UNIVERSITY INSTITUTE OF ENGINEERING AND TECHNOLOGY, PANJAB UNIVERSITY, CHANDIGARH

**COMPILER DESIGN**

****

**Lab File**

Roll No: UE163095

Computer Science and Engineering | 6th SEM

Submitted By: Shivank Bali

Submitted To: Ms. Surbhi

**INDEX**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **S.No.** | **Page No.** | **Date** | **Practical** | **Remarks** |
| **1** | 1-5 | 16/1/19 | Introduction to compiler |  |
| **2** | 6-10 | 30/1/19 | Design a program of lexical analyzer |  |
| **3** | 11-14 | 27/2/19 | Design a code to convert a regular expression to NFA |  |
| **4** | 15-20 | 20/3/19 | Design a code to convert NFA to DFA |  |
| **5** | 21-23 | 27/3/19 | Design a code to implement first and follow of the grammar |  |
| **6** | 24-25 | 10/4/19 | Design a code to rectify left recursion |  |
| **7** | 26-28 | 10/4/19 | Design a code to right factor the given grammar |  |
| **8** | 29-31 | 16/4/19 | Design a code to Parse a string using recursive descent parser |  |

**Practical 1**

Introduction to the compilers

**Compilers**

Compiler is a computer programs which transforms the programs/ source codes written in high level languages to the target languages (low level languages). For eg. in case of C language, the compiler converts the source code in C language to the Assembly language.

Compilers are a type of translator that support digital devices, primarily computers.

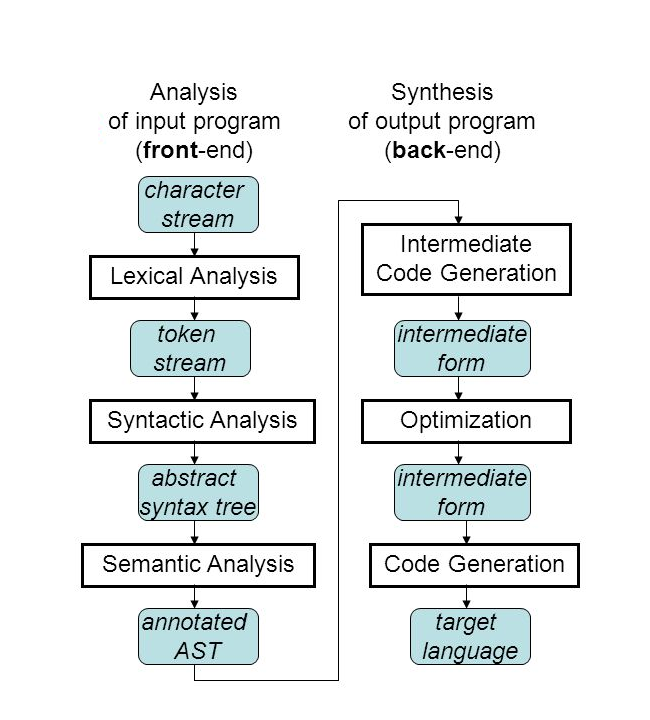
**Types of Compilers:**

1. **Native code compiler:** The compiler used to compile a source code for same type of platform only. The output generated by this type of compiler can only be run on the same type of computer system and OS that the compiler itself runs on.
2. **Cross compiler:** The compiler used to compile a source code for different kinds platform. Used in making software’s for embedded systems that can be used on multiple platforms.
3. **Source to source compiler:** the compiler that takes high-level language code as input and outputs source code of another high- level language only.Unlike other compilers which convert high level language into low level machine language, it can take up a code written in Pascal and can transform it into C-conversion of one high level language into another high level language having same type of abstraction . Thus, it is also known as transpiler.
4. **One pass compiler:** It is a type of compiler that compiles the whole process in only one-pass.
5. **Threaded code compiler:** The compiler which simply replace a string by an appropriate binary code.
6. **Incremental compiler:** The compiler which compiles only the changed lines from the source code and update the object code.
7. **Source compiler:** The compiler which converts the source code high level language code in to assembly language only.

**Phases of compilers**

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.

**Token:** A token is the smallest element(character) of a computer language program that is meaningful to the compiler. The parser has to recognize these as tokens: identifiers, keywords, literals, operators, punctuators, and other separators.



**Figure 1 : Phases of Compiler**

### **Lexical Analysis**

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

### **Syntax Analysis**

The next phase is called the syntax analysis or **parsing**. 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.



**Figure 2 : Working of compiler**

### **Semantic Analysis**

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

### **Intermediate Code Generation**

After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

### **Code Optimization**

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

### **Code Generation**

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

### **Symbol Table**

It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

**Compilers vs interpreter**

|  |  |  |
| --- | --- | --- |
| **No** | **Compiler** | **Interpreter** |
| **1** | Compiler Takes Entire program as input | Interpreter Takes Single instruction as input. |
| **2** | Intermediate Object Code is Generated | No Intermediate Object Code is Generated |
| **3** | Conditional Control Statements are Executes faster | Conditional Control Statements are Executes slower |
| **4** | Memory Requirement: More (Since Object Code is Generated) | Memory Requirement is Less |
| **5** | Program need not be compiled every time | Every time higher level program is converted into lower level program |
| **6** | Errors are displayed after entire program is checked | Errors are displayed for every instruction interpreted (if any) |
| **7** | Example: C, C++ Compiler | Example: Python |

Table 1 : Difference Between Compiler vs Interpreter

**Practical 2**

Introduction to Lexical Analyser and its Implementation

**Lexical Analyser**

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. 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. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands.

**Lexical Analyser In Python**

In [1]:

# for identifying header files in the code

header = {'.h': 'header file'}

header\_keys = header.keys()

sp\_header\_files = {'<stdio.h>':'Standard Input Output Header',

'<conio.h>':'console Input Output Header',

'<string.h>':'String Manipulation Library'}

In [2]:

# for identifying the datatypes available

datatype = {'int': 'Integer','float' : 'Floating Point', 'char': 'Character','long': 'long int'}

datatype\_keys = datatype.keys()

In [3]:

# for identifying operators in the input code

operator\_list = { '=': 'Assignment Operator','+': 'Additon Operator', '-' : 'Substraction Operator',

'/' : 'Division Operator', '\*': 'Multiplication Operator', '++' : 'increment Operator',

'--' : 'Decrement Operator'}

operator\_keys = operator\_list.keys()

In [4]:

#for identifying other keywords in the code

keyword = {'return' : 'keyword that returns a value from a block'}

keyword\_keys = keyword.keys()

In [5]:

# for statement termination in the code

delimiter = {';':'terminator symbol semicolon (;)'}

delimiter\_keys = delimiter.keys()

In [6]:

#for identifying different blocks in the code

blocks = {'{' : 'Blocked Statement Body Open', '}':'Blocked Statement Body Closed'}

block\_keys = blocks.keys()

In [7]:

# for recognising the builtin functions in the code

builtin\_functions = {'printf':'printf prints its argument on the console'}

In [9]:

# for identifying puctuations

import string

non\_identifiers = list(string.punctuation)

In [10]:

# for digits in code

numerals = ['0','1','2','3','4','5','6','7','8','9','10']

In [11]:

# opening file

f = open('code.cpp','r')

# Flags

dataFlag = False

i = f.read()

count = 0

program = i.split('\n')

for line in program:

count = count+1

tokens = line.split(' ')

print(tokens)

print("Tokens are",tokens)

print("Line #",count,'properties \n')

for token in tokens:

if token in block\_keys:

print( blocks[token] )

if token in operator\_keys:

print( "Operator is: ", operator\_list[token] )

if '.h' in token:

print( "Header File is: ",token, sp\_header\_files[token] )

if '()' in token:

print( "Function named", token )

if dataFlag == True and (token not in non\_identifiers) and ('()'not in token):

print( "Identifier: ",token )

if token in datatype\_keys:

print( "type is: ", datatype[token] )

dataFlag = True

if token in keyword\_keys:

print( keyword[token] )

if token in delimiter:

print( "Delimiter" , delimiter[token] )

if '#' in token:

match = re.search(r'#\w+', token)

print( "Header", match.group() )

if token in numerals:

print( token,type(int(token)) )

if token in builtin\_functions:

print( 'Builtin Function used is: ', token )

dataFlag = False

print( "\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_")

f.close()

['#include', '<stdio.h>']

Tokens are ['#include', '<stdio.h>']

Line # 1 properties

Header #include

Header File is: <stdio.h> Standard Input Output Header

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

['#include', '<conio.h>']

Tokens are ['#include', '<conio.h>']

Line # 2 properties

Header #include

Header File is: <conio.h> console Input Output Header

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

['int', 'main(){']

Tokens are ['int', 'main(){']

Line # 3 properties

type is: Integer

Function named main(){

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

['int', 'a', '=', '4;']

Tokens are ['int', 'a', '=', '4;']

Line # 4 properties

type is: Integer

Identifier: a

Operator is: Assignment Operator

Identifier: 4;

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

['printf("%d",a);']

Tokens are ['printf("%d",a);']

Line # 5 properties

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

['return', '0;']

Tokens are ['return', '0;']

Line # 6 properties

keyword that returns a value from a block

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

['}']

Tokens are ['}']

Line # 7 properties

Blocked Statement Body Closed

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

['']

Tokens are ['']

Line # 8 properties

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In [ ]:

**INPUT CODE :**

#include <stdio.h>

#include <conio.h>

int main(){

int a = 4;

printf("%d",a);

return 0;

}

**Practical 3**

Regular Expression to NFA

**Regular Expression**

A regular expression is a method used in programming for pattern matching. Regular expressions provide a flexible and concise means to match strings of text.

**Non-Deterministic Finite Automaton (NFA/NDFA)**

In NDFA, for a particular input symbol, the machine can move to any combination of the states in the machine. In other words, the exact state to which the machine moves cannot be determined. Hence, it is called Non-deterministic Automaton. As it has finite number of states, the machine is called Non-deterministic Finite Machine or Non-deterministic Finite Automaton

**Regular Expression to NFA converter in Python:**

In [2]:

reg = str(input("Enter the Regex :"))

Enter the Regex :ab\*cd\*|abdc

In [8]:

import string

varlist = []

var = list(reg)

for i in var:

if i not in string.punctuation:

varlist.append(i)

varlist = set(varlist)

varlist = list(varlist)

varlist.sort()

varliste = varlist.copy()

varliste.append('e')

print(varlist,varliste)

['a', 'b', 'c', 'd'] ['a', 'b', 'c', 'd', 'e']

In [9]:

i = 0

j = 0

f=[]

q=[]

base = 0

q.append({})

while(i<len(reg)):

#only symbol

if reg[i] in varliste :

if len(reg) > i+1:

if (not reg[i+1] == '\*'):

q[j][reg[i]] = []

q[j][reg[i]].append(j+1)

j=j+1

if len(q) <= j:

q.append({})

else:

if len(q) <= j:

q.append({})

q[j][reg[i]] = []

q[j][reg[i]].append(j+1)

j=j+1

if len(q) <= j:

q.append({})

#for | operator

if len(reg) > i+1 :

if reg[i] == '|' and reg[i+1] in varlist:

f.append(j)

if len(q) <= j:

q.append({})

if 'e' in q[base]:

q[base]['e'].append(j+1)

else:

q[base]['e'] = []

q[base]['e'].append(j+1)

j+=1

if len(q) <= j:

q.append({})

#For \* operator

if len(reg) > i+1 :

if reg[i] in varlist and reg[i+1] == '\*':

if reg[i] in q[j]:

q[j][reg[i]].append(j+1)

else:

q[j][reg[i]] = []

q[j][reg[i]].append(j+1)

if 'e' in q[j]:

q[j]['e'].append(j+1)

else:

q[j]['e'] = []

q[j]['e'].append(j+1)

j = j+1

if len(q) <= j:

q.append({})

if 'e' in q[j]:

q[j]['e'].append(j-1)

else:

q[j]['e'] = []

q[j]['e'].append(j-1)

i+=1

i+=1

if 'e' in q[j]:

q[j]['e'].append(j+1)

else:

q[j]['e'] = []

q[j]['e'].append(j+1)

q.append({})

j+=1

for i in f:

if 'e' in q[i]:

q[i]['e'].append(j-1)

else:

q[i]['e'] = []

q[i]['e'].append(j-1)

In [10]:

print(q)

[{'a': [1], 'e': [5]}, {'b': [2], 'e': [2]}, {'e': [1], 'c': [3]}, {'d': [4], 'e': [4]}, {'e': [3, 9]}, {'a': [6]}, {'b': [7]}, {'d': [8]}, {'c': [9]}, {'e': [10]}, {}]

In [11]:

def variable(x,k):

if x in q[k].keys():

return q[k][x]

else:

return "-"

In [12]:

print ("State\t", ("\t".join(map(str,varliste))))

for i in range(len(q)):

print(i,"\t","\t".join( str(variable(var,i)) for var in varliste ))

State a b c d e

0 [1] - - - [5]

1 - [2] - - [2]

2 - - [3] - [1]

3 - - - [4] [4]

4 - - - - [3, 9]

5 [6] - - - -

6 - [7] - - -

7 - - - [8] -

8 - - [9] - -

9 - - - - [10]

10 - - - - -

In [ ]:

**Practical 4**

Non-Deterministic Finite Automata to Deterministic Finite Automata

**DFA :** Deterministic Finite Automatais a finite-state machine that accepts or rejects strings of symbols and only produces a unique computation (or run) of the automaton for each input string. *Deterministic* refers to the uniqueness of the computation.

In [1]:

reg = str(input("Enter the Regex :"))

Enter the Regex :ab\*cd\*|adbc

In [2]:

import string

varlist = []

var = list(reg)

for i in var:

if i not in string.punctuation:

varlist.append(i)

varlist = set(varlist)

varlist = list(varlist)

varlist.sort()

varliste = varlist.copy()

varliste.append('e')

print(varlist,varliste)

['a', 'b', 'c', 'd'] ['a', 'b', 'c', 'd', 'e']

In [3]:

i = 0

j = 0

f=[]

q=[]

base = 0

q.append({})

while(i<len(reg)):

#only symbol

if reg[i] in varliste :

if len(reg) > i+1:

if (not reg[i+1] == '\*'):

q[j][reg[i]] = []

q[j][reg[i]].append(j+1)

j=j+1

if len(q) <= j:

q.append({})

else:

if len(q) <= j:

q.append({})

q[j][reg[i]] = []

q[j][reg[i]].append(j+1)

j=j+1

if len(q) <= j:

q.append({})

#for | operator

if len(reg) > i+1 :

if reg[i] == '|' and reg[i+1] in varlist:

f.append(j)

if len(q) <= j:

q.append({})

if 'e' in q[base]:

q[base]['e'].append(j+1)

else:

q[base]['e'] = []

q[base]['e'].append(j+1)

j+=1

if len(q) <= j:

q.append({})

#For \* operator

if len(reg) > i+1 :

if reg[i] in varlist and reg[i+1] == '\*':

if reg[i] in q[j]:

q[j][reg[i]].append(j+1)

else:

q[j][reg[i]] = []

q[j][reg[i]].append(j+1)

if 'e' in q[j]:

q[j]['e'].append(j+1)

else:

q[j]['e'] = []

q[j]['e'].append(j+1)

j = j+1

if len(q) <= j:

q.append({})

if 'e' in q[j]:

q[j]['e'].append(j-1)

else:

q[j]['e'] = []

q[j]['e'].append(j-1)

i+=1

i+=1

if 'e' in q[j]:

q[j]['e'].append(j+1)

else:

q[j]['e'] = []

q[j]['e'].append(j+1)

q.append({})

j+=1

for i in f:

if 'e' in q[i]:

q[i]['e'].append(j-1)

else:

q[i]['e'] = []

q[i]['e'].append(j-1)

In [5]:

#Function for Displaying NFA Transition table

def variable(x,k):

if x in q[k].keys():

return q[k][x]

else:

return "-"

In [6]:

# Displaying NFA Transition table

print ("State\t", ("\t".join(map(str,varliste))))

for i in range(len(q)):

print(i,"\t","\t".join( str(variable(var,i)) for var in varliste ))

State a b c d e

0 [1] - - - [5]

1 - [2] - - [2]

2 - - [3] - [1]

3 - - - [4] [4]

4 - - - - [3, 9]

5 [6] - - - -

6 - - - [7] -

7 - [8] - - -

8 - - [9] - -

9 - - - - [10]

10 - - - - -

In [7]:

#Function for finding all possible e moves from given state or list of states

def e\_moves(q,l\_val):

st=[]

for x in l\_val:

state\_set = set([x])

if 'e' in q[x].keys():

state\_set.update(set(q[x]['e']))

t = set([x])

while not t == state\_set:

t = state\_set.copy()

for i in t:

if 'e' in q[i].keys():

state\_set.update(set(q[i]['e']))

st.extend(list(state\_set))

st = set(st)

st = list(st)

st.sort()

return st

In [8]:

#Function generating state for moves with given symbol

def moves(q, l\_val, move):

state\_set = set()

l\_val = e\_moves(q,l\_val)

for x in l\_val:

if move in q[x].keys():

state\_set.update(set(q[x][move]))

state\_set = list(state\_set)

state\_set = e\_moves(q,state\_set)

return state\_set

In [11]:

#Selection of starting and Final states

start\_st = []

start\_st.append(0)

end\_st = (len(q)-1)

start\_st = e\_moves(q,start\_st)

In [12]:

dfa\_st = [] #list for states of dfa

dfa\_vis = [] #list for unvisited states of dfa

dfa\_q = [] #list for dictionary of transitions from states

dfa\_st.append(start\_st)

dfa\_vis.append(start\_st)

p=-1 # For count of states in dfa

while(dfa\_vis):

curr\_st = dfa\_vis.pop(0) # selecting first state

dfa\_q.append({})

p+=1

for i in varlist: #loop for evaluating moves

new\_st = moves(q,curr\_st,i)

if new\_st:

dfa\_q[p][i] = new\_st

if not (new\_st in dfa\_st):

dfa\_st.append(new\_st)

dfa\_vis.append(new\_st)

In [13]:

# Function for displaying transition table

def variable(x,k):

if x in dfa\_q[k].keys():

return dfa\_q[k][x]

else:

return "-"

In [14]:

#printing transition table for DFA

s=''

for i in varlist:

s += i.center(15)

print ("State".ljust(15), s)

for i in range(len(dfa\_q)):

s = str(dfa\_st[i]).ljust(15)

for var in varlist:

s += str(variable(var,i)).center(15)

print( s, "\tFinal State" if end\_st in dfa\_st[i] else "")

State a b c d

[0, 5] [1, 2, 6] - - -

[1, 2, 6] - [1, 2] [3, 4, 9, 10] [7]

[1, 2] - [1, 2] [3, 4, 9, 10] -

[3, 4, 9, 10] - - - [3, 4, 9, 10] Final State

[7] - [8] - -

[8] - - [9, 10] -

[9, 10] - - - - Final State

In [ ]:

**Practical 5**

Finding first and follow for the given grammar

# Code for First and Follow using the following grammar

**Grammar**

E->TW

W->TW|e

T->FR

R->+FR|e

F->(E)|d

In [1]:

lang = {}

gn = []

n = int(input("Enter number of productions :"))

for j in range(n):

st = str(input("Enter production :"))

u = st.split('->')[0]

gn.append(u)

t = st.split('->')[1]

lis = []

for x in t.split('|'):

lis.append(list(x))

lang[u] = lis

Enter number of productions :5

Enter production :E->TW

Enter production :W->TW|e

Enter production :T->FR

Enter production :R->+FR|e

Enter production :F->(E)|d

In [2]:

def First(x):

first = set()

if x in lang.keys():

for i in lang[x]:

if i[0] in gn:

k=0

while(True):

if k>=len(i):

break

f = First(i[k])

first.update(f)

k+=1

if (not 'e' in f):

break

else:

first.update(i[0])

return first

In [3]:

first\_list = {}

for i in gn:

first\_list[i] = First(i)

In [4]:

print("\tFirst ")

for i in first\_list.keys():

print(i,"\t", ", ".join(t for t in first\_list[i]))

First

E (, d

W (, d, e

T (, d

R +, e

F (, d

In [5]:

follow\_list = {}

follow\_list[gn[0]] = set('$')

for i in range(1,len(gn)):

follow\_list[gn[i]] = set()

print('Initail Follow',follow\_list)

def Follow(x):

for t in lang.keys():

for k in lang[t]:

if x in k:

ind = k.index(x)

if ind+1 == len(k):

follow\_list[x].update(follow\_list[t])

else:

n=ind+1

while(True):

if n>=len(k):

follow\_list[x].update(follow\_list[t])

break

if k[n] not in gn:

follow\_list[x].update(set(k[n]))

break

f = first\_list[k[n]].copy()

if 'e' in f:

f.remove('e')

follow\_list[x].update(f)

n+=1

continue

break

Initail Follow {'E': {'$'}, 'W': set(), 'T': set(), 'R': set(), 'F': set()}

In [6]:

for i in gn:

Follow(i)

In [14]:

print("\tFirst \t\tFollow")

line\_width = 16

for i in gn:

print(i, ", ".join(t for t in first\_list[i]).rjust(12),'\t\t',", ".join(t for t in follow\_list[i]))

First Follow

E (, d ), $

W (, d, e ), $

T (, d (, d, ), $

R +, e ), (, d, $

F (, d (, +, ), d, $

In [ ]:

**Practical 6**

Removing left recursion from the given grammar

**Left Recursion :** In the formal language theory of computer science, **left recursion** is a special case of recursion where a string is recognized as part of a language by the fact that it decomposes into a string from that same language (on the left) and a suffix (on the right)

In [13]:

# Code to input the Grammar

lang = {}

gn = []

n = int(input("Enter number of productions :"))

for j in range(n):

st = str(input("Enter production :"))

u = st.split('->')[0]

gn.append(u)

t = st.split('->')[1]

lis = []

lis2 = []

lr\_fl = 0

for x in t.split('|'):

if x[0] == u:

lr\_fl = 1

t = list(x)

t.remove(u)

lis2.append(t)

else:

lis.append(list(x))

if lr\_fl == 1:

u1 = u + '1'

gn.append(u1)

for i in lis:

i.append(u1)

for i in lis2:

i.append(u1)

lis2.append(['e'])

lang[u] = lis

lang[u1] = lis2

else:

lang[u] = lis

Enter number of productions :1

Enter production :E->E+T|E\*T|T|F

In [14]:

def printGrammar(l,g):

for i in g:

for j in l[i]:

print(i+'->'+''.join(j))

In [15]:

printGrammar(lang,gn)

E->TE1

E->FE1

E1->+TE1

E1->\*TE1

E1->e

In [ ]:

**Practical 7**

Left factoring the given grammar

**Left Factoring :** Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. When it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the productions to defer the decision until we have some enough of the input to make the right choice.

In [16]:

def LeftFactoring(language, nonTerminals):

# Code to input the Grammar

lang = {}

gn = []

for i in nonTerminals:

total = []

for j in language[i]:

total.append(''.join(j))

st = i+'->'+'|'.join(total)

u = st.split('->')[0]

gn.append(u)

t = st.split('->')[1]

lis = []

lis2 = []

lr\_fl = 0

start = {}

comm = []

for x in t.split('|'):

if x[0] in start.keys():

start[x[0]] += 1

else:

start[x[0]] = 1

comm.append(x[0])

temp = 1

for i in comm:

temp += 1

for x in t.split('|'):

if x[0] == i and start[i] <= 1:

lis.append(list(x))

elif x[0] == i and start[i] >1:

tem = []

tem.append(x[0])

tem.append(u + str(temp))

if not (tem in lis):

lis.append(list(tem))

if start[i] <= 1:

comm.remove(i)

lang[u] = lis

temp = 1

for i in comm:

temp+=1

lis\_new = []

u\_new = u+str(2)

for x in t.split('|'):

if x[0] == i and start[i] > 1:

tem = list(x)

tem.remove(i)

if not tem:

tem = ['e']

lis\_new.append(tem)

if lis\_new:

lang[u\_new] = lis\_new

gn.append(u\_new)

return lang,gn

In [17]:

lang = {}

gn = []

n = int(input("Enter number of productions :"))

for j in range(n):

st = str(input("Enter production :"))

u = st.split('->')[0]

gn.append(u)

t = st.split('->')[1]

lis = []

for x in t.split('|'):

lis.append(list(x))

lang[u] = lis

Enter number of productions :1

Enter production :A->Ab|Ac|d|A

In [18]:

l,g = LeftFactoring(lang, gn)

In [19]:

printGrammar(l,g)

A->AA2

A->d

A2->b

A2->c

A2->e

In [ ]:

**Practical 8**

Recursive Descent Parser to parse a string

**Recursive Descent Parser :** It is a kind of Top-Down Parser. A top-down parser builds the parse tree from the top to down, starting with the start non-terminal.

**Grammar**

S → rXd | rZd

X → oa | ea

Z → ai

In [1]:

import numpy as np

In [2]:

class rdParser:

string = ''

i = 0

def \_\_init\_\_(self):

self.string = str(input("Enter the string :"))

self.i = 0

if self.S():

print("String Accepted")

else:

print("String Rejected")

def match(self,t):

if self.string[self.i] == t:

return True

return False

def S(self):

save = self.i

if self.match('r'):

self.i+=1

if(self.X()):

if self.match('d'):

self.i+=1

return True

self.i = save

if self.match('r'):

self.i+=1

if self.Z():

if self.match('d'):

self.i+=1

return True

return False

def X(self):

save = self.i

if self.match('o'):

self.i+=1

if self.match('a'):

self.i+=1

return True

self.i=save

if self.match('e'):

self.i+=1

if self.match('a'):

self.i+=1

return True

return False

def Z(self):

save = self.i

if self.match('a'):

self.i+=1

if self.match('i'):

self.i+=1

return True

return False

In [3]:

t=rdParser()

Enter the string :read

String Accepted

In [4]:

t2=rdParser()

Enter the string :raid

String Accepted

In [5]:

t3=rdParser()

Enter the string :road

String Accepted

In [6]:

t4=rdParser()

Enter the string :rand

String Rejected

In [ ]:

Notebook to html

Html to docx

<https://cloudconvert.com/html-to-docx>