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

Department of Computer Engineering

Course - System Programming and Compiler Construction (SPCC)

UID	2021300101
Name	Adwait Purao
Class and Batch	TE Computer Engineering - Batch B
Date	15/03/24
Lab #	5
Aim	Intermediate Code Generation - ICG
Objective	This experiment seeks to illustrate the translation of a Postfix expression into Intermediate Code Generation (ICG) utilizing Quadruple format. It employs a stack data structure for efficient operand management. By parsing the Postfix expression and generating Quadruples for each operator through operand retrieval from the stack, the experiment aims to systematically organize these Quadruples into a tabular format. This endeavor applies fundamental data structure concepts and ICG techniques, offering an intermediate representation of the given expression. Such a process underscores the conversion of high-level language constructs into an intermediary format that serves as a bridge to machine-level instructions. It underscores the practical application of compilation principles, offering insights into the transformation process without the actual execution of the expression.
Theory	**Intermediate Code Generation (ICG)** is a critical stage in compiler construction [1]. It acts as a bridge between the high-level source code written in a programming language and the low-level machine code understood by the computer [1]. During this phase, the compiler translates the source code into an intermediate representation (IR) that is closer to machine code but remains abstract enough to enable optimizations and function across different platforms [1]. **Importance of Intermediate Code Generation:** ● **Optimization:** ICG serves as a foundation for various optimization techniques. These techniques, like constant folding, common subexpression elimination, and dead code elimination, improve the efficiency and performance of the final code [1]. ● **Platform Independence:** By generating an IR, compilers can target multiple platforms without needing to rewrite code specific to each platform in the earlier stages [1]. This simplifies compiler development and maintenance. ● **Simplified Compiler Design:** Working with an IR simplifies the implementation of subsequent compiler phases, such as code optimization and generation [1].

Techniques of Intermediate Code Generation:

1. **Three-Address Code:** This common IR format limits each instruction to at most three operands, simplifying analysis and optimization [2].
2. **Quadruples:** These represent program operations with four fields: operator, operand1, operand2, and result. They offer a compact and efficient way to represent program operations for optimization and code generation [2].
3. **Abstract Syntax Trees (ASTs):** Hierarchical representations of the source code's syntactic structure, ASTs can be traversed to directly generate IR or converted into other IR formats [2].
4. **Control Flow Graphs (CFGs):** Representing the flow of control within a program, CFGs are used to generate IR and perform optimizations like loop optimization and dead code elimination [2].

Challenges in Intermediate Code Generation:

1. **Semantic Analysis:** Ensuring the generated IR accurately reflects the source code's semantics, including type checking and error detection [2].
2. **Optimization Trade-offs:** Balancing the benefits of optimization with compilation time. Aggressive optimization techniques can significantly increase compilation time [2].
3. **Platform-Specific Considerations:** Generating platform-independent IR while considering the target architecture's specific features and potential optimizations [2].
4. **Debugging and Error Reporting:** Providing clear error messages and debugging information based on the IR to aid developers in identifying and resolving issues in their code [2].

Quadruples in Intermediate Code Generation

Structure of Quadruples: A quadruple is a data structure with four fields:

1. **Operator:** Represents the operation to be performed (arithmetic operations, assignment, conditional jumps, etc.) [2].
2. **Operand1:** The first operand involved in the operation.
3. **Operand2:** The second operand involved in the operation (if applicable).
4. **Result:** The outcome of the operation.

Significance of Quadruples:

1. **Compact Representation:** Quadruples offer a space-efficient way to represent program operations compared to other IRs [2].
2. **Simplicity:** Their straightforward structure makes them easy to manipulate and analyze in subsequent compiler phases [2].
3. **Flexibility:** Quadruples can represent various operations, including arithmetic, logical, and control flow, making them versatile for various compiler tasks [2].
4. **Ease of Optimization:** The clear representation of program operations and dependencies in quadruples facilitates optimization techniques like common subexpression elimination, dead code elimination, and constant folding [2].

Converting a Postfix Expression to Quadruples:

1. **Read the Expression:** Begin by reading the postfix expression from left to right.
2. **Operand or Operator:** Process each character:
    - For operands (variables or constants), push them onto a stack.
    - For operators (+, -, *, /, =):
        - Pop the top two operands from the stack.
        - Arrange them with the operator and a temporary variable (T1, T2, etc.) to store the result in a quadruple format.
        - Add this quadruple to a table.
3. **Processing the Expression:** Here's an example for the expression "a b c + d e - *=":
    - Push "a", "b", and "c" onto the stack.
    - Encountering "+" pops "c" and "b", arranges them with "+" as the operator, and stores the result in "T1".
    - Continue for remaining operators and operands until processing the entire expression.

Implementation / Code

```
from prettytable import PrettyTable

print("Enter the postfix expression: ", end="")
postfix_expression = str(input())
operand_stack = []
temp_index = 0
quadruples_table = PrettyTable(["Operator", "Operand1", "Operand2",
"Result"])

for token in postfix_expression:
    if token.isalpha():
        operand_stack.append(token)
    else:
        operand2 = operand_stack.pop()
        operand1 = operand_stack.pop()
        temp_index += 1
        temp_result = "t" + str(temp_index)
        operand_stack.append(temp_result)
        quadruples_table.add_row([token, operand1, operand2,
temp_result])

print("Operand stack:", operand_stack)
print(quadruples_table)
```

Output

```
aspur@LAPTOP-LG4IQEFB MINGW64 ~/OneDrive/SPCC/EXPERIMENTS/05. ICG
$ python icg.py
Enter the postfix expression: abc+de-*=
Operand stack: ['t4']
+----------+----------+----------+--------+
| Operator | Operand1 | Operand2 | Result |
+----------+----------+----------+--------+
|    +     |    b     |    c     |   t1   |
|    -     |    d     |    e     |   t2   |
|    *     |    t1    |    t2    |   t3   |
|    =     |    a     |    t3    |   t4   |
+----------+----------+----------+--------+
```

Conclusion

The experiment exemplified the conversion of Postfix expressions into Quadruple format employing stacks, emphasizing significant aspects of compiler design. It underscored the pivotal role of data structures in facilitating the compilation process and establishing an efficient linkage between high-level constructs and machine-level code.

References

[1]Aho, A. V., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, techniques, and tools (2nd ed.). Addison-Wesley.

[2] Muchnick, Steven S. Advanced Compiler Design and Implementation. San Francisco, Calif, Morgan Kaufmann, 2014.