CS301P Compiler Design Laboratory Exercises #6

Date: October 07 2024

Objectives

- To implement a parser to verify arithmetic expressions.
- To deal with error handling and error reporting mechanisms using *error* token.
- To add functionality of pre/post increment/decrement and compound assignment operators (+=, -=, *=, /=) operators
- To learn construction of syntax tree (one of the intermediate representations).
- Convert the given syntax tree to three address code

Exercise Problems

1. Extend the Lab #5, to include the pre/post increment and decrement operators $++, --$, and compound assignment operators $+=, -=, *=, /=$.

2. Construct a parser to verify the given arithmetic expressions and accept all valid arithmetic expressions, and reject the invalid expressions by providing appropriate error diagnostic messages.

3. Construct a syntax tree by adding semantic actions, and then conver the same to three address code intermediate form.

Sample Input and expected Output:

result = a++ b;	→	Rejected - missing operator
result = x++ + ++y;	→	Accepted
result = m−− * n;	→	Accepted
result = ++a + b−−;	→	Accepted
result = −−x - ++y;	→	Accepted
result = i++ + −−j - k−−;	→	Accepted
result = a++ + b++ * a;	→	Accepted
result = −−x / −−y;	→	Accepted
result = (++p * q−−) + (−−q - p++);	→	Accepted
result = 10++;	→	Rejected - cannot increment a constant value
result = a + ++b−−;	→	Rejected - expression is not assignable
result = a + ++b−;	→	Rejected - missing operand

Note

- Three Addree Code (a simple concept, to be discussed in detail in the class). However, essential details are provided at the end.
- In case you are finding difficulty, consider a limited set of functionality, then come up with a CFG and the corresponding parser.
- Error handing is important.
- All other details including submission guidelines are same as the previous lab.

Three Address Code (TAC) Essential Details

Three-Address Code (TAC) is an intermediate representation used in compilers for generating machine or assembly code. It's called "three-address" code because each instruction typically involves at most three operands. TAC simplifies complex expressions and statements into sequences of simple operations, making it easier for the compiler to generate target machine code.
Key Features of Three-Address Code:

1. Simple Instructions:
    - Each instruction performs a simple operation (e.g., addition, subtraction, assignment) on two operands, storing the result in a third operand. The general format is: x = y op z
      Where:
        - 'x' is the result.
        - 'y' and 'z' are operands.
        - 'op' is a binary operator (like '+', '-', '*', '/').
2. Use of Temporary Variables:
    - TAC uses **temporary variables** to hold intermediate values during the evaluation of complex expressions. For example, in an expression like 'a + b * c', a temporary variable might first store the result of 'b * c', followed by adding that to 'a'.
3. Types of Instructions: TAC typically includes the following types of instructions:
    - Assignment: 'x = y'
    - Binary operations: 'x = y op z' (e.g., 'x = a + b')
    - Unary operations: 'x = op y' (e.g., 'x = -y')
    - jumps: 'if condition goto label'
    - Unconditional jumps: 'goto label'
    - Function calls: 'call function_name'
    - Array indexing: 'x = a[i]'
4. Handling of Complex Expressions:
    - Complex expressions involving multiple operators, increments, and function calls are broken down into simpler statements using temporary variables. For example, an expression like 'a++ + ++b' requires multiple instructions to handle both the post-increment and pre-increment before performing the addition.

Why is TAC Important?

- Simplifies Expression Evaluation: TAC breaks down expressions into small, easily processed units, making it easier to generate efficient machine code.

- Easy Translation to Assembly: Since each TAC statement corresponds closely to a machine instruction, it's straightforward to translate TAC into low-level code.
- Optimization: TAC allows various optimizations such as constant folding, dead code elimination, and common subexpression elimination.

TAC vs. High-Level Code: In high-level languages like C or Java, complex expressions are written in a single line (e.g., 'r = a++ + ++b'), but during compilation, the process breaks down this line into multiple instructions using TAC. This helps the compiler handle each operation (like increments or arithmetic) in a step-by-step, controlled manner.

Example of TAC: Given the expression r = a++ + ++b, TAC translates it into simple operations like:

```
t1 = a          // Store current value of a in t1
a = a + 1       // Post-increment a
b = b + 1       // Pre-increment b
t2 = b          // Store updated value of b in t2
t3 = t1 + t2    // Add values of t1 and t2
r = t3          // Assign result to r
```

Summary:

TAC is an intermediate form of code that simplifies the transition from high-level code to machine code. It breaks down complex expressions into smaller, manageable instructions, utilizing temporary variables to store intermediate values. This makes TAC an essential tool for generating and optimizing assembly or machine code during the compilation process.

This detailed structure also enables various optimizations during compilation, which can improve the efficiency of the resulting machine code.