7CS5A COMPILER CONSTRUCTION (TAC, triples, quadruples)

Lecture 7: Oct. 18, 2018

Instructor: K.R. Chowdhary *: Professor of CS*

7.1 Three-Address Code

In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted. Thus a source-language expression like $x + y * z$ might be translated into the sequence of three-address instructions

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

where 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, as discussed in next lectures. The use of names for the intermediate values computed by a program allows three-address code to be rearranged easily.

Example 7.1 *Dag for the expression* $a + a * (b - c) + (b - c) * d$.

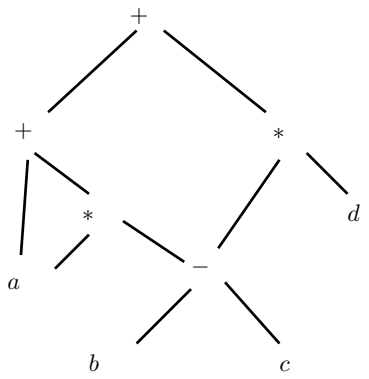

Figure 7.1: Dag for the expression $a + a * (b - c) + (b - c) * d$.

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. The DAG in Fig. 7.1, together with a corresponding three-address code sequence. □

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

Figure 7.2: Three address code for DAG in Fig. 7.1.

7.1.1 Addresses and Instructions

Three-address code is built from two concepts: addresses and instructions. In object-oriented terms, these concepts correspond to classes, and the various kinds of addresses and instructions correspond to appropriate subclasses. Alternatively, three-address code can be implemented using records with fields for the addresses; records called quadruples and triples are discussed briefly in 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. In an implementation, a source name is replaced by a pointer to its symbol-table entry, where all information about the name is kept.
- *A constant.* In practice, a compiler must deal with many different types of constants and variables. Type conversions within expressions are considered in next section.
- *A compiler-generated temporary.* It is useful, especially in optimizing compilers, to create a distinct name each time a temporary is needed. These temporaries can be combined, if possible, when registers are allocated to variables.

We now consider the common three-address instructions used in the rest of this text. Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a three-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by "back-patching," discussed later in this text. Here is a list of the common three-address instruction forms:

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 $x = opy$, 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.
5. 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.

6. 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. If not, the three-address instruction following $if\ x\ relop\ y\ goto\ L$ is executed next, in sequence.

7. 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. 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 $p(x_1, x_2 2, ..., x_n)$. 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. The implementation of procedure calls is outlined later on.

8. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$. The instruction $x = y[i]$ sets x to the value in the location i memory units beyond location y. The instruction $x[i] = y$ sets the contents of the location i units beyond x to the value of y.

9. Address and pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$. The instruction $x = \&y$ sets the r-value of x to be the location (I-value) of y.2 Presumably y is a name, perhaps a temporary, that denotes an expression with an bvalue such as A [il [jl , and x is a pointer name or temporary. In the instruction x = * y, presumably y is a pointer or a temporary whose r-value is a location. The r-value of x is made equal to the contents of that location. Finally, * x = y sets the r-value of the object pointed to by x to the r-value of y.

Example 7.2 *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:

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

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. Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine. However, if the front-end is required to generate long sequences of instructions for some source-language operations, then the optimizer and code generator may have to work harder to rediscover the structure and generate good code for these operations. □

7.2 Quadruples

The three-address instructions discussed above specifies the components of each type of instruction, but it does not specify the representation of these instructions in some data structure. In a compiler, these instructions can be implemented as objects or as records with fields for the operator and the operands. Three such representations are *quadruples*, *triples*, and *indirect triples*.

A quadruple (or just "quad") has four fields: *op*, arg_1, arg_2, and *result*. The op field contains an internal code for the operator. For instance, the three-address instruction $x = y + x$ is represented by placing + in *op*, y in arg_1, z in arg_2, and x in *result*. The following are some exceptions to this rule:

1. Instructions with unary operators like $x = minus \; y$ or $x = y$ do not use arg_2. Note that for a copy statement like $x = y$, *op* is =, while for most other operations, the assignment operator is implied.

2. Operators like *param* use neither arg_2 nor result.

3. Conditional and unconditional jumps put the target label in result.

Example 7.3 *Find out the three-address code for the assignment* $a = b * -c + b * -c$;

Solution. The special operator minus is used to distinguish the unary minus operator, as in $-c$, from the binary minus operator, as in $b - c$. Note that the unary-minus "three-address" statement has only two addresses, as does the copy statement $a = t_5$.

The following is three address code.

$$
\begin{aligned}
t_1 &= minus\ c \\
t_2 &= b * t_1 \\
t_3 &= minus\ c \\
t_4 &= b * t_3 \\
t_5 &= t_2 + t_4 \\
a &= t_5
\end{aligned}
$$

The following is quadruples representation.

Table 7.1: Quadruples.

S.no.	*op*	arg_1	arg_2	*result*
0	*minus*	c		t_1
1	$*$	b	t_1	t_2
2	*minus*	c		t_3
3	$*$	b	t_3	t_4
4	$+$	t_2	t_4	t_5
5	$=$	t_5		a
	...	...	...	...

For readability, we use actual identifiers like a, b, and c in the fields arg_1, arg_2 and *result* in Table 7.1, instead of pointers to their symbol-table entries. Temporary names can either be entered into the symbol table like programmer-defined names, or they can be implemented as objects of a class $Temp$ with its own methods.

7.3 Triples

A triple has only three fields, which we call *op*, arg_1, and arg_2. Note that the result field in Table 7.1 is used primarily for temporary names. Using triples, we refer to the result of an operation x *op* y by its position, rather than by an explicit temporary name. Thus, instead of the temporary t_1 in Table 7.1, a triple representation would refer to position (0). Parenthesized numbers represent pointers into the triple structure itself. Earlier, we called positions or pointers to positions as value numbers.

Triples are equivalent to signatures. Hence, the DAG and triple representations of expressions are equivalent. The equivalence ends with expressions, since syntax-tree variants and three-address code represent control flow quite differently.

Example 7.4 *Represent the triples corresponding to three-address code (TAC) and quadruples in example 7.3.*

Solution. Corresponding to the three-address code and quadruples in example 7.3, we represent the syntax tree and triples below. The Fig. 7.3 represent the syntax tree and Table 7.2 as Triples. In the triple representation in Table 7.2, the copy statement $a = t_5$ is encoded in the triple representation by placing a in the arg_1, field and (4) in the arg_2, field.

□

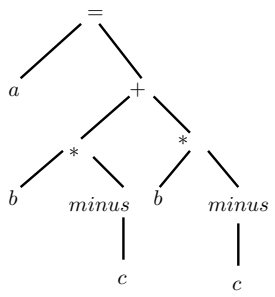

Figure 7.3: Syntax tree for $a + a * (b - c) + (b - c) * d$.

Table 7.2: Triples.

S.no.	*op*	arg_1	arg_2
0	*minus*	c	
1	$*$	b	(0)
2	*minus*	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)
	...	...	...

7.4 Static Single-Assignment Form

Static single-assignment form (SSA) is an intermediate representation that facilitates certain code optimizations. Two distinctive aspects distinguish SSA from three-address code. The first is that all assignments in SSA are to variables with distinct names; hence the term static single-assignment. Figure **??** shows the same intermediate program in three-address code and in static single-assignment form. Note that subscripts distinguish each definition of variables p and q in the SSA representation

Following is three-address code:

$$
\begin{aligned}
p &= a + b \\
q &= p - c \\
p &= q * d \\
p &= e - p \\
q &= p + q
\end{aligned}
$$

Following is static single-assignment form:

$$p_1 = a + b$$
$$q_1 = p_1 - c$$
$$p_2 = q_1 * d$$
$$p_3 = e - p_2$$
$$q_2 = p_3 + q_1$$

The same variable may be defined in two different control-flow paths in a program. For example, the source program

$$if(flag)x = -1; else\ x = 1;$$
$$y = x * a;$$

has two control-flow paths in which the variable x gets defined. If we use different names for x in the true part and the false part of the conditional statement, then which name should we use in the assignment $y = x * a$? Here is where the second distinctive aspect of SSA comes into play. SSA uses a notational convention called the 4-function to combine the two definitions of x:

$$if(flag)x_1 = -1; else\ x_2 = 1;$$
$$x_3 = \phi(x_1, x_2);$$

Here, $\phi(x_1, x_2)$ has the value x_1 if the control flow passes through the true part of the conditional and the value x_2 if the control flow passes through the false part. That is to say, the ϕ-function returns the value of its argument that corresponds to the control-flow path that was taken to get to the assignment-statement containing the ϕ-function.

7.5 Exercises

1. "Operators that are close to machine instructions make it easier to implement the intermediate form on a target machine." Justify.
2. Construct Label based and position number based translations for the following sections of codes:
   (a) $for(i = 0; i < 10; i++)j = i^2; k = i^3;$
   (b) $for(i = 0; i < 10; i++)while(i++ < 10);$
   (c) $while(i++ < 10);$
3. Translate the arithmetic expression $a + -(b + c)$ into:
   (a) A syntax tree.
   (b) Quadruples.
   (c) Triples.

(d) Indirect triples.

4. Repeat above exercise for the following assignment statements:

   (a) $a = b[i] + c[j]$
   (b) $a[i] = b * c - b * d$
   (c) $x = f(y + 1) + 2$
   (d) $x = *p + \&y$