## 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



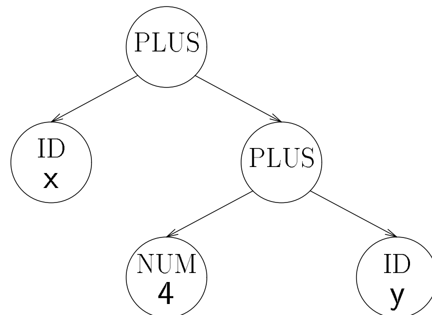
## Generating stack code

- make use of implicit stack
- still need variable addresses

Simple expressions

```
expr( node )
   switch( kind of node )
      case PLUS:
         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




```
iload 1       $ push addr(x)       ...:  x
ldc_int, 4    $ push constant 4    ...:  x, 4
iload 2       $ push addr(y)       ...:  x, 4, y
iadd          $ add 4 & y          ...:  x, 4+y
iadd          $ add x & (4 + y)    ...:  x+(4+y)
```



## Boolean expressions

Algorithm

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

$E_1 == E_2$

```
E1
E2
if_icmpeq L1
iconst_0
goto L2
L1:
iconst_1
L2:
```

$E_1 < E_2$

```
E1
E2
if_icmplt L1
iconst_0
goto L2
L1:
iconst_1
L2:
```

## Boolean expressions

| | |
|---|---|
| $E_1$ && $E_2$ | $E_1$<br>dup<br>ifeq L1<br>pop<br>$E_2$<br>L1: |
| $E_1$ \|\| $E_2$ | $E_1$<br>dup<br>ifne L1<br>pop<br>$E_2$<br>L1: |
| ! E | E<br>ifeq L1<br>iconst_0<br>goto L2<br>L1:<br>iconst_1<br>L2: |



## Control structures

| | |
|---|---|
| x = E | E<br>istore addr(x) |
| if ( E ) S | E<br>ifeq L<br>S<br>L: |
| if ( E ) $S_1$ else $S_2$ | E<br>ifeq L1<br>$S_1$<br>goto L2<br>L1:<br>$S_2$<br>L2: |
| while ( E ) S | L1:<br>E<br>ifeq L2<br>S<br>goto L1<br>L2: |

## Procedure calls

Code for procedure calls [†]

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

† FP is *frame pointer*, RA is *return address*



## 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)`

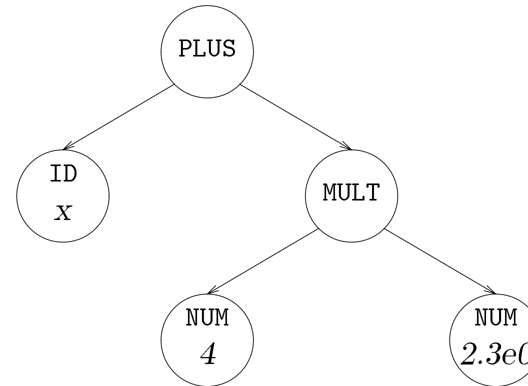
*It has no address.*

- allocate space for the result
  - — *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 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*(`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 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