COMPILER CONSTRUCTION LAB

(CSL-323)

PROJECT PROJECT

Group Members

Abdullah Abdul Wahid	**02-134192-015**
Fazeel Zafar	**02-134192-010**

Compiler for Usage by Non-technical Users, with Basic Knowledge

Lecturer(s): Miss Asia Samreen
Lab Engineer: Saba Imtiaz
Abdullah Abdul Wahid
02-134192-015

Assignments for Fall-2021

Assignment 3: (4+4 Marks) (CLO3)
Design a simplified Lexical Analyzer and Parser showing the following steps:

1. **Problem specification:**
2. **Specification of rules and attributes**
3. **Specification of programming Language constructs**
4. **Possible techniques to implement lexical analyzer**
5. **Lexical analyzer implementation**
6. **Available techniques/algorithms to implement Parser**
7. **Implementation of Parser (Table)**

(Submission due on 28th JAN-2022)

You can select your problem from the following areas:

1. Scripting Language construction
2. Language translation
3. Context Free Grammar's Applications
4. Regular Expressions and their implementation/Significance
5. Describe implementation of compiler taking any well-known compiler as case study (for instance Python, Java etc)
6. Natural Language Processing
7. Human language based compiler : Problem and issues
8. Text mining :Role of python and compiler theory
9. Theory of Computation : History, theorems, applications in Compiler Construction
10. Compiler and System s/w: relationship, differences, working

What kind of problem you can choose

1. **Translation one language to another (eg: C++ to Java, Natural language command to any programming language command, kindly print var-→print(var)**
2. **Designing some new(simplified) language**
3. **Describing some natural language translation automatically**
4. **etc**

CSL 323 Compiler Construction
Semester 05 (Fall 2021)
Lecturer(s): Miss Asia Samreen
Lab Engineer: Saba Imtiaz
Abdullah Abdul Wahid
02-134192-015

1. Problem Specification:

Our main motivation behind this project is to try to make programming languages easy, usable & understandable for every person. Most people don't learn coding because of its technicalities & think that it's difficult to understand it.
This project's intended audience consists of individuals who are new to the world of programming and so require a basic grasp of how a compiler may evaluate and process code.

2. Specification of Rules and Attributes:

LANGUAGE PARADIGM: Structured

CASE SENSITIVE: Yes, the language is Case Sensitive.

KEYWORDS:

Keyword	Keyword
def = define	while = run_till
break = halt	class=def_cl
return = back	if
for	else = el/elif
char = chr	import = impt
string = strn	none = null
bool = check	do while = do until
range = rng	print

DATA TYPES: int, strn, float, check, chr, list, rng etc.

Compiler Construction (CSC-323)

Course Instructor: Asia Samreen

3. Specification of Programming Language Constructs:

ITERATIVE STATEMENTS:

```
for i in range(...):
        statement

run_till i>5:
        statement
do:
    statement
until i>5
```

CONDITIONAL STATEMENTS:

```
if i>5:
   statement
el:
   statement
```

COMMENTS (Multi line + Single line):

1. ## (Single-line)
2. #* *# (Multi-line)

LINE TERMINATOR: **";"**

Compiler Construction (CSC-323)

Course Instructor: Asia Samreen

OPERATORS:

- ### Arithmetic Operators:
  1. +
  2. -
  3. *
  4. /
  5. %
- ### Assignment Operators:
  1. +=
  2. -=
  3. *=
  4. /=
  5. =
- ### Relational Operators:
  1. >
  2. <
  3. >=
  4. <=
  5. ==
  6. !=
- ### Logical Operators:
  1. &&
  2. ||

Compiler Construction (CSC-323)

Course Instructor: Asia Samreen

PUNCTUATORS:

1. [ ]
2. ( )
3. :
4. +
5. { }
6. ‘ ‘
7. “ “
8. /

IDENTIFIERS:

1. _a to z, a to z
2. a to z
3. A to Z
4. a to z 0 to 9
5. a to z_ 0 to 9

Compiler Construction (CSC-323)

Course Instructor: Asia Samreen

SYNTAX SPECIFICATION: There are 2 basic syntax specification in this language:

1. **Method Declaration and Calling:**

```
define course (inp): print("hello")   ##defining a function
course (a)   ##calling course function with single paramater
course(a,b) ##calling course function with double parameter
```

2. **List Declaration:**

```
##identifiers, array declaration:
##for arrays we use list:
    cars = ["Chevrolet", "BMW", "Mercedes-AMG"]
    days = ["Monday","Tuesday","Wednesday",.....]

  ##for declaring an int or any data type:
    i = 0 //declaring int
    x = "random string”
```

Compiler Construction (CSC-323)

Course Instructor: Asia Samreen

4. Possible Techniques to implement Lexical Analyzer:

The primary purpose of lexical analysis is to interpret input characters in code and generate tokens. The lexical analyzer searches the whole program's source code. It recognises each token one by one. Scanners are often designed to create tokens only when a parser requests them. When producing these tokens, Lexical Analyzer skips over **whitespaces** and **comments**. If an error is found, the Lexical analyzer will associate it with the source file and line number.

The lexical analyzer accomplishes the following tasks:

- Aids in the identification of tokens in the symbol table.
- Removes whitespace and comments from the original code.
- Read the source program's input characters.

Lexical Analysis breaks the code into tokens, as per the following implementation:

Identifier	_A-Za-z0-9
Keywords	for, if, el, elif, run_till, define, halt, back, def, def_cl, impt, null, do until, print, rng, null
Relational operators	<,>,<=,>=,<>
Mathematical operator	+, -,/,*,%
Logical operator	AND OR NOT
Punctuation	() [] {} , . ; : “”
Increment decrement	++ , --
literals	Strings, Numerical, Boolean with “ “ around them
Data types	int, strn, float, chr, list, rng, check

5. Lexical Analyzer Implementation:

```
from distutils.command.clean import clean
import re

def removecomments(code):
    single = re.sub(r"(#|(//))(.+?)[\n]","",code)
    multireg = re.compile(r"((\"\"\")(.+?)(\"\"\"))|((/\*)(.+?)(\*/))", re.DOTALL)
    multi = re.sub(multireg,"", single)
    return multi

def writecleancode(cleancode):
    file3 = open("cleancode.txt", "w")
    file3.write(cleancode)
    file3.close()

def readinput():
    file1 = open("ini_code.txt","r")
    fileVal = file1.read()
    file1.close()
    return fileVal


def lex_Analyzer(store):

    file = open("lex.txt", "w")

    keyword = r"main|if|el|elif|back|rng|elif|print|in|func|define|out|halt|chr|run_
until|def_class|impt|null|do_until|print"
    relationOp = r"(\=\=)|(\<\=)|(\>\=)|[<>]"
    identifier =r"[A-Za-z_][\w]*"
    integ = r"[0-9]"
    increment = r"[++]"
    decrement = r"[--]"
    punc = r"[\[\]\(\)\:\,\{\}]"
    terminate = r"[\;]"
    mathOp = r"[\+\-\*\/\%]"
    logicOp =  r"(\&\&)|(\|\|)|(\>\=)|[<>]|and|or|not"
    litr = r"(\".+?\")|(\'.+?\')"
    whitesp =  r"\s"
    dType = r"int|strn|float|check|chr|list|rng|check"
    storeVal = store.split()
```

```
    for word in storeVal:
        if word in logicOp:
            file.write("(logical-Operator(s): "+ word +" )"+"\n")

        elif word in punc:
            file.write("(Punctuation: "+ word +" )"+ "\n")

        elif word in terminate:
            file.write("(Terminator: "+ word +" )"+ "\n")

        elif word in keyword:
            file.write("(Keyword: "+ word +" )"+"\n")

        elif word in dType:
            file.write("(Data-type: "+ word +" )"+"\n")

        elif word in relationOp:
            file.write("(Relational-Operator(s): "+ word +" )"+"\n")

        elif re.match(integ,word):
            file.write("(Integer: "+ word +" )"+"\n")

        elif word in mathOp:
            file.write("(Mathematical-Operator(s): "+ word +" )"+ "\n")

        elif re.match(increment,word):
            file.write("(Increment: "+ word +" )"+"\n")

        elif re.match(decrement,word):
            file.write("(Decrement: "+ word +" )"+"\n")

        elif re.match(identifier, word):
            file.write("(Identifier(s): "+ word +" )"+"\n")

        elif re.match(litr,word):
            file.write("(Literal(s): "+ word +" )"+"\n")

    file = open("lex.txt",'r')
    print(file.read())
    return storeVal


ini_code = readinput()
clean_code = removecomments(ini_code)
writecleancode(clean_code)
tokens = lex_Analyzer(clean_code)
```

6. Available Techniques/Algorithms to implement Parser:

The parser is the step of the compiler that accepts a token string as input and turns it into the matching Intermediate Representation using existing grammar. Syntax Analyzer is another name for the parser.

The Parser is divided into two types:

- **Top-down Parsers**
- **Bottom-up Parsers.**

Top-Down Parser: A *top-down parser* is a parser that creates a parse for a given input string using grammatical productions by expanding non-terminals, starting from the start symbol and ending on the terminals. It employs the leftmost derivation.

Top-down parsers are further categorized into two types:

- **Recursive Descent Parser.**
- **Non-recursive Descent Parser.**

Recursive Descent Parser: *Recursive Descent Parser* is also known as the Brute force or backtracking parser. It creates the parse tree mostly by brute force and backtracking.
Non-recursive Descent Parser: *Non-Recursive Descent* Parser is also known as an LL (1) parser, a predictive parser, a parser without backtracking, or a Dynamic Parser. Instead of backtracking, it generates the parse tree using a parsing table.

Bottom-up Parser: *Bottom-up Parser* is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.
Further Bottom-up parser is classified into two types:

- **LR Parser (Shift-Reduce Parser)**
- **Operator Precedence Parser.**

- **LR parser:** *LR parser* is the bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follows the reverse of the rightmost derivation.
- **Operator precedence parser:** *Operator precedence parser* generates the parse tree form given grammar and string but the only condition is two consecutive non-terminals and epsilon never appear on the right-hand side of any production.

7. Implementation of Parser (Table):

Shift-Reduce Parser:

For The CFG:

E --> T * F
T --> F + n | F
F --> F * n
F --> n

```
from asyncio.windows_events import NULL
import re

print("BOTTOM UP (SHIFT REDUCE) PARSER\n")
print("E -> E + T\n")
print("T' -> T + F\n")
print("F -> n\n")

inp = input("Enter Your Input to Test Against the CFG: ")
splt = inp.split()
print(splt)

i=0
strng=[None]*len(inp)

def parseReduce():
    global i
    if strng[i]=='n':
        strng[i]='E'
        print('Reducing E--> n', strng[i])
        i=i+1
    elif (strng[i]=='E') and (strng[i-2]=='E'):
        strng[i]=NULL
        strng[i-1]=NULL
        print('Reducing E--> E+E')
        i=i-2
    else:
        i=i+1

while(splt[i]!='$'):
    strng[i]=splt[i]
    print('SHIFT-->',strng[i])
    parseReduce()
print(strng)
```

Compiler Construction (CSC-323)

Course Instructor: Asia Samreen

PARSE TABLE:

LR(1) closure table			
Goto	Kernel	State	Closure
	{[E -> .T * F, $]}	0	{[E -> .T * F, $]; [T -> .F + n | F, *]; [F -> .F * n, +/*]; [F -> .n, +/*]}
goto(0, T)	{[E -> T.* F, $]}	1	{[E -> T.* F, $]}
goto(0, F)	{[T -> F.+ n | F, *]; [F -> F.* n, +/*]}	2	{[T -> F.+ n | F, *]; [F -> F.* n, +/*]}
goto(0, n)	{[F -> n., +/*]}	3	{[F -> n., +/*]}
goto(1, *)	{[E -> T *.F, $]}	4	{[E -> T *.F, $]; [F -> .F * n, $/*]; [F -> .n, $/*]}
goto(2, +)	{[T -> F +.n | F, *]}	5	{[T -> F +.n | F, *]}
goto(2, *)	{[F -> F *.n, +/*]}	6	{[F -> F *.n, +/*]}
goto(4, F)	{[E -> T * F., $]; [F -> F.* n, $/*]}	7	{[E -> T * F., $]; [F -> F.* n, $/*]}
goto(4, n)	{[F -> n., $/*]}	8	{[F -> n., $/*]}
goto(5, n)	{[T -> F + n.| F, *]}	9	{[T -> F + n.| F, *]}
goto(6, n)	{[F -> F * n., +/*]}	10	{[F -> F * n., +/*]}
goto(7, *)	{[F -> F *.n, $/*]}	11	{[F -> F *.n, $/*]}
goto(9, |)	{[T -> F + n |.F, *]}	12	{[T -> F + n |.F, *]; [F -> .F * n, *]; [F -> .n, *]}
goto(11, n)	{[F -> F * n., $/*]}	13	{[F -> F * n., $/*]}
goto(12, F)	{[T -> F + n | F., *]; [F -> F.* n, *]}	14	{[T -> F + n | F., *]; [F -> F.* n, *]}
goto(12, n)	{[F -> n., *]}	15	{[F -> n., *]}
goto(14, *)	{[F -> F *.n, *]}	16	{[F -> F *.n, *]}
goto(16, n)	{[F -> F * n., *]}	17	{[F -> F * n., *]}

PARSING FOR INPUT: “n * n + n”

Input (tokens): n * n + n

Maximum number of steps: 100

PARSE

Trace				Tree
Step	Stack	Input	Action	
1	0	n * n + n $	s3	
2	0 n 3	* n + n $	r_3	
3	0 F	* n + n $	2	
4	0 F 2	* n + n $	s6	
5	0 F 2 * 6	n + n $	s10	
6	0 F 2 * 6 n 10	+ n $	r_2	
7	0 F	+ n $	2	
8	0 F 2	+ n $	s5	
9	0 F 2 + 5	n $	s9	
10	0 F 2 + 5 n 9	$		