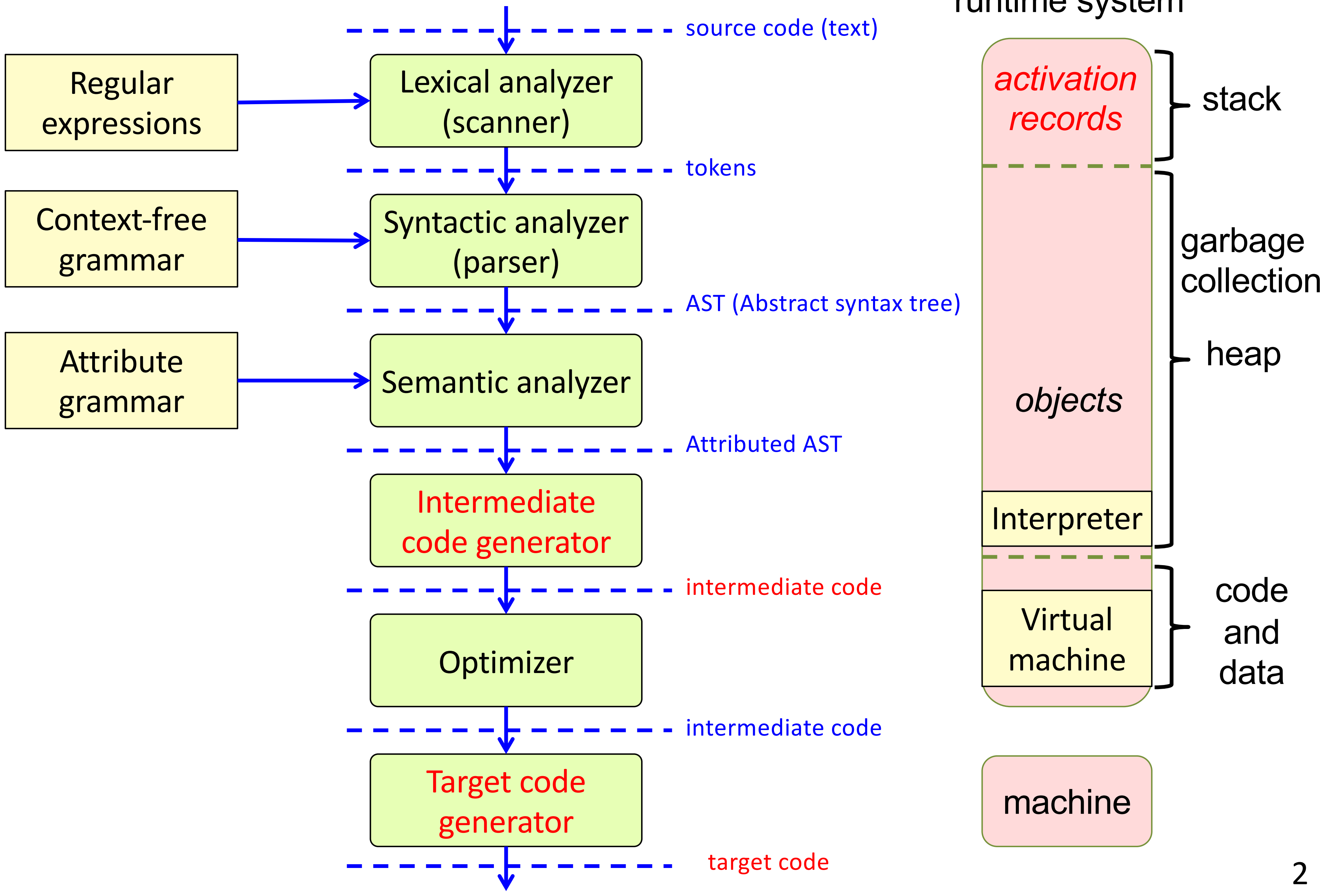
EDAN65: Compilers, Lecture 11

Code generation

Görel Hedin

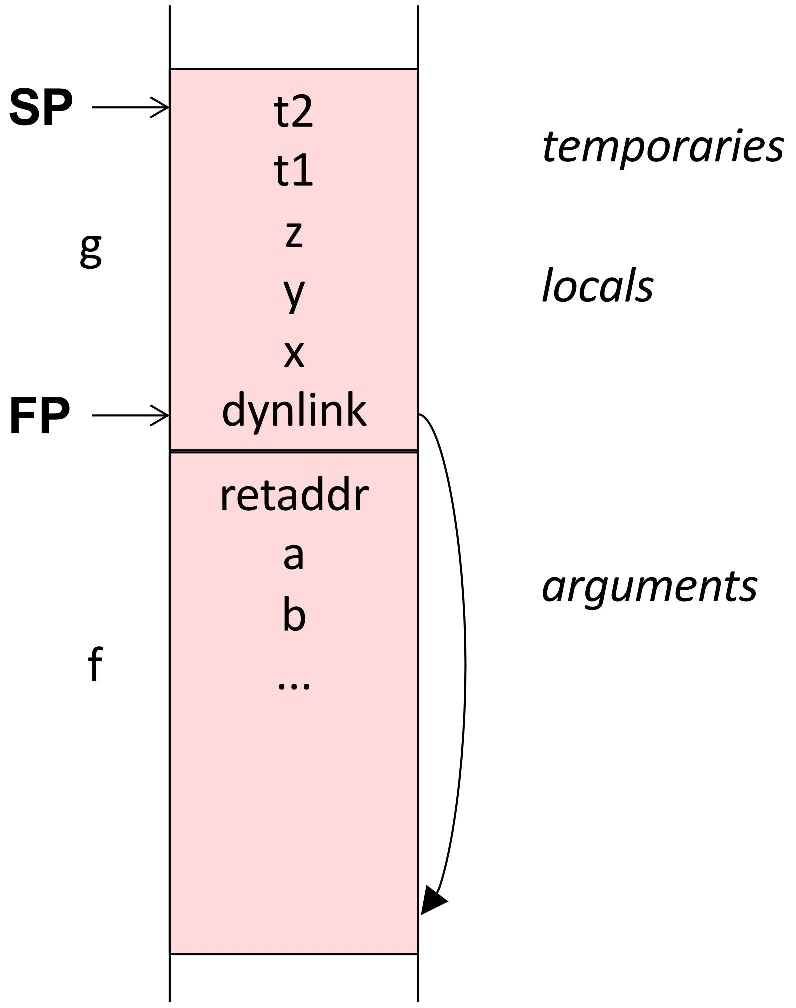
Revised: 2024-10-07

This lecture



Recall: example framelayout

```
void f() {
  ...
  g(1,2);
  ...
}

void g(int a, int b) {
  int x = 1;
  int y = 2;
  int z = 3;
  ...
  ...          <- PC
  ...
}
```

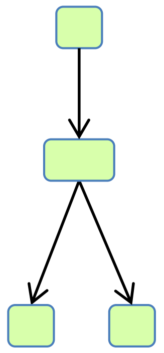


Generating code

Source code

```
if (v1+v2!=v3)
  v1 = v3-1;
v4 = v2;
...
```

AST



Intermediate code

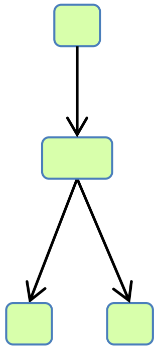
Machine code

Generating code

Source code

```
if (v1+v2!=v3)
  v1 = v3-1;
v4 = v2;
...
```

AST



Intermediate code

```
   ADD v1 v2 t1
   JEQ t1 v3 L1
   SUB v3  1 t2
   MOV t2 v1
L1:
   MOV v2 v4
   ...
```

Pseudo machine code

```
  MOV    1(FP) R1
  ADD    2(FP) R1
  MOV    R1 5(FP)
  CMP    5(FP) 3(FP)
  JEQ    L1
  MOV    3(FP) R1
  SUB    1 R1
  MOV    R1 6(FP)
  MOV    6(FP) 1(FP)
L1:
  MOV    2(FP) 4(FP)
```

Variable addresses

v1	1(FP)
v2	2(FP)
v3	3(FP)
v4	4(FP)
t1	5(FP)
t2	6(FP)

Intermediate code:

- Sequence of instructions with jumps and labels
- Each temporary result saved in new temporary variable
- Symbolic names for variables

Machine code (assembly code):

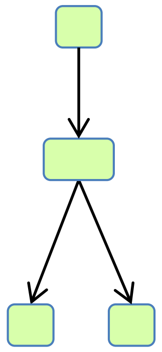
- Replace symbolic names by FP-relative addresses
- Move data between registers and memory,
  since instructions typically operate on registers
- ("Register allocation": Optimize temps by placing as many as possible in registers)

Two kinds of intermediate code

Source code

```
if (v1+v2!=v3)
  v1 = v3-1;
v4 = v2;
...
```

AST



Three-address code

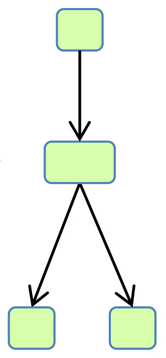
```
   ADD v1 v2 t1
   JEQ t1 v3 L1
   SUB v3  1 t2
   MOV t2 v1
L1:
   MOV v2 v4
   ...
```

Two kinds of intermediate code

Source code

```
if (v1+v2!=v3)
  v1 = v3-1;
v4 = v2;
...
```

AST



Three-address code

```
   ADD v1 v2 t1
   JEQ t1 v3 L1
   SUB v3  1 t2
   MOV t2 v1
L1:
   MOV v2 v4
   ...
```

Stack code

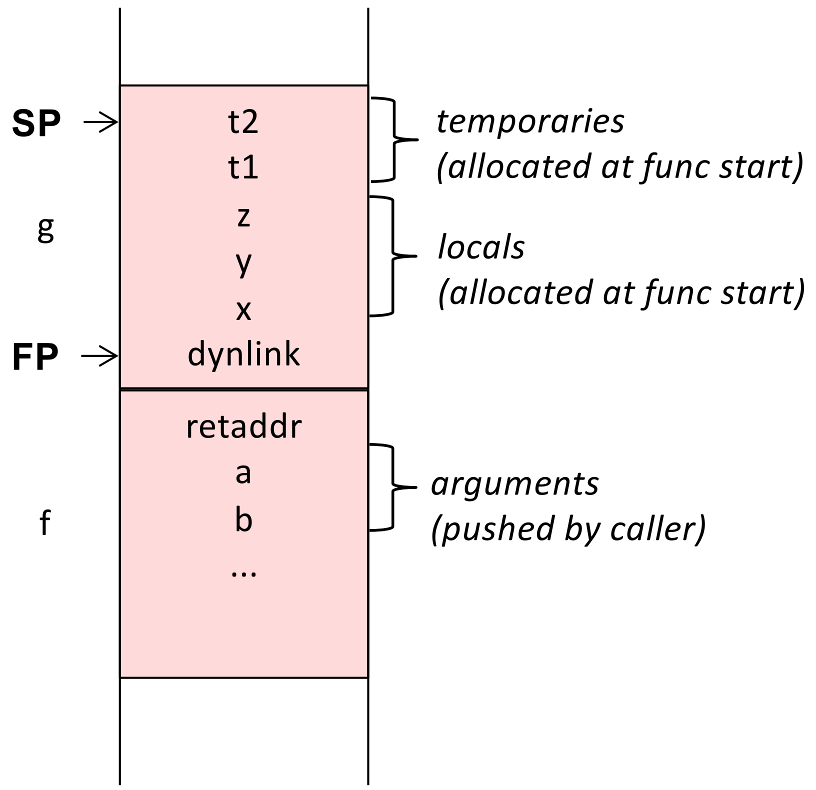
```
   PUSH v1
   PUSH v2
   ADD
   PUSH v3
   JEQ  L1
   PUSH v3
   PUSH  1
   SUB
   POP  v1
L1:
   PUSH v2
   POP  v4
   ...
```

Three-address code
Each instruction typically has three operands:
  op src1 src2 dest

Uses temporary variables.
Close to ordinary register-based machine.
Good for optimization.

Stack code
Uses a *value stack* instead of temporary variables.
op: pops operands, performs op, pushes result
Commonly used for interpreters and virtual machines.

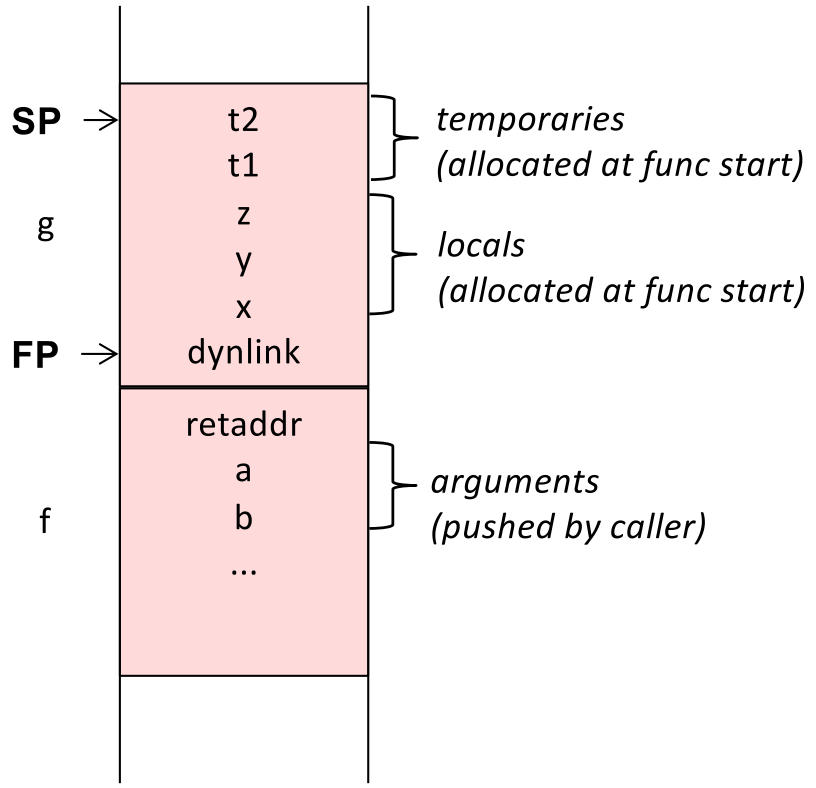
Three-address code

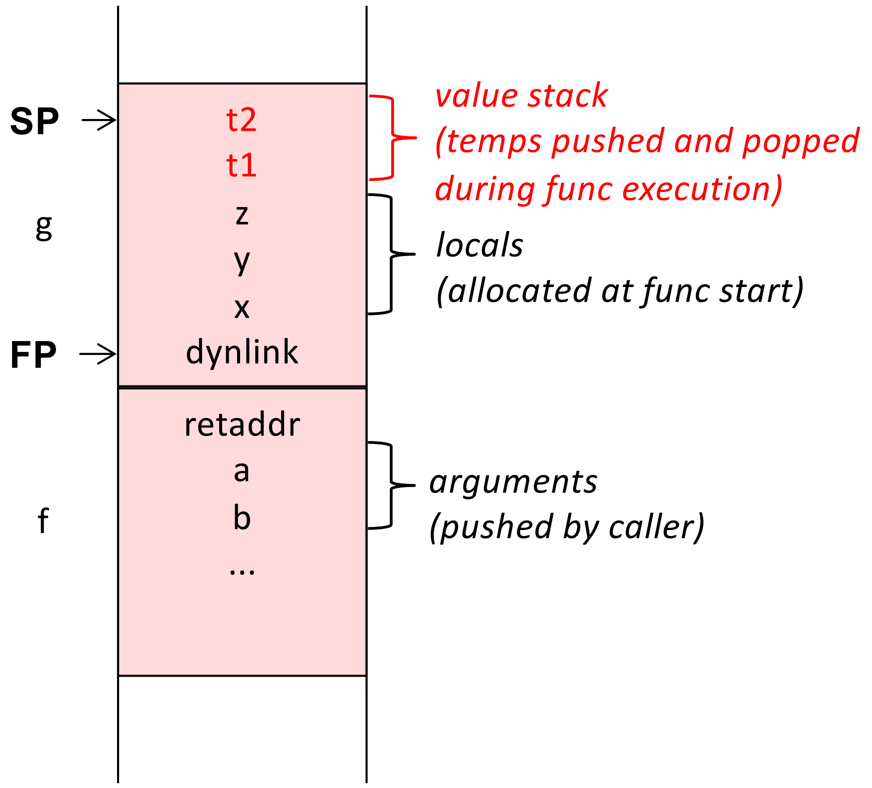


SP
FP
g
f
t2
t1
z
y
x
dynlink
retaddr
a
b
...
temporaries (allocated at func start)
locals (allocated at func start)
arguments (pushed by caller)

temporaries accessed by explicit addresses (via FP)

Three-address code *vs* ***Stack code***





temporaries accessed by explicit addresses (via FP)

temporaries accessed by push and pop instructions

Translate to three-address code

Source code

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

Translate to three-address code

Source code

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

Three-address code

```
ADD b c t1
ADD d e t2
MUL t1 t2 t3
MOV t3 a
```

One new temporary for each operation result.

Why not try to reuse the temporaries?
And remove useless MOVs?
In principle, two temps would suffice here:

```
ADD b c t1
ADD d e t2
MUL t1 t2 a
```

Minimizing the number of temporaries (not meaningful).

Typically, the intermediate code is optimized at a later stage. The optimizations transform the code and introduce new temporaries. Temporaries are optimized as a final step, as part of register allocation. Trying to minimize the number of temporaries at the code generation stage is therefore meaningless.

Translate three-address code to AT&T x86-64 assembly code

Source code

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

3-address code

```
  ...
  ADD a b t1
  MOV t1 c
  ...
```

Variable addresses

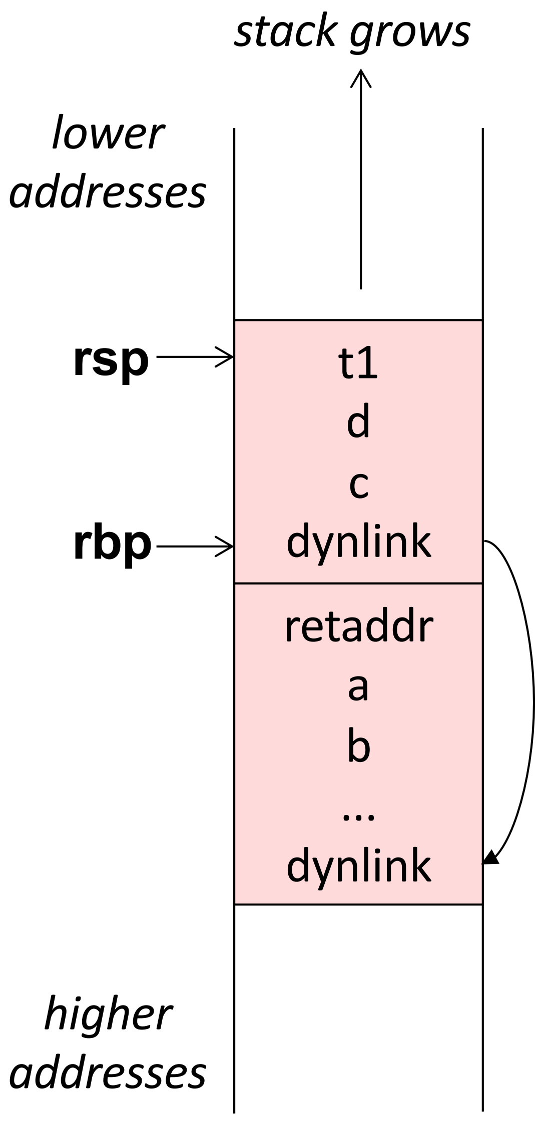
a	16(%rbp)
b	24(%rbp)
c	-8(%rbp)
d	-16(%rbp)
t1	-24(%rbp)

Registers and instructions

```
%rsp: stack pointer (points to top of stack)
%rbp: base pointer (frame pointer)
%rip: instruction pointer (program counter)
%rax, %rbx, %rcx, %rdx, ...: general registers
8(%r): the memory content at the address %r + 8
addq $3, %r  # %r + 3 -> %r (q: quad word 64 bits)
```

Assembly code

```
  ...
  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
  ...
```



Translate to assembly code

Source code

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

Three-address code

Translate to assembly code

Source code

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

Three-address code

```
ADD a b t1
ADD a c t2
MUL t1 t2 t3
MOV t3 d
```

Variable addresses

a	-8(%rbp)
b	-16(%rbp)
c	-24(%rbp)
d	-32(%rbp)
t1	-40(%rbp)
t2	-48(%rbp)
t3	-56(%rbp)

Assembly code (unoptimized):

```
movq  -8(%rbp), %rax        # a -> rax
addq  -16(%rbp), %rax       # b + rax -> rax
movq  %rax, -40(%rbp)       # rax -> t1
movq  -8(%rbp), %rax        # a -> rax
addq  -24(%rbp), %rax       # c + rax -> rax
movq  %rax, -48(%rbp)       # rax -> t2
movq  -40(%rbp), %rax       # t1 -> rax
imulq -48(%rbp), %rax       # t2 * rax -> rax
movq  %rax, -56(%rbp)       # rax -> t3
movq  -56(%rbp), -32(%rbp) # t3 -> d
```

Comparison to optimized code

Source code

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

Three-address code

```
   ADD a b t1
   ADD a c t2
   MUL t1 t2 t3
   MOV t3 d
```

Variable addresses

a	-8(%rbp)
b	-16(%rbp)
c	-24(%rbp)
d	-32(%rbp)
t1	-40(%rbp)
t2	-48(%rbp)
t3	-56(%rbp)

Unoptimized assembly code: 11 memory accesses, 7 vars

```
  movq  -8(%rbp), %rax        # a -> rax
  addq  -16(%rbp), %rax       # b + rax -> rax
  movq  %rax, -40(%rbp)       # rax -> t1
  movq  -8(%rbp), %rax        # a -> rax
  addq  -24(%rbp), %rax       # c + rax -> rax
  movq  %rax, -48(%rbp)       # rax -> t2
  movq  -40(%rbp), %rax       # t1 -> rax
  imulq -48(%rbp), %rax       # t2 * rax -> rax
  movq  %rax, -56(%rbp)       # rax -> t3
  movq  -56(%rbp), -32(%rbp) # t3 -> d
```

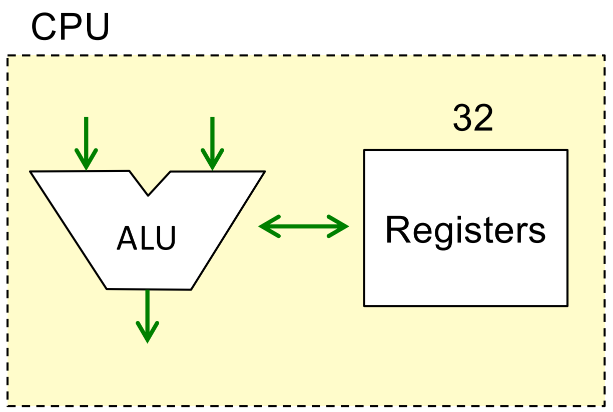
- Keep temps in registers
- Eliminate unnecessary mov instructions

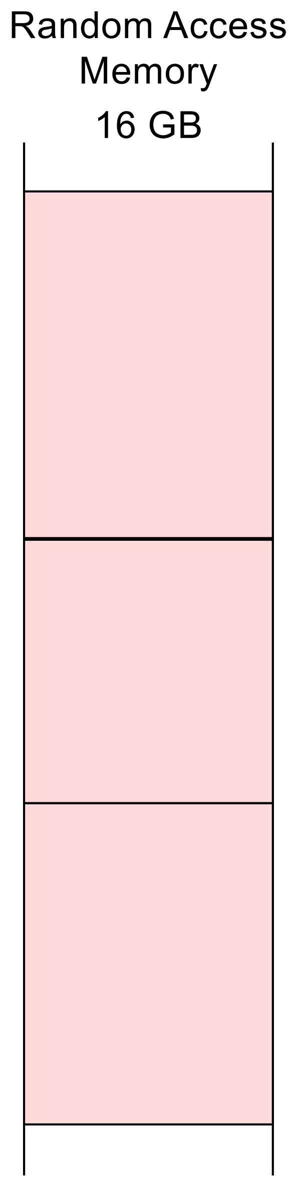
(See course on optimizing compilers)

Optimized assembly code: 4 memory accesses, 4 vars

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

Typical sizes and access times

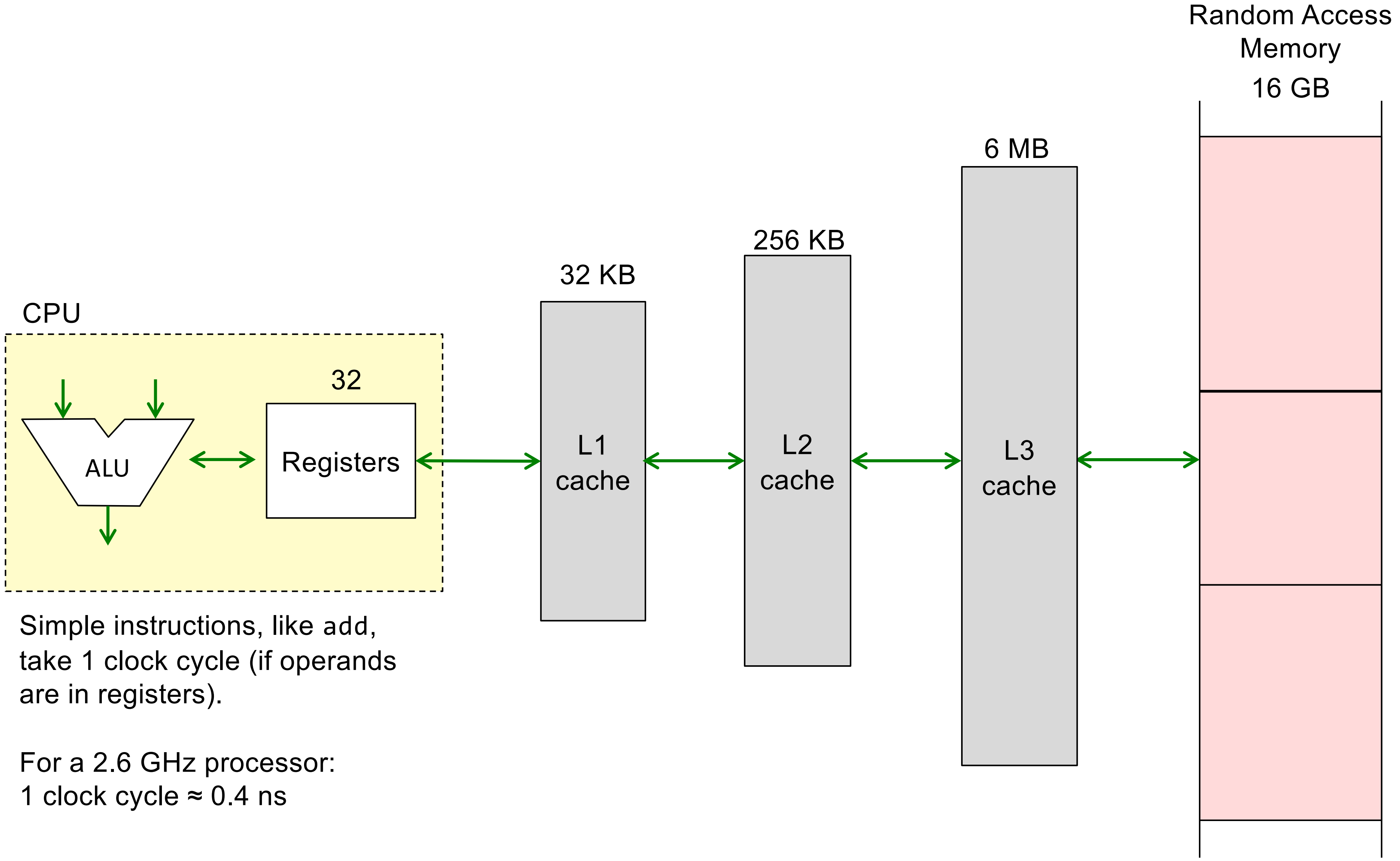




Simple instructions, like `add`, take 1 clock cycle (if operands are in registers).

For a 2.6 GHz processor:
1 clock cycle ≈ 0.4 ns

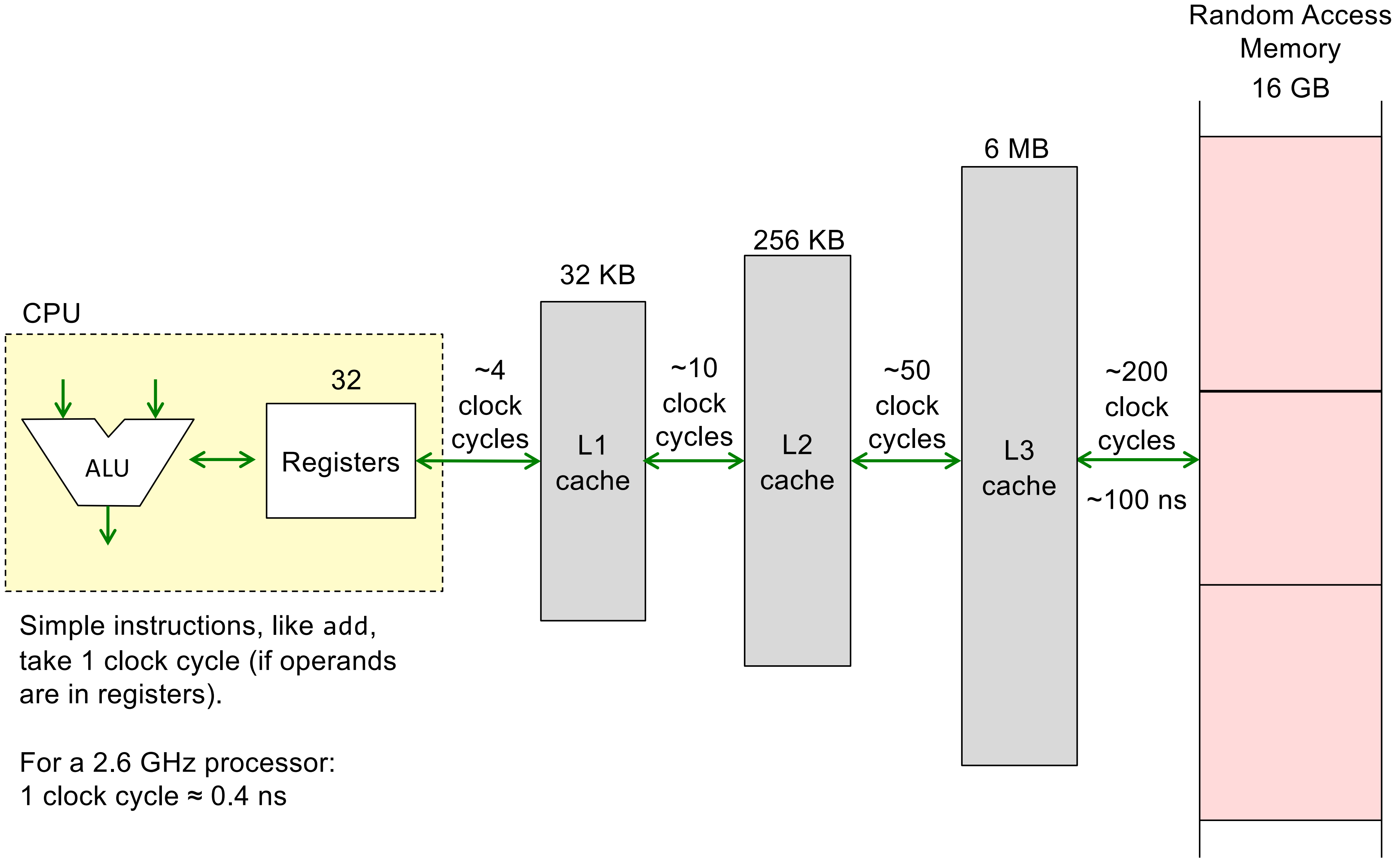
Typical sizes and access times



Simple instructions, like `add`, take 1 clock cycle (if operands are in registers).

For a 2.6 GHz processor:
1 clock cycle ≈ 0.4 ns

Typical sizes and access times



Simple instructions, like `add`, take 1 clock cycle (if operands are in registers).

For a 2.6 GHz processor:
1 clock cycle ≈ 0.4 ns

Register allocation

Register allocation

Keep as many variables and temporaries as possible in registers, "spilling" as few of them as possible to memory.

Done after intermediate-code optimization, when translating to actual machine code (when we know how many registers there are).

Good algorithms exist, based on graph coloring.
See course on Optimizing Compilers.

In assignment 6, we will use naive code generation (no optimization).

Control structures

Source code

```
void m() {
  int x, s;
  ...
  while (x > 1) {
    s = s + x;
  }
  ...
}
```

3-address code

```
m:
   ...
m_1:
   JLE x 1 m_2    # if x <= 1 jump to m_2
   ADD s x t1     # s + x -> t1
   MOV t1 s       # t1 -> s
   JMP m_1        # jump to label m_1
m_2:
   ...
```

Note:
Flip the condition to get simpler code
All labels must be unique in the program

Control structures

Source code

```
void m() {
  int x, s;
  ...
  while (x > 1) {
    s = s + x;
  }
  ...
}
```

Variable addresses

```
x      -8(%rbp)
s     -16(%rbp)
t1    -24(%rbp)
```

New instructions used

```
cmpq a, b: compares a and b, sets condition codes
jle lbl:   jumps to label lbl if le condition code is set
jmp lbl:   jumps to label lbl
```

3-address code

```
m:
   ...
m_1:
   JLE x 1 m_2

   ADD s x t1


   MOV t1 s
   JMP m_1
m_2:
   ...
```

x86 assembly code

```
m:
   ...
m_1:
   cmpq -8(%rbp), $1     # Compare x and 1
   jle m_2               # Jump if previous cmp was less-or-equal
   movq -16(%rbp), %rax  # s -> rax
   addq -8(%rbp), %rax   # x + rax -> rax
   movq %rax, -24(%rbp)  # rax -> t1
   movq -24(%rbp), -16(%rbp)  # t1 -> s
   jmp m_1
m_2:
   ...
```

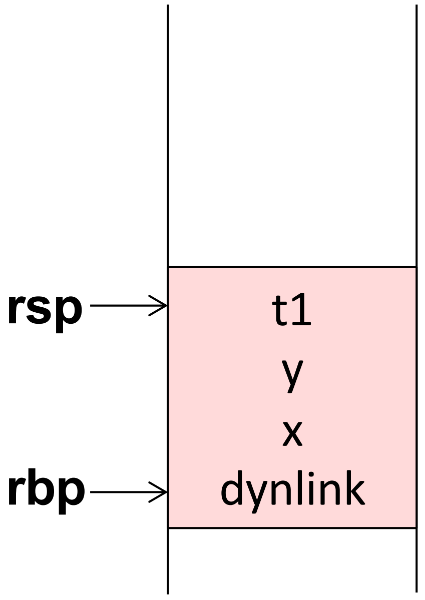
Method call

Source code

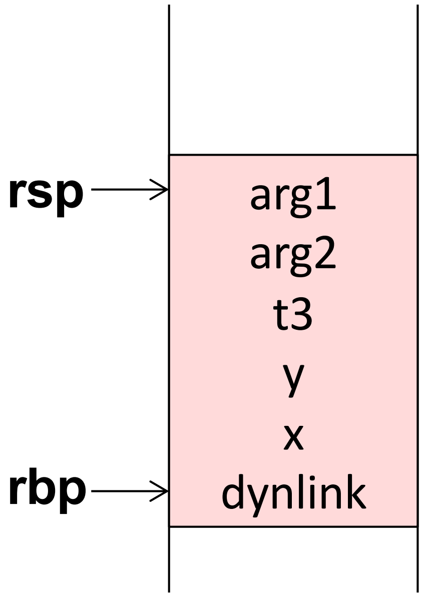
```
int x, y;
...
y = p(x+1, 2);
...
```

3 -address code

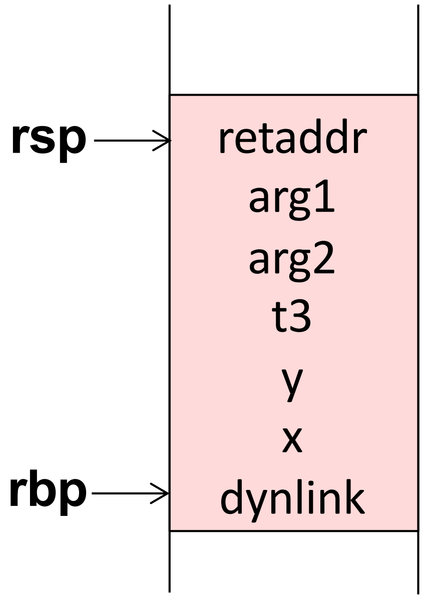
```
...
ADD x 1 t1 # Eval arg 1
PUSH 2     # Pass arg 2
PUSH t1    # Pass arg 1
CALL p     # Do the call
POP        # Pop arg 1
POP        # Pop arg 2
MOV rv y   # Assign return value to y
...
```



Original situation



Passing the args



Calling p

Method call

Calling conventions:
Args pushed in reverse order on stack
Return value stored in RAX register

Source code

```
   int x, y;
   ...
   y = p(x+1, 2);
   ...
```

New instructions used

```
pushq v: pushes a value to the stack (moves rsp)
call m: pushes the return address and jumps to m
```

3-address code

```
  ...
  ADD x 1 t1


  PUSH 2
  PUSH t1
  CALL p
  POP
  POP
  MOV rv y
  ...
```

x86 Assembly code

```
  ...
  movq -8(%rbp), %rax         # x -> rax
  addq $1, %rax               # 1 + rax -> rax
  movq %rax, -24(%rbp)        # rax -> t1
  pushq $2                    # push arg 2
  pushq -24(%rbp)             # push arg 1
  call p                      # call p
  addq $16, %rsp              # pop 2 arguments

  movq %rax, -16(%rbp)        # return value -> y
  ...
```

Variable allocation

```
x    -8(%rbp)
y    -16(%rbp)
t1   -24(%rbp)
```

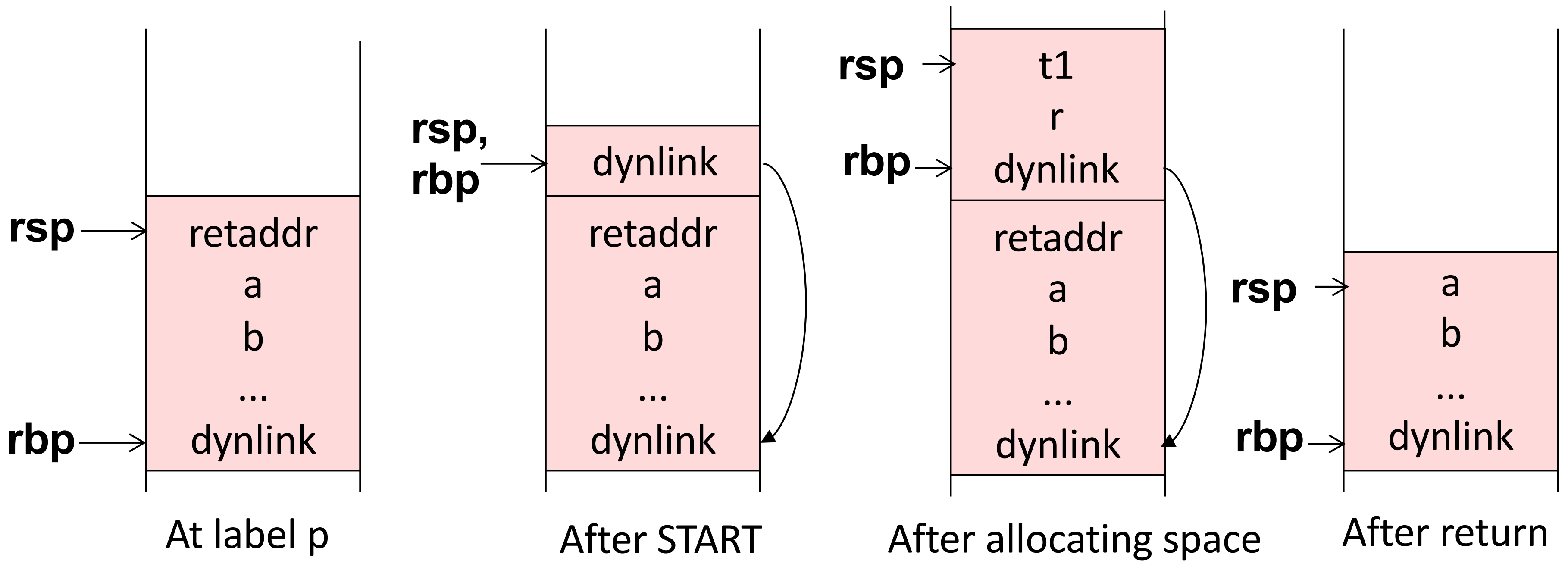
Method activation and return

Source code

```
int p(int a, int b) {
  int r;
  ...
  return r+1
}
```

3-address code

```
p:
  START      # Start of activation
  SPACE 2    # Make space for 2 vars and temps
  ...
  ADD r 1 t1 # Compute the value to return
  MOV t1 rv  # Store the return value
  RETURN     # Return to the caller
```



Method activation and return

Source code

```
int p(int a, int b) {
  int r;
  ...
  return r+1
}
```

New instructions used

popq r: pops top of stack, and stores it to reg r
ret: pops the return address and jumps to it

3-address code

```
p:
  START

  SPACE 2
  ...
  ADD r 1 t1


  MOV t1 rv
  RETURN
```

Assembly code

```
p:                          # Label for p
  pushq %rbp                # Push the dynamic link
  movq %rsp, %rbp           # Set the new frame pointer
  subq $16 %rsp             # Make space for 2 vars and temps
  ...
  movq -8(%rbp), %rax       # r -> rax
  addq $1, %rax             # 1 + rax -> rax
  movq %rax, -16(%rbp)      # rax -> t1
  movq -16(%rbp), %rax      # t1 -> rax
  movq %rbp, %rsp           # move back the stack pointer
  popq %rbp                 # restore the frame pointer
  ret
```

Variable addresses

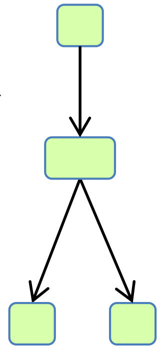
a	16(%rbp)
b	24(%rbp)
r	-8(%rbp)
t1	-16(%rbp)

Generating code

Source code

```
c = a + b
```

AST



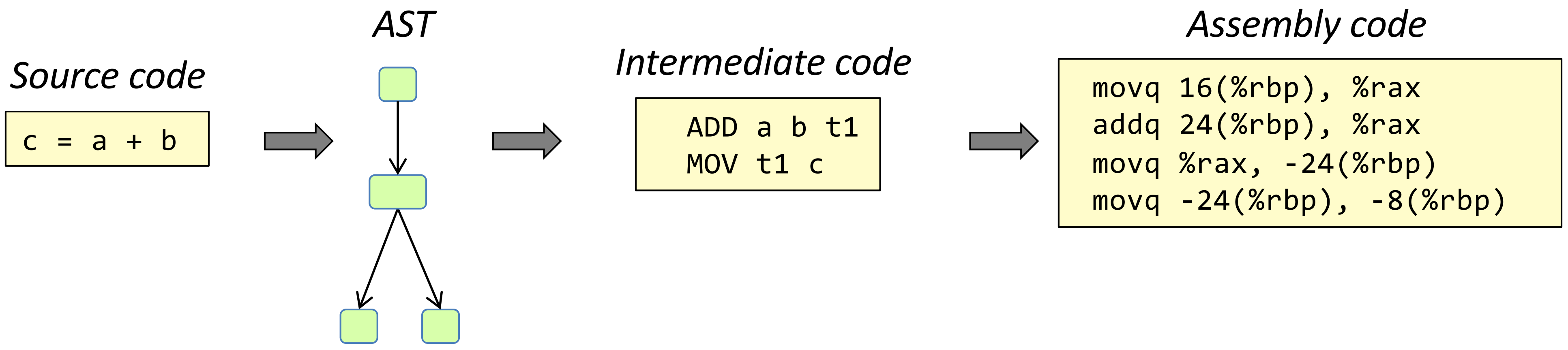
Intermediate code

```
ADD a b t1
MOV t1 c
```

Assembly code

```
movq 16(%rbp), %rax
addq 24(%rbp), %rax
movq %rax, -24(%rbp)
movq -24(%rbp), -8(%rbp)
```

Generating code



Source code

```
c = a + b
```

AST

Intermediate code

```
ADD a b t1
MOV t1 c
```

Assembly code

```
movq 16(%rbp), %rax
addq 24(%rbp), %rax
movq %rax, -24(%rbp)
movq -24(%rbp), -8(%rbp)
```

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.

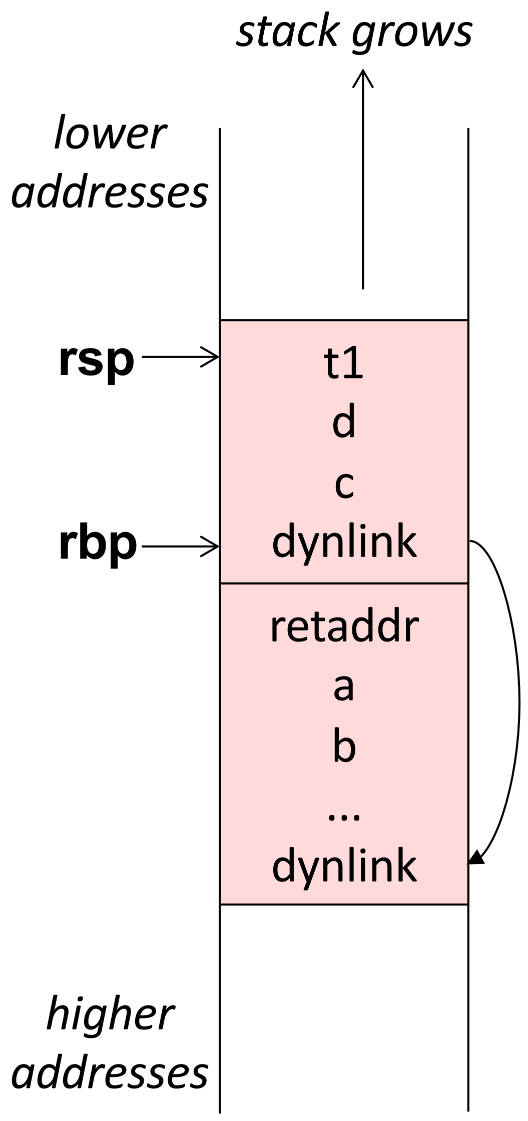
AT&T x86-64 assembly code

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 in each node:
  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);
   }
}
```

Generating code using attribute grammars

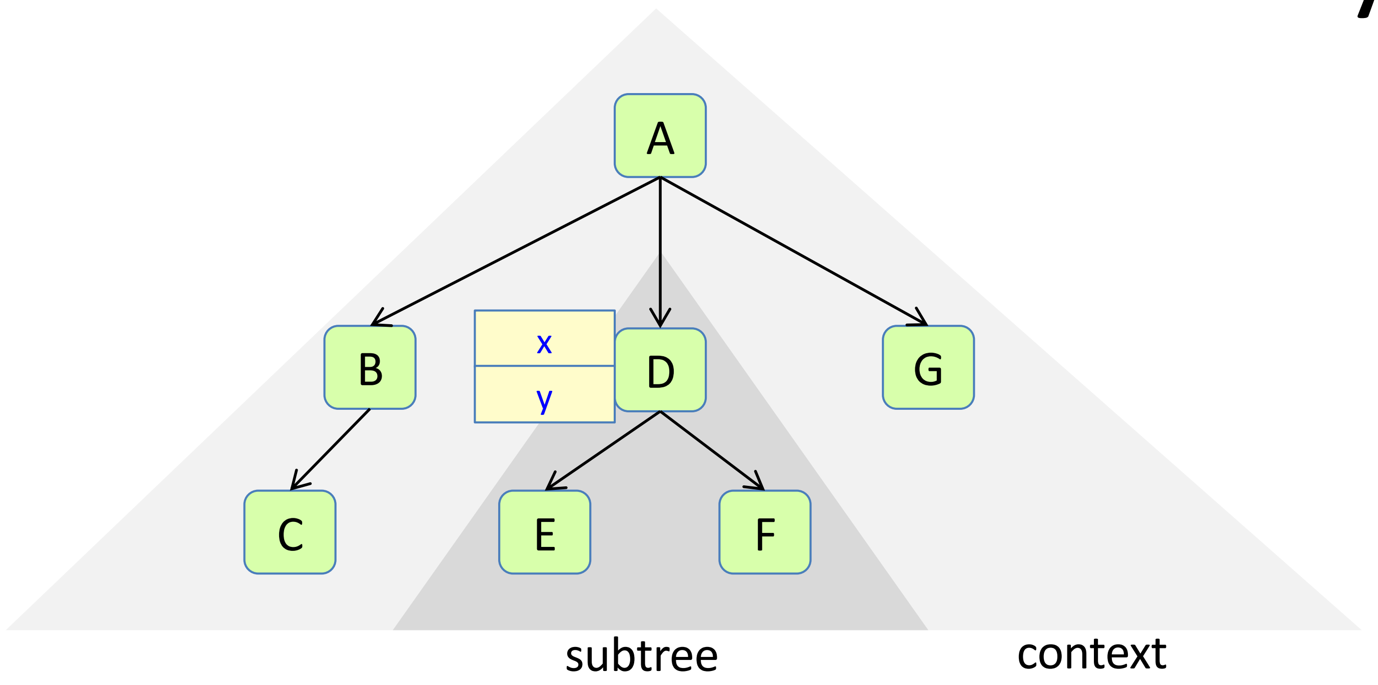
Main idea:

- Recursive traversal of the AST using a method in each node:
  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);
   }
}
```

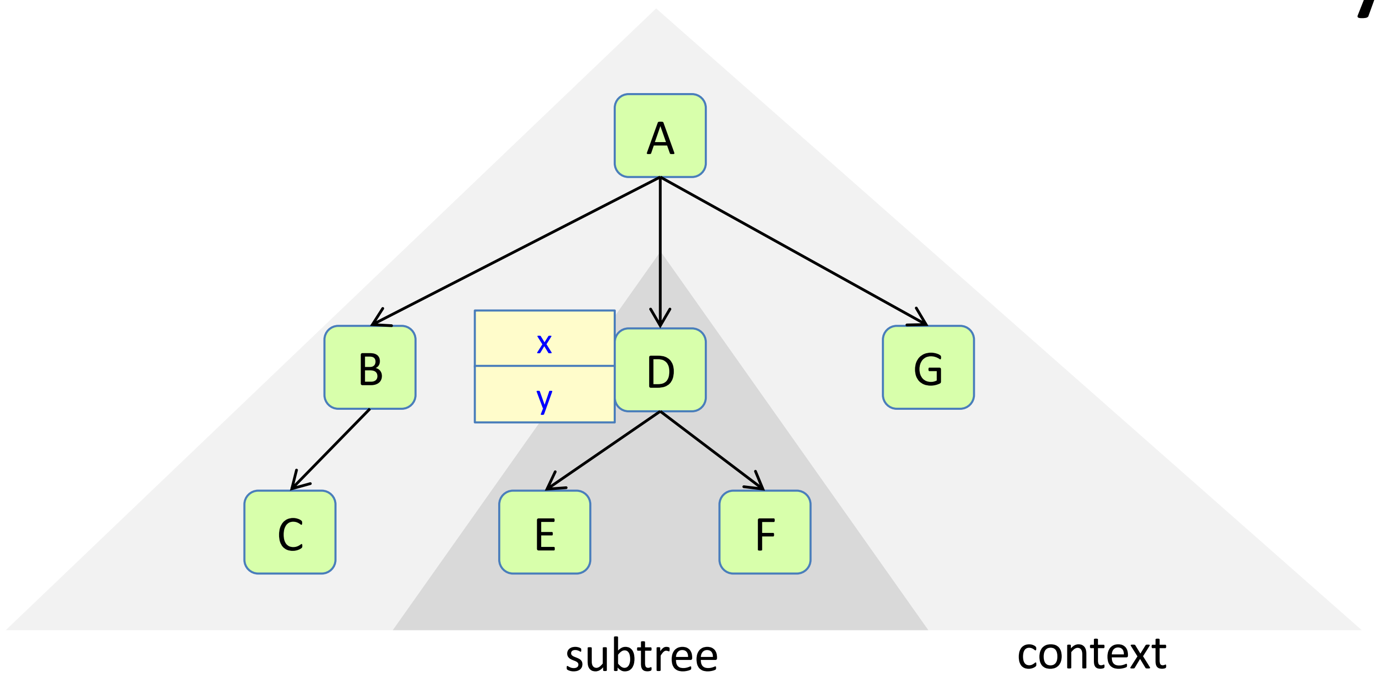
```
// Example genCode method
void Assignment.genCode(PrintStream out) {
   getRight().genCode(out);
   out.println("movq " + getRight().address() + ", " + getLeft().address());
}
```

Attributes: think declaratively!



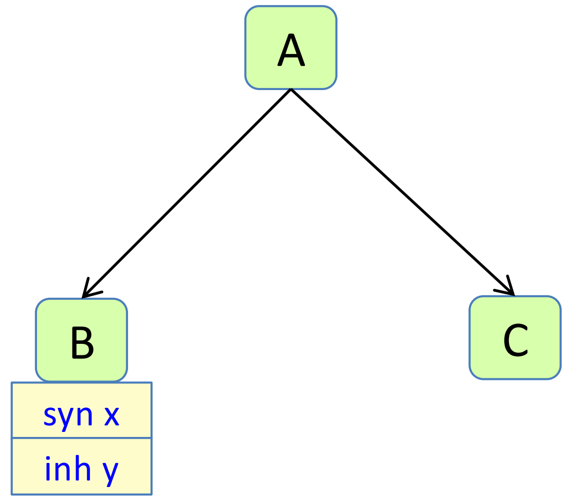
- Need property of node? Define an attribute for it!

Attributes: think declaratively!



- Need property of node? Define an attribute for it!
- Define the attribute using other attributes.
- Use synthesize attribute when the node can define it.
- Use inherited attributes when a node in the context needs to define it.

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() = ...;
```

Handling temps

Two approaches:

- Explicit temps (access via FP, like locals)
- Stacked temps (access via push/pop)

Explicit temps

(like in previous examples)

Source code

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

Main idea:

- Each nontrivial operation puts its result in a new temp

Code generation for binary operation, assignment, IdUse?

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
```

Explicit temps

(like in previous examples)

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:

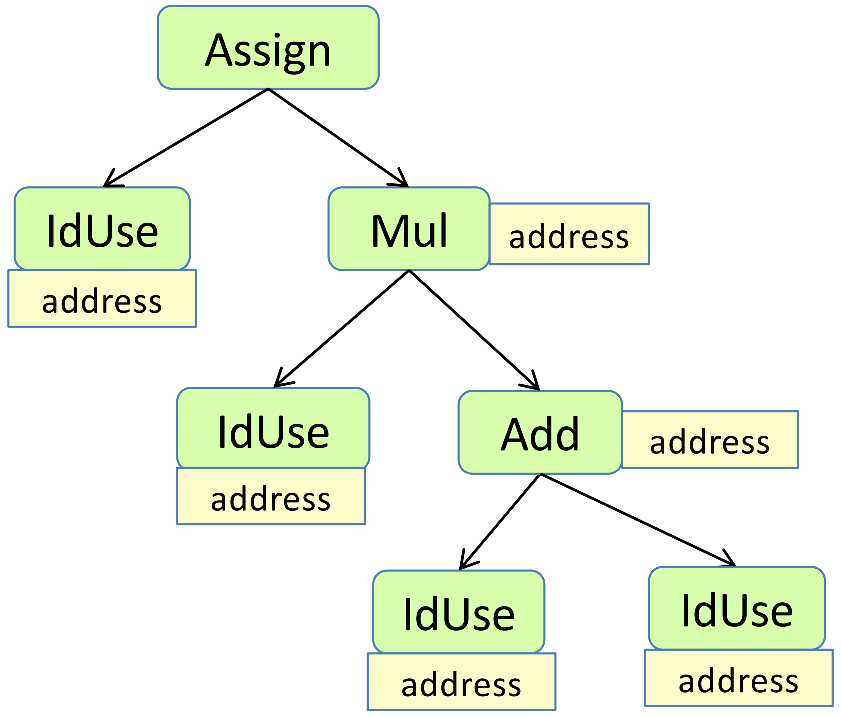
- generate code for right-hand side (result at some address)
- move result to left var

Code generation for IdUse:

- No code needed.

Example code generation with explicit temps

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

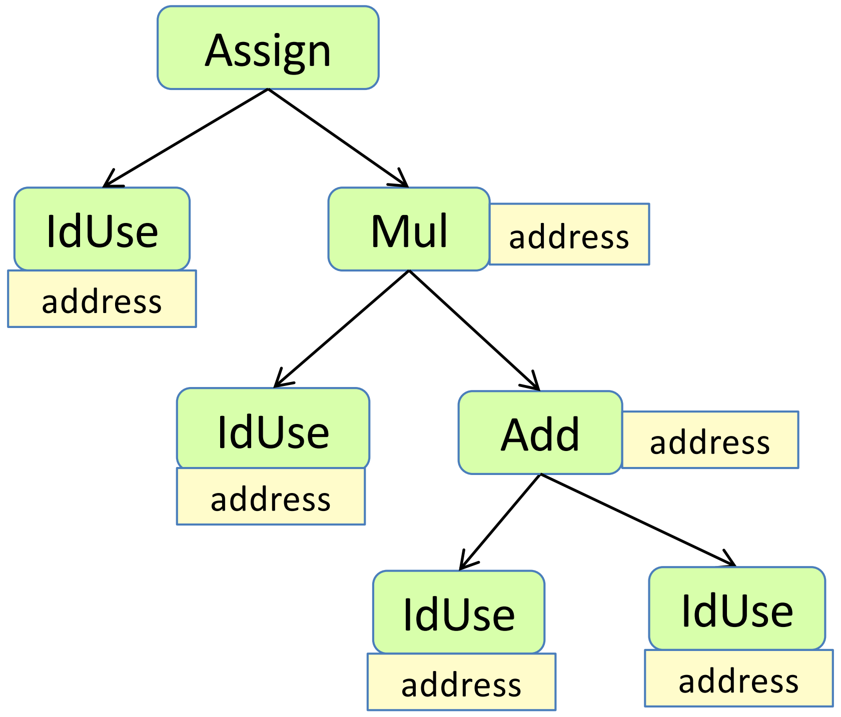


```
public void Mul.genCode(PrintStream s) {
```

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

Example code generation with explicit temps

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

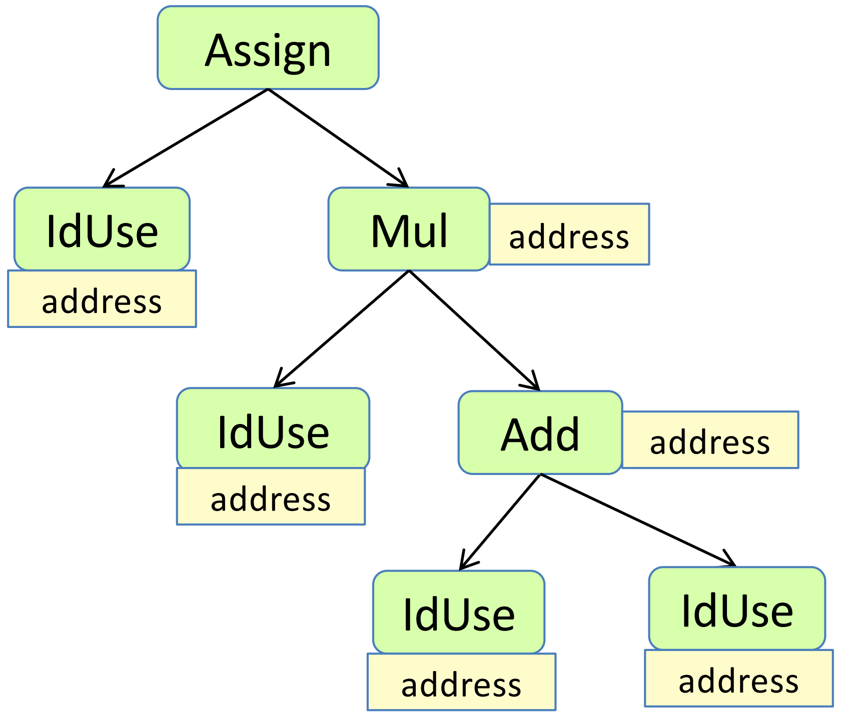


```
public void Mul.genCode(PrintStream s) {
   getLeft(). genCode(s);
   getRight(). genCode(s);
   s.println("movq " + getLeft().address() + ", %rax");
   s.println("imulq " + getRight().address() + ", %rax");
   s.println("movq %rax, " + address());
}
```

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

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) { }
```

Value stack

an alternative to explicit temps

Main idea: each expression puts its result in rax

Source code

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

Variable addresses

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

x86 assembly code, temps on value 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
```

Value stack

an alternative to explicit temps

Main idea: each expression puts its result in rax

Code generation for binary operation, assignment, IdUse?

Source code

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

Variable addresses

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

x86 assembly code, temps on value 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
```

Value stack

an alternative to explicit temps

Main idea: each expression puts its result in rax

Source code

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

Variable addresses

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

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 value 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:

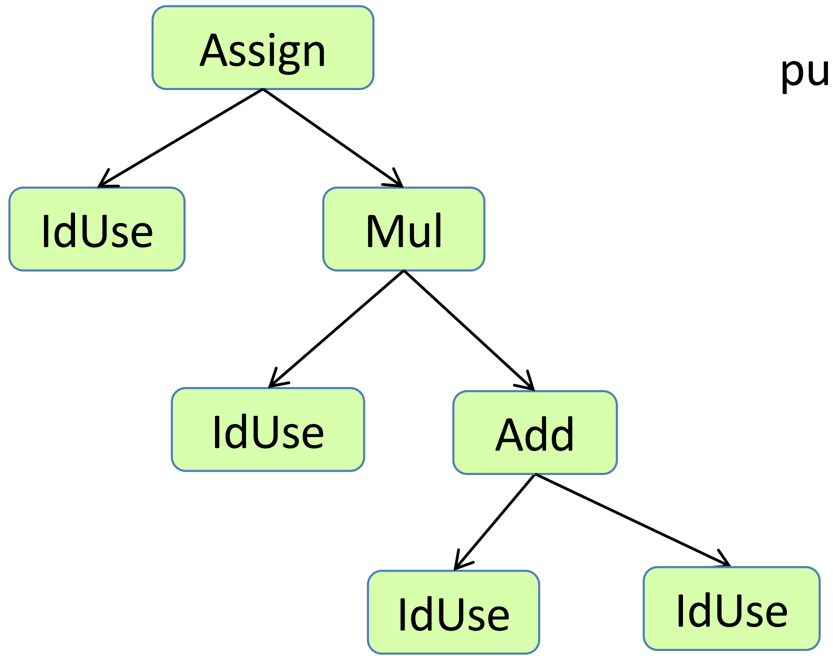
- generate code for right-hand side (result in rax)
- move rax to left var

Code generation for IdUse:

- move value into rax

Example code generation with value stack

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

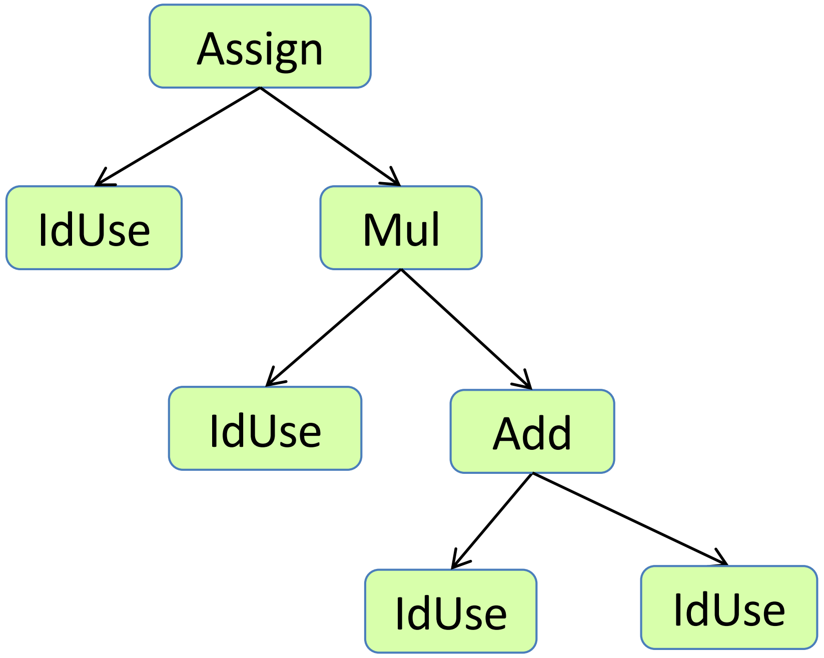


public void Mul.genCode(PrintStream s) {

public void IdUse.genCode(PrintStream s) {

Example code generation with value stack

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

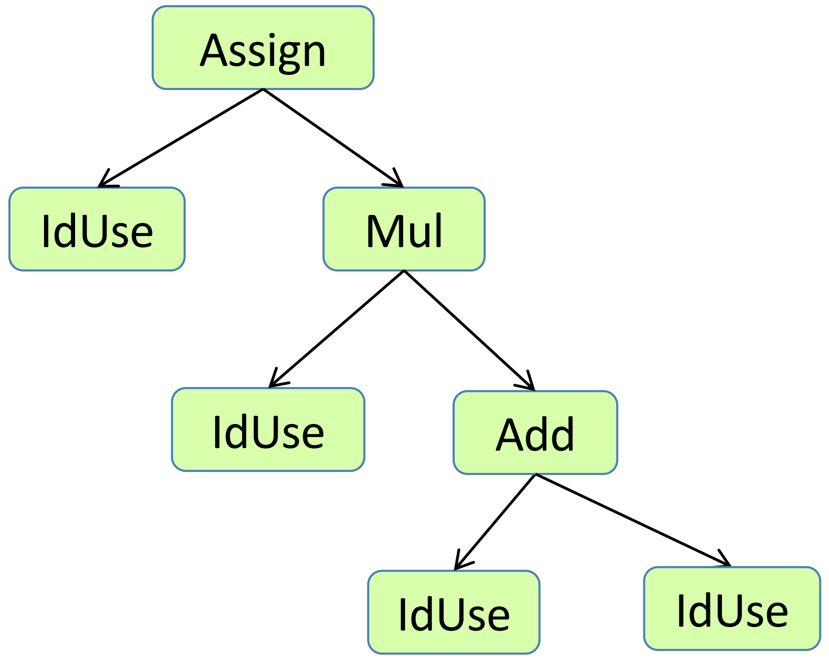


```
public void Mul.genCode(PrintStream s) {
   getLeft().genCode(s);
   s.println("pushq %rax");
   getRight().genCode(s);
   s.println("popq %rbx");
   s.println("imulq %rbx, %rax");
}
```

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

Example code generation with value stack

```
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 temps or value stack?

Explicit temps or value stack?

Code generation is simpler for the value stack approach – we don't need to compute addresses for temps.

But: to generate code for method calls, most languages require that we evaluate arguments from left to right (since they can have side effects). Pushing them from right to left is then slightly intricate to do when they are on the stack.

In assignment 6, we will use the value stack approach. You may evaluate the arguments from right to left to simplify the code generation.

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?

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?

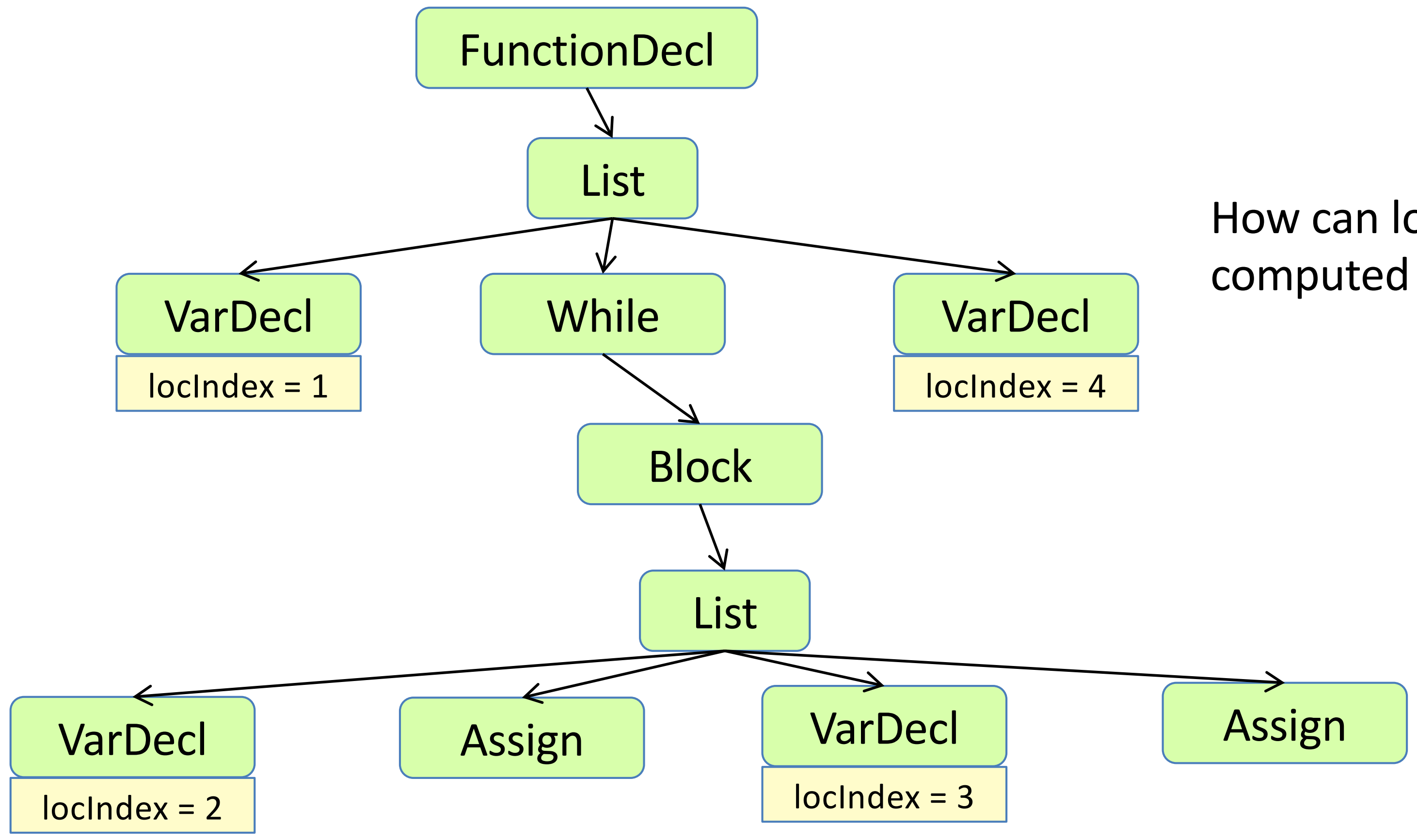
- The address of each variable declaration, for example "-8(%rbp)".
- The number of local variables of a method (to reserve space on the stack).
- The address of each formal argument, for example, "16(%rbp)".
- Unique labels for each control structure.

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)", ...



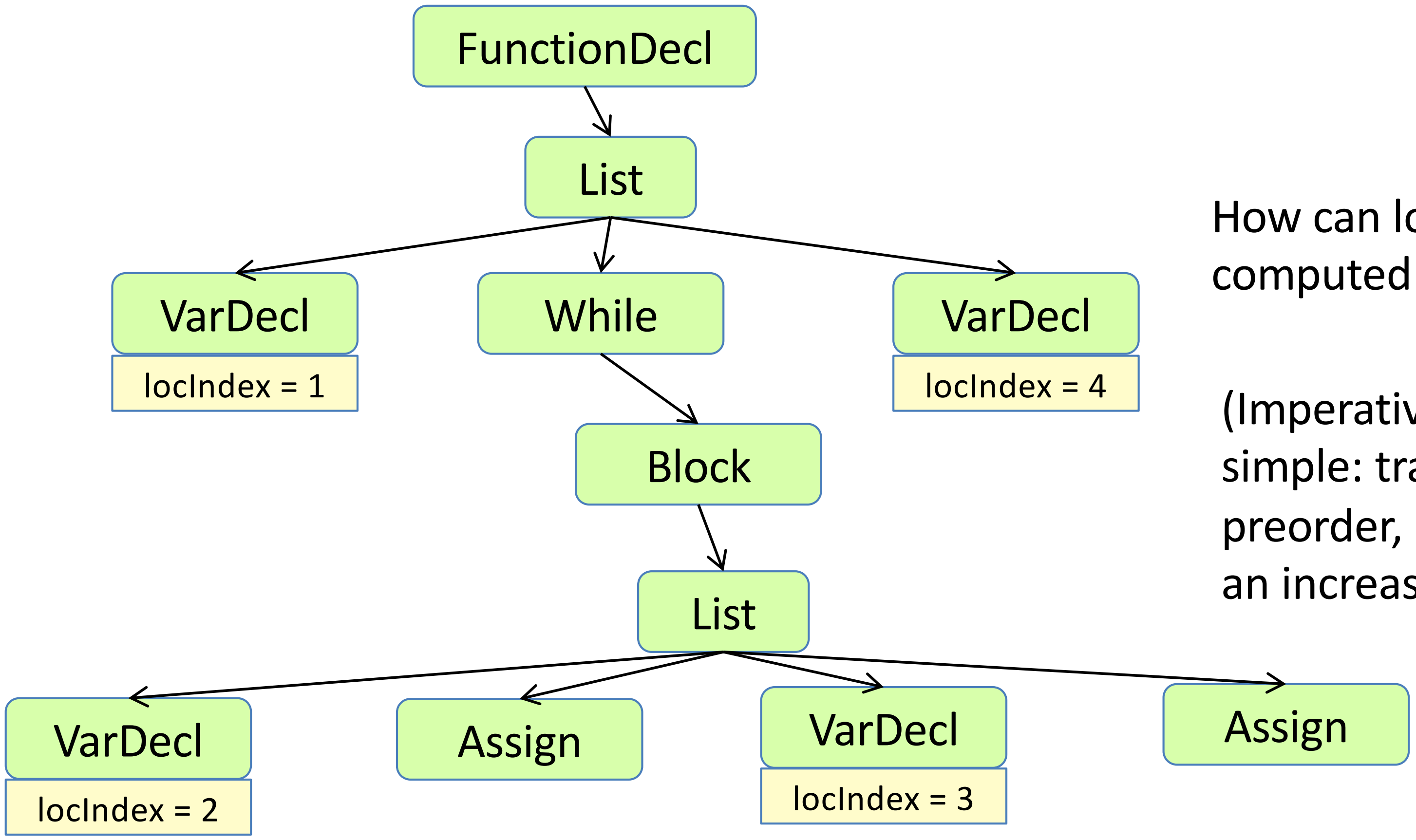
How can locIndex be computed using attributes?

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)", ...

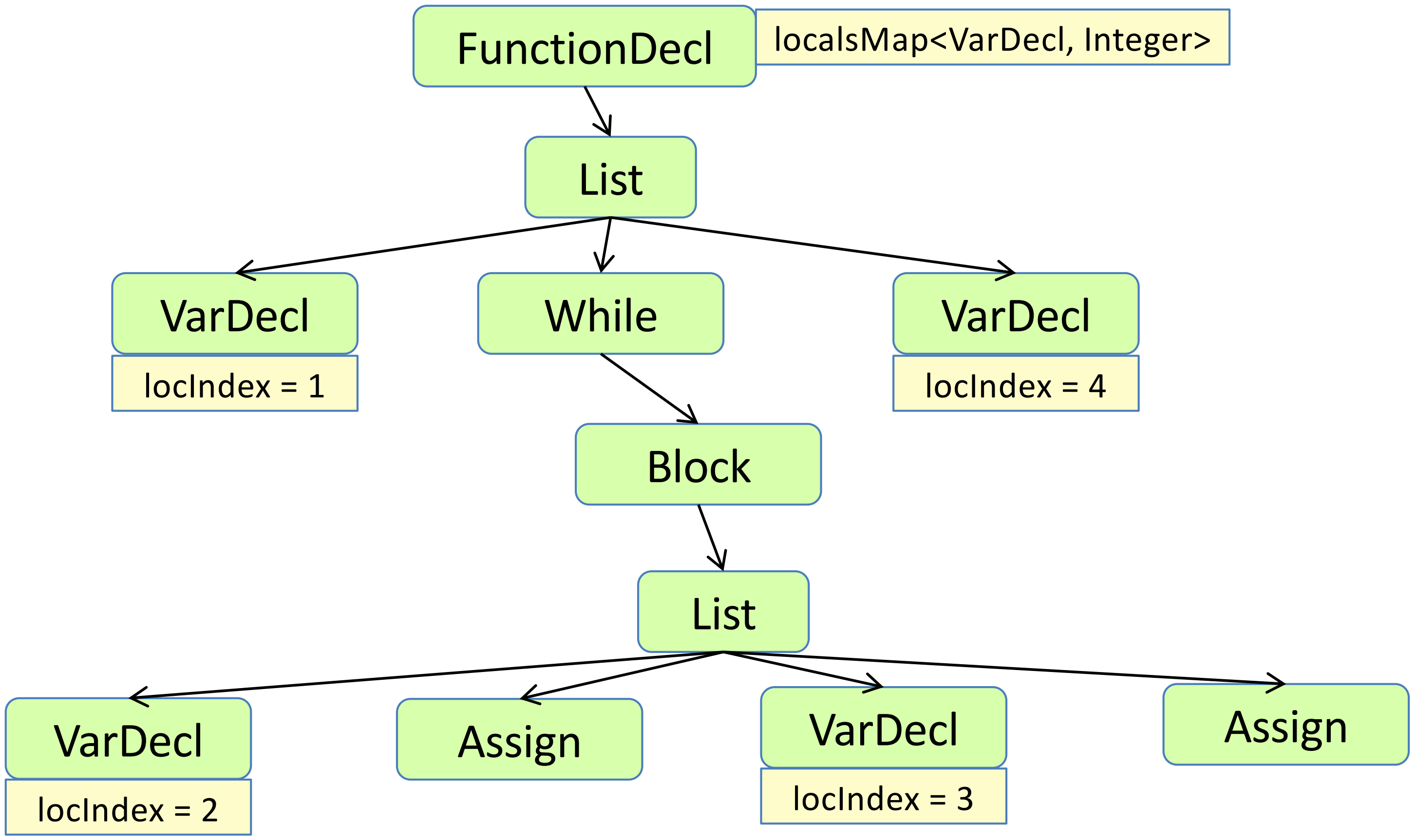


How can locIndex be computed using attributes?

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

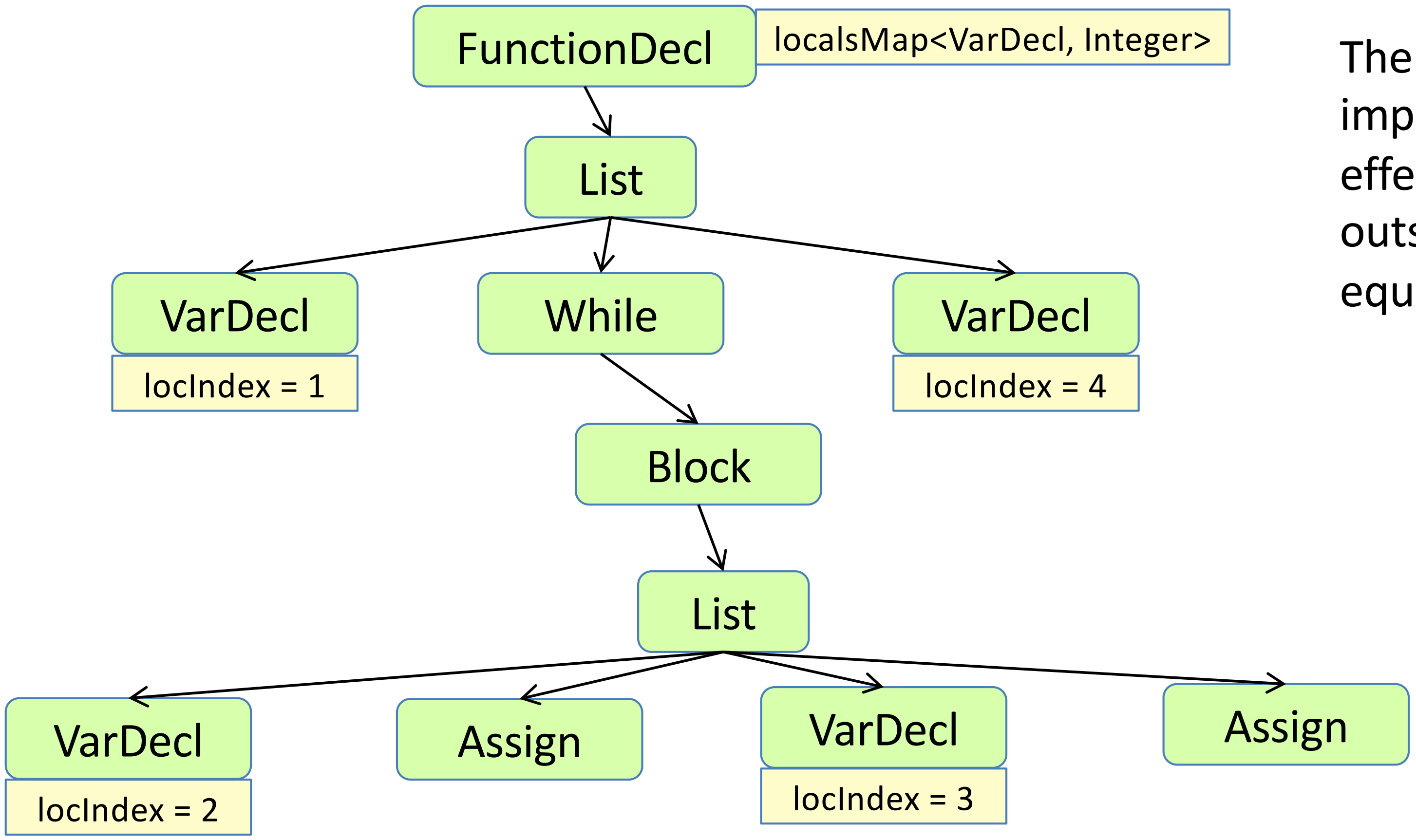
Solution using a map attribute

- 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.



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.

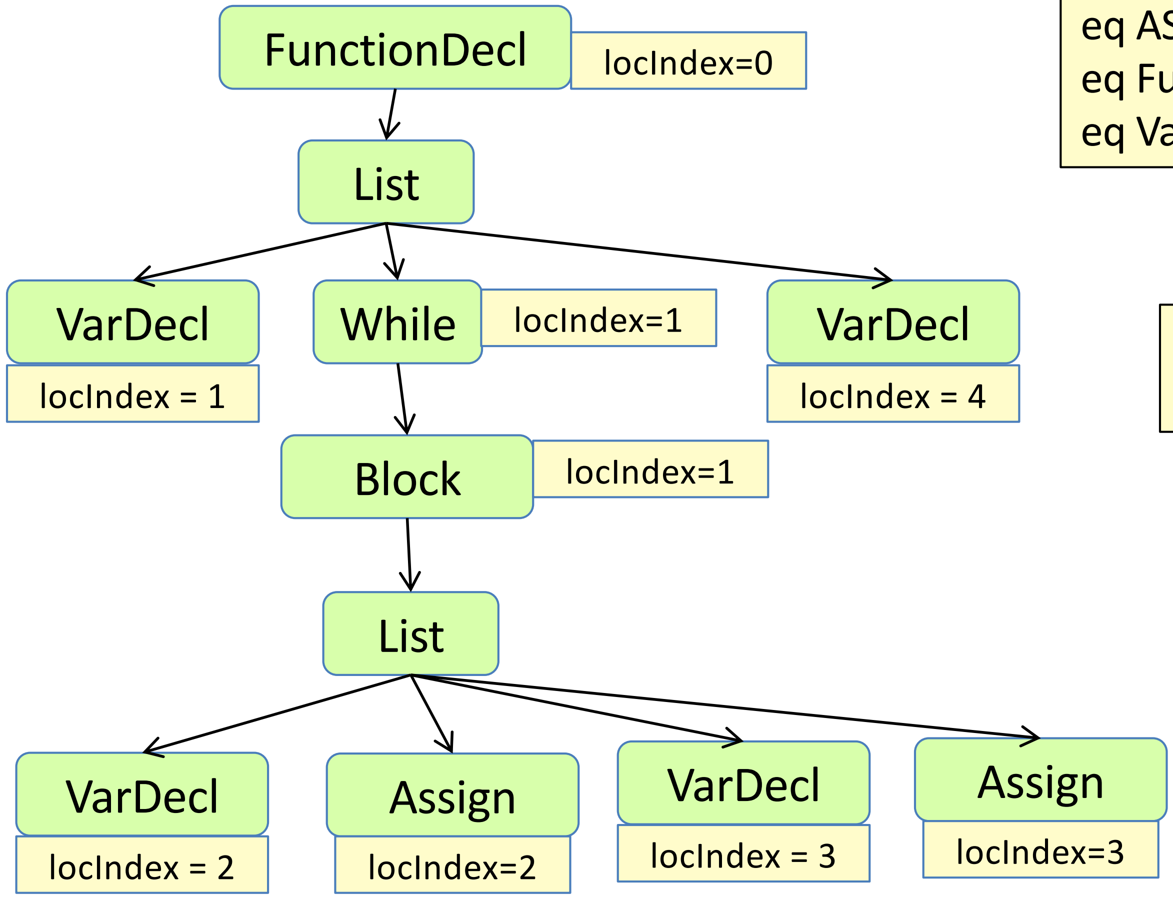


The **addLocals** method is imperative, but the side effects do not escape outside the **localsMap** equation, so that is ok.

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();
```

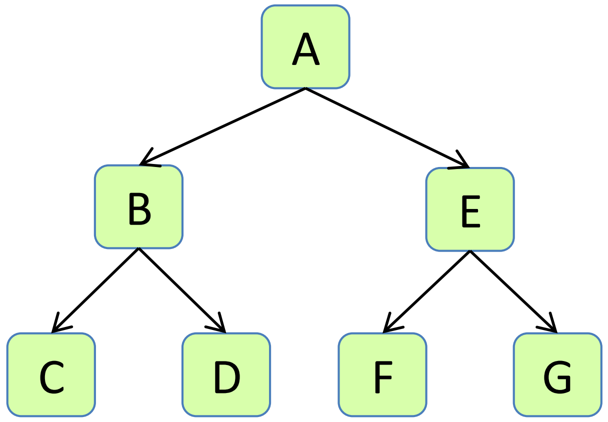
But how are prev() and last() defined?

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.prev() == A.prev(1) == B.last() == B.prev(2) == D.last() == D.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

```
.global _start              # the _start label is globally visible
.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:

...
```

An example assembly program

Generated by:

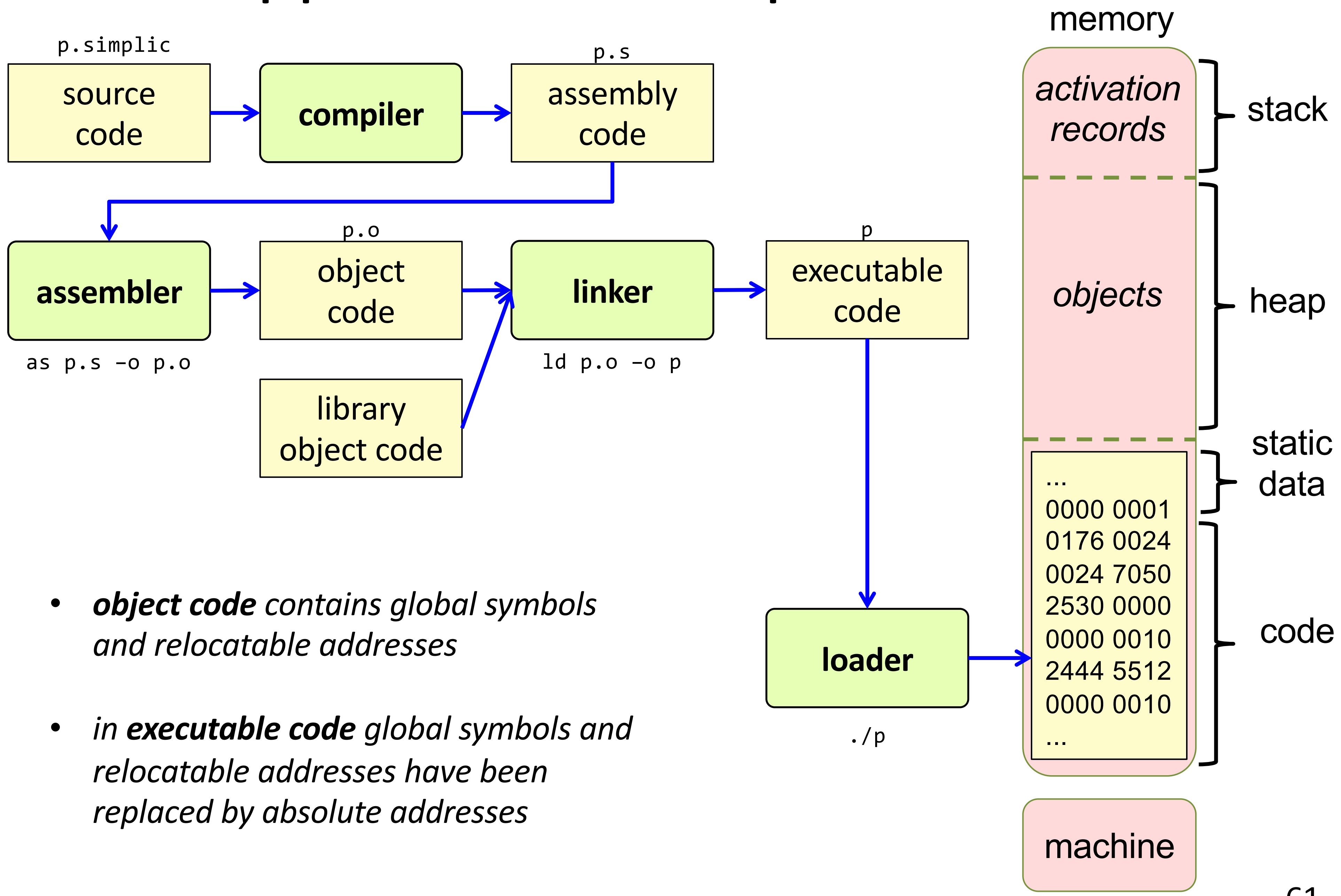
```
.global _start                # the _start label is globally visible
.data                         # data segment (for global data)
...
.text                         # text segment (for code, write protected)
_start:                       # execution starts here
        call m1               # call 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 is the difference between intermediate code and assembly code?
- Mention two kinds of typical intermediate code. When are they useful?
- Why is it not meaningful to minimze the number of temporaries in intermediate code?
- What is register allocation?
- Given a source program, sketch intermediate three address code.
- Given a source program, sketch x86 assembly code.
- 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 steps are needed to transform a program in assembly code to a binary executable program?