



PES
UNIVERSITY

COMPILER DESIGN

A Project Report On

MINI- COMPILER

(IF- ELSE & DO -WHILE)

Rohan Sharma - 01FB16ECS311

Rohit R - 01FB16ECS312

Shadan Alam Kaiffee - 01FB16ECS346

Shashwat Mishra - 01FB16ECS361

INDEX

1. Project description

1.1 Objective of the Project

2. Project Content

2.1 Project category

2.1.1 Compiler

3. Platform(Technology/Tools)

3.1 Characteristics

3.2 Features

3.3 Operators

3.4 Data Structures

4. Phases

4.1 phase - 1

4.2 phase - 2

4.3 phase - 3

4.4 phase - 4

4.5 phase - 5

ACKNOWLEDGEMENT

I would like to express my special thanks of gratitude to my “Compiler design lab professor” for their able guidance and support in completing my project.

I would like to extend my gratitude to “Department of Computer Science” for providing me with all the facility that was required.

DATE:

22nd April 2019

PREFACE

The lexical analyzer is responsible for scanning the source input file and translating lexemes into small objects that the compiler can easily process. These small values are often called “tokens”.

The lexical analyzer is also responsible for converting sequences of digits into their numeric form as well as processing other literal constants, for removing comments and whitespace from the source file, and for taking care of many other mechanical details. The lexical analyzer reads a string of characters and checks if a valid token in the grammar.

Lexical analysis terminology:

Token:

- > Terminal Symbol in grammar
- > Class of Sequences of character with a collective meaning
- > Constants, Operators, Punctuation, Keywords.

Lexemes:

- > Character sequence matched by an instance of the token

PROJECT DESCRIPTION

We need to implement a mini compiler which takes as input a C programme and produces as output optimized intermediate code/assembly code . The programme mini construct to focus IF and ELSE BLOCK as well as DO-WHILE LOOP.

The syntax of an IF.....ELSE IF.....ELSE statement in C programming language is -

```
if(boolean_expression 1) {  
    /*Executes when the boolean expreesion 1 is true */  
}else if( boolean_expression 2) {  
    /*Executes when the boolean expression 2 is true*/  
}else if( boolean_expression 3) {  
    /*Executes when the boolean expression 3 is true*/  
}else {  
    /*Executes when the none of the above condition is true*/  
}
```

The syntax of DO.....WHILE Loop in C programming languages is -

```
do {  
    statement(s);  
} while (condition);
```

Lexical Analyzer converts stream of input characters into a stream of tokens. The different tokens that our lexical analyzers identifies are as follows:

KEYWORD: int, char, float, double, if, for, while, else, switch, struct, printf, scanf, case, break, return, typedef, void.

IDENTIFIERS: main, fopen, getch etc

NUMBER: positive and negative integer, positive and negative floating point numbers.

OPERATORS: +, ++, -, --, ||, *, ?, /, >, >=, <, <=, =, ==, &, &&.

BRACKETS: [], { }, ().

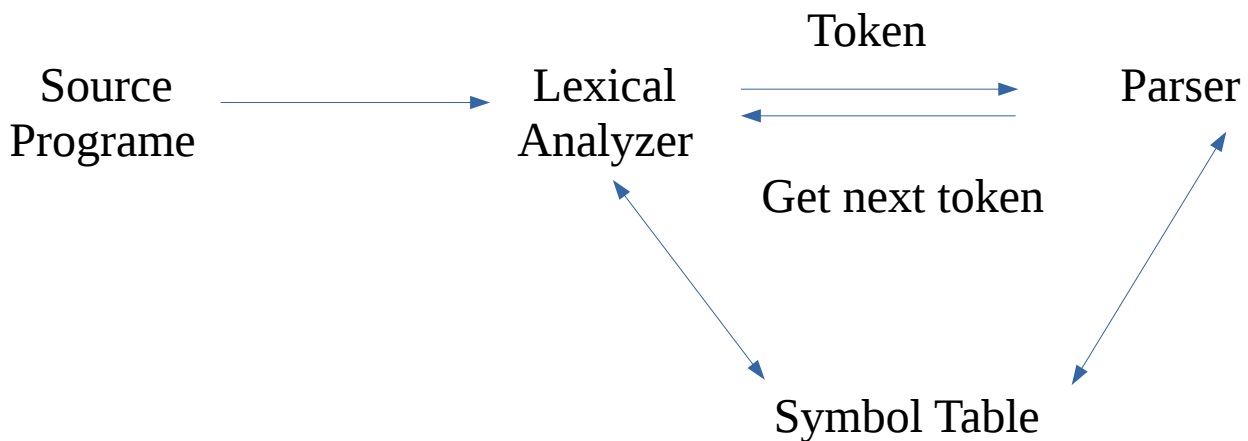
STRING: Set of characters enclosed within the quotes.

COMMENT LINES: Ignore single line , multi line comments.

SYSTEM DESIGN:

Process:

The Lexical Analyzer is the first phase of a compiler. Its main task is to read the input characters and produces as output a sequences of tokens that the parser uses for syntax analysis. The interaction , summarized schematically.



Upon receiving a “get next token” command from the parser, the lexical analyzer read the input characters until it can identify next token.

Sometimes, lexical analyzers are divided into a cascade of two phases the first called “scanning”,and the second “lexical analysis”. The scanner is responsible for doing simple tasks, while the lexical analyzer proper does the more complex operations.

The lexical analyzer which we have designed tasks the input from an input file. It reads one character at a time from the input file , and continues to read until end of the file is reached. It recognizes the valid identifiers, keywords and specifies the token values of the keywords

GRAMMAR:

main:

P-> #include<H.h>

H-> header

M-> T main() {B}

B-> S|I|D|R|lamda

T-> type

R-> return V

IF-ELSE Loop:

I-> if(C){S}E

E-> else if(C){S}E|else {S}|lamda

C-> VOV

V-> [a-zA-Z][a-zA-Z0-9]*[1-9][0-9]*

O-> ==|!=|<|=|>|=|<|>

S-> S|I|lamda

DO- WHILE Loop:

D-> do {S} while(C)

S-> S|D|lamda

C-> VOV

V-> [a-zA-Z][a-zA-Z0-9]*[1-9][0-9]*

O-> ==|!=|<|=|>|=|<|>

OBJECTIVE

AIM OF THE PROJECT:

Aim of the project is to develop mini -compiler which takes as input a C programme and produces as output optimized intermediate code/assembly code. The programme construct to focus is IF and ELSE BLOCK as well as DO- WHILE Loop.

SCOPE OF THE PROJECT:

Lexical analyzers converts the input programme into character stream of valid words of language, known as tokens.

The parser looks into the sequences of these tokens & identifies the language construct occurring in the input programme. The parser and the lexical analyzers work hand in hand; in the sense that whenever the parser needs further tokens to proceed, it request the lexical analyzer. The lexical analyzers in turn scan the remaining input stream and returns the next token occurring there.

Apart from that, the lexical analyzer also participates in the creation and maintenance of symbol. If the symbol is getting defined for the first time, it needs analyzer is most widely used for doing the same.

PROJECT CONTENT

Category of the project is Compiler Design based.

COMPILER:

To define what a compiler is one must first define what a translator is. A translator is a program that takes another program written in one language, also known as the source language, and outputs a program written in another language, known as the target language.

Now that the translator is defined, a compiler can be defined as a translator. The source language is a high-level language such as Java or Pascal and the target language is a low-level language such as machine or assembly.

There are five parts of compilation (or phases of compiler)

1. Lexical Analysis
2. Syntax Analysis
3. Semantic Analysis
4. Code Optimization
5. Code Generation

Lexical Analysis is the act of taking an input source program and outputting a stream of tokens. This is done with a scanner. The scanner can also place identifiers into something called the symbol table or place strings into the string table.

Syntax Analysis is the act of taking the token stream from the scanner and comparing them against the rule and pattern of the specified language.

Semantic Analysis is the act of determining whether or not the parse tree is relevant and meaningful. The output is intermediate code, also known as intermediate representation (IR).

Code optimization makes the IR more efficient. Code optimization is usually done in a sequence of steps. Some optimizations include code hoisting, or moving constant values to better places within the code.

Code Generation is the final step in the compilation process. The input to the code generator is the IR and the output is machine language code.

PLATFORM(TECHNOLOGY / TOOLS)

In computing , C is a general -purpose computer programming language originally developed by 1972 by Dennis Ritchie at the bell Telephone Laboratories to implement the unix operating system.

Characteristics:

Like more imperative languages in the ALGOL tradition, C has facilities for structured programming and allows lexical variable scope and recursion.

1. Non -nest able function definations
2. Variables may be hidden in nested blocks
3. Partially weak typing ; for instance, characters can be used as integers.

Features:

The relatively low -level nature of the language affords the programmer close control over what the computer does.

‘C’ doesnot have some features that are available in some other programming languages:

1. No assignment of array or strings
2. No automatics garbage collector
3. No requirement for bound checking of array.
4. No operations on whole arrays.

Operators:

Bitwise shift (<<, >>)

Assignment (=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=)

Arithmetics (+-, --, *-, /-, %-)

Equality testing (==, !=)

Boolean logic (!, &&, ||)

Bitwise logic (~, &, |, ^)

Data Structures:

C has a static weak typing type system that shares some similarities with that of other ALGOL descendants such as pascal. C is often used in low-level systems programming where escapes from the type system may be necessary.

Arrays:

Array type in 'C' are traditionally, of a fixed, static size specified at compile time. However it is also possible to allocate a block of memory at run time, using standard library's malloc function , and treat it as an array.

'C' does not have a special provision for declaring multidimensional arrays, but rather relies on recursion within the type system to declare arrays of arrays, which effectively accomplishes the same thing.

Although 'C' supports static arrays, it is not required that array indices be validated (bounds checking).

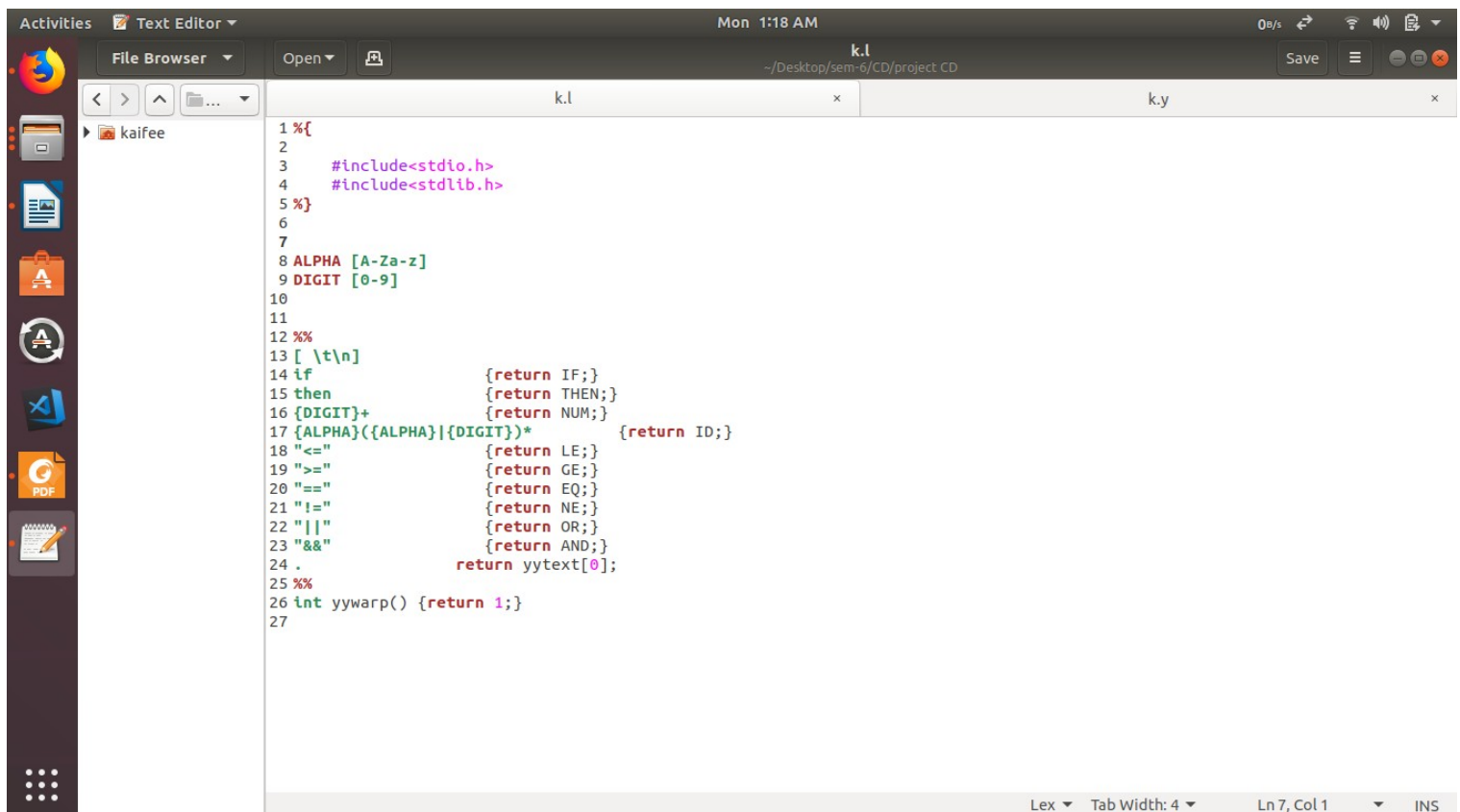
PHASES

PHASE 1 : LEXICAL ANALYSIS

We have used flex/lex to create a scanner for our programming language.

Our scanner will transform the source file from a stream of bits and bytes into a series of meaningful tokens containing information that will be used by the later stages of the compiler. All token names must start with T_(token name).

Scanner implementation: The yylval global variable is used to record the value for each lexeme scanned and the yylloc global records the lexeme position (line number and column). The action for each pattern will update the global variables and return the appropriate token code.

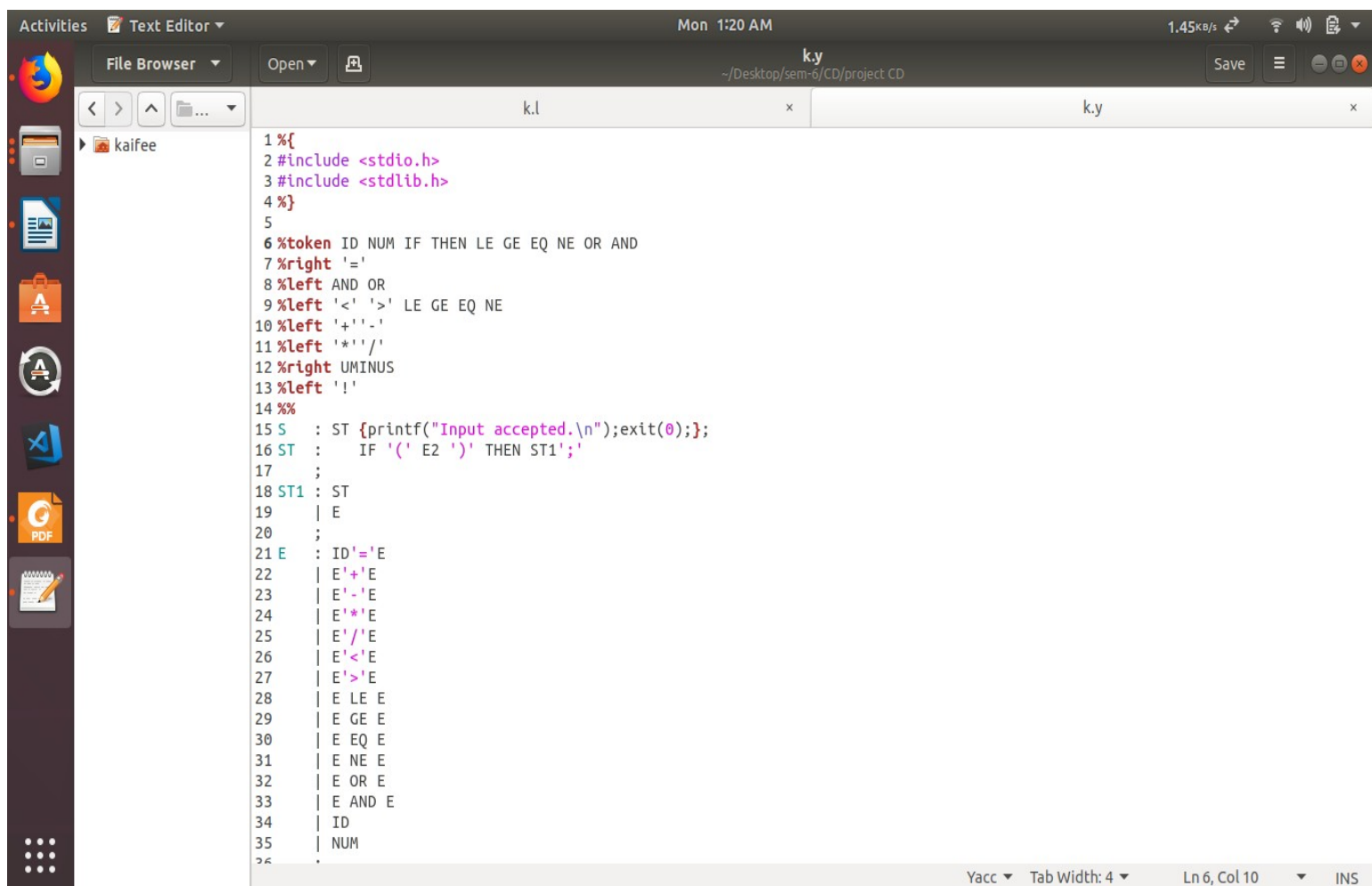


The screenshot shows a Linux desktop with a text editor window titled 'Text Editor' at 'Mon 1:18 AM'. The window contains a Lex scanner implementation for a file named 'k.l'. The code defines tokens for whitespace, identifiers, numbers, and various operators. The action part of the scanner returns token codes like IF, THEN, NUM, ID, LE, GE, EQ, NE, OR, and AND. The status bar at the bottom indicates 'Lex', 'Tab Width: 4', 'Ln 7, Col 1', and 'INS'.

```
1 %{
2
3     #include<stdio.h>
4     #include<stdlib.h>
5 %}
6
7
8 ALPHA [A-Za-z]
9 DIGIT [0-9]
10
11
12 %%
13 [ \t\n]
14 if {return IF;}
15 then {return THEN;}
16 {DIGIT}+ {return NUM;}
17 {ALPHA}({ALPHA}|{DIGIT})* {return ID;}
18 "<=" {return LE;}
19 ">=" {return GE;}
20 "==" {return EQ;}
21 "!=" {return NE;}
22 "||" {return OR;}
23 "&&" {return AND;}
24 . {return yytext[0];}
25 %%
26 int yywrap() {return 1;}
27
```

PHASE II : SYNTAX ANALYZER

Syntax analysis is only responsible for verifying that the sequence of tokens forms a valid sentence given the definition of your Programming Language grammar. The parser will read your source programs and construct a Abstract Syntax Tree. If no syntax errors are encountered, your code will print the completed parse tree as flat text. At this stage, you aren't responsible for verifying meaning, just structure.



The screenshot shows a Linux desktop environment with a text editor window open. The window title is "k.y" and the file path is "~/Desktop/sem-6/CD/project CD". The editor contains a Yacc grammar file with the following content:

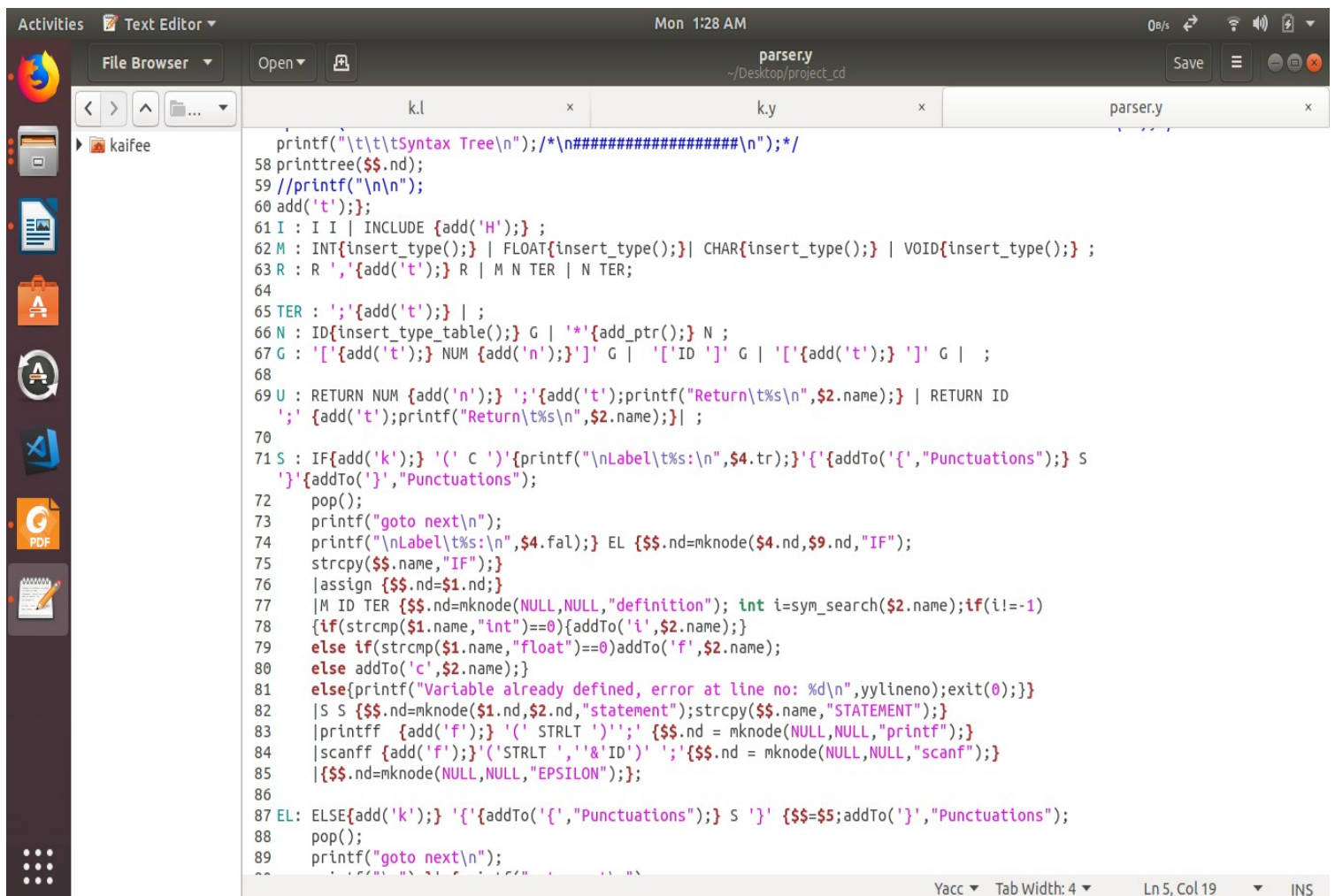
```
1 %{
2 #include <stdio.h>
3 #include <stdlib.h>
4 %}
5
6 %token ID NUM IF THEN LE GE EQ NE OR AND
7 %right '='
8 %left AND OR
9 %left '<' '>' LE GE EQ NE
10 %left '+' '-'
11 %left '*' '/'
12 %right UMINUS
13 %left '|'
14 %%
15 S : ST {printf("Input accepted.\n");exit(0);};
16 ST : IF '(' E2 ')' THEN ST1;
17 ;
18 ST1 : ST
19 | E
20 ;
21 E : ID '=' E
22 | E '+' E
23 | E '-' E
24 | E '*' E
25 | E '/' E
26 | E '<' E
27 | E '>' E
28 | E LE E
29 | E GE E
30 | E EQ E
31 | E NE E
32 | E OR E
33 | E AND E
34 | ID
35 | NUM
36 ;
```

The status bar at the bottom indicates "Yacc", "Tab Width: 4", "Ln 6, Col 10", and "INS".

PHASE III : SEMANTIC ANALYSIS

Write appropriate rules to check for semantic validity (type checking, declare before use, etc.). Make sure variables must be declared and can only be used in ways that are acceptable for the declared type.

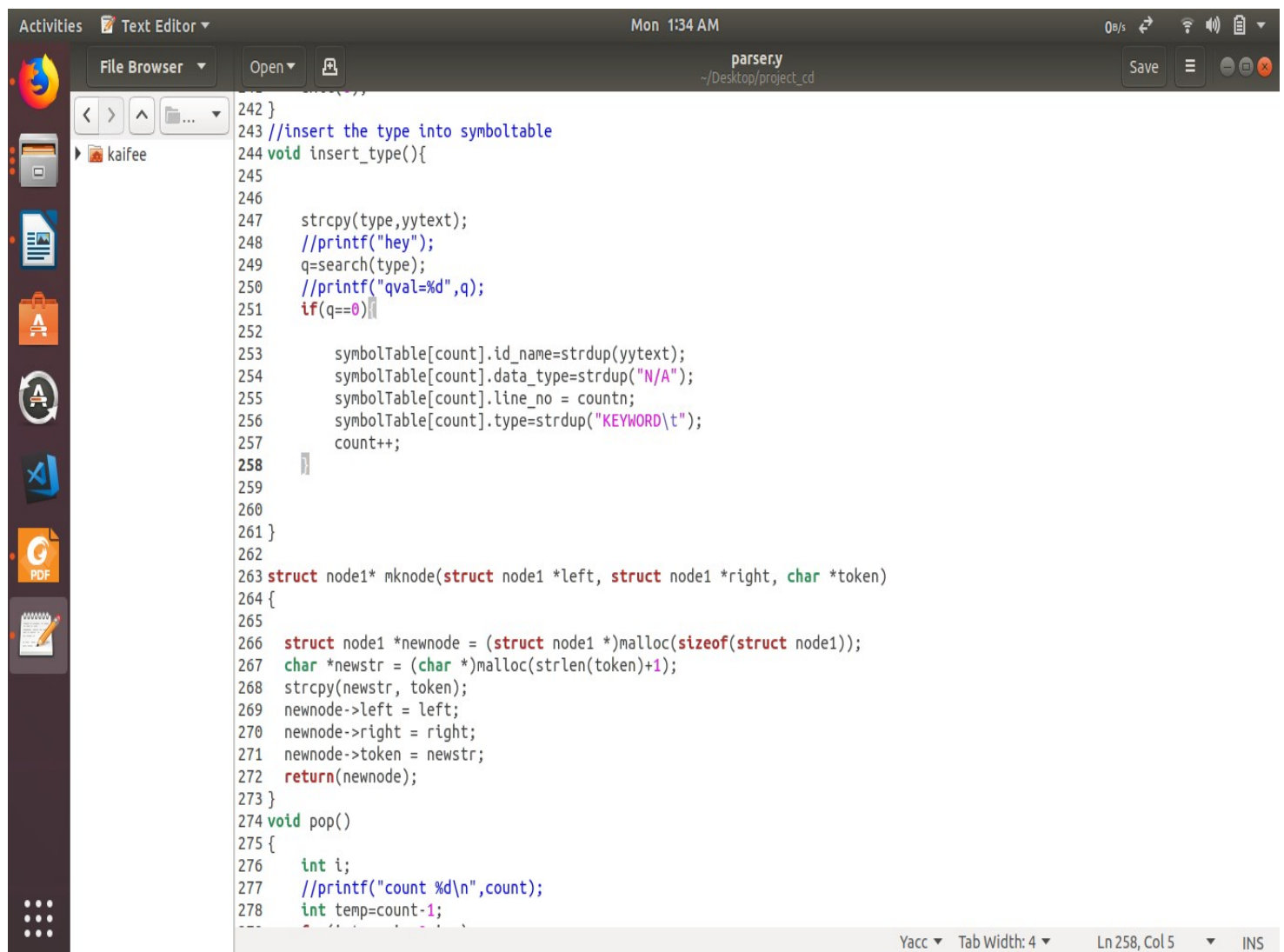
The test expression used in an if/while/for statement must evaluate to a Boolean value. our program will be considered correct if it verifies the semantic rules and reports appropriate errors. your analyzer needs to show it can handle errors related to scoping and declarations, because these form the foundation for the later work.



```
Mon 1:28 AM
0B/s
Save
File Browser
Open
parser.y
~/Desktop/project_cd
k.l x k.y x parser.y x
kaifee
printf("\t\t\tSyntax Tree\n");/*\n#####\n*/;
58 printtree($$.nd);
59 //printf("\n\n");
60 add('t');
61 I : I I | INCLUDE {add('H');} ;
62 M : INT{insert_type();} | FLOAT{insert_type();} | CHAR{insert_type();} | VOID{insert_type();} ;
63 R : R ' ' {add('t');} R | M N TER | N TER;
64
65 TER : ' ' {add('t');} | ;
66 N : ID{insert_type_table();} G | '*' {add_ptr();} N ;
67 G : '[' {add('t');} NUM {add('n');} ']' G | '[' ID ']' G | '[' {add('t');} ']' G | ;
68
69 U : RETURN NUM {add('n');} ' ' {add('t');} printf("Return\t%s\n", $2.name); | RETURN ID
    ' ' {add('t');} printf("Return\t%s\n", $2.name); | ;
70
71 S : IF {add('k');} '(' C ')' {printf("\nLabel\t%s:\n", $4.tr);} '{' {addTo('{', "Punctuations");} S
    '}' {addTo('}', "Punctuations");}
72 pop();
73 printf("goto next\n");
74 printf("\nLabel\t%s:\n", $4.fal); EL {$$.nd=mknnode($4.nd,$9.nd,"IF");
75 strcpy($$.name,"IF");}
76 |assign {$$.nd=$1.nd;}
77 |M ID TER {$$.nd=mknnode(NULL,NULL,"definition"); int i=sym_search($2.name);if(i!=-1)
78 {if(strcmp($1.name,"int")==0){addTo('i',$2.name);}
79 else if(strcmp($1.name,"float")==0){addTo('f',$2.name);}
80 else addTo('c',$2.name);}
81 else{printf("Variable already defined, error at line no: %d\n",yylineno);exit(0);}
82 |S S {$$.nd=mknnode($1.nd,$2.nd,"statement");strcpy($$.name,"STATEMENT");}
83 |printf {add('f');} '(' STRLT ' ' ' ' {$$.nd = mknnode(NULL,NULL,"printf");}
84 |scanf {add('f');} '(' STRLT ' ' '&' ID ' ' ' ' {$$.nd = mknnode(NULL,NULL,"scanf");}
85 |{$$.nd=mknnode(NULL,NULL,"EPSILON");};
86
87 EL: ELSE {add('k');} '{' {addTo('{', "Punctuations");} S '}' {$$=$5;addTo('}', "Punctuations");}
88 pop();
89 printf("goto next\n");
90
Yacc Tab Width: 4 Ln 5, Col 19 INS
```


PHASE IV : INTERMEDIATE CODE GENERATION

We have to write appropriate rules in Parser to generate Three address code and code generated must be in Quadruple format. All temporaries must get a place in symbol table.

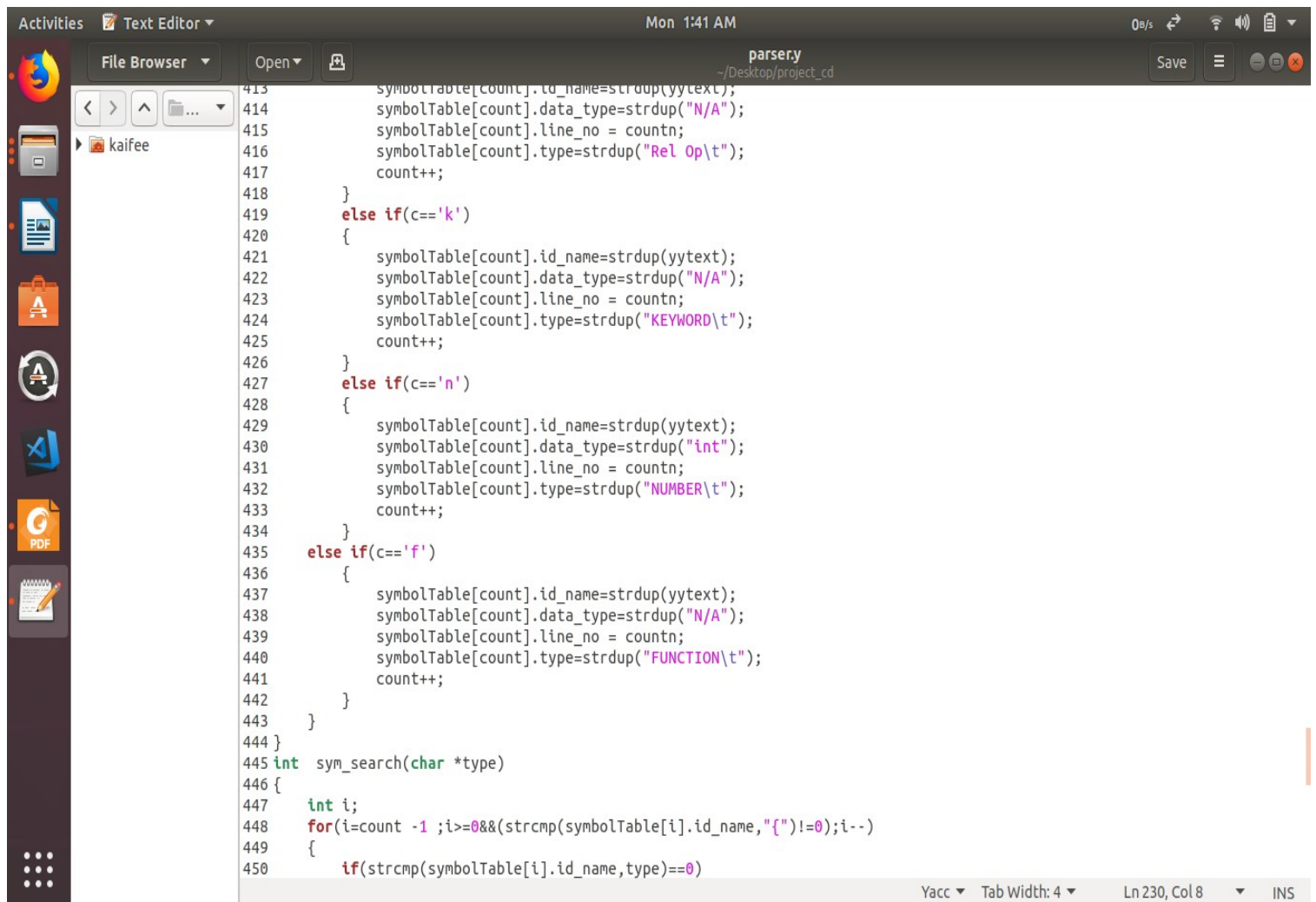


```
242 }
243 //insert the type into symboltable
244 void insert_type(){
245
246     strcpy(type,yytext);
247     //printf("hey");
248     q=search(type);
249     //printf("qval=%d",q);
250     if(q==0){
251         symbolTable[count].id_name=strdup(yytext);
252         symbolTable[count].data_type=strdup("N/A");
253         symbolTable[count].line_no = countn;
254         symbolTable[count].type=strdup("KEYWORD\t");
255         count++;
256     }
257 }
258
259
260
261 }
262
263 struct node1* mknode(struct node1 *left, struct node1 *right, char *token)
264 {
265     struct node1 *newnode = (struct node1 *)malloc(sizeof(struct node1));
266     char *newstr = (char *)malloc(strlen(token)+1);
267     strcpy(newstr, token);
268     newnode->left = left;
269     newnode->right = right;
270     newnode->token = newstr;
271     return(newnode);
272 }
273
274 void pop()
275 {
276     int i;
277     //printf("count %d\n",count);
278     int temp=count-1;
279 }
```

PHASE V : INTERMEDIATE CODE OPTIMIZATION PHASE

We have eliminate dead code/ unreachable code and implement common subexpression elimination.

We have also implement constant folding and constant propagation and move loop invariant code outside the loop.



The screenshot shows a text editor window titled 'parser.y' with a file browser on the left. The code is in C and implements a symbol table and a search function. The symbol table is an array of structures, each containing an id_name, data_type, line_no, and type. The search function iterates through the symbol table to find a matching entry.

```
413     symbolTable[count].id_name=strdup(yytext);
414     symbolTable[count].data_type=strdup("N/A");
415     symbolTable[count].line_no = countn;
416     symbolTable[count].type=strdup("Rel Op\t");
417     count++;
418 }
419 else if(c=='k')
420 {
421     symbolTable[count].id_name=strdup(yytext);
422     symbolTable[count].data_type=strdup("N/A");
423     symbolTable[count].line_no = countn;
424     symbolTable[count].type=strdup("KEYWORD\t");
425     count++;
426 }
427 else if(c=='n')
428 {
429     symbolTable[count].id_name=strdup(yytext);
430     symbolTable[count].data_type=strdup("int");
431     symbolTable[count].line_no = countn;
432     symbolTable[count].type=strdup("NUMBER\t");
433     count++;
434 }
435 else if(c=='f')
436 {
437     symbolTable[count].id_name=strdup(yytext);
438     symbolTable[count].data_type=strdup("N/A");
439     symbolTable[count].line_no = countn;
440     symbolTable[count].type=strdup("FUNCTION\t");
441     count++;
442 }
443 }
444 }
445 int sym_search(char *type)
446 {
447     int i;
448     for(i=count -1 ;i>=0&&(strcmp(symbolTable[i].id_name,"")!=0);i--)
449     {
450         if(strcmp(symbolTable[i].id_name,type)==0)
```

Yacc Tab Width: 4 Ln 230, Col 8 INS

REFERENCE

- www.google.com
- ALSU Dragon book : Alfred V. Aho
- www.stackoverflow.com
- www.geeksforgeeks.org

THANK YOU!