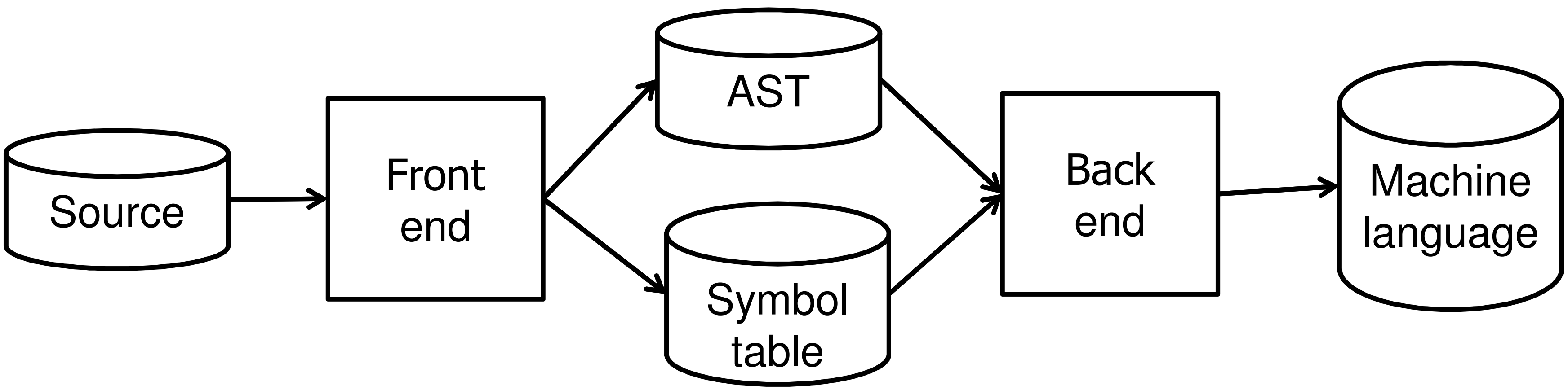
CS 421 Lecture 11: Code generation

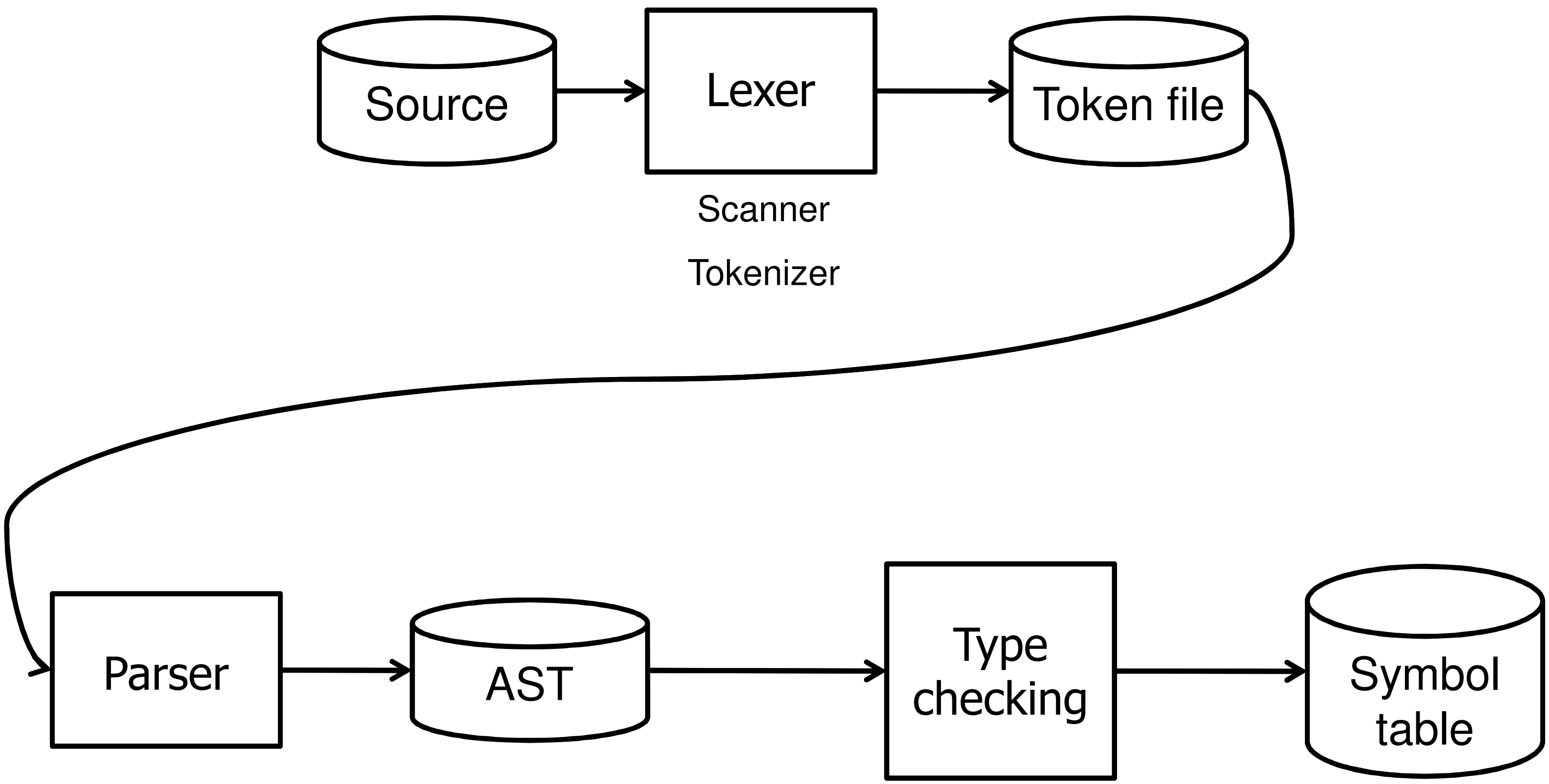
- Lecture outline
  - Compiler structure
  - Run-time environment
  - Execution of static languages
  - Code optimization – why?
  - Code generation
  - Code optimization – how?

Compiler structure

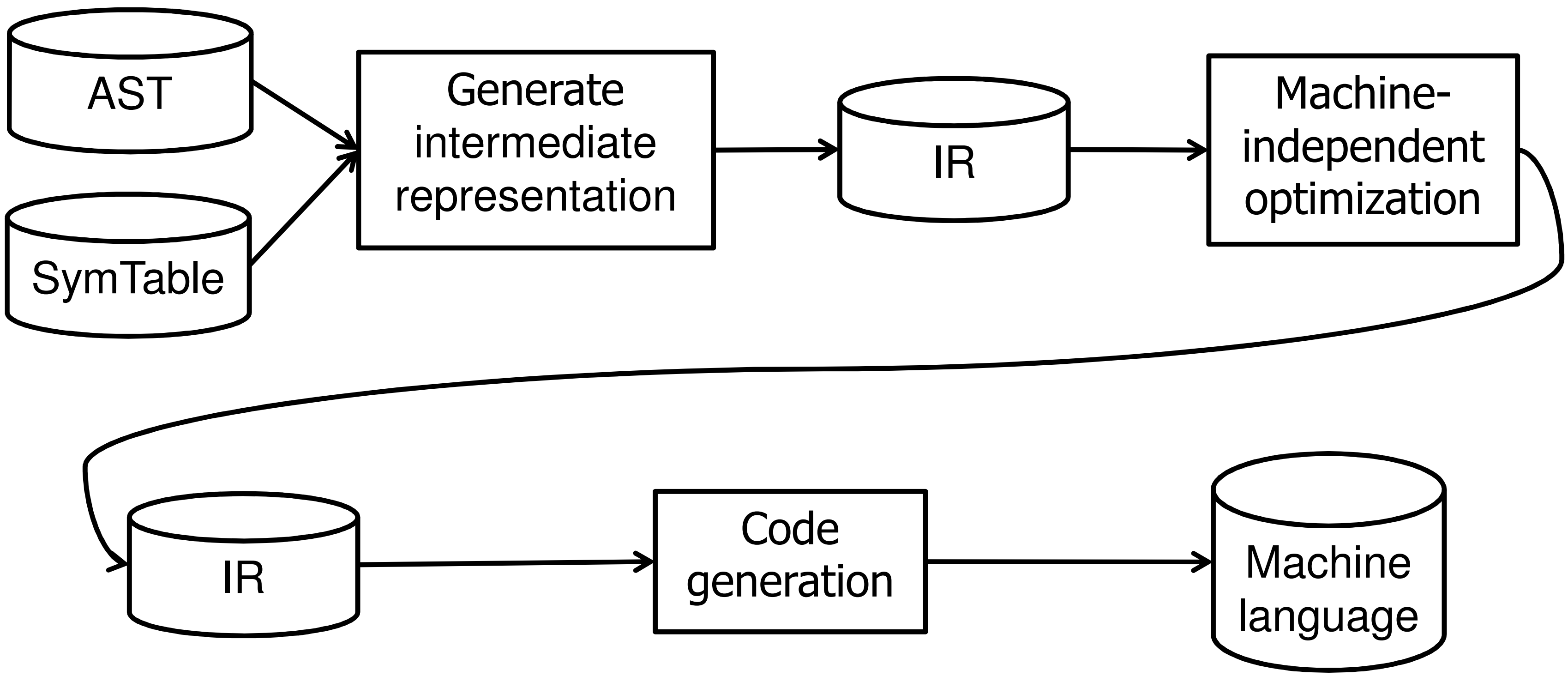
- For traditional (static) languages:



Review: front end



Back end



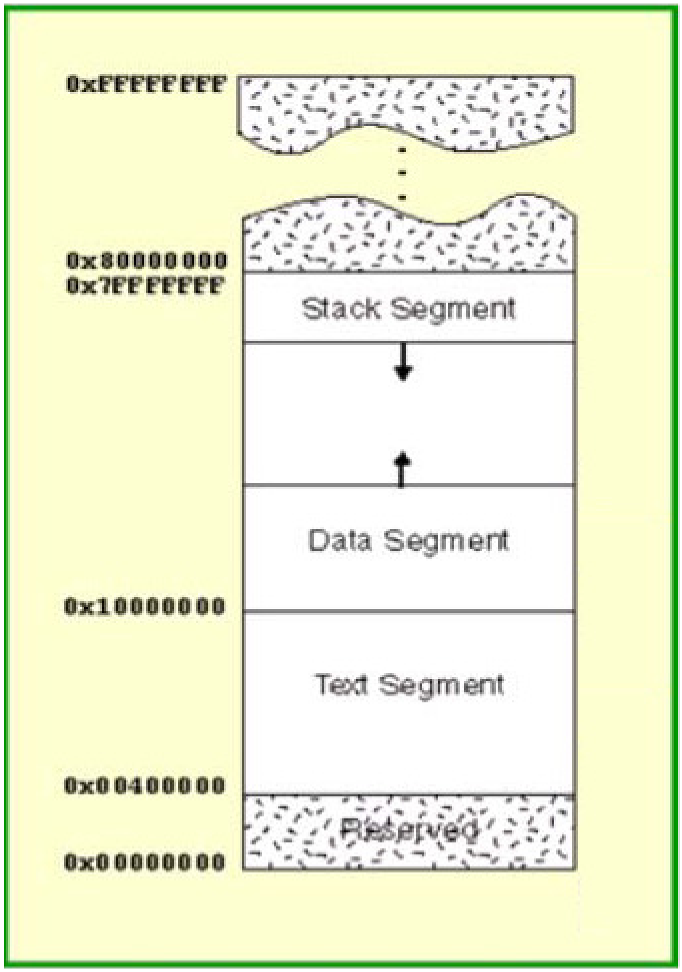
- *Intermediate representation* (IR) is simpler to operate on than machine language

Run-time environment

- Once we have the program in machine code, how do we run it?
  - Instructions executed directly on HW
  - OS calls?
  - Variables?
  - Function/procedure calls?
- Memory architecture
- Information about the program

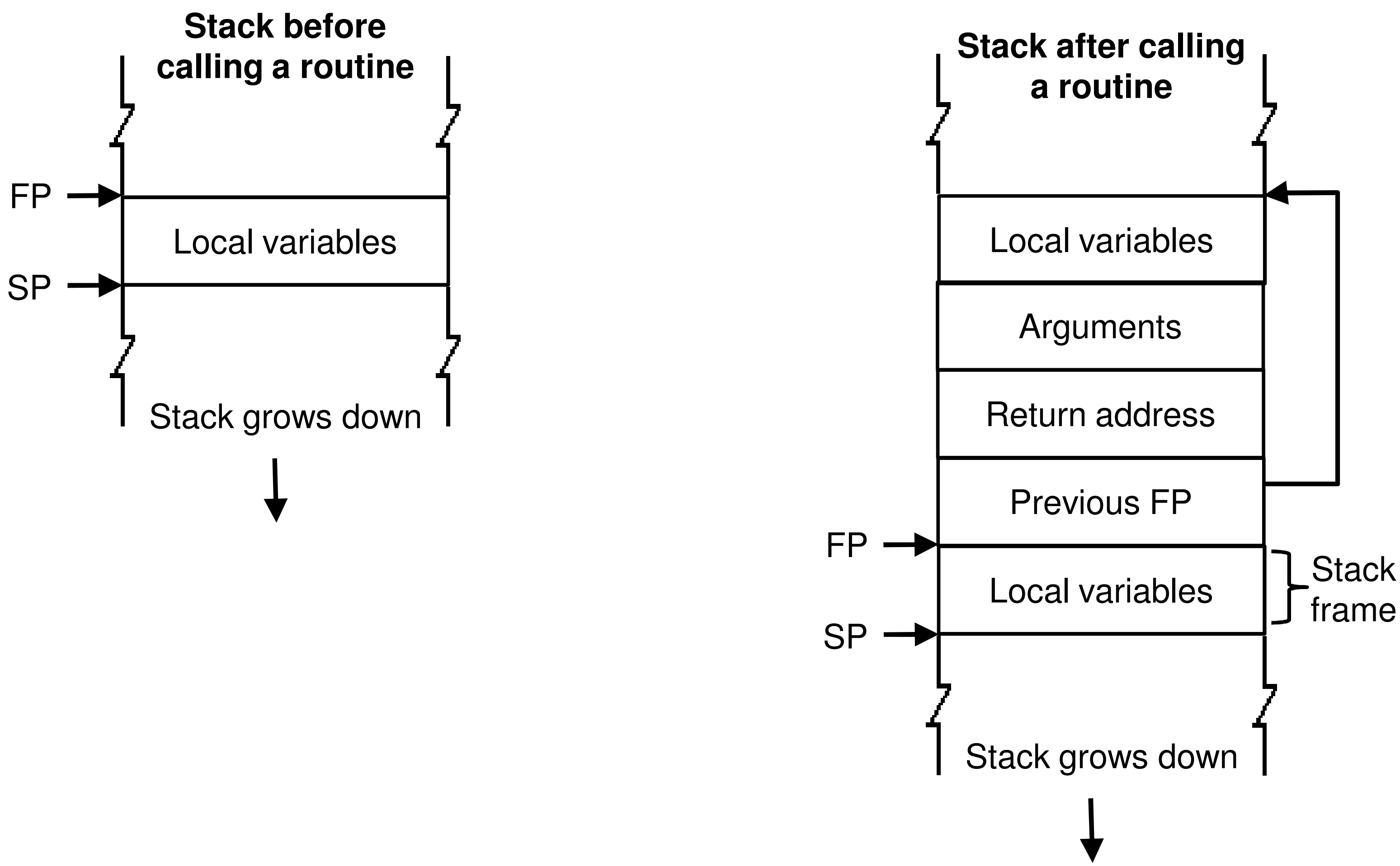
Run-time environment

- Memory layout:



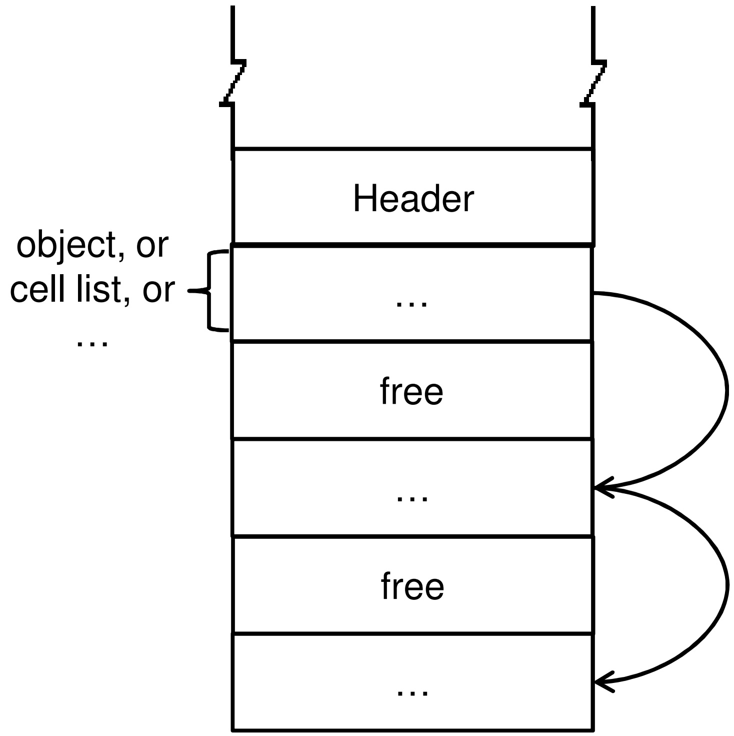
Run-time environment

- Stack structure:



Run-time environment

- Heap structure:



Code optimization – motivating 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 optimization – motivating example

- Machine code produced by 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 optimization – motivating example

- Machine code produced by 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
```

Code optimization

- How can we get there?
  - Go directly from AST+ST -> ML
    - What is the problem?
  - Use intermediate representation
- IR
  - Close enough to machine language that IR -> ML translation is easy
  - Abstracts over some details, platform-specific features, *etc.*

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, we are not thinking about machine registers. Just give every value a location name.
  - In later stage, decide whether value will to in memory, in register, or on stack.

Translation to machine language

- Expressions:

  - ```
    [ n ] (n is a constant) = let t = newloc()
                              in (t = n, t)
    ```
  - ```
    [ x ] (x is a variable) = (ε , x)
    ```
  - ```
    [ e1 + e2 ] = let (I1, t1) = [ e1 ]
                      (I2, t2) = [ e2 ]
                      t3 = newloc()
                  in (I1          , t3)
                      I2
                      t3 = t1 + t2
    ```
    ```

Example 1

- ```
  [ n ] = let t = newloc() in (t = n, t)
  ```
- ```
  [ x ] = ( ε , x )
  ```
- $[\ e1 + e2\ ] = \text{let } (I_1, t_1) = [\ e1\ ], (I_2, t_2) = [\ e2\ ]$
  $\qquad t_3 = \text{newloc}\ ()$
  $\quad \text{in } (I_1 \qquad\qquad , t_3)$
  $\qquad I_2$
  $\qquad t_3 = t_1 + t_2$

- x + (10 * y)

Example 2

- $[\,(j + 1) * (k - 1)\,]$

Translation to machine language

- Statements
- Assignment
  - ```
    [ x = e ] = let (I, t) = [ e ]
                in   I
                     x = t
    ```
- Sequence (block)
  - ```
    [ {S1; S2; … ; Sn} ] = [ S1 ]
                           [ S2 ]
                            ...
                           [ S3 ]
    ```

Translation to machine language

- If-then-else

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

Translation to machine language

- While

    - ```
      [ while e do S1 ] =
                  let (I, t) = [ e ]
                     L1, L2, L3 = newlabels()
                  in   ??
      ```
      ```

Translation to machine language

- While

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

Translation to machine language

- Procedure call
  - ```
    [ f(e1, … , en) ] =
              let (Iᵢ, tᵢ) = [ eᵢ ] for all I
              in   I₁
                   PUSH t₁
                   I₂
                   PUSH t₂
                   ...
                   Iₙ
                   PUSH tₙ
                   CALL f
    ```
    ```

What is left?

- Next class
  - Finish up statements
    - Switch
    - Break
  - Finish up expressions
    - Arrays
    - Booleans
  - Implementing code optimization