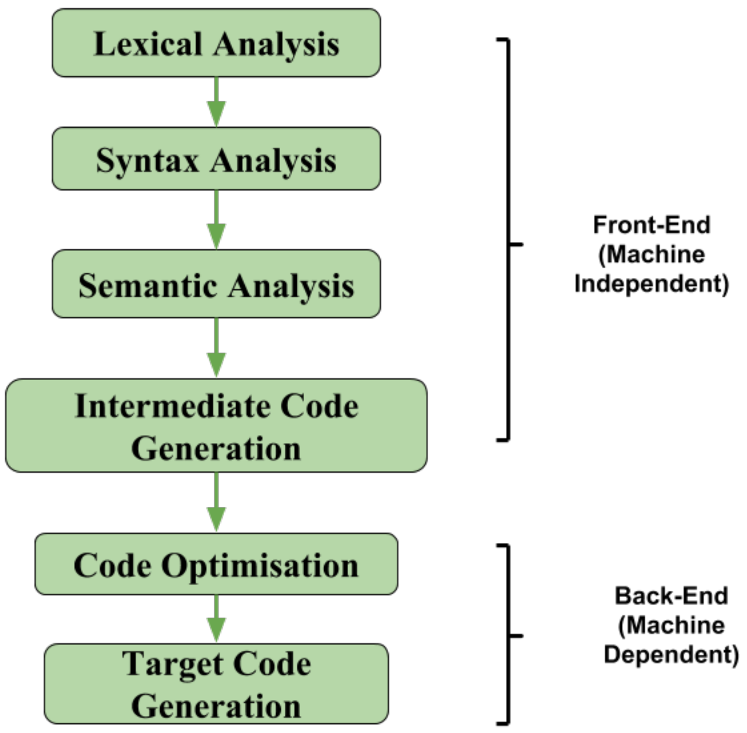
BHARATIYA VIDYA BHAVAN'S
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Empowered Autonomous Institute Affiliated to University of Mumbai)
[Knowledge is Nectar]

Department of Computer Engineering

e

Course - System Programming and Compiler Construction (SPCC)

UID	2021300126
Name	Pranay Singhvi
Class and Batch	TE Computer Engineering - Batch C
Date	19-03-2024
Lab #	5
Aim	Develop a Python program to convert postfix expressions to quadruples and display them in a visually appealing table format.
Objective	Develop a Python program to parse postfix expressions, generating quadruples to represent operations, and utilize the PrettyTable library for presenting the quadruples in a visually structured format, enhancing readability and understanding of expression evaluation processes.
Theory	**Intermediate Code Generation in Compiler Design** In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine). The benefits of using machine-independent intermediate code are: • Because of the machine-independent intermediate code, portability will be enhanced. For ex, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications. • Retargeting is facilitated. • It is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.


Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Code Optimisation
Target Code Generation
Front-End (Machine Independent)
Back-End (Machine Dependent)

If we generate machine code directly from source code then for n target machine we will have optimizers and n code generator but if we will have a machine-independent intermediate code, we will have only one optimizer. Intermediate code can be either language-specific (e.g., Bytecode for Java) or language. independent (three-address code). The following are commonly used intermediate code representations:

1. Postfix Notation:

   - Also known as reverse Polish notation or suffix notation.
   - In the infix notation, the operator is placed between operands, e.g., *a* + *b*. Postfix notation positions the operator at the right end, as in *ab* +.
   - For any postfix expressions *e1* and *e2* with a binary operator *(+)* , applying the operator yields *e1e2*+.
   - Postfix notation eliminates the need for parentheses, as the operator's position and arity allow unambiguous expression decoding.
   - In postfix notation, the operator consistently follows the operand.
     Example 1: The postfix representation of the expression (a + b) * c is : ab + c *

Example 2: The postfix representation of the expression (a – b) * (c + d) + (a – b) is : ab – cd + *ab -+

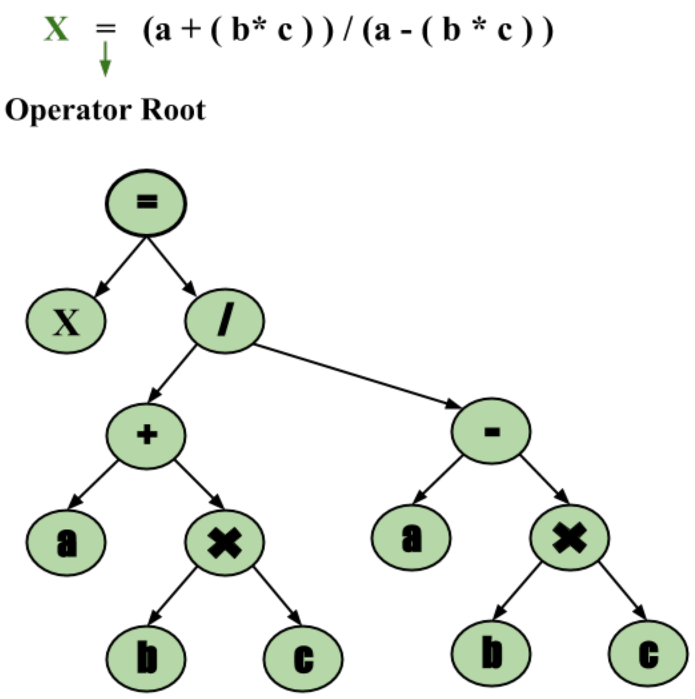
2. **Three-Address Code:**

- A three address statement involves a maximum of three references, consisting of two for operands and one for the result.
- A sequence of three address statements collectively forms a three address code.
- The typical form of a three address statement is expressed as *x = y op z*, where *x, y*, and *z* represent memory addresses.
- Each variable *(x, y, z)* in a three address statement is associated with a specific memory location.
- While a standard three address statement includes three references, there are instances where a statement may contain fewer than three references, yet it is still categorized as a three address statement.
  Example: The three address code for the expression a + b * c + d : T1 = b * c T2 = a + T1 T3 = T2 + d; T 1 , T2 , T3 are temporary variables.

There are 3 ways to represent a Three-Address Code in compiler design:
i) Quadruples
ii) Triples
iii) Indirect Triples

3. **Syntax Tree:**

- A syntax tree serves as a condensed representation of a parse tree.
- The operator and keyword nodes present in the parse tree undergo a relocation process to become part of their respective parent nodes in the syntax tree. the internal nodes are operators and child nodes are operands.
- Creating a syntax tree involves strategically placing parentheses within the expression. This technique contributes to a more intuitive representation, making it easier to discern the sequence in which operands should be processed.
- The syntax tree not only condenses the parse tree but also offers an improved visual representation of the program's syntactic structure,
  Example: x = (a + b * c) / (a – b * c)



Advantages of Intermediate Code Generation:

Easier to implement: Intermediate code generation can simplify the code generation process by reducing the complexity of the input code, making it easier to implement.

Facilitates code optimization: Intermediate code generation can enable the use of various code optimization techniques, leading to improved performance and efficiency of the generated code.

Platform independence: Intermediate code is platform-independent, meaning that it can be translated into machine code or bytecode for any platform.

Code reuse: Intermediate code can be reused in the future to generate code for other platforms or languages.

Easier debugging: Intermediate code can be easier to debug than machine code or bytecode, as it is closer to the original source code.

Disadvantages of Intermediate Code Generation:

Increased compilation time: Intermediate code generation can significantly increase the compilation time, making it less suitable for real-time or time-critical applications.

Additional memory usage: Intermediate code generation requires additional memory to store the intermediate representation, which can be a concern for memory-limited systems.

Increased complexity: Intermediate code generation can increase the complexity of the compiler design, making it harder to implement and maintain.

Reduced performance: The process of generating intermediate code can result in code that executes slower than code generated directly from the source code.

Three address code in Compiler

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

1. Quadruple – It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantage –

- Easy to rearrange code for global optimization.
- One can quickly access value of temporary variables using symbol table.

Disadvantage –

- Contain lot of temporaries.
- Temporary variable creation increases time and space complexity.

2. Triples – This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage –

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

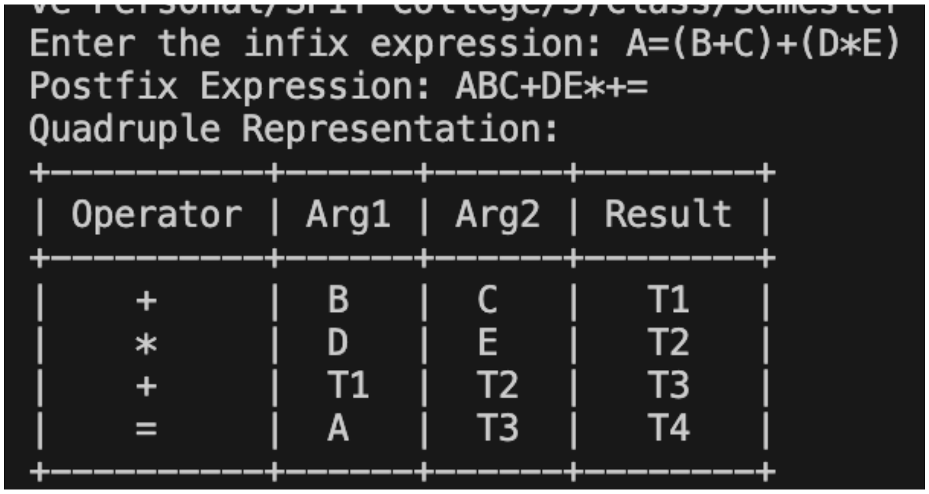
3. Indirect Triples – This representation makes use of pointer to the listing of all references to computations which is made separately and stored. Its similar in utility as compared to

	quadruple representation but requires less space than it. Temporaries are implicit and easier to rearrange code.

Implementation/ Code

```
from prettytable import PrettyTable

def is_operator(char):
    return char in ['+', '-', '*', '/', '^', '=']

def precedence(op):
    precedence_dict = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    return precedence_dict.get(op, 0)

def infix_to_postfix(infix_exp):
    postfix_exp = []
    stack = []
    for char in infix_exp:
        if char.isalnum():  # Operand
            postfix_exp.append(char)
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack and stack[-1] != '(':
                postfix_exp.append(stack.pop())
            stack.pop()  # Discard '('
        else:  # Operator
            while stack and precedence(stack[-1]) >= precedence(char):
                postfix_exp.append(stack.pop())
            stack.append(char)

    while stack:
        postfix_exp.append(stack.pop())

    return ''.join(postfix_exp)

def display_quadruple(quadruples):
    table = PrettyTable()
    table.field_names = ["Operator", "Arg1", "Arg2", "Result"]
    for quadruple in quadruples:
        table.add_row(quadruple)
    print("Quadruple Representation:")
    print(table)

def postfix_to_quadruple(exp):
    stack = []
    quadruples = []
    temp = 1
    for char in exp:
        if char.isalnum():
            stack.append(char)
        elif is_operator(char):
            op2 = stack.pop()
            op1 = stack.pop()
            temp_var = "T" + str(temp)
            temp += 1
            quadruples.append([char, op1, op2, temp_var])
            stack.append(temp_var)
    return quadruples

def main():
    infix_exp = input("Enter the infix expression: ")
    postfix_exp = infix_to_postfix(infix_exp)
    print("Postfix Expression:", postfix_exp)

    quadruples = postfix_to_quadruple(postfix_exp)
    display_quadruple(quadruples)

if __name__ == "__main__":
    main()
```

Output



```
ve Personal/SPIT College/3/class/Semester
Enter the infix expression: A=(B+C)+(D*E)
Postfix Expression: ABC+DE*+=
Quadruple Representation:
+----------+------+------+--------+
| Operator | Arg1 | Arg2 | Result |
+----------+------+------+--------+
|    +     |  B   |  C   |   T1   |
|    *     |  D   |  E   |   T2   |
|    +     |  T1  |  T2  |   T3   |
|    =     |  A   |  T3  |   T4   |
+----------+------+------+--------+
```

Conclusion

In conclusion, I successfully implemented a postfix expression parser in Python, generating quadruples.

References

[1] Geeksforgeeks. (April 6, 2023). Three address code in Compiler
https://www.geeksforgeeks.org/three-address-code-compiler/

[2]] Geeksforgeeks. (January 19, 2024). Intermediate Code Generation in Compiler Design
https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/