College of Engineering Trivandrum

Compiler Design Lab

Abhishek Manoharan
S7 CSE Roll No:2

TVE17CS002

Department of Computer Science

November 20, 2020

Exp 13

1 Intermediate Code Generation

1.1 Aim

Implement Intermediate code generation for simple expressions

1.2 Theory

Intermediate Code Generation.

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).

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 representation:

1. Postfix Notation.
2. Three-Address Code.
3. Syntax Tree.

Three address code.

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code.It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.
A statement involving no more than three references(two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement.
General representation –

$$a = b\ op\ c$$

Where a, b or c represents operands like names, constants or compiler generated temporaries and op represents the operator.
Example – The three address code for the expression a + b * c + d :
$T_1 = b * c$
$T_2 = a + T_1$
$T_3 = T_2 + d$

T_1, T_2, T_3 are temporary variables.

1.3 Algorithm

Algorithm 1: Algorithm for 3 address code generation

```
1. while there are still tokens to be read in,
   1.1 Get the next token.
   1.2 if the token is:
       1.2.1 A Variable: push it onto the value stack.
       1.2.2 A left parenthesis: push it onto the operator stack.
       1.2.3 A right parenthesis:
         1 while the thing on top of the operator stack is not a
           left parenthesis,
             1 Pop the operator from the operator stack.
             2 Pop the value stack twice, getting two operands.
             3 Apply the operator to the operands, in the correct order and print.
             4 Push the temporary variable onto the value stack.
         2 Pop the left parenthesis from the operator stack, and discard it.
       1.2.4 An operator (call it thisOp):
         1 while the operator stack is not empty, and the top thing on the
           operator stack has the same or greater precedence as thisOp,
           1 Pop the operator from the operator stack.
           2 Pop the value stack twice, getting two operands.
           3 Apply the operator to the operands, in the correct order and print.
           4 Push the temporary variable onto the value stack.
         2 Push thisOp onto the operator stack.
2. while the operator stack is not empty,
    1 Pop the operator from the operator stack.
    2 Pop the value stack twice, getting two operands.
    3 Apply the operator to the operands, in the correct order and print.
    4 Push the temporary variable onto the value stack.
3. At this point the operator stack should be empty, and the value
   stack should have only one value in it, assign it to the LHS of = variable.
```

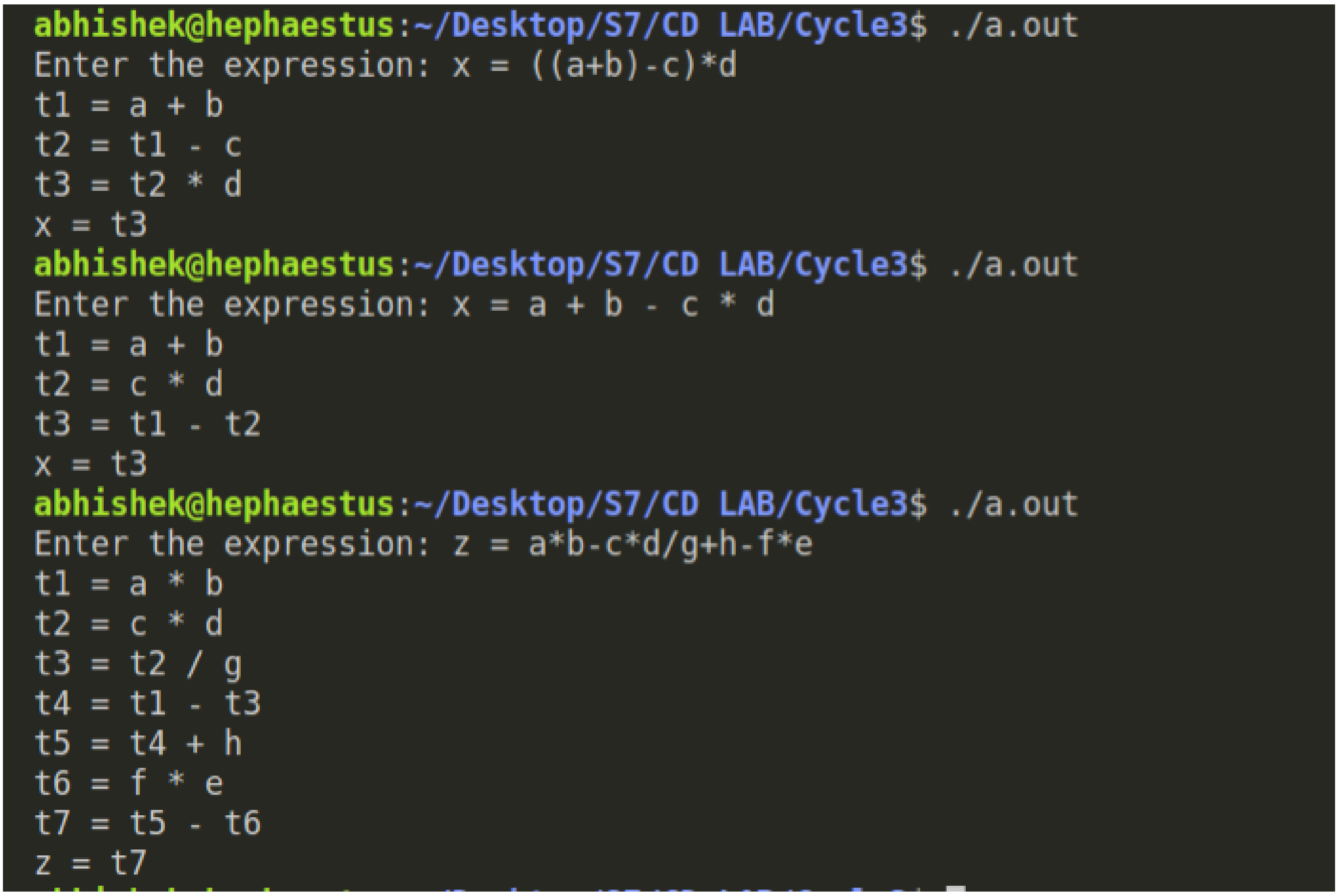
1.4 Code

```cpp
#include <bits/stdc++.h>
using namespace std;
int precedence(char a)
{
    if (a == '+' || a == '-')
    {
        return 0;
    }
    if (a == '(')
        return -1;
    return 1;
}
bool isop(char s)
{
    if (s == '+' || s == '-' || s == '*' || s == '/')
    {
        return true;
    }
    else
    {
        return false;
    }
}
void print_star(int n)
{
    for (int i = 0; i < n; ++i)
    {
        cout << "*";
    }
    cout << endl;
}
string charint(char s)
{
    string res = "";
    if (s >= '1' && s <= '9')
    {
        res += 't';
        res += s;
        return res;
    }
    return res + s;
}
int main()
{
    string s;
    cout << "Enter the expression: ";
    getline(cin, s);
    vector<char> input;
    int len = s.size();
    int start = 0;
    while (s[start] != '=')
    {
        start++;
    }
    for (int i = start + 1; i < len; ++i)
    {
        if (s[i] == ' ')
            continue;
        input.push_back(s[i]);
    }
    // for (auto x : input)
    // {
    //     cout << x;
    // }
    char count = '1';
    stack<char> value;
    stack<char> op;
    for (int i = 0; i < input.size(); ++i)
    {
        //cout << input[i] << "current reading" << endl;
        //cout << isop(input[i]);
        if (isalpha(input[i]))
        {
            //cout << input[i] << " pushed into the stack" << endl;
            value.push(input[i]);
        }
        else if (input[i] == '(')
        {
            op.push(input[i]);
        }

        else if (isop(input[i]))
        {
            //cout << "operant found: " << input[i] << endl;
            while (!op.empty() && precedence(op.top()) >= precedence(input[i]))
            {
                char a1, a2, o1;
                a2 = value.top();
                value.pop();
                a1 = value.top();
                value.pop();
                o1 = op.top();
                op.pop();
                string b1, b2;
                b1 = charint(a1);
                b2 = charint(a2);
                cout << "t" << count << " = " << b1 << " " << o1 << " " << b2 << endl;
                value.push(count);
                count++;
            }
            op.push(input[i]);
        }
        else
        { // closing bracket present
            //cout << "closing bracket found " << endl;
            while (!op.empty() && op.top() != '(')
            {
                char a1, a2, o1;
                a2 = value.top();
                value.pop();
                a1 = value.top();
                value.pop();
                o1 = op.top();
                op.pop();
                string b1, b2;
                b1 = charint(a1);
                b2 = charint(a2);
                cout << "t" << count << " = " << b1 << " " << o1 << " " << b2 << endl;
                value.push(count);
                count++;
            }
            op.pop();
        }
    }
    while (!op.empty())
    {
        char a1, a2, o1;
        a2 = value.top();
        value.pop();
        a1 = value.top();
        value.pop();
        o1 = op.top();
        op.pop();
        string b1, b2;
        b1 = charint(a1);
        b2 = charint(a2);
        cout << "t" << count << " = " << b1 << " " << o1 << " " << b2 << endl;
        value.push(count);
        count++;
    }
    cout << s[0] << " = " << charint(count - 1) << endl;
    return 0;
}
```

Code for 3 address code generation

1.5 Output



```
abhishek@hephaestus:~/Desktop/S7/CD LAB/Cycle3$ ./a.out
Enter the expression: x = ((a+b)-c)*d
t1 = a + b
t2 = t1 - c
t3 = t2 * d
x = t3
abhishek@hephaestus:~/Desktop/S7/CD LAB/Cycle3$ ./a.out
Enter the expression: x = a + b - c * d
t1 = a + b
t2 = c * d
t3 = t1 - t2
x = t3
abhishek@hephaestus:~/Desktop/S7/CD LAB/Cycle3$ ./a.out
Enter the expression: z = a*b-c*d/g+h-f*e
t1 = a * b
t2 = c * d
t3 = t2 / g
t4 = t1 - t3
t5 = t4 + h
t6 = f * e
t7 = t5 - t6
z = t7
```

1.6 Result

Implemented the program for Intermediate code generation(3 Address code). It was compiled using g++ version 9.3.0, and executed in Ubuntu 20.04 and the above output was obtained.