CS 421 Lecture 12: More code generation

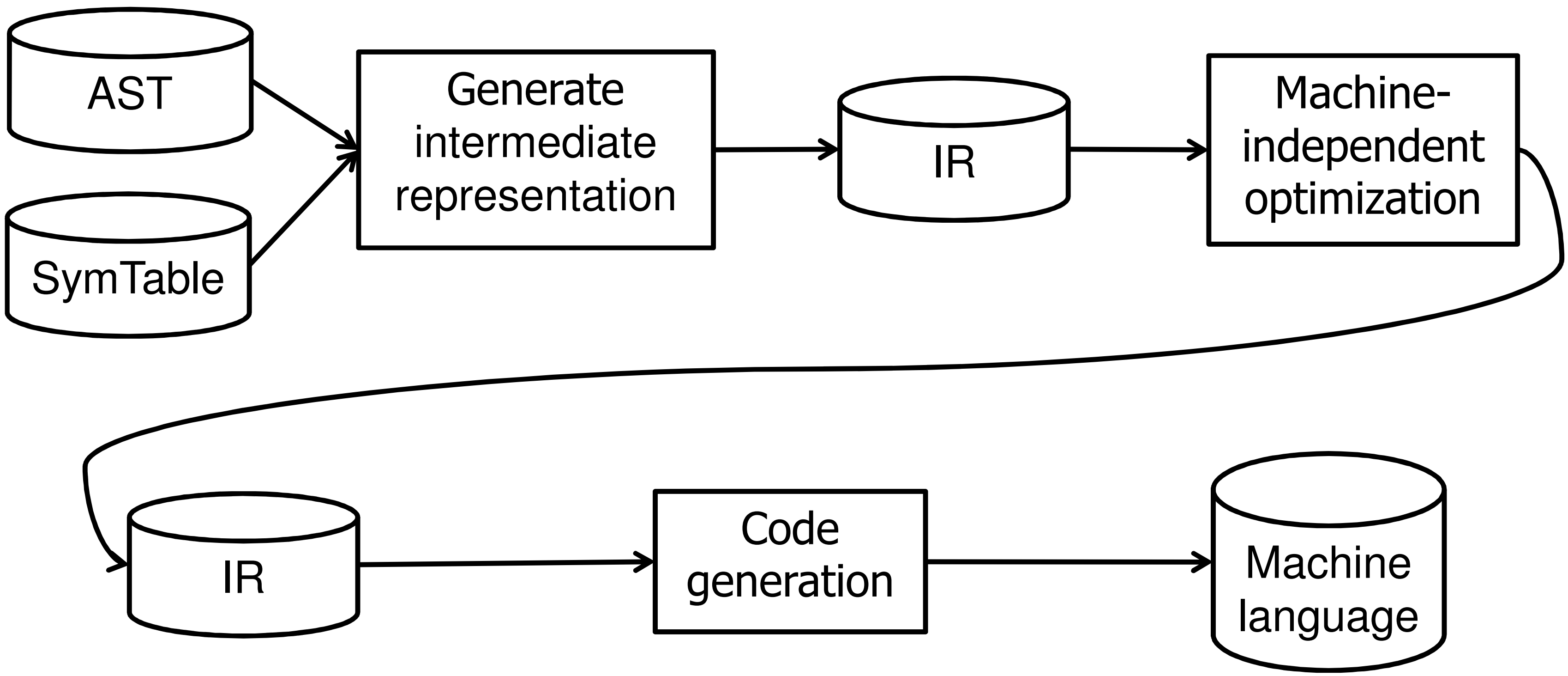
- Announcements
  - MP5 posted
  - Compass issues
  - Midterm pre-review

- Lecture outline
  - Compiling in context
    - Assignment
    - Break statements
    - Short-circuit evaluation of boolean expressions
  - Switch statements
  - Arrays
  - Code optimization

Announcements

- MP5 posted
  - Parser for MiniJava
  - Due 1:00pm Wed, July 1
- Compass issues
- Midterm pre-review
  - Next Tuesday: midterm review session
  - Past exams and sample questions posted later today
    - See the "Exams" section of the web site
  - Submit your questions on the class newsgroup
    - In the "Midterm review questions" topic

Review: compiler back-end



Notation

- Old:
  - [S] = generated code for S
  - [e] = generated code for e
- New:
  - Use subscripts on brackets for additional arguments
  - $[\ S\ ]_L$ is compiled code for S, assuming S occurs within a switch statement labeled L.
  - $[\ e\ ]_x$ is compiled code for e, assigned to variable x

Assignment statements

- Old scheme:

```
[ x=e ] = let  (I,t) = [ e ] in I; x = t
```

- Can give poor resuts:

  `[ x=3 ] = t = 3; x = t`

  `[ x=x+1 ] =` t_1 `= 1;` t_2 `= x +` t_1`;  x =` t_2

- Compile expressions *in context* of target location:

  $[\ e\ ]_x$ `= code to calculate value of e` *and* `store it in x`

  $[\ e\ ]_x$ `: instructruction list`

Examples

- Expressions within a variable context

```
[ 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 = newloc() in
                  [ e1 ]t; [ e2 ]x; x = x + t

[ x=x+1 ] =


[ x=1+x ] =
```

break statements

- Definition: breaks from one level of switch or while
  - Cannot translate "break" without knowing the 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.
  - More generally:

```
[ break ]Lb,Lc = JUMP Lb

[ continue ]Lb,Lc = JUMP Lc
```

Example: while

- Old method (no break/continue)

```
[ while e do S1 ] = JUMP L2
             L1:  [ S1 ]
             L2:  I
                  CJUMP t, L1, L3
             L3:
```

- New method (break/continue OK)

```
[ while e do S ] = JUMP L2
               L1: [ S ]L3,L2
               L2: [ e ]
                   CJUMP t,L1,L3
               L3:
```

Boolean expressions

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

```
[ e1 < e2 ] = let (I1,t1) = [ e1 ], (I2,t2) = [ e2 ], t = newloc()
              in (I1; I2; t = t1 < t2, t)

[ e1 && e2 ] = let (I1,t1) = [ e1 ], (I2,t2) = [ e2 ], t = newloc()
             in (I1; I2; t = t1 && t2, t)

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

- What's wrong with this?

Example

```
[ if (x < y && y < z) then S1 else S2 ] =
```

Short-circuit evaluation

- Improved method:

```
[ e1 && e2 ] = let t = newloc(),
                   I1 = [ e1 ]t,
                   I2 = [ e2 ]t,
                   L1, L2 = newlabel()
               in (I1
                   CJUMP t, L1, L2
               L1: I2
               L2: …               , t)
```

  - t contains value of e1 && e2
  - e2 is evaluated only if needed

Example

```
[ if (x < y && y < z) then S1 else S2 ] = let … in
                    t = x < y
                    CJUMP t, L1, L3
                L1: t = y < z
                    CJUMP t, L2, L3
                L2: [ S1 ]
                    JUMP L4
                L3: [ S2 ]
                L4:
```

- What’s wrong now?

Compiling boolean exprs 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 L_t if it is true, L_f if it is false.
  - The code does not save the value anywhere
- Examples

```
[ true ]Lt,Lf =


[ !e ]Lt,Lf =
```

Compiling boolean exprs in context

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

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

$[\ e1\ ||\ e2\ ]_{Lt,Lf} =$

Compiling boolean exprs in context

```
[ while e do S ] =
```

```
[ if e then S1 else S2 ] =
```

Example

```
[ if (x < y && y < z) then S1 else S2 ] =
```

Compiling switch statement

- Use “jump table” and address calculation

```
[ switch (e) {
  case 0: S0;
          break;
  case 1: S1;
          break;
  ...
  }]
```

```
let (I,t) = [ e ] in
       I
       δ = t*4
       i = table + δ
       JUMPIND i
   L0: [ S0 ]
       JUMP L
   L1: [ S1 ]
       ...
    L:

table: L0,L1, ...
```

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 (I,t), where t contains pointer to object

```
[ e.t ] = let (I,t) = [ e ]
              t1 = newloc()
          in (I; t1 = t + o, t1)
```

  - t1 is the address of e.x. To get value, add:

```
t2 = LOADIND t1
```

- Method calls e.t(…) more complicated – will discuss in future classes

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

Compiling array references

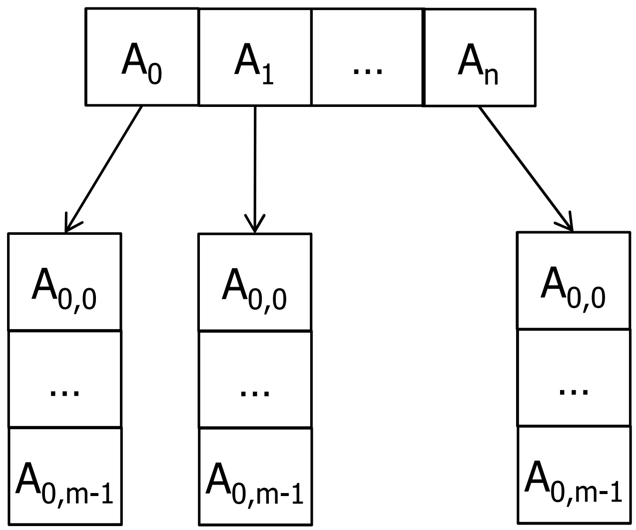
- Idea extends to multi-dimensional arrays
  - Traditional 2D arrays (C, FORTRAN)

$A_{0,0}$	…	$A_{0,m-1}$	$A_{1,0}$	…	$A_{1,m-1}$	…	$A_{n-1,0}$	…	$A_{n-1,m-1}$

```
[ A[i][j] ]  =  t1 = &A
                t2 = i * 4 * m
                t3 = t1 + t2
                t4 = j * 4
                t5 = t3 + t4
                t6 = LOADIND t5
```

Compiling array references

- 2D arrays (Java)
  - Use LOADIND t3 for location of array; use 4 instead of 4*m



Machine-independent optimizations

- Optimizations that can be done a the level of IR
  - *I.e.*, does not depend upon features of the 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
L1: t3 = t2 + j
    t4 = t3 * 4
    t5 = t1 + t4
    t6 = LOADIND t5
    x = x + t6
    j = j + 1
    CJUMP …,L1,L2
L2:
```

Machine-dependent optimizations

- Optimizations that exploit features of the target machine such as registers, pipeline, special instructions
  - Register allocation
  - Instruction selection
  - Instruction scheduling