COMPILER CONSTRUCTION
(Intermediate Code: three address, control flow)

Prof. K R Chowdhary
Email: kr.chowdhary@jietjodhpur.ac.in

Campus Director, JIET, Jodhpur

Thursday 18th October, 2018

Three-Address Code

- In three-address code, there is at most one operator on the right side of an instruction; Thus a source-language expression like $x + y * z$ might be translated into the sequence of three-address instructions

$$t_1 = y + z$$
$$t_2 = x + t_1$$

- t_1 and t_2 are compiler-generated temporary names. This unraveling of multi-operator arithmetic expressions and of nested flow-of-control statements makes three-address code desirable for target-code generation and optimization.

Three-Address Code

Example

Dag for the expression $a + a * (b - c) + (b - c) * d$.

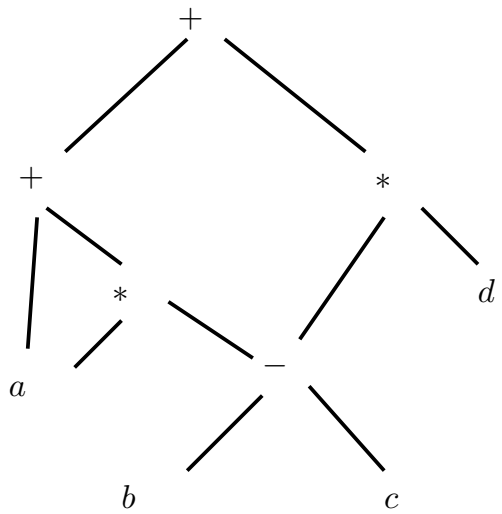

Figure 1: Dag for the expression $a + a * (b - c) + (b - c) * d$.

$$t_1 = b - c$$
$$t_2 = a * t_1$$
$$t_3 = a + t_2$$
$$t_4 = t_1 * d$$
$$t_5 = t_3 + t_4$$

Solution. Three-address code is a linearized representation of a syntax tree or a DAG in which explicit names correspond to the interior nodes of the graph.

Addresses and Instructions

Three-address code can be implemented using records with fields for the addresses; records called quadruples and triples next section. An address can be one of the following:

- *A name.* For convenience, we allow source-program names to appear as addresses in three-address code.
- *A constant.*
- *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. variables.

Addresses and Instructions

We will use three-address instructions. Symbolic labels used by instructions alter the flow of control.

1. Assignment instructions of the form $x = yopz$, where op is a binary arithmetic or logical operation, and x, y, and z are addresses.
2. Assignments of the form $x = op\ y$, where op is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert an integer to a floating-point number.
3. Copy instructions of the form $x = y$, where x is assigned the value of y.
4. An unconditional jump *goto* L. The three-address instruction with label L is the next to be executed.

Addresses and Instructions...

1. Conditional jumps of the form *if x goto L and if False x goto L*. These instructions execute the instruction with label L next if x is true and false, respectively. Otherwise, the following three-address instruction in sequence is executed next, as usual.
2. Conditional jumps such as *if x relop y goto L*, which apply a relational operator ($<, ==, >=$, etc.) to x and y, and execute the instruction with label L next if x stands in relation *relop* to y.
3. Procedure calls and returns are implemented using the following instructions: *param x* for parameters; *call* p, n and $y =$ *call* p, n for procedure and function calls, respectively; and *return y*, where y, representing a returned value, is optional.

Addresses and Instructions...

1. Their typical use is as the sequence of three- address instructions

$$
\begin{aligned}
&param\ x_1\\
&param\ x_2\\
&....\\
&param\ x_n\\
&call\ p, n
\end{aligned}
$$

generated as part of a call of the procedure $p(x_1, x_2, ..., x_n)$. The integer n, indicating the number of actual parameters in "$call\ p, n$," is not redundant because calls can be nested. That is, some of the first *param* statements could be parameters of a call that comes after p returns its value; that value becomes another parameter of the later call.

Addresses and Instructions...

Example

Find out the translation of the following statement:

$$do\ i = i + 1; while\ (a[i] < v);$$

Solution. One possible translation of this statement is shown below with symbolic labels.

$$\begin{aligned} L\ :\ & t_1 = i + 1 \\ & i = t_1 \\ & t_2 = i * 8 \\ & t_3 = a[t_2] \\ & if\ \ t_3 < v\ goto\ L \end{aligned}$$

The other translation with position number is shown below:

Addresses and Instructions...

$$
\begin{aligned}
100 \quad & t_1 = i + 1 \\
101 \quad & i = t_1 \\
102 \quad & t_2 = i * 8 \\
103 \quad & t_3 = a[t_2] \\
104 \quad & \textit{if } t_3 < v \textit{ goto } 100
\end{aligned}
$$

The choice of permissible operators is an important issue in the design of an intermediate code. The operator set must be rich enough to implement the all operations in the source language, and efficiently also.