CSE306L

Compiler Design Project

R programming language

Final Report

AP19110010393 - Akash Perla
AP19110010424 - Kreethi Mishra
AP19110010426 - Hanish Vaitla
AP19110010432 - Lalith Veerla
AP19110010442 - Harshini Chintalacheruvu
AP19110010448 - Lehar Sancheti
AP19110010449 - Sumana Bandarupalli

CSE C
Group 8

Contents

Data types available in the R language

R supports five data types- Numeric, Integer, Complex, Character(a.k.a. String), Logical(a.k.a. Boolean).

There are three number types in R:

1. Numeric - A numeric data type is the most common type in R, and contains any number with or without a decimal, like: 10.5, 55, 787.
2. Integers - They are numeric data without decimals. This is used when you are certain that you will never create a variable that should contain decimals. To create an integer variable, you must use the letter L after the integer value (500L, 4000000L)
3. Complex - A complex number is written with an "i" as the imaginary part: 3+4i, 15+i.

4. Character(a.k.a. String) - R supports character data types where you have all the alphabets and special characters. It stores character values or strings. Strings in R can contain alphabets, numbers, and symbols. The easiest way to denote that a value is of character type in R is to wrap the value inside single or double inverted commas. (“Hello World”)
5. Logical(a.k.a. Boolean) - R has logical data types that take either a value of TRUE or FALSE. A logical value is often created via a comparison between variables. A conditional expression that evaluates to true or false is also of boolean type.

To find the data type of an object you have to use the *class()* function.

Syntax of variable declaration & assumptions for R language

Ways of Declaring a Variable:

- In R, a variable always starts with a letter or with a period. A variable if started with a dot cannot be succeeded by a number.
- Variables cannot be created with keywords that are already predefined in R.
- A variable in R can be defined using just letters or an underscore with

letters, dots along with letters. We can even define variables as a mixture of digits, dots, underscore, and letters.

Assigning values to variables:

- In R, assigning values to a variable can be specified or achieved using the syntax of left angular brackets signifying the syntax of an arrow.

  Ex: a <- 12
  b <- "Hello"

- After assigning values to a variable, these values can be printed using the predefined cat() function and print() function.
- To integrate multiple statements assigned to a single variable and separated by double quotation marks and commas, we can use the function of cat().

CODE	OUTPUT
X <- "Hi, This is CD Phase 1 Report By Group 8" cat(X)	Hi, This is CD Phase 1 Report By Group 8

Decision-making statements in R language

Decision-making is about deciding the order of execution of statements based on certain conditions.
In decision making programmer needs to provide some condition that is evaluated by the program, along with it there also provided some statements which are executed if the condition is true and optionally other statements if the condition is evaluated to be false.

- if-else statement
  If-else provides us with an optional else block which gets executed if the condition for if block is false. If the condition provided to if block is true then the statement within the if block gets executed, else the statement within the else block gets executed.

- Syntax of if-else:

```
if(condition is true) {
        execute this statement
    }  else {
      execute this statement
    }
```

Iterative statements in R language

- for Loop:
  This type of loop is to be used when someone knows exactly how many times you want the code to repeat. The for loop accepts an iterator variable and a vector. It repeats the loop, giving the iterator each element from the vector in turn. In the simplest case, the vector contains integers:

  EX:
  ```
  x <- c(1,9,3,5,8,7,2)
  count <- 0
  for (val in x) {
  if(val %% 2 == 0)  count = count+1
  }
  print(count)

  OUTPUT: [1] 2
  ```

  In the above example, the loop iterates 7 times as the vector x has got 7 elements. In each iteration, the variable takes on the value of the corresponding element of x. Here we have used a counter to count the number of even numbers in x. We can see that x contains 2 (2 and 8) even numbers.

- while Loop:
  'While' loops are more like backward repeat loops. Instead of executing some code and then checking to see if the loop should end or not, this type of loop checks first and then (maybe) executes. Since the check happens at the very beginning, the contents of the loop may never be executed (unlike

in a repeat loop).

EX:
```
i <- 20
while (i < 30) {
print(i)
i = i+1
}
```

In general, it is always possible to convert a ‘repeat’ loop to a ‘while’ loop or a ‘while’ loop to a ‘repeat’ loop, but usually the syntax is much cleaner one way or the other. If you know that the contents must execute at least once, use repeat; otherwise, use while.

CFG for constructs in R Language

Data Types

identifier → < . ><letter><identifier> | (<letter> | <digit>| _ | .) <identifier>) | λ
numeric → <integer> | <double>
integer → (+ | -) <digit> ‘L’
double → (+ | -) <digit> . <digit> | (+ | -) <digit> .
letter → a | b | c | d |.....| y | z | A | B | C | D |.....| Y | Z
digit → [0-9] <digit> | [0-9]

if statement

if_statement → if (<condition>) { <statements> }
condition → <expr> <op> <expr>
op → < | > | == | >= | <= | !=
statements → <identifier> <- <expr> <statements> | <identifier> <ops> <expr> <statements> |print(<identifier>) <statements> | λ
ops → += | -= | *= | *= | /=
expr → <identifier> | <numeric>

if else statement

if_else_statement → if (<condition>) { <statements> } else { <statements> }
condition → <expr> <op> <expr>
op → < | > | == | >= | <= | !=
statements → <identifier> <- <expr> <statements> | <identifier> <ops> <expr> <statements> |print(<identifier>) <statements> | λ
ops → += | -= | *= | *= | /=
expr → <identifier> | <numeric>

for loop

for_loop → for(<identifier> in <range>) { <statements> }
range → <integer> : <integer> | seq(<integer> , <integer>) | seq(<integer> , <integer> , <numeric>)
statements → <identifier> <- <expr> <statements> | <identifier> <ops> <expr> <statements> |print(<identifier>) <statements> | λ
ops → += | -= | *= | *= | /=

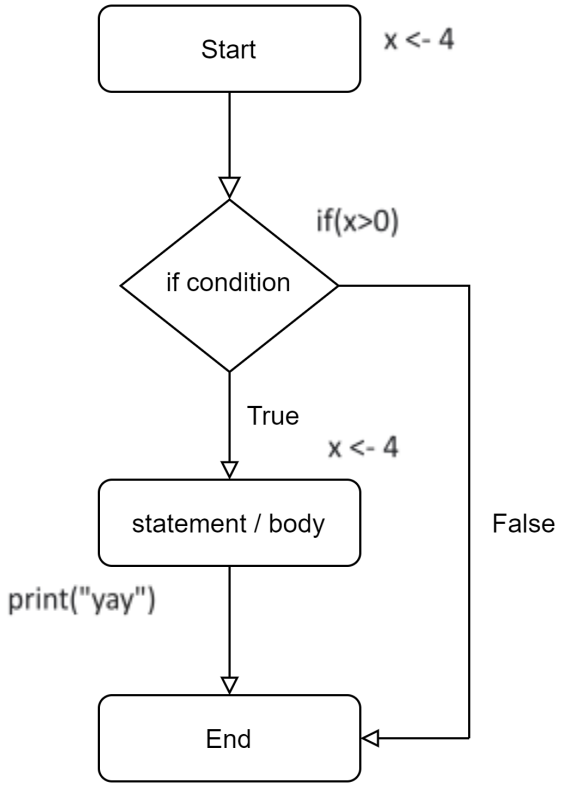
while loop

while_loop → while (<condition>) { <statements> }
condition → <expr> <op> <expr>
op → < | > | == | >= | <= | !=
statements → <identifier> <- <expr> <statements> | <identifier> <ops> <expr> <statements> |print(<identifier>) <statements> | λ
ops → += | -= | *= | *= | /=
expr → <identifier> | <numeric>

Examples with flow graphs:

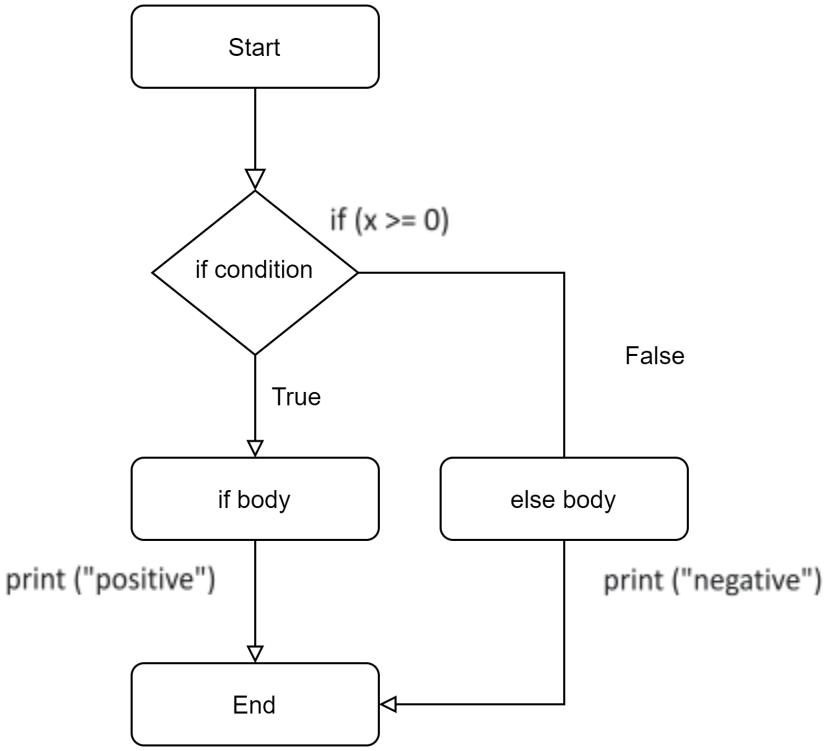
if statement

```
x <- 4
if(x>0) {
print("yay")
}
```



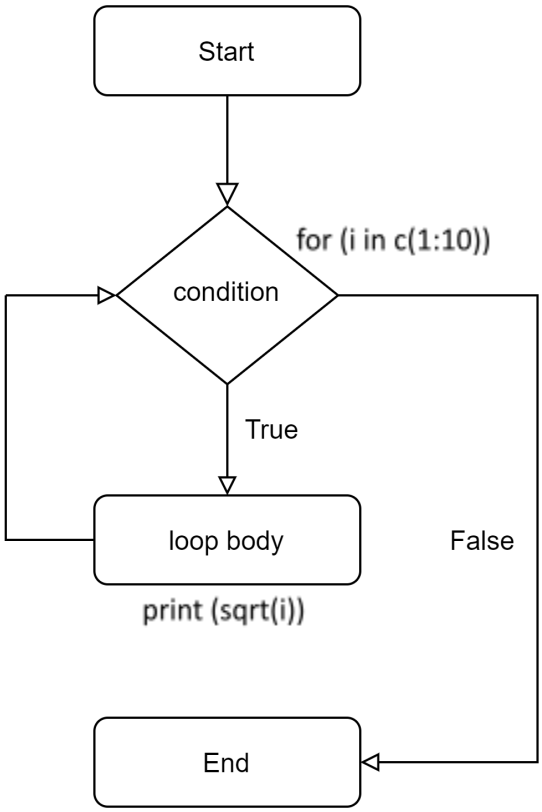
if-else statement

```
x <- 4
if (x >= 0) {
print ("positive")
} else {
print ("negative")
}
```



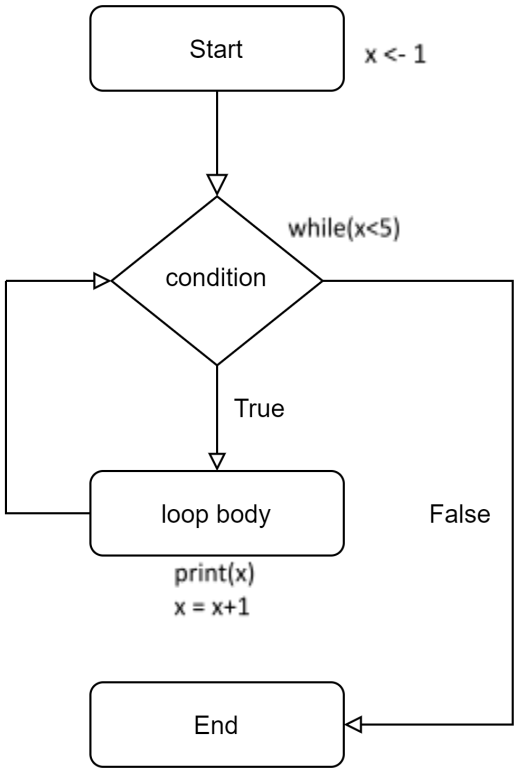
for Loop:

```
for (i in c(1:10)) {
print (sqrt(i))
}
```



while Loop:

```
x <- 1
while(x<5) {
print(x)
x = x+1
}
```



Design of Parser

Syntax Analysis or parsing is the second phase after the lexical analyzer. A syntax analyzer or parser takes the input from the lexical analyzer in the form of tokens. The parser analyses the source code against the production rules to detach the errors. It checks whether the given input is incorrect syntax. It dies it by building parse tree. The parse tree is defined by using predefined Grammer. If the given string can be used by predefined Grammer then the string is accepted as correct syntax.

Parse Tree
A parse tree is the graphical depiction of a derivation. The start symbol becomes the root node of the tree

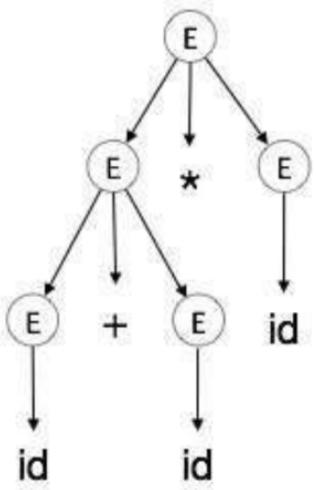
Consider the following example with given production rules

Input String:id+id*id
Production rules:

E->E+E
E->E*E
E->id

The leftmost Derivation is
E->E*E
E->E+E*E
E->id+E*E
E->id+id*E
E->id+id*id

The parse tree for the following is



In a Parse tree:

1)All leaf nodes are terminal

2)All interior nodes are non-terminals

3)In order terminal gives original output

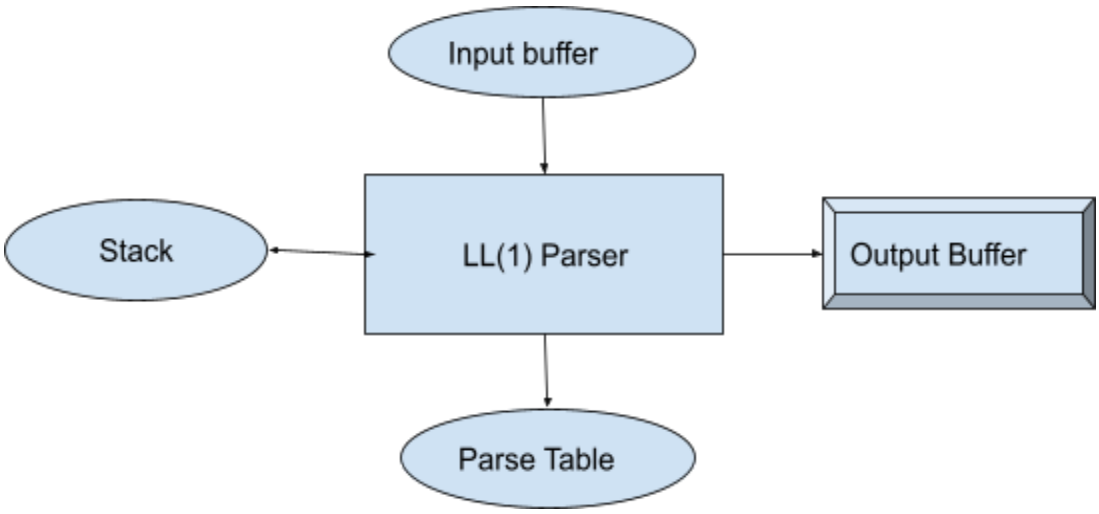
LL(1) Parsing

It is a non descent recursive parser used for the formation of parse tree by using CFG. The character sick of LL(1) Parser are as follows-:

1)It reads from Left to Right

2)It is top to down parser

3)It reads one token at a time from lexical analyzer output



To Explain the working of the above diagram let's take an example: let the steam of token me **a+b**

1)The tokens will be stored in input buffer 

2)From input each token is pushed in a stack and popped out when an operation is done

3)A parsing table is formed with the help of the First and Follow the method

4)All these create an output thus a parse tree

Construction of LL(1) Parser

There are five factors responsible for the construction of LL(1) Parser

1)Elimination of Left Recursion

2)Elimination of Left Factoring

3)Find First and Follow

4)Construction of Parsing Table

5)To check whether String matches ie accepted by the Parse tree

For example let the CFG of the language be-:

if_statement ⟶ if (condition) { statements }

condition ⟶ expr op expr

op ⟶ < | > | == | >= | <= | !=

statements ⟶ identifier <- exprstatements| expr ops expr statements |print(identifier) statements| λ

ops ⟶ + | - | * | *| /

expr ⟶ identifier | numeric

identifier ⟶ . letter identifier| letter | digit| _ | .) identifier| λ

numeric ⟶ integer | double

Step 1-First Check the Cfg shouldn't have any left recursion or factoringa ,the above CFG doesn't have any

Step2-Find the first and follow of the CFG

First Function

First(if-statement)={if}

First(condition)=First(expr)={identifier,numeric}

First(op)={<,>,=,>,!}

First(statement)={identifier,print,$}

First(ops)={+,-,*,/}

First(expr)=First(identifier)={.,(,$}

First(identifier)={.,(,$}

First(numeric)={integer,double}

Follow Function

Follow(if-statement)={$}

Follow(condition)={(}

Follow(op)=First(expr)={identifier,numeric}

Follow(statements)={ } }

Follow of ops=First(expr)={identifier,numeric}

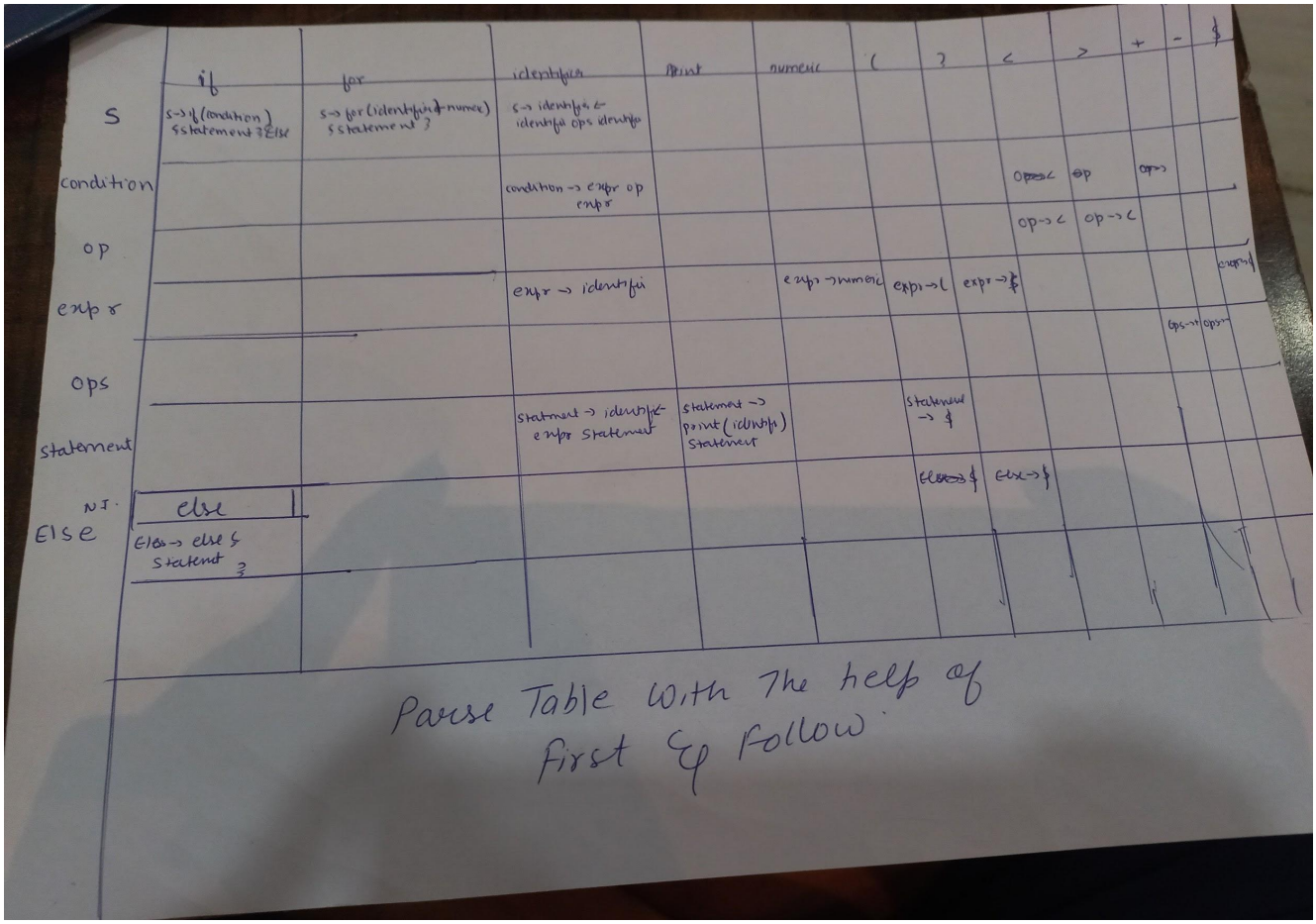
Follow(expr)=Follow(condition)UFollow(statements)={ (, } }

Follow(identifier)={(}UFollow(expr)U{<-}={ (, } , <- }

Follow(numeric)=Follow(expr)=Follow(condition)UFollow(statements)={ (, } }

Step3-> Create Parse table with help of First and Follow

Parse table



Step4->Do the Stack implementation

Stack Implementation

String required->if(5>=3){

c<-a+b

print(c)

}

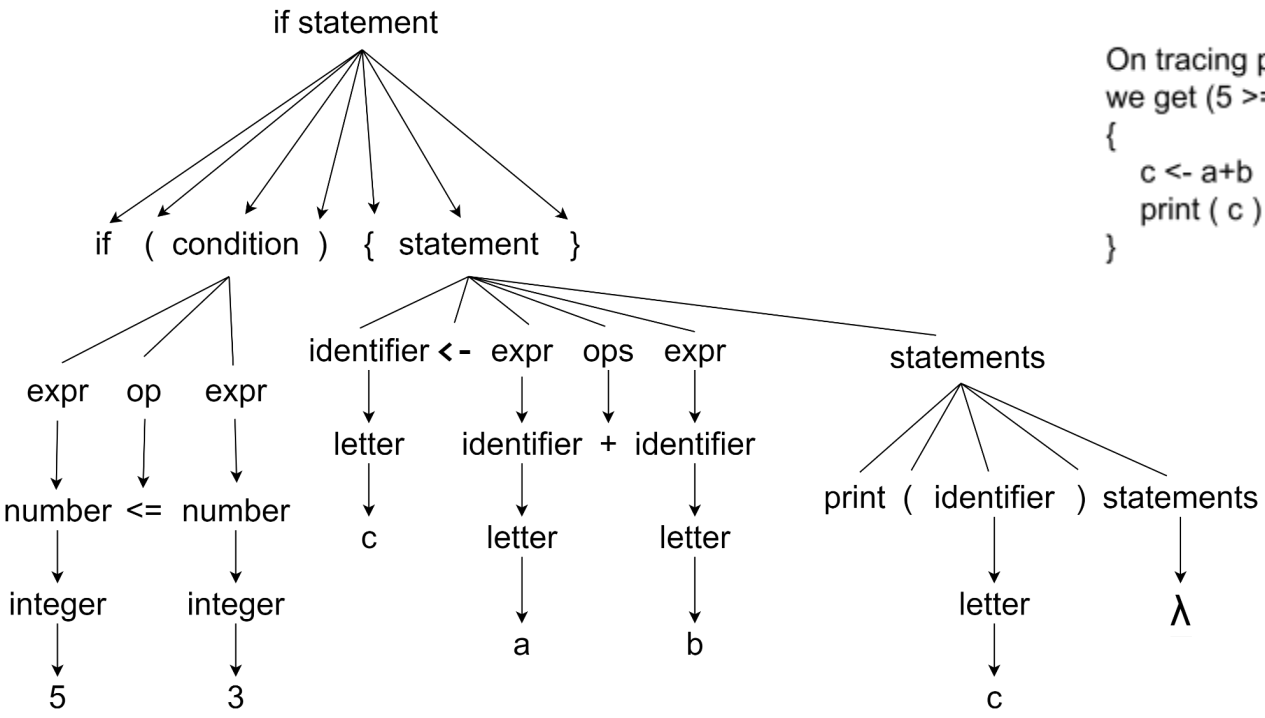
Stack	Implementation	Production rule
if_statements	if(5>=3){c=a+b print(c)}	if_statement → if (condition) { statements}
if (condition) {statements }	if(5>=3){c=a+b print(c)}	Pop if, Pop(
condition) {statements}	5>=3){c=a+b print(c)}	condition →expr op expr
expr op expr){statements }	5>=3){c=a+b print(c)}	expr->numeric
numeric op expr){statements }.	5>=3){c=a+b print(c)}	numeric->integer
integer op expr) {statements }.	5>=3){c=a+b print(c)}	integer->5
5 op expr) {statements }.	5>=3){c=a+b print(c)}	Pop 5
op expr) {statements }.	>=3){c=a+b print(c)}	Op- > >=
>= expr) {statements }.	>=3){c=a+b print(c)}	Pop >=

expr) {statements }	3){c=a+b print(c)}	expr->numeric
numeric){statements}	3){c=a+b print(c)}	numeric->integer
integer){statements}	3){c<-a+b print(c)}	integer->3
3){statements}	3){c<-a+b print(c)}	Pop 3
){statements}	){c<-a+b print(c)}	Pop(
{statements}	{c<-a+b print(c)}	Pop{
statements}	c<-a+b print(c)}	statements → identifier <- expr ops expr statements

identifier <- expr ops expr statements}	c<-a+b print(c)}	identifier->letter
letter <- expr ops expr statements}	c<-a+b print(c)}	letter->c
c <- expr ops expr statements}	c<-a+b print(c)}	Pop c
<- expr ops expr statements}	<-a+b print(c)}	Pop <-
expr ops expr statements}	a+b print(c)}	expr->identifier
identifier ops expr statements}	a+b print(c)}	identifier->letter
letter ops expr statements}	a+b print(c)}	letter->a
a ops expr statements}	a+b print(c)}	Pop a
ops expr statements}	+b print(c)}	ops->+
+expr statements}	+b print(c)}	Pop +
expr statements}	b print(c)}	expr->identifier
identifier statements}	b print(c)}	identifier->letter
letter statements}	b print(c)}	letter->b
b statements}	b print(c)}	Pop b
statements}	print(c)}	statements->print(<identifier>) statements
print(identifier) statements}	print(c)}	Pop print
(identifier) statements}	(c)}	Pop (
identifier) statements}	c)}	identifier->letter
Letter)statements}	c)}	letter->c
c)statements}	c)}	Pop c
)statements}	)}	Pop (

statements}	}	statements->$
$}	}	Pop }
$$		String Accepted

Step5->Create a Parse tree to find whether a string is readable



The following list of features have been included in the parser

1)Syntax checking for Arithmetic, Logical and Relational Expression

2)If, if….else and nested if statements

3)Validation of Unary Operators

4)Validation for loops

For loop

While loop

5)Unbalanced Parentheses

Semantic Analysis

The semantics of a language provide meaning to its constructs, like tokens and syntax structure.
Semantics help to interpret their types, and their relations with each other.
Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not

The semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interprets symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not

CFG+semantics rule=Syntax Directed Derivation
For Example:

int a-> ”value”

The above declaration won’t show error in the lexical analyzer and syntax analyzer, as it is lexically and syntactically correct, but it should give an error as a string is assigned to int

Semantics Analysis is the task to check the declaration in which statement and structure and data type are supposed to be used.

It includes the following problems:
Type checking-Data types and the declaration and assignment
Scope-Resolution-Need to find the correct scope in which variable to use
Flow-Control->Control structure must be used properly

Symbol Table
It contains fields:

1) Name :identifier name
2) Token: token is constant or identifier

3) Type :type of identifier whether int,float,char
4) Scope: Scope could be global or function
5) Scope-id: Unique value is given to each block of code
6) Function-id :Unique for each functions

Syntax of the source language R

An R program is made up of three things: Variables, Comments, and Keywords.

- R does not have a command for declaring a variable. A variable is created the moment you first assign a value to it.
- In many other programming languages, we commonly use = as an assignment operator. In R, we can use both = and <-.
- However, <- is preferred in most cases because the = operator can be forbidden in some context in R.
- The operator <- can be used anywhere, whereas the operator = is only allowed at the top level
- While assigning a value to a variable, we use the <- sign.
  To output (or print) the variable value, just type the variable name.

  Ex:
  ```
  name <- "John"
  age <- 40
  name
  age
  ```

  Out:
  ```
  [1] "John"
  [1] 40
  ```

- To output text in R, use single or double quotes and to output numbers, just type the number.

  Ex:
  ```
  "Hello World!"
  55
  ```

  Out:
  ```
  [1] "Hello World!"
  [1] 55
  ```

- To output integers, we need to represent them by the letter L.

  Ex:2L,40L,1000L.

- Comments start with a #. When executing the R-code, R will ignore anything that starts with #.

  Ex:
  ```
  # This is a comment
  #More than a line
  "Hello World!"

  Out:  [1] "Hello World!"
  ```

- A keyword can't be used as a variable name, function name, etc.
  Ex: Keywords in R are if, else, while, for, function, break, TRUE, FALSE, etc.

- To call a function, use the function name followed by parenthesis, like my_function().

  Ex:
  ```
  my_function() <- function(){
  print("Yay!!")
  }
  my_function()

  Out:  [1] Yay!!
  ```

- Variables that are created outside of a function are known as global variables. Global variables can be used by everyone, both inside of functions and outside.

  Ex:
  ```
  txt <-"hot"
  my_function <-function(){
  paste("tea is",txt)
  }
  my_function()
  ```

Out: [1] tea is hot

Syntax of the target language C++

- C++ language defines several headers, which contain information that is either necessary or useful to your program. Header files add functionality to C++ programs.
- #include <iostream> is a header file library that lets us work with input and output objects
- The header using namespace std means that we can use names for objects and variables from the standard library.

Ex:
```
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello World!";
    return 0;
}
```

- C++ ignores whitespaces.
- int main() is the main function where program execution begins. Any code inside curly brackets {} will be executed.
- 'cout' is an object used together with the insertion operator (<<) to output/print text.

  Ex:
```
cout << "Hello World!";
```
- Every C++ statement ends with a semicolon ';'

  Ex:
```
x = y;
y = y + 1;
```

```
add(x, y);
```

- Single-line comments begin with // and stop at the end of the line.
  Ex:

```
cout << "Hello World!"; //prints the statement
return 0;
```

- return 0; terminates main( ) function and makes it to return value 0.
- A block is a set of logically connected statements that are surrounded by opening and closing braces.

Ex:

```
{
        cout << "Hello World!";
        return 0;
}
```

- To create a variable, you must specify the type and assign it a value
  type variable = value;

Ex:

```
int myNum = 6;
double myFloatNum = 6.01;
char myLetter = 'A';
```

- The cout object is used together with the << operator to display variables.

Ex:

```
int myAge = 35;
cout << "I am " << myAge << " years old.";
```

- We will use ‘cin’ to get user input. ‘cin’ is a predefined variable that reads data from the keyboard with the extraction operator >>.

Ex: int x;
cin >> x;

- Keywords may not be used as constant or variable or any other identifier names.
  Ex: if, else, for, return, case, break, etc

Implementation of Compiler for R:

A compiler implements a formal transformation from a high-level source program to a low-level target program. Compiler design can define an end-to-end solution or tackle a defined subset that interfaces with other compilation tools e.g. preprocessors, assemblers, linkers.

Overview

Implementation part of compiler breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program.

Lexical Analyzer for R language:

Lexical analysis is the first phase of a compiler. Lexical analysis is the process of converting a sequence of characters from source program into a sequence of tokens. A program which performs lexical analysis is termed as a lexical analyzer(lexer), tokenizer or scanner.
Lexical analysis consists of two stages of processing which are as follows:
• Scanning
• Tokenization

The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer.

CODE:

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
#include<stdlib.h>
#define SIZE 20
void display();
struct DataItem {
   char data[20];
   int key;
   char type[15];
};
struct DataItem* hashArray[SIZE];
struct DataItem* obj1;
struct DataItem* obj2;
int search(int);
int hashCode(int key) {
   return key % SIZE;
}

void insert(int key,char *data,char *type) {

   struct DataItem *obj2 = (struct DataItem*) malloc(sizeof(struct
DataItem));
   strcpy(obj2->data,data);
   obj2->key = key;
   strcpy(obj2->type,type);
   int hashIndex = hashCode(key);
    while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1)
{
     ++hashIndex;
       hashIndex %= SIZE;
   }

   hashArray[hashIndex] = obj2;
}

char keyword[5][30]={"for","while","if","else","print"};
char id[20], num[10],rel[5];
```

```
int check_keyword(char s[])
{
int i;
for(i=0;i<5;i++)
if(strcmp(s,keyword[i])==0)
return 1;
return 0;
}
int convertkey(char *);
int main()
{int k=0;
obj1 = (struct DataItem*) malloc(sizeof(struct DataItem));
obj1->key = -1;
FILE *fp1,*fp2;
char c;
int state=0;
int i=0,j=0,t=0;
fp1=fopen("x.txt","r");//input file containing src prog
fp2=fopen("y.txt","w");//output file name

while((c=fgetc(fp1))!=EOF)
{
switch(state)
{
case 0: if(isalpha(c)){
state=1; id[i++]=c;}
else if(c=='.'){
    state=1; id[i++]=c;
}else if(c=='_'){
     state=1; id[i++]=c;
}
else if(isdigit(c)){
state=3; num[j++]=c;}
else if(c=='<' || c=='>'){
rel[t]=c;
state=5;
t++;
}
```

```
else if(c=='=' || c=='!')
{
rel[t]=c;
state=8;
t++;
}
else if(c=='/')
state=10;
else if(c==' ' || c=='\t' || c=='\n')
state=0;
else
fprintf(fp2,"\n%c",c);
break;
case 1:if(isalnum(c)){
state=1; id[i++]=c;
}
else{
id[i]='\0';
if(check_keyword(id)){

fprintf(fp2," \n %s  ",id);

}
else{
fprintf(fp2,"\n %s ",id);
// call a function which stores id in symbol table
k=convertkey(id);
if(k!=0){
    int found=search(k);
    if(found==0){
        insert(k,id,"identifier");
    }
}



}

state=0;
i=0;
ungetc(c,fp1);
```

```
}
break;
case 3:if(isdigit(c)){
num[j++]=c;
state=3;
}
else{
num[j]='\0';
fprintf(fp2," \n%s ",num);
state=0;
j=0;
ungetc(c,fp1);

}
break;
case 5:if(c=='='){
  rel[t]=c;
  t++;
  rel[t]='\0';
fprintf(fp2,"\n%s ",rel);
t=0;

state=0;
}else if(c=='-'){
    rel[t]=c;
  t++;
  rel[t]='\0';
fprintf(fp2,"\n%s  ",rel);
t=0;

}
else{
    rel[t]='\0';
fprintf(fp2,"\n%s  ",rel);

state=0;
ungetc(c,fp1);
t=0;
}
break;
```

```
case 8:if(c=='='){
  rel[t]=c;
  t++;
  rel[t]='\0';
fprintf(fp2,"\n%s  ",rel);
t=0;
state=0;
}
else{
ungetc(c,fp1);
state=0;
}
break;
case 10:if(c=='*')
state=11;
else
fprintf(fp2,"\n invalid lexeme");
break;
case 11: if(c=='*')
state=12;
else
state=11;
break;
case 12:if(c=='*')
state=12;
else if(c=='/')
state=0;
else
state=11;
break;

}//End of switch
}//end of while
if(state==11)
fprintf(fp2,"comment did not close");
fclose(fp1);
fclose(fp2);
display();
return 0;
}
```

```
void display() {
   int i = 0;
      printf("%s","SRNO\t\t\tID\t\t\ttype");
   for(i = 0; i<SIZE; i++) {

      if(hashArray[i] != NULL){
         printf("\n");
         printf("
%d%s%s%s%s",hashArray[i]->key,"\t\t\t",hashArray[i]->data,"\t\t\t",hashArr
ay[i]->type);

      }

   }

   printf("\n");
}
int convertkey(char *id){
    int sum =0;
    for(int i=0;i< strlen(id);i++){

       int num =(int)id[i]*(i+1);
       sum= sum+num;
    }

   return sum%2069;
}
int search(int key) {

   int hashIndex = hashCode(key);
      while(hashArray[hashIndex] != NULL) {
        if(hashArray[hashIndex]->key == key)
         return 1;
       ++hashIndex;
     hashIndex %= SIZE;
   }

   return 0;
}
```

```
if
(
a
>=
b
)
{
c
<-
d
+
e
}
for
(
i
<-
7
)
{
print
(
1
)
}
```

```
 r
<-

11
 .t
<-

117
 if
(
 rf
<
3
)
{
 print
(
"
 lehar
"
)
}
```

Error detection in Compiler:

In this phase of compilation, all possible errors made by the user are detected and reported to the user in the form of error messages. This process of locating errors and reporting them to users is called the Error Handling process.

Functions of an Error handler

- Detection
- Reporting
- Recovery

Parser/Syntax Analysis for R language:

Syntax analysis is the second phase of compiling. Syntax analysis is also known as parsing.Parsing is the pare non-descentrocess of determining whether a string of tokens can be generated by a grammar.It is performed by syntax analyzer which can also be termed as parser.In addition to construction of the parse tree, syntax analysis also checks and reports syntax errors accurately. Parser is a program that obtains tokens from lexical analyzer and constructs the parse tree which is passed to the next phase of the compiler for further processing. Parser implements context free grammar for performing error checks.It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

CODE:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
int i=0,top=0;
```

```
char stack[2000][50],ip[20];
int nttop=0;

void push(char c[])
{
    if (top>=2000)
        printf("Stack Overflow");
    else
        strcpy(stack[top++],c);

}




void pop(void)
{
    if(top<0)
        printf("Stack underflow");
    else
        top--;
}

void error(void)
{
printf("\n\nSyntax Error!!!! String is invalid\n");
getch();
exit(0);
}

int main()
{
int n;
char filename[] = "y.txt";
FILE *file = fopen ( filename, "r" );
int m=0;
char line1[1000];
while(fgets(line1,sizeof line1,file)!= NULL){

    m++;
```

```
}

char line[m+1][50];
char line3[m+1][50];
strcpy(line3[m],"$");
strcpy(line3[m+1],"/0");
int l;

push("$");
push("S");
char filename1[] = "y.txt";
FILE *file1 = fopen ( filename1, "r" );
char line2[256];
int t=1;

while(m+1>= i)
 {fgets(line2,sizeof(line2),file1);

    l=0;
    l=strlen(line2);
    line2[l-1]='\0';
    strcpy(line[i],line2);
    if(i>=m && strcmp(stack[top-1],"$")==0 )
  { t=0;
    printf("\n\n Successful parsing of string \n");
    break;
  }
  else

    if(strcmp(line[i],stack[top-1])==0)
    {
       printf("\n    match of %s \n", line[i]);
      pop();
    }
    else
    {
        if(strcmp(stack[top-1],"S")==0)
        {
        if(strcmp(line[i], "if")==0){
```

```
        printf(" \n S->if(condition){statement}S");

          pop();
          push("S");
          push("}");
          push("statement");
          push("{");
          push(")");
          push("condition");
          push("(");


}else if(strcmp(line[i],"for")==0){
    printf("\n S->for ( identifier <- numeric){ statement }S");
    pop();
    push("S");
    push("}");
    push("statement");
    push("{");
    push(")");
    push("numeric");
    push("<-");
    push("identifier");
    push("(");

}else if(atoi(line[i])==50){
    printf("\n S->identifier <-identifier ops identifier ");
   pop();
   push("identifier");
   push("ops");
   push("identifier");
   push("<-");


}
else{
  printf("\n S-> $")  ;
  pop();
}
}
```

```
       else if(strcmp(stack[top-1],"condition")==0){
           if(atoi(line[i])){
            printf("  condition->exp(numeric) op expr");
            pop();
            push("expr");
            push("op");



       }else{
           printf(" condition->exp(identifier) op expr");
           pop();
            push("expr");
            push("op");

            }
      }else if(strcmp(stack[top-1],"op")==0)
      {

         if((strcmp(line[i],">")==0) || (strcmp(line[i],"<")==0)||
(strcmp(line[i],">=")==0) ||
(strcmp(line[i],"!=")==0)||(strcmp(line[i],"==")==0))
         {
             printf( " \n op->  %s",line[i]);
             pop();
         }
      }else if(strcmp(stack[top-1],"expr")==0){
          pop();
          if(atoi(line[i])==0){
              printf("\nexpr-> identifier %s", line[i]);
          }else {
              printf("\n %sexpr->numeric", line[i]);
          }
      }else if(strcmp(stack[top-1],"ops")==0){
          if((strcmp(line[i],"+")==0) || (strcmp(line[i],"-")==0)||
(strcmp(line[i],"*")==0) ||
(strcmp(line[i],"%")==0)||(strcmp(line[i],"/")==0))
         {
             printf( " \n op->  %s",line[i]);
```

```
                pop();
            }  else{
                pop();
                pop();
            }

        }

        else if(strcmp(stack[top-1],"statement")==0){
            printf("%s%s",line[i],stack[top-1]);
            if(atoi(line[i])==0 && strcmp(line[i],"print")!=0 &&
strcmp(line[i],"}")!=0 && strcmp(line[i],"<-")){
                printf("\nstatment-> identifier <- expr ops expr
statement");
                pop();
                push("statement");
                push("expr");
                push("ops");
                push("expr");
                push("<-");
            }else if(strcmp(line[i],"print")==0){
                printf("\nstatement-> print( identifier ) statement");
                pop();
                push("statement");
                push(")");
                push("identifier");
                push("(");

            }else if(strcmp(line[i], "}")==0){
                printf("\n statement-> $");
                pop();
                pop();
            }

        }else if(strcmp(stack[top-1],"identifier")==0){
            if(atoi(line[i])==0){
                printf("\n identifier found");
                pop();

            }
```

```
          }else if(strcmp(stack[top-1],"numeric")==0){
              if(atoi(line[i])!=0){
                  printf("\n numeric found");
                  pop();

              }

          } else {
           printf("String invalid");
           break;
        }
    }

i++; }


}
```

OUTPUTS:

```
    match of {
cstatement
statment-> identifier <- expr ops expr statement
    match of <-

expr-> identifier d
 op->  +
expr-> identifier e}statement
 statement-> $
 S->for ( identifier <- numeric){ statement }S
    match of (

 identifier found
    match of <-

 numeric found
    match of )

    match of {
printstatement
statement-> print( identifier ) statement
    match of (

 identifier found
    match of )
}statement
 statement-> $
 S-> $

 Successful parsing of string


...Program finished with exit code 0
```

Symbol Table

Symbol table is an important data structure used in a compiler.Symbol table is used to store the information about the occurrence of various entities such as objects, classes, variable name, interface, function name etc. it is used by both the analysis and synthesis phases.

The symbol table used for following purposes:

- It is used to store the name of all entities in a structured form at one place.
- It is used to verify if a variable has been declared.
- It is used to determine the scope of a name.
- It is used to implement type checking by verifying assignments and expressions in the source code are semantically correct.

Intermediate Code Generation:

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation eg. postfix notation. Intermediate code tends to be machine independent code.
We generated the intermediate code by making use of the parse tree indirectly.

● We have written a quadruple form of code to a file based on the syntax we are currently parsing in the program.
● We used multiple stacks for this process.
● We maintained a stack of operators and identifiers we parse. We also have a stack for maintaining the labels.
● When a production symbol is completely parsed, we pop from the stack and print it to a file according to the symbol we parsed.
● We make use of the labels stack when dealing with the while loop and switch construct. Appropriately consuming the stack to print labels for loop and case statements.

● Also, an arithmetic code generation function, which generates suitable code for any required operator.

Insights of Building a Compiler

We have chosen Ubuntu as our Linux environment. We have used packages such as Yacc and lex to build this project.We faced many issues while building a Lexical Analyzer for this language.We faced many issues while generating tokens accordingly.Most of our time was spent building this parser.We developed this parsing code in lines covering a few CFG's.We couldn't upgrade the parser as we are facing many bugs later.we tried working so much on intermediate code generation but we had to face a lot of errors.

Conclusion

In a compiler the process of Intermediate code generation is independent of the machine and the process of conversion of Intermediate code to target code is independent of language used. Thus we have done the front end of the compilation process. It includes 3 phases of compilation lexical analysis, syntax analysis and semantic analysis which is then followed by intermediate codegeneration.In computer programming, the translation of source code into object code by a compiler. This report outlines the analysis phase in compiler construction. In its implementation and source language is converted to assembly level language.