

COMPILER DESIGN

A MINI PROJECT REPORT SUBMITTED BY:

Shashank V Jogi

(4NM18CS166)

Shreyas K Shetty

(4NM18CS181)

UNDER THE GUIDANCE OF:

Mrs. Pallavi K N

Assistant Professor Gd II

Department of Computer science and Engineering

In partial fulfilment of the requirement for the award of the Degree of

BACHELOR OF ENGINEERING IN COMPUTER SCIENCE AND ENGINEERING
From

Visvesvaraya Technological University, Belagavi



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

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
N.M.A.M. INSTITUTION OF TECHNOLOGY**



(An Autonomous Institution affiliated to VTU, Belagavi)
(NBA Accredited, ISO 9001:2008 Certified)
Nitte-574110, Karkala, Udupi District, Karnataka, India

Department of Computer Science and Engineering
CERTIFICATE

Certified that the Mini Project work entitled

COMPILER DESIGN

Is a Bonafede work carried out by

Shashank V Jogi(4NM18CS166)

Shreyas K Shetty(4NM18CS181)

In partial fulfilment of requirements for the award of Bachelor of Engineering Degree in
Computer Science and Engineering prescribed by Visvesvaraya Technology University,
Belgaum during the year 2020-2021

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

Name & Signature of Guide(s)

Mrs. Pallavi K N
Assistant Professor, Gd II
Department of CSE

Name & Signature of HOD

Dr. Jyothi Shetty
Head of the Department
Department of CSE

ACKNOWLEDGMENT

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 **Mrs. Pallavi K N**, Assistant Professor GD II, Department of Computer Science and Engineering, for her inspiring guidance, constant encouragement, support and suggestion for improvement during the course of our project.

We sincerely thank **Dr Jyothi Shetty**, Head of Department of Computer Science and Engineering, Nitte Maralinga 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.

Shashank V Jogi(4NM18CS166)

Shreyas K Shetty(4NM18CS181)

TABLE OF CONTENTS

Sl. No.	Contents	Page No.
1.	Abstract	1
2.	Introduction <ul style="list-style-type: none">• Compiler• Phases of Compiler• Classification of compiler Phases	2 - 5
3.	Implementation	6 – 8
4.	Screenshots	9 – 15
5.	Result & Conclusion	16

1. ABSTRACT

Programming languages are notations for describing computations to people and to machines. The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language. But, before a program can be run, it first must be translated into a form in which it can be executed by a computer. The software systems that do this translation are called compilers.

This report contains the details of how one can develop the simple compiler for given language using Lex (Lexical Analyzer Generator) and YACC (Yet Another Compiler). Lex tool helps write programs whose control flow is directed by instances of regular expressions in the input stream.

Lex tool source is the table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. On the other hand, YACC tool receives input of the user grammar. Starting from this grammar it generates the C source code for the parser. YACC invokes Lex to scan the source code and uses the tokens returned by Lex to build a syntax tree. With the help of YACC and Lex tool one can write their own compiler.

2.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.

Source program: It is normally written in a high-level programming language. It contains a set of rules, symbols and special words used to construct a computer program.

Target program: It is normally the equivalent program in machine code. It contains the binary representation of the instructions that the hardware of computer can perform.

Error Message: A message issued by the compiler due to detection of syntax errors in the source program.

Compilation is a large process. It is often broken into stages. Many phases of the compiler try and optimize by translating one form into a better (more efficient) form. Most of compiling is about “pattern matching” languages and tools that support pattern matching, are very useful. An efficient compiler must preserve semantics of the source program and it should create an efficient version of the target language.

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:

- Lexical Analyser
- Syntax Analyser
- Semantic Analyser

In synthesis phase, the equivalent target program is created from this intermediate representation. This contains:

- Intermediate code Generator
- Code Optimisation
- Code Generation

1. LEXICAL ANALYZER:

Lexical analyser 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:

2. 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 analyser (list of tokens) and produces a parse tree. A syntax analyser is also called as a parser. The parser checks if the expression made by the tokens is syntactically correct.

3. SEMANTIC ANALYSER:

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

4. INTERMEDIATE CODE GENERATION:

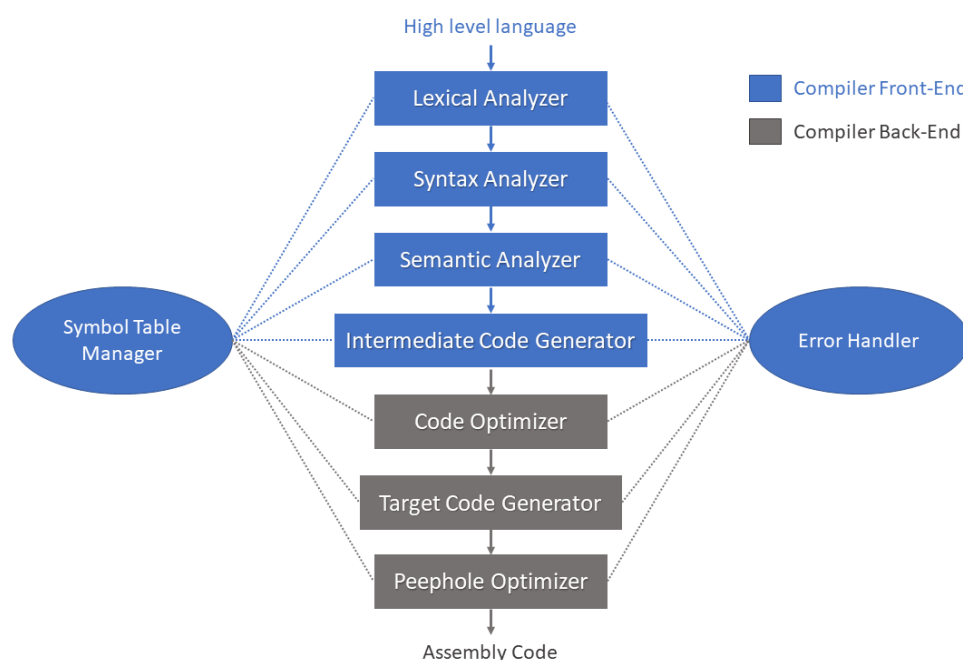
After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language.

5. 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.

6. 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.

3. IMPLEMENTATION

PROBLEM STATEMENT:

```
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
end
```

PHASE -I

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 analyser breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. If the lexical analyser finds a token invalid, it generates an error. The lexical analyser works closely with the syntax analyser. It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyser when it demands. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase.

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.

```
['t','m','(','')','b','t','v',' ',' ','o','v',' ',' ','v',' ',' ','l','(','v','')','b','v','o','v','o','v',' ',' ','d','d']
```

```
sudesh@sudesh-HP-Notebook:~$ lex cdp.l
sudesh@sudesh-HP-Notebook:~$ cc lex.yy.c
sudesh@sudesh-HP-Notebook:~$ ./a.out code.txt
```

STRING	TOKEN
int	==> keyword
main	==> keyword
(==> special symbols
begin	==> keyword
int	==> keyword
n1	==> identifier
,	==> special symbols
n2	==> identifier
,	==> special symbols
i	==> identifier
,	==> special symbols
gcd	==> identifier
;	==> special symbols
if	==> keyword
(==> special symbols
expr	==> identifier
relop	==> keyword
expr	==> identifier
)	==> special symbols
gcd	==> identifier
=	==> operators
i	==> identifier
;	==> special symbols
for	==> keyword
(==> special symbols
i	==> identifier
=	==> operators
1	==> constant
;	==> special symbols
expr	==> identifier
relop	==> keyword
expr	==> identifier
,	==> special symbols
n2	==> identifier
,	==> special symbols
i	==> identifier
,	==> special symbols
gcd	==> identifier
;	==> special symbols
if	==> keyword
(==> special symbols
expr	==> identifier
relop	==> keyword
expr	==> identifier
)	==> special symbols
gcd	==> identifier
=	==> operators
i	==> identifier
;	==> special symbols
for	==> keyword
(==> special symbols
i	==> identifier
=	==> operators
1	==> constant
;	==> special symbols
expr	==> identifier
relop	==> keyword
expr	==> identifier
;	==> special symbols
++	==> operators
i	==> identifier
)	==> special symbols
begin	==> keyword
gcd	==> identifier
=	==> operators
1	==> constant
;	==> special symbols
end	==> keyword
end	==> keyword

SYNTAX ANALYSIS:

In our compiler model, the parser obtains a string of tokens from the lexical analyser, as shown in the figure below, and verifies that the string of token names can be generated by the grammar for the source language. We expect the parser to report any syntax errors in an intelligible fashion and to recover from commonly occurring errors to continue processing the remainder of the program. Conceptually, for well-formed programs, the parser constructs a parse tree and passes it to the rest of the compiler for further processing

Parser:

Parser is that phase of compiler which takes token string as input and with the help of existing grammar, converts it into the corresponding parse tree. Parser is also known as Syntax Analyzer. The parser obtains a string of tokens from the lexical analyser and verifies that the string can be the grammar for the source language. It detects and reports any syntax errors and produces a parse tree from which intermediate code can be generated.

4. SCREEN SHOTS

Code Screen Shots:

The image displays two screenshots of a Jupyter Notebook interface, likely from a web browser. The interface includes a top menu bar with options like File, Edit, View, Insert, Runtime, Tools, and Help. Below the menu is a toolbar with icons for saving, running, and other actions. The main area is divided into three panes: a file explorer on the left, a code editor in the center, and a text editor on the right.

Top Screenshot:

- File Explorer:** Shows a directory structure with files like `sample_data`, `grammar.txt`, `lexer(1).py`, `lexer.py`, `newf(1).txt`, and `newf.txt`.
- Code Editor:** Contains Python code for a grammar parser. It includes imports for `pandas` and `numpy`, a function `get_tokens` to read from `newf.txt`, a function `get_productions` to parse grammar rules, and a function `closure` to calculate the closure of a set of items.
- Text Editor:** Shows a snippet of C code for a main function that initializes variables and calls the `get_productions` function.

Bottom Screenshot:

- Code Editor:** Continues the Python code from the top screenshot, showing the `closure` function implementation and a `get_symbols` function to extract terminals and non-terminals from the grammar.
- Text Editor:** Shows the same C code snippet as in the top screenshot.
- Status Bar:** At the bottom, it indicates "0s completed at 11:53 AM".

CDproject.ipynb

```

final_set = []
for X in symbols:
    first_set = [] # Will contain the first(X)
    if isTerminal(X):
        final_set.extend(X)
        return final_set
    else:
        for production in grammar:
            # For each production in the grammar
            lhs, rhs = production.split('->')
            if lhs == X:
                # Check if the LHS is 'X'
                for i in range(len(rhs)):
                    # To find the first of the RHS
                    y = rhs[i]
                    # Check one symbol at a time
                    if y == X:
                        # Ignore if it's the same symbol as X
                        # This avoids infinite recursion
                        continue
                    first_y = first(y)
                    first_set.extend(first_y)
                    # Check next symbol only if first(current) contains EPSILON
                    if EPSILON in first_y:
                        first_y.remove(EPSILON)
                        continue
                else:
                    # No EPSILON. Move to next production
                    continue
            # No EPSILON. Move to next production
            break
        return first_set

def isTerminal(symbol):
    # This function will return if the symbol is a terminal or not
    return symbol in terminals

def shift_dot(production):
    # This function shifts the dot to the right
    lhs, rhs = production.split('->')
    x, y = rhs.split(".")
    if len(y) == 0:
        return None
    else:
        return lhs + "." + y

```

newf.txt

```

1 int main()
2 begin
3 int n1, n2, i, gcd;
4 if(expr relop expr)
5 gcd = i;
6 for(i=1; expr relop expr; ++i)
7 begin
8 gcd=i;
9 end
10 end

```

CDproject.ipynb

```

else:
    # No EPSILON. Move to next production
    break

else:
    # All symbols contain EPSILON. Add EPSILON to first(X)
    # Check to see if some previous production has added epsilon al
    if EPSILON not in first_set:
        first_set.extend(EPSILON)

    # Move onto next production
    final_set.extend(first_set)
    if EPSILON in first_set:
        continue
    else:
        break
return final_set

def isTerminal(symbol):
    # This function will return if the symbol is a terminal or not
    return symbol in terminals

def shift_dot(production):
    # This function shifts the dot to the right
    lhs, rhs = production.split('->')
    x, y = rhs.split(".")
    if len(y) == 0:
        return None
    else:
        return lhs + "." + y

```

newf.txt

```

1 int main()
2 begin
3 int n1, n2, i, gcd;
4 if(expr relop expr)
5 gcd = i;
6 for(i=1; expr relop expr; ++i)
7 begin
8 gcd=i;
9 end
10 end

```

CDproject.ipynb

```

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

def goto(I, X):
    # Function to calculate GOTO
    J = []
    for production, look_ahead in I:
        lhs, rhs = production.split('->')
        # Find the productions with .X
        if "." + X in rhs and not rhs[-1] == '.':
            # Check if the production ends with a dot, else shift dot
            new_prod = shift_dot(production)
            J.append((new_prod, look_ahead))
    return closure(J)

def set_of_items(display=False):
    # Function to construct the set of items
    num_states = 1
    items = {}
    for X in symbols:
        if X != EPSILON:
            first_set = []
            if isTerminal(X):
                first_set.extend(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)
                            first_set.extend(first_y)
                            if EPSILON in first_y:
                                first_y.remove(EPSILON)
                                continue
                        else:
                            continue
                    break
                return first_set
            final_set = []
            for X in symbols:
                first_set = []
                if isTerminal(X):
                    final_set.extend(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)
                                first_set.extend(first_y)
                                if EPSILON in first_y:
                                    first_y.remove(EPSILON)
                                    continue
                            else:
                                continue
                        break
                    return first_set
            final_set.extend(first_set)
            if EPSILON in first_set:
                continue
            else:
                break
            return final_set
        else:
            first_set = []
            for production in grammar:
                lhs, rhs = production.split('->')
                if lhs == EPSILON:
                    first_set.extend(rhs)
            return first_set
    return final_set

```

newf.txt

```

1 int main()
2 begin
3 int n1, n2, i, gcd;
4 if(expr relop expr)
5 gcd = i;
6 for(i=1; expr relop expr; ++i)
7 begin
8 gcd=i;
9 end
10 end

```

CDproject.ipynb

```

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 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

def pending_shifts(I):
    # This function will check which symbols are to be shifted in I
    symbols = [] # Will contain the symbols in order of evaluation
    for production, _ in I:
        lhs, rhs = production.split('->')
        if rhs.endswith('.'):
            # dot is at the end of production. Hence, ignore it

```

newf.txt

```

1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr relop expr)
5     gcd = i;
6   for(i=1; expr relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end

```

Disk 66.03 GB available

completed at 11:53 AM

CDproject.ipynb

```

if rhs.endswith('.'):
    # dot is at the end of production. Hence, ignore it
    continue
# beta is the first symbol after the dot
beta = rhs.split('.')[1][0]
if beta not in symbols:
    symbols.append(beta)
return symbols

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):
    # This function returns the State name, given a set of items.
    key_list = list(C.keys())
    val_list = list(C.values())
    i = val_list.index(I)
    return key_list[i]

def CLR_construction(num_states):
    # Function that returns the CLR Parsing Table function ACTION and GOTO

```

newf.txt

```

1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr relop expr)
5     gcd = i;
6   for(i=1; expr relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end

```

Disk 66.03 GB available

completed at 11:53 AM

CDproject.ipynb

```

def CLR_construction(num_states):
    # Function that returns the CLR Parsing Table function ACTION and GOTO
    C = set_of_items() # Construct collection of sets of LR(1) items

    # Initialize two tables for ACTION and GOTO respectively
    ACTION = pd.DataFrame(columns=terminals, index=range(num_states))
    GOTO = pd.DataFrame(columns=non_terminals, index=range(num_states))

    for Ii in C.values():
        # For each state in the collection
        i = int(get_state(C, Ii)[1:])
        pending = pending_shifts(Ii)
        for a in pending:
            # For each symbol 'a' after the dots
            Ij = goto(Ii, a)
            j = int(get_state(C, Ij)[1:])
            if isTerminal(a):
                # Construct the ACTION function
                ACTION.at[i, a] = "Shift "+str(j)
            else:
                # Construct the GOTO function
                GOTO.at[i, a] = j

    # For each production with dot at the end
    for production, look_ahead in done_shifts(Ii):
        # Set GOTO[i, a] to "Reduce"
        ACTION.at[i, look_ahead] = "Reduce " + str(grammar.index(production)+1)

```

newf.txt

```

1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr relop expr)
5     gcd = i;
6   for(i=1; expr relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end

```

Disk 66.03 GB available

completed at 11:53 AM

CDproject.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Files

- sample_data
- grammar.txt
- lexer(1).py
- lexer.py
- newf(1).txt
- newf.txt

```
for production, look_ahead in done_shifts(I1):
    # Set GOTO[I, a] to "Reduce"
    ACTION.at[i, look_ahead] = "Reduce " + str(grammar.index(production)+1)

    # If start production is in I1
    if ('P->S.', '$') in I1:
        ACTION.at[i, '$'] = "Accept"

    # Remove the default NaN values to make it clean
    ACTION.replace(np.nan, '', regex=True, inplace=True)
    GOTO.replace(np.nan, '', regex=True, inplace=True)

    return ACTION, GOTO

def parse_string(string, ACTION, GOTO):
    # This function parses the input string and returns the table
    row = 0
    # Parse table column names:
    cols = ['Stack', 'Input', 'Output']
    if not string.endswith('$'):
        # Append $ if not already appended
        string = string+'$'
    ip = 0 # Initialize input pointer
    # Create an initial (empty) parsing table:
    PARSE = pd.DataFrame(columns=cols)
    # Initialize input stack:
    input = list(string)
```

newf.txt

```
1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr_relop expr)
5     gcd = i;
6   for(i=1; expr_relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end
```

0s completed at 11:53 AM

CDproject.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Files

- sample_data
- grammar.txt
- lexer(1).py
- lexer.py
- newf(1).txt
- newf.txt

```
# initialize input stack:
input = list(string)
# Initialize grammar stack:
stack = ['$ ', '0']
while True:
    S = int(stack[-1]) # Stack top
    a = input[ip] # Current input symbol
    action = ACTION.at[S, a]
    # New row to be added to the table:
    new_row = ["", "".join(stack), "", "".join(input[ip:]), action]
    if 'S' in action:
        # If it is a shift operation:
        S1 = action.split()[1]
        stack.append(a)
        stack.append(S1)
        ip += 1
    elif "R" in action:
        # If it's a reduce operation:
        i = int(action.split()[1])-1
        A, beta = grammar[i].split('->')
        for _ in range(2*len(beta)):
            # Remove 2 * rhs of the production
            stack.pop()
        S1 = int(stack[-1])
        stack.append(A)
        stack.append(str(GOTO.at[S1, A]))
        # Replace the number with the production for clarity:
        new_row[-1] = "Reduce "+grammar[i]
```

newf.txt

```
1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr_relop expr)
5     gcd = i;
6   for(i=1; expr_relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end
```

0s completed at 11:53 AM

CDproject.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Files

- sample_data
- grammar.txt
- lexer(1).py
- lexer.py
- newf(1).txt
- newf.txt

```
for production in F:
    grammar.append(production[:-1])
return grammar

if __name__ == "__main__":

    # Demo grammars
    # grammar = ['S->S+T', 'S->T', 'T->T*F', 'T->F', 'F->(S)', 'F->1']
    # grammar = ['S->CC', 'C->C', 'C->d']
    # grammar = ['S->L-R', 'S->R', 'L->*R', 'L->1', 'R->L']
    grammar = get_grammar("grammar.txt")
    # grammar = ['S->AB', 'A->aB', 'S->A', 'A->B', 'B->C', 'C->d']
    terminals, non_terminals = get_symbols(grammar)
    symbols = terminals.union(non_terminals)

    # Demonstrating main functions
    start = [('P->S.', '$')]
    I0 = closure(start)
    goto(I0, '')
    C = set_of_items(display=True)
    ACTION, GOTO = CLR_construction(num_states=len(C))

    # Demonstrating helper functions:
    get_productions('L')
    shift_dot('L->*R')
    pending_shifts(I0)
```

newf.txt

```
1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr_relop expr)
5     gcd = i;
6   for(i=1; expr_relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end
```

0s completed at 11:53 AM

CDproject.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Files

- sample_data
- grammar.txt
- lexer(1).py
- lexer.py
- newf(1).txt
- newf.txt

State 10 :
('P->.S', '\$')
('S->.ATIGFcd', '\$')
('A->.tm()'b', 't')

State 11 :
('P->S.', '\$')

State 12 :
('S->A.TIGFcd', '\$')
('T->.tL;', 'f')

State 13 :
('A->t.m()'b', 't')

State 14 :
('S->AT.IGFcd', '\$')
('I->.f(0)', 'v')

State 15 :
('T->t.L;', 'f')
('L->.v,L', ';')
('L->.v', ';')

State 16 :
('A->tm()'b', 't')

State 17 :

newf.txt

```
1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr relop expr)
5     gcd = 1;
6   for(i=1; expr relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end
```

0s completed at 11:53 AM

CDproject.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Files

- sample_data
- grammar.txt
- lexer(1).py
- lexer.py
- newf(1).txt
- newf.txt

State 17 :
('S->ATI.GFcd', '\$')
('G->.ZoX;', 'l')
('Z->.v', 'o')

State 18 :
('I->f.(0)', 'v')

State 19 :
('T->tL;', 'f')

State 110 :
('L->v.,L', ';')
('L->v.', ';')

State 111 :
('A->tm()'b', 't')

State 112 :
('S->ATIG.Fcd', '\$')
('F->.l(GD;M)', 'b')

State 113 :
('G->Z.oX;', 'l')

State 114 :
('Z->v.', 'o')

State 115 :
('I->f(.D)', 'v')
('D->.eRe', ';')

newf.txt

```
1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr relop expr)
5     gcd = 1;
6   for(i=1; expr relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end
```

0s completed at 11:53 AM

CDproject.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Files

- sample_data
- grammar.txt
- lexer(1).py
- lexer.py
- newf(1).txt
- newf.txt

State 122 :
('I->f(D.)', 'v')

State 123 :
('D->e.Re', ';')
('R->.r', 'e')

State 124 :
('L->v.,L', ';')

State 125 :
('A->tm()'b', 't')

State 126 :
('S->ATIGFC.d', '\$')

State 127 :
('C->b.Gd', 'd')
('G->.ZoX;', 'd')
('Z->.v', 'o')

State 128 :

newf.txt

```
1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr relop expr)
5     gcd = 1;
6   for(i=1; expr relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end
```

0s completed at 11:53 AM

CDproject.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Files

- sample_data
- grammar.txt
- lexer.py
- newf.txt

State I28 :
('F->l(GD;N)', 'b')
('G->.ZoX;', 'e')
('Z->.v', 'o')

State I29 :
('G->.ZoX;', 'l')

State I30 :
('X->.v', ';')

State I31 :
('X->.h', ';')

State I32 :
('I->.f(D).', 'v')

State I33 :
('D->.eR.e', 'v')

State I34 :
('R->.r', 'e')

State I35 :
('S->.ATIGFcD.', '\$')

State I36 :
('C->.bG.d', 'd')

State I37 :

newf.txt

```
1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr relop expr)
5     gcd = i;
6   for(i=1; expr relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end
```

0s completed at 11:53 AM

CDproject.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Files

- sample_data
- grammar.txt
- lexer.py
- newf.txt

State I37 :
('G->.ZoX;', 'd')

State I38 :
('F->l(GD;N)', 'b')
('D->.eRe', ';')

State I39 :
('G->.ZoX;', 'e')

State I40 :
('G->.ZoX;', 'l')

State I41 :
('D->.eRe.', 'v')

State I42 :
('C->.bGd.', 'd')

State I43 :
('G->.ZoX;', 'd')
('X->.v', ';')
('X->.h', ';')

State I44 :
('F->l(GD;N)', 'b')

State I45 :
('D->.eRe', ';')
('R->.r', 'e')

newf.txt

```
1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr relop expr)
5     gcd = i;
6   for(i=1; expr relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end
```

0s completed at 11:53 AM

CDproject.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Files

- sample_data
- grammar.txt
- lexer.py
- newf.txt

State I49 :
('D->.eR.e', ';')

State I50 :
('G->.ZoX;', 'e')

State I51 :
('G->.ZoX;', 'd')

State I52 :
('F->l(GD;N)', 'b')

State I53 :
('N->.o.ov', 'v')

State I54 :
('D->.eRe.', 'v')

State I55 :
('G->.ZoX;', 'e')

State I56 :
('F->l(GD;N).', 'b')

State I57 :
('N->.o.ov', 'v')

State I58 :
('N->.o.ov.', 'v')

newf.txt

```
1 int main()
2 begin
3   int n1, n2, i, gcd;
4   if(expr relop expr)
5     gcd = i;
6   for(i=1; expr relop expr; ++i)
7     begin
8       gcd=1;
9     end
10 end
```

0s completed at 11:53 AM

Input.txt

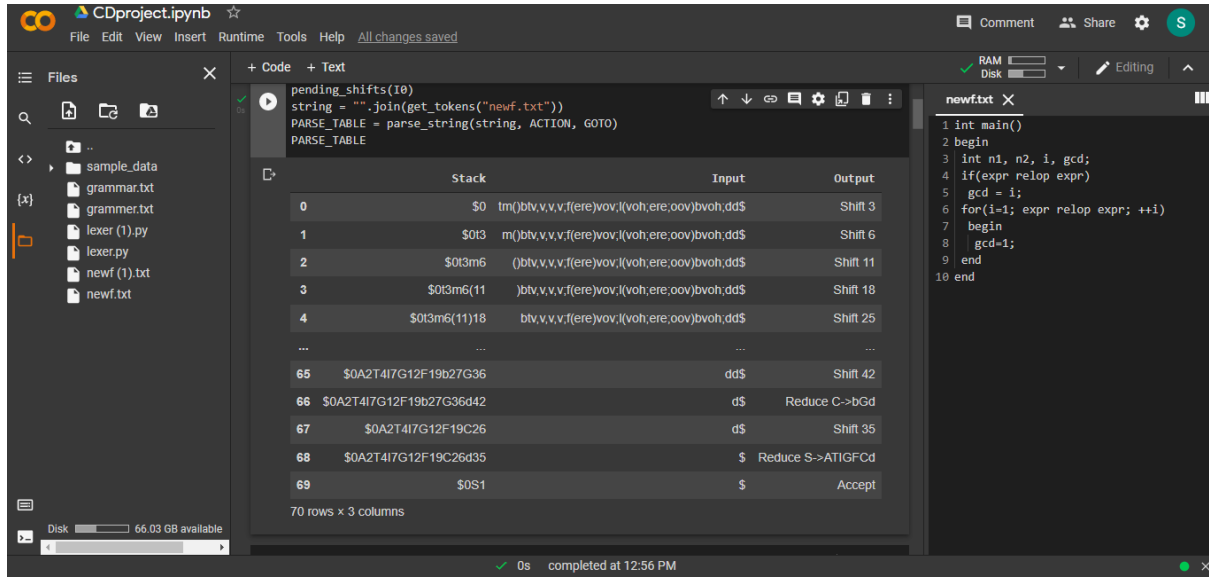
```
int main ()
begin
    int n1, n2, i, gcd;
    if (expr relop expr)
        gcd = i;
    for (i=1; expr relop expr; ++i)
        begin
            gcd=1;
        end
    end
end
```

grammer.txt

```
S->tm()B
B->tv,N
N->v,M
M->v;D
D->f(F)Q
F->EoE
E->v
Q->bp(v);L
L->dD
D->d
```

5. Result and Conclusion

String Parsing:



The screenshot displays a Jupyter Notebook environment with the following components:

- Files Panel (Left):** Shows a file explorer with directories like `sample_data` and files including `grammar.txt`, `lexer(1).py`, `lexer.py`, `newf(1).txt`, and `newf.txt`.
- Code Cell:** Contains the following Python code:

```
pending_shifts(I0)
string = "".join(get_tokens("newf.txt"))
PARSE_TABLE = parse_string(string, ACTION, GOTO)
PARSE_TABLE
```
- Output Table:** A table with 3 columns: `Stack`, `Input`, and `Output`. It shows the state of the parser at various steps (0 to 69).

	Stack	Input	Output
0	\$0	m()btv,v,v,f(ere)vov,(voh,ere,oov)bvoh,dd\$	Shift 3
1	\$0i3	m()btv,v,v,f(ere)vov,(voh,ere,oov)bvoh,dd\$	Shift 6
2	\$0i3m6	()btv,v,v,f(ere)vov,(voh,ere,oov)bvoh,dd\$	Shift 11
3	\$0i3m6(11)btv,v,v,f(ere)vov,(voh,ere,oov)bvoh,dd\$	Shift 18
4	\$0i3m6(11)18	btv,v,v,f(ere)vov,(voh,ere,oov)bvoh,dd\$	Shift 25
...
65	\$0A2T4I7G12F19b27G36	dd\$	Shift 42
66	\$0A2T4I7G12F19b27G36d42	d\$	Reduce C->bGd
67	\$0A2T4I7G12F19C26	d\$	Shift 35
68	\$0A2T4I7G12F19C26d35	\$	Reduce S->ATIGFCd
69	\$0S1	\$	Accept
- newf.txt File:** Contains the following C code:

```
1 int main()
2 begin
3 int n1, n2, i, gcd;
4 if(expr relop expr)
5 gcd = i;
6 for(i=1; expr relop expr; ++i)
7 begin
8 gcd=i;
9 end
10 end
```
- Status Bar:** Indicates "0s completed at 12:56 PM".