

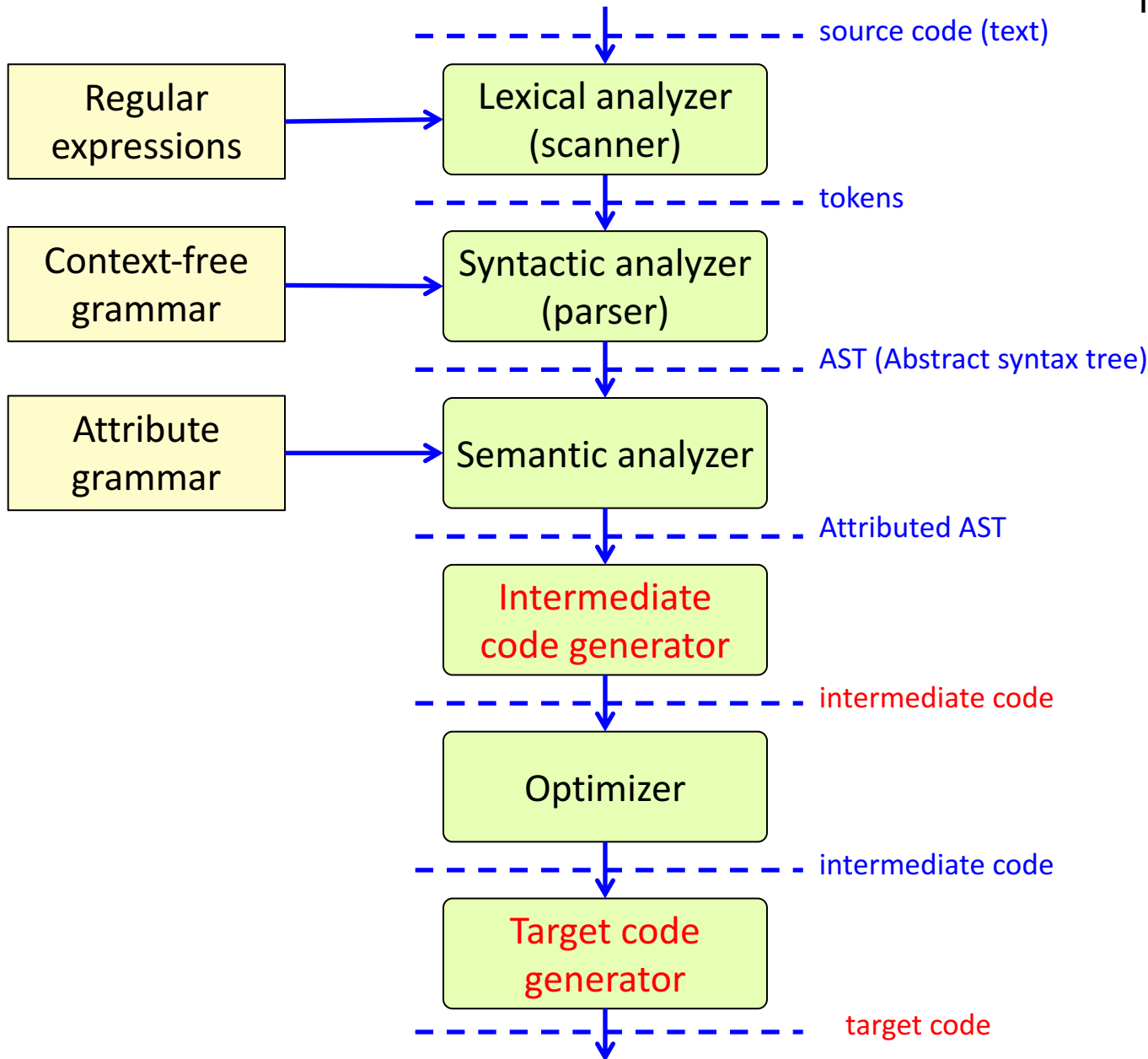
EDAN65: Compilers, Lecture 12

# More on code generation

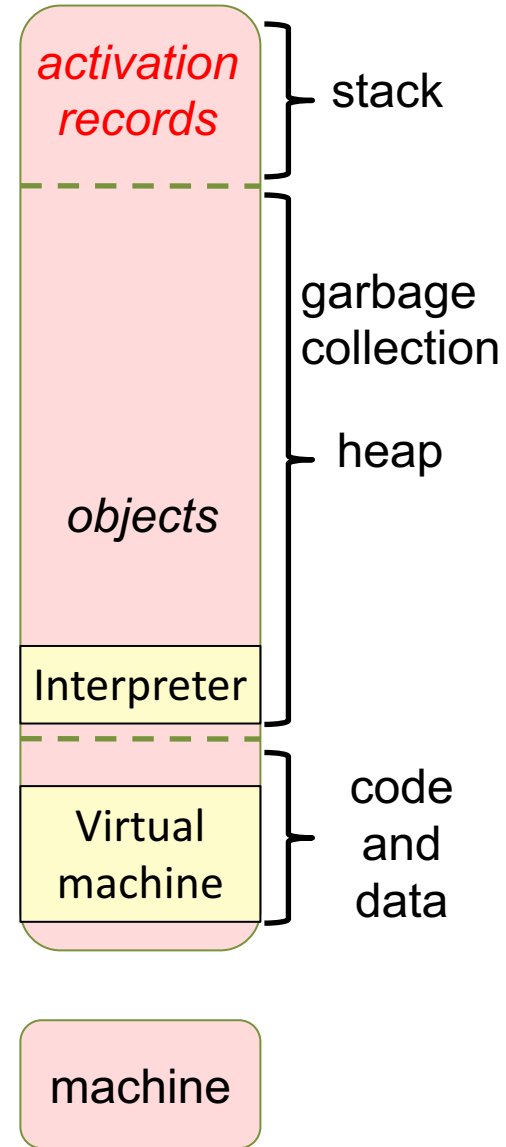
Görel Hedin

Revised: 2017-10-03

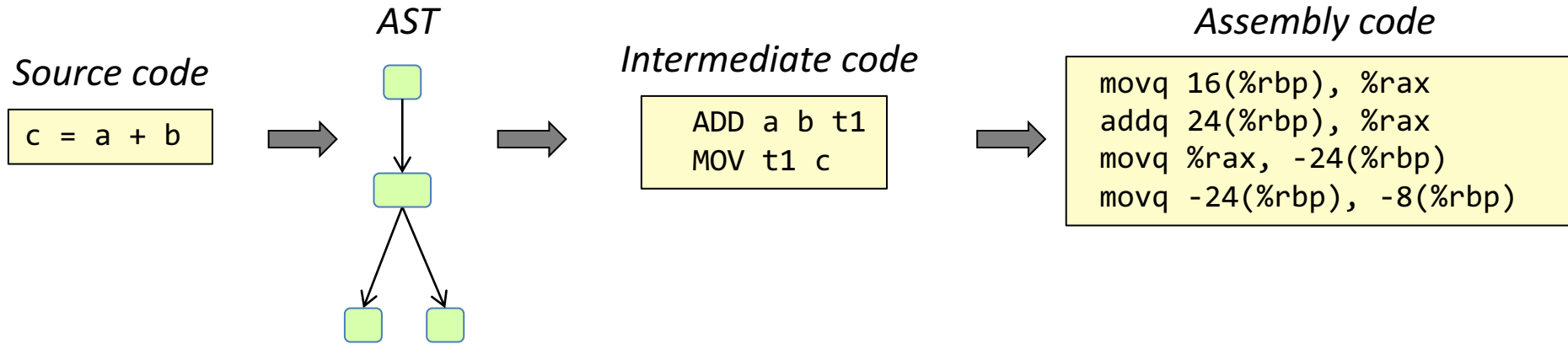
# This lecture



## runtime system



# Generating code



## Intermediate code:

- Where most optimizations are done

## Assembly code:

- For given machine, operating system, assembler, and calling conventions

## In assignment 6

- Generate AT&T assembly code for x86-64, using simple calling conventions
- No intermediate code – we generate the assembly code directly from the AST.

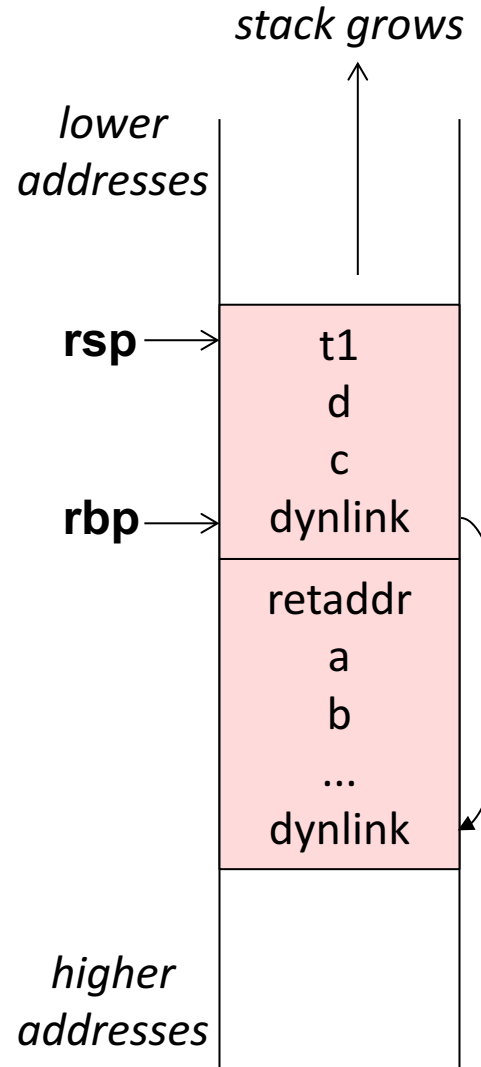
# AT&T x86-64 assembly code

## Source code

```
void m(int a, int b) {  
    int c, d;  
    ...  
    c = a + b  
    ...  
}
```

## Assembly code

```
m:                # Label for the method  
    pushq %rbp    # Push the dynamic link  
    movq %rsp, %rbp # Set the new frame pointer  
    subq $24, %rsp # Make room on stack for c, d, t1  
    ...  
    movq 16(%rbp), %rax # a -> rax  
    addq 24(%rbp), %rax # b + rax -> rax  
    movq %rax, -24(%rbp) # rax -> t1  
    movq -24(%rbp), -8(%rbp) # t1 -> c  
    ...  
  
    movq %rbp, %rsp # Move back the stack pointer  
    popq %rbp      # Restore the frame pointer  
    ret           # Return to the calling method
```



# Generating code for different constructs

**Method activation and return**, setting up a new frame, restoring it

**Expression evaluation**, using temporaries, local variables, formal arguments

**Control structures**, labels and jumps

**Method calls**, passing arguments and return values

# Generating code using attribute grammars

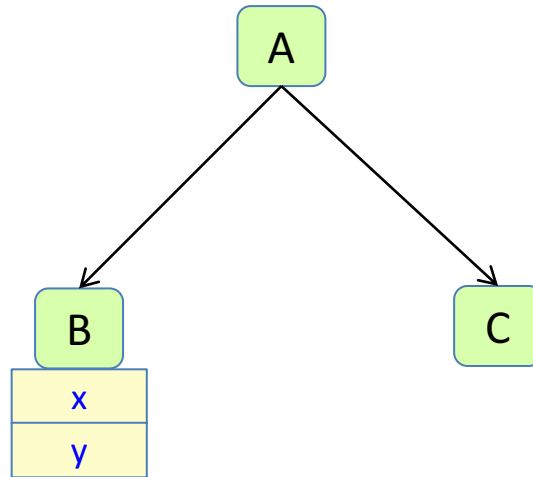
Main idea:

- Recursive traversal of the AST using a method  
void genCode(PrintStream)
- Use attributes for everything that is not straightforward.

```
// Example compiler program
public class Compiler {
    public static void main(String[] args) {
        Program p = ... // Parse in program to compile
        ... // Check for compile time errors
        p.genCode(System.out);
    }
}
```

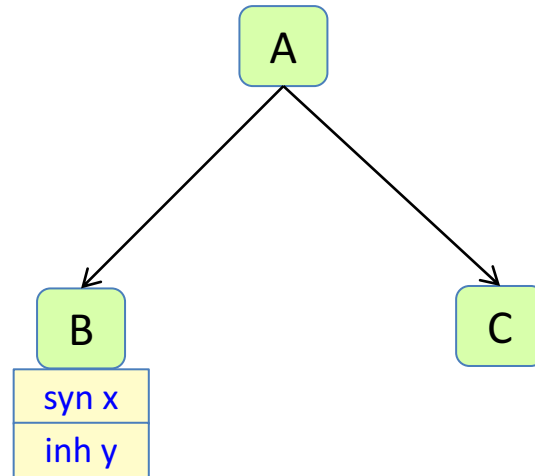
```
// Example genCode method
void Assignment.genCode(PrintStream out) {
    getRight().genCode(out);
    out.println("movq " + getRight().address() + ", " + getLeft().address());
}
```

# Attributes: think declaratively!



- What do you want to compute?
- What properties would allow you to easily compute that? Declare as attributes.
- Make an attribute synthesised if it depends on information in the node.  
Inherited if it only depends on context.
- When defining an attribute, make up new properties/attributes that would make it easy to compute.
- Don't think about the order of computation.

# Synthesized/Inherited attributes



```
// Declare syn attribute:  
syn B.x();  
  
// Equation defining syn attr  
eq B.x() = ...;
```

```
// Declare inh attribute  
inh B.y();  
  
// Equation defining inh attr  
eq A.getB().y() = ...;
```



# Explicit temps

(like in previous lecture)

## Source code

```
a = b * (c + d)
```

## Variable addresses

a	-8(%rbp)
b	-16(%rbp)
c	-24(%rbp)
d	-32(%rbp)
t1	-40(%rbp)
t2	-48(%rbp)

## x86 assembly code, explicit temps

```
movq  -24(%rbp), %rax    # c -> rax
addq  -32(%rbp), %rax    # d + rax -> rax
movq  %rax,          -40(%rbp) # rax -> t1
movq  -16(%rbp), %rax    # b -> rax
imulq -40(%rbp), %rax    # t1 * rax -> rax
movq  %rax,          -48(%rbp) # rax -> t2
movq  -48(%rbp), -8(%rbp) # t2 -> a
```

Main idea:

- Each nontrivial operation puts its result in a new temp

Code generation for binary operation:

- generate code for left op (result at some address)
- generate code for right op (result at some address)
- move left op to %rax
- perform operation on right op and %rax
- move %rax into new temp

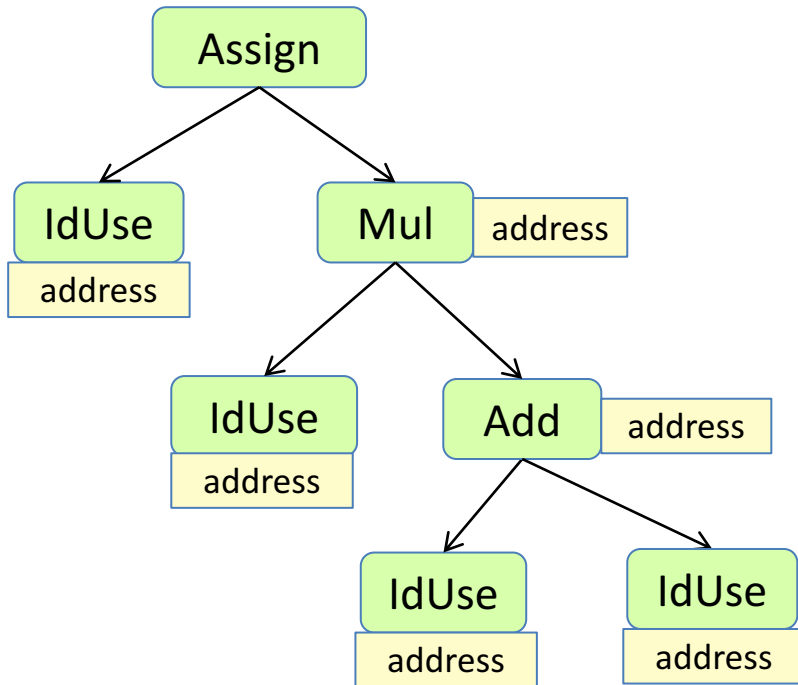
Code generation for assignment:

- generate code for right-hand side (result at some address)
- move result to left var

Code generation for ldUse:

- No code needed.

# Example code generation with explicit temps



```
syn String Exp.address();
```

```
public void Mul.genCode(PrintStream s) {  
    getLeft(). genCode(s);  
    getRight(). genCode(s);  
    s.println("movq " + getLeft().address() + ", %rax");  
    s.println("imulq " + getRight().address() + ", %rax");  
    s.println("movq %rax, " + address());  
}
```

```
public void IdUse.genCode(PrintStream s) { }
```

# Stack of temps

an alternative to explicit temps

## Source code

```
a = b * (c + d)
```

## Variable addresses

```
a    -8(%rbp)
b    -16(%rbp)
c    -24(%rbp)
d    -32(%rbp)
```

Main idea:

- each expression puts its result in rax

Code generation for binary operation

- generate code for left op (result in rax)
- push rax
- generate code for right op (result in rax)
- pop left op into rbx
- op rbx rax (result in rax)

## x86 assembly code, temps on stack

```
movq  -16(%rbp), %rax    # b -> rax
pushq %rax              # push rax
movq  -24(%rbp), %rax    # c -> rax
pushq %rax              # push rax
movq  -32(%rbp), %rax    # d -> rax
popq  %rbx              # pop -> rbx
addq  %rbx, %rax        # rbx + rax -> rax
popq  %rbx              # pop -> rbx
imulq %rbx, %rax        # rbx * rax -> rax
movq  %rax, -8(%rbp)    # rax -> a
```

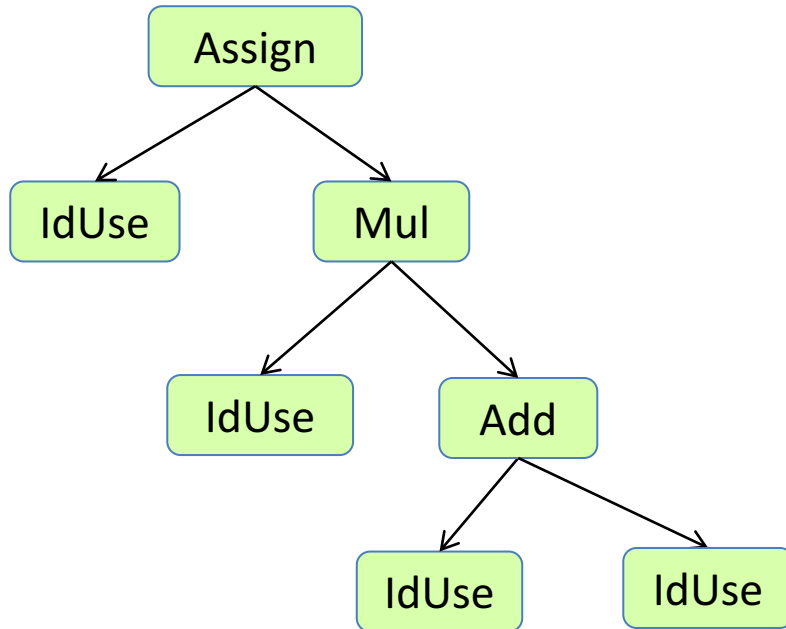
Code generation for assignment:

- generate code for right-hand side (result in rax)
- move rax to left var

Code generation for ldUse:

- move value into rax

# Example code generation with stack of temps



```
syn String IdDecl.address();
```

```
public void Mul.genCode(PrintStream s) {
    getLeft().genCode(s);
    s.println("pushq %rax");
    getRight().genCode(s);
    s.println("popq %rbx");
    s.println("imulq %rbx, %rax");
}

public void IdUse.genCode(PrintStream s) {
    s.println("movq " + decl().address() + ", %rax");
}
```

# Explicit or stacked temps?

Code generation is simpler for stacked temps – we don't need to compute addresses for temps.

But: to generate code for method calls, we need to evaluate the arguments from right to left, to push them in the appropriate order on the stack. Not all languages allow this.

If evaluating the arguments have side effects, the evaluation order can make a difference.

Some languages, like Java, define the evaluation order on arguments from left to right.

In assignment 6, we will use stacked temps. (For SimpliC it is ok to evaluate the arguments from right to left.)

# Generating code from the AST

Define suitable node properties, using attributes, to make the code generation easy.

Then write the code generation as a recursive method, printing the code to a file.

What properties do we need?

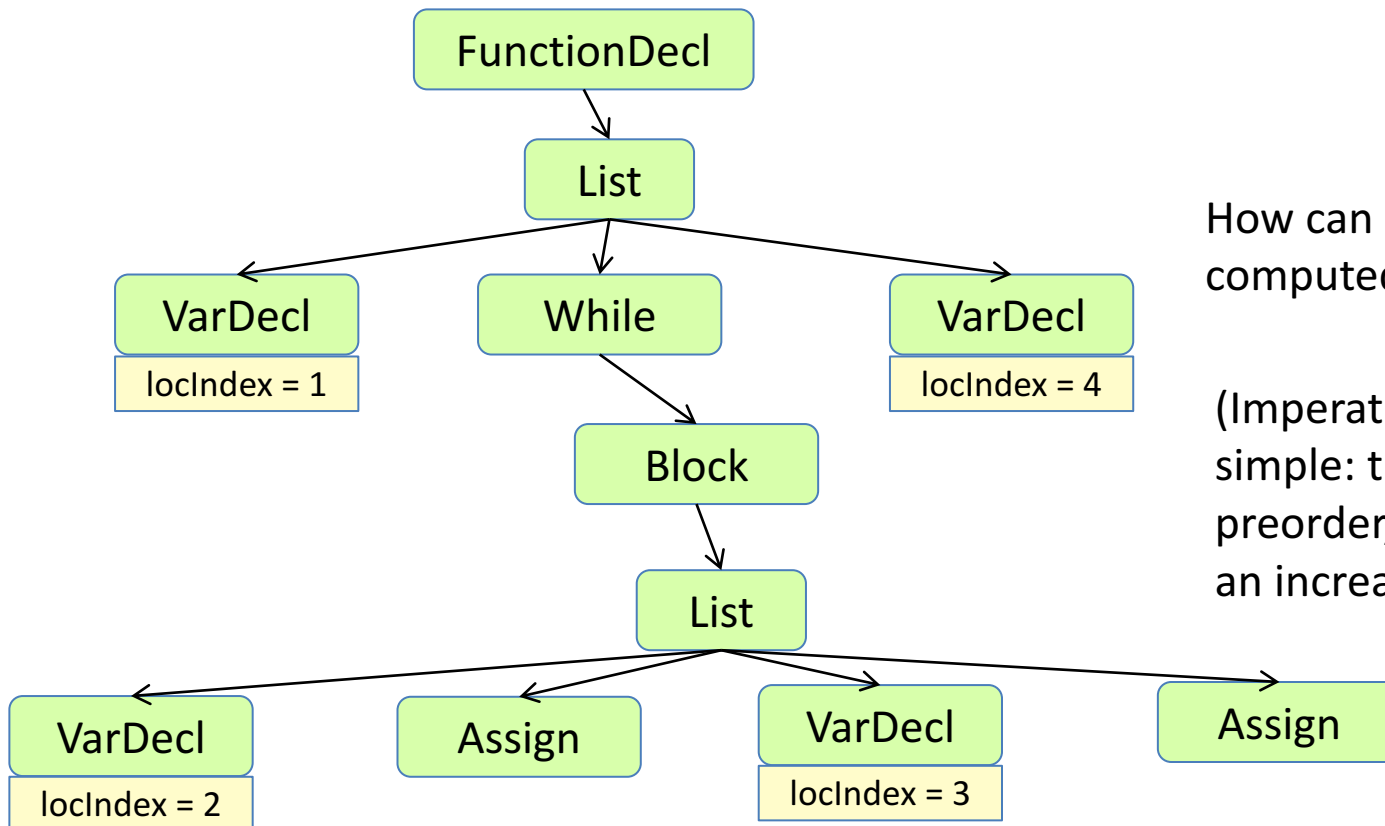
- The address of each variable declaration, for example "-8(%rbp)".
- The number of local variables of a method (to reserve space on the stack).
- The address of each formal argument, for example, "16(%rbp)".
- Unique labels for each control structure.

# Computing addresses of declarations

using attributes

Main idea:

- Enumerate the variable declarations inside each function, giving them local indexes: 1, 2, ...
- Translating to the address is then simple: "-8(%rbp)", "-16(%rbp)", ...



How can locIndex be computed using attributes?

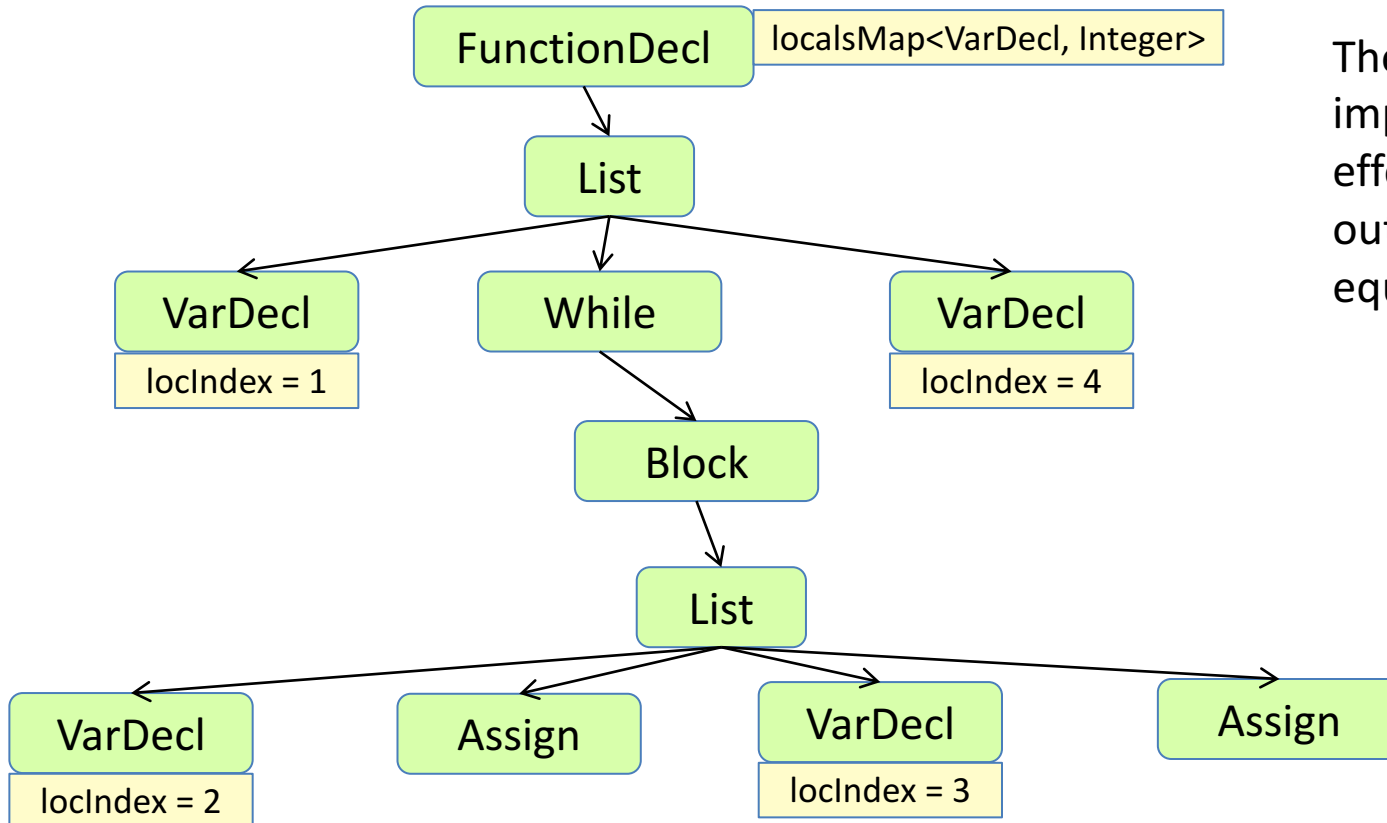
(Imperatively, it would be simple: traverse the tree in preorder, and give each VarDecl an increasing index.)

# Solution using a map attribute

- Define an attribute

```
syn HashMap<VarDecl, Integer> FunctionDecl.localsMap();
```
- Compute it by traversing the function with a method

```
void addLocals(HashMap<VarDecl, Integer> map, Counter c) ...
```
- Use the Root Attribute pattern to give each VarDecl access to the map, and let them look up their index.



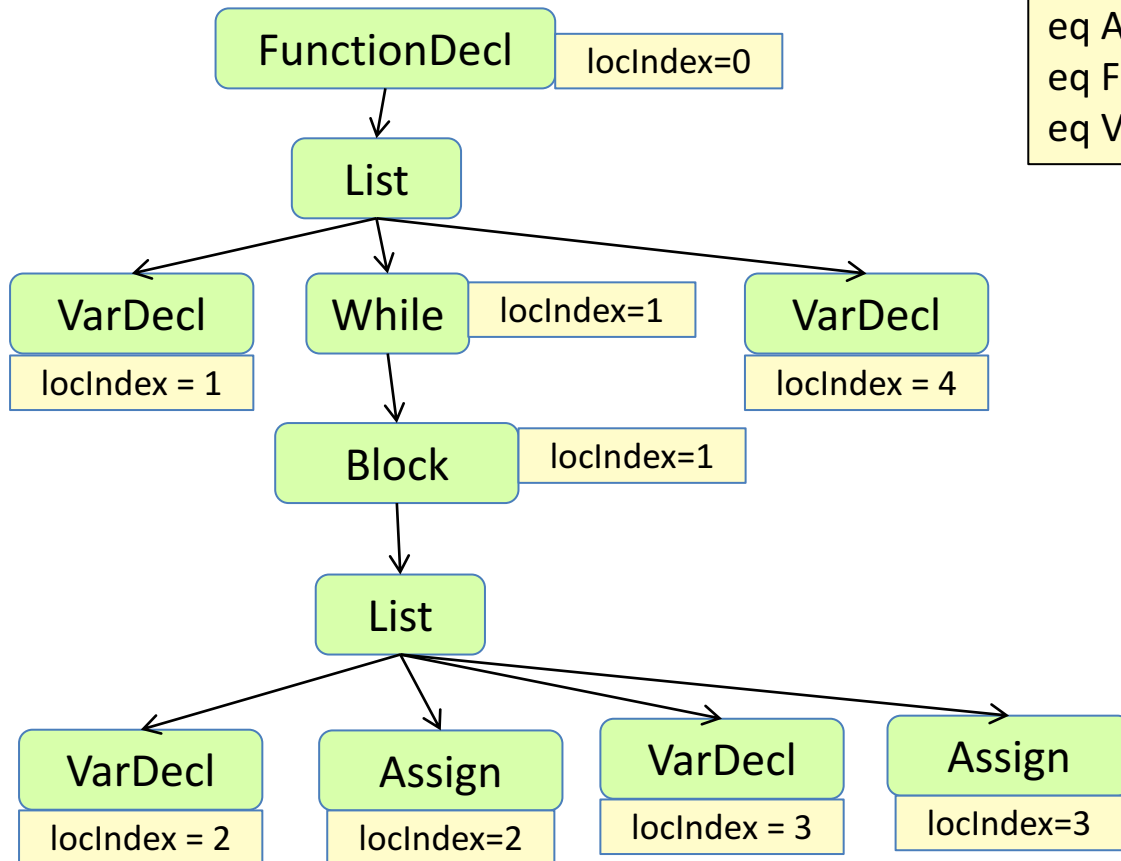
The **addLocals** method is imperative, but the side effects do not escape outside the **localsMap** equation, so that is ok.



# Alternative solution using prev() and last()

Main idea:

- Give *all* nodes a **locIndex**, the locIndex of the latest VarDecl in a preorder traversal.
- Normally the same as for the previous node in the traversal.
- But 0 for the root, and one more for each VarDecl.



```
syn int ASTNode.locIndex();
eq ASTNode.locIndex() = prev().locIndex();
eq FunctionDecl.locIndex() = 0;
eq VarDecl.locIndex() = prev().locIndex() + 1;
```

computing the number of locals

```
syn int FunctionDecl.numLocals() =
    last().locIndex();
```

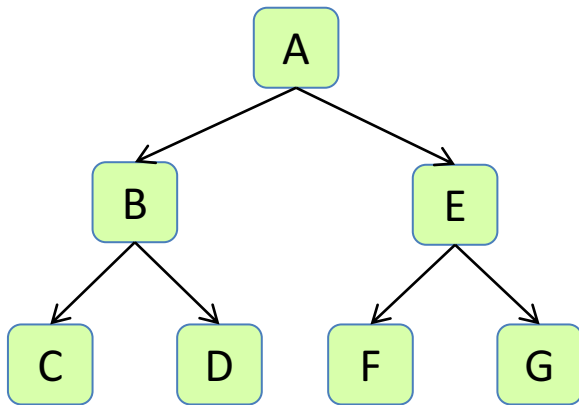
But how are `prev()` and `last()` defined?

# Defining prev() and last()

Preorder traversal: Visit the nodes in the order A, B, C, D, E, F, G

Each node  $n$  has

- a **prev** attribute, the previous node in a preorder traversal.
- a **prev(i)** attribute, the previous node before traversing the  $i$ 'th child of  $n$ .
- a **last** attribute, the last attribute in a preorder traversal of the  $n$  subtree.



```
inh ASTNode ASTNode.prev();  
eq ASTNode.getChild(int i).prev() =  
  prev(i);  
syn ASTNode ASTNode.prev(int i) =  
  i=0 ? this : getChild(i-1).last();  
syn ASTNode ASTNode.last() =  
  prev(getNumChild());
```

$E.\text{prev}() == A.\text{prev}(1) == B.\text{last}() == B.\text{prev}(2) == D.\text{last}() == D.\text{prev}(0) == D$

# Computing unique labels

Main idea:

- Give each statement a "pathname" relative to the function.
- E.g., 3\_2 means the 2nd statement in the 3rd statement in the function.
- Generate labels like m\_3\_2\_whilestart and m\_3\_2\_whileend

```
void m(int a) {  
    int x = 1;  
    int y;  
    while (a > x) {  
        y = a*2;  
        if (y > 3) {  
            ...  
        }  
    }  
}
```

```
m:  
    ...  
m_3_whilestart:  
    ...  
m_3_2_ifend:  
    ...  
m_3_whileend:  
    ...
```

Compute the "pathnames" in a similar way as the unique variable names were implemented in assignment 5.

# An example assembly program

Generated by:

```
.global _start          # global segment
.data                  # data segment (for global data)
...
.text                  # text segment (for code, write protected)
_start:                # execution starts here
    call m1             # call the main function
    movq %rax, %rdi     # use the result as the exit code
    movq $60, %rax
    syscall             # call system exit

m1:                    # pushing the frame for the m1 function
    pushq %rbp
    movq %rsp, %rbp
    subq $0, %rsp
    ...                 # the code of the m1 function
m1_end:                # popping the frame for the m1 function
    movq %rbp, %rsp
    popq %rbp
    ret

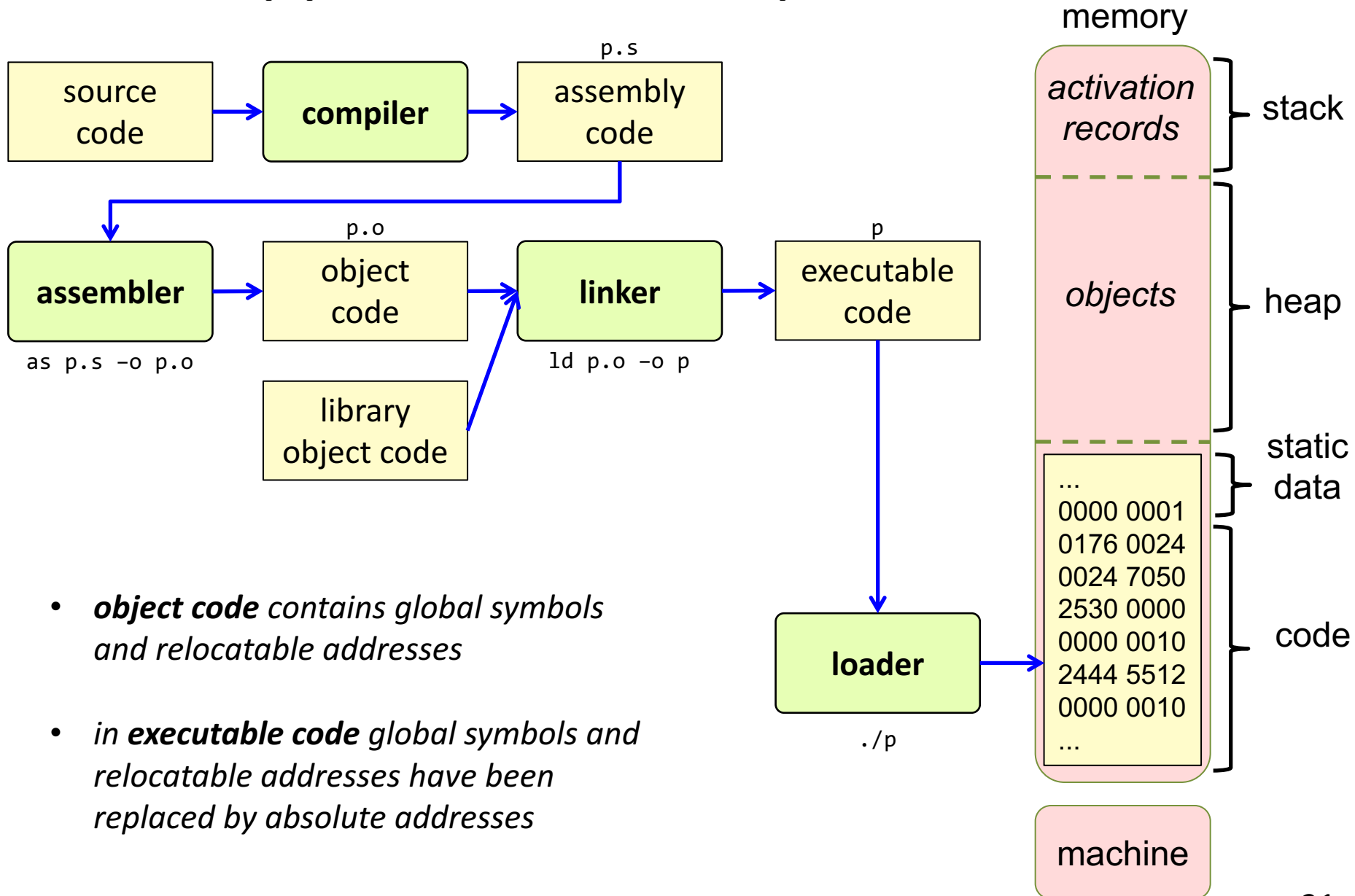
m2:
    ...
m2_end:
...
...
```

Program  
node

FunctionDecl  
node

FunctionDecl  
node

# What happens after compilation?



- **object code** contains global symbols and relocatable addresses
- in **executable code** global symbols and relocatable addresses have been replaced by absolute addresses

# Summary questions

- What information needs to be computed before generating code?
- How do explicit temporaries work? How do stacked temporaries work? What are the advantages and disadvantages of these implementation techniques?
- How can local variable numbers be computed using attributes?
- How can unique labels be computed?
- What is the difference between a text and a data segment in an assembly program?
- What needs to be done to run a program in assembly code?