# CS 5500 – The Structure of a Compiler

## Intermediate Code Generation

### <u>Motivation</u>

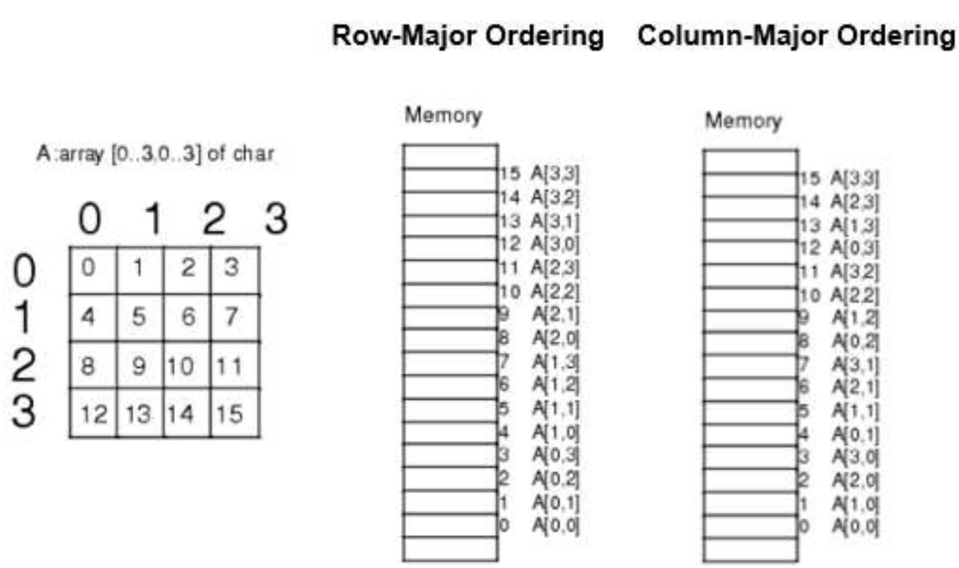
- Need a **simple representation** of source code that is: (1) easy to manipulate for code optimization, and (2) easier to translate to target code (e.g., Assembly language code)
- Should be **relatively independent of target machine**

### <u>Three-Address Code</u>

- A label can be attached to any instruction by prefixing it with **L:** and then you can **goto L**
- Assignment statements are of the form **x = y** or **x = y *op* z**
- Conditional statements are of the forms: **ifFalse x goto L** and **ifTrue x goto L**
- Array references count as 2 addresses; for example, **x[y] = z** or **z = x[y]**
- <u>Ex</u>: A[i] = 2 * A[j-k]

```
t3 = j - k;
t2 = A[t3];
t1 = 2 * t2;
A[i] = t1
```

- **Multi-dimensional** arrays must be "converted" to **1D**; use **row-major ordering** (used for Java and C) or **column-major ordering** (used for Fortran and some versions of Basic)



- **Row-major ordering**

Assume indexing starts at 0

**Let w = width of each array element**

**base = relative address of A[0] (i.e., very 1st element in array)**

If A is 1D, then A[ i ] begins in location: **base + (i * w)**

If A is 2D, let $n_2$ **= # elements along dimension 2**

Then $A[i_1][i_2]$ is in location: $\mathbf{base + (i_1 * n_2 + i_2) * w}$

If A is k dimensions, let $n_j$ **= # elements along dimension j for** $1 \le j \le k$

Then $A[i_1][i_2]\ldots[i_k]$ is in location:

$$\mathbf{base + ((\ldots (i_1 * n_2 + i_2) * n_3 + i_3) \ldots) * n_k + i_k) * w}$$

Note: If array indexing doesn't begin at 0, use $\mathbf{i_j - low_j}$ instead of just $i_j$

Ex: A : array[1..10, 1..20] of integer
integers take 4 bytes each
array A stored starting at address 0

Find the location of A[4, 5]

$\mathbf{base + ((i_1 - low_1) * n_2 + (i_2 - low_2)) * w}$

**0 + ((4-1) * 20 + (5-1)) * 4 = 256**

**Compile-time precalculation:**

**base + (i – low) * w** can instead be **i * w + c where c = base – low * w**

**Store value of c in symbol table for this array identifier**
**Then only generate code to compute i * w + c instead of generating code to compute base + (i – low) * w each time array is referenced**

**Remember you only have 3 addresses per statement, so conserve!**

## Example of Syntax-Directed Intermediate Code Generation

Assume ***gen*** is a function that outputs its parameters to stdout, a file, or wherever you want the 3-address code to be output to

***Temp()*** is a temp variable

***get*** is a function that, given an ident token, returns its lexeme

***addr***, ***array***, and ***type*** are attributes associated with nonterminals

| | |
|---|---|
| $S \rightarrow$ id = E ; | { gen(get(id.lexeme), '=', E.addr); } |
| \| L = E; | { gen(L.addr.base, '[', L.addr, ']', '=', E.addr); } |
| $E \rightarrow E_1 + E_2$ | { E.addr = new Temp( );<br>gen(E.addr, '=', $E_1$.addr, '+', $E_2$.addr); } |
| \| id | { E.addr = get(id.lexeme); } |
| \| L | { E.addr = new Temp( );<br>gen(E.addr, '=', L.array.base, '[', L.addr, ']'); } |
| $L \rightarrow$ id [ E ] | { L.array = get(id.lexeme);<br>L.type = L.array.type.elem;<br>L.addr = new Temp( );<br>gen(L.addr, '=', E.addr, '*', L.type.width); } |
| \| $L_1$ [ E ] | { L.array = $L_1$.array;<br>L.type = $L_1$.type.elem;<br>t = new Temp( );<br>L.addr = new Temp( );<br>gen(t, '=', E.addr, '*', L.type.width);<br>gen(L.addr, '=', $L_1$.addr, '+', t); } |

Using this translation, generate 3-address code for **x + A[ i ][ j ]** where:

A : [0..1][0..2] of integer;
x, i, j : integer;

Assume an integer is 4 bytes

**Type** of **A** is **array(2, array(3, integer))**, and **width** is **24**
**Type** of **A[i]** is **array(3, integer)**, and width is **12**

**$E_1$ => id => x** — **E.addr = get(id.lexeme) = x**

**E => id => i** — **E.addr = get(id.lexeme) = i**

**E => id => j** — **E.addr = get(id.lexeme) = j**

**L => id [ E ] => A [ E ]**
**=> A [ id ]**
**=> A[ i ]**

**L.array = get(id.lexeme) = A**
**L.type = L.array.type.elem =**
**array(2, array(3, int)).elem = array(3, int)**
**L.addr = new Temp( ) = t1**
**gen(L.addr, '=', E.addr, '*', L.type.width)= "t1=i*12"**

**L => $L_1$ [ E ]**
**=>* A[ i ][ E ]**
**=> A[ i ][ id ]**
**=> A[ i ][ j ]**

**L.array = $L_1$.array = A**
**L.type = $L_1$.type.elem = array(3, int).elem = int**
**t = new Temp( ) = t2**
**L.addr = new Temp( ) = t3**
**gen(t, '=', E.addr, '*', L.type.width) = "t2 = j * 4"**
**gen(L.addr, '=', $L_1$.addr, '+', t) = "t3 = t1 + t2"**

**$E_2$ => L =>*A[ i ][ j ]** — **E.addr = new Temp( ) = t4**
**gen(E.addr, '=', L.array, '[', L.addr, ']') =**
**"t4 = A[t3]"**

**E => $E_1$ + $E_2$** — **E.addr = new Temp( ) = t5**
**gen(E.addr, '=', $E_1$.addr, '+', $E_2$.addr) = "t5 = x + t4"**

**So generated code is:**

**t1 = i * 12**
**t2 = j * 4**
**t3 = t1 + t2**
**t4 = A[t3]**
**t5 = x + t4**

- Other kinds of statements:
  - Procedure calls: **param $x_1$**

    **param $x_2$**

    **…**

    **param $x_n$**

    **call p, n** or **y = call p, n**
  - Address and pointer assignments: **x = &y** or **x = *y** or ***x = y**
  - I/O: **read x** or **write x**

**Representations**

- **Quadruple** is of the form ***(op, arg1, arg2, result)***
- Exceptions: (1) Instructions with unary operators don't use $arg_2$

  (2) Copy statement uses = for *op*, and doesn't use $arg_2$

  (3) *param* doesn't use $arg_2$ or *result*

  (4) Jumps put the target label in *result*
- In actual implementation, $arg_1$, $arg_2$, and *result* are likely **references to symbol table entries**; temporary variables can be entered into symbol table as needed
- Example: a = b * -c + b * -d

| | |
|---|---|
| **t1 = minus c** | **(minus, c, , t1)** |
| **t2 = b * t1** | **(*, b, t1, t2)** |
| **t3 = minus d** | **(minus, d, , t3)** |
| **t4 = b * t3** | **(*, b, t3, t4)** |
| **t5 = t2 + t4** | **(+, t2, t4, t5)** |
| **a = t5** | **(=, t5, , a)** |

- **Triple** is of the form ***(op, arg1, arg2)***
- *result* is treated as **location of instruction** that's responsible for that value
- Example: a = (b * -c) + (b / -d)

| | op | arg1 | arg2 |
|---|---|---|---|
| **0** | **minus** | **c** | |
| **1** | ***** | **b** | **[0]** |
| **2** | **minus** | **d** | |
| **3** | **/** | **b** | **[2]** |
| **4** | **+** | **[1]** | **[3]** |
| **5** | **=** | **a** | **[4]** |

- ***Why are quadruples better than triples???***

  **Hint: In an optimizing compiler, instructions often moved around!**

  **Quadruples: move an instrx that computes temp t, then instrx's that use t require no change**

  **Triples: result of operation is referred to by its position, so moving instrx may require changing all references to that result**

- **Indirect triples** consist of a listing of pointers to triples, rather than just a listing of the triples themselves; facilitates moving instructions around!

- **Static Single-Assignment Form (SSA)** differs from 3-address code in 2 ways

- First, all assignments are to variables with **distinct names**

  <u>Ex</u>: p = a + b    vs.    p1 = a + b
  
  q = p – c    q1 = p1 – c
  
  p = q * d    p2 = q1 * d

- Second, uses Φ–function when it needs to “combine” definitions of a variable

  <u>Ex</u>: if (flag) x = -1; else x = 1;    vs.    … x1 = -1 … x2 = 1
  
  y = x * a;    y = **?** * a;

  **x3 = Φ(x1, x2);**
  
  **y = x3 * a;**

  **Φ–function returns the value of its arg that corresponds to the control-flow path taken to get to the statement containing the Φ–function**

- Facilitates certain code optimizations…