

Compiler Construction, G22.3130

Fall 2007: Intermediate Code Generation

Due Tuesday 12.4.07

The next phase to implement in your compiler is intermediate code generation. The code will be three-address code represented as quadruples (very much as described in the textbook). Recall that, at this stage, the source language contains three predefined types, integer, string, and boolean, and two predefined type constructors, array and record.

Intermediate Code

The three-address code language consists of items of the following forms:

| | | | | | |
|---------------------------------|---|--------|---|---|---|
| x := y binop z | where binop is one of: +, -, *, /, and , or , <, <=, =, >=, >, <>. The quadruple representation of this command is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>Op</td><td>x</td><td>y</td><td>z</td></tr></table> where Op ∈ {PLUS, MINUS, TIMES, DIV, AND, OR, LT, LE, EQ, GE, GT, NE}. | Op | x | y | z |
| Op | x | y | z | | |
| x := unop y | where unop is one of: -, not . The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>Op</td><td>x</td><td>y</td><td>z</td></tr></table> where Op ∈ {NEG, NOT}. | Op | x | y | z |
| Op | x | y | z | | |
| x := y | The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>ASSIGN</td><td>x</td><td>y</td><td>-</td></tr></table> | ASSIGN | x | y | - |
| ASSIGN | x | y | - | | |
| L: | where L is a label. The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>LABEL</td><td>L</td><td>-</td><td>-</td></tr></table> | LABEL | L | - | - |
| LABEL | L | - | - | | |
| goto L | The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>GOTO</td><td>L</td><td>-</td><td>-</td></tr></table> | GOTO | L | - | - |
| GOTO | L | - | - | | |
| if x goto L | This instruction jumps to L if x is non-zero. The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>IF</td><td>L</td><td>x</td><td>-</td></tr></table> . | IF | L | x | - |
| IF | L | x | - | | |
| param x | precedes call operation, passing a parameter. The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>PAR</td><td>-</td><td>x</td><td>-</td></tr></table> NOTE: these operations should be issued in left-to-right order. The next phase of your compiler, the assembly-code generator, may decide to generate "push" operations in reverse order, but that is machine dependent. | PAR | - | x | - |
| PAR | - | x | - | | |
| call p,n | procedure call with $n \geq 0$ parameters, no return value. The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>CALL</td><td>-</td><td>p</td><td>n</td></tr></table> | CALL | - | p | n |
| CALL | - | p | n | | |
| x := fcall f,n | function call with $n \geq 0$ parameters, x is assigned the value returned by f. The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>FCALL</td><td>x</td><td>f</td><td>n</td></tr></table> | FCALL | x | f | n |
| FCALL | x | f | n | | |
| return | procedure return. The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>RET</td><td>-</td><td>-</td><td>-</td></tr></table> | RET | - | - | - |
| RET | - | - | - | | |
| freturn x | function return, returning x. The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>FRET</td><td>-</td><td>x</td><td>-</td></tr></table> | FRET | - | x | - |
| FRET | - | x | - | | |

| | | | | | |
|-------------|---|------|---|---|---|
| $x := y[i]$ | In this case, $y[i]$ refers to the $(i + 1)$ 'st element of array y , and is independent of the size of the elements. Assembly-code generation, the next phase, will convert this to a size-dependent array reference. The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>LDAR</td><td>x</td><td>y</td><td>i</td></tr></table> | LDAR | x | y | i |
| LDAR | x | y | i | | |
| $x[i] := y$ | Array store. The quadruple representation is given by <table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>STAR</td><td>x</td><td>i</td><td>y</td></tr></table> | STAR | x | i | y |
| STAR | x | i | y | | |

Note that each of the variables x , y , and z , can correspond to either variables in the source program (in particular, local variables of a procedure, formal parameters of a procedure, or global variables in the program), or temporary variables created by the intermediate code generator. The right-hand side operands can also be constants, in which case, we represent them as “=5” or =“string”.

The code produced out of the source statement

proc(x, y)

is

param x ; **param** y ; **call** *proc*,2

Assume that x has the type *array*[3..6] of *integer*, then the source statement $y := x[5]$ produces the intermediate code $y := x[2]$.

Example

Consider the following function “fact”:

```

function fact( $x : integer$ ) : integer;
var  $u : integer$ ;
begin
    if  $x = 0$  then  $u := 1$ 
        else  $u := fact(x - 1) * x$ ;
    fact :=  $u$ 
end {fact}

```

The following table contains the intermediate code produced for this function, where on the left we present the C-like representation of the code and, on the left, the quadruple presentation of the same code.

| Micro-C | Operation | Op1 | Op2 | Op3 |
|----------------------------------|-----------|---------|---------|---------|
| T1 := x=0 | EQ | Symb[4] | Symb[2] | Ast[1] |
| T2 := not T1 | NOT | Symb[5] | Symb[4] | |
| if T2 then goto L1 | IF | Symb[6] | Symb[5] | |
| u := 0 | ASSIGN | Symb[3] | Ast[1] | |
| goto L2 | GOTO | Symb[7] | | |
| L1: | LABEL | Symb[6] | | |
| T1 := x-1 | MINUS | Symb[4] | Symb[2] | Ast[2] |
| param T1 | PAR | Symb[4] | | |
| T1 := fcall fact,1 | FCALL | Symb[4] | Symb[1] | 1 |
| u := T1*x | TIMES | Symb[3] | Symb[4] | Symb[2] |
| L2: | LABEL | Symb[7] | | |
| fret u | FRET | | Symb[3] | |

The quadruple table refers to the symbol table which is given by

| Entry No. | Name | Kind | Type | Scope |
|-----------|------|------------------|---------|--------|
| 1. | fact | <i>function</i> | integer | global |
| 2. | x | <i>parameter</i> | integer | fact |
| 3. | u | <i>variable</i> | integer | fact |
| 4. | T1 | <i>temporary</i> | integer | fact |
| 5. | T2 | <i>temporary</i> | integer | fact |
| 6. | L1 | <i>label</i> | — | fact |
| 7. | L2 | <i>label</i> | — | fact |

and to the AST which contains the following fragment:

| Node No. | Node Type | Child-1 | Child-2 | Child-3 |
|----------|-----------|---------|---------|---------|
| ... | | | | |
| 1. | INT | 0 | — | — |
| 2. | INT | 1 | — | — |
| ... | | | | |

Note that the code generation process generates temporary variables (such as T1 and T2) and labels (L1, L2) which should be added to the symbol table. Also note that the boolean operation, such as “x=0” generate boolean values which we represent as 1 for *true* and 0 for *false*. The first two quadruples place in T2 a value which is 1 iff x is unequal to 0.

Accessing Structured Variables

The only intermediate code operations that enable us to access components of a structured type are the indexed references $x[i]$. In case x has a structured type which is not a simple array, we have to compute in i a more complicated index.

We will illustrate the necessary operations on the following example:

```

type
  foo = array[1..10] of
    record
      x : array[-2..2] of integer;
      y : array[1..5] of array[4..6] of integer
    end;
var
  i,j,k,num  : integer;
  all       : foo

```

To explain the handling of this declaration, we observe that it is equivalent to the following declarations:

```

type
  foo = array[1..10] of t_1;
  t_1 = record x: t_2; y: t_3 end;
  t_2 = array[-2..2] of integer;
  t_3 = array[1..5] of t_4;
  t_4 = array[4..6] of integer;
var
  i,j,k,num  : integer;
  all       : foo

```

where t_1 , t_2 , t_3 , t_4 , are anonymous types introduced in order to facilitate the processing. For each of the included types, we compute its *size*. This is computed as follows:

- If t is a *basic type*, i.e., **integer**, **string**, **boolean**, then $size(t) = 1$. Note that the size of a **string**'s entry is also 1, because a standard string entry is a pointer to the actual string which is stores elsewhere.
- If t is an *array* defined by “ $t = \text{array}[c_1..c_2]$ of t_1 ”, then $size(t) = (c_2 - c_1 + 1) \times size(t_1)$.
- If t is an *record* defined by “ $t = \text{record } x_1 : t_1; \dots; x_k : t_k \text{ end}$ ”, then $size(t) = size(t_1) + \dots + size(t_k)$.

In addition, for each field designator (such as x or y in the declaration above) we compute a *base offset* which measures the distance from the beginning of the record. Thus, if type t is a record defined by

$$t = \text{record } x_1 : t_1; \dots; x_k : t_k \text{ end}$$

then, for each $i = 1, \dots, k$, we set

$$base(x_i) = size(t_1) + \dots + size(t_{i-1})$$

Following is a symbol table which contains the types and variables contained in the example declaration. The table contains the additional attributes *size* and *base*.

| No. | Name | Kind | Type | Scope | size | base |
|-----|------|-------------------------|-------------------------|---------|------|------|
| 1. | foo | <i>type</i> | array[1..10] of Symb[2] | F | 200 | |
| 2. | t_1 | <i>type</i> | record x:t_2; y:t_3 end | F | 20 | |
| 3. | x | <i>field_designator</i> | Symb[4] | Symb[2] | 5 | 0 |
| 4. | t_2 | <i>type</i> | array[-2..2] of integer | Symb[2] | 5 | |
| 5. | y | <i>field_designator</i> | Symb[6] | Symb[2] | 15 | 5 |
| 6. | t_3 | <i>type</i> | array[1..5] of Symb[7] | Symb[2] | 15 | |
| 7. | t_4 | <i>type</i> | array[4..6] of integer | Symb[2] | 3 | |

Having computed the necessary attributes for all types and field designators, we can now issue a code for any compound reference, such as $all[i].y[j][k]$.

Recall that a compound reference has the general form “A $S_1 \cdots S_k$ ”, where A is a variable and each S_i is a selector which is either of the form “[i]” or “. d ”, where d is a field designator. We provide an inductive definition of an expression $offset(AS_1 \cdots S_i)$ for each $i = 1, \dots, k$.

- $offset(A) = 0$.
- Consider the case that $S_i = [j]$. In that case, the type of $AS_1 \cdots S_{i-1}$ must be “array[$c_1 \dots c_2$] of t ”. We therefore define

$$offset(AS_1 \cdots S_i) = offset(AS_1 \cdots S_{i-1}) + (j - c_1) \times size(t)$$

- Next, consider the case that $S_i = .d$. In that case, the type of $AS_1 \cdots S_{i-1}$ must be a record t , where d is a field designator in t . We therefore define

$$offset(AS_1 \cdots S_i) = offset(AS_1 \cdots S_{i-1}) + base(d)$$

Let us apply this definition in order to compute $offset(all[i].y[j][k])$.

$$\begin{aligned}
offset(all) &= 0 \\
offset(all[i]) &= 0 + 20(i - 1) &= 20i - 20 \\
offset(all[i].y) &= 20i - 20 + 5 &= 20i - 15 \\
offset(all[i].y[j]) &= 20i - 15 + 3(j - 1) &= 20i + 3j - 18 \\
offset(all[i].y[j][k]) &= 20i + 3j - 18 + (k - 4) &= 20i + 3j + k - 22
\end{aligned}$$

Thus, the code that should be issued for the source statement

$num := all[i].y[j][k]$

is given by:

```

T1 := i - 1
T1 := 20 * T1
T1 := T1 + 5
T2 := j - 1
T2 := T2 * 3
T1 := T1 + T2
T2 := k - 4
T1 := T1 + T2
num := all[T1]

```

Representation of quadruples

Quadruples can be represented by a record (or C++/Java object) with four fields: `op`, `target`, `arg1`, and `arg2`. The `target`, `arg1`, and `arg2` fields might refer to variables in the source program or to temporary variables created by the intermediate code generator. If the former, then those fields should point to the node in the symbol table containing the type information, etc. that your parser and type checker created. Otherwise, as your intermediate code generator generates a temporary variable, it should create a node with the necessary type information, which the field in the quadruple will point to.

Each procedure in the program can be represented by an array of quadruples. This array should be contained in a record or object that also points to the node giving type and formal parameter information about the procedure.

Deliverables

For each processed program, print the symbol table and AST and also the intermediate generated code in the Micro-C format as introduced above.

Run your compiler on the test programs `test0009.pas` — `test0025.pas`. Submit your program with the output produced for these test programs.