REPORT ON

CD Mini Project

Carried out on

Design And Implementation of Simple Lexer and Parser for Example Procedure with Conditional Statements in a Hypothetical Language

Submitted to

Dr. Raju K
Associate Professor

NMAM INSTITUTE OF TECHNOLOGY, NITTE
(An Autonomous Institution under VTU, Belagavi)

In partial fulfillment of the requirements for the award of the

Degree of Bachelor of Engineering
in
Computer Science Engineering

Submitted by

Aditi Diwakar (4NM21CS006)

Anvitha Shetty (4NM21CS029)

Apeksha S Shetty (4NM21CS031)

NITTE EDUCATION TRUST | NMAM INSTITUTE OF TECHNOLOGY

CERTIFICATE

This is to certify that ***Aditi Diwakar(4NM21CS006), Anvitha Shetty(4NM21CS029), Apeksha Shetty(4NM21CS031)*** *bonafide students of NMAM Institute ofTechnology, Nitte***,** *have completed CD mini project on* ***Design and implementation of Simple Lexer and Parser for Example Procedure with Conditional Statements in Hypothetical Language*** *during March 2024- April 2024 fulfilling the partial requirements for the award of degree of Bachelor of Engineering in* **Computer Science & Engineering** *at NMAM Institute of Technology, Nitte***.**

Name and Signature of Mentor

Signature of HOD

Date: 3rd May 2024

ACKNOWLEDGEMENT

We are utmost grateful to all those whose guidance and encouragement illuminated our path and contributed to the success.

First and foremost, we would like to thank our principal **Dr. Niranjan N. Chiplunkar** for providing us with this unique opportunity to do the mini project in the 6th semester of Computer Science and engineering.

We express our gratitude to our mentors **Dr. Raju K** and **Dr. Guruprasad** for their continuous encouragement, guidance, and support throughout this project. We sincerely thank **Dr. Jyothi Shetty**, Head of Department of Computer Science and Engineering, Nitte Mahalinga Adyanthaya Memorial Institute of Technology, Nitte.

We also thank all those who have supported us throughout the entire duration of our project.

Finally, we thank the staff members of the Department of Computer Science and Engineering and all our friends for their honest opinions and suggestions throughout the course of our project.

ABSTRACT

This project focuses on designing a Lexer and Parser for a simple hypothetical language. These components are essential parts of the Analysis phase in the compilation process, responsible for identifying tokens within a given program and verifying their syntactic correctness based on predefined production rules. The main program requires two inputs: the source program to be processed and the grammar rules used for parsing. The goal of the project is to generate a parsed sequence that can be utilized in subsequent stages of the compiler.

We generate the parse table which has entries for each terminal and non-terminal identified in them. Before the generation of the parse table, we identified the FIRST of each terminal using a recursive method. The final stage is the parsing which is done by using the standard CLR parsing steps.

The outcome of the project is to identify the parsing actions taken by the grammar for proper and invalid source code along with this we are generating the parse table for the given input.

TABLE OF CONTENTS

INTRODUCTION

COMPILER:

A compiler is a software that takes a program written in a high-level language and translates it into an equivalent program in a target language. Most specifically a compiler takes a computer program and translates it into an object program. Some other tools associated with the compiler are responsible for making an object program into executable form

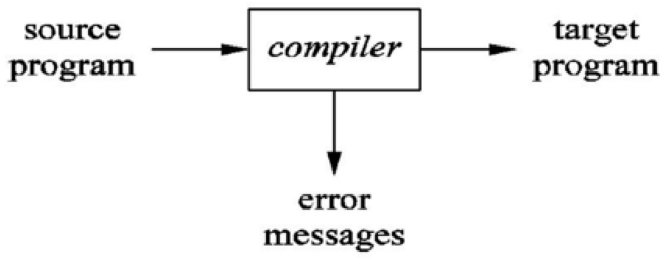


Figure 1: Compiler

PHASES OF COMPILERS:

Typically, a compiler includes several functional parts. For example, a conventional compiler may include a lexical analyser that looks at the source program and identifies successive "tokens" in the source program. A conventional compiler also includes a parser or syntactical analyser, which takes as an input a grammar defining the language being compiled and a series of actions associated with the grammar.

The syntactical analyser builds a "parse tree" for the statements in the source program in accordance with the grammar productions and actions.

For each statement in the input source program, the syntactical analyser generates a parse tree of the source input in a recursive, "bottom-up" manner in accordance with relevant productions and actions in the grammar. Generation of the parse tree allows the syntactical analyser to determine whether the parts of the source program comply with the grammar. If not, the syntactical analyser generates an error.

CLASSIFICATION OF COMPILER PHASES:

There are two major parts of a compiler phases: Analysis and Synthesis. In analysis phase, an intermediate representation is created from the given source program that contains:

1.Lexical Analyzer

2.Syntax Analyzer

3.Semantic Analyzer

4.In synthesis phase, the equivalent target program is created from this intermediate representation.

1.Intermediate code Generator

2.Code Optimization

3.Code Generation

LEXICAL ANALYZER:

Lexical analyzer takes the source program as an input and produces a string of tokens or lexemes. Lexical Analyzer reads the source program character by character and returns the tokens of the source program. The process of generation and returning the tokens is called lexical analysis. Representation of lexemes in the form of tokens as:

<token-name, attribute-value>

SYNTAX ANALYSER:

A Syntax Analyzer creates the syntactic structure (generally a parse tree) of the given program. In other words, a Syntax Analyzer takes output of lexical analyzer (list of tokens) and produces a parse tree. A syntax analyzer is also called as a parser. The parser checks if the expression made by the tokens is syntactically correct.

SEMANTIC ANALYSER:

Semantic analyzer takes the output of syntax analyzer. Semantic analyzer checks a source program for semantic consistency with the language definition. It also gathers type information for use in intermediate-code generation.

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.

CODE OPTIMISER:

The code optimizer takes the code produced by the intermediate code generator. The code optimizer reduces the code (if the code is not already optimized) without changing the meaning of the code. The optimization of code is in terms of time and space.

CODE GENERATION:

This produces the target language in a specific architecture. The target program is normally is an object file containing the machine codes. Memory locations are selected for each of the variables used by the program.

SYMBOL TABLE:

It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' 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.

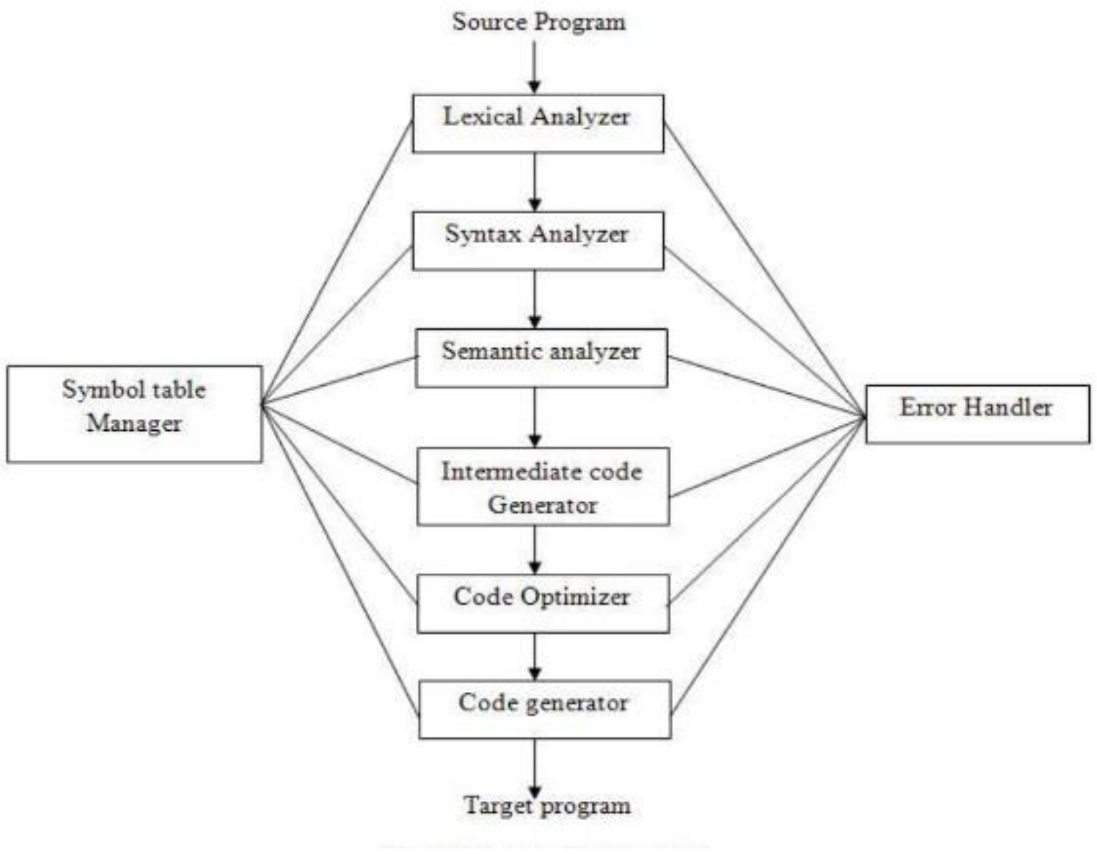


Figure 2: Phases Of A Compiler

DESIGN

LEXICAL ANALYZER

Lexical analysis is the starting phase of the compiler. It gathers modified source code that is written in the form of sentences from the language pre-processor. The lexical analyzer is responsible for breaking these syntaxes into a series of tokens, by removing whitespace in the source code. If the lexical analyzer gets any invalid token, it generates an error. The stream of character is read by it and it seeks the legal tokens, and then the data is passed to the syntax analyzer, when it is asked for.

Lexical Analysis Process:

During lexical analysis, the source code undergoes the following steps:
Tokenization: The source code is divided into individual tokens, representing the fundamental units of information within the program.
Whitespace Removal: Any extraneous whitespace characters, such as spaces, tabs, and newline characters, are eliminated to streamline the tokenization process.
Comment Removal: Comments within the source code are omitted from further consideration during tokenization, as they do not contribute to the program's semantics.
Token Identification: Each token is identified and classified based on its type, such as operators, keywords, identifiers, constants, and delimiters.
Error Detection: Lexical analysis includes error detection mechanisms to identify invalid tokens or syntax errors within the source code.

SYNTAX ANALYZER

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or Syntax tree.
The parse tree is constructed by using the pre-defined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. if not, the error is reported by the syntax analyzer.

LR parser

LR parser is a bottom-up parser for context-free grammar that is very generally used by computer programming language compiler and other associated tools. LR parser reads their input from left to right and produces a right-most derivation.

CLR Parser

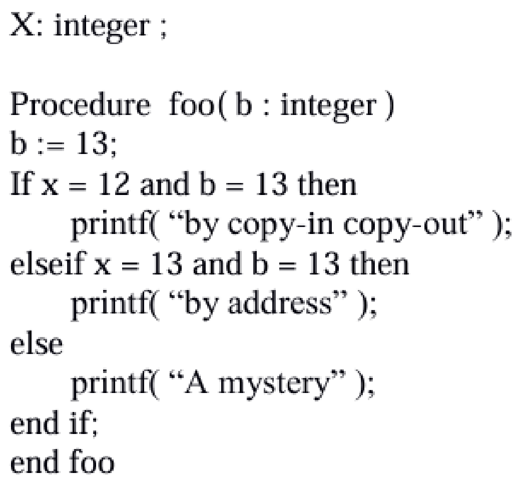
Constructing a CLR(1) parsing table involves starting with a properly built LR(1) parsing table that outlines shift, reduce, or goto actions for each state and symbol in your grammar. To enhance this table, incorporate lookahead symbols to handle conflicts like shift-reduce and reduce-reduce situations. For each item [A -> α . β, a] in state i, determine if a terminal symbol t follows β, and if so, add either a shift action for t in state i or a reduce action for A -> α β using lookahead t in state i. When conflicts arise, such as in shift-reduce or reduce-reduce scenarios, follow specific rules like preferring shift in shift-reduce conflicts when a terminal symbol can follow both a production and a terminal symbol in a lookahead set, or resolving reduce-reduce conflicts by choosing the production that appears first in the grammar. Finally, construct the CLR(1) parsing table by merging LR(1) table entries with lookahead information, including entries for states, terminal symbols, and non-terminal symbols to reflect appropriate parsing actions (shift, reduce, or goto). Debugging and testing using example inputs and utilizing parsers or tools supporting CLR(1) parsing table generation can ensure accuracy and efficiency in parsing operations.

PROBLEM STATEMENT

The project develops a compiler for Language X, which supports integer data type and includes a procedure `foo` with a conditional statement. The compiler should handle lexical analysis, syntax checking, semantic analysis for variable scoping and type checking, and generate optimized intermediate representation for translation into machine code. Ensure error handling for syntax and semantic errors and provide documentation detailing the design and usage of the compiler.

The problem statement is as follows:

Design a compiler for the following Hypothetical Language



```
X: integer ;

Procedure  foo( b : integer )
b := 13;
If x = 12 and b = 13 then
      printf( “by copy-in copy-out” );
elseif x = 13 and b = 13 then
      printf( “by address” );
else
      printf( “A mystery” );
end if;
end foo
```

OBJECTIVES

The objective of this project is to develop a Parser and Lexer for a simple hypothetical language as part of the compilation process. Specifically, the goals are:

- Design and implement a Lexer to tokenize the input source program, identifying the lexical units such as keywords, identifiers, operators, and literals.
- Design and implement a Parser to analyze the tokenized input according to predefined grammar rules, verifying the syntactic correctness of the source program.

METHODOLOGY

CLR parsing, or Canonical LR parsing, is a parsing technique used in compiler construction to parse context-free grammars efficiently. It combines the efficiency of LR parsing with the ability to handle a wider class of grammars than simple LR parsers.

The context free grammar used for this problem statement :

P->S

S->v:t;L

L->v(:t):X

X->ov;N

N->oop();

R R->ooT

T->p();J

J->p();Z

Z->dv;d

Algorithm to construct LR (1) Parsing Table:

Step 1: First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.

Step 2: Calculate First () and Follow () for all non-terminals.

First(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.

Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.

Step 3: For each production A –> α. (A tends to alpha)

Find First(α) and for each terminal in First(α), make entry A –> α in the table.

If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.

If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –>α in the table for the $.

Construction Of CLR Parsing Table:

Constructing a CLR(1) parsing table involves similar steps to constructing an SLR parsing table, but with the added complexity of calculating lookahead symbols.

1. Construct C’ = { I0, I1, … In }, the collection of sets of LR(1) items for G’ (augmented grammar).
2. Calculate Lookahead Symbols:

For each LR(1) item [A -> α . B β, a], calculate the lookahead symbols for items in the closure of that item.

Lookahead symbols are determined by:

If [A -> α . B β, a] is in Ii and B -> γ is a production rule, then add FIRST(βa) to LOOKAHEAD[B, i].

If [A -> α . B β, a] is in Ii and B -> ε is a production rule (B can derive ε), then add FOLLOW(A) to LOOKAHEAD[B, i].

3. State i is constructed from Ii: The parsing actions for state i are determined as follows:

If [A -> α . a β, b] is in Ii, and GOTO(Ii, a) = Ij, then set ACTION[i, a] to “shift j”.

If [A -> α ., a] is in Ii, A ≠ S’, then set ACTION[i, a] to “reduce A -> α” for each terminal a in LOOKAHEAD[A, i].

If [S’ -> S ., $] is in Ii, then set ACTION[i, $] to “accept”.

4. 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.

4. 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.

5. All entries not defined by rules (3) and (4) are made “error”.

6. The initial state of the parser is the one constructed from the set of items containing [S’ -> .S, $].

IMPLEMENTATION

Question: program.txt

X: integer ;
Procedure foo(b : integer)b :=
13;
If x = 12 and b = 13 then
printf(“by copy-in copy-out”);elseif x
= 13 and b = 13 then
printf(“by address”);
else
printf(“A mystery”);
end if; end
foo

PROGRAM CODE:

```
import numpy as np
import pandas as pd
```

This code snippet represents a basic implementation of a lexical analyzer for a programming language, capable of identifying tokens such as keywords, symbols, digits, and identifiers.

The code defines various symbols, keywords, and whitespace characters used in the programming language being analyzed.

It iterates through each character in the input string, building lexemes until it encounters whitespace or a character that indicates the end of a lexeme.

It classifies the lexemes into categories such as symbols, keywords, digits, and identifiers.

```
LEXER_TABLE=pd.DataFrame(columns=['TOKEN','IDENTITY'],index=range(1))

def lexer(filename):
  with open(filename, 'r') as myfile:
    string = myfile.read().replace('\n',' ')
  SYMBOLS=['(',')',';',',',':','\'']
  SYMBOLS_1=['(',
             ')',
             ';',
             ',',
             ':',
             '\'',
             '+',
             '=',
             '>'
             ]
```

```
SYMBOLS_1_DUP=[['Open bracket','('],
               ['Close bracket',')'],
               ['Semi colon',';'],
               ['Comma',','],
               ['Colon',':'],
               ['Single quote','\''],
               ['Plus','+'],
               ['Equal','='],
               ['Greater than','>']]

keywords=['integer','main','while','begin','end',
          'expr',' ','\n','Procedure','If','elseif',
          'else','then','printf','and','or']

keywords_def=[['t','integer'],
['m','main'],
['l','while'],
['b','begin'],
['d','end'],
['e','expr'],
['s',' '],
['o','+'],
['o','='],
['n','\n'],
['p','Procedure'],
['x','If'],
['z','elseif'],
['y','else'],
['f','then'],
['c','printf'],
['&','and'],
['|','or'],
['o','>']
]

KEYWORDS=SYMBOLS_1 + keywords
white_space=' '
lexeme=''
list=[]
string=string.replace('\t','')
for i,char in enumerate(string):
  if char != white_space:
    lexeme += char
  if(i+1 < len(string)):
      if string[i+1] == white_space or string[i+1] in KEYWORDS or
lexeme   in KEYWORDS:
        if lexeme!='':
          if string[i+1]=='i' and lexeme=='else':
            continue
          else:
            list.append(lexeme.replace('\n','<newline>'))
            lexeme=''
            list.append(lexeme.replace('\n','<newline>'))
  s=''
  j=0
  try:
    while(True):
      list.remove('')
  except ValueError:
    pass
  for item in list:
    for i in keywords_def:
      if i[1]==item:
        s=s+i[0]
    if item in SYMBOLS:
      s=s+item
    elif item.isdigit():
      s=s+'a'
    elif item not in KEYWORDS:
      s=s+'v'
  for i in list:
    for k in SYMBOLS_1_DUP:
      if i==k[1]:
        LEXER_TABLE.at[j,'TOKEN']=i
        LEXER_TABLE.at[j,'IDENTITY']=k[0]
        j=j+1
        break
    if i in keywords:
      LEXER_TABLE.at[j,'TOKEN']=i
      LEXER_TABLE.at[j,'IDENTITY']='Keyword'
      j=j+1
      continue
    if i.isdigit():
      LEXER_TABLE.at[j,'TOKEN']=i
      LEXER_TABLE.at[j,'IDENTITY']='Digit'
      j=j+1
      continue
    elif i not in KEYWORDS:
      LEXER_TABLE.at[j,'TOKEN']=i
      LEXER_TABLE.at[j,'IDENTITY']='Identifier'
      j=j+1
  print(LEXER_TABLE)
  return s

EPSILON = "ε"
```

This code snippet defines a function named get_productions(X) which takes a nonterminal symbol X as input and returns a list of productions in a given grammar where X is on the left-hand side (lhs) of the production.

```
def get_productions(X):
  productions = []
  for prod in grammar:
    lhs, rhs = prod.split('->')
    if lhs == X:
      rhs = '.'+rhs
      productions.append('->'.join([lhs, rhs]))
  return productions
```

This code snippet defines a function named closure(I) which computes the closure of an item set I in a parsing context.

```
def closure(I):
  for production, a in I:
    if production.endswith("."):
      continue
    lhs, rhs = production.split('->')
    alpha, B_beta = rhs.split('.')
    B = B_beta[0]
    beta = B_beta[1:]
    beta_a = a
    if beta:
      beta_a = beta[0]+a
    first_beta_a = first(beta_a)
    for b in first_beta_a:
      B_productions = get_productions(B)
      for gamma in B_productions:
        new_item = (gamma, b)
        if (new_item not in I):
          I.append(new_item)
  return I
```

This code defines a function get_symbols(grammar) which extracts the set of terminal symbols and non-terminal symbols from a given grammar.

- Function Definition: It defines a function get_symbols(grammar) which takes a grammar (a set of production rules) as input.
- Initialization: It initializes two sets, terminals and non_terminals, to store terminal symbols and non-terminal symbols, respectively.

```
def get_symbols(grammar):
  terminals = set()
  non_terminals = set()
  for production in grammar:
    lhs, rhs = production.split('->')
    non_terminals.add(lhs)
    for x in rhs:
      terminals.add(x)
      terminals = terminals.difference(non_terminals)
  terminals.add('$')
  print(terminals, non_terminals)
  return terminals, non_terminals
```

This code defines a function named first(symbols) which computes the FIRST set for a given string of symbols in the context of a grammar. The FIRST set of a string represents the set of terminals that can start a string derived from the given symbols

```
def first(symbols):
  final_set = set()
  for X in symbols:
    first_set = set()
    if isTerminal(X):
      final_set.add(X)
    else:
      for production in grammar:
        lhs, rhs = production.split('->')
        if lhs == X:
          for i in range(len(rhs)):
            y = rhs[i]
            if y == X:
              continue
            first_y = first(y)
            if EPSILON in first_y:
              first_y.replace(EPSILON,"")
              first_set.add(first_y)
            continue
          else:
            first_set.add(first_y)
            break
    for i in first_set:
      if EPSILON not in i:
        final_set.add(i)
      else:
        i.replace(EPSILON,"")
        final_set.add(i)
  return "".join(list(final_set))
```

This code defines a function named shift_dot(production) that shifts the dot ('.') symbol one position to the right in a production rule.

1.Function Definition: It defines a function shift_dot(production) which takes a production rule as input.

2.Production Splitting: It splits the production rule into its left-hand side (lhs) and right-hand side (rhs) components using the arrow symbol '->'.

3.Dot Shifting:

- It splits the rhs at the dot ('.') symbol.
- If the dot is at the end of the rhs, it prints a message indicating that the dot cannot be shifted further and returns.
- If the length of the rhs after the dot is 1, it means the dot is just before the last symbol. In this case, it moves the dot to the end of the rhs by appending a dot.
- If the length of the rhs after the dot is greater than 1, it moves the dot one position to the right by appending a dot after the first symbol following the dot.
- It reconstructs the rhs with the shifted dot.

4.Return: It returns the modified production rule with the dot shifted.

```
def isTerminal(symbol):
  return symbol in terminals

def shift_dot(production):
  lhs, rhs = production.split('->')
  x, y = rhs.split(".")
  if(len(y) == 0):
    print("Dot at the end!")
    return
  elif len(y) == 1:
    y = y+"."
  else:
    y = y[0]+"."+y[1:]
  rhs = "".join([x, y])
  return "->".join([lhs, rhs])
```

This code defines a function named goto(I, X) that computes the GOTO set for a given LR(1) item set I and a symbol X. The GOTO set represents the set of LR(1) items that can be reached from the items in I by shifting the dot over the symbol X.

```
def goto(I, X):
  J = []
  for production, look_ahead in I:
    lhs, rhs = production.split('->')
    if "."+X in rhs and not rhs[-1] == '.':
      new_prod = shift_dot(production)
      J.append((new_prod, look_ahead))
  return closure(J)
```

This code defines a function named set_of_items(display=False) that generates a set of LR(1) items and their corresponding states for constructing LR(1) parsing tables. It essentially implements the construction of the LR(1) automaton.

```
def set_of_items(display=False):
  num_states = 1
  states = ['I0']
  items = {'I0': closure([('P->.S', '$')])}
  for I in states:
      for X in pending_shifts(items[I]):
        goto_I_X = goto(items[I], X)
        if goto_I_X != None:
          if len(goto_I_X) > 0 and goto_I_X not in items.values():
            new_state = "I"+str(num_states)
            states.append(new_state)
            items[new_state] = goto_I_X
            num_states += 1
  if display:
    for i in items:
      print("State", i, ":")
      for x in items[i]:
```

```
      print(x)
    print()
  return items
```

This code defines a function named pending_shifts(I) that determines the symbols for which a shift operation is pending in a given LR(1) item set I. The shift operation refers to shifting the dot ('.') symbol one position to the right in LR(1) items.

```
def pending_shifts(I):
  symbols = []
  for production, _ in I:
    lhs, rhs = production.split('->')
    if rhs.endswith('.'):
      continue
    beta = rhs.split('.')[1][0]
    if beta not in symbols:
      symbols.append(beta)
  return symbols
```

The following code snippet performs two functions:

- done_shifts(I): This function determines LR(1) items in a given LR(1) item set I for which the shift operation is completed. It identifies the items where the dot ('.') symbol is at the end of the production, indicating that the shift operation for that symbol has been done.
- get_state(C, I): This function retrieves the state name corresponding to a given LR(1) item set I from a dictionary of LR(1) item sets C.done.

```
def done_shifts(I):
  done = []
  for production, look_ahead in I:
    if production.endswith('.') and production != 'P->S.':
      done.append((production[:-1], look_ahead))
  return done
def get_state(C, I):
  key_list = list(C.keys())
  val_list = list(C.values())
  i = val_list.index(I)
  return key_list[i]
```

This code snippet implements the construction of the parsing tables (ACTION and GOTO tables) for a CLR(1) parser. CLR parsing is a type of bottom-up parsing used to parse context-free grammars.

```
def CLR_construction(num_states):
  C = set_of_items()
  ACTION = pd.DataFrame(columns=list(terminals),
index=range(num_states))
  GOTO = pd.DataFrame(columns=list(non_terminals),
index=range(num_states))
  for Ii in C.values():
    i = int(get_state(C, Ii)[1:])
    pending = pending_shifts(Ii)
    for a in pending:
      Ij = goto(Ii, a)
      j = int(get_state(C, Ij)[1:])
      if isTerminal(a):
        ACTION.at[i, a] = "Shift "+str(j)
      else:
        GOTO.at[i, a] = j
    for production, look_ahead in done_shifts(Ii):
      ACTION.at[i, look_ahead] = "Reduce " +
str(grammar.index(production)+1)
    if ('P->S.', '$') in Ii:
      ACTION.at[i, '$'] = "Accept"
  ACTION.replace(np.nan, '', regex=True, inplace=True)
  GOTO.replace(np.nan, '', regex=True, inplace=True)
  print(ACTION)
  print(GOTO)
  return ACTION, GOTO
```

This code snippet defines a function named parse_string(string, ACTION, GOTO) which performs the parsing of a string according to the provided parsing tables (ACTION and GOTO) for a CLR(1) parser. Overall, this function performs CLR(1) parsing of a string based on the provided parsing tables and logs the parsing process for analysis and debugging.

```
def parse_string(string, ACTION, GOTO):
  row = 0
  cols = ['Stack', 'Input', 'Output']
  if not string.endswith('$'):
    string = string+'$'
  ip = 0
  PARSE = pd.DataFrame(columns=cols)
  input = list(string)
  stack = ['$', '0']
  while True:
    S = int(stack[-1])
    a = input[ip]
    action = ACTION.at[S, a]
    new_row = ["".join(stack), "".join(input[ip:]), action]
    if 'S' in action:
      S1 = action.split()[1]
      stack.append(a)
      stack.append(S1)
      ip += 1
    elif "R" in action:
      i = int(action.split()[1])-1
      A, beta = grammar[i].split('->')
      for _ in range(2*len(beta)):
        stack.pop()
      S1 = int(stack[-1])
      stack.append(A)
      stack.append(str(GOTO.at[S1, A]))
      new_row[-1] = "Reduce "+grammar[i]
    elif action == "Accept":
      PARSE.loc[row] = new_row
      print(PARSE,"\n")
      print("\nThe Source code entered is according to the grammar\n")
      return PARSE
    else:
      print(PARSE,"\n")
      print("\nThe Source code entered is incorrect\n")
      break
    PARSE.loc[row] = new_row
    row += 1
```

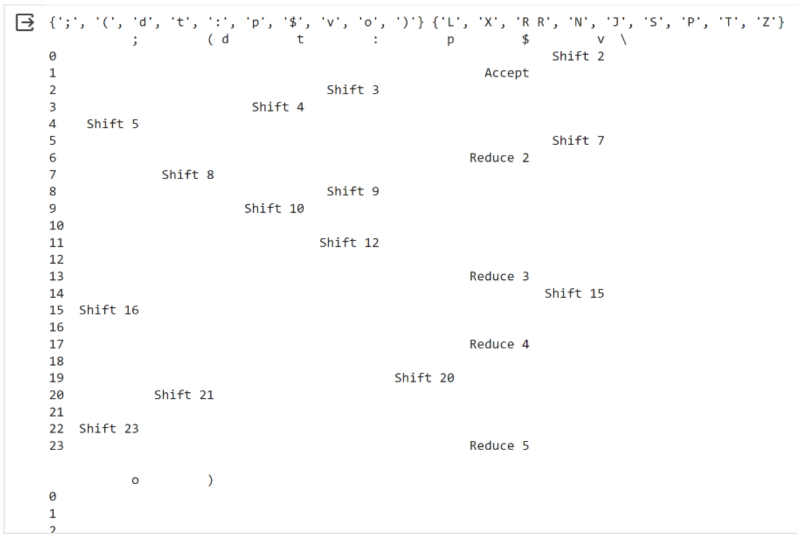
This code snippet performs a series of operations for CLR(1) parsing based on a given grammar file and a program file. It demonstrates the workflow of CLR(1) parsing, starting from reading the grammar, constructing LR(1) item sets and parsing tables, performing lexical analysis, and finally parsing the input program string.

```
def get_grammar(filename):
  grammar = []
  F = open(filename, "r")
  for production in F:
    grammar.append(production[:-1])
  return grammar
grammar = get_grammar("gramm3.txt")
```

```
terminals, non_terminals = get_symbols(grammar)
symbols = terminals.union(non_terminals)
C = set_of_items()
ACTION, GOTO = CLR_construction(num_states=len(C))
string = "".join(lexer('program.txt'))
print(string)
PARSE_TABLE = parse_string(string, ACTION, GOTO)
```

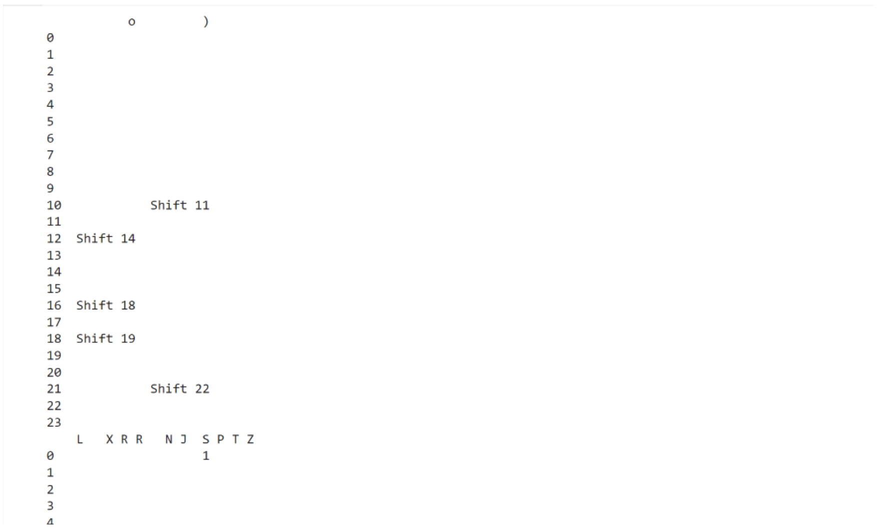
RESULT

CLR Table:



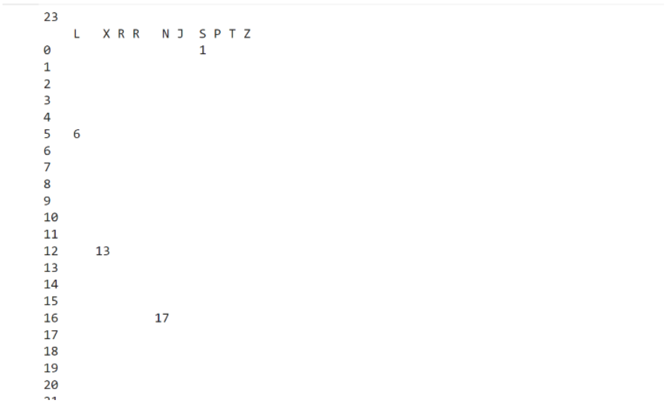
```
{';', '(', 'd', 't', ':', 'p', '$', 'v', 'o', ')'} {'L', 'X', 'R R', 'N', 'J', 'S', 'P', 'T', 'Z'}
           ;         ( d         t         :         p         $         v  \
0                                                                  Shift 2
1                                                         Accept
2                                    Shift 3
3                          Shift 4
4    Shift 5
5                                                                  Shift 7
6                                                        Reduce 2
7              Shift 8
8                                    Shift 9
9                         Shift 10
10
11                                  Shift 12
12
13                                                      Reduce 3
14                                                                Shift 15
15  Shift 16
16
17                                                      Reduce 4
18
19                                            Shift 20
20            Shift 21
21
22  Shift 23
23                                                      Reduce 5

           o         )
0
1
2
```

Figure 3



```
           o         )
0
1
2
3
4
5
6
7
8
9
10            Shift 11
11
12  Shift 14
13
14
15
16  Shift 18
17
18  Shift 19
19
20
21            Shift 22
22
23
    L   X R R   N J  S P T Z
0                    1
1
2
3
4
```

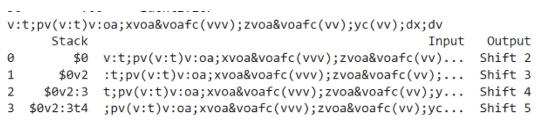
Figure 4



```
23
    L   X R R   N J  S P T Z
0                    1
1
2
3
4
5   6
6
7
8
9
10
11
12     13
13
14
15
16             17
17
18
19
20
21
```

Figure 5

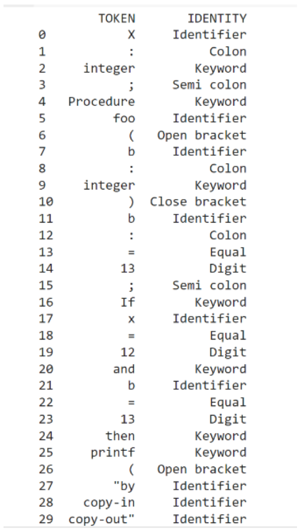
Parsing Table:



```
v:t;pv(v:t)v:oa;xvoa&voafc(vvv);zvoa&voafc(vv);yc(vv);dx;dv
      Stack                                             Input   Output
0        $0  v:t;pv(v:t)v:oa;xvoa&voafc(vvv);zvoa&voafc(vv)...  Shift 2
1      $0v2  :t;pv(v:t)v:oa;xvoa&voafc(vvv);zvoa&voafc(vv);...  Shift 3
2    $0v2:3  t;pv(v:t)v:oa;xvoa&voafc(vvv);zvoa&voafc(vv);y...  Shift 4
3  $0v2:3t4  ;pv(v:t)v:oa;xvoa&voafc(vvv);zvoa&voafc(vv);yc...  Shift 5
```

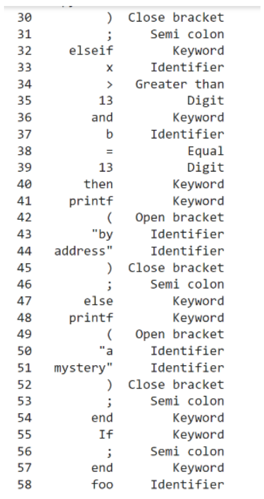
Figure 6

Tokens:



```
         TOKEN       IDENTITY
0            X     Identifier
1            :          Colon
2      integer        Keyword
3            ;      Semi colon
4    Procedure        Keyword
5          foo     Identifier
6            (   Open bracket
7            b     Identifier
8            :          Colon
9      integer        Keyword
10           )  Close bracket
11           b     Identifier
12           :          Colon
13           =          Equal
14          13          Digit
15           ;     Semi colon
16          If        Keyword
17           x     Identifier
18           =          Equal
19          12          Digit
20         and        Keyword
21           b     Identifier
22           =          Equal
23          13          Digit
24        then        Keyword
25      printf        Keyword
26           (   Open bracket
27         "by     Identifier
28     copy-in     Identifier
29   copy-out"     Identifier
```

Figure 7



```
30           )  Close bracket
31           ;     Semi colon
32      elseif        Keyword
33           x     Identifier
34           >   Greater than
35          13          Digit
36         and        Keyword
37           b     Identifier
38           =          Equal
39          13          Digit
40        then        Keyword
41      printf        Keyword
42           (   Open bracket
43         "by     Identifier
44    address"     Identifier
45           )  Close bracket
46           ;     Semi colon
47        else        Keyword
48      printf        Keyword
49           (   Open bracket
50          "a     Identifier
51    mystery"     Identifier
52           )  Close bracket
53           ;     Semi colon
54         end        Keyword
55          If        Keyword
56           ;     Semi colon
57         end        Keyword
58         foo     Identifier
```

Figure 8

CONCLUSION

The hypothetical language described by the provided grammar exhibits a well-defined syntax structure, featuring non-terminals like P, S, L, X, N, R, T, J, and Z that represent distinct syntactic constructs within programs. This structure is complemented by a set of terminals such as v, :, ;, (,), o, p, and d, which correspond to lexical elements like variable names, type declarations, punctuation symbols, and specific keywords. The grammar encompasses rules for variable declarations (e.g., v:t;), function calls (e.g., oop();), and structured statements (e.g., p();, dv;d), indicating support for basic expressions and procedural or functional programming styles. Additionally, the presence of type declarations using t (e.g., v:t;) suggests a rudimentary type system. Although control flow constructs like conditionals and loops are not explicitly defined in the grammar, the language's design allows for the incorporation of such features and more complex expressions through potential extensions to the grammar. These extensions could include conditional statements, loops, data structures, and enhanced expression capabilities, aligning the language more closely with comprehensive programming paradigms and requirements.

REFERENCES

- https://www.geeksforgeeks.org/program-calculate-first-follow-sets-givengrammar
- https://www.geeksforgeeks.org/closure-properties-of-context-freelanguages