COMPILER DESIGN PROJECT

A Mini Project Report Submitted by

FARAZ SHABBIR SHAIKH (4NM17CS061)

KARTHIK U KUMAR (4NM17CS080)

UNDER THE GUIDANCE OF

Ms. Minu P Abraham

Assistant Professor

Department of Computer Science and Engineering

in partial fulfillment of the requirements for the award of the Degree of

Bachelor of Engineering in
Computer Science & Engineering

from

Visveshvaraya Technological University, Belgaum

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

N.M.A.M. INSTITUTE OF TECHNOLOGY

(An Autonomous Institution under VTU, Belgaum) (AICTE approved, NBA Accredited, ISO 9001:2008 Certified) NITTE -574 110, Udupi District, KARNATAKA.

May 2020

NITTE EDUCATION TRUST | N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India
(ISO 9001:2015 Certified), Accredited with 'A' Grade by NAAC
☎: 08258 - 281039 – 281263, Fax: 08258 – 281265
Department of Computer Science and Engineering
B.E. CSE Program Accredited by NBA, New Delhi from 1-7-2018 to 30-6-2021

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CERTIFICATE

"Compiler Design Project"

is a bonafide work carried out by

Faraz Shabbir Shaikh(4NM17CS061)
Karthik U Kumar(4NM17CS080)

in partial fulfillment of the requirements for the award of
Bachelor of Engineering Degree in Computer Science and Engineering
prescribed by Visvesvaraya Technological University,
Belgaum during the year 2018-2019.

It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report.

The Mini project report has been approved as it satisfies the academic requirements in respect of the project work prescribed for the Bachelor of Engineering Degree.

Signature of Guide Signatureof HOD

ACKNOWLEDGEMENT

We believe that our project will be complete only after we thank the people who have contributed to make this project successful.

First and foremost, our sincere thanks to our beloved principal, **Dr. Niranjan N. Chiplunkar** for giving us an opportunity to carry out our project work at our college and providing us with all the needed facilities.

We express our deep sense of gratitude and indebtedness to our guide **Ms. Minu P Abraham**, Assistant Professor, Department of Computer Science and Engineering, for his inspiring guidance, constant encouragement, support and suggestions for improvement during the course of our project.

We sincerely thank **Dr. K.R. Udaya Kumar Reddy**, Head of Department of Computer Science and Engineering, Nitte Mahalinga Adyantaya 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.

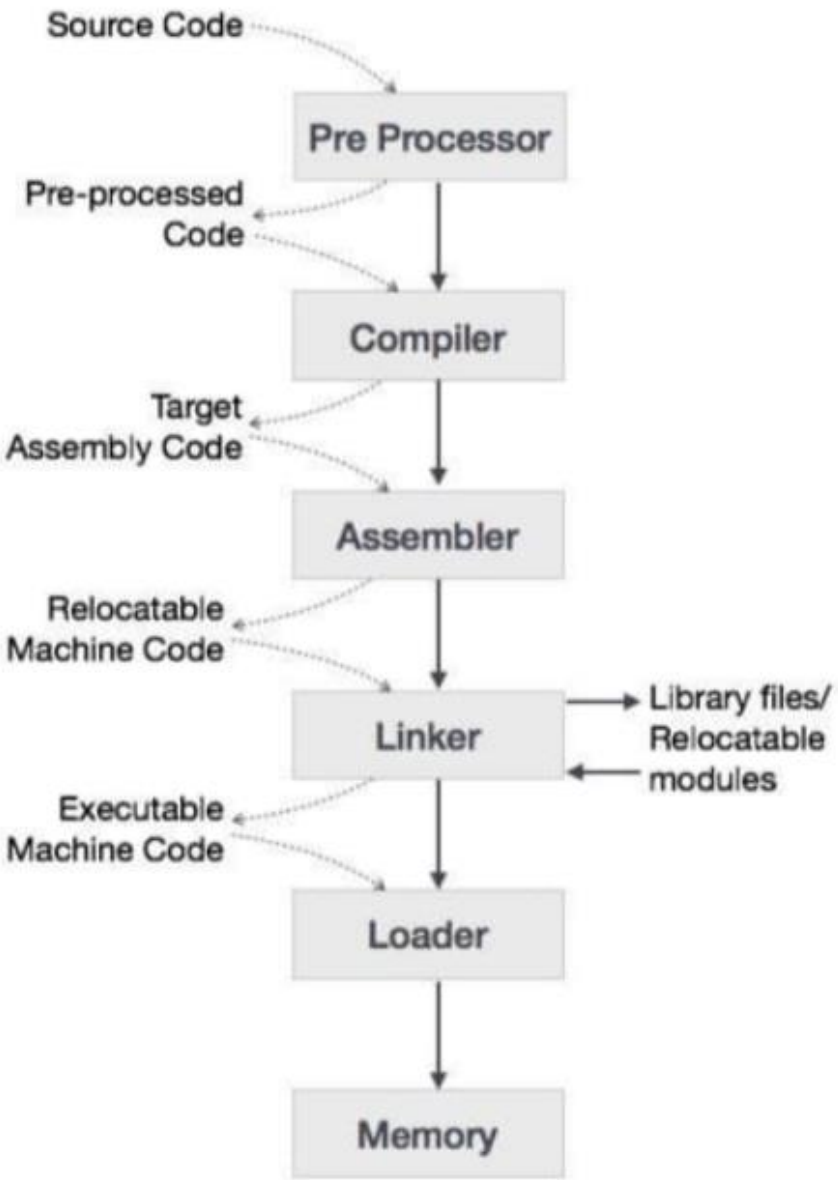
Faraz Shabbir Shaikh(4NM17CS061)
Karthik U Kumar(4NM17CS080)

TABLE OF CONTENTS

ABSTRACT

Computers are balanced mixture of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of electronic charge, which is counterpart of binary language in software programming. Binary language has only two characters ‘0’ and ‘1’ which forms the low level language. Compiler is a software which converts a program written in high level language(Source Language) to low level language(Object/Target/Machine Language).

LANGUAGE PROCESSING SYSTEM



INTRODUCTION

What is a compiler?

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required. This is where the compiler comes in. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. Using a high-level language for programming has a large impact on how fast programs can be developed.

The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.
- Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.
- On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some timecritical programs are still written partly in machine language.
- A good compiler will, however, be able to get very close to the speed of handwritten machine code when translating well-structured programs.

THE PHASES OF A COMPILER

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.

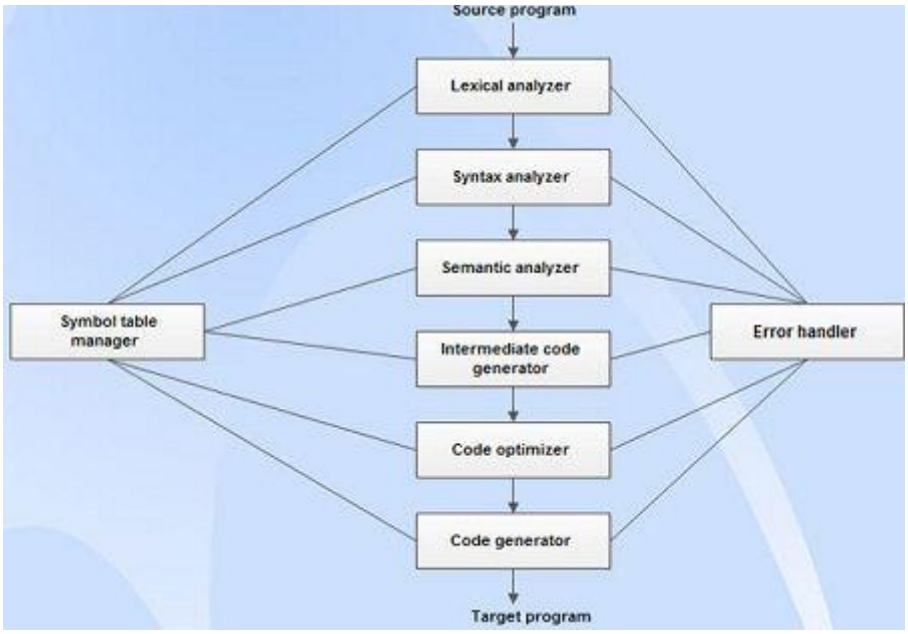


Fig: Phases of Compiler

Lexical Analysis

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

<token-name, attribute-value>

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this 3 phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

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

Intermediate Code Generation

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

Code Optimization

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

Code Generation

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

SYMBOL TABLE

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

LEXICAL ANALYSIS

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre-processors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any hitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase. In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done. However, there are reasons for keeping the phases separate:

- Efficiency: A lexer may do the simple parts of the work faster than the more general parser can. Furthermore, the size of a system that is split in two may be smaller than a combined system. This may seem paradoxical but, as we shall see, there is a non- linear factor involved which may make a separated system smaller than a combined system.
- Modularity: The syntactical description of the language need not be cluttered with small lexical details such as white-space and comments.
- Tradition: Languages are often designed with separate lexical and syntactical phases in mind, and the standard documents of such languages typically separate lexical and syntactical elements of the languages.

Token:

Token is a sequence of characters that can be treated as a single logical entity.
Typical tokens are,
1) Identifiers
2) keywords
3) operators
4) special symbols
5)constants

Pattern:

A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme:

A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Problem statement:

Design a compiler(Lexical and Parser phase) for the following hypothetical languages.

```
int main()
begin
   int n1, n2, i, gcd;
   if(expr relop expr)
          gcd = i;
   for(i=1; expr relop expr; ++i)
   begin
      gcd=1;
   end
end
```

LEXICAL PROGRAM FOR THE ABOVE PROBLEM STATEMENT

```
file=open('Question.txt','r')
operators=[]
keyw=[]
ide=[]
num=[]
k=["int","main","for","if","end","begin"]
op=["<",">","=","!"]
for line in file:
      for word in line.split():
            if word in ["=",">","<"]:
                  operators.append(word)
            elif word.isalnum():
                  if word in k:
                        keyw.append(word)
                  else:
                        if word.isdigit():
                              num.append(word)
                        else:
                              ide.append(word)
            flag=0
            j=0
            for x in list(word):
                  if x in op and flag==0:
                        flag=1
                        l=x
                        continue
                  if x in ["=","-","+","%","/","*"] and flag==0:
                        operators.append(x)
                        continue
                  if(x in ["="] and flag==1):
                        flag=0
                        operators.append(l+x)
                  if(x.isalnum() and j==0):
                        str=""
```

```
                        j=1
                        str+=x
                        continue
                if(x.isalnum() and j==1):
                        str+=x
                        continue
                if(x.isalnum()!="True" and j==1):
                        if str in k:
                                keyw.append(str)
                        else:
                                if str.isdigit():
                                        num.append(str)
                                else:
                                        ide.append(str)
                        j=0
print "There are total of",len(operators),"Operators. They are:"
for w in operators:
        print(w)
print "There are total of",len(keyw),"Keywords. They are:"
for w in keyw:
        print(w)
print "There are total of",len(ide),"Identifiers. They are:"
for w in ide:
        print(w)
print "There are total of",len(num),"Numbers. They are:"
for w in num:
        print(w)
```

Explanation for the above code

In the above code , the list of keywords that are used in the program have been declared.

When a file containing the problem statement is given as input to the program(Questions.txt), the split function divides the input with space,semicolon and comma as delimiter and assigns it to a variable named token everytime in the loop.

Every time the token is compared with each keyword in the array and if it matches any of them, then it is displayed as a keyword.

If it matches with any of the operators, then its an operator.
If it matches with numbers, then its a number.
Else it is an identifier

Output:

```
<terminated> Token.py [C:\Python27\python.exe]
There are total of 5 Operators. They are:
=
=
+
+
=
There are total of 9 Keywords. They are:
int
main
begin
int
if
for
begin
end
end
There are total of 15 Identifiers. They are:
n1
n2
i
gcd
expr
relop
expr
gcd
i
i
expr
relop
expr
i
gcd
There are total of 2 Numbers. They are:
1
1
```

SYNTAX ANALYSIS

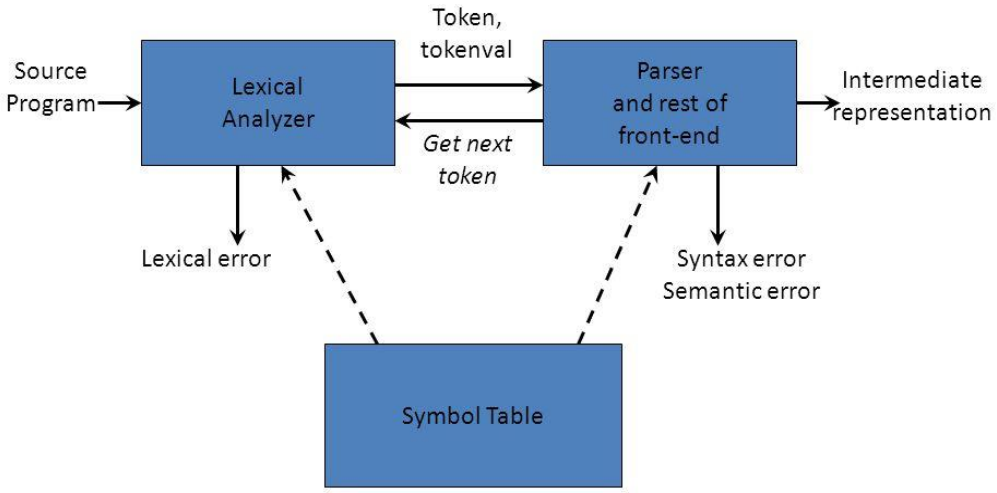
Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

Role of the Parser

In the compiler model, the parser obtains a string of tokens from the lexical analyser, and verifies that the string can be generated by the grammar for the source language.

The parser returns any syntax error for the source language.

Position of a Parser in the Compiler Model



3

. There are three general types' parsers for grammars.

· Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar. These methods are too inefficient to use in production compilers.

· The methods commonly used in compilers are classified as either top-down parsing or bottom-up parsing.

· Top-down parsers build parse trees from the top (root) to the bottom (leaves).

· Bottom-up parsers build parse trees from the leaves and work up to the root.

· In both case input to the parser is scanned from left to right, one symbol at a time.
. The output of the parser is some representation of the parse tree for the stream of tokens.
· There are number of tasks that might be conducted during parsing. Such as;

o Collecting information about various tokens into the symbol table.
o Performing type checking and other kinds of semantic analysis.
o Generating intermediate code.
o Syntax Error Handling:
o Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.
o The program can contain errors at many different levels. e.g.,

§ Lexical – such as misspelling an identifier, keyword, or operator.
§ Syntax – such as an arithmetic expression with unbalanced parenthesis.
§ Semantic – such as an operator applied to an incompatible operand.
§ Logical – such as an infinitely recursive call.

. Much of the error detection and recovery in a compiler is centered on the syntax analysis phase.
o One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyser disobeys the grammatical rules defining the programming language.
o Another is the precision of modern parsing methods; they can detect the presence of syntactic errors in programs very efficiently.

· The error handler in a parser has simple goals:
o It should the presence of errors clearly and accurately.
o It should recover from each error quickly enough to be able to detect subsequent errors.
• It should not significantly slow down the processing of correct programs.

Error-Recovery Strategies:

o There are many different general strategies that a parser can employ to recover from a syntactic error.
§ Panic mode
§ Phrase level
§ Error production
§ Global correction

CONTEXT-FREE GRAMMAR

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

• A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
• A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.
• A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.
• One of the non-terminals is designated as the start symbol (S); from where the production begins. The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

Parse Tree

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree.

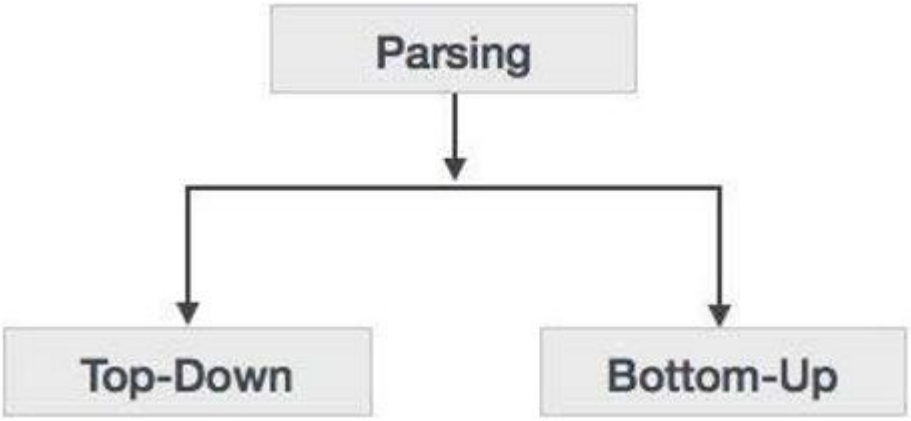
In a parse tree:

• All leaf nodes are terminals.
• All interior nodes are non-terminals.
• In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

Types Of Parser

Syntax analyzers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types : topdown parsing and bottom-up parsing.



Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

 Recursive descent parsing : It is a common form of top-down parsing. It is called recursive, as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
 Backtracking : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

NOTE:THE TYPE OF PARSER WE HAVE USED IS BOTTOM UP PARSER

Context free grammar for above problem statement is:

S -> K
K -> I M ( )
K -> B
I -> int
M -> main
B -> begin
K -> I V ;
V -> N , A , G
N -> n1 , n2
A -> i
G -> gcd
K -> F (C)
C -> E R E
E -> n1
E -> n2
E -> G
E -> A
F -> if
E -> expr
R -> ==
R -> !=
R -> >
R -> <
R -> >=
R -> <=
K -> G = A ;
K -> O (A = 1 ; C ; ++ A)
O -> for
K -> G = 1 ;
K -> D
D -> end

Augmented Grammar

0: A -> i
1: C -> E R E
2: B -> begin
3: E -> n1

4: E -> n2
5: E -> G
6: E -> A
7: E -> expr
8: D -> end
9: G -> gcd
10: F -> if
11: I -> int
12: K -> I M ( )
13: K -> B
14: K -> I V ;
15: K -> F (C)
16: K -> G = A ;
17: K -> O (A = 1 ; C ; ++ A)
18: K -> G = 1 ;
19: K -> D
20: M -> main
21: O -> for
22: N -> n1 , n2
23: S -> K
24: R -> ==
25: R -> !=
26: R -> >
27: R -> <
28: R -> >=
29: R -> <=
30: V -> N , A , G
31: S' -> S

First and Follow Sets

An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

First Set

This set is created to know what terminal symbol is derived in the first position by a nonterminal. For example,

$\alpha \rightarrow t\ \beta$

That is, α derives t (terminal) in the very first position. So, t ∈ FIRST(α).

Algorithm for Calculating First Set

Look at the definition of FIRST(α) set:

- if α is a terminal, then FIRST(α) = { α }.
- if α is a non -terminal and α → Ɛ is a production, then FIRST(α) = { Ɛ }.
- if α is a non -terminal and α → $\gamma 1\ \gamma 2\ \gamma 3 \ldots \gamma n$ and any FIRST(γ) contains t, then t is in FIRST(α).

First set can be seen as: FIRST(α) = { t | α →* t β } ∪ { Ɛ | α →* ε}

Follow Set

Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, we see what would be the next terminal symbol that follows the productions of a non-terminal.

Algorithm for Calculating Follow Set:

- if α is a start symbol, then FOLLOW() = $

- if α is a non -terminal and has a production α → AB, then FIRST(B) is in FOLLOW(A) except Ɛ.
- if α is a non -terminal and has a production α → AB, where B Ɛ, then FOLLOW(A) is in FOLLOW(α).

Follow set can be seen as: FOLLOW(α) = { t | S *αt*}

First and Follow for the above problem statement is:

FIRST:

FIRST(A) = { i }
FIRST(C) = { n1 , n2 , gcd , i , expr , == , != , > , < , >= , <= }
FIRST(B) = { begin }
FIRST(E) = { n1 , n2 , gcd , i , expr }
FIRST(D) = { end }
FIRST(G) = { gcd }
FIRST(F) = { if }
FIRST(I) = { int }
FIRST(K) = { int , main , (,) , begin , n1 , , , i , gcd , ; , if , n2 , expr , == , != , > , < , >= , <= , = , for , 1 , ++ , end }
FIRST(M) = { main }
FIRST(O) = { for }
FIRST(N) = { n1 }
FIRST(S) = { int , main , (,) , begin , n1 , , , i , gcd , ; , if , n2 , expr , == , != , > , < , >= , <= , = , for , 1 , ++ , end }
FIRST(R) = { == , != , > , < , >= , <= }
FIRST(V) = { n1 , , , i , gcd }
FIRST(S') = { int , main , (,) , begin , n1 , , , i , gcd , ; , if , n2 , expr , == , != , > , < , >= , <= , = , for , 1 , ++ , end }

FOLLOW:

FOLLOW(A) = { == , != , > , < , >= , <= , ; , = , , }
FOLLOW(C) = {) , ; }
FOLLOW(B) = { $ }
FOLLOW(E) = { == , != , > , < , >= , <= }
FOLLOW(D) = { $ }
FOLLOW(G) = { == , != , > , < , >= , <= , = , ; }
FOLLOW(F) = { (}
FOLLOW(I) = { main , n1 , , , i , gcd }
FOLLOW(K) = { $ }
FOLLOW(M) = { (}
FOLLOW(O) = { (}
FOLLOW(N) = { , }
FOLLOW(S) = { $ }
FOLLOW(R) = { n1 , n2 , gcd , i , expr }
FOLLOW(V) = { ; }
FOLLOW(S') = { $ }

PARSING TABLE

Building the complete table

- Need a row for every NT & a column for every T
- Need an algorithm to build the table

Filling in TABLE[X,y], X ∈ NT, y ∈ T

- entry is the rule X ::= β . if y ∈ FIRST+(β)
- entry is error otherwise (can treat empty entry as implicit error)

Parser code for the above problem statement is :

```
cfg = open("cfg.txt")
G = {}
C = {}
start = ""
terminals = []
nonterminals = []
symbols = []
error=0

def parsecfg():
    global G, start, terminals, nonterminals, symbols
    for line in cfg:
        line = " ".join(line.split())
        if line == '\n':
            break
        head = line[:line.index("->")].strip()
        prods = [l.strip().split(' ') for l in ''.join(line[line.index("->") + 2:]).split('|')]
        if not start:
            start = head + "'"
            G[start] = [[head]]
            nonterminals.append(start)
        if head not in G:
            G[head] = []
        if head not in nonterminals:
            nonterminals.append(head)
        for prod in prods:
            G[head].append(prod)
            for char in prod:
                if not char.isupper() and char not in terminals:
                    terminals.append(char)
                elif char.isupper() and char not in nonterminals:
                    nonterminals.append(char)
                    G[char] = []
    symbols =  nonterminals+terminals
first_seen = []

def FIRST(X):
    global first_seen
    first = []
    first_seen.append(X)
    if X in terminals:
        first.append(X)
    elif X in nonterminals:
        for prods in G[X]:
            if prods[0] in terminals and prods[0] not in first:
                first.append(prods[0])
            else:
```

```
                for nonterm in prods:
                    if nonterm not in first_seen:
                        for terms in FIRST(nonterm):
                            if terms not in first:
                                first.append(terms)
    first_seen.remove(X)
    return first


follow_seen = []
def FOLLOW(A):
    global follow_seen
    follow = []
    follow_seen.append(A)
    if A == start:
        follow.append('$')
    for heads in G.keys():
        for prods in G[heads]:
            follow_head = False
            if A in prods:
                next_symbol_pos = prods.index(A) + 1
                if next_symbol_pos < len(prods):
                    for terms in FIRST(prods[next_symbol_pos]):
                        if terms not in follow:
                            follow.append(terms)
                else:
                    follow_head = True
                if follow_head and heads not in follow_seen:
                    for terms in FOLLOW(heads):
                        if terms not in follow:
                            follow.append(terms)
    follow_seen.remove(A)
    return follow

def closure(I):
    J = I
    while True:
        item_len = len(J) + sum(len(v) for v in J.itervalues())
        for heads in J.keys():
            for prods in J[heads]:
                dot_pos = prods.index('.')
                if dot_pos + 1 < len(prods):
                    prod_after_dot = prods[dot_pos + 1]
                    if prod_after_dot in nonterminals:
                        for prod in G[prod_after_dot]:
                            item = ["."] + prod
                            if prod_after_dot not in J.keys():
                                J[prod_after_dot] = [item]
                            elif item not in J[prod_after_dot]:
                                J[prod_after_dot].append(item)
        if item_len == len(J) + sum(len(v) for v in J.itervalues()):
            return J

def GOTO(I, X):
    goto = {}
    for heads in I.keys():
```

```
        for prods in I[heads]:
            for i in range(len(prods) - 1):
                if "." == prods[i] and X == prods[i + 1]:
                    temp_prods = prods[:]
                    temp_prods[i], temp_prods[i + 1] = temp_prods[i + 1], temp_prods[i]
                    prod_closure = closure({heads: [temp_prods]})
                    for keys in prod_closure:
                        if keys not in goto.keys():
                            goto[keys] = prod_closure[keys]
                        elif prod_closure[keys] not in goto[keys]:
                            for prod in prod_closure[keys]:
                                goto[keys].append(prod)
    return goto

def items():
    global C
    i = 1
    C = {'I0': closure({start: [['.'] + G[start][0]]})}
    while True:
        item_len = len(C) + sum(len(v) for v in C.itervalues())
        for I in C.keys():
            for X in symbols:
                if GOTO(C[I], X) and GOTO(C[I], X) not in C.values():
                    C['I' + str(i)] = GOTO(C[I], X)
                    i += 1
        if item_len == len(C) + sum(len(v) for v in C.itervalues()):
            return


def ACTION(i, a):
    global error
    for heads in C['I' + str(i)]:
        for prods in C['I' + str(i)][heads]:
            for j in range(len(prods) - 1):
                if prods[j] == '.' and prods[j + 1] == a:
                    for k in range(len(C)):
                        if GOTO(C['I' + str(i)], a) == C['I' + str(k)]:
                            if a in terminals:
                                if "r" in parse_table[i][terminals.index(a)]:
                                    if error!=1:
                                        print "ERROR: Shift-Reduce Conflict at State " + str(i)
+ ", Symbol \'" + str(terminals.index(a))+"\'"
                                    error=1
                                    if "s"+str(k) not in parse_table[i][terminals.index(a)]:
                                        parse_table[i][terminals.index(a)] =
parse_table[i][terminals.index(a)]+ "/s" + str(k)
                                    return parse_table[i][terminals.index(a)]
                                else:
                                    parse_table[i][terminals.index(a)] = "s" + str(k)
                            else:
                                parse_table[i][len(terminals) + nonterminals.index(a)] = str(k)
                            return "s" + str(k)
    for heads in C['I' + str(i)]:
        if heads != start:
            for prods in C['I' + str(i)][heads]:
                if prods[-1] == '.':
```

```
                    k = 0
                    for head in G.keys():
                        for Gprods in G[head]:
                            if head == heads and (Gprods == prods[:-1] ) and (a in terminals or
a == '$'):
                                for terms in FOLLOW(heads):
                                    if terms == '$':
                                        index = len(terminals)
                                    else:
                                        index = terminals.index(terms)
                                    if "s" in parse_table[i][index]:
                                        if error!=1:
                                            print "ERROR: Shift-Reduce Conflict at State " +
str(i) + ", Symbol \'" + str(terms)+"\'"
                                        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(terms)+"\'"
                                        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 C['I' + str(i)] and G[start][0] + ['.'] in C['I' + str(i)][start]:
        parse_table[i][len(terminals)] = "acc"
        return "acc"
    return ""


def process_input():
    get_input = raw_input("\nEnter Input: ")
    to_parse = " ".join((get_input + " $").split()).split(" ")
    pointer = 0
    stack = ['0']

    print "\n+--------+---------------------------+---------------------------+-------------
--------------+"
    print "|{:^8}|{:^28}|{:^28}|{:^28}|".format("STEP", "STACK", "INPUT", "ACTION")
    print "+--------+---------------------------+---------------------------+---------------
------------+"

    step = 1
    while True:
        curr_symbol = to_parse[pointer]
        top_stack = int(stack[-1])
        stack_content = ""
```

```
        input_content = ""

        print "|{:^8}|".format(step),
        for i in stack:
            stack_content += i
        print "{:27}|".format(stack_content),
        i = pointer
        while i < len(to_parse):
            input_content += to_parse[i]
            i += 1
        print "{:>26} | ".format(input_content),

        step += 1
        get_action = ACTION(top_stack, curr_symbol)
        if "/" in get_action:
            print "{:^26}|".format(get_action+". So conflict")
            break
        if "s" in get_action:
            print "{:^26}|".format(get_action)
            stack.append(curr_symbol)
            stack.append(get_action[1:])
            pointer += 1
        elif "r" in get_action:
            print "{:^26}|".format(get_action)
            i = 0
            for head in G.keys():
                for prods in G[head]:
                    if i == int(get_action[1:]):
                        for j in range(2 * len(prods)):
                            stack.pop()
                        state = stack[-1]
                        stack.append(head)
                        stack.append(parse_table[int(state)][len(terminals) +
nonterminals.index(head)])
                    i += 1
        elif get_action == "acc":
            print "{:^26}|".format("ACCEPTED")
            break
        else:
            print "ERROR: Unrecognized symbol", curr_symbol, "|"
            break
    print "+--------+---------------------------+---------------------------+----------------
------------+"

def main():
    parsecfg()
    items()
    global parse_table
    parse_table = [["" for c in range(len(terminals) + len(nonterminals) + 1)] for r in
range(len(C))]
    for i in range(len(parse_table)):
        for j in symbols:
            ACTION(i, j)
    print "PARSING TABLE:"
    print "+" + "--------+" * (len(terminals) + len(nonterminals) + 1)
    print "|{:^8}|".format('STATE'),
```

```
    for terms in terminals:
        print "{:^7}|".format(terms),
    print "{:^7}|".format("$"),
    for nonterms in nonterminals:
        if nonterms == start:
            continue
        print "{:^7}|".format(nonterms),
    print "\n+" + "--------+" * (len(terminals) + len(nonterminals) + 1)
    for i in range(len(parse_table)):
        print "|{:^8}|".format(i),
        for j in range(len(parse_table[i]) - 1):
            print "{:^7}|".format(parse_table[i][j]),
        print
    print "+" + "--------+" * (len(terminals) + len(nonterminals) + 1)
    ch=1
    while(ch):
        process_input()
        choice=raw_input("Do you wish to continue?Press Yes or No: ")
        if(choice=="No" or choice=="NO" or choice=="no" or choice=="nO"):
            ch=0
if __name__ == '__main__':
      main()
```

The context free grammar for the parser in present in the file cfg.txt. Methods first() and follow() find first and follow of the non-terminals in the grammar. Main() method calls parsecfg() which parses the CFG and produces a parsing table. The program takes the input and checks whether it's syntactically correct according to the question.

RESULT:

```
Enter Input: int n1 , n2 , i , gcd ;
```

STEP	STACK	INPUT	ACTION
1	0	intn1,n2,i,gcd;$	s9
2	0int9	n1,n2,i,gcd;$	r11
3	0I3	n1,n2,i,gcd;$	s19
4	0I3n119	,n2,i,gcd;$	s34
5	0I3n119,34	n2,i,gcd;$	s51
6	0I3n119,34n251	,i,gcd;$	r22
7	0I3N17	,i,gcd;$	s36
8	0I3N17,36	i,gcd;$	s24
9	0I3N17,36i24	,gcd;$	r0
10	0I3N17,36A42	,gcd;$	s52
11	0I3N17,36A42,52	gcd;$	s11
12	0I3N17,36A42,52gcd11	;$	r9
13	0I3N17,36A42,52G55	;$	r30
14	0I3V16	;$	s37
15	0I3V16;37	$	r14
16	0K2	$	r23
17	0S1	$	ACCEPTED

```
Do you wish to continue?Press Yes or No:
```

FOR SYNTAX ERROR CONDITION:

```
Enter Input: int n1 , g ;
```

STEP	STACK	INPUT	ACTION
1	0	intn1,g;$	s9
2	0int9	n1,g;$	r11
3	0I3	n1,g;$	s19
4	0I3n119	,g;$	s34
5	0I3n119,34	g;$	ERROR: Unrecognized symbol g

```
Do you wish to continue?Press Yes or No:
```

CONCLUSION

In lexical analysis when we give a program statement as input ,the keywords ,identifiers ,operators and numbers are displayed. Each of these are the tokens of the statement.

In the bottom up parsing, the parsing starts with the input symbol and construct the parse tree up to the start symbol by tracing out the rightmost derivations of string in reverse. If the parsing