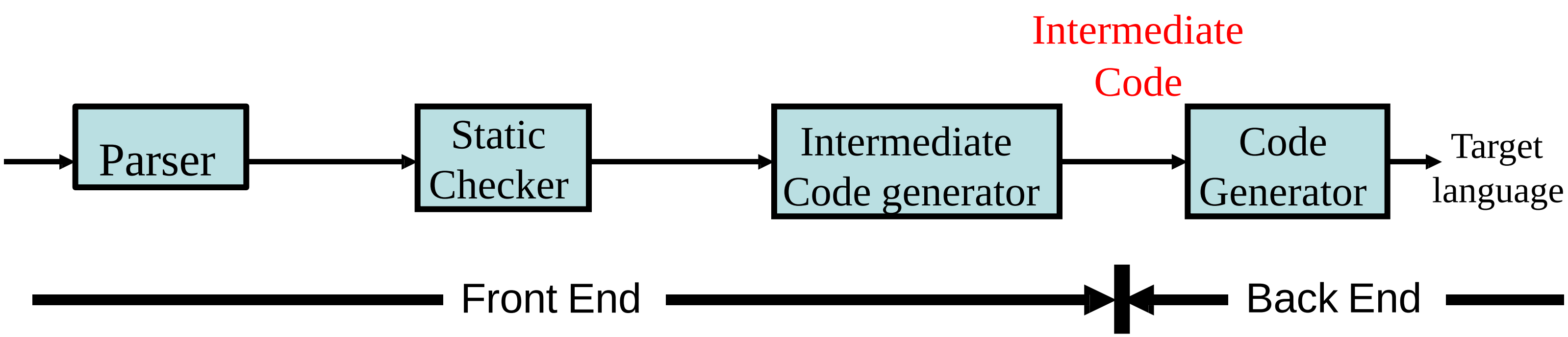
CSE 4102

Intermediate Code Generation

Compilers principles, techniques, & tools-ULLMAN
Chapter 06

Compiler Architecture



- m $\times$ n compliers can be built by writing m front ends and n back ends – save considerable amount of effort
- We assume parsing, static checking and IC generation is done sequentially
  - These can be combined and done during parsing
- Static checking
  - Operator operand compatibility
  - Proper placement of break/continue keywords etc.

Intermediate Code (IC)

- The given program in a source language is converted to an equivalent program in an intermediate language by the IC generator.
- Ties the front and back ends together
- Language and Machine neutral
- Many forms
- Level depends on how being processed
- More than one intermediate language may be used by a compiler

Intermediate Representations

- In most compilers, the parser builds an *intermediate representation* of the program
- Rest of the compiler transforms the IR to "improve" (optimize) it and eventually translates it to final code
  - Often will transform initial IR to one or more different IRs along the way

IR Design

- Decisions affect speed and efficiency of the rest of the compiler
- Desirable properties
  - Easy to generate
  - Easy to manipulate
  - Expressive
  - Appropriate level of abstraction
- Different tradeoffs depending on compiler goals
- Different tradeoffs in different parts of the same compiler

Types of IRs

- Three major categories
  - Structural
  - Linear
  - Hybrid
- Some basic examples now; more when we get to later phases of the compiler

Levels of Abstraction

- Key design decision: how much detail to expose
  - Affects possibility and profitability of various optimizations
  - Structural IRs are typically fairly high-level
  - Linear IRs are typically low-level
  - But these generalizations don’t always hold

Intermediate language levels

- **High**

T1 ← a[i,j+2]

- **Medium**

t1 ← j + 2
t2 ← i * 20
t3 ← t1 + t2
t4 ← 4 * t3
t5 ← addr a
t6 ← t5 + t 4
t7 ← *t6

- **Low**

r1 ←[fp-4]
r2 ← r1 + 2
r3 ← [fp-8]
r4 ← r3*20
r5 ← r4 + r2
r6 ← 4 * r5
r7 ← fp – 216
f1 ← [r7+r6]

Intermediate Languages Types

- Graphical IRs:
  - Syntax trees
  - Abstract Syntax trees
  - Directed Acyclic Graphs (DAGs)
  - Control Flow Graphs
- Linear IRs:
  - Stack based (postfix)
  - Three address code (quadruples)

Graphical IRs

- Concrete Syntax Trees -Parse Trees
- Abstract Syntax Trees (AST) – retain essential structure of the parse tree, eliminating unneeded nodes.
- Directed Acyclic Graphs (DAG) – compacted AST to avoid duplication – smaller footprint as well
- Control flow graphs (CFG) – explicitly model control flow

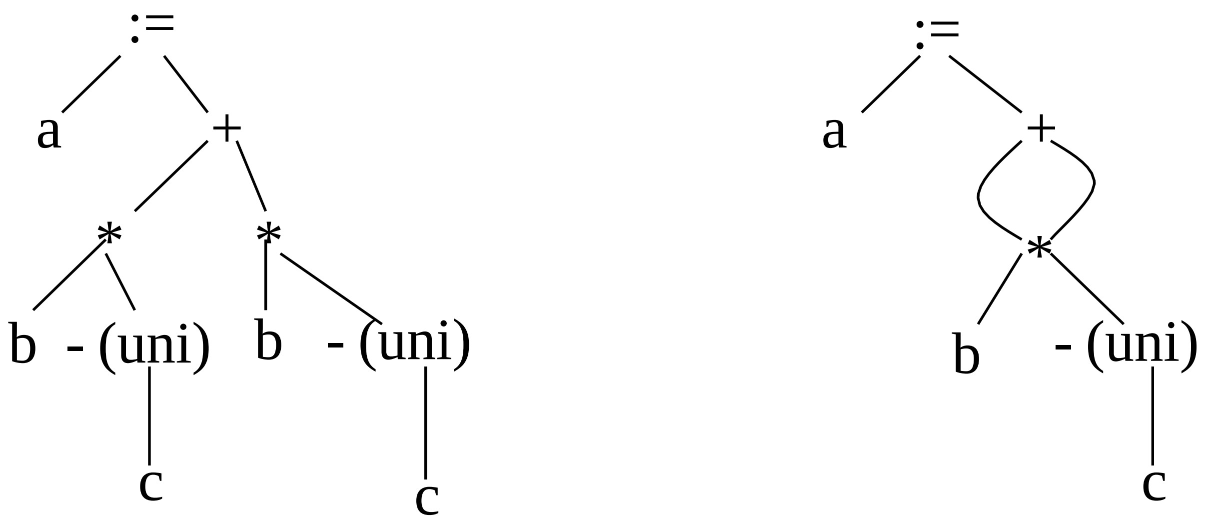
Structural IRs

- *Typically reflect source (or other higher-level) language structure*
- *Tend to be large*
- Examples: syntax trees, DAGs
- Particularly useful for source-to-source transformations

- The full grammar is needed to guide the parser, but contains many extraneous details
- Typically the full syntax tree does not need to be used explicitly

ASTs and DAGs:

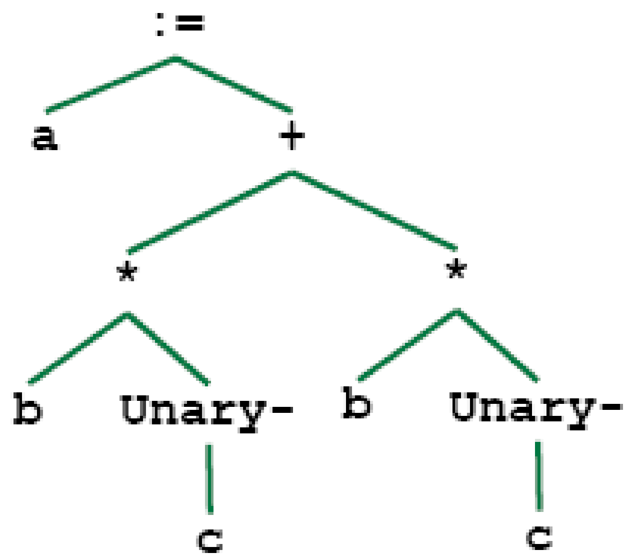
a := b *-c + b*-c



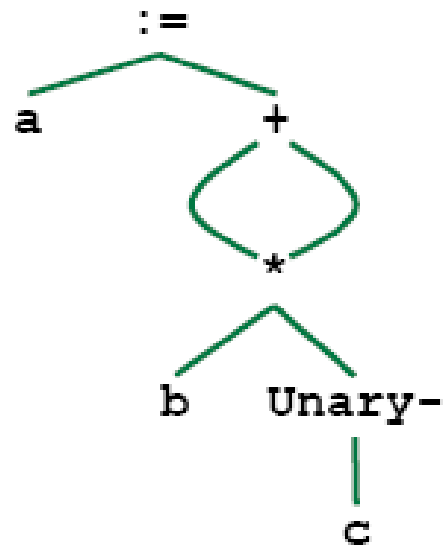
AST DAG

Implementation of DAG/AST: Value Number Method

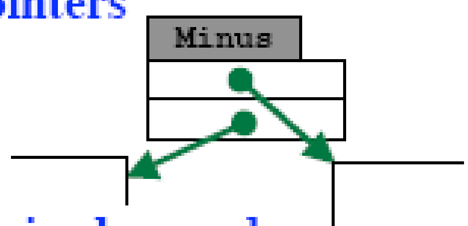
Tree:



DAG:



Structures and Pointers



An array of fixed-sized records

Q: How to build DAG's (i.e., trees with shared, common parts)?

A: When you are about to allocate a new node; look to see if you already have one with the same info.

0	id	b	–
1	id	c	–
2	unary-	1	–
3	mult	0	2
4	id	b	–
5	id	c	–
6	unary-	5	–
7	mult	4	6
8	add	3	7
9	id	a	–
10	assign	9	8

Three-Address code

- Many different representations
- General form: $x \leftarrow y \text{ (op) } z$
  - One operator
  - Maximum of three names
- Example: x=2*(n+m); becomes

  $t1 \leftarrow n + m$

  $t2 \leftarrow 2 * t1$

  $x \leftarrow t2$

Three Address Code (cont)

- Advantages
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange
- Various representations
  - Quadruples, triples, SSA
  - Much more later…

Three-Address Code

- A three-address code is:

```
x  :=  y  op  z
```

where x, y and z are names, constants or compiler-generated temporaries; **op** is any operator.

- But we may also the following notation for three-address code (it looks like a machine code instruction)

```
op      y,z,x
```

apply operator op to y and z, and store the result in x.

- We use the term **“three-address code” because each statement usually contains three addresses** (two for operands, one for the result).

Linearized Representation of DAG/AST

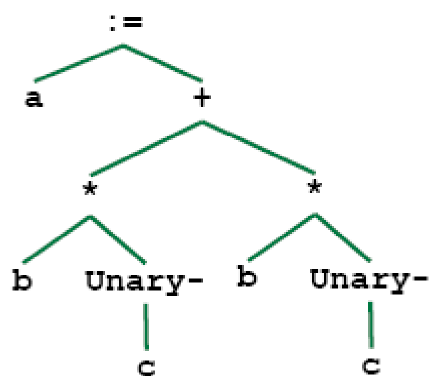
- Source Code
  - a = b * -c + b * -c
- Three address code

Each instruction has (up to) 3 operands.

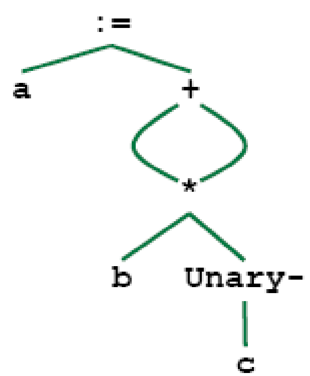
```
t1  := -c              neg    c      ⇒ t1
t2  := b * t1          mult   b,t1   ⇒ t2
t3  := -c              neg    c      ⇒ t3
t4  := b * t3          mult   b,t3   ⇒ t4
t5  := t2 + t4         add    t2,t4  ⇒ t5
a  := t5               move   t5     ⇒ a
```

- Tree Representation

Tree:



DAG:



Three-Address Statements

Binary Operator: `op y,z,result` or

```
result := y op z
```

where `op` is a binary arithmetic or logical operator. This binary operator is applied to `y` and `z`, and the result of the operation is stored in `result`.

Ex:

```
add  a,b,c
gt   a,b,c
addr a,b,c
addi a,b,c
```

Unary Operator: `op y,,result` or

```
result := op y
```

where `op` is a unary arithmetic or logical operator. This unary operator is applied to `y`, and the result of the operation is stored in `result`.

Ex: `uminus a,,c`

Three-Address Code

- Two concepts
  - Address
  - Instruction

- Address
  - Name: source-program names to appear as addresses
  - Constant: Different types of constants
  - Compiler Generated temporary:

Three-Address Instruction

Assignment Type 1: `x := y op z`

op is a binary arithmetic or logical operation

x, *y* and *z* are addresses

Assignment Type 2: `x := op z`

op is a unary arithmetic or logical operation

x and *z* are addresses

Copy Instruction: `x := y`

x and *z* are addresses and x is assigned the value of y

Three-Address Instructions

Unconditional Jump: goto L

We will jump to the three-address code with the label L, and the execution continues from that statement.

Ex: goto L1 // jump to L1

jmp 7 // jump to the statement 7

Conditional Jump 1: if x goto L and if False x goto L

We will jump to the three-address code with the label L if x is TRUE and FALSE, respectively. Otherwise, the following three-address instruction in sequence is executed next.

Conditional Jump 2: if x ***relop*** y goto L

We will jump to the three-address code with the label L if the result of y relop z is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Three-Address Statements (cont.)

Procedure Parameters: `param x`

Procedure Calls: `call p,n`

where x is an actual parameter, we invoke the procedure p with n parameters.

Ex:

```
param x1
param x2
......                         →  p(x1,...,xn)
param xn
call  p,n,   ←  n is necessary because call can be nested
```

```
f(x+1,y)  →    add    x,1,t1
               param  t1,,
               param  y,,
               call   f,2,
```

Three-Address Statements (cont.)

Indexed Assignments:

```
x := y[i]
```

sets x to the value in location i memory units beyond location y

```
y[i] := x
```

sets contents of the location i memory units beyond location y to the value of x

Address and Pointer Assignments:

```
x := &y
```

sets the r-value of x to l-value of y

```
x := *y
```

where y is a pointer whose r-value is a location

sets the r-value of x equal to the contents of that location

```
*x := y
```

sets the r-value of the object pointed by x to the r-value of y

Three address code example

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

L: t_1=i+1	100: t_1 = i +1
i=t_1	101: i = t_1
t_2=i*8	102: t_2=i*8
t_3=a[t_2]	103: t_3=a[t2]
if t_3 < v goto L	104: if t_3 < v goto 100
(A) Symbolic Labels	(B) Position Numbers

Representing 3-Address Statements

- Quadruples ("Quads")
- Triples
- Indirect Triples

```
t0 := x + y
t1 := t0 * z
a := t1
```

op, arg1, arg2, result

- **`x = minus y`**
  - Does not use arg2
- **`x = y`**
  - Op is =
- **`param a1`**
  - Uses neither arg2 nor result
- Conditional/Unconditional jumps
  - Put the target label in result

Quadruples

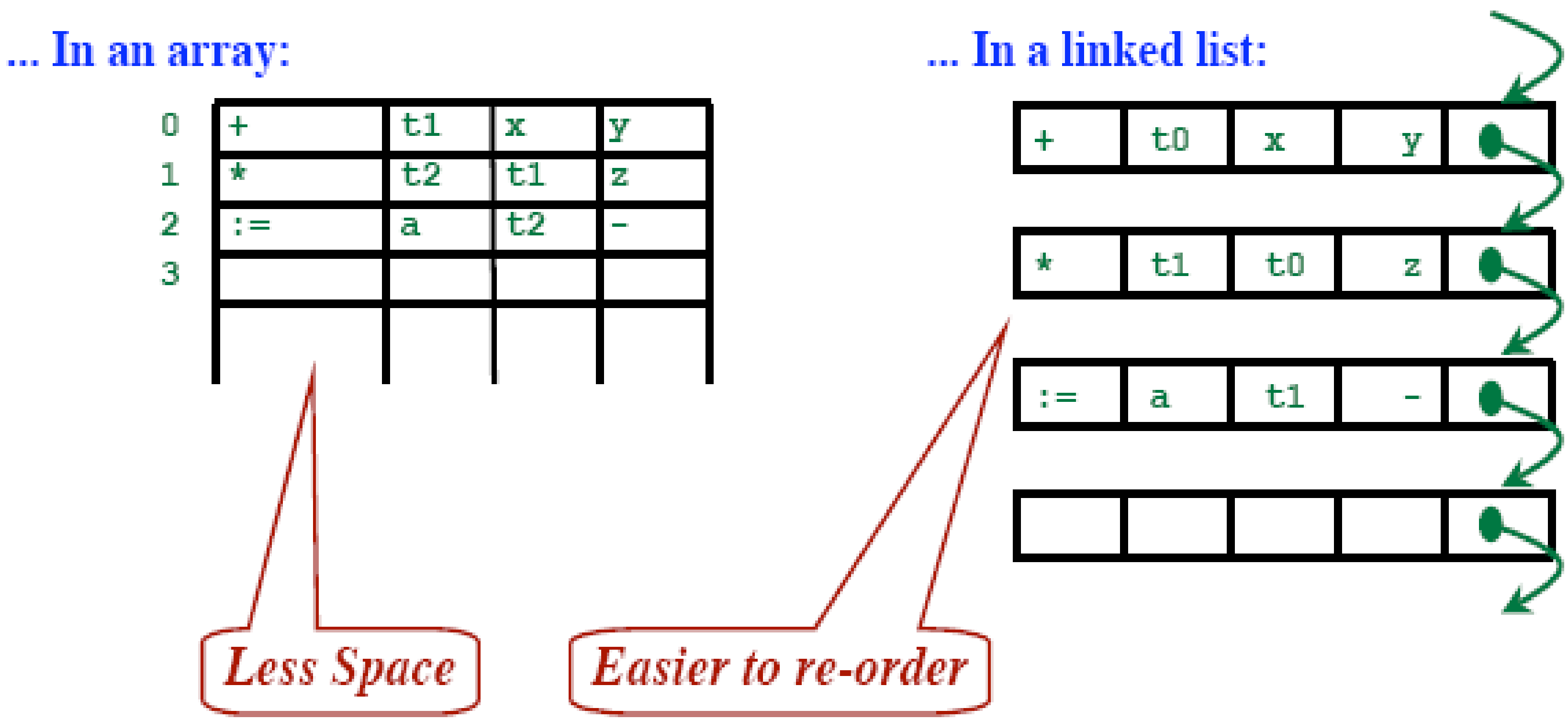
- a = b * -c + b * -c

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Instr	Operation	Arg 1	Arg 2	Result
(0)	uminus	c		t_1
(1)	mult	b	t_1	t_2
(2)	add	a	t_2	t_3
(3)	move	t_3		a

Quadruples

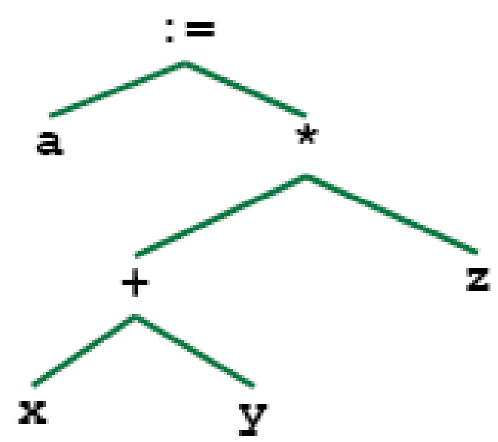
- Store each fields directly



Triples

Don't store the result directly.
Implicitly associate a temporary result with each triple.

```
t0 := x + y
t1 := t0 * z
a := t1
```



Avoids creating the temporaries.
Saves storage.
Difficult to re-order instructions.

0	+	x	y
1	*	0	z
2	:=	a	1
3			

The following instruction is difficult

```
x[i] := y
```

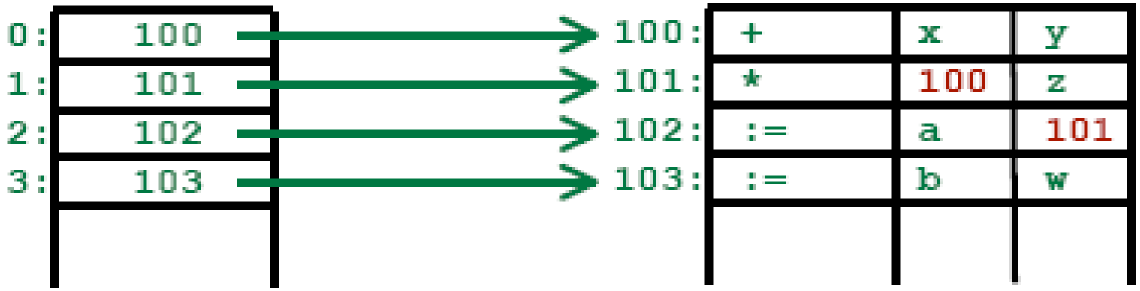
It takes 2 triples.

	op	arg1	arg2
0	[ ]=	x	i
1	:=	0	y

Indirect Triples

Get around the re-ordering problem

... by introducing another data structure.



Quadruples

Less indirection, simpler

Easier to manipulate, reorder

Triples

Indirect Triples

About same amount of space as quadruples

May save space when lots of shared sub-expressions

More complex

Three Address Code: Quadruples

Naïve representation of three address code

- Table of k * 4 small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

RISC assembly code

load	1	y	
loadi	2	2	
mult	3	2	1
load	4	x	
sub	5	4	3

Quadruples

Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

(1)	load	y	
(2)	loadI	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Implicit names occupy no space

Remember, for a long time, 640Kb was a lot of RAM

Three Address Code: Indirect Triples

- List first triple in each statement
- Implicit name space
- Uses more space than triples, but easier to reorder

Stmt List	Implicit Names	Indirect Triples		
(100)	(100)	load	y	
(105)	(101)	loadI	2	
	(102)	mult	(100)	(101)
	(103)	load	x	
	(104)	sub	(103)	(102)

- Major tradeoff between quads and triples is compactness versus ease of manipulation
  – In the past compile-time space was critical
  – Today, speed may be more important

Comparison

- By using quadruples, we can move a statement that computes A without requiring any changes in the statements using A, because the result field is explicit.
- However, in a triple representation, if we want to move a statement that defines a temporary value, then we must change all of the pointers in the operand1 and operand2 fields of the records in which this temporary value is used.
- Thus, quadruple representation is easier to work with when using an optimizing compiler, which entails a lot of code movement.
- Indirect triple representation presents no such problems, because a separate list of pointers to the triple structure is maintained.
- When statements are moved, this list is reordered, and no change in the triple structure is necessary; hence, the utility of indirect triples is almost the same as that of quadruples.

Goal

Take a source statement and produce a sequence of IR quads:

Example:

```
x := y + z;
```

IR Quads:

```
t1 := y + z
x := t1
```

Example:

```
x := (y + z) * (u + v);
```

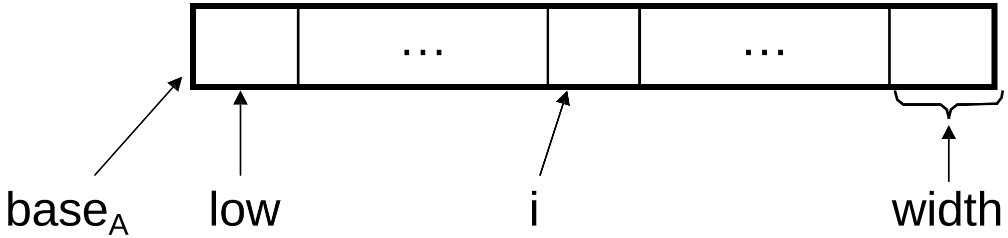
IR Quads:

```
t1 := y + z
t2 := u + v
t3 := t1 * t2
x := t3
```

Addressing Array Elements

- Elements of arrays can be accessed quickly if the elements are stored in a block of consecutive locations.

A one-dimensional array **A**:



base$_A$ is the address of the first location of the array A, **width** is the width of each array element. **low** is the index of the first array element

location of A[i] ➔ $base_A + (i - low) * width$

Addressing Array Elements (cont.)

$\texttt{base}_A\texttt{+(i-low)*width}$

can be re-written as $\underbrace{\texttt{i*width}}_{\text{should be computed at run-time}} + \underbrace{\texttt{(base}_A\texttt{-low*width)}}_{\text{can be computed at compile-time}}$

should be computed at run-time

can be computed at compile-time

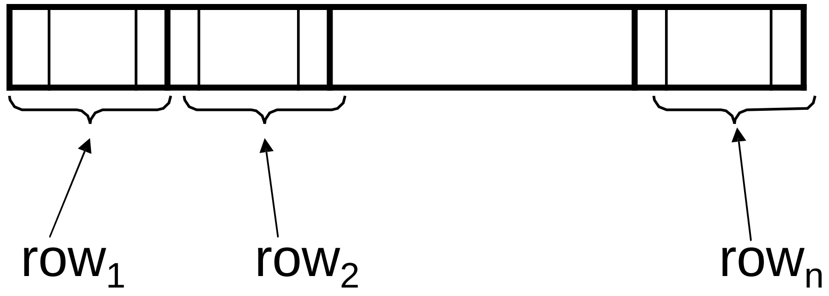
- So, the location of A[i] can be computed at the run-time by evaluating the formula i*width+c where c is ($base_A$-low*width) which is evaluated at compile-time.

- Intermediate code generator should produce the code to evaluate this formula i*width+c (one multiplication and one addition operation).

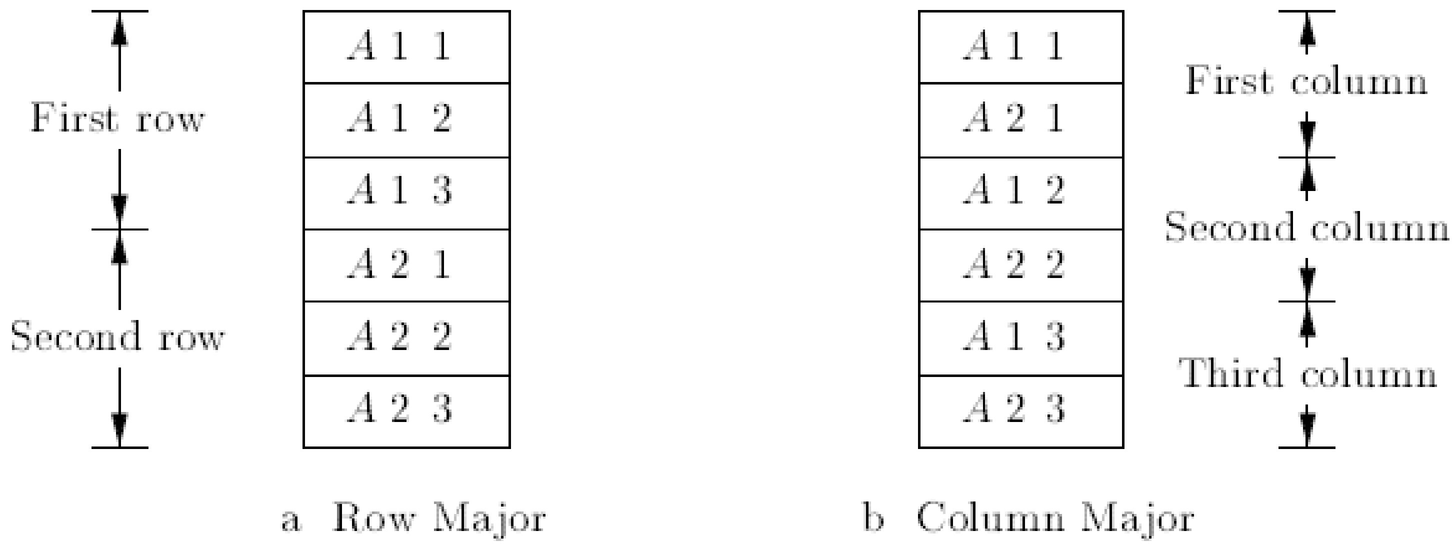
Two-Dimensional Arrays

- A two-dimensional array can be stored in
  - either **row-major** (*row-by-row*) or
  - **column-major** (*column-by-column*).
- Most of the programming languages use **row-major** method.

- Row-major representation of a two-dimensional array:



Layout of Two-Dimensional Arrays



Two-Dimensional Arrays (cont.)

- The location of $A[i_1,i_2]$ is

$$base_A + ((i_1-low_1)*n_2+i_2-low_2)*width$$

$base_A$ is the location of the array A

low_1 is the index of the first row

low_2 is the index of the first column

n_2 is the number of elements in each row

width is the width of each array element

- Again, this formula can be re-written as

$$\underbrace{((i_1*n_2)+i_2)*width}_{\text{should be computed at run-time}} + \underbrace{(base_A-((low_1*n_1)+low_2)*width)}_{\text{can be computed at compile-time}}$$

Multi-Dimensional Arrays

- In general, the location of $A[i_1,i_2,\ldots,i_k]$ is
- $((\ \ldots((i_1*n_2)+i_2)\ \ldots)*n_k+i_k)*width$ + $(base_A-((\ldots((low_1*n_1)+low_2)\ldots)*n_k+low_k)*width)$

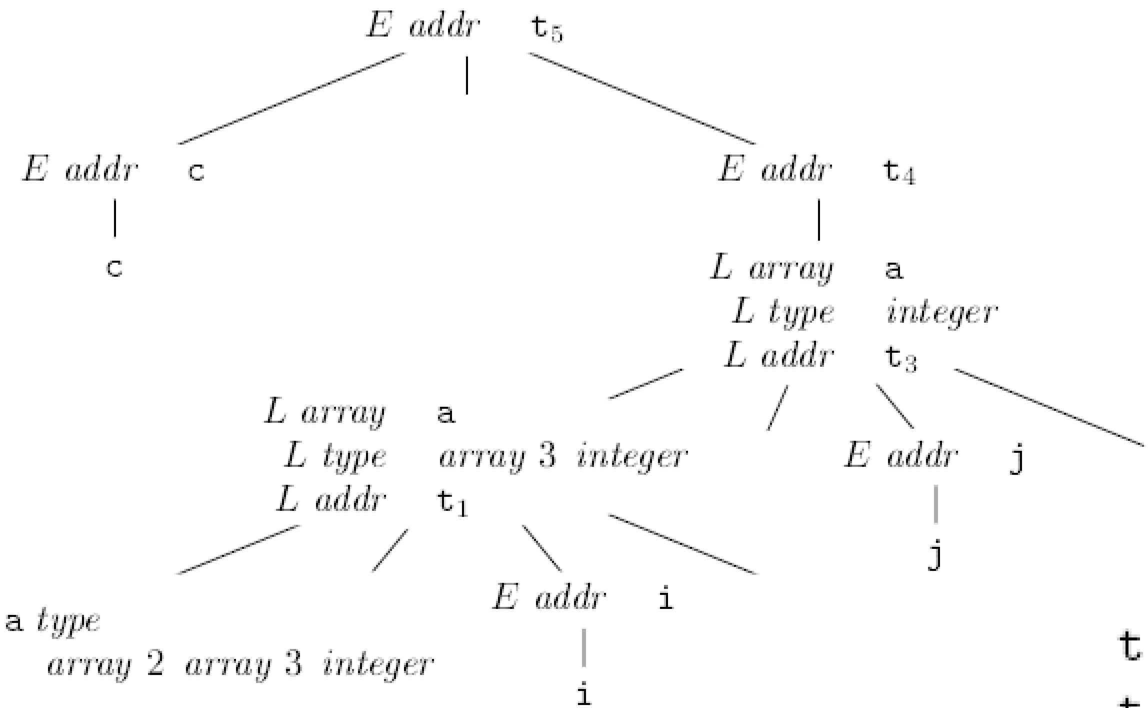
- So, the intermediate code generator should produce the codes to evaluate the following formula (to find the location of $A[i_1,i_2,\ldots,i_k]$) :
- $((\ \ldots((i_1*n_2)+i_2)\ \ldots)*n_k+i_k)*width\ + c$

- To evaluate the $((\ \ldots((i_1*n_2)+i_2)\ \ldots)*n_k+i_k$ portion of this formula, we can use the recurrence equation:
- $e_1 = i_1$
- $e_m = e_{m-1} * n_m + i_m$

Translation of Array Elements

- One dimensional
  - $base + i \times w$
  - w: width of each array element
- Two Dimensional
  - $base + i_1 \times w_1 + i_2 \times w_2$
  - w_1: width of a row
  - w_2: width of an element in a row
- k dimensional (generalized)
  - $base + i_1 \times w_1 + i_2 \times w_2 + \ldots + i_k \times w_k$

- Let a denote a 2x3 array of integers, and let c, i, j all integers. Then, type of a is array(2, array(3, integer)). $w_1 = 12$, $w_2 = 4$.



t_1	i	12
t_2	j	4
t_3	t_1	t_2
t_4	a	t_3
t_5	c	t_4

Any Question?