CS 421 Lecture 12

- Compilation static languages, continued
  - Compiling in context
    - Assignment
    - Break and labeled statements
    - Short-circuit evaluation of boolean expressions
  - Switch statements
  - Arrays
  - Code optimization
- Friday's class: dynamic languages – code generation, garbage collection, reflection

Notation

- [S] = compiled code for S
- [e] = compiled code for e
- Use subscripts on brackets for additional arguments, e.g. $[S]_L$ is compiled code for S, assuming S occurs within a switch statements labeled L.

Assignment statements

- Old scheme: [x=e] = let (l,t) = [e] in l; x=t.
- Can give poor results: [x=3] = t=3; x=t

  [x=x+1] = t1=1; t2=x+t1; x=t2
- Compile expressions in context of target location:

  $[e]_x$ = code to calculate value of e *and* store it in x. $[e]_x$: instruction list
- [x=e] = [e]x
- $[n]_x$ = "x=n"
- $[y]_x$ = "x=y", if y a different variable from x; ϵ, otherwise
- $[e1+e2]_x$ = let t = new location in $[e1]_t$; $[e2]_x$; x=t+x

break statements

- break statement breaks from one level of switch or while.   Cannot translate “break” without knowing context.
- $[S]_L$ = code for statement S, given that S occurs inside a switch or while statement, and L is the label just after that enclosing statement.

Boolean expressions

- Current scheme: boolean expressions evaluated like any other, placing value in a temporary location:

  [e1<e2] = let (l_1, t1) = [e1], (l_2, t2) = [e2], t = newloc()
  in (l_1; l_2; t = t1<t2, t)

  [e1 && e2] = let (l_1, t1) = [e1]
  (l_2, t2) = [e2]
  in (l_1; l2; t = t1 && t2, t)

  [if e then S1 else S2] = let (l, t) = [e]
  in (l; CJUMP t L1 L2; …)

- *What's wrong?*

Boolean expressions w/ short-circuit evaluation

- Improved scheme:

```
[e1 && e2] = let  t = newlocation()
                  l1 = [e1]t
                  l2 = [e2]t
                  L1, L2 = newlabel()
             in  (l1
                  CJUMP t, L1, L2
             L1:  l2
             L2:                  ,      t)
```

- What's wrong now?

Compiling boolean expressions in context

- Get better code if boolean expression can jump to correct label as soon as possible
- $[e]_{Lt,Lf}$ = code that calculates e and jumps to Lt if it is true, Lf if it is false. The code does not save the value anywhere.

- $[true]_{Lt,Lf}$

  $[e1<e2]_{Lt,Lf}$

Compiling boolean expressions in context

- $[e1 \;\&\& \; e2]_{Lt,Lf}$

[while e do S]

Compiling switch statement

- Use “jump table” and address calculation

Compiling object references

- In expression e.t:
  - Type of e is known; call its class C
  - Location of field t within C is known; say its offset is o
  - [e] will produce (l, t), where t contains pointer to object
- [e.t] = let (l,t) = [e]

  t1 = newlocation()

  in (l; t1=t+o, t1)
- Method calls e.t(…) more complicated – will discuss in a couple of weeks

Compiling array references

- Simple rule: If A has elements of type T, and if elements of type T occupy n bytes, then address of A[i] is address of A + i*n.
- [A[e]] = let (l, t) = [e]

```
[A[e]]  =  let (l, t) = [e]
           in   (l
                  t1 = &A
                  t2 = t*w    (w size of A's elements)
                  t3 = t1+t2
                  t4 = LOADIND t3,         t4)
```

Compiling array references

- Idea extends to multi-dimensional arrays.

Machine-independent optimizations

- Machine-independent optimization = optimizations that can be done at the level of IR – i.e. does not depend upon features of target machine such as registers, pipeline, special instructions
- E.g. “loop-invariant code motion”:

```
int A[100][100]

while (j<n) {
    x = x + A[i][j]
    j++;
}
```

```
t1 = &A
t2 = i*100
t3 = t2+j
t4 = t3*4
t5 = t1+t4
t6 = LOADIND t5
x = x+t6
j = j+1
```

Machine-dependent optimizations

- Machine-dependent optimization = optimizations that exploit features of target machine such as registers, pipeline, special instructions
  - Register allocation
  - Instruction selection
  - Instruction scheduling