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

\[
\text{expr}( \text{node} )
\]

\[
\text{switch}( \text{kind of node} )
\]

\[
\text{case PLUS:}
\]
\[
\text{expr}( \text{left child} );
\]
\[
\text{expr}( \text{right child} );
\]
\[
\text{if (node.type() == INT)}
\]
\[
\text{emit( iadd );}
\]
\[
\text{break;}
\]

\[
\text{case ID:}
\]
\[
\text{int offset = addr( node.str );}
\]
\[
\text{if (node.type() == INT)}
\]
\[
\text{emit( iload, offset );}
\]
\[
\text{break;}
\]

\[
\text{case NUM:}
\]
\[
\text{emit( ldc_int, node.val );}
\]
\[
\text{break;}
\]

\[
\text{return result;}
\]
Simple expressions

![Expression Diagram]

\[
\text{PLUS} \quad \text{ID} \quad x \quad \text{PLUS} \quad \text{NUM} \quad 4 \quad \text{ID} \quad y
\]

- \text{iload 1} \quad \text{push addr}(x) \quad \ldots: \quad x
- \text{ldc\_int, 4} \quad \text{push constant 4} \quad \ldots: \quad x, 4
- \text{iload 2} \quad \text{push addr}(y) \quad \ldots: \quad x, 4, y
- \text{iadd} \quad \text{add 4 \& y} \quad \ldots: \quad x, 4+y
- \text{iadd} \quad \text{add x \& (4 + y)} \quad \ldots: \quad x+(4+y)
Boolean expressions

Algorithm

• evaluate expression

• leave 0 on top of stack if false, 1 if true

\[ E_1 == E_2 \]

E_1
E_2
if_icmpeq L1
iconst_0
goto L2
L1:
iconst_1
L2:

\[ E_1 < E_2 \]

E_1
E_2
if_icmplt L1
iconst_0
goto L2
L1:
iconst_1
L2:
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

\[
x = E \\
\text{E} \\
\text{istore addr(x)}
\]

\[
\text{if ( E ) S} \\
\text{E} \\
\text{ifeq L} \\
\text{S} \\
\text{L:}
\]

\[
\text{if ( E ) S_1 else S_2} \\
\text{E} \\
\text{ifeq L1} \\
\text{S_1} \\
\text{goto L2} \\
\text{L1:} \\
\text{S_2} \\
\text{L2:}
\]

\[
\text{while ( E ) S} \\
\text{L1:} \\
\text{E} \\
\text{ifeq L2} \\
\text{S} \\
\text{goto L1} \\
\text{L2:}
\]
Procedure calls

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

- save registers
- extend basic frame
- find static data area
- initialize locals
- allocate child’s frame
- evaluate & store params.
- store FP and RA
- set FP for child
- jump to child

RA: copy return value
restore params.
free child’s frame

- store return value
- unextend basic frame
- restore registers
- restore parent’s FP
- jump to RA

prolog code
for local data
if needed

start of a call
in child’s frame
may handle RA

post-call code
if needed

epilog code

hardware ret
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: \texttt{foo(a + 1)}

It has no address.

- allocate space for the result
  - call-by-reference (\texttt{c-b-r}) \Rightarrow treat as temporary
  - call-by-value (\texttt{c-b-v}) \Rightarrow take parameter slot
- evaluate the expression (\textit{evaluation order})
  - may include other function calls
- store the value (\texttt{c-b-v} or \texttt{c-b-r})
- store the address (\texttt{c-b-r})
- redefinition in callee is lost to caller

And, of course, the expression may contain function calls \ldots
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]
\[\text{base} + (i - 1) \times w\]
where \(w\) is \(\text{sizeof(element)}\)
\text{in general: base} + (i - \text{low}) \times w

\text{row-major order, two dimensions}
base + ((i_1 - \text{low}_1) \times (\text{high}_2 - \text{low}_2 + 1) + i_2 - \text{low}_2) \times w

\text{column-major order, two dimensions}
base + ((i_2 - \text{low}_2) \times (\text{high}_1 - \text{low}_1 + 1) + i_1 - \text{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: $\text{foo}( A[12] )$

*If the corresponding formal is a scalar, it’s easy*

Example: $\text{foo}( \text{int } X ) \{ \ldots \}$

- 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: $\text{foo}( \text{int } X[100] ) \{ \ldots \}$

- 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