Kathmandu University

Department of Computer Science and Engineering

Dhulikhel, Kavre

A Mini-Project Report

[Code No: COMP 409]

Submitted by

Saugat Adhikari (03)

Aayush Dip Giri (16)

Sanjeev Kumar Khatri (23)

Utsav Maskey (28)

Saharsha Ojha (30)

Ayush Uprety (57)

Yogesh Pandey (79)

Submitted to:

Department of Computer Science and Engineering

Submission Date: 19th May, 2022

Table of Contents

Chapter 1: Introduction

1.1. Context Free Grammar

To define context-free languages, context-free grammars (CFGs) are utilized. A context-free grammar is a collection of recursive rules for generating string patterns. A context-free grammar can explain all regular languages and more, but not all possible languages.

Theoretical computer science, compiler design, and linguistics are all interested in context-free grammars. CFGs are used to define computer languages, and context-free grammars can be used to construct parser programs in compilers.

A context-free grammar can be described by a four-element tuple (V,Σ,R,S), where

- V is a finite set of variables (which are non-terminal);
- Σ is a finite set (disjoint from V) of terminal symbols;
- R is a set of production rules where each production rule maps a variable to a string $s \in (V \cup \Sigma)*$;
- S (which is in V) which is a start symbol.

1.2. Why Python?

1. One of the most appealing aspects of this language is that it employs attractive syntax, making applications easy to comprehend.
2. Python is a programmable language that may be embedded into applications to provide a programmable interface.
3. It's an easy-to-use language that makes getting the software to function a breeze.

4. It is also straightforward to extend the code in Python by attaching additional modules written in other compiled languages such as C++ or C.

1.3. Rightmost Derivation

The process of deriving a string by expanding the rightmost non-terminal at each step is called the rightmost derivation.

Example:

Consider the following grammar-

S → aB / bA

S → aS / bAA / a

B → bS / aBB / b

Let us consider a string w = aaabbabbba

Rightmost Derivation-

S → aB

→ aaB**B** (Using B → aBB)

→ aaBaB**B** (Using B → aBB)

→ aaBaBb**S** (Using B → bS)

→ aaBaBbb**A** (Using S → bA)

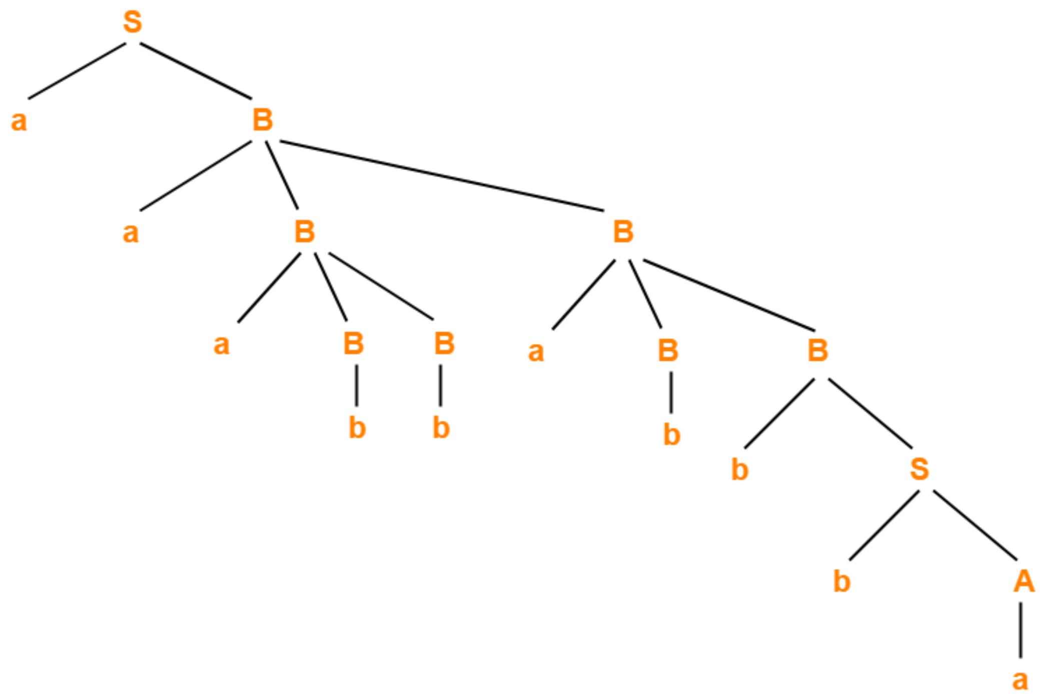
→ aaBa**B**bba (Using A → a)

→ aa**B**abbba (Using B → b)

→ aaaB**B**abbba (Using B → aBB)

→ aaa**B**babbba (Using B → b)

→ aaabbabbba (Using B → b)



1.4. SLR Parser:

SLR stands for simple linear regression. It is the simplest type of grammar, with only a few states. SLR is very easy to construct and is similar to LR parsing. The only difference between SLR parser and LR(0) parser is that in the LR(0) parsing table, there's a chance of 'shift reduce' conflict because we are entering 'reduce' corresponding to all terminal states. We can solve this problem by entering 'reduce' corresponding to the FOLLOW of LHS of production in the terminating state. This is called SLR(1) collection of items.

Chapter 2: Implementation

2.1. Reverse of the rightmost derivation of the string

The bottom-up parsers are created for the largest class of LR grammars. As the bottom-up parsing corresponds to the process of reducing the string to the starting symbol of the grammar.
The reduction process is just the reverse of derivation in top-down parsing. Thus, the bottom-up parsing derives the input string reverse.

2.2. LR(0) Items

A production G with a dot at some point on the right side of the production is an LR (0) item.The LR(0) items reflect how much of the input has been scanned up to a certain point in the parsing process.

E -> E + T | T T -> T F | F F -> F * | a | b **Augmented Grammar** E' -> E E -> E + T E -> T T -> T F T -> F F -> F * F -> a F -> b	**Items** **I0**: E' -> . E E -> . E + T E -> . T T -> . T F T -> . F F -> . F * F -> . a F -> . b **I1**: E' -> E .	**I4**: F -> a . I5: F -> b . **I5**: E -> E + . T **I6**: E -> E + . T T -> . T F T -> . F F -> . F *

Terminals : ['+', '*', 'a', 'b'] **Nonterminals**: ["E'", 'E', 'T', 'F'] **Symbols** : ["E'", 'E', 'T', 'F', '+', '*', 'a', 'b'] **First** E' = { a , b } E = { a , b } T = { a , b } F = { a , b } **Follow** E' = { $ } E = { $, + } T = { $, + , a , b } F = { $, + , a , b , * }	E -> E . + T **I2**: E -> T . T -> T . F F -> . F * F -> . a F -> . b **I3**: T -> F . F -> F . * **I4**: F -> a . **I5**: F -> b .	F -> . a F -> . b T -> . T F T -> . F F -> . F * F -> . a F -> . b **I7**: T -> T F . F -> F . * **I8**: F -> F * . **I9**: E -> E + T . T -> T . F F -> . F * F -> . a F -> . b

2.3. SLR Parsing Table

The action and goto fields in the parsing table are associated with each state in the DFA. The following algorithms are used to calculate them:

Construct C ={I0,I1,…..In}, the collection of sets of LR(0) items for G' State i is constructed from Ii. The parsing actions for state i are determined as follows:

- If [A→α.aβ] is in Ii and goto(Ii, a) = Ij, then set action[i, a] to “shift j.” Here, a is required to be a terminal
- If [A→α.] is in Ii then set action [i, a][to “reduce A→ α] for all a in FOLLOW(A), here A may not be S’
- Set action[i, $] to "accept" if [S' S.] is in Ii.
- If the aforementioned rules produce any contradictory actions, we declare the grammar is not SLR (1). The algorithm fails to produce a parser in this case.
- The goto transitions for state I are constructed for all non-terminals ‘A' using the rule: if goto(Ii, A) = Ij, then goto[i, A] =j
- All entries not defined by rules (2) and (3) are made “error”
- The initial state of the parser is the one constructed from the set containing item [S’ → S]

Number of productions in the grammar from onwards and use the production number while making a reduction entry.

For instance, in the given grammar,

1. E E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

This construction requires FOLLOW of each non-terminal present in the grammar to be computed

The grammar that has a SLR parsing table is known as SLR(1) grammar. Generally, 1 is omitted.

Consider a grammar,

E -> E + T | T

T -> T F | F

F -> F * | a | b

The SLR parsing table for the above grammar is:

State	+	*	a	b	$	E	T	F
0			s4	s5		1	2	3
1	s6				acc			
2	r2		s4	s5	r2			7
3	r4	s8	r4	r4	r4			
4	r6	r6	r6	r6	r6			
5	r7	r7	r7	r7	r7			
6			s4	s5			9	3
7	r3	s8	r3	r3	r3			
8	r5	r5	r5	r5	r5			
9	r1		s4	s5	r1			7

2.4. Simulation of SLR Parsing Table

Here, we show step by step the procedure of how the parsing table determines if the given string is accepted by the provided grammar or not. We consider the following string:

a + b * a

We take an input stack containing our input string and another stack for our implementation. The input string is appended with a dollar sign ($) which denotes the end of the stack.

The simulation for the above string is given below:

Step	Stack	Input	Action
1	0	a+b*a$	s4
2	0a4	+b*a$	r6
3	0F3	+b*a$	r4

4	0T2	+b*a$	r2
5	0E1	+b*a$	s6
6	0E1+6	b*a$	s5
7	0E1+6b5	*a$	r7
8	0E1+6F3	*a$	s8
9	0E1+6F3*8	a$	r5
10	0E1+6F3	a$	r4
11	0E1+6T9	a$	s4
12	0E1+6T9a4	$	r6
13	0E1+6T9F7	$	r3
14	0E1+6T9	$	r1
15	0E1	$	Accepted

Chapter 3: Program and Outputs

3.1. Program

3.1.1. grammar.py

```
E -> E + T | T
T -> T F | F
F -> F * | a | b
```

3.1.2. Global_vars.py

```
grammar = {}
lr0_items = {}
terms = []
nonterms = []
symbols = []
```

3.1.3. Main.py

```
import os

from global_vars import grammar, lr0_items, nonterms, symbols, terms
from parse_input_string import parse_input_string
from slr_helpers import get_first, get_follow
from slr_parser import collect_lr0_items, display, perform_action

start_sym = ''
error_sym = 0
```

```
def main():
  with open(GRAMMAR_FILE_PATH, encoding='utf-8') as grammar_file:
    parse_grammar(grammar_file)

  collect_lr0_items(start_sym)

  parse_table = [['' for _ in range(len(terms) + len(nonterms) + 1)]
           for _ in range(len(lr0_items))]

  display(start_sym, error_sym, parse_table)
  parse_input_string(start_sym, error_sym, parse_table)


def parse_grammar(file):
  global grammar, start_sym, terms, nonterms, symbols

  for line in file:
    if line == '\n':
      break

    head = line[:line.index('->')].strip()
    prods = [
      rule.strip().split(' ')
      for rule in line[line.index('->') + 2:].split('|')
    ]

    if not start_sym:
      start_sym = head + "'"
      grammar[start_sym] = [[head]]
      nonterms.append(start_sym)

    if head not in grammar:
```

```
            grammar[head] = []

        if head not in nonterms:
            nonterms.append(head)

        for prod in prods:
            grammar[head].append(prod)

            for char in prod:
                if not char.isupper() and char not in terms:
                    terms.append(char)
                elif char.isupper() and char not in nonterms:
                    nonterms.append(char)
                    # non terminals dont produce other symbols
                    grammar[char] = []

    symbols.extend([*nonterms, *terms])


if __name__ == '__main__':
    GRAMMAR_FILE_PATH = os.path.join(os.path.dirname(__file__), 'grammar.txt')
    main()
```

3.1.4.parse_input_string.py

```
from global_vars import grammar, nonterms, terms
from slr_parser import perform_action


def parse_input_string(start, error, parse_table):
    input_str = input('\nEnter Input String' +
```

```
                '(Whitespaces is required in between lexemes): ')
    parse_str = (input_str + ' $').split()
    inp_ptr = 0
    stack = ['0']

    print(
'\n+--------+----------------------------+----------------------------+-------------------------
---+'
    )
    print('|{:^8}|{:^28}|{:^28}|{:^28}|'.format('Step', 'Stack', 'Input',
                                               'Action'))
    print(
'+--------+----------------------------+----------------------------+----------------------------
+'
    )

    step = 1

    while True:
        curr_symbol = parse_str[inp_ptr]
        stack_top = int(stack[-1])
        stack_content = ''
        input_content = ''

        print('|{:^8}|'.format(step), end=' ')

        for i in stack:
            stack_content += i
        print('{:27}|'.format(stack_content), end=' ')
        i = inp_ptr
```

```
while i < len(parse_str):
    input_content += parse_str[i]
    i += 1

print('{:>26} | '.format(input_content), end=' ')

step += 1
action = perform_action(stack_top, curr_symbol, start, error,
                        parse_table)

if '/' in action:
    print('{:^26}|'.format(action + '. Multiple conflicting actions.'))
    break

if 's' in action:
    print('{:^26}|'.format(action))
    stack.append(curr_symbol)
    stack.append(action[1:])
    inp_ptr += 1
elif 'r' in action:
    print('{:^26}|'.format(action))
    i = 0

    for head, prods in grammar.items():
        for prod in prods:
            if i == int(action[1:]):
                for _ in range(2 * len(prod)):
                    stack.pop()

                state = stack[-1]
                stack.append(head)
```

```
                    stack.append(
                        parse_table[int(state)][len(terms) +
                                                nonterms.index(head)])
                i += 1
        elif action == 'acc':
            print('{:^26}|'.format('Accepted'))

            break
        else:
            print('ERROR: Illegal symbol', curr_symbol, '|')

            break
    print(

'+--------+--------------------------+--------------------------+--------------------------
+'
    )
```

3.1.5 slr_helpers.py

```
from global_vars import grammar, nonterms, terms



def get_first(symbol, seen_syms=None):
    if seen_syms is None:
        seen_syms = []

    first_syms = []

    if symbol not in seen_syms:
        seen_syms.append(symbol)
```

```
    if symbol in terms:  # For terminal symbols
        first_syms.append(symbol)
    elif symbol in nonterms:  # For nonterminal symbols
        for prod in grammar[symbol]:
            if prod[0] in terms and prod[0] not in first_syms:
                first_syms.append(prod[0])
            elif prod[0] in nonterms:
                if prod[0] not in seen_syms:
                    first_syms += [
                        term for term in get_first(prod[0], seen_syms)
                        if term not in first_syms
                    ]

    return first_syms


def get_follow(symbol, start, seen_syms=None):
    if seen_syms is None:
        seen_syms = []

    follow_syms = []

    if symbol not in seen_syms:
        seen_syms.append(symbol)

    if symbol == start:  # Add $ to follow set of start symbol
        follow_syms.append('$')

    for head, prods in grammar.items():
        for prod in prods:
            to_follow = False
```

```
            if symbol in prod:
                next_sym_pos = prod.index(symbol) + 1

                if next_sym_pos < len(prod):
                    follow_syms += [
                        term for term in get_first(prod[next_sym_pos])
                        if term not in follow_syms
                    ]
                else:
                    to_follow = True

                if to_follow and (head not in seen_syms):
                    follow_syms += [
                        term for term in get_follow(head, start, seen_syms)
                        if term not in follow_syms
                    ]

    return follow_syms


def get_closure(items):
    closure = {**items}

    while True:
        item_len = len(closure) + sum(len(v) for v in closure.values())

        for head in list(closure):
            for prod in closure[head]:
                dot_pos = prod.index('.')

                # Checks whether or not item final
```

```
            if dot_pos + 1 >= len(prod):
                continue

            # Item not final
            prod_after_dot = prod[dot_pos + 1]

            if prod_after_dot not in nonterms:
                continue

            for prd in grammar[prod_after_dot]:
                itm = ['.'] + prd

                if prod_after_dot not in closure:
                    closure[prod_after_dot] = [itm]
                elif itm not in closure[prod_after_dot]:
                    closure[prod_after_dot].append(itm)

        if item_len == len(closure) + sum(len(v) for v in closure.values()):

            return closure
```

3.1.6 slr_parser.py

```
from global_vars import grammar, lr0_items, nonterms, symbols, terms
from slr_helpers import get_closure, get_first, get_follow



def display(start, error, parse_table):
    global grammar, lr0_items, nonterms, symbols, terms
    print('Grammar')

    for head, prods in grammar.items():
```

```
    if head == start:
        continue

    print('{:>{width}} ->'.format(head,
                              width=len(
                                  max(list(grammar.keys()), key=len))),
          end=' ')

    nprods = 0

    for prod in prods:
        if nprods > 0:
            print('|', end=' ')

        for char in prod:
            print(char, end=' ')

        nprods += 1

    print()

print('\nAugmented Grammar')
i = 0

for head, prods in grammar.items():
    for prod in prods:
        print('{:>{width}} ->'.format(head,
                                  width=len(
                                      max(list(grammar.keys()),
                                          key=len))),
              end=' ')
```

```
        for char in prod:
            print(char, end=' ')

        print()
        i += 1

print('\nTerminals   :', terms)
print('Nonterminals:', nonterms)
print('Symbols     :', symbols)

print('\nFirst')

for head in grammar:
    print('{:>{width}} ='.format(head,
                                 width=len(
                                     max(list(grammar.keys()), key=len))),
          end=' ')
    print('{', end=' ')
    nterms = 0

    for term in get_first(head):
        if nterms > 0:
            print(', ', end=' ')
        print(term, end=' ')
        nterms += 1

    print('}')

print('\nFollow')

for head in grammar:
    print('{:>{width}} ='.format(head,
```

```
                              width=len(
                                  max(list(grammar.keys()), key=len))),
          end=' ')
    print('{', end=' ')
    nterms = 0

    for term in get_follow(head, start):
        if nterms > 0:
            print(', ', end=' ')
        print(term, end=' ')
        nterms += 1

    print('}')

print('\nItems')

for i in range(len(lr0_items)):
    print('I' + str(i) + ':')

    for head, prods in lr0_items['I' + str(i)].items():
        for prod in prods:
            print('{:>{width}} ->'.format(head,
                                          width=len(
                                              max(list(grammar.keys()),
                                                  key=len))),
                  end=' ')

            for char in prod:
                print(char, end=' ')

            print()
```

```
for i in range(len(parse_table)):  # len gives number of states
    for sym in symbols:
        perform_action(i, sym, start, error, parse_table)

print('\nParsing Table')
print('+' + '--------+' * (len(terms) + len(nonterms) + 1))
print('|{:^8}|'.format('State'), end=' ')

for term in terms:
    print('{:^7}|'.format(term), end=' ')

print('{:^7}|'.format('$'), end=' ')

for nonterm in nonterms:
    if nonterm == start:
        continue

    print('{:^7}|'.format(nonterm), end=' ')

print('\n+' + '--------+' * (len(terms) + len(nonterms) + 1))

for i in range(len(parse_table)):
    print('|{:^8}|'.format(i), end=' ')

    for j in range(len(parse_table[i]) - 1):
        print('{:^7}|'.format(parse_table[i][j]), end=' ')

    print()

print('+' + '--------+' * (len(terms) + len(nonterms) + 1))
```

```
def collect_lr0_items(start):
  global lr0_items
  i = 1
  lr0_items['I0'] = get_closure({start: [['.'] + grammar[start][0]]})

  while True:
    item_len = len(lr0_items) + sum(len(v) for v in lr0_items.values())

    for idx in list(lr0_items):
      for sym in symbols:
        if goto(lr0_items[idx], sym) and (goto(lr0_items[idx], sym)
                                    not in lr0_items.values()):
          lr0_items['I' + str(i)] = goto(lr0_items[idx], sym)
          i += 1

    if item_len == len(lr0_items) + sum(
        len(v) for v in lr0_items.values()):

      return


def goto(item, symbol):
  goto_states = {}

  for head, prods in item.items():
    for prod in prods:
      for i in range(len(prod) - 1):
        if prod[i] != '.' or prod[i + 1] != symbol:
          continue

        tmp_prod = prod[:]
        tmp_prod[i], tmp_prod[i + 1] = (tmp_prod[i + 1], tmp_prod[i])
```

```
            prod_closure = get_closure({head: [tmp_prod]})

            for sym in prod_closure:
                if sym not in goto_states:
                    goto_states[sym] = prod_closure[sym]
                elif prod_closure[sym] not in goto_states[sym]:
                    goto_states[sym].extend(list(prod_closure[sym]))
    return goto_states


def perform_action(i, symbol, start, error, parse_table):
    for _, prods in lr0_items['I' + str(i)].items():
        for prod in prods:
            for j in range(len(prod) - 1):
                if prod[j] == '.' and prod[j + 1] == symbol:
                    for k in range(len(lr0_items)):
                        if goto(lr0_items['I' + str(i)],
                                symbol) == lr0_items['I' + str(k)]:

                            if symbol in terms:
                                if 'r' in parse_table[i][terms.index(symbol)]:

                                    if error != 1:
                                        print('ERROR: Shift-Reduce conflict' +
                                              ' at State ' + str(i) +
                                              ', Symbol \'' +
                                              str(terms.index(symbol)) + '\'')

                                    error = 1

                                    if 's' + str(k) not in parse_table[i][
                                            terms.index(symbol)]:
```

```
                parse_table[i][terms.index(
                    symbol
                )] = parse_table[i][terms.index(
                    symbol)] + '/s' + str(k)

                return parse_table[i][terms.index(
                    symbol)]
        else:
            parse_table[i][terms.index(
                symbol)] = 's' + str(k)
    else:
        parse_table[i][len(terms) +
                       nonterms.index(symbol)] = str(k)

    return 's' + str(k)

for lr0_head, lr0_prods in lr0_items['I' + str(i)].items():
    if lr0_head != start:
        for lr0_prod in lr0_prods:
            if lr0_prod[-1] == '.':  # final item
                k = 0

                for gram_head, gram_prods in grammar.items():
                    for gram_prod in gram_prods:
                        if (gram_head == lr0_head
                                and gram_prod == lr0_prod[:-1]
                                and (symbol in terms or symbol == '$')):

                            for term in get_follow(lr0_head, start):
                                if term == '$':
                                    index = len(terms)
                                else:
```

```
    index = terms.index(term)

if 's' in parse_table[i][index]:
    if error != 1:
        print(
            'ERROR: Shift-Reduce conflict'
            + ' at State ' + str(i) +
            ', Symbol \'' + str(term) +
            '\'')

    error = 1

    if 'r' + str(k) not in parse_table[i][
            index]:
        parse_table[i][index] = (
            parse_table[i][index] + '/r' +
            str(k))

    return parse_table[i][index]
elif parse_table[i][index] and (
        parse_table[i][index] !=
        'r' + str(k)):

    if error != 1:
        print(
            'ERROR: Reduce-Reduce conflict'
            + ' at State ' + str(i) +
            ', Symbol \'' + str(term) +
            '\'')

    error = 1
```

```
                        if 'r' + str(k) not in parse_table[i][
                                index]:
                            parse_table[i][index] = (
                                parse_table[i][index] + '/r' +
                                str(k))

                        return parse_table[i][index]
                    else:
                        parse_table[i][index] = 'r' + str(k)

                return 'r' + str(k)

            k += 1

    if start in lr0_items['I' + str(i)] and (
            grammar[start][0] + ['.'] in lr0_items['I' + str(i)][start]):
        parse_table[i][len(terms)] = 'acc'

        return 'acc'

    return ''
```

3.2 Output

Grammar

E -> E + T | T

T -> T F | F

F -> F * | a | b

Augmented Grammar

E' -> E
 E -> E + T
 E -> T
 T -> T F
 T -> F
 F -> F *
 F -> a
 F -> b

Terminals : ['+', '*', 'a', 'b']
Nonterminals: ["E'", 'E', 'T', 'F']
Symbols : ["E'", 'E', 'T', 'F', '+', '*', 'a', 'b']

First
E' = { a , b }
 E = { a , b }
 T = { a , b }
 F = { a , b }

Follow
E' = { $ }
 E = { $, + }
 T = { $, + , a , b }
 F = { $, + , a , b , * }

Items
I0:
 E' -> . E
 E -> . E + T
 E -> . T
 T -> . T F
 T -> . F

F -> . F *

F -> . a

F -> . b

I1:

E' -> E .

E -> E . + T

I2:

E -> T .

T -> T . F

F -> . F *

F -> . a

F -> . b

I3:

T -> F .

F -> F . *

I4:

F -> a .

I5:

F -> b .

I6:

E -> E + . T

T -> . T F

T -> . F

F -> . F *

F -> . a

F -> . b

I7:

T -> T F .

F -> F . *

I8:

F -> F * .

I9:

E -> E + T .

T -> T . F

F -> . F *

F -> . a

F -> . b

Parsing Table

State	+	*	a	b	$	E	T	F
0			s4	s5		1	2	3
1	s6				acc			
2	r2		s4	s5	r2			7
3	r4	s8	r4	r4	r4			
4	r6	r6	r6	r6	r6			
5	r7	r7	r7	r7	r7			
6			s4	s5			9	3
7	r3	s8	r3	r3	r3			
8	r5	r5	r5	r5	r5			
9	r1		s4	s5	r1			7

Enter Input String(Whitespaces is required in between lexemes): a + b * a

Step	Stack	Input	Action
1	0	a+b*a$	s4
2	0a4	+b*a$	r6
3	0F3	+b*a$	r4

4	0T2	+b*a$	r2
5	0E1	+b*a$	s6
6	0E1+6	b*a$	s5
7	0E1+6b5	*a$	r7
8	0E1+6F3	*a$	s8
9	0E1+6F3*8	a$	r5
10	0E1+6F3	a$	r4
11	0E1+6T9	a$	s4
12	0E1+6T9a4	$	r6
13	0E1+6T9F7	$	r3
14	0E1+6T9	$	r1
15	0E1	$	Accepted

Chapter 4: Conclusion

Parsing is the process of analyzing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. A recursive grammar if it contains production rules that are recursive, meaning that expanding a non-terminal according to these rules can eventually lead to a string that includes the same non-terminal again.The implementation of Non-Recursive Predictive Parsing method for a context free grammar as well as programs and algorithms to remove left recursion and left factoring. A production of grammar having right recursion does not create any problem for the Top down parsers. Therefore, there is no need of eliminating right recursion from the grammar.