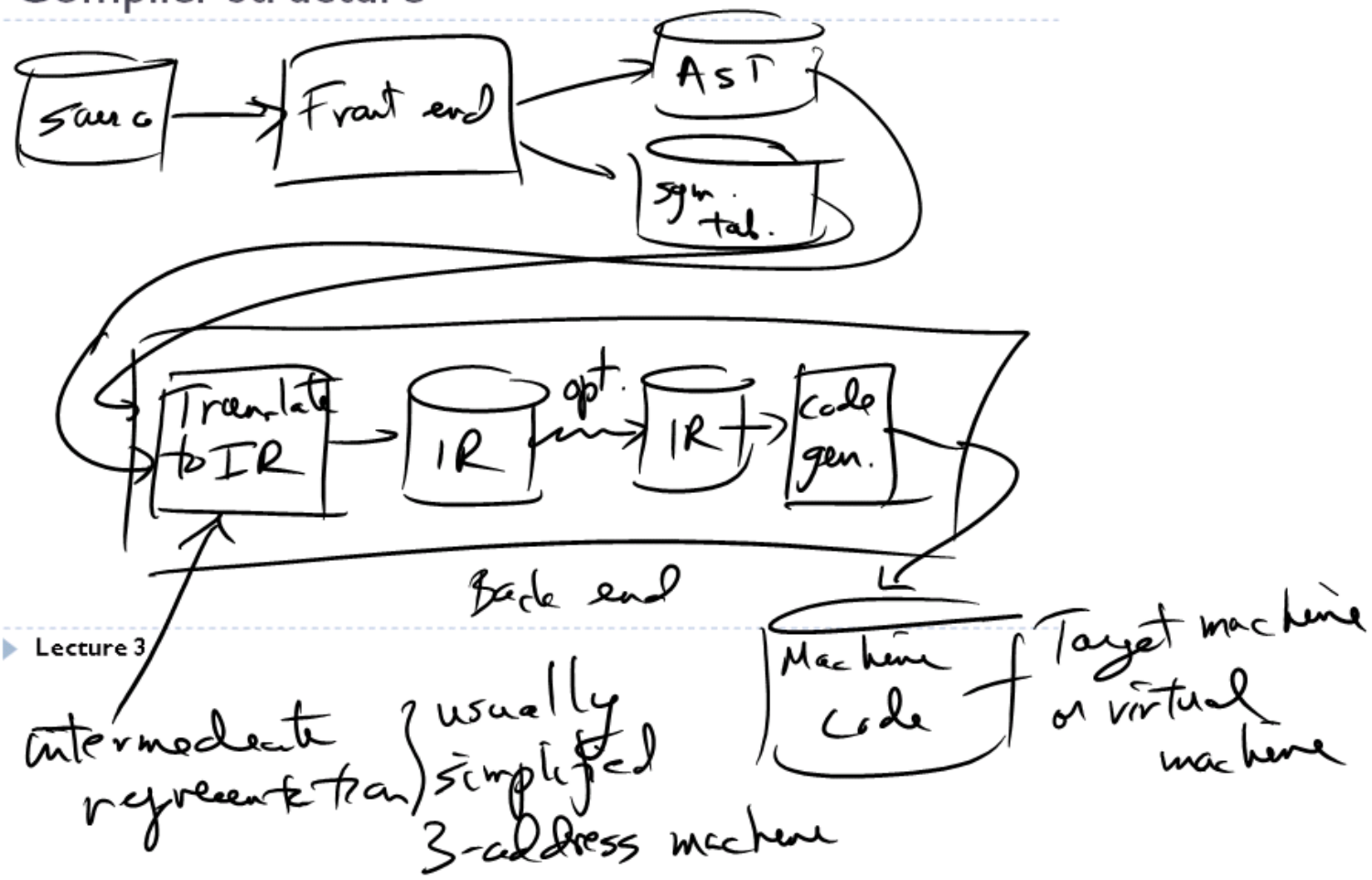
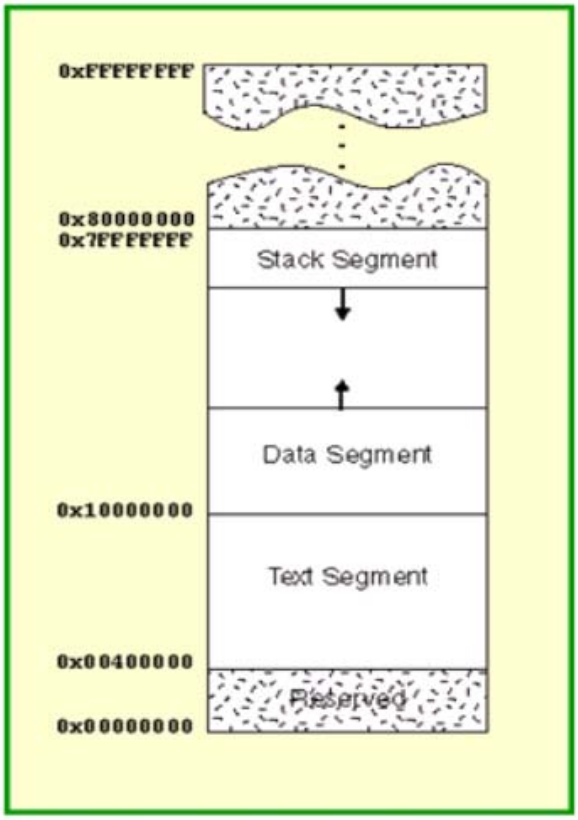
CS 421 Lecture 11

- Compilation and execution
  - Compilers
  - Execution of static languages
  - Code optimization – why?
  - Code generation
  - Code optimization – how?
- Tuesday's class: more compilation
- Thursday's class: execution of dynamic languages – tagged values, "just-in-time" code generation, garbage collection, reflection

Compiler structure

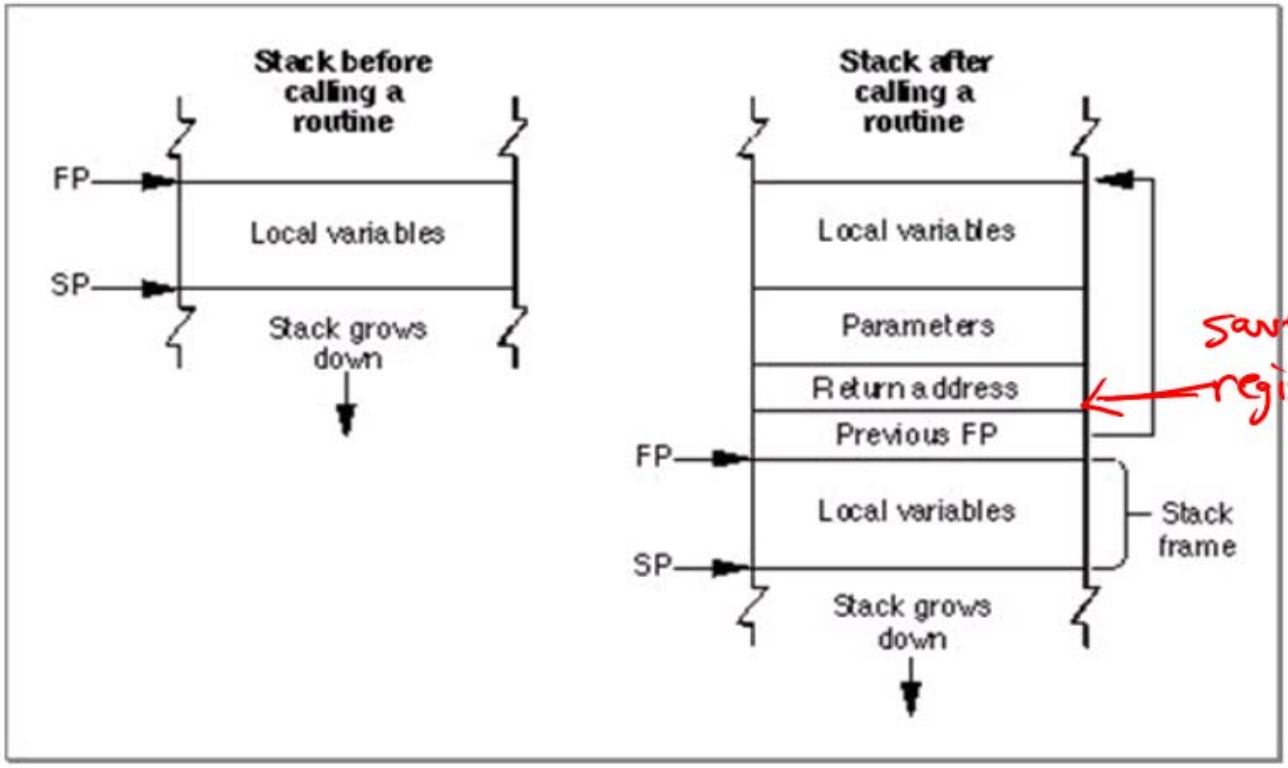


Run-time environment – memory layout



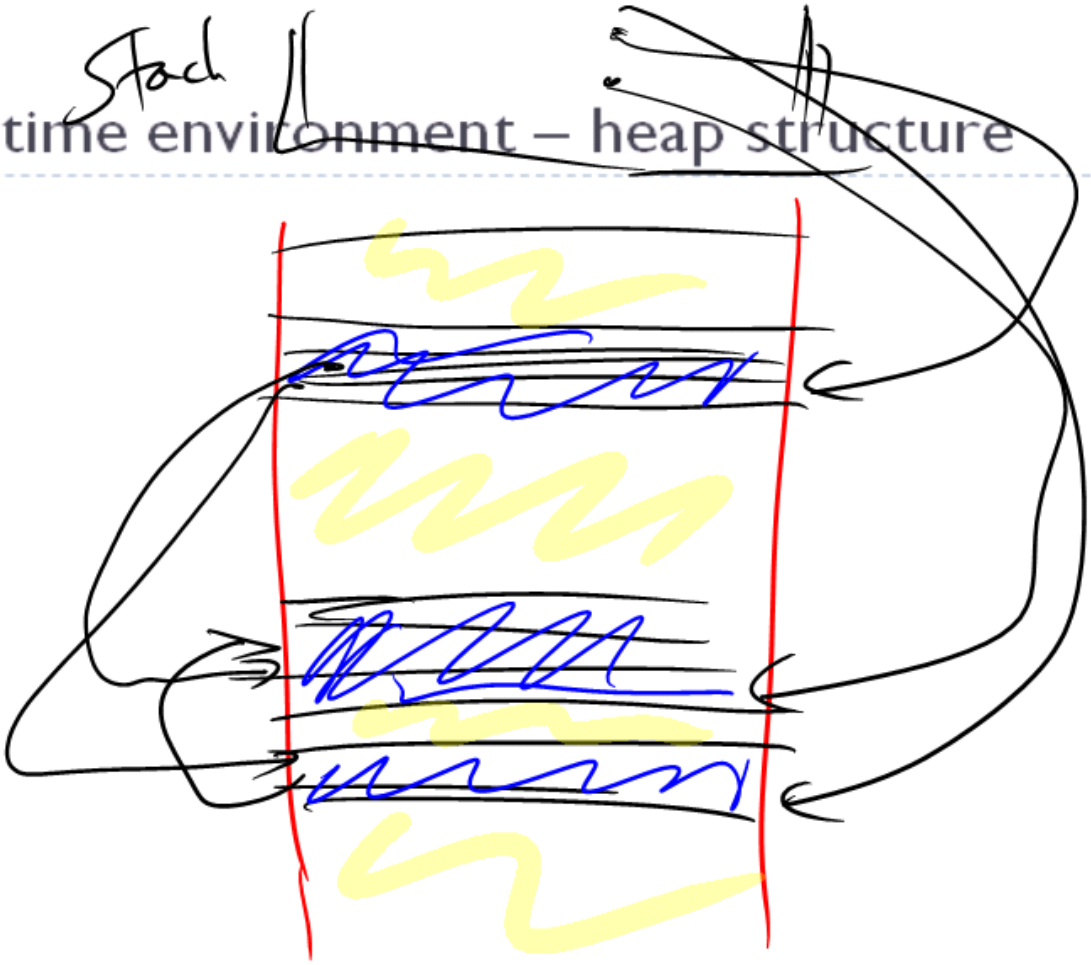
- **Lecture 3**

Run-time environment – stack structure



- Lecture 3

Run-time environment – heap structure



Code optimization - example

- Just to show effect of code optimization, here's a C program:

```
f () {
   int i, j, k;

   i = (j+1)*(k-1);

   printf("%d", i);

}
```

Code produced from C compiler

```
_f:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $24, %esp
        movl    -8(%ebp), %edx
        incl    %edx
        movl    -12(%ebp), %eax
        decl    %eax
        imull   %edx, %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $LC0, (%esp)
        call    _printf
        leave
        ret
```

Code produced from C compiler with –O4

```
_f:
        pushl   %ebp
        decl    %edx
        incl    %eax
        imull   %edx, %eax
        movl    %esp, %ebp
        subl    $8, %esp
        movl    $LC0, (%esp)
        movl    %eax, 4(%esp)
        call    _printf
        leave
        ret
```

Translation to IR

- Different types of intermediate representations
  - Stack machine
  - 3-address instructions
  - 2-address instructions
  - Various graph structures showing control flow and data dependencies
- Consider translation to 3-address form:
  - [S] : Statement -> instruction list
  - [e] : Expression -> instruction list * location
  - (At this stage, are not thinking about machine registers. Just give every location a name. In later stage, decide whether value will go in memory, in register, or on stack.)

Translation to IR

- Will give a number of translation schemes, showing how to translate different expressions and statement to intermediate form. (We will not translate to any actual machine language, but machine languages are like our IR, just more complicated.)
- Will present code sequences either stacked vertically (as is usually done for assembly language), or horizontally separated by semicolons:

  $instr_1$
  $instr_2$
  ...
  $instr_n$

  or $instr_1; instr_2; \ldots\ instr_n$
- Will often write [e] or [S] in the middle of an instruction sequence: splice the instructions given by [e] or [S] into the instruction sequence.

Translation to IR

- Here, assume a three-address IR with machine instruction-like instructions (but simpler). These include:

  loc = loc loc = - loc

  loc = loc + loc (or -, *, /, <, >, ==, &&, ||, etc. All including boolean operations – operate on numbers)

  JUMP label

  CJUMP v, label1, label2 (jump if v = 1)

  PUSH v (push value or loc onto stack)

  CALL f (jump to function f, after adjusting pointers)

  RET

  LOADIND v (load from memory location given by v)

References to variables implicitly get them from the stack.

Referencing variables

References to variables implicitly get values from the stack.

Unlike real machine language, will not use explicit offsets from frame pointer – but that is how these references would be implemented in machine language.

In machine language, inside function definition

```
void f ( int x, double y, ...) { char c; ... }
```

x, y, c, etc. would occupy specific locations in the stack frame (chosen by compiler). References in the body of f would use those offsets. As a simplification, we will simply refer to their locations as x, y, c, etc.

Translation to IR

- **Expressions**. Recall, [e] return a pair consisting of a sequence of (zero or more) instructions, and a location.
- [n] (n a constant) = let t = newlocation()
  in ("t = n", t)
- [x] (x a variable) = ("", x)
- [e1 + e2] = let (l_1, t_1) = [e1]
  (l_2, t_2) = [e2]
  t_3 = new location()
  in (l_1 ; l_2 ; $t_3 = t_1 + t_2$, t_3)

x + 1

[x] = ("", x)

[1] = ("t=1", t)

[t=1; $t_3 = x + t$]

Translation to IR

- **Statements:**

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

- [{S1; S2; . . . ; Sn}] = [S1]
  [S2]
  .
  .
  .
  [Sn]

Translation to IR

- [if e then S1 else S2] =

```
let (l, t) = [ e ]
    L1, L2, L3 = newlabels()
in  l
     CJUMP t, L1, L2
L1:  [S1]
     JUMP L3
L2:  [S2]
L3:
```

Translation to IR

- [while e do S1] =

```
let (l, t) = [ e ]
    L1, L2, L3 = newlabels()
in  JUMP L2
L1: [S1]
L2: l
    CJUMP t, L1, L3
L3:
```

```
while (x > 0) {
    y = y + x;
    x = x - 1;
}
```

```
while (x > 0) {
    y = y + x;
    x = x - 1;
}
```

$[x>0] = [x] = (\text{"}\text{"}, x)$

$[0] = (\text{"}t_1 = 0\text{"}, t_1)$

$\left(\boxed{\begin{array}{l} t_1 = 0 \\ t_2 = x > t_1 \end{array}}, t_2 \right)$

$[y = y + x;\ x = x - 1] =$

$[y = y + x] = [y + x] = (t_3 = y + x, t_3)$

$\boxed{t_3 = y + x;\ y = t_3}$

$[x = x - 1] \rightsquigarrow \boxed{t_4 = 1;\ t_5 = x - t_4;\ x = t_5}$

```
        Jump L2
L1:     t3 = y + x
        y = t3
        t4 = 1
        t5 = x - t4
        x = t5
L2:     t1 = 0
        t2 = x > t1
        (Jump t2, L1, L3)
L3:
```

Translation to IR

- [f(e1, …, en)] =

  let (l_i, t_i) = [ei], for all i

  in l_1

  PUSH t_1

  .

  .

  l_n

  PUSH t_n

  CALL f