

Code generation

Generating stack code

- simple expressions
- boolean expressions
- control flow

Code for complex expressions

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

Storage

- local variables
- stack limit

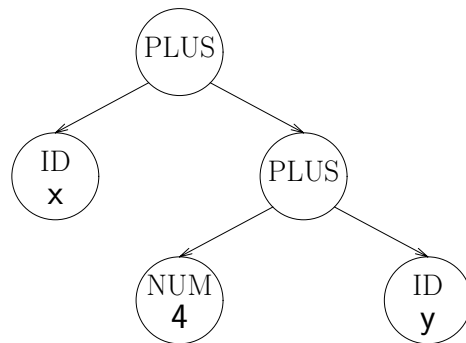
Generating stack code

Simple expressions

- make use of implicit stack
- still need variable addresses

```
expr( node )
  switch( kind of node )
    case PLUS:
      expr( left child );
      expr( right child );
      if (node.type() == INT)
        emit( iadd );
      break;
    case ID:
      int offset = addr( node.str );
      if (node.type() == INT)
        emit( iload, offset );
      break;
    case NUM:
      emit( ldc_int, node.val );
      break;
  return result;
```

Simple expressions



<code>iload 1</code>	<code>\$ push addr(x)</code>	<code>...: x</code>
<code>ldc_int, 4</code>	<code>\$ push constant 4</code>	<code>...: x, 4</code>
<code>iload 2</code>	<code>\$ push addr(y)</code>	<code>...: x, 4, y</code>
<code>iadd</code>	<code>\$ add 4 & y</code>	<code>...: x, 4+y</code>
<code>iadd</code>	<code>\$ add x & (4 + y)</code>	<code>...: x+(4+y)</code>

Boolean expressions

Algorithm

- evaluate expression
- leave 0 on top of stack if false, 1 if true

<code>E₁ == E₂</code>	<code>E₁</code>
	<code>E₂</code>
	<code>if_icmpeq L1</code>
	<code>iconst_0</code>
	<code>goto L2</code>
	<code>L1:</code>
	<code>iconst_1</code>
	<code>L2:</code>

<code>E₁ < E₂</code>	<code>E₁</code>
	<code>E₂</code>
	<code>if_icmplt L1</code>
	<code>iconst_0</code>
	<code>goto L2</code>
	<code>L1:</code>
	<code>iconst_1</code>
	<code>L2:</code>

Boolean expressions

$E_1 \ \&\& \ E_2$	E_1 dup ifeq L1 pop E_2 L1:
$E_1 \ \ E_2$	E_1 dup ifne L1 pop E_2 L1:
$! \ E$	E ifeq L1 iconst_0 goto L2 L1: iconst_1 L2:

Control structures

$x = E$	E istore addr(x)
$\text{if} (E) S$	E ifeq L S L:
$\text{if} (E) S_1 \ \text{else} \ S_2$	E ifeq L1 S_1 goto L2 L1: S_2 L2:
$\text{while} (E) S$	L1: E ifeq L2 S goto L1 L2:

Procedure calls

Code for procedure calls (FP is *frame pointer*, RA is *return address*)

save registers	<i>prolog code</i>
extend basic frame	<i>for local data</i>
find static data area	<i>if needed</i>
initialize locals	
...	
allocate child's frame	<i>start of a call</i>
evaluate & store params.	
store FP and RA	<i>in child's frame</i>
set FP for child	
jump to child	<i>may handle RA</i>
...	
RA: copy return value	<i>post-call code</i>
restore params.	<i>if needed</i>
free child's frame	
...	
store return value	<i>epilog code</i>
unextend basic frame	
restore registers	
restore parent's FP	
jump to RA	<i>hardware ret</i>

Function calls

How do we handle a function call in an expression?

Example: `a + 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 + foo(a,b) + b`

Function calls

How do we handle an expression in a function call?

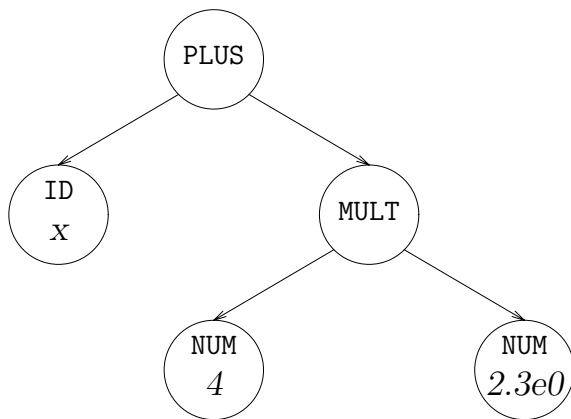
Example: `foo(a + 1)`

It has no address.

- allocate space for the result
 - call-by-reference (*c-b-r*) \Rightarrow treat as temporary
 - call-by-value (*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



Mixed type expressions:

- must have a clearly defined meaning
- typically
 1. convert operands to more general type
 2. perform operation
- generate complicated, machine dependent code

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

vector 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?

$A[i]$

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

where w is $sizeof(\text{element})$

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

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

May be able to apply optimizations to reduce cost...

Array parameters

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 vectors*
- 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 parameters

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

Memory management

Stack limit

- maximum height of stack during evaluation of an expression
- trace possible flows of control
- requires detailed knowledge of
 - code generated
 - virtual machine

Local storage

- For each procedure
 - calculate total size of local variables
 - assign offset to all local variables on stack