A SIMPLE SUB-C COMPILER USING L LEX AND YACC

TABLE OF CONTENTS:

* Introduction
* Lex and Yacc
* Three Address Code
* Problem Statement
* Problem Decomposition
* Algorithm
* Lex Code
* Explanation of Lex code
* Yacc code
* Grammar rules
* Sample Test Cases
* Assumptions
* Limitations

SUBMITTED BY:

G.Avinash

Roll No: 157118

B.Tech 3rd Year

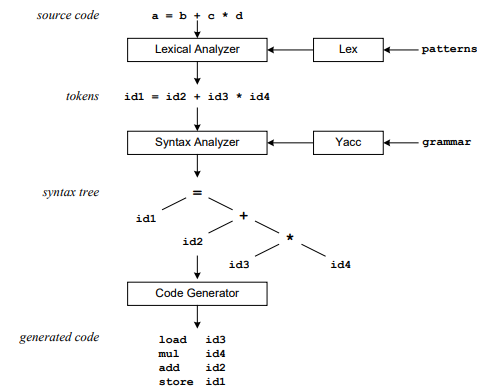
CSE Sec-A

Compiler Using Lex and Yacc

**Compiler** is a computer software that translates (compiles) source code written in a high-level language (e.g., C++) into a set of machine-language instructions that can be understood by a digital computer’s CPU. Compilers are very large programs, with error-checking and other abilities. Some compilers translate high-level language into an intermediate assembly language, which is then translated (assembled) into machine code by an assembly program or assembler. Other compilers generate machine language directly.

A compiler is likely to perform many or all of the following operations: pre-processing, lexical analysis, parsing, semantic analysis (syntax-directed translation), conversion of input programs to an intermediate representation, code optimization and code generation. Compilers implement these operations in phases that promote efficient design and correct transformations of source input to target output. Program faults caused by incorrect compiler behaviour can be very difficult to track down and work around; therefore, compiler implementers invest significant effort to ensure compiler correctness.

When executing, the compiler first parses all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code. Traditionally, the output of the compilation has been called *object code* or sometimes an *object module*. The object code is machine code that the processor can execute one instruction at a time.



L**ex** and Yacc:

In this project, we use Lex and Yacc to create a compiler.

A compiler or interpreter for a programming language is often decomposed into two parts:

1. Read the source program and discover its structure.
2. Process this structure, e.g. to generate the target program.

*Lex* and *Yacc* can generate program fragments that solve the first task.

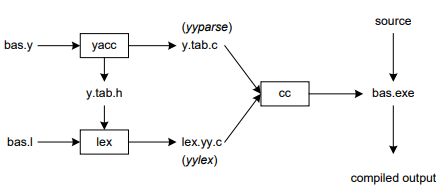
The task of discovering the source structure again is decomposed into subtasks:

1. Split the source file into tokens (*Lex*).
2. Find the hierarchical structure of the program (*Yacc*).

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.



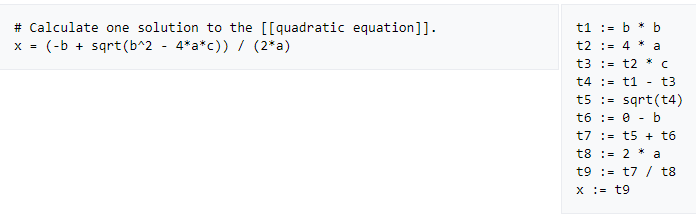
Three Address Code:

Three Address Code is an intermediate code used by optimizing compilers to aid in the implementation of code-improving transformations. Each TAC instruction has at most three operands and is typically a combination of assignment and a binary operator. For example, t1 := t2 + t3. The name derives from the use of three operands in these statements even though instructions with fewer operands may occur.

Since three-address code is used as an intermediate language within compilers, the operands will most likely not be concrete memory addresses or processor registers, but rather symbolic addresses that will be translated into actual addresses during register allocation. It is also not uncommon that operand names are numbered sequentially since three-address code is typically generated by the compiler.

Intermediate code generator receives input from its predecessor phase, semantic analyser, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of registers to generate code.

Example for three Address Code:



Problem Statement:

The Sub C language taken includes the following:

|  |  |
| --- | --- |
| Data Types | int, unsigned int, Boolean |
| Assignment Operators | =, +=, -=, \*=, /= |
| Binary Operators | + ,- ,\* ,/ ,@(exponentiation) |
| Bitwise Operations | |, &, ~, ^ |
| Logical Operations | || , && , ! |
| Relational Operations | == , <= , >= , > , < , != |
| Conditional Statements | if statement, if\_else statement, switch |
| Repetitive Statements | While loop |

Identifiers Should support simple ids

Now we have to generate three address code for C statements supporting the above operations and statements.

Problem Decomposition:

The programs are written in Lex and Yacc program. Lex helps you by taking a set of description of possible tokens and producing a C routine, which we call as lexical analyser or a lexer, that identify the tokens. As input is divided into tokens, a program often needs to establish the relationship among the tokens. A compiler needs to find the expression, statement, declaration, blocks and procedure in the program. This task is known as parsing and the list of rules that define the relationship that the program is a grammar. Yacc takes a concise description of a grammar and produces a C routine that can parse the grammar, a parser. The Yacc parser automatically detects whenever a sequence of input token matches one of the rule of the grammar and also detects a syntax error whenever its input does not match any of the rule. When a task involves dividing the input into units and establishing some relationship among those units then we should use Lex and Yacc.

The project is divided into three phases. Development, Control, Testing.

In the development phase first the variables to be used in the code are declared and then those variables are initialised. All the code is written in this phase.

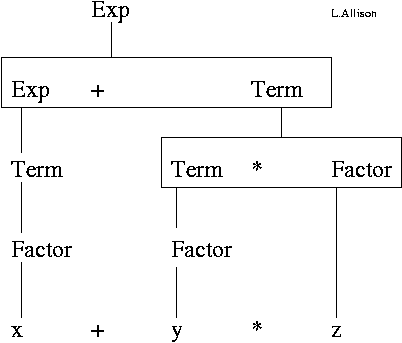
In the Control Phase we observe how different units interact with each other and check if it happens properly. If the different units do not interact properly we go back to development phase and fix the issues

In the testing phase, we give a sample input and check if the output matches the required output.

Algorithm:

We need to first declare variables. The code will give a syntax error if any undeclared variables are used. The variables are then initialised to a certain value. Int can be initialised to any integer value whereas Boolean values can be initialised to true or false.

We use the following grammar for the evaluation of integer expressions



Addition and Subtraction operations have the same priority and Multiplication and Division operations have the same priorities.

Exponentiation has a priority higher than that of Multiplication/Division.

Addition, Subtraction, Multiplication, Division operations are left associative whereas Exponentiation is right associative.

We implement a similar kind of grammar for the evaluation of Boolean expressions where Boolean “and” has a higher priority than Boolean “or” and Boolean “not” has a higher priority than both of them.

For the implementation of the conditional if..then.. statement we use a Boolean expression for condition statement and then part can include any kind of statement inside it so we use the start symbol for then part.

The Conditional if..then..else.. is implemented in the same way as the conditional if statement and the else part also has a start symbol inside.

The conditional

While loop is implemented similar to the conditional if statement but we jump to the start of the block at the end of the execution of block of statements in the while loop

Syntax directed translations that can be implemented during parsing can be characterized by introducing distinct marker non terminals in place of each embedded action; each marker M has only one production, M-> Є.

We use markers for the labels and for the jump and goto statements as they should be executed before the block of statements which should be executed after the Boolean evaluation.

Code:

Compiler.l

%{

#include<string.h>

#include "compiler.tab.h"

#include<stdio.h>

%}

**letter** [A-Za-z]

**digit** [0-9]

**id** {letter}({letter}|{digit})\*

**number** [digit]+

**datatype** "int"|"bool"|"unsigned int"

**boolValue** "true"|"false"

**relop** "<"|">"|"<="|">="|"=="|"!="

%%

{datatype} {**strcpy**(yylval.str,yytext); return types;}

{boolValue} {**strcpy**(yylval.str,yytext); return bval;}

"while" {**strcpy**(yylval.str,yytext); return WHILE;}

"if" {**strcpy**(yylval.str,yytext); return IF;}

"else" {**strcpy**(yylval.str,yytext); return ELSE;}

"switch" {**strcpy**(yylval.str,yytext); return SWITCH;}

"case" {**strcpy**(yylval.str,yytext); return CASE;}

"break" {**strcpy**(yylval.str,yytext); return BREAK;}

"default" {**strcpy**(yylval.str,yytext); return DEFAULT;}

{id} {**strcpy**(yylval.str,yytext); return ID;}

{relop} {**strcpy**(yylval.str,yytext); return RELOP;}

[;:={}()!] {/\***printf**("L %s\n", yytext);\*/ return yytext[0];}

[0-9]+ { **strcpy**(yylval.str,yytext); return ival;}

"&&" {return BAND;}

"||" {return BOR;}

"+" {return ADD;}

"-" {return SUB;}

"\*" {return MUL;}

"/" {return DIV;}

"|" {return OR;}

"^" {return XOR;}

"&" {return AND;}

"@" {return POW;}

"~" {return NEG;}

\n {}

[' '] {}

%%

The yyparse() function in Yacc parses the input based on the rules given in the lex file.

In this Lex program, datatype denotes all the different datatypes identified by the parser i.e int, unsigned int, bool.

id denotes the name of the identifier which should start with a letter and can contain digits in it.

We assign specific tokens for each logical and Boolean operations so that each individual operation can be identified individually.

ADD for +, SUB for -, MUL for \*, DIV for /,

BOR for ||, BAND for &&, XOR for ^

Our compiler also has a token for exponentiation. We use the symbol ‘@’ and the token POW for it.

We use the tokens IF, ELSE, SWITCH, WHILE, DEFAULT, CASE, BREAK when the input matches with “if”, “else”, “while”, “default”, “case” and “break” respectively.

“relop” matches for all Boolean relational operations like >, >=, <, <=, !=, == and returns the token

Lex does not automatically skip over white space – in some languages white space is important and lex needs to be able to process those languages as well as languages where whitespace is ignored. Thus, we need to tell lex what to ignore. We can make lex ignore whitespace by adding rules that match whitespace that have no actions, as follows:

\n {}

[' '] {}

Compiler.y:

%{

#include<stdio.h>

#include<bits/stdc++.h>

using namespace **std**;

struct **variable**{

string type;

string name;

string value;

};

vector<variable> table;

int **yylex**(void);

void **yyerror**(char \*);

void **printTable**();

void **intialiseValue**(char\*, char\*);

int **findVal**(char\*);

string **threeAddress**(char\*, char\*, string);

string **threeAddress**(char\*, string);

void **threeAddressSelf**(char\*, char\*, string);

void **checkID**(char\*);

int tcount=0, label\_count=0, d=0, s\_count=0;

string result = "", bresult = "";

map<int, string> tvalues;

vector<int> repeat;

stack<int> s;

%}

%union

{

int number;

char str[100];

}

%token <str> types ID RELOP ival bval WHILE boolValue IF ELSE SWITCH BREAK DEFAULT CASE

%token <number> intVal

%token ADD SUB MUL DIV ABS NEG BAND BOR

%left AND OR XOR

%right POW

%type <str> declarations datatype identifier initialisation

%type <str> eval bitor bitxor bitand exp factor powterm term bitneg

%type <str> booleval bool\_and bool\_not bool\_rel

%nonassoc ELSE IFX

%%

S:

| S declarations {**printTable**();}

| S initialisation {**printTable**();}

| S eval {cout<<result<<endl; result =""; **printTable**(); tcount =0; tvalues.**clear**();}

| S booleval {cout<<bresult<<endl; result ="";tcount =0; tvalues.**clear**();}

| S boolassign {cout<<bresult<<endl; result ="";tcount =0; tvalues.**clear**();}

| S cond\_while {}

| S cond\_if {}

| S cond\_if\_else {}

| S cond\_switch {}

| '{' S '}' {}

;

cond\_while: WHILE '(' M1 booleval M2 ')' '{' S M3 '}' {cout<<"mid"<<s.**top**()<<":"<<endl;s.**pop**();d++;tvalues.**clear**();}

;

cond\_if: IF '(' M1 booleval M2 ')' '{' S '}' %prec IFX  { cout<<"mid"<<s.**top**()<<":"<<endl;s.**pop**();}

;

cond\_if\_else: IF '(' M1 booleval M2 ')' '{' S '}' M4 ELSE M5 '{' S '}'  {cout<<"end"<<s.**top**()<<":"<<endl;s.**pop**(); }

;

M1: {cout<<"start"<<label\_count<<":\n";s.**push**(label\_count);label\_count++;}

;

M2: {cout<<"jump !t"<<tcount-1<<" "<<"mid"<<s.**top**()<<endl;}

;

M3: {cout<<"jump start"<<s.**top**()<<"\n";}

;

M4: {cout<<"jump end"<<s.**top**()<<"\n";}

;

M5: {cout<<"mid"<<s.**top**()<<":\n";}

;

cond\_switch:  SWITCH '(' booleval M11 ')' '{' cases '}' { cout<<"last:"<<s\_count<<endl;s\_count++;}

;

cases:

| M44 CASE booleval ':' M22 S BREAK M33 ';' cases {}

| DEFAULT ':' S {}

;

M11:  {cout<<"k=t"<<tcount-1<<endl;}

;

M22:  {cout<<"jump !k==t"<<tcount-1<<" end"<<s.**top**()<<endl;}

;

M33:  {cout<<"jump last"<<s\_count<<"\n";cout<<"end"<<s.**top**()<<":"<<endl;s.**pop**();}

;

M44:  {cout<<"start"<<label\_count<<":\n";s.**push**(label\_count);label\_count++;}

;

boolassign: ID '=' booleval ';'{ **checkID**($1);

int c1;

string **s1**($3);

for(map<int, string> ::iterator itr = tvalues.**begin**(); itr!=tvalues.**end**(); itr++){

if(itr->second == s1) c1 = itr->first;

}

cout<<$1<<" = t"<<c1<<endl;

  }

;

booleval:bool\_and

| booleval BOR bool\_and { **strcpy**($$,((**threeAddress**($1, $3, "||")).**c\_str**())); }

;

bool\_and:bool\_not

| bool\_and BAND bool\_not {**strcpy**($$,((**threeAddress**($1, $3, "&&")).**c\_str**()));}

;

bool\_not:bool\_rel

| '!' bool\_not {

**strcpy**($$,((**threeAddress**($2, "!")).**c\_str**()));}

;

bool\_rel:bval{**strcpy**($$,$1); string s1=**string**($1); tvalues[tcount]=s1; cout<<"t"<<tcount++<<" = "<<$$<<endl;}

| term RELOP term { string **relop**($2); **strcpy**($$,((**threeAddress**($1, $3, relop)).**c\_str**()));}

| term {**strcpy**($$,$1);}

;

eval: ID '=' exp ';'{ **checkID**($1);

int c1;

string **s1**($3);

for(map<int, string> ::iterator itr = tvalues.**begin**(); itr!=tvalues.**end**(); itr++){

if(itr->second == s1) c1 = itr->first;

}

cout<<$1<<" = t"<<c1<<endl;

repeat.**clear**();

  }

| ID ADD '=' exp ';'{ **checkID**($1);

**threeAddressSelf**($1,$4," + ");

  }

| ID SUB '=' exp ';'{ **checkID**($1);

**threeAddressSelf**($1,$4," - ");

  }

| ID MUL '=' exp ';'{ **checkID**($1);

**threeAddressSelf**($1,$4," \* ");

  }

| ID DIV '=' exp ';'{ **checkID**($1);

**threeAddressSelf**($1,$4," / ");

  }

| ID POW '=' exp ';'{ **checkID**($1);

**threeAddressSelf**($1,$4," @ ");

  }

;

exp: factor

| exp ADD factor {

**strcpy**($$,((**threeAddress**($1, $3, "+")).**c\_str**()));}

| exp SUB factor {

**strcpy**($$,((**threeAddress**($1, $3, "-")).**c\_str**()));}

;

factor: powterm

| factor MUL powterm {

**strcpy**($$, ((**threeAddress**($1, $3, "\*")).**c\_str**()));}

| factor DIV powterm {

**strcpy**($$,((**threeAddress**($1, $3, "/")).**c\_str**()));}

;

powterm:bitor

| bitor POW powterm {

**strcpy**($$,((**threeAddress**($1, $3, "@")).**c\_str**()));

}

;

bitor:bitxor

| bitor OR bitxor {

**strcpy**($$,((**threeAddress**($1, $3, "|")).**c\_str**()));}

;

bitxor:bitand

| bitxor XOR bitand {

**strcpy**($$,((**threeAddress**($1, $3, "^")).**c\_str**()));}

;

bitand:bitneg

| bitand AND bitneg {

**strcpy**($$,((**threeAddress**($1, $3, "&")).**c\_str**()));}

;

bitneg:term

| NEG bitneg {

**strcpy**($$,((**threeAddress**($2, "~")).**c\_str**()));}

;

term: ival { **strcpy**($$,$1); string s1=**string**($1); tvalues[tcount]=s1; cout<<"t"<<tcount++<<" = "<<$$<<endl; */\*stringstream ss;ss<<tcount; result+="t"+ss.str()+" = "+string($$)+"\n"; tcount++;\*/* }

| ID { **checkID**($1);**strcpy**($$,$1); string s1=**string**($1);tvalues[tcount]=s1; cout<<"t"<<tcount++<<" = "<<$$<<endl; */\*stringstream ss;ss<<tcount; result+="t"+ss.str()+" = "+s1+"\n"; tcount++;\*/* }

| SUB ival {string **s1**($2), s2 ="-"; s2=s2+s1; tvalues[tcount]=s2; **strcpy**($$,(s2.**c\_str**())); cout<<"t"<<tcount++<<" = "<<$$<<endl; */\*stringstream ss;ss<<tcount; result+="t"+ss.str()+" = "+s2+"\n"; tcount++;\*/* }

| '(' exp ')' {**strcpy**($$,$2);}

;

initialisation: ID '=' ival ';' { **checkID**($1); **intialiseValue**($1, $3);}

| ID '=' bval ';' { **checkID**($1); **intialiseValue**($1, $3);}

;

declarations: datatype identifier ';' { struct **variable** temp; string **st**($1);temp.type=st; string **st1**($2);temp.name=st1; temp.value = ""; table.**push\_back**(temp); }

;

datatype: types {**strcpy**($$,$1);}

;

identifier: ID {**strcpy**($$,$1);}

;

%%

int **main**(int argc, char \*\*argv)

{

**yyparse**();

}

void **yyerror**(char \*s)

{

**fprintf**(stderr, "error: %s\n", s);

}

*//Function for printing Symbol Table*

void **printTable**()

{

*//cout<<"\nPrinting Table\n";*

for(int i=0; i<table.**size**(); i++){

cout<<table[i].type<<" "<<table[i].name<<" "<<table[i].value<<endl;

}

}

*//Function for intialising values to variables Ex:a=6, b=true*

void **intialiseValue**(char\* s1, char\* s2)

{

string id(s1), **val**(s2);

int i;

for(i=0; i<table.**size**(); i++){

if(id.**compare**(table[i].name) == 0) break;

}

*//Checking if the initialised variable has been declared*

if(i==table.**size**()){

cout<<id<<" not declared\n"; **exit**(0);

}

*//Generating an error if an int is assigned a boolean value*

if(table[i].type == "int" && (val=="true"|val=="false"))

{cout<<"isyntax error\n"; **exit**(0);}

*//Generating an error if a bool is assigned an integer value*

else if(table[i].type == "bool" && (val!="true"&&val!="false"))

{cout<<"bsyntax error\n"; **exit**(0);}

table[i].value = val;

cout<<"t"<<tcount++<<" = "<<val<<endl;

cout<<id<<" = t"<<tcount-1<<endl;

}

*//function for checking if the variable used has been declared*

void **checkID**(char\* a1){

string id(a1);

int i;

*//Checking the symbol table for the variable*

for(i=0; i<table.**size**(); i++){

if(id.**compare**(table[i].name) == 0) break;

}

if(i==table.**size**()){

cout<<id<<" not declared\n"; **exit**(0);

}

}

int **findVal**(char\* s)

{

string id(s); int i;

for(i=0; i<table.**size**(); i++){

if(id.**compare**(table[i].name) == 0) break;

}

if(table[i].value == "")

{cout<<"vsyntax error\n"; **exit**(0);}

return **atoi**((table[i].value).**c\_str**());

}

*//Function for generating three address code for the evaulation of a1 \*symbol\* a2 Ex: 2+3*

string **threeAddress**(char\* a1, char\* a2, string symbol){

int c1=-1,c2=-1;

string s1(a1), **s2**(a2);

for(map<int, string> ::iterator itr = tvalues.**begin**(); itr!=tvalues.**end**(); itr++){

if(itr->second == s1 && **find**(repeat.**begin**(), repeat.**end**(),itr->first )==repeat.**end**()&&c1==-1) {c1 = itr->first;repeat.**push\_back**(c1);}

if(itr->second == s2 && **find**(repeat.**begin**(), repeat.**end**(),itr->first)==repeat.**end**()&& c2==-1) {c2 = itr->first;repeat.**push\_back**(c2);}

}

string s3 = s1;

s3+= symbol;

s3+= s2;

tvalues[tcount]=s3;

cout<<"t"<<tcount++<<" = t"<<c1<<symbol<<"t"<<c2<<endl;

return s3;

}

*//Function for generating three address code for the evaulation of \*symbol\* a1 Ex: ~9*

string **threeAddress**(char\* a1, string symbol){

int c1;

string s1(a1);

for(map<int, string> ::iterator itr = tvalues.**begin**(); itr!=tvalues.**end**(); itr++){

if(itr->second == s1) c1 = itr->first;

}

string s3= "";

s3+=symbol;

s3+= s1;

tvalues[tcount]=s3;

cout<<"t"<<tcount++<<" = "<<symbol<<"t"<<c1<<endl;

return s3;

}

*//Function for generating three address code for the evaulation of a1 \*symbol\*= a2 Ex: a+=2;*

void **threeAddressSelf**(char\* a1, char\*a2, string symbol){

int c1;

string s(a1),**s1**(a2);

for(map<int, string> ::iterator itr = tvalues.**begin**(); itr!=tvalues.**end**(); itr++){

if(itr->second == s1) c1 = itr->first;

}

cout<<"t"<<tcount<<" = "<<s<<"\n";

cout<<s<<" = t"<<c1<<symbol<<"t"<<tcount++<<"\n";

}

Grammar Rules:

Declaration:

We use the below grammar rules for declaration

Declarations -> datatype identifier ';'

datatype -> types

identifier -> ID

where datatype denotes the datatype of the variable and identifier denotes the name of the variable. types is the token returned by the lex file when it reads a datatype like int, unsigned int, bool

Initialisation:

We use the below grammar rule for initialisation

Initialisation -> ID '=' ival ';'| ID '=' bval ';'

Where ID is the name of the variable and ival is the integer value if the datatype is int or unsigned int and bval is the Boolean value if the datatype is bool

Expression Evaluation:

We use the below grammar rules for expression evaluation

exp: factor| exp ADD factor | exp SUB factor

factor: powterm | factor MUL powterm | factor DIV powterm

powterm: term| term POW powterm

ADD, SUB, MUL, DIV, POW respresent the tokens of addition, subtraction, multiplication, division and exponentiation respectively.

Addition, subtraction, multiplication, division are left associate whereas exponentiation is right associative as we can observe in the grammar rules.

Boolean Expression Evaluation:

Booleval -> bool\_and | booleval BOR bool\_and

bool\_and -> bool\_not | bool\_and BAND bool\_not

bool\_not -> bool\_rel | '!' bool\_not

bool\_rel -> bval | term RELOP term | term

where bval has Boolean values like true and false, relop is for relational operators like <, <=, >, >=, ==, !=. BOR is for Boolean or (“||”) and BAND is for Boolean and (“&&”)

Conditional if..then.. statement:

cond\_if -> IF '(' M1 booleval M2 ')' '{' S '}' %prec IFX

where IF is the token for “if” and booleval is the Boolean expression to be evaluated for condition checking. M1 and M2 are markers used to create a label at the start of the if block and to jump to the end of the if block if the condition fails respectively. We use the %nonassoc IFX as we also have the if..else.. statement in the grammar rules and there will be an error if there is no else block in the input. We use S, which is the start symbol, inside the if block as our compiler allows for nesting

Conditional if..then..else.. statement:

cond\_if\_else -> IF '(' M1 booleval M2 ')' '{' S '}' M4 ELSE M5 '{' S '}'

where IF is the token for “if”, ELSE is the token for “else” and booleval is the Boolean expression to be evaluated for condition checking. M1 , M2 , M4 and M5 are the markers used to create a label at the start of the if block, to jump to the else block if the condition fails, to jump of the end of the entire if..else.. block and to create a label for the else block respectively. We use S, which is the start symbol, inside the if and else blocks as our compiler allows for nesting

While Loop:

cond\_while -> WHILE '(' M1 booleval M2 ')' '{' S M3 '}'

where WHILE is the token for “while” and booleval is the Boolean expression to be evaluated for condition checking. M1 , M2 and M3 are markers used to create a label at the start of the if block, to jump to the end of the if block if the condition fails and to the jump to the start of the block after an iteration of the loop respectively. We use S, which is the start symbol, inside the while block as our compiler allows for nesting.

Conditional Switch Statement:

cond\_switch ->  SWITCH '(' booleval M11 ')' '{' cases '}'

cases ->  ɛ| M44 CASE booleval ':' M22 S BREAK M33 ';' cases

| DEFAULT ':'

Where SWITCH, CASE, BREAK and DEFAULT are the tokens for “switch”, “case”, “break” and “default” respectively and booleval is the Boolean expression to be evaluated. Marker M11 is used to jump to the appropriate case block based on the result of the Boolean expression. M44, M22 and M33 are the markers used to create a label for the case, to jump to the appropriate case if the current case is not the required case and to jump to the end after reading the break statement respectively. We use S, which is the start symbol, inside the case block as our compiler allows for nesting.

Functions used in the Program:

void **printTable**();

This function is used to pint the symbol table which displays the name and datatype of the variable

void **intialiseValue**(char\*, char\*);

This function is used to insert the value of the variable in the symbol table and to check if there is a datatype mismatch

string **threeAddress**(char\* a1, char\* a2, string symbol);

This function is used to generate three address code for binary expressions like a+b

t0= a1

t1 = a2

t2 = t0\*symbol\* t1

string **threeAddress**(char\* a1, string symbol);

This function is used to generate three address code for unary expressions like ~a

t0 = a1

t1 = \*symbol\* t0

void **threeAddressSelf**(char\* a1, char\* a2, string symbol);

This function is used to generate three address code for expressions like a+=b

t0 = a1

t1 = a2

t2 = t0 \*symbol\* t1

void **checkID**(char\*);

This function is used to produce a error when a variable which was not declared is used.

Sample Test Cases:

Input:

int a;

Output:

int a

Input:

a=6;

Output:

t0 = 6

a = t0

Input:

bool b;

Output:

bool b

Input:

a=2+3\*4;

Output:

t0 = 2

t1 = 3

t2 = 4

t3 = t1\*t2

t4 = t0+t3

a = t4

Input:

a+=3+(2&3);

Output:

t0 = 3

t1 = 2

t2 = 3

t3 = t1&t0

t4 = t2+t3

t5 = a

a = t4 + t5

Input:

b=2&&3<=5;

Output:

t0 = 2

t1 = 3

t2 = 5

t3 = t1<=t2

t4 = t0&&t3

b = t4

Input:

if(a<=2){a+=1;}

Output:

start0:

t0 = a

t1 = 2

t2 = t0<=t1

jump !t2 mid0

t3 = 1

t4 = a

a = t3 + t4

mid0:

Input:

if(a<=2){a+=1;}else{a-=1;}

Output:

start0:

t0 = a

t1 = 2

t2 = t0<=t1

jump !t2 mid0

t3 = 1

t4 = a

a = t3 + t4

jump end0

mid0:

t0 = 1

t1 = a

a = t1 - t0

end0:

Input:

while(!b){if(a<=5){a\*=2;}else{a+=2;}}

Output:

start2:

t0 = b

t1 = !t0

jump !t1 mid2

start3:

t2 = a

t3 = 5

t4 = t2<=t3

jump !t4 mid3

t5 = 2

t6 = a

a = t5 \* t6

jump end3

mid3:

t0 = 2

t1 = a

a = t0 + t1

end3:

jump start2

mid2:

Input:

switch(a<=2){case true:a=6;break;case false:a=1;break;default:a=0;}

Output:

t0 = a

t1 = 2

t2 = t0<=t1

k=t2

start4:

t3 = true

jump !k==t3 end4

t4 = 6

a = t4ast0

end4:

start5:

t5 = false

jump !k==t5 end5

t6 = 1

a = t6

jump last0

end5:

t7 = 0

a = t7

last0:

Assumptions:

* There are three types of datatypes – int, unsigned int and bool.
* There are floating point units for addition, subtraction, multiplication, division and exponentiation.
* There are computational units for performing operations like and, or, negation.
* Input expressions given are in sub C format.
* Only generation of three address code is sufficient. Evaluation of them isn’t necessary.
* Only the symbols specified in the lex file should be present in the input.

Limitations:

* Functions are not supported.
* Macros are not supported.
* Usage of inbuilt libraries isn’t supported.
* The compiler only supports in-order execution. It doesn’t support out of order execution.
* Declaration and initialisation in the same statement isn’t supported Ex: int a=6;
* Three address code generated is not optimised.
* Double, float, long and short datatypes are not supported.
* Characters and character arrays are not supported.
* Dynamic memory allocation isn’t supported.

THANK YOU