Compiler Design LAB

Project Report

SUBMITTED BY

Abdul Wahid
ID: **193-15-2992**

Ata-E-Elahi
ID: **193-15-2946**

MD RAKIBUL ISLAM
ID: **193-15-3015**

AND
Samanta Sajjad Shraboni
ID: **193-15-3023**

Submitted To

Md. Tazmim Hossain
Research Associate,
Department of CSE
Daffodil International University

DAFFODIL INTERNATIONAL UNIVERSITY

DHAKA, BANGLADESH

INTRODUCTION

A compiler is a special program that translates a programming language's source code into machine code, bytecode or another programming language. The source code is typically written in a high-level, human-readable language such as Java or C++. In our project we are developed a compiler for C. Programmer writes the source code in a code editor or an integrated development environment (IDE) that includes an editor, saving the source code to one or more text files. In our project our IDE is code.txt file. Project.py file is the compiler of that project.

Compilers vary in the methods they use for analyzing and converting source code to output code. Despite their differences, they typically carry out the following some steps. In this project we first tokenized our HHL code. Then from the tokenized code we eliminate the left factoring. After that in 3 no program we find the First and Follow of that code. In 4 no a parsing table is generated. After that 5 no we generate canonical table to understand the follow of code. In 6, from a equation we generated a intermediate code (Three address code), for convent the program into assembly code.

OBJECTIVES

The main objectives of this project are:

- Lexical Analysis
- Elimination of Left Factoring
- Finding First and Follow
- Construction of Predictive Parsing Table
- Computation of LR (0) items
- Intermediate Code Generation: Three Address Code

MOTIVATION

A compiler is a computer program that decodes computer code composed in one programming language into another language. Or we can say that the compiler helps in translating the source code composed in a high-level programming language into the machine code. 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.

METHODOLOGY

This project was developed using Python Programming Language. At the beginning of the program, we do lexical analysis of the program by using tokenization. The lexical analyzer is responsible for reading the code line by line. Whenever it reads a word, it checks if the word is a keyword or not using a symbol table and generates a token for that particular word. The lexical analysis removes blank spaces, argument lines, the next line character, and braces: <Token_name,number>. For the Elimination of Left Factoring, left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers. In left factoring, we make one production for each common prefixes. The common prefix may be a terminal or a non-terminal or a combination of both. Rest of the derivation is added by new productions. For First and Follow, first is a function that gives the set of terminals that begin the strings derived from the production rule. A symbol c is in FIRST (α) if and only if $\alpha \Rightarrow c\beta$ for some sequence β of grammar symbols. A terminal symbol a is in FOLLOW (N) if and only if there is a derivation from the start symbol S of the grammar such that S

⇒ αNαβ, where α and β are a (possible empty) sequence of grammar symbols. In other words, a terminal c is in FOLLOW (N) if c can follow N at some point in a derivation.

Predictive parsing can also be accomplished using a predictive parsing table and a stack. It is sometimes called non-recursive predictive parsing. The idea is that we construct a table M [X, token] which indicates which production to use if the top of the stack is a nonterminal X and the current token is equal to token; in that case we pop X from the stack and we push all the rhs symbols of the production M[X, token] in reverse order. We use a special symbol $ to denote the end of file. After that in LR (0), item is a production G with dot at some position on the right side of the production. LR(0) items is useful to indicate that how much of the input has been scanned up to a given point in the process of parsing. In the LR (0), we place the reduce node in the entire row. For constructing a canonical table at first, we need to check if a state is going to some other state on a terminal, then it corresponds to a shift move. If a state is going to some other state on a variable, then it corresponds to go to move. If a state contain the final item in the particular row then write the reduce node completely. After this phase Three Address Code. It is a form of an intermediate code. They are generated by the compiler for implementing Code Optimization. They use maximum three addresses to represent any statement. They are implemented as a record with the address fields.

IMPLEMENTATION

For lexical analysis in code.txt is used for it. From this file it converted into string. Then the sting is tokenized. Keywords and Operators are kept in separate list. If a token is not keywords or operator it is a identifier.

```
def lexical_Analyzer():

    print("\n")

    print("\n")

    print("\n")

    keywords = {"auto","break","case","char","const","continue","default","do",

    "double","else","enum","extern","float","for","goto",

    "if","int","long","register","return","short","signed",

    "sizeof","static","struct","switch","typedef","union",


"unsigned","void","volatile","while","printf","scanf","%d","include","stdio.h","main"}

    operators = {"+","-","*","/","<",">","=","<=",">=","==","!=","++","--","%"}

    delimiters = {'(',')','{','}','[',']','"',"'",';','#',',',''}

    def detect_keywords(text):

        arr = []

        for word in text:

            if word in keywords:

                arr.append(word)

        return list(set(arr))

    def detect_operators(text):

        arr = []

        for word in text:

            if word in operators:

                arr.append(word)

        return list(set(arr))
```

```
def detect_delimiters(text):
    arr = []
    for word in text:
        if word in delimiters:
            arr.append(word)
    return list(set(arr))
def detect_num(text):
    arr = []
    for word in text:
        try:
            a = int(word)
            arr.append(word)
        except:
            pass
    return list(set(arr))
def detect_identifiers(text):
    k = detect_keywords(text)
    o = detect_operators(text)
    d = detect_delimiters(text)
    n = detect_num(text)
    not_ident = k + o + d + n
    arr = []
    for word in text:
        if word not in not_ident:
            arr.append(word)
    return arr
with open('code.txt') as t:
    text = t.read().split()
```

```
    print("Keywords: ",detect_keywords(text))
    print("Operators: ",detect_operators(text))
    print("Delimiters: ",detect_delimiters(text))
    print("Identifiers: ",detect_identifiers(text))
    print("Numbers: ",detect_num(text))
```

For the Elimination of Left Factoring, left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers. In left factoring, we make one production for each common prefixes. The common prefix may be a terminal or a non-terminal or a combination of both. Rest of the derivation is added by new productions.

```
def eli_of_lf():
    from itertools import takewhile
    print("\n")
    print("\n")
    print("\n")
    s= "S->iEtS|iEtSeS|a"
    s=input("Enter production rule: ")
    def groupby(ls):
        d = {}
        ls = [ y[0] for y in rules ]
        initial = list(set(ls))
        for y in initial:
            for i in rules:
                if i.startswith(y):
                    if y not in d:
                        d[y] = []
                    d[y].append(i)
        return d
```

```
def prefix(x):
    return len(set(x)) == 1
starting=""
rules=[]
common=[]

while(True):
    rules=[]
    common=[]
    split=s.split("->")
    starting=split[0]
    for i in split[1].split("|"):
        rules.append(i)

#logic for taking commons out
    for k, l in groupby(rules).items():
        r = [l[0] for l in takewhile(prefix, zip(*l))]
        common.append(''.join(r))
#end of taking commons
    for i in common:
        newalphabet=alphabetset.pop()
        print(starting+"->"+i+newalphabet)
        index=[]
        for k in rules:
            if(k.startswith(i)):
                index.append(k)
        print(newalphabet+"->",end="")
        for j in index[:-1]:
```

```
                stringtoprint=j.replace(i,"", 1)+"|"
                if stringtoprint=="|":
                    print("\u03B5","|",end="")
                else:
                    print(j.replace(i,"", 1)+"|",end="")
            stringtoprint=index[-1].replace(i,"", 1)+"|"
            if stringtoprint=="|":
                print("\u03B5","",end="")
            else:
                print(index[-1].replace(i,"", 1)+"",end="")
            print("")
        break
```

For First and Follow, first is a function that gives the set of terminals that begin the strings derived from the production rule. A symbol c is in FIRST (α) if and only if α ⇒ cβ for some sequence β of grammar symbols. A terminal symbol a is in FOLLOW (N) if and only if there is a derivation from the start symbol S of the grammar such that S ⇒ αNαβ, where α and β are a (possible empty) sequence of grammar symbols. In other words, a terminal c is in FOLLOW (N) if c can follow N at some point in a derivation.

```
def first_follow():
    print("\n")
    print("\n")
    print("\n")
    # #example for direct left recursion
    # gram = {"A":["Aa","Ab","c","d"]
    # }

    gram = {
        "E":["E+T","T"],
```

```
    "T":["T*F","F"],
    "F":["(E)","i"]
}
def rem(gram):
    c = 1
    conv = {}
    gramA = {}
    revconv = {}
    for j in gram:
        conv[j] = "A"+str(c)
        gramA["A"+str(c)] = []
        c+=1
    for i in gram:
        for j in gram[i]:
            temp = []
            for k in j:
                if k in conv:
                    temp.append(conv[k])
                else:
                    temp.append(k)
            gramA[conv[i]].append(temp)
    #print(gramA)
def follow(gram, term):
    a = []
    for rule in gram:
        for i in gram[rule]:
            if term in i:
                temp = i
                indx = i.index(term)
```

```
                    if indx+1!=len(i):
                        if i[-1] in firsts:
                            a+=firsts[i[-1]]
                        else:
                            a+=[i[-1]]
                    else:
                        a+=["e"]
                    if rule != term and "e" in a:
                        a+= follow(gram,rule)
        return a
    follows = {}
    for i in result:
        follows[i] = list(set(follow(result,i)))
        if "e" in follows[i]:
            follows[i].pop(follows[i].index("e"))
        follows[i]+=["$"]
        print(f'Follow({i}):',follows[i])
```

Predictive parsing can also be accomplished using a predictive parsing table and a stack. It is sometimes called non-recursive predictive parsing. The idea is that we construct a table M [X, token] which indicates which production to use if the top of the stack is a nonterminal X and the current token is equal to token; in that case we pop X from the stack and we push all the rhs symbols of the production M[X, token] in reverse order. We use a special symbol $ to denote the end of file.

```
def lr_0():
    gram = {
                                                    "S":["CC"],
                                                    "C":["aC","d"]
    }
```

```
    start = "S"

    terms = ["a","d","$"]


    non_terms = []

    for i in gram:

        non_terms.append(i)

    gram["S'"]= [start]

    new_row = {}

    for i in terms+non_terms:

        new_row[i]=""


    non_terms += ["S'"]

    # each row in state table will be dictionary {nonterms ,term,$}

    stateTable = []

    # I = [(terminal, closure)]

    # I = [("S","A.A")]


    def Closure(term, I):

        if term in non_terms:

            for i in gram[term]:

                I+=[(term,"."+i)]

        I = list(set(I))

        for i in I:

            # print("." != i[1][-1],i[1][i[1].index(".")+1])

            if "." != i[1][-1] and i[1][i[1].index(".")+1] in non_terms and
i[1][i[1].index(".")+1] != term:

                I += Closure(i[1][i[1].index(".")+1], [])

        return I

    Is = []
```

```
Is+=set(Closure("S'", []))
countI = 0
omegaList = [set(Is)]
while countI<len(omegaList):
    newrow = dict(new_row)
    vars_in_I = []
    Is = omegaList[countI]
    countI+=1
    for i in Is:
        if i[1][-1]!=".":
            indx = i[1].index(".")
            vars_in_I+=[i[1][indx+1]]
    vars_in_I = list(set(vars_in_I))
    # print(vars_in_I)
    for i in vars_in_I:
        In = []
        for j in Is:
            if "."+i in j[1]:
                rep = j[1].replace("."+i,i+".")
                In+=[(j[0],rep)]
        if (In[0][1][-1]!="."):
            temp = set(Closure(i,In))
            if temp not in omegaList:
                omegaList.append(temp)
            if i in non_terms:
                newrow[i] = str(omegaList.index(temp))
            else:
                newrow[i] = "s"+str(omegaList.index(temp))
            print(f'Goto(I{countI-1},{i}):{temp} That is I{omegaList.index(temp)}')
```

```
            else:
                temp = set(In)
                if temp not in omegaList:
                    omegaList.append(temp)
                if i in non_terms:
                    newrow[i] = str(omegaList.index(temp))
                else:
                    newrow[i] = "s"+str(omegaList.index(temp))
                print(f'Goto(I{countI-1},{i}):{temp} That is I{omegaList.index(temp)}')
        stateTable.append(newrow)
    # print("\n\nList of I's\n")
    # for i in omegaList:
    #     print(f'I{omegaList.index(i)}: {i}'
    for i in omegaList:
        for j in i:
            if "." in j[1][-1]:
                if j[1][-2]=="S":
                    stateTable[omegaList.index(i)]["$"] = "Accept"
                    break
                for k in terms:
                    stateTable[omegaList.index(i)][k]                                    =
"r"+str(I0.index(j[1].replace(".","")))
    print("\nCononical table")

    print(f'{" ": <9}',end="")
    for i in new_row:
        print(f'|{i: <11}',end="")

    print(f'\n{"-":-<66}')
```

```
    for i in stateTable:
        print(f'{"I("+str(stateTable.index(i))+")": <9}',end="")
        for j in i:
            print(f'|{i[j]: <10}',end=" ")
        print()
```

Three Address Code. It is a form of an intermediate code. They are generated by the compiler for implementing Code Optimization. They use maximum three addresses to represent any statement. They are implemented as a record with the address fields.

```
def code_gen():
    OPERATORS = set(['+', '-', '*', '/', '(', ')'])
    PRI = {'+':1, '-':1, '*':2, '/':2}

    def generate3AC(pos):
        print("### THREE ADDRESS CODE GENERATION ###")
        exp_stack = []
        t = 1
        for i in pos:
            if i not in OPERATORS:
                exp_stack.append(i)
            else:
                print(f't{t} := {exp_stack[-2]} {i} {exp_stack[-1]}')
                exp_stack=exp_stack[:-2]
                exp_stack.append(f't{t}')
                t+=1
    expres = input("INPUT THE EXPRESSION: ")
    pre = infix_to_prefix(expres)
    pos = infix_to_postfix(expres)
    generate3AC(pos)
```

SOURCE CODE

GitHub repository: https://github.com/abdul-wahid789/Compiler-Design

GitHub Repository QR code.

PROJECT OUTPUT

```
1. Lexical Analyzer
2. Elimination of Left Factoring
3. Computation of First and Follow sets
4. Construction of Predictive Parsing Table
5. Computation of LR(0) items
6. Intermediate Code Generation: Three Address Code
Enter your choice: 1



Keywords:  ['return', '%d', 'int']
Operators:  ['=']
Delimiters:  ['}', '{']
Identifiers:  ['#include', '<stdio.h>', '//', 'This', 'is', 'a', 'header', 'file', 'main()', 'a;', 'a', '10;', 'printf("The',
 'value', 'of', 'a', 'is', '",a);', '0;']
Numbers:  []
```

```
1. Lexical Analyzer
2. Elimination of Left Factoring
3. Computation of First and Follow sets
4. Construction of Predictive Parsing Table
5. Computation of LR(0) items
6. Intermediate Code Generation: Three Address Code
Enter your choice: 2



Enter production rule: S->iEtS|iEtSeS|a
S->aZ'
Z'->ε
S->iEtSY'
Y'->ε |eS
```

```
1. Lexical Analyzer
2. Elimination of Left Factoring
3. Computation of First and Follow sets
4. Construction of Predictive Parsing Table
5. Computation of LR(0) items
6. Intermediate Code Generation: Three Address Code
Enter your choice: 3




First(E): ['(', 'i']
First(T): ['(', 'i']
First(F): ['(', 'i']
First(E'): ['+', 'e']
First(T'): ['*', 'e']
Follow(E): [')', '$']
Follow(T): [')', '+', '$']
Follow(F): [')', '*', '+', '$']
Follow(E'): [')', '$']
Follow(T'): [')', '+', '$']
```

```
1. Lexical Analyzer
2. Elimination of Left Factoring
3. Computation of First and Follow sets
4. Construction of Predictive Parsing Table
5. Computation of LR(0) items
6. Intermediate Code Generation: Three Address Code
Enter your choice: 4




          |)         |i         |+         |(         |*         |$
-------------------------------------------------------------------------
E         |          |E->TE'    |          |E->TE'    |          |
-------------------------------------------------------------------------
T         |          |T->FT'    |          |T->FT'    |          |
-------------------------------------------------------------------------
F         |          |F->i      |          |F->(E)    |          |
-------------------------------------------------------------------------
E'        |E'->e     |          |E'->+TE'  |          |          |E'->e
-------------------------------------------------------------------------
T'        |T'->e     |          |T'->e     |          |T'->*FT'  |T'->e
```

```
1. Lexical Analyzer
2. Elimination of Left Factoring
3. Computation of First and Follow sets
4. Construction of Predictive Parsing Table
5. Computation of LR(0) items
6. Intermediate Code Generation: Three Address Code
Enter your choice: 5


Goto(I0,d):{('C', 'd.')} That is I1
Goto(I0,S):{("S'", 'S.')} That is I2
Goto(I0,a):{('C', '.d'), ('C', 'a.C'), ('C', '.aC')} That is I3
Goto(I0,C):{('C', '.d'), ('C', '.aC'), ('S', 'C.C')} That is I4
Goto(I3,d):{('C', 'd.')} That is I1
Goto(I3,C):{('C', 'aC.')} That is I5
Goto(I3,a):{('C', '.d'), ('C', 'a.C'), ('C', '.aC')} That is I3
Goto(I4,d):{('C', 'd.')} That is I1
Goto(I4,C):{('S', 'CC.')} That is I6
Goto(I4,a):{('C', '.d'), ('C', 'a.C'), ('C', '.aC')} That is I3
['d', 'S', 'aC', 'CC']

Canonical table
        |a          |d          |$          |S          |C
-------------------------------------------------------------------
I(0)    |s3         |s1         |           |2          |4
I(1)    |r0         |r0         |r0         |           |
I(2)    |           |           |Accept     |           |
I(3)    |s3         |s1         |           |           |5
I(4)    |s3         |s1         |           |           |6
I(5)    |r2         |r2         |r2         |           |
I(6)    |r3         |r3         |r3         |           |
```

```
1. Lexical Analyzer
2. Elimination of Left Factoring
3. Computation of First and Follow sets
4. Construction of Predictive Parsing Table
5. Computation of LR(0) items
6. Intermediate Code Generation: Three Address Code
Enter your choice: 6




INPUT THE EXPRESSION: a=b+1 b=c+d
PREFIX: ++=cd
POSTFIX: a=b1 b=c+d+
### THREE ADDRESS CODE GENERATION ###
t1 := = + c
t2 := t1 + d
```

CONCLUSION

Compiler Design is the structure and set of defined principles that guide the translation, analysis, and optimization of the entire compiling process. The compiler process runs through syntax, lexical, and semantic analysis in the front end. It generates optimized code in the back end. Intermediate code generation is independent of machine and the process of conversion of Intermediate code to target code is independent of language used. We have done the front end of compilation process. It includes 3 phases of compilation lexical analysis, syntax analysis and then followed by intermediate code generation.