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

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

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

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 + $(i - \text{low}) \times w$
2. $i \times w + \text{base} - \text{low} \times w$

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

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
  - $c-b-r$ ⇒ treat as temporary or error; we don’t consider it an error here
  - $c-b-v$ ⇒ 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

\[
\text{if } (\text{Type}_x \neq \text{Type}_4) \text{ then}
\]

1. convert \(x\) to \(\text{Type}_{\text{result}}\)
2. convert 4 to \(\text{Type}_{\text{result}}\)
3. add the converted values \((\text{Type}_{\text{result}})\)

The relation between \(\text{Type}_i\) and \(\text{Type}_{\text{result}}\) is specified by a conversion table.

<table>
<thead>
<tr>
<th>PLUS</th>
<th>int</th>
<th>real</th>
<th>double</th>
<th>complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>real</td>
<td>double</td>
<td>complex</td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>real</td>
<td>double</td>
<td>complex</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
<td>double</td>
<td>double</td>
<td>complex</td>
</tr>
<tr>
<td>complex</td>
<td>complex</td>
<td>complex</td>
<td>complex</td>
<td>complex</td>
</tr>
</tbody>
</table>

### “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

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.

A numerical representation handles logic well.
Boolean expressions

What about "short circuiting" (see previous example)

Do the semantics require evaluating all terms of an expression?

• once value established, stop evaluating
• \((\text{true or } \text{expr})\) is \text{true}
• \((\text{false and } \text{expr})\) is \text{false}
• save cycles in evaluation

Order of evaluation

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

Next lecture

Code generation

• DAG construction
• Optimal code generation

Please read ASU Chapter 9.10