n |

<functioncall>\n | <switch>\n

(Grammar representing if-else)

<condstmts> ----> if <expr> then <stmts> stop <condstmts> |

elif <expr> then <stmts><condstmts> |

if<expr>then<stmts>else<stmts>stop<condstmts> |

<stmts>|Null

(Loops including for, while, do-while)

<loopstmts> ----> for <expr>,<expr>,<expr>\n<stmts>\n stop\n

<loopstmts> |

while <expr>\n<stmts>\nstop\n<loopstmts> |

do\n<stmts>\n while <expr>\n<loopstmts> | <stmts>

| Null

(switch case)

<switch> ----> switch<id>:\n<case><default>\n<stop>\n<switch> |

<stmts> | Null

<case> ----> case<id>: \n \t <stmts> \n \t cut \n <case> | Null

<default> ----> default : <stmts> cut

(Arithmetic expressions)

<expr> ----> <id> <expr> | <constant> | (<expr>) | <cond_expr> |

<assn_expr> | <strings> | <stmts> | Null

<cond_expr> ----> <expr><op1><expr>\n<cond_expr> | <expr> |

Null

<assn_expr>----><expr>=<expr>\n<assn_expr> |

<expr>=<expr><op2><expr>\n<assn_expr> |

<expr>,A=<expr><op2><expr>\n<assn_expr> |

<expr> | Null

<strings>---->"<id><strings>" | <id><strings> | "<digit><strings>" |

\n<strings> | Null

<op1>---> == | > | < | >= | <= | != | <op2>

<op2>---> + | - | / | * | %

<io> ----> take <id>\n<io> | take<id>,<id> | show<expr>\n<io> |

show<id>,<id> | <stmts> | Null

(Function calls inside the body)

<function_call> ----> <fun_name> : <parameter_list> : \n <function_call>

| <stmts> | Null

<fun_name> ----> <id> <strings> | Null

<parameter_list> ----> <id>,<parameter_list> | Null

Part 3 :

Functions:

<functions> ----> <return_type> <fun_name> : <parameters> :\n

<stmts> \n <returns> \n stop \n <functions> | Null

<returns> ----> return <expr> | return<id> | Null

<parameters> ---> <parameter>,<parameters> | <parameter> | Null

<parameter> ---> <datatype> <parameter_id>

<parameter_id> ---> <id>

<return_type> ---> int | float | char | double | void

Comment Section:

<comments> ----> ; <stringz><comment> |

;;<multiline>;;<comment> | <program> | <body>

| <functions> | Null

<multiline> ----> <strings> \n <multiline> | Null

5. Parsing using LEX+YACC

Implementation of our grammar in LEX and YACC:

Defining the tokens

```
%token      SET SHOW EQUAL SWITCH PLUS MINUS STATE MODE
            MULT DIV GET STRING EOLN FUN_STOP HASH NUM
            DOUBLEQ INT FLOAT DOUBLE SPACE
%token      COMMA IF THEN ELSE STOP ELIF FOR WHILE DO A
            COLON SEMICOLON CASE CUT DEFAULT BRO BRC
            BOOL CHAR
%right      '='
%left       AND OR
%left       '<' '>' LE GE EQ NE LT GT
```

Defining keywords

```
show            return SHOW;
int             return INT;
float           return FLOAT;
double          return DOUBLE;
char            {return CHAR;}
bool            {return BOOL;}
switch          return SWITCH;
case            return CASE;
default         return DEFAULT;
cut             return CUT;
```

Files Related to this section:

Names : core#.l
core#.y

ToRun : lex core#.l
yacc core#.y
gcc lex.yy.c -ll -ly
./a.out test_file_name

test_file_name : expr.txt
for_loop.txt
function.txt
Hello.txt
if_else.txt
switch_case.txt

6. Symbol Table and Semantic Analysis

Symbol Table Generation:

Algorithm :-

```
void update symbolVal(symbol,val)
{
        bucket = computeSymbolIndex(symbol)
        symbols[bucket] = val
}

int symbolVal(symbol)
{
        bucket = computeSymbolIndex(symbol)
        return symbols(bucket)
}

int computeSymcbolIndex(token)
{
        idx = -1
        if(islower(token))
        {
                idx = token_'a'+26
        }
        elseif(isupper(token))
        {
```

```
            idx = token)'A'
        }
        return idx
}
```

Example:

Sample Code:

```
a = 10
b = 15
c = a + b
d = a - b
f = c - d
```

Update symbolVal

a	10
b	15
c	a + b
d	a - b
f	c - d

Suppose the function **computeSymbolIndex** generates indexes for following tokens

```
a -> 1
b -> 2
c -> 3
d -> 4
f -> 5
```

1	10
2	15
3	$1+$2 = 25
4	$2-$2 = -5
5	$3-$4 = 30

Semantic Analysis:

1. Arithmetic operations. (+ , - , * , / , pow)
2. if else with conditions. (< , > , <= , >= ,==)
3. show numbers.
4. while and for loops with conditions.(< , > , <= , >= , ==)
5. Checking for undeclared variable.

1. Arithmetic Operations:

Using Symbol Table as mentioned in symbol table generation.

2. if else with conditions. (< , > , <= , >=)

Algorithm:

flag = 11 (take any temporary variable and assign some random value)

Semantics:

```
cond_expr --> cond_expr LE cond_expr
                    {
                        if symbolVal($1) < symbolVal($1)
                        {
                            flag = 1 ;
                    }
```

```
                else
                        flag = 101 ;
            }

        | cond_expr GE cond_expr
                {
                        if symbolVal($1) > symbolVal($3)
                        {
                                flag = 1 ;
                        }
                        else
                                flag = 101 ;
                }
```

[Similar semantics for <= , >= , ==]

```
        | id { $$ = $1 }
        | num { $$ = $1 }
```

cond_stmt : if cond_expr then exec1
else exec2
stop
;

exec1 : show expr {
        if (flag == 1)
                printf(expr);
}

exec2 : show expr {
        if (flag == 101)
                printf(expr);
}

3. Show Numbers:

Using symbol table as mentioned in symbol table generation.

4. While and For loops:

We made two functions -

Increment

Decrement

Algorithm :

```
int increment(symbol)
{
        bucket = computeSymbolIndex(symbol);
        c = symbol(bucket);
        c = c+1;
        return c;
}

int decrement(symbol)
{
        bucket = computeSymbolIndex(symbol);
        c = symbol(bucket);
        c = c-1;
        return c;
}
```

Semantics for loop statements

loopstmts : for assignment comma identifier LE expr
{ printf (symbolVal($4); }

comma assignment
{ while (symbolVal($4) < $6) {
print(symbolVal($2);
val = increment($4);
updatesymbolVal($4, val);
} ;
}

[Similar semantics for <=]

| for assignment comma identifier GE expr
{ print(symbolVal($4) } ;

comma assignment
{ while (symbolVal($4)>($6)){
print(symbolVal($2);
val = decrement($4);
updatesymbolVal($4, val);
};
}

[Similar semantics for >=]

[Similar semantics for evaluating while loop]

5. Checking for undeclared variable:

Algorithm:

```
int declcheck(char symbol)
{
        bucket = computeSymbolIndex(symbol);
        return Symbolexist(bucket);
}

void decval(symbol,int val)
{
        bucket = computeSymbolIndex(symbol);
        Symbolexist(bucket) = val;
}
```

declarations : declarations datatype identifier comma
{ decVal($3, 1) ; }

assignment : id = expr
{ if (declcheck($1) == 1) {
-------do----------
}
else {
print(Undeclared variable)
}

Example :

int a,int b,int c

Suppose index computed for symbols is

a->1
b->2
c->3

Symbol Index	Checkval
1	1
2	1
3	1

Body :

a = 10
b = 10
c = 15
d = 20

Algorithm:

{ if declcheck($1) ==1}
----------do----------
else
print(undeclared variable)

declcheck(a) = declcheck($1)=1
declcheck(b) = declcheck($2)=1
declcheck(c) = declcheck($3)=1

declcheck(d) = declcheck(d) !=1
print(undeclared variable)

Output: undeclared variable

Files Related to this section:

Names : core#1.l
core#1.y

Run : lex core#1.l
yacc core#1.y
gcc lex.yy.c -ll -ly -lm
./a.out test_file_name

test_file_name : arith_semantic.txt
for_semantic.txt
function_semantic.txt
if_else_semantic.txt

7. Three address code

Algorithm :

```
   char * newTemp(){
   char *newTemp = (char *)malloc(20);
   strcpy(newTemp,"t");
   snprintf(num_to_concatinate, 10,"%d",n);
   strcat(newTemp,num_to_concatinate);

   n++;
   return newTemp;
}

char * newLabel(){

   char *newLabel = (char *)malloc(20);
   strcpy(newLabel,"L");
   snprintf(num_to_concatinate_l, 10,"%d",nl);
   strcat(newLabel,num_to_concatinate_l);

   nl++;
   return newLabel;
}

construct : WHILE  bool  block STOP
        {
                printf("Inside WHILE\n");
                puts($3);
```

```
b1 = $2;
s1 = $3;

begin = newLabel();
label = newLabel();

check = strstr (b1,"TRUE");

while(check!=NULL){
        strncpy (check,label,strlen(label));
        strncpy (check+strlen(label),"       ",(4-strlen(label)));
        check = strstr (b1,"TRUE");
        }

check = strstr (b1,"FAIL");

while(check!=NULL){
        strncpy (check,"NEXT",4);
        //strncpy (check+strlen(label),"     ",(4-strlen(label)));
        check = strstr (b1,"FAIL");
        }

check = strstr (s1,"NEXT");

while(check!=NULL){
        strncpy (check,begin,strlen(begin));
        strncpy (check+strlen(begin),"       ",(4-strlen(begin)));
        check = strstr (s1,"NEXT");
        }

ret = (char *)malloc(strlen(b1)+strlen(s1)+20);
ret[0] = 0;
strcat(ret,begin);
```

```
        strcat(ret," : ");
        strcat(ret,b1);
        strcat(ret,"\n");
        strcat(ret,label);
        strcat(ret," : ");
        strcat(ret,s1);

        strcat(ret,"\n");
        strcat(ret,"goto ");
        strcat(ret,begin);

        printf("Final return from while\n");
        puts(ret);

        $$ = ret;

 }
|
 IF  bool  block STOP
 {
        printf("Inside IF\n");

        label = newLabel();
        b1 = $2;

        check = strstr (b1,"TRUE");

        while(check!=NULL){
                strncpy (check,label,strlen(label));
                strncpy (check+strlen(label),"      ",(4-strlen(label)));
                check = strstr (b1,"TRUE");
                }
```

```
		check = strstr (b1,"FAIL");

		while(check!=NULL){
			strncpy (check,"NEXT",4);
			//strncpy (check+strlen(label),"    ",(4-strlen(label)));
			check = strstr (b1,"FAIL");
			}

		ret = (char *)malloc(strlen(b1)+strlen($3)+4);
		ret[0] = 0;
		strcat(ret,b1);
		strcat(ret,"\n");
		strcat(ret,label);
		strcat(ret," : ");
		strcat(ret,$3);

		puts(ret);
		$$ = ret;
	}
statement : text '=' expr
	{
	printf("Assignment statement \n");

	 to_return_expr = (struct exprType *)malloc(sizeof(struct
exprType));
	 to_return_expr->addr = (char *)malloc(20);
	 to_return_expr->addr = newTemp();

	 ret = (char *)malloc(20);
	 ret[0] = 0;

	 strcat(ret,$1);
```

```
        strcat(ret,"=");
        strcat(ret,$3->addr);
        printf("RET  = \n");
        puts(ret);

        temp = (char *)malloc(strlen($3->code)+strlen(ret)+6);

        temp[0] = 0;

        if ($3->code[0]!=0){
                strcat(temp,$3->code);
                strcat(temp,"\n");
                }
        strcat(temp,ret);
        printf("TEMP = \n");

        puts(temp);

        to_return_expr->code = temp;

                $$ = to_return_expr;


        //printf(" %s = %s \n",$1,$3->addr);
       }
       |
       expr '+' expr
       {
       printf("Addition : ");
        to_return_expr = (struct exprType *)malloc(sizeof(struct
exprType));
        to_return_expr->addr = (char *)malloc(20);
        to_return_expr->addr = newTemp();
```

```
        ret = (char *)malloc(20);
        ret[0] = 0;

        strcat(ret,to_return_expr->addr);

        strcat(ret,"=");
        strcat(ret,$1->addr);
        strcat(ret,"+");
        strcat(ret,$3->addr);
        printf("RET  = \n");
        puts(ret);

        temp =
(char*)malloc(strlen($1->code)+strlen($3->code)+strlen(ret)+6);

        temp[0] = 0;

        if ($1->code[0]!=0){
                strcat(temp,$1->code);
                strcat(temp,"\n");
                }
        if ($3->code[0]!=0){
                strcat(temp,$3->code);
                strcat(temp,"\n");
                }
        strcat(temp,ret);
        printf("TEMP = \n");

        puts(temp);

        to_return_expr->code = temp;
```

```
                $$ = to_return_expr;
        }

text {
        printf("Inside text\n");
        to_return_expr = (struct exprType *)malloc(sizeof(struct
exprType));
        to_return_expr->addr = (char *)malloc(20);
        to_return_expr->addr = $1;

        to_return_expr->code = (char *)malloc(2);
        to_return_expr->code[0] = 0;

        $$ = to_return_expr;}
        |
        number {
        printf("Inside Number\n");
        to_return_expr = (struct exprType *)malloc(sizeof(struct
exprType));
        to_return_expr->addr = (char *)malloc(20);
        to_return_expr->addr = $1;

        to_return_expr->code = (char *)malloc(2);
        to_return_expr->code[0] = 0;

        $$ = to_return_expr;}
        ;

number:  DIGIT
        {
        printf("Inside DIGIT : %d\n",$1);
        var = (char *)malloc(20);
                snprintf(var, 10,"%d",$1);
```

```
        $$ = var;

        }
    |
        FLOAT
        {
         printf("Inside FLOAT : %f\n",$1);
         var = (char *)malloc(20);
                snprintf(var, 10,"%f",$1);
         $$ = var;

        }
```

Sample Example :

```
int a, int b, int c, int x, int y,
#
a=10
b=15
c=a+b
while x>2
        if a>b
                x=x+y
        else
                y=y+1
        stop
stop
#
```

Three address code :

```
a = 10
b = 15
t3 = a + b
c = t3
L3 : if(x < 2) goto L4
goto L6
L4 : if(a < b) goto L1
goto L2
L1 : t7 = x+y
x = t7
goto L3
L2 : t8 = ty+1
y = t8
goto L3
L6 : End of Three Address Code
```

Files Related to this section:

Names : core#2.l
core#2.y

Run : lex core#2.l
yacc core#2.y
gcc lex.yy.c -ll -ly
./a.out test_file_name

test_file_name : test
test1
test2

8. Conclusion

There were number of issues faced during the making of this project. Simple problems like shift-reduce and reduce-reduce conflicts was a tough challenge and we successfully minimized as many as possible.
On the whole, the project outcome came out better than expected and it was a good opportunity to actually implement the whole “Compiler Design” practically.

9. References

1.https://www.tutorialspoint.com/compiler_design/compiler_design_quick_g

uide.htm

2.http://symbolaris.com/course/Compilers/waitegoos.pdf

3. http://www.wikihow.com/Create-a-Programming-Language