

Complex expressions

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

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

Array references

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

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

indirection vectors $A[v[i], w[j]]$

- vectors of pointers to pointers to ... to values

- much more space

- not amenable to analysis

Array references

integer A[1:10]

...

x = A[i]

How do we compute the address of an array element?

A[i]

base + (i - 1) × w

where w is sizeof(element)

in general: base + (i - low) × w

row-major order, two dimensions

base + ((i₁ - low₁) × (high₂ - low₂ + 1) + i₂ - low₂) × w

column-major order, two dimensions

base + ((i₂ - low₂) × (high₁ - low₁ + 1) + i₁ - low₁) × w

This looks expensive!

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 – ≥ 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
- re-factor expressions to compute false $A[0, 0]$

Consider refactoring $A[i]$

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

The second form is better

- compile-time evaluable(partially)
- reference independence

Array addressing

the general address polynomial for row-major order

$$((..(i_1 n_2 + i_2) n_3 + i_3)...) n_k + i_k) \times w$$

+ base –

$$((..((low_1 + n_2) + low_2) n_3 + low_3)...) n_k + low_k)$$

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

Array references

What about arrays as actual parameters?

Example: `int A[100]; foo(A);`

- Call-by-reference(c-b-r) – address of variable
- Call-by-value(c-b-v) – value of variable

Whole arrays

- need dimension information – dope vector
- stuff in all the values in calling sequence
- pass the address of dope vector as parameter
- generate the complete address polynomial

Some improvement is possible

- save n_i and low_i
- pre-compute terms on entry to procedure(if used)

Restricting the language can eliminate this problem

Array references

What does `A[12]` mean as an actual parameter?

Example: `foo(A[12])`

If the corresponding formal is a scalar, it's easy

Example: `foo(int X) { ... }`

- simply pass the value or the address
- must know about arguments on both sides
- language must force this interpretation

What if the corresponding formal is an array?

Example: `foo(int X[100]) { ... }`

- requires knowledge on both sides of call
- meaning must be well-defined and understood
- cross-procedural checking of conformability

Array references

What about variable-sized arrays?

Local arrays dimensioned by actual parameters

- same set of problems as parameter arrays
- requires dope vectors(or equivalent)
- different access costs for textually similar references

This presents a lot of opportunity for a good optimizer

- common subexpressions in the address polynomial
- contents of dope vector are fixed for an invocation
- 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
- register save-restore covers intermediate values

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

Function calls

How do we handle an expression in a function call?

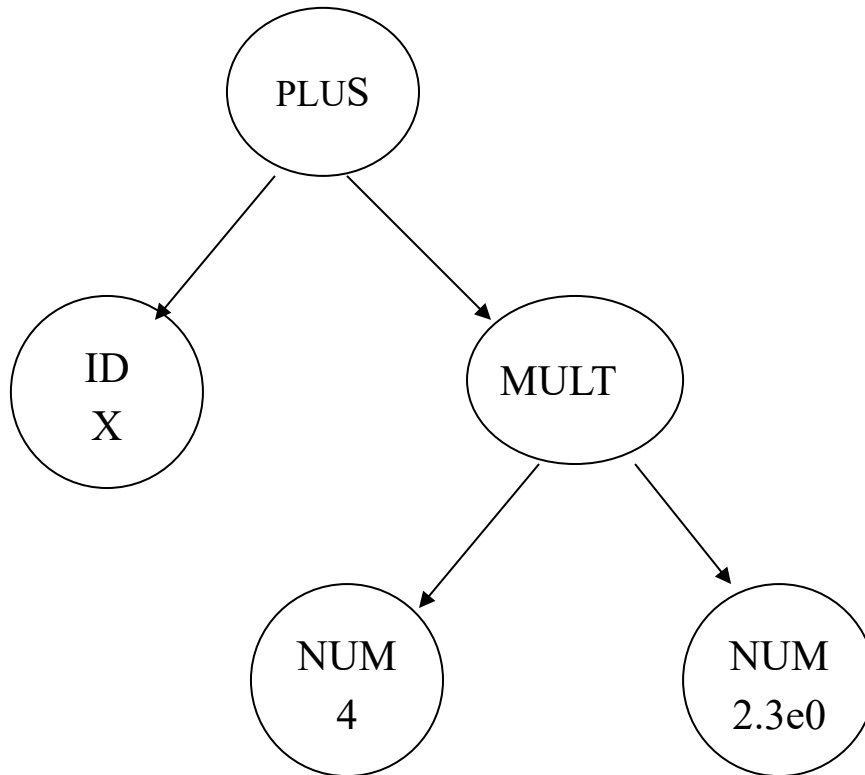
Example: `foo(a + 1)`

It has no address.

- allocate space for the result
 - `c-b-r` => treat as temporary
 - `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...

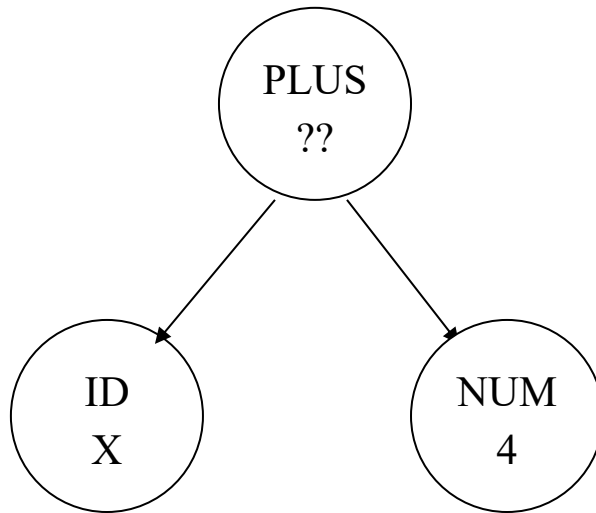
Mixed type expression



Mixed type expressions:

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

Mixed type expressions



Behavior is defined by the language

Most Algol-like languages use a variant on this rule

if ($T_x \neq T_4$) then

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

The relation between T_i and T_{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

There are some good reasons for inserting type information into each node in the AST(or encoding it in the IR)

1. error detection and reporting
2. cheaper to infer information once
3. simplifies later passes of compiler
4. annotate each expression node with a type

Mixed type expressions

“Typing the tree”

- usually done as part of context-sensitive analysis
- classical languages have simple type system
- tree-attribution process
 - attribute grammars
 - ad-hoc tree walk
- basis information embedded the source
 - declarations for variables
 - form of constants

Harder problems

- inference without declaration
- type systems that allow generated 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 for 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

Source Expression	Generated Code
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.

Boolean expressions

Control Flow

- use conditional branches and comparator
- chain of branches to evaluate expression
- code looks terrible

Conditional flow representation works well for expressions in conditional statements.

Clean up:

- branch to next statement
- branch to branch

Boolean expressions

Control Flow

Source Expression	Generated Code
$a < b$ or $(c < d \text{ and } e < d)$	if $a < b$ br LT br L1 L1: if $c < d$ br L2 br LF L2: if $e < f$ br LT br LF LF: code under false or $t1 \leftarrow \text{false}$ br LEXIT LT: code under true or $t1 \leftarrow \text{true}$ br LEXIT

This works well when the expression's value is tested but not preserved for later reuse