

We will consider code generation for the following examples of complex expressions

- array references
- function calls
- mixed type expressions
- boolean expressions

What about  $A[i, j]$ ?

First, we must agree to a storage scheme

*row-major order*

lay out as sequence of consecutive rows

rightmost subscript varies fastest

$A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]$

*column-major order*

lay out as sequence of consecutive columns

leftmost subscript varies fastest

$A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]$

## Array references

---

```
integer A[1:10];    // general: A[low:high]
...
x = A[i]
```

How do we compute the address of an array element?

$A[i]$

$base + (i - 1) \times w$

where

- $base$  is relative address (offset) of  $A[0]$
- $w$  is  $sizeof(element)$

*in general:*  $base + (i - low) \times w$

*row-major order, two dimensions*

$base + ((i_1 - low_1) \times (high_2 - low_2 + 1) + i_2 - low_2) \times w$

*column-major order, two dimensions*

$base + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times w$

This looks *expensive!*

Aho, Sethi, and Ullman, §8.3, pp. 481-482

## Array references

---

To minimize run-time costs, we need to know what they are

*On a typical RISC machine*

- integer **add** — 1 cycle
- integer **loadi** — 1 cycle
- integer **load** —  $\geq 1$  cycle (3-25 on i860)
- integer **mult** — 16 to 32 or more cycles

We should implement **mult** with **shift** whenever one argument is  $2^i$  and the other is **unsigned**

Element sizes are *always* in this form

*Of course, integer multiply via shift & add is often a win*

## Array addressing

---

The compiler should minimize the time spent in array addressing

*Several optimizations*

- pre-evaluation of subexpressions
- adopt a zero-based indexing scheme (example: C)

Consider refactoring  $A[i]$

1.  $base + (i - low) \times w$
2.  $i \times w + base - low \times w$

The second form is better since it allows partial compile-time evaluation

## Array addressing

---

*the general address polynomial for row-major order*

$$\begin{aligned} & ((\dots(i_1 n_2 + i_2) n_3 + i_3) \dots) n_k + i_k) \times w \\ & + base - \\ & ((\dots((low_1 \times n_2) + low_2) n_3 + low_3) \dots) n_k + low_k) \end{aligned}$$

where  $n_i = high_i - low_i + 1$

*For fixed-size arrays,*

- final term is compile-time evaluable
- the  $n_i$  terms are compile-time evaluable

*For variable-size arrays,*

- local arrays dimensioned by actual parameters
- generate code to compute common subexpressions in address polynomial
- should be able to recover much of the lost ground

## Function calls

---

How do we handle a function call in an expression?

Example:  $a + \text{foo}(1)$

*Treat it like a function call*

- set up the arguments
- generate the call and return sequence
- get the return value into a register

*Cautions*

- function may have side effects
- evaluation order is suddenly important

Example:  $a + \text{foo}(a, b) + b$

## Function calls

---

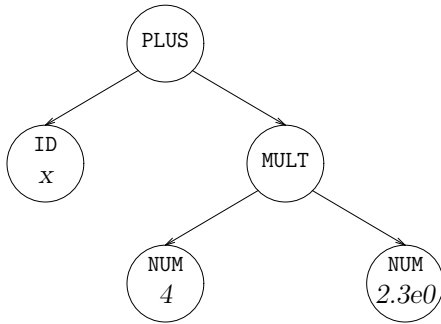
How do we handle an expression in a function call?

Example:  $\text{foo}(a + 1)$

*It has no address.*

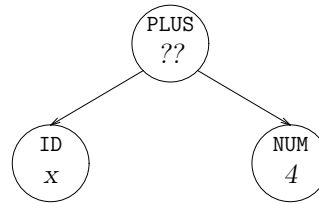
- allocate space for the result
  - $c-b-r \Rightarrow$  treat as temporary or **error**; we don't consider it an error here
  - $c-b-v \Rightarrow$  take parameter slot
- evaluate the expression (*evaluation order!*)
  - may include other function calls
- store the value ( $c-b-v$  or  $c-b-r$ )
- store the address ( $c-b-r$ )
- redefinition in callee is lost to caller

*And, of course, the expression may contain function calls ...*



Mixed type expressions:

- must have a clearly defined meaning
- typically, convert to more general type
- generate complicated, machine dependent code



Behavior is defined by the language

Most Algol-like languages use a variant on this rule if  $(Type_x \neq Type_4)$  then

1. convert  $x$  to  $Type_{result}$
2. convert  $4$  to  $Type_{result}$
3. add the converted values  $(Type_{result})$

The relation between  $Type_i$  and  $Type_{result}$  is specified by a conversion table.

Mixed type expressions

Sample Conversion Table:

PLUS	int	real	double	complex
int	int	real	double	complex
real	real	real	double	complex
double	double	double	double	complex
complex	complex	complex	complex	complex

What about assignment?

- evaluate to “natural” type
- convert to type of *lhs*

Mixed type expressions

“Typing the tree”

- usually done as part of context-sensitive analysis
- classical languages have simple type systems
- tree-attribution process
  - attribute grammars
  - tree walk to evaluate attributes
- basis information embedded in source code
  - declarations for variables
  - form of constants, e.g.,  $1$ ,  $2.3e0$ ,  $(0.2, 1.3)$

Harder Problems:

- inference without declarations
- type systems that allow recursive types

*A great use for attribute grammars*

## Boolean expressions

---

*Most languages include boolean expressions.*

### Sample Grammar

```
<expr> ::= <expr> or <expr>
         | <expr> and <expr>
         | not <expr>
         | ( <expr> )
         | id <relop> id
         | true
         | false

<relop> ::= <
           | ≤
           | =
           | ≠
           | ≥
           | >
```

*Used to compute logical values and to alter control flow*

*Relational versus logical operators*

## Boolean expressions

---

*Two schools of thought on representation:*

### Numerical Values

- assign numerical values to true and false
- evaluate booleans like arithmetic expressions

### Control Flow

- represent boolean value by location in code
- convert to numerical value when stored

*Neither representation dominates the other.*

## Boolean expressions

---

### Numerical Values

- assign a value to true (say 1)
- assign a value to false (say 0)
- use hardware — **and**, **or**, **not**, **xor**

*Choose values that make the hardware work.*

## Boolean expressions

---

### Numerical Values

<i>Source Expression</i>	<i>Generated Code</i>
b or ( c and not d )	t1 ← not d t2 ← c and t1 t3 ← b or t2
a < b	if (a<b) br L1 t1 ← 0 br L2 L1: t1 ← 1 L2: nop

*A numerical representation handles logic well.*

What about "short circuiting" (see previous example)

Do the semantics require evaluating all terms of an expression?

- once value established, stop evaluating
- ( `true or <expr>` ) is `true`
- ( `false and <expr>` ) is `false`
- save cycles in evaluation

### Order of evaluation

- if specified, must be observed in short circuiting
- if not, reorder by cost and short-circuit

### Code generation

- DAG construction
- Optimal code generation

Please read ASU Chapter 9.10