

## **EXPERIMENT 1**

**Aim:** Implementation of Single pass Assembler

### **Theory:**

A single-pass assembler is a type of assembler that performs only one pass or iteration through the source code to generate the object code. Unlike a two-pass assembler that requires two passes through the source code, a single-pass assembler processes the source code linearly, from the beginning to the end, and generates the corresponding machine code in a single pass.

The main advantage of a single-pass assembler is its simplicity and speed. Since it processes the code in one pass, it requires less memory and processing power than a two-pass assembler. This makes it suitable for use in resource-constrained environments, such as embedded systems or microcontrollers.

However, a single-pass assembler has some limitations. It cannot handle certain types of constructs that require cross-referencing between different parts of the code, such as forward references or global variables. Additionally, it may not be able to perform certain types of optimizations, since it cannot analyze the entire code before generating the object code.

To overcome these limitations, a single-pass assembler may use some techniques such as symbol tables, forward declarations, and partial code generation. A symbol table is used to store the addresses and values of symbols as they are encountered in the source code, allowing the assembler to resolve forward references later on. Forward declarations are used to inform the assembler about symbols that are defined later in the code. Partial code generation involves generating parts of the code as they are encountered and filling in the missing parts later.

In summary, a single-pass assembler is a simple and efficient method for generating object code from source code in a single pass. While it may have some limitations, it can be augmented with various techniques to overcome these limitations and provide an effective means of generating code for resource-constrained environments.

**Advantages of Single Pass Assembler:**

1. **Speed:** A single-pass assembler is faster than a two-pass assembler because it reads the source code only once.
2. **Low Memory Requirement:** A single-pass assembler requires less memory than a two-pass assembler because it does not need to store intermediate data for a second pass.
3. **Simple to Implement:** Single-pass assemblers are easier to implement than two-pass assemblers since they do not require complex logic for handling forward references.
4. **Efficient for Small Programs:** Single-pass assemblers are efficient for small programs that do not have complex dependencies between code segments.

#### Disadvantages of Single Pass Assembler:

1. **Limited Functionality:** Single-pass assemblers have limited functionality and cannot handle complex program structures like recursive functions, complex data structures, etc.
2. **Lack of Optimization:** Single-pass assemblers cannot perform advanced optimization techniques since they cannot analyze the entire program before generating the object code.
3. **Dependency Issues:** Single-pass assemblers cannot handle forward references or symbols that are defined later in the program.
4. **No Modularization:** Single-pass assemblers do not support modular programming, which is an essential feature for developing large software projects.
5. **Poor Error Handling:** Single-pass assemblers have limited error handling capabilities and may not be able to detect all types of errors during the first pass.

#### Conclusion:

A single-pass assembler is a type of assembler that reads the source code only once and generates the corresponding machine code in a single pass. It is faster and requires less memory than a two-pass assembler but has limited functionality and cannot handle complex program structures. Single-pass assemblers are suitable for small programs that do not have complex dependencies between code segments, but they lack support for modular programming and advanced optimization techniques. Despite its limitations, a single-pass assembler is a simple and efficient method for generating object code from source code and is commonly used in resource-constrained environments.

## // Expt. 1 Single Pass Assembler Implementation

```
from sys import exit
```

```
motOpCode = {  
    "STOP": 0,  
    "ADD": 1,  
    "SUB": 2,  
    "MULT": 3,  
    "MOVER": 4,  
    "MOVEM": 5,  
    "COMP": 6,  
    "BC": 7,  
    "DIV": 8,  
    "READ": 9,  
    "PRINT": 10,  
    "START": 1,  
    "END": 2,  
    "EQU": 3,  
    "ORIGIN": 4,  
    "LTORG": 5,  
    "DS": 1,  
    "DC": 2,  
    "AREG": 1,  
    "BREG": 2,  
    "CREG": 3,  
    "DREG": 4,  
    "A": 1,  
    "B": 2,  
}
```

```
motSize = {  
    "STOP": 1,  
    "ADD": 1,  
    "SUB": 1,  
    "MULT": 1,  
    "MOVER": 1,  
    "MOVEM": 1,  
    "COMP": 1,  
    "BC": 1,  
    "DIV": 1,  
    "READ": 1,  
    "PRINT": 1,  
    "START": 1,  
    "END": 1,  
    "EQU": 1,  
    "ORIGIN": 1,  
    "LTORG": 1,  
    "DS": 1,  
    "DC": 1,  
    "AREG": 1,  
    "BREG": 1,  
    "CREG": 1,  
    "DREG": 1,  
    "A": 1,  
    "B": 1,  
}
```

```
l = []  
relativeAddress = []  
machineCode = []  
RA = 0  
current = 0  
count = 0  
n = int(input("Enter the no of instruction lines : "))for i in range(n):
```

```

instructions = input("Enter instruction line {} : ".format(i + 1))l.append(instructions)
l = [x.upper() for x in l] # Converting all the instructions to upper case
for i in range(n):x = l[i]
    if " " in x:
        s1 = ".join(x) a, b =
        s1.split()
        if a in motOpCode:

            value = motOpCode.get(a)size =
            motSize.get(a) previous = size
            RA += current current = previous
            relativeAddress.append(RA) if
            b.isalpha() is True:
                machineCode.append(str(value)) else:
                    temp = list(b)
                    for i in range(len(temp)):if count == 2:
                        temp.insert(i, ' ')count = 0
                    else:
                        count = count + 1s = ".join(temp)
                        machineCode.append(str(value) + " " + s)
            else:

                print("Instruction is not in Op Code Table.")exit(0) # EXIT if
                Mnemonics is not in MOT
        else:

            if x in motOpCode:
                value = motOpCode.get(x)size =
                motSize.get(x) previous = size
                RA += current current =
                previous
                relativeAddress.append(RA)
                machineCode.append(value)
            else:
                print("Instruction is not in Op Code Table.")exit(0)

print("Relative Address Instruction OpCode")
for i in range(n):
    print("{} {} {}".format(relativeAddress[i], l[i], machineCode[i]))

```

## Output :

```
Run C:\Python3.10\python.exe C:\COLLEGE\SPCC\EXP-1.py
Enter the no of instruction lines : 7
Enter instruction line 1 : START 200
Enter instruction line 2 : MOVER AREG
Enter instruction line 3 : A 20
Enter instruction line 4 : MOVER BREG
Enter instruction line 5 : LTORG
Enter instruction line 6 : DS
Enter instruction line 7 : END
Relative Address      Instruction      OpCode
0                     START 200      1 20 0
1                     MOVER AREG      4
2                     A 20          1 20
3                     MOVER BREG      4
4                     LTORG          5
5                     DS              1
6                     END              2

Process finished with exit code 0
```

## **EXPERIMENT 2**

**Aim :** Implement Two pass Assembler

### **Theory:**

A two-pass assembler is a type of assembler that reads the source code twice and generates the corresponding machine code in two passes. The first pass of the assembler scans the source code and builds a symbol table that contains information about labels, symbols, and their corresponding memory addresses. The symbol table is used in the second pass to generate the final machine code.

During the first pass, the assembler reads the source code and identifies all the labels, symbols, and instructions used in the program. The assembler assigns memory addresses to each label and symbol and stores this information in a symbol table. The assembler also generates intermediate code that includes information about the memory addresses of the labels and symbols used in the program.

During the second pass, the assembler reads the intermediate code generated in the first pass and generates the final machine code. The assembler uses the symbol table to resolve all the references to labels and symbols in the program and generates the corresponding machine code. The assembler also performs error checking during the second pass to ensure that the program is syntactically and semantically correct.

Two-pass assemblers are capable of handling more complex program structures, such as recursive functions and data structures, since they can analyze the entire program before generating the object code. However, they are slower and require more memory than single-pass assemblers since they need to store intermediate data for a second pass.

Overall, a two-pass assembler is a more powerful tool for generating machine code from source code than a single-pass assembler, and is commonly used in software development environments where program complexity and modularity are critical factors.

### **Advantages:**

1. Able to handle complex program structures: Two-pass assemblers are capable of handling complex program structures such as recursive functions and data structures since they can analyze the entire program before generating the object code.
2. Can resolve external references: Two-pass assemblers are capable of resolving external references to symbols, labels, and other parts of the program that are not defined in the current module.
3. Error checking: Two-pass assemblers can perform more thorough error checking on the source code, since they can analyze the entire program before generating the object code.

### **Disadvantages:**

1. Slower: Two-pass assemblers are slower than single-pass assemblers because they have to read and analyze the source code twice.
2. Requires more memory: Two-pass assemblers require more memory than single-pass assemblers because they have to store intermediate data for a second pass.
3. Not suitable for small programs: Two-pass assemblers are not suitable for small programs since the overhead of reading and analyzing the source code twice is not justified for small programs.

**Conclusion:**

Assemblers are essential tools for translating high-level programming languages into machine code. Two common types of assemblers are single-pass assemblers and two-pass assemblers. Single-pass assemblers read the source code once and generate the corresponding machine code in a single pass. On the other hand, two-pass assemblers read the source code twice and generate the machine code in two passes.

Each type of assembler has its own set of advantages and disadvantages. Single-pass assemblers are fast and require less memory, but are limited in their ability to handle complex program structures. Two-pass assemblers, on the other hand, can handle more complex program structures and perform more thorough error checking, but are slower and require more memory.

## // Expt. 2 Two Pass Assembler Implementation

```
with open('EXP.txt') as t: data = []
    for line in t.readlines(): data.append(line.split())
# print(data)

symbols = []
value = 0

def contains(string):
    string = list(string)
    for i in string:
        if i == "F":
            return 4
        elif i == "D":
            return 8
    return 1

def contains_literal(string):
    string = list(string)
    if "=" in string:
        return True

for j, i in enumerate(data):

    if len(i) == 2 and i[0].lower() == "using":
        value = 0
        continue

    if len(i) == 2:
        value += 4
    if j == 1:
        value = 0
        continue

    if len(i) == 3:
        length = contains(i[2])
        if i[1].lower() == "eqv":
            symbols.append([i[0], int(i[2]), length, 'A'])
            base = int(i[2])
        else:
            symbols.append([i[0], value, length, "R"])
            if (length != 4):
                value += length
            else:
                value += 4

print("OUTPUT of Pass 1\n\nSymbol Table (ST)")
print("Symbol\tValue\tLength\tRelocation")
for i in symbols:
    print(i[0], "\t", hex(i[1])[2:], '(', i[1], ')', "\t\t", i[2], "\t", i[3])

literals = []
lvalue = value

for j, i in enumerate(data):
    if len(i) == 2:
        if contains_literal(i[1]):
            a = list((i[1].split('='))[1])
            length = contains(a[0])
            literals.append([(i[1].split(',')[1]), lvalue, length, "R"])
            if (length != 4):
                lvalue += length
            else:
                lvalue += 4
            # print(a)
print("\nLiteral Table (LT)")
print("Literal\tValue\tLength\tRelocation")
for i in literals:
    print(i[0], "\t", hex(i[1])[2:], '(', i[1], ')', "\t\t", i[2], "\t", i[3])
```



```

main = symbols + literals
mot = [['L', int('58', 16)], ['ST', int('50', 16)], ['A', int('5A', 16)]]

def getOpHex(op):
    for i in mot:
        if i[0] == op:
            return i[1]
    return None

def getOpOperand(op):
    for i in main:
        if i[0] == op:
            return i[1]
    return None

print("-----")
print("\nOUTPUT of Pass 2\n\nMachine Code")
print("Instruction\tMachine Code")

one = 100
for i, j in enumerate(data[2:], 1):
    if len(j) == 2:
        final = getOpHex(j[0]) + getOpOperand(j[1].split(',')[1]) + one +
        base
        print(j[0], '\t\t\t', hex(final)[2:], '(', final, ')')

bases = []
for i in range(0, 16):
    if (i == base):
        bases.append(['Y', 000000])
    else:
        bases.append(['N', None])
print("\nBase Table (BT)")
print("Base Availability Indicator Contents")
for j, i in enumerate(bases):
    if (i[1] == 0):
        print(j, "\t", i[0], "\t\t\t\t\t", str(i[1]) * 6)
    else:
        print(j, "\t", i[0])

```

## OUTPUT:

### INPUT

```

PG1 START 0
  USING *,BASE
  L 1,FOUR
  A 1,FIVE
  A 1,=F'7'
  A 1,=D'8'
  ST 1,TEMP
FOUR DC F'4'
FIVE DC F'5'
BASE EQV 8
TEMP DC '1'D
END

```

Run



## OUTPUT of Pass 1

### Symbol Table (ST)

Symbol	Value	Length	Relocation
PG1	0 ( 0 )	1	R
FOUR	14 ( 20 )	4	R
FIVE	18 ( 24 )	4	R
BASE	8 ( 8 )	1	A
TEMP	1c ( 28 )	8	R

### Literal Table (LT)

Literal	Value	Length	Relocation
=F'7'	24 ( 36 )	4	R
=D'8'	28 ( 40 )	8	R

## OUTPUT of Pass 2

### Machine Code

#### Instruction Machine Code

L	d8 ( 216 )
A	de ( 222 )
A	ea ( 234 )
A	ee ( 238 )

## **EXPERIMENT 3**

**Aim :** Implementation Two Pass Macro Processor

**Theory:**

A two-pass macro processor is a program that processes source code containing macro definitions and macro calls in two passes. During the first pass, the macro definitions are extracted and stored in a macro table. During the second pass, the macro calls are replaced with their corresponding macro definitions.

The first pass of the two-pass macro processor extracts macro definitions and builds a macro table. The macro definition is typically of the form:

```
MACRO
    macro_name parameter_list macro_body
MEND
```

macro\_name is the name of the macro, parameter\_list is a comma-separated list of parameters that the macro takes, and macro\_body is the code that the macro expands into. The macro body can contain any valid assembly language code, including other macro calls.

The second pass of the two-pass macro processor replaces macro calls with their corresponding macro definitions. The macro call is typically of the form:

```
macro_name argument_list
```

macro\_name is the name of the macro being called, and argument\_list is a comma-separated list of arguments that are passed to the macro. During the second pass, the macro processor looks up the macro in the macro table and replaces the macro call with the expanded macro body, replacing the parameter names with the actual arguments passed to the macro.

**Advantages of a two-pass macro processor:**

1. It can handle complex macro definitions and calls that involve other macros.
2. It allows the use of symbolic constants as arguments, which can make code more readable and easier to maintain.
3. It can perform error checking on macro definitions during the first pass, ensuring that macro calls have the correct number and type of arguments.

**Disadvantages of a two-pass macro processor:**

1. It requires more memory to store the macro table and expanded macro bodies.
2. It takes longer to process than a single-pass macro processor, as it requires two passes over the source code.
3. The macro definition and call syntax can be more complex than that of a single-pass macro processor, requiring more knowledge and expertise on the part of the programmer.

**Conclusion:**

A macro processor is a valuable tool for reducing code duplication and improving code readability and maintainability. Single-pass macro processors are simple and efficient but have limited capabilities, while two-pass macro processors can handle more complex macro definitions and calls but are more memory-intensive and time-consuming. The implementation of a macro processor in Python demonstrated the use of regular expressions and string manipulation to extract macro definitions and replace macro calls with expanded code.

### // Expt. 3 Two Pass Macro Processor Implementation

import re

# Regular expressions for macro definition and macro call  
DEFINITION\_REGEX = r'^\s\*MACRO\s+(\w+)\s+(.\*)\$'  
CALL\_REGEX = r'(\w+)\s\*(((.\*)\s\*)\s\*)'

class Macro:  
def init (self, name, parameters, code): self.name = name  
self.parameters = parameters self.code = code

class TwoPassMacroProcessor: def init (self):  
self.macros = {}

def first\_pass(self, source):  
# Extract macro definitions and build macro table  
lines = source.split('\n') i = 0  
while i < len(lines):  
match = re.match(DEFINITION\_REGEX, lines[i]) if match:  
name, parameters = match.groups()  
parameters = [p.strip() for p in parameters.split(',')]  
code = []  
i += 1  
while i < len(lines) and not re.match(DEFINITION\_REGEX,  
lines[i]):  
code.append(lines[i]) i += 1  
macro = Macro(name, parameters, code) self.macros[name] = macro  
else:  
i += 1

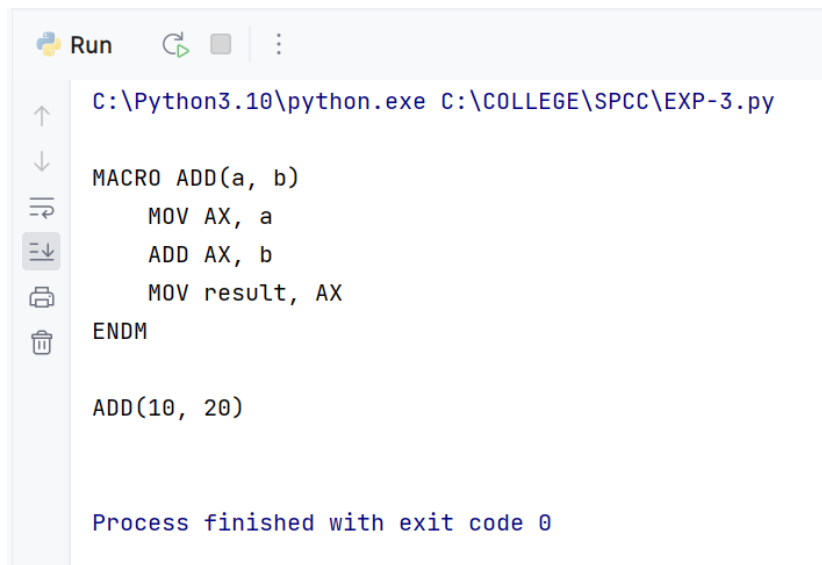
def second\_pass(self, source):  
# Replace macro calls with expanded code  
lines = source.split('\n') for i in range(len(lines)):  
match = re.search(CALL\_REGEX, lines[i]) if match:  
name, arguments = match.groups()  
arguments = [a.strip() for a in arguments.split(',')]  
macro = self.macros.get(name)  
if macro:  
expanded\_code = macro.code.copy() for j in range(len(arguments)):  
expanded\_code = [re.sub(r'\b' + macro.parameters[j]  
+ r'\b', arguments[j], line) for line in expanded\_code]  
lines[i:i+1] = expanded\_code return '\n'.join(lines)

def process(self, source): self.first\_pass(source)  
return self.second\_pass(source) source = '''  
MACRO ADD(a, b) MOV AX, a ADD AX, b  
MOV result, AX ENDM

ADD(10, 20) '''

processor = TwoPassMacroProcessor()  
result = processor.process(source) print(result)

## Output :



The screenshot shows a code editor window with a toolbar at the top containing a Python icon, a 'Run' button, a refresh icon, a close icon, and a menu icon. The main text area contains assembly code. On the left side of the text area is a vertical toolbar with icons for undo, redo, expand/collapse, print, and delete. The code defines a macro and uses it.

```
C:\Python3.10\python.exe C:\COLLEGE\SPCC\EXP-3.py

MACRO ADD(a, b)
    MOV AX, a
    ADD AX, b
    MOV result, AX
ENDM

ADD(10, 20)

Process finished with exit code 0
```

```
MOV AX, 10
ADD AX, 20
MOV result, AX
```

## **EXPERIMENT 4**

**Aim :** Implementation Lexical Analyzer

### **Theory:**

A lexical analyzer, also known as a scanner, is the first phase of a compiler that breaks down the source code into a sequence of tokens for further analysis. The purpose of the lexical analyzer is to identify and classify individual lexemes, which are the smallest units of meaning in a programming language, such as identifiers, keywords, operators, literals, and punctuation symbols.

The process of lexical analysis involves scanning the source code character by character, recognizing patterns that match the language grammar rules, and grouping them into tokens. This is achieved by using regular expressions, a set of patterns that define the valid lexemes for a language, and a finite automaton, a machine that reads the input stream and transitions between states based on the current character and the defined rules.

The output of the lexical analyzer is a sequence of tokens, each consisting of a token type, which identifies the lexeme category, and a value, which represents the actual lexeme. The token sequence is passed to the next phase of the compiler, the syntax analyzer, for further analysis and processing.

### **Advantages:**

1. **Simplifies Parsing:** The primary advantage of a lexical analyzer is that it simplifies the process of parsing and analyzing the source code by breaking it down into manageable units.
2. **Easy Error Detection:** A lexical analyzer detects lexical errors like misspelled keywords or invalid operators that can improve the quality of the compiled code.
3. **Precise Tokenization:** The lexical analyzer can precisely recognize and tokenize every lexeme from the input source code.
4. **Better Optimization:** The lexical analyzer can enable better optimization of the compiled code since the code is broken down into smaller, more manageable pieces.

### **Disadvantages:**

1. **High Overhead:** The process of lexical analysis has a high overhead cost since it requires the development and maintenance of the regular expressions and finite automata that recognize the lexemes.
2. **Difficulty in Debugging:** Debugging errors can be difficult because lexical errors can be cascading and hard to trace back to their source.
3. **Limitations of Regular Expressions:** There are limitations to regular expressions, which can make it difficult to define the lexemes of some programming languages.
4. **Limited Context Sensitivity:** A lexical analyzer has a limited ability to recognize context-sensitive constructs like comments, macros, and preprocessor directives.

### **Conclusion :**

A lexical analyzer is an essential component of a compiler that helps in breaking down the source code into smaller, manageable units called lexemes. While it simplifies the process of parsing and analyzing source code, it also has some limitations like high overhead cost, difficulty in debugging, and limited context sensitivity. Despite its limitations, a lexical analyzer is a critical component of modern compilers and plays an important role in generating high-quality, optimized code.

## // Expt. 4 Lexical Analyzer Implementation

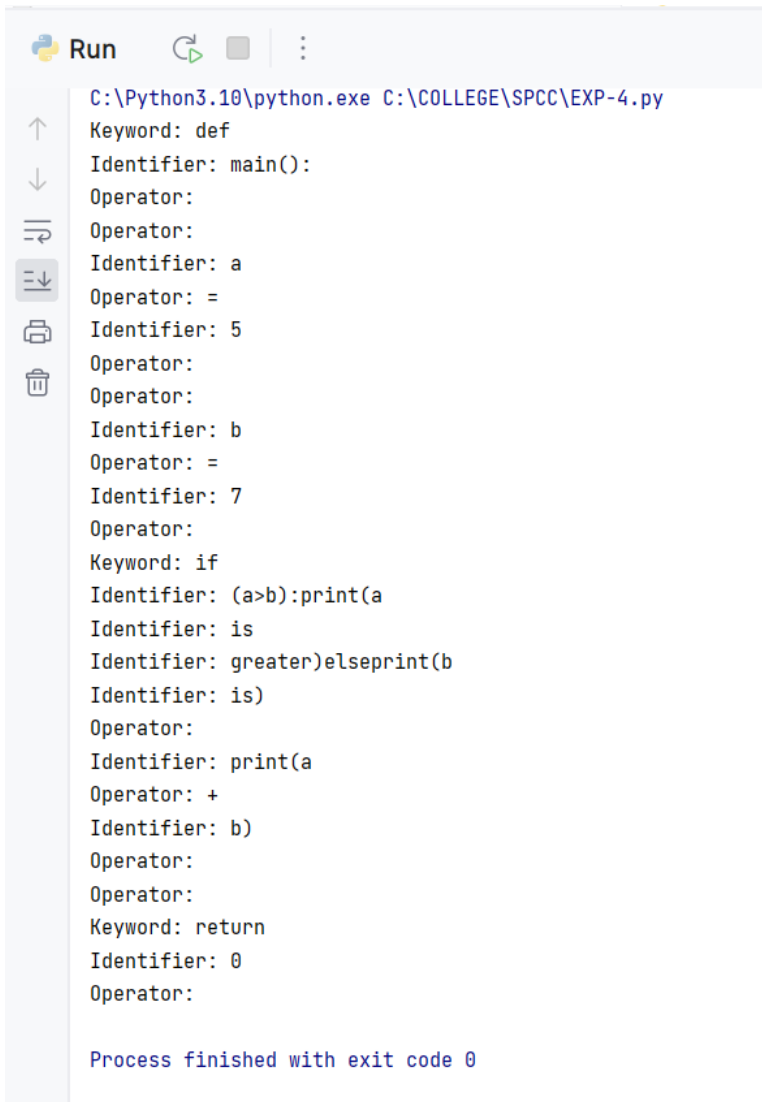
```
KEYWORDS = ["for", "while", "if", "else", "def", "return", "in", "not", "and", "or", "print", "range", "input"]
FUNCTIONS = ["len", "int", "str", "float", "bool", "list", "tuple", "dict", "set", "sorted", "max", "min"]
OPERATORS = ["+", "-", ":", "/", "%", "//", ":", "+=", "-=", "=", "/=", "=",
"==", "!=", "<", ">", "<=", ">=", "not", "in", "and", "or"]
```

```
def parse_code(code):
    for line
        in code:
            parts = line.split(" ")
            for part in parts:
                if part in KEYWORDS: print("Keyword: " + part)
                elif part in FUNCTIONS: print("Function: " + part)
                elif part in OPERATORS: print("Operator: " + part)
                else:
                    print("Identifier: " + part)
```

```
code = [
    "def main():", "    a =",
    "    5",
    "    b = 7",
    "    if (a>b):",
    "        print(a is greater)",
    "    else:",
    "        print(b is)",
    "    print(a + b)", "    return",
    "    0",
    "]"
```

```
parse_code(code)
```

## Output :



```
Run  [Refresh] [Close] [Menu]

C:\Python3.10\python.exe C:\COLLEGE\SPCC\EXP-4.py

↑ Keyword: def
↓ Identifier: main():
⇌ Operator:
⇌ Operator:
⇌ Identifier: a
⇌ Operator: =
⇌ Identifier: 5
⇌ Operator:
⇌ Operator:
⇌ Identifier: b
⇌ Operator: =
⇌ Identifier: 7
⇌ Operator:
⇌ Keyword: if
⇌ Identifier: (a>b):print(a
⇌ Identifier: is
⇌ Identifier: greater)elseprint(b
⇌ Identifier: is)
⇌ Operator:
⇌ Identifier: print(a
⇌ Operator: +
⇌ Identifier: b)
⇌ Operator:
⇌ Operator:
⇌ Keyword: return
⇌ Identifier: 0
⇌ Operator:

Process finished with exit code 0
```



## **EXPERIMENT 5**

**Aim:** Implementation of Parser techniques

**Theory:**

Top-down and bottom-up parsing are two main techniques used in computer science to analyze and understand the structure of a given input sequence, such as a program written in a programming language.

**Top-Down Parsing:** Top-down parsing is a type of parsing algorithm that starts at the root of the parse tree and works its way down to the leaves. The process begins with the start symbol of the grammar, and the parser tries to derive the input sequence from the start symbol using a set of production rules. This process is also known as recursive descent parsing, as the parser repeatedly applies production rules to nonterminal symbols until it reaches the leaves of the parse tree, which are the terminal symbols of the input sequence. Top-down parsing is often used in compilers for programming languages with LL(1) grammars.

**Bottom-Up Parsing:** Bottom-up parsing is a type of parsing algorithm that starts at the leaves of the parse tree and works its way up to the root. The process begins with the input sequence, and the parser tries to apply production rules in reverse to reduce the input sequence to the start symbol of the grammar. The most commonly used bottom-up parsing algorithm is shift-reduce parsing, in which the parser shifts input tokens onto a stack and reduces them using production rules when possible. Bottom-up parsing is often used in compilers for programming languages with LR(1) grammars.

**Differences:** The main difference between top-down and bottom-up parsing is the direction in which they construct the parse tree. Top-down parsing constructs the parse tree from the top (the start symbol) to the bottom (the leaves), while bottom-up parsing constructs the parse tree from the bottom (the leaves) to the top (the start symbol). Top-down parsing can be easier to implement and faster than bottom-up parsing, but it can be less efficient in handling ambiguous grammars. Bottom-up parsing is more powerful and flexible than top-down parsing, but it can be more complex to implement and slower than top-down parsing.

Parsing is a crucial process in computer science that is used to analyze and understand the structure of a given input sequence, such as a program written in a programming language. Here are some advantages and disadvantages of parsing:

**Advantages:**

1. **Error Detection:** Parsing is an effective way to detect syntax errors in a program. A parser can identify syntax errors in the input sequence and report them to the user, making it easier to debug and fix the program.
2. **Code Optimization:** A parser can also optimize code by analyzing the structure of the input sequence and generating more efficient code.
3. **Language Translation:** Parsing is used in language translation tools that can translate code written in one programming language to another language.
4. **Code Generation:** A parser can generate code from a high-level programming language to machine code that can be executed on a computer.

**Disadvantages:**

1. **Time-Consuming:** Parsing can be a time-consuming process, especially for large programs. It can take a lot of time to analyze the structure of the input sequence and generate a parse tree.
2. **Resource-Intensive:** Parsing requires a significant amount of computing resources, such as memory and processing power, especially when parsing complex input sequences.
3. **Ambiguity:** Ambiguity is a significant issue in parsing, as it can lead to multiple interpretations of the input sequence. Resolving ambiguity can be complex and time-consuming.
4. **Complexity:** The complexity of parsing can make it difficult to implement and maintain, especially for programming languages with complex grammars.

Overall, parsing is an essential process in computer science, but it comes with its own set of advantages and disadvantages. While it can help to detect errors, optimize code, and translate languages, it can also be time-consuming, resource-intensive, and complex.

**Conclusion:**

Parsing is a critical process in computer science that is used to analyze and understand the structure of input sequences, such as programs written in programming languages. There are two main types of parsing: top-down and bottom-up. Top-down parsing starts at the root of the parse tree and works its way down to the leaves, while bottom-up parsing starts at the leaves and works its way up to the root. Both techniques have advantages and disadvantages, and the choice of which one to use depends on the specific requirements of the task at hand.

Parsing can be beneficial for detecting errors, optimizing code, translating languages, and generating code, but it can also be time-consuming, resource-intensive, and complex. Overall, parsing is an essential tool in computer science that plays a significant role in enabling the development of robust and efficient software systems.

## // Expt. 5 Parser Techniques Implementation

```
import re
```

```
class Token:
```

```
    def __init__(self, token_type, value): self.token_type =  
        token_type self.value = value
```

```
class Parser:
```

```
    def __init__(self, text):  
        self.tokens = self.tokenize(text) self.pos = 0
```

```
    def parse(self):
```

```
        return self.expr()
```

```
    def tokenize(self, text): token_exprs
```

```
        = [  
            (r'\d+', 'INT'),  
            (r'\+', 'PLUS'),  
            (r'\-', 'MINUS'),  
            (r'\*', 'MULTIPLY'),  
            (r'\/', 'DIVIDE'),  
            (r'\(', 'LPAREN'),  
            (r'\)', 'RPAREN'),  
            (r'\s', None) # skip whitespace  
        ]
```

```
        tokens = [] pos =
```

```
        0
```

```
        while pos < len(text): match =
```

```
            None
```

```
            for token_expr in token_exprs: pattern, token_type
```

```
                = token_expr regex = re.compile(pattern)
```

```
                match = regex.match(text, pos) if match:
```

```
                    value = match.group(0) if
```

```
                        token_type:
```

```
                            token = Token(token_type, value)
```

```
                            tokens.append(token)
```

```
                    break if
```

```
                not match:
```

```
                    raise ValueError(f'Invalid input at position {pos}') else:
```

```
                        pos = match.end(0) return
```

```
        tokens
```

```
    def consume(self, token_type):
```

```
        if self.pos < len(self.tokens) and self.tokens[self.pos].token_type
```

```
        == token_type:
```

```
            self.pos += 1 else:
```

```
            raise ValueError(f'Expected token type {token_type} at position
```

```
{self.pos}')
```

```
    def factor(self):
```

```
        token = self.tokens[self.pos] if
```

```
            token.token_type == 'INT':
```

```
                self.consume('INT') return
```

```
                int(token.value)
```

```
            elif token.token_type == 'LPAREN':
```

```
                self.consume('LPAREN')
```

```
                value = self.expr()
```

```
                self.consume('RPAREN') return
```

```
                value
```

```

def term(self):
    value = self.factor()
    while self.pos < len(self.tokens): token =
        self.tokens[self.pos]
        if token.token_type == 'MULTIPLY':
            self.consume('MULTIPLY')
            value *= self.factor()
        elif token.token_type == 'DIVIDE':
            self.consume('DIVIDE')

            value /= self.factor() else:
                break
    return value

def expr(self):
    value = self.term()
    while self.pos < len(self.tokens): token =
        self.tokens[self.pos] if token.token_type ==
        'PLUS':
            self.consume('PLUS') value +=
            self.term()
        elif token.token_type == 'MINUS':
            self.consume('MINUS')
            value -= self.term() else:
                break
    return value

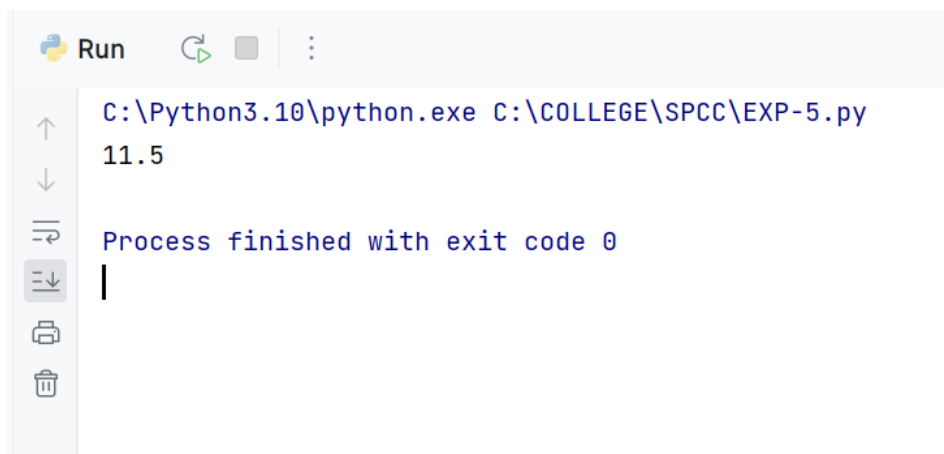
```

```

text = '2 * (3 + 4) - 5 / 2' parser =
Parser(text)
result = parser.parse() print(result) # Output:
12.5

```

**Output :**



```

Run
C:\Python3.10\python.exe C:\COLLEGE\SPCC\EXP-5.py
11.5

Process finished with exit code 0

```

## **EXPERIMENT 6**

**Aim :** Implement Intermediate code generation phase of compiler

### **Theory:**

The intermediate code generation phase of a compiler is the process of transforming the source code into an intermediate representation that is easier to analyze and optimize. The intermediate code is a low-level representation of the program that is independent of the source and target languages. It is often used as an input to other compiler phases, such as optimization and code generation.

The main goal of the intermediate code generation phase is to provide a way to represent the program's control flow and data flow in a way that can be easily analyzed by subsequent phases of the compiler. The intermediate code is typically represented in a form that is similar to assembly language, with a set of instructions that perform simple operations on registers or memory.

During the intermediate code generation phase, the compiler may perform several transformations on the source code to simplify the intermediate code. For example, it may remove redundant expressions, replace complex control flow constructs with simpler ones, and transform loops and recursion into equivalent forms.

One of the advantages of using an intermediate representation is that it allows the compiler to separate the analysis and optimization phases from the code generation phase. This makes it possible to apply different optimization techniques to the intermediate code without having to worry about the details of the target architecture.

Another advantage is that it provides a way to generate code for multiple target architectures without having to modify the analysis and optimization phases. The intermediate code can be translated into machine code for any target architecture, allowing the compiler to support multiple platforms with a single codebase.

Overall, the intermediate code generation phase plays a crucial role in the compilation process. It provides a way to transform the source code into a form that is easier to analyze and optimize, and allows the compiler to generate efficient code for a wide range of target architectures.

### **Advantages of Intermediate Code Generation:**

- Simplifies the process of analyzing and optimizing code: The intermediate code representation provides a simplified view of the source code that can be easily analyzed and optimized by the compiler.
- Increases portability: By generating an intermediate code representation, the compiler can generate code for multiple target architectures without having to modify the analysis and optimization phases.
- Facilitates debugging: Intermediate code can provide a more detailed view of program execution than the source code, which can make it easier to debug complex programs.
- Enables Just-In-Time (JIT) compilation: JIT compilers can use intermediate code as a starting point for optimizing and generating machine code at runtime, which can improve the performance of dynamic language implementations.

### **Disadvantages of Intermediate Code Generation:**

- Adds extra compilation time: Generating an intermediate code representation adds an extra compilation phase to the overall compilation process, which can increase compilation time.
- May result in less efficient code: Depending on the quality of the intermediate code representation and the optimizations performed by the compiler, the resulting machine code may be less efficient than code generated directly from the source code.
- Increases memory usage: Intermediate code may require additional memory to store, which can be a concern for memory-constrained systems.
- Increases complexity: The use of an intermediate code representation can add complexity to the compiler implementation and increase the likelihood of bugs and errors.

### **Conclusion:**

Intermediate code generation is an important phase in the compilation process that transforms the source code into a low-level representation that is easier to analyze and optimize.

While intermediate code generation offers several advantages, such as increased portability, simplified analysis and optimization, and better debugging support, it also has some disadvantages, including increased compilation time, potential inefficiencies in the generated code, increased memory usage, and increased complexity in the compiler implementation.

Overall, intermediate code generation is an important tool that compilers can use to generate efficient code for a wide range of target architectures.

### // Expt. 6 Implement Intermediate code generation phase of compiler

import ast

```
class IntermediateCodeGenerator(ast.NodeVisitor):  
    def __init__(self):  
        self.instructions = []  
        self.temp_count = 0
```

```
    def new_temp(self):  
        temp = f"t{self.temp_count}"  
        self.temp_count += 1  
        return temp
```

```
    def visit_Assign(self, node):  
        target = node.targets[0].id  
        value = self.visit(node.value)  
        self.instructions.append(('ASSIGN', target, value))
```

```
    def visit_BinOp(self, node):  
        left = self.visit(node.left)  
        right = self.visit(node.right)
```

```
        op = node.op.__class__.__name__  
        temp = self.new_temp()  
        self.instructions.append((op, temp, left, right))  
        return temp
```

```
    def visit_Num(self, node):  
        return node.n
```

```
    def visit_Name(self, node):  
        return node.id
```

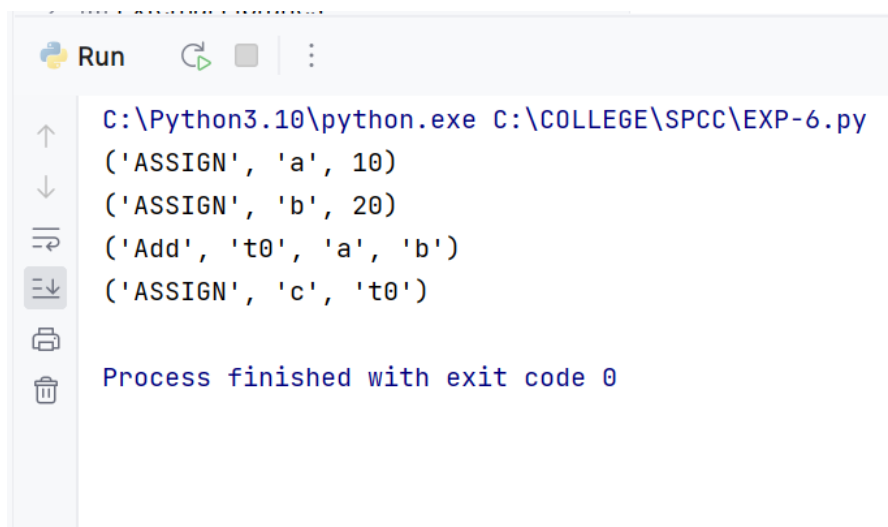
```
    def visit_Print(self, node):  
        value = self.visit(node.values[0])  
        self.instructions.append(('PRINT', value))
```

```
def generate_intermediate_code(source_code):  
    ast_tree = ast.parse(source_code)  
    icg = IntermediateCodeGenerator()  
    icg.visit(ast_tree)  
    return icg.instructions
```

```
# Example usage  
source_code = """  
a = 10  
b = 20  
c = a + b  
print(c)"""
```

```
instructions = generate_intermediate_code(source_code)  
for instruction in instructions:  
    print(instruction)
```

### Output :



```
Run  
C:\Python3.10\python.exe C:\COLLEGE\SPCC\EXP-6.py  
( 'ASSIGN', 'a', 10)  
( 'ASSIGN', 'b', 20)  
( 'Add', 't0', 'a', 'b')  
( 'ASSIGN', 'c', 't0')  
  
Process finished with exit code 0
```

## EXPERIMENT 7

**Aim :** Implementation of code generation algorithm of compiler

### **Theory:**

The code generation phase of a compiler is responsible for translating the intermediate representation of the source code into executable code for the target machine.

### **Internal or Data Structure:**

#### **1. Register Descriptors:**

- It is used to inform the code generator about the availability of registers
- It keeps track of values stored in each register.
- Whenever a new register is required during code generation, this descriptor is consulted for register availability
  - ✓ *Keep track of what is currently in each register.*
  - ✓ *Initially, all the registers are empty.*

#### **2. Address Descriptors:**

- It is used to keep track of locations (L) where the values of identifiers/variables are stored.
- These locations might be a register, heap, stack, memory or a combination of the mentioned locations/address.

#### **3. getReg( ) function:**

- It is used to determine *the status of available registers and the location of name values.*
- *getReg works as follows:*
  - a. If variable Y is already in register R, it uses that register.
  - b. Else if some register R is available, it uses that register.
  - c. Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions

### **Code Generation Algorithm Steps**

- **Input:** Three address statement of the form  **$x = y \text{ op } z$**
- **The code generator performs the following actions**
  - Let **L** be the **location** where the output of  $y \text{ op } z$  is to be stored.
- I. **Call function getReg**, to decide the location of **L**.
- II. **Determine the present location** (register or memory) of **y** by consulting the Address Descriptor of **y**.
  - If **y** is not presently in register **L**, then generate the following instruction to copy the value of **y** to **L**: **MOV y', L**. (where **y'** represents the copied value of **y**)
- III. **Determine the present location of z** using the same method used in step 2 for **y** and generate the following instruction: **OP z', L** (where **z'** represents the copied value of **z**)
- IV. Now **L contains the value of y op z**, that is intended to be assigned to **x**. So, if **L** is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.
- V. If **y** and **z** has **no further use**, they can be given **back to the system**.



**Conclusion:**

The code generation phase of a compiler is responsible for generating efficient and optimized code for the target machine.

It involves several techniques and theories such as three- address code, register allocation, instruction selection, peephole optimization, and code compression. The advantages of code generation include efficient code, machine independence, automatic memory management, and easy debugging.

However, the process may also have some disadvantages such as increased compilation time, limited optimization, limited error detection, and increased memory usage. Overall, the benefits of code optimization and efficient code generation make the code generation phase a crucial part of the compilation process.

## // Expt. No. 7 Code Generation Algorithm Implementation

```
class CodeGenerator:
    def __init__(self, intermediate_code):
        self.intermediate_code = intermediate_code
        self.generated_code = []

    def generate_code(self):
        self.generated_code.append('_start:')

        for instruction in self.intermediate_code:
            opcode = instruction['opcode']
            operands = instruction['operands']

            if opcode == 'ADD':
                self.add(operands)
            elif opcode == 'SUB':
                self.sub(operands)
            elif opcode == 'MULT':
                self.mult(operands)
            elif opcode == 'DIV':
                self.div(operands)
            else:
                raise ValueError(f"Invalid opcode '{opcode}'")

        self.generated_code.append('MOVER R0, %REGB')
        self.generated_code.append('MOVER R1, %REGA')
        self.generated_code.append('int $0x80')

    def add(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f'MOVER {op1}, %REGA')
        self.generated_code.append(f'ADD {op2}, %REGA')
        self.generated_code.append(f'MOVER %REGA, {result}')

    def sub(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f'MOVER {op1}, %REGA')
        self.generated_code.append(f'SUB {op2}, %REGA')
        self.generated_code.append(f'MOVER %REGA, {result}')

    def mult(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f'MOVER {op1}, %REGA')
        self.generated_code.append(f'MULT {op2}, %REGA')
        self.generated_code.append(f'MOVER %eax, {result}')

    def div(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f'MOVER {op1}, %REGA')
        self.generated_code.append(f'cdq')
        self.generated_code.append(f'DIV {op2}')
        self.generated_code.append(f'MOVER %REGA, {result}')

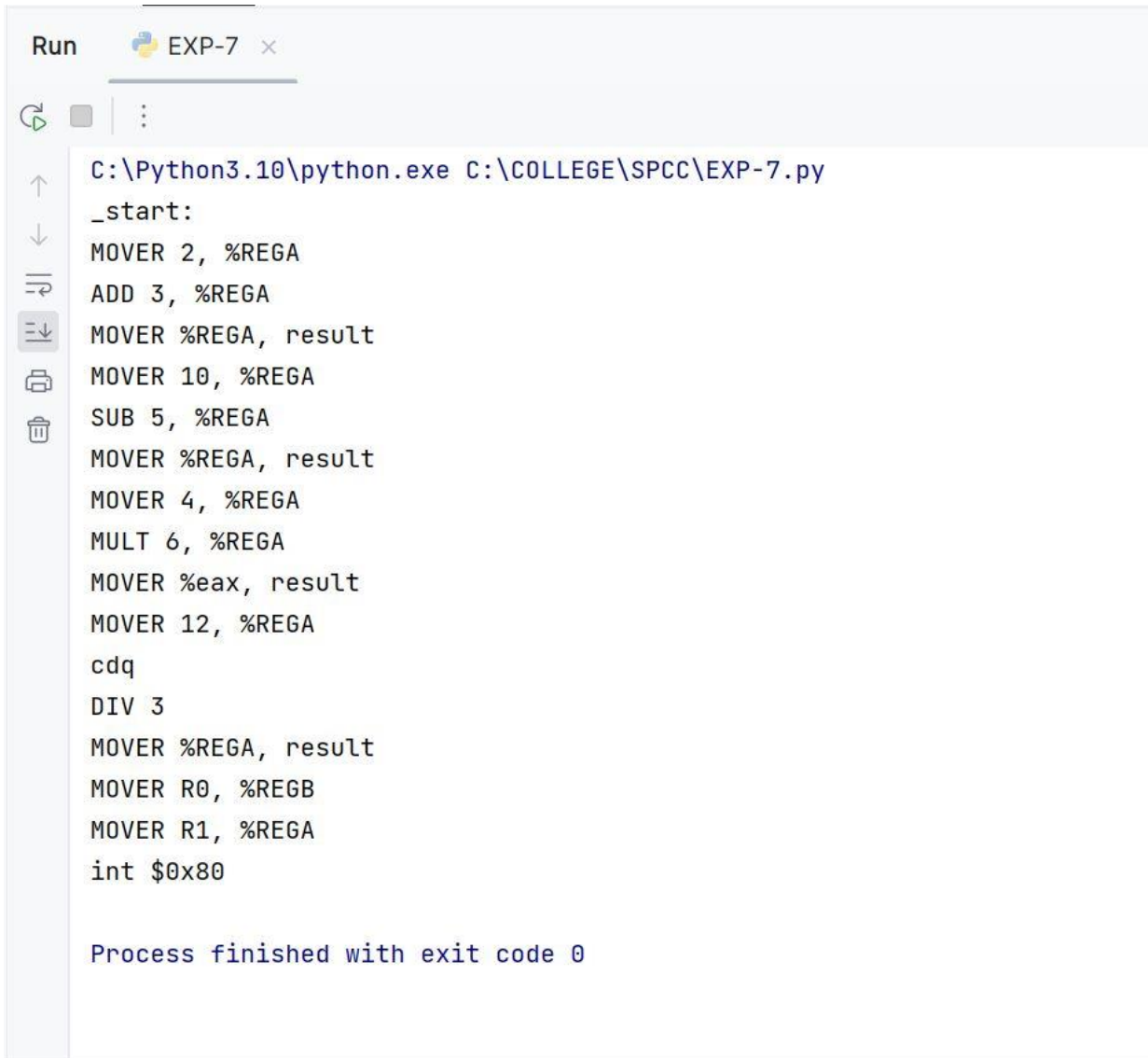
# Example intermediate code
intermediate_code = [
    {'opcode': 'ADD', 'operands': [2, 3, 'result']},
    {'opcode': 'SUB', 'operands': [10, 5, 'result']},
    {'opcode': 'MULT', 'operands': [4, 6, 'result']},
    {'opcode': 'DIV', 'operands': [12, 3, 'result']}
```

]

```
# Generate x86 assembly code
code_generator = CodeGenerator(intermediate_code)
code_generator.generate_code()
assembly_code = '\n'.join(code_generator.generated_code)
```

```
# Print generated x86 assembly code
print(assembly_code)
```

### Output :



```
Run EXP-7 x
C:\Python3.10\python.exe C:\COLLEGE\SPCC\EXP-7.py
_start:
MOVER 2, %REGA
ADD 3, %REGA
MOVER %REGA, result
MOVER 10, %REGA
SUB 5, %REGA
MOVER %REGA, result
MOVER 4, %REGA
MULT 6, %REGA
MOVER %eax, result
MOVER 12, %REGA
cdq
DIV 3
MOVER %REGA, result
MOVER R0, %REGB
MOVER R1, %REGA
int $0x80

Process finished with exit code 0
```

## EXPERIMENT 8

**Aim:** Study and Implement LEX and YACC

**Theory:**

### LEX

Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general-purpose language which recognizes regular expressions. The regular expressions are specified by the user in the source specifications given to Lex. The Lex written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions. Lex is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages."

#### ■ Structure of Lex Program

- Lex source is separated into three sections by %% delimiters
- The general format of Lex source is

{ definitions }	
%%	(required)
{ Transition rules }	
%%	(optional)
{ User code }	

- The absolute minimum Lex program is thus %%

#### ■ Definitions

- Declarations of ordinary C variables, constants and Libraries.

```
%{  
#include <math.h>  
#include <stdio.h>  
#include <stdlib.h>  
%}
```

- flex definitions :- name definition  
Digit [0-9] (Regular Definition)

#### ■ Operators → " \ [ ] ^ - ? . \* + | ( ) \$ / { } % < >

- If they are to be used as text characters, an escape should be used

```
\$ = "$"  
\\ = "\"
```

- Every character but *blank*, *tab* (\t), *newline* (\n) and the list above is always a text character

#### ■ Translation Rules

- The form of rules are:

```
Pattern { action }
```

- The actions are C/C++ code.

```
[0-9]+ { return(Integer); } // RE  
{DIGIT}+ { return(Integer); } // RD
```

### ■ User Subroutines Section

- You can use your Lex routines in the same ways you use routines in other programming languages (Create functions, identifiers).
- The section where main() is placed

```
%{  
    void print(String x);  
%}  
%%  
{letter}+      print("ID");  
%%  
main() {  
    yylex();  
}  
void print(String x) {  
    printf(x);  
}
```

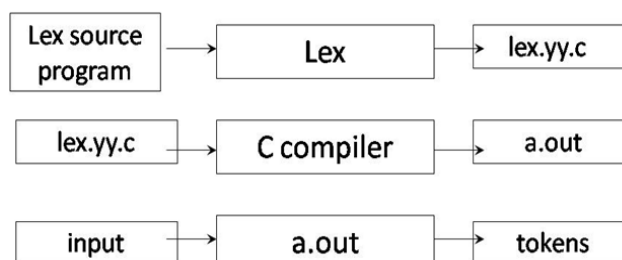
### ■ Lex Predefined Variables

- yytext -- A string containing the lexeme
- yyleng -- The length of the lexeme
- yyin -- The input stream pointer. The default input of default main() is stdin
- yyout -- The output stream pointer. The default output of default main() is stdout.

### ■ Lex Library Routines

- yylex() : The default main() contains a call of yylex()
- yymore(): Return the next token
- yyless(n): Retain the first n characters in yytext
- yywarp(): Called whenever Lex reaches an end-of-file. The default yywarp() always returns 1

### Compilation & Execution Process Of Lex



**Save a file:** example.l  
**Compile:** lex example.l  
gcc -o example lex.yy.c  
**Run:** ./example

**EXAMPLE:** To create a lex program that converts uppercase to lowercase, removes blanks at the end of a line, and replaces multiple blanks by single blanks.

%%

[A-Z] putchar(yytext[0]+'a'-'A');

[ ]+\$ ;

[ ]+ putchar(' ');

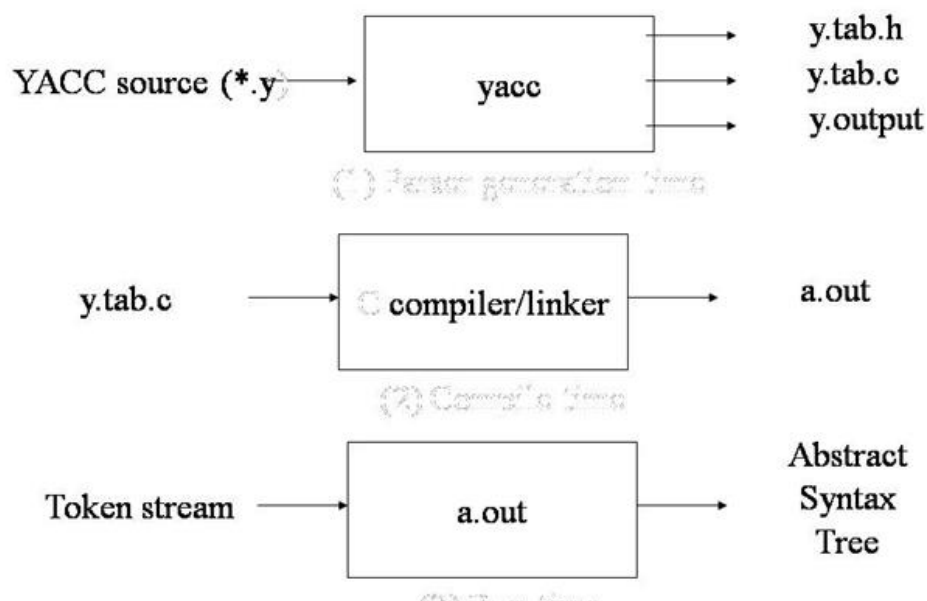
## YACC: Yet Another Compiler-Compiler

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

### YACC Format

```
% {  
    C declarations  
% }  
    yacc declarations  
%%  
    Grammar rules  
%%  
    Additional C code
```

### Compilation & Execution Process Of Yacc



**Compile:** yacc example.y

**Execution:** ./example

## YACC Declaration Summary

`%start': Specify the grammar's start symbol

`%union'

Declare the collection of data types that semantic values may have

`%token'

Declare a terminal symbol (token type name) with no precedence or associativity specified

`%type'

Declare the type of semantic values for a nonterminal symbol

`%right'

Declare a terminal symbol (token type name) that is right-associative

`%left'

Declare a terminal symbol (token type name) that is left-associative

`%nonassoc'

Declare a terminal symbol (token type name) that is non-associative (using it in a way that would be associative is a syntax error)

## YACC Example

```
%{
#include <stdio.h>
%}

%token NAME NUMBER
%%

statement: NAME '=' expression
          | expression          { printf("= %d\n", $1); }
          ;

expression: expression '+' NUMBER { $$ = $1 + $3; }
           | expression '-' NUMBER { $$ = $1 - $3; }
           | NUMBER                { $$ = $1; }
           ;

%%

int yyerror(char *s)
{
    fprintf(stderr, "%s\n", s);
    return 0;
}

int main(void)
{
    yyparse();
    return 0;
}
```

## Conclusion:

Implementation of Lex and Yacc requires a solid understanding of the theoretical foundations, familiarity with the syntax, practice with real-world examples, debugging and troubleshooting skills, optimization techniques, error handling and recovery mechanisms, and exploration of advanced topics. Utilizing documentation and available resources, as well as gaining practical experience through implementation and experimentation, will enable you to effectively utilize Lex and Yacc to build robust compilers or interpreters for programming languages. With dedication, practice, and a thorough understanding of these tools, you can become proficient in using Lex and Yacc to develop powerful language processing systems.

## // Expt. 8 LEX and YACC Implement

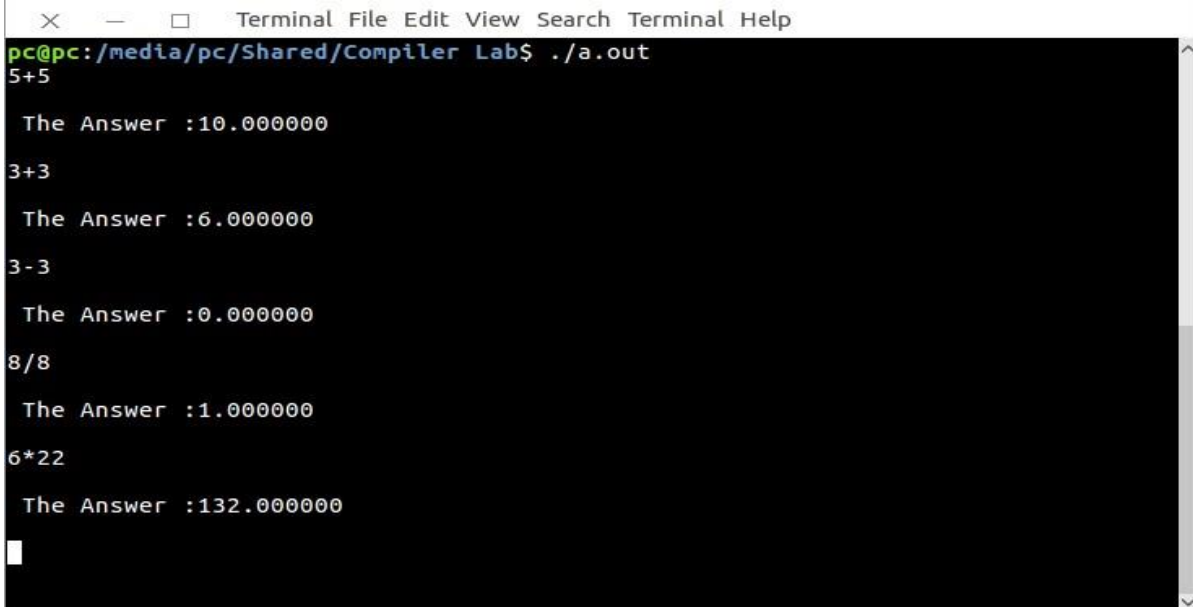
/\*8.1 lex program to implement - a simple calculator.\*/

```
% {  
    int op = 0,i;  
    float a, b;  
% }  
dig [0-9]+|([0-9]*)."([0-9]+)  
add "+"  
sub "-"  
mul "*"  
div "/"  
pow "^"  
ln "\n"  
%%  
/* digi() is a user defined function */  
{dig} {digi();}  
{add} {op=1;}  
{sub} {op=2;}  
{mul} {op=3;}  
{div} {op=4;}  
{pow} {op=5;}  
{ln} {printf("\n The Answer :%f\n\n",a);}  
%%  
digi()  
{  
if(op==0)  
  
/* atof() is used to convert  
- the ASCII input to float */  
a=atof(yytext);  
else  
{  
b=atof(yytext);  
switch(op)  
{  
case 1:a=a+b;  
break;  
case 2:a=a-b;  
break;  
case 3:a=a*b;  
break;  
case 4:a=a/b;  
break;  
case 5:for(i=a;b>1;b--)  
a=a*i;  
break;  
}  
op=0;  
}  
}  
main(int argv,char *argc[])  
{  
yylex();  
}  
yywrap()
```



```
{  
return 1;  
}
```

**// Output:**



```
pc@pc:/media/pc/Shared/Compiler Lab$ ./a.out  
5+5  
The Answer :10.000000  
3+3  
The Answer :6.000000  
3-3  
The Answer :0.000000  
8/8  
The Answer :1.000000  
6*22  
The Answer :132.000000
```

## // Expt. 8 LEX and YACC Implement

/\* 8.2 lex program to implement *to validate a character in c\**/

### // USING LEX

```
%{
/* To validate a character in C using lex*/
%}

%%
[\\t]+      ;
[0-9]+      {printf("This is a valid number\\n");}
int|char|float|if|else {printf("%s- is a keyword \\n",yytext);}
[a-zA-Z]+   {printf("%s- is a character\\n",yytext);}
"++"|"--"|"=="|["+*/] {printf("%s- is an operator\\n",yytext);}
[(){}.;"]{printf("%s- is a special symbol\\n",yytext);}
.           ECHO;
%%
main()
{
    yylex();
}
int yywrap()
{
    return(1);
}
```

-----**OUTPUT**-----

[root@localhost Desktop]# lex 38spcc2.l

[root@localhost Desktop]# gcc lex.yy.c

[root@localhost Desktop]# ./a.out

int a=10;

int- is a keyword

a- is a character

=- is a special symbol

This is a valid number

;- is a special symbol

char b;float a=90;

char- is a keyword

b- is a character

;- is a special symbol

float- is a keyword

a- is a character

=- is a special symbol

This is a valid number

;- is a special symbol

## // Expt. 8 LEX and YACC Implement

*/\* 8.3 Parser Using Lex And Yacc \*/*

*// LEX FILE*

```
%{
#include "y.tab.h"
extern int yylval;
}%

%%
[\t]+    ;
[0-9]+   {yylval=atoi(yytext); return(num);}
\n       {return 0;}
.        {return(yytext[0]);}
%%
```

```
int yywrap()
{
return 1;
}
```

*// YACC FILE*

```
%{
#include<stdio.h>
#include<stdlib.h>
}%

%start s
%token num
%left '+' '-'
%left '*' '/'
%%

s:E      {printf("Result=%d\n",$1);};
E:E'+E' {$$=$1+$3;}
|E'-E' {$$=$1-$3;}
|E'*E' {$$=$1*$3;}
|E'/E' {$$=$1/$3;}
|num {$$=$1;}
%%
```

```
main()
{
printf("Enter the expression:\n");
yyparse();
}
int yyerror()
{
return(1);
}
```

-----OUTPUT-----

```
[root@localhost Desktop]# yacc -d 38.y  
[root@localhost Desktop]# lex spexp3.l  
[root@localhost Desktop]# cc -o abc lex.yy.c y.tab.c -ll
```

```
[root@localhost Desktop]# ./abc
```

Enter the expression:

4+6-7+8

Result=11

```
[root@localhost Desktop]# ./abc
```

Enter the expression:

8\*9/3

Result=24

```
[root@localhost Desktop]# ./abc
```

Enter the expression:

6\*5-7

Result=23