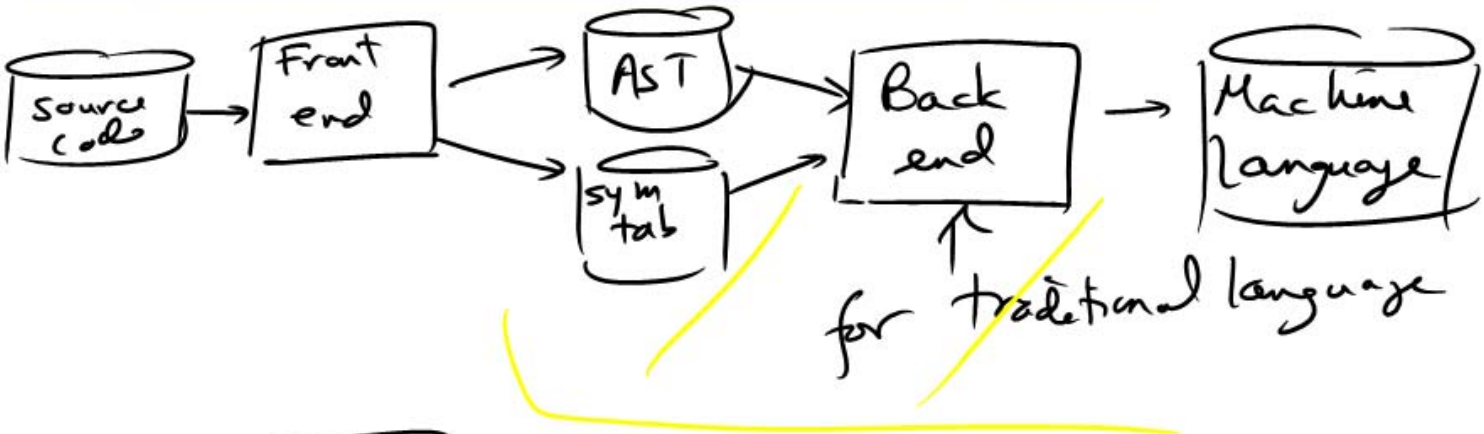
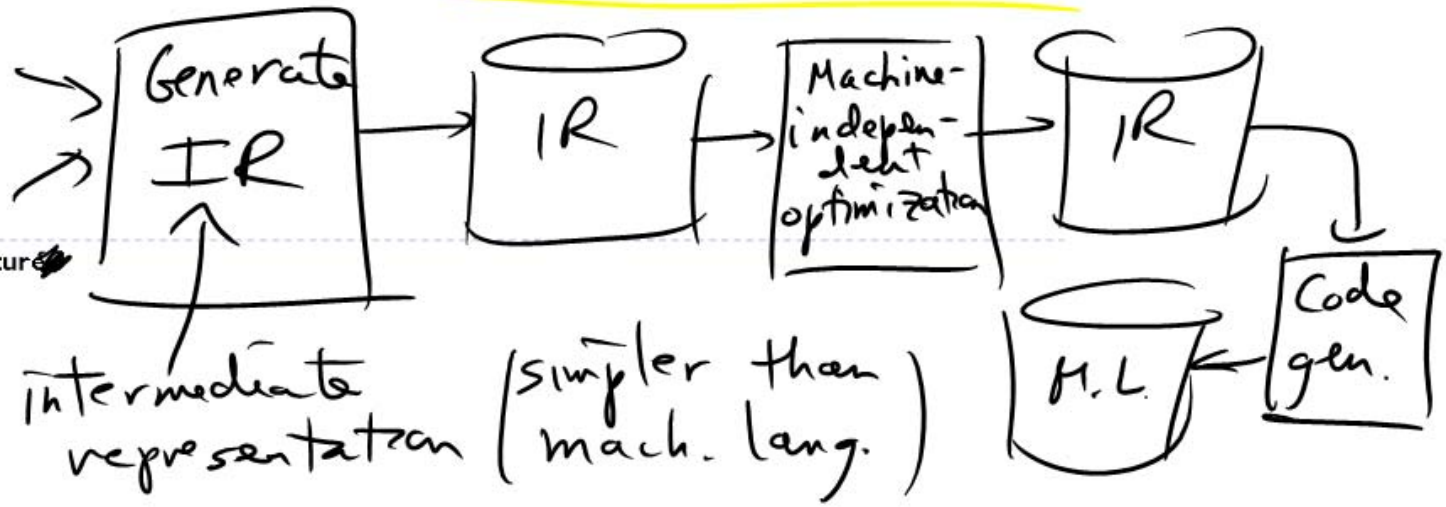
CS 421 Lecture 11

- Compilation and execution – static languages
  - Compilers
  - Execution of static languages
  - Code optimization – why?
  - Code generation
  - Code optimization – how?
- Wednesday's class: dynamic languages – code generation, garbage collection, reflection

- Lecture 6

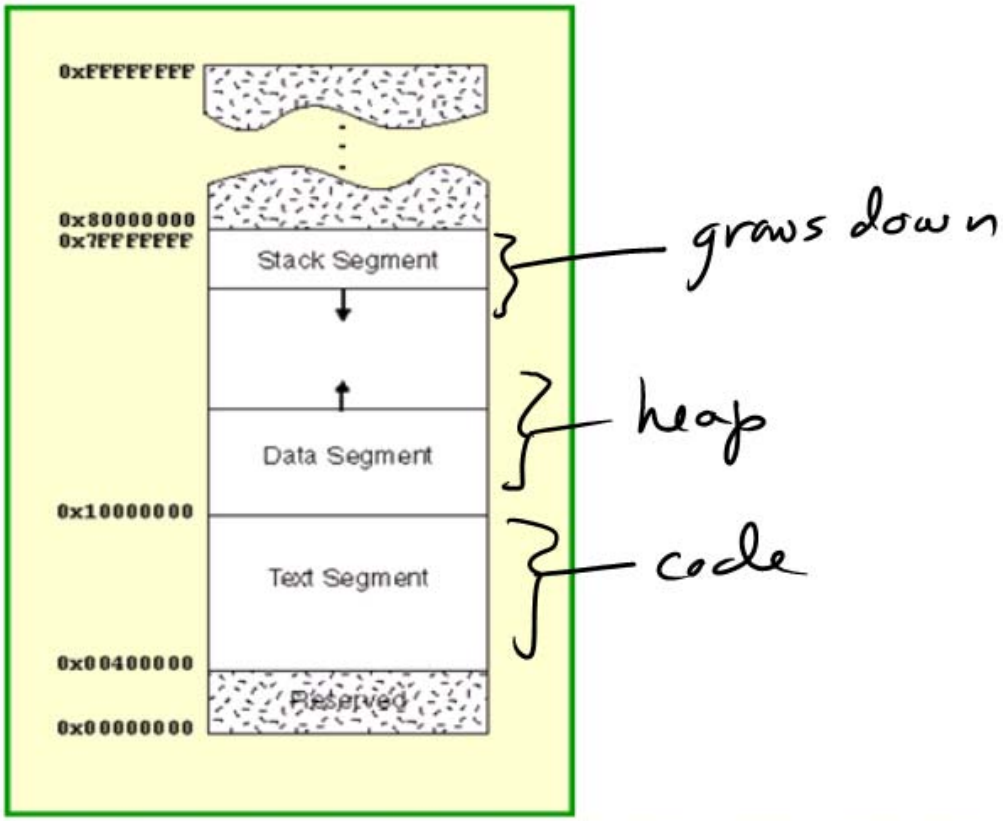
Compiler structure





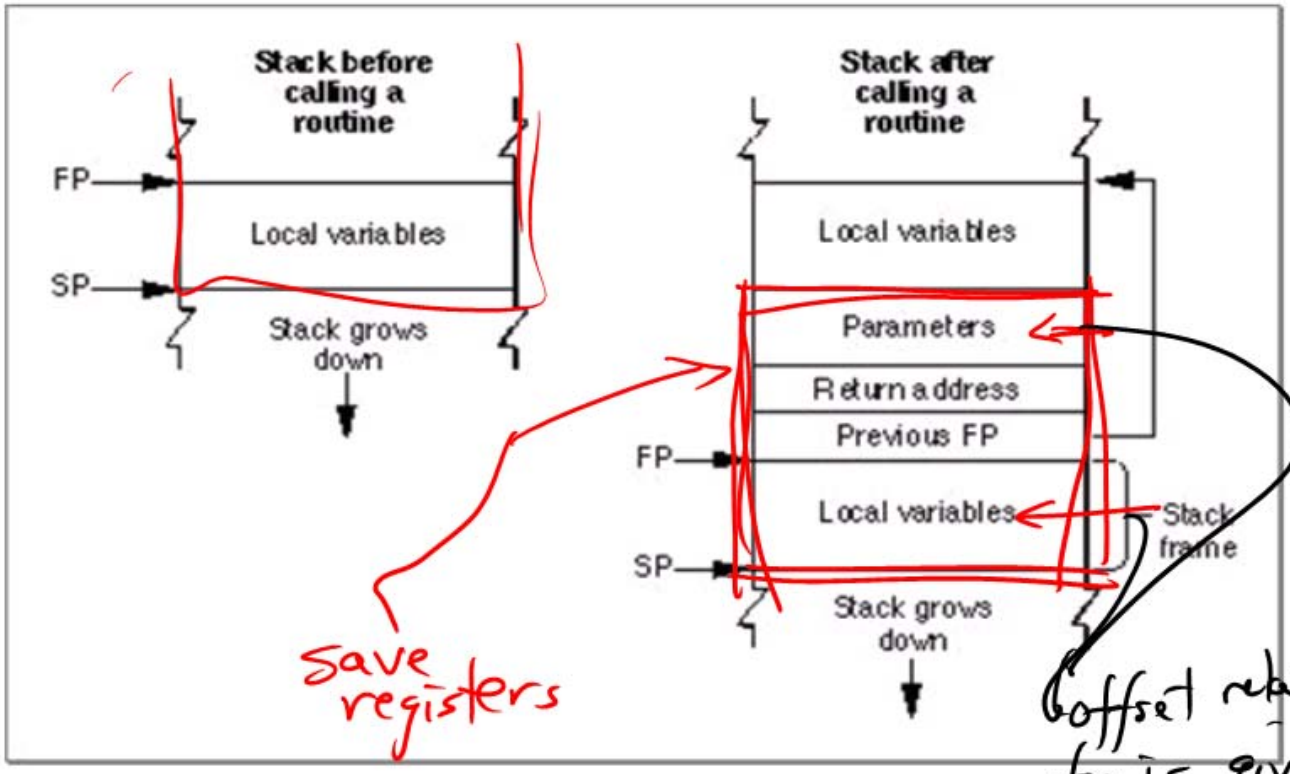
- Lecture

Run-time environment – memory layout

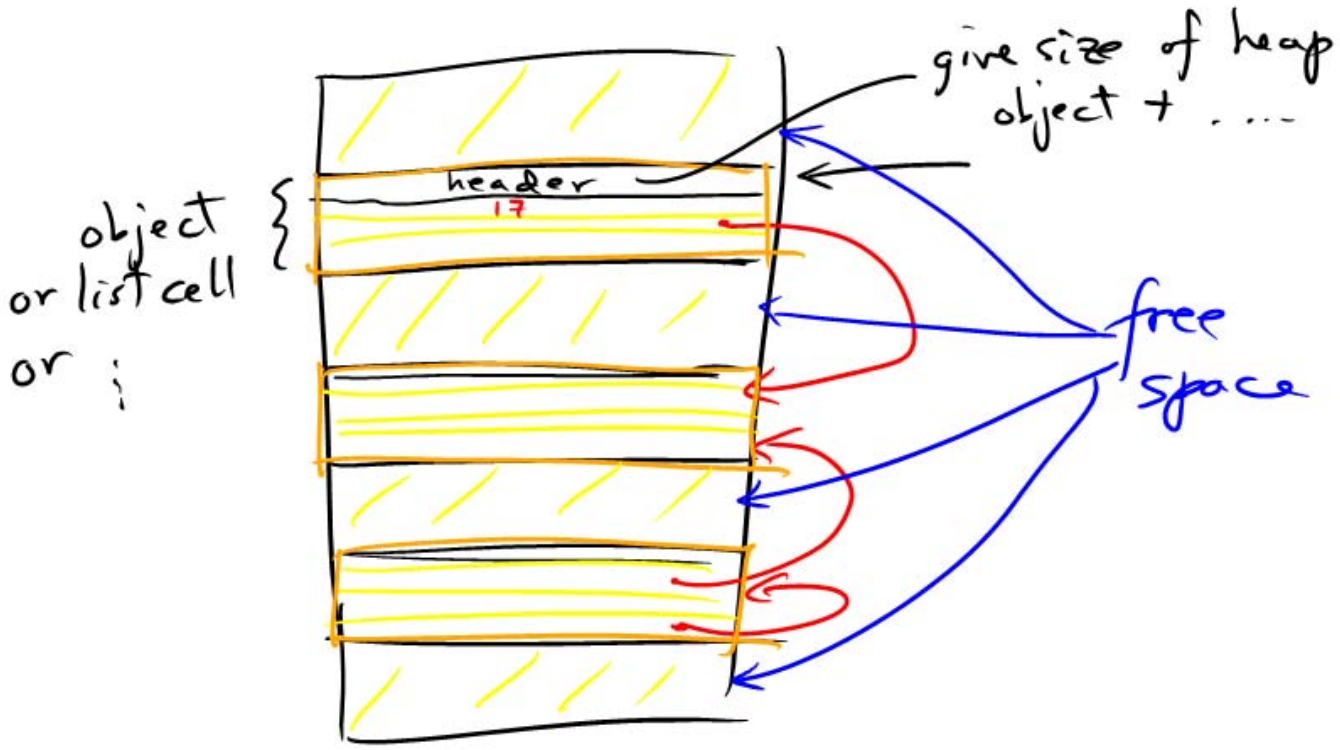


- Lecture 3

Run-time environment – stack structure



Run-time environment – heap structure



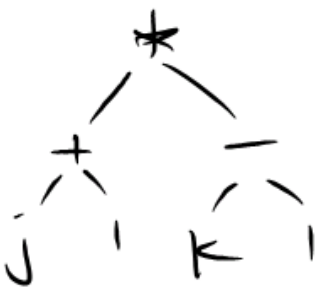
Code optimization - example

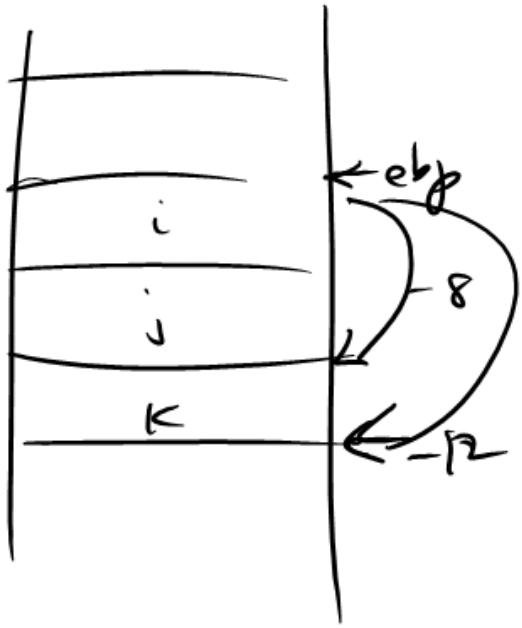
- Just to show effect of code optimization, here's a C program:

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

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

   printf("%d", i);

}
```

Code produced from C compiler

```
leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $36, %esp
        movl    -12(%ebp), %edx
        addl    $1, %edx
        movl    -8(%ebp), %eax
        subl    $1, %eax
        imull   %edx, %eax
        movl    %eax, -16(%ebp)
        movl    -16(%ebp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
```





Code produced from C compiler with –O4

```
leal    4(%esp), %ecx
        andl    $-16, %esp
        pushl   -4(%ecx)
        addl    $1, %eax
        leal    -1(%eax), %edx
        imull   %edx, %eax
        pushl   %ebp
        movl    %esp, %ebp
        pushl   %ecx
        subl    $20, %esp
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
```

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 * variable
  - (At this stage, are not thinking about machine registers. Just give every value a location name. In later stage, decide whether value will go in memory, in register, or on stack.)

Translation to machine language

- **Expressions**:

- [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 , t_3)
  l_2
  $t_3 = t_1 + t_2$

$[\![x]\!] = (\epsilon, x)$

$[\![e_1 + e_2]\!] = \text{let } (I_1, t_1) = [\![e_1]\!]$
$(I_2, t_2) = [\![e_2]\!]$
$x = \text{newlocation}()$
in $\begin{pmatrix} I_1 \\ I_2 \\ x = t_1 + t_2 \end{pmatrix}, x)$

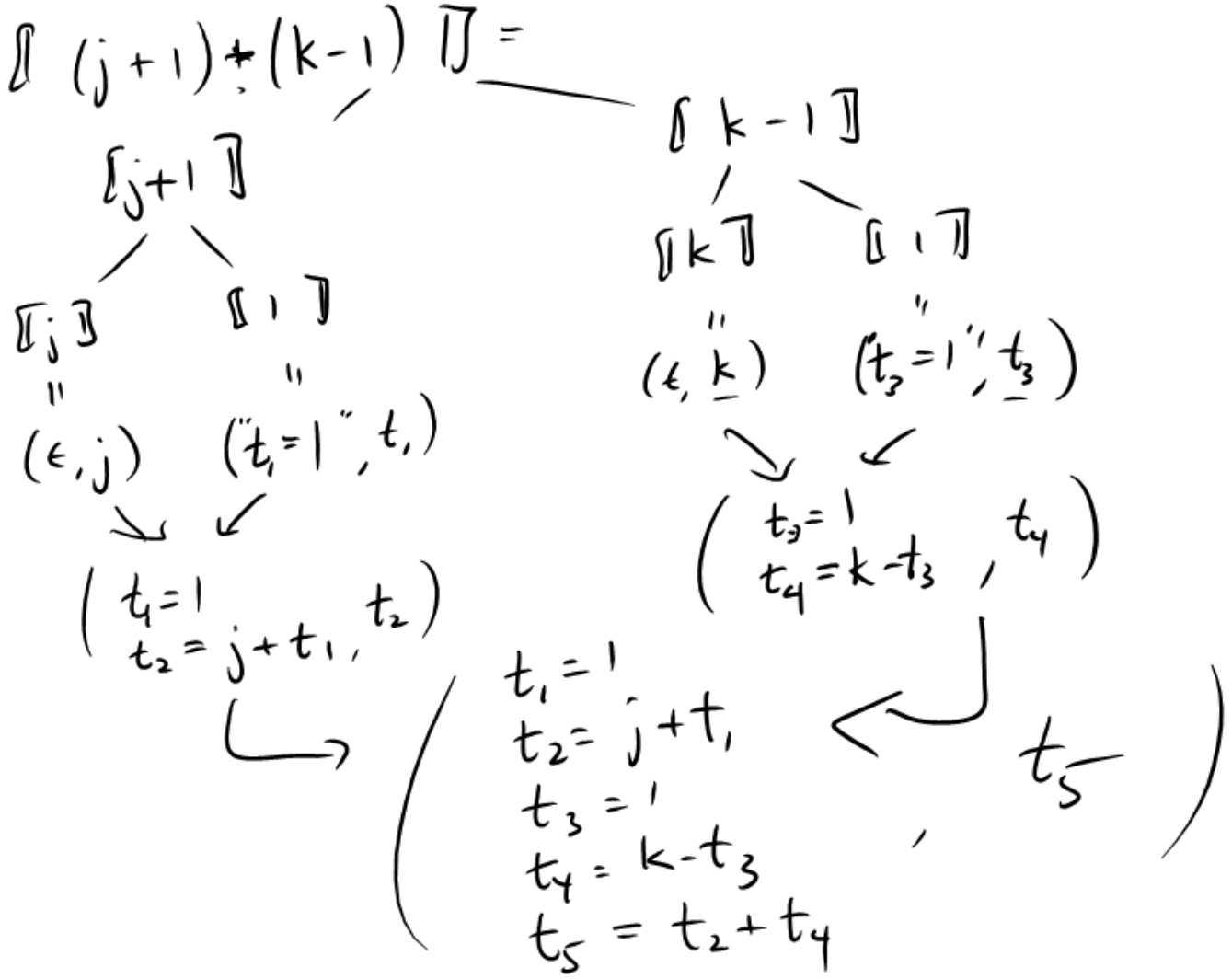
Eg. $x + (10 * y)$

$[\![x]\!] = (\epsilon, x)$

$[\![10 * y]\!] = \begin{cases} [\![10]\!] = ("t_1 = 10", t_1) \\ [\![y]\!] = (\epsilon, y) \end{cases}$ newplace

$\Rightarrow \left(\begin{pmatrix} t_1 = 10 \\ t_2 = t_1 * y \end{pmatrix}, t_2 \right)$

$\left(\begin{pmatrix} t_1 = 10 \\ t_2 = t_1 * y \\ t_3 = x + t_2 \end{pmatrix}, t_3 \right)$


⟦ (j+1) + (k-1) ⟧ =
⟦ j+1 ⟧
⟦ k-1 ⟧
⟦ j ⟧
⟦ 1 ⟧
⟦ k ⟧
⟦ 1 ⟧
(ϵ, j)
("t1 = 1", t1)
(ϵ, k)
("t3 = 1", t3)
t1 = 1
t2 = j + t1
t3 = 1
t4 = k - t3
t5 = t2 + t4

Translation to machine language

- **Statements:**

- [x = e] = let (I, t) = [e]
  in I
  x=t
- [{S1; S2; … ; Sn}] = [S1]
  [S2]
  .
  .
  .
  [Sn]

Translation to machine language

- [if e then S1 else S2] =

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

Translation to machine language

- [while e do S1] =

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

Translation to machine language

- [f(e1, …, en)] =

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

  in I_1

  PUSH t_1

  .

  .

  I_n

  PUSH t_n

  CALL f

What statements are left?

switch

break

array expr's

boolean expr's