

Compilation

0368-3133 2014/15a

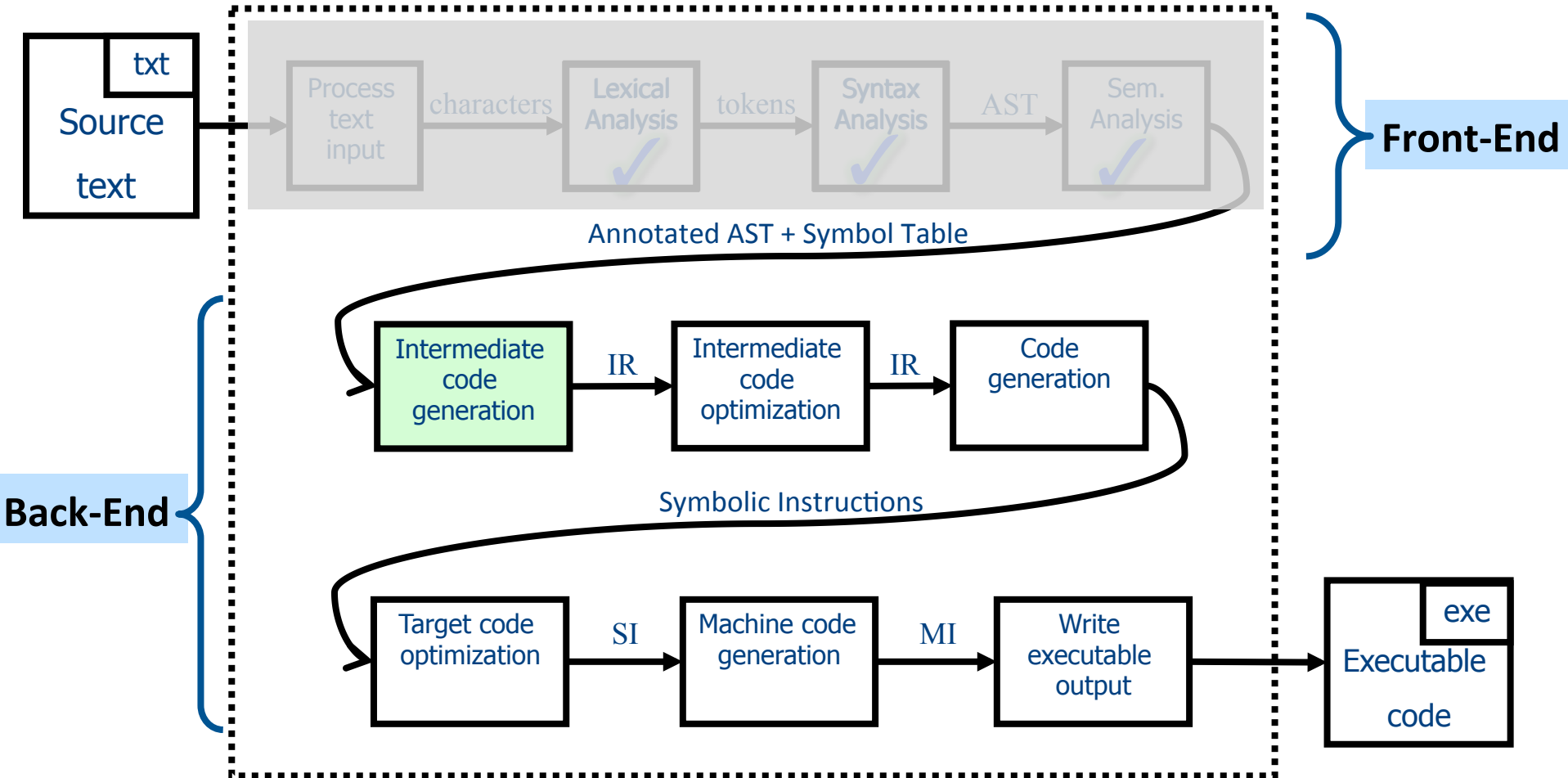
Lecture 8



IR for Procedure calls + Optimizations

Noam Rinetzky

Where we are



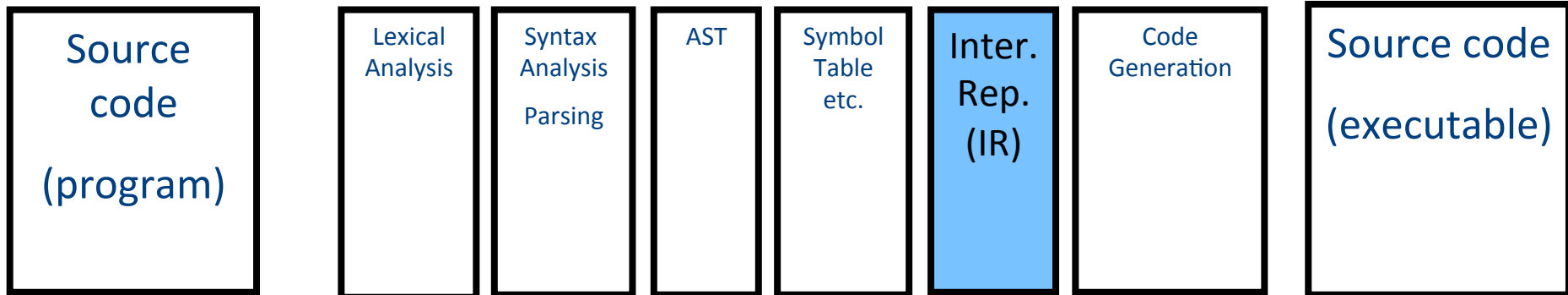


Intermediate Representation

1 Note: Compile Time vs Runtime

- **Compile time:** Data structures used during program compilation
 - AST, Symbol table
- **Runtime:** Data structures used during program execution
 - Activation record stack, memory management
- The **compiler generates code** that allows the program to **interact with the runtime**

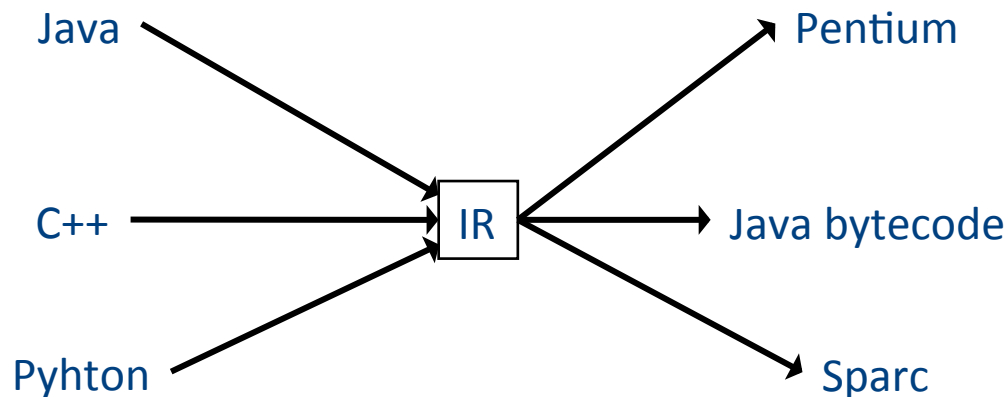
Code Generation: IR



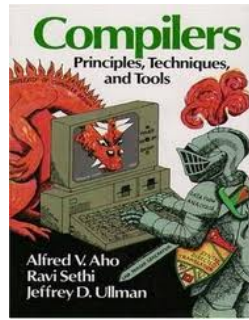
- Translating from abstract syntax (AST) to intermediate representation (IR)
 - **Three-Address Code**
- ...

Intermediate representation

- A language that is between the source language and the target language – not specific to any machine
- Goal 1: retargeting compiler components for different source languages/target machines
- Goal 2: machine-independent optimizer
 - Narrow interface: small number of instruction types



Three-Address Code IR



Chapter 8

- A popular form of IR
- High-level assembly where instructions have at most three operands

Reminder: **cgen** for expressions

```
cgen( $e_1 + e_2$ ) = {  
  Choose a new temporary  $t$   
  Let  $t_1 = \mathbf{cgen}(e_1)$   
  Let  $t_2 = \mathbf{cgen}(e_2)$   
  Emit(  $t = t_1 + t_2$  )  
  Return  $t$   
}
```

Algorithm by Ravi Sethi and Jeffrey D. Ullman
to minimize number of temporaries

Reminder: **cgen** for control

cgen(if (e) s_1 else s_2)

Let $_t$ = **cgen**(e)

Let L_{true} be a new label

Let L_{false} be a new label

Let L_{after} be a new label

Emit(IfZ $_t$ Goto L_{false} ;)

cgen(s_1)

Emit(Goto L_{after} ;)

Emit(L_{false} :)

cgen(s_2)

Emit(Goto L_{after} ;)

Emit(L_{after} :)

Code generation for procedure calls



Code generation for procedure calls

- Compile time generation of code for procedure invocations
- Activation Records (aka Stack Frames)

Supporting Procedures

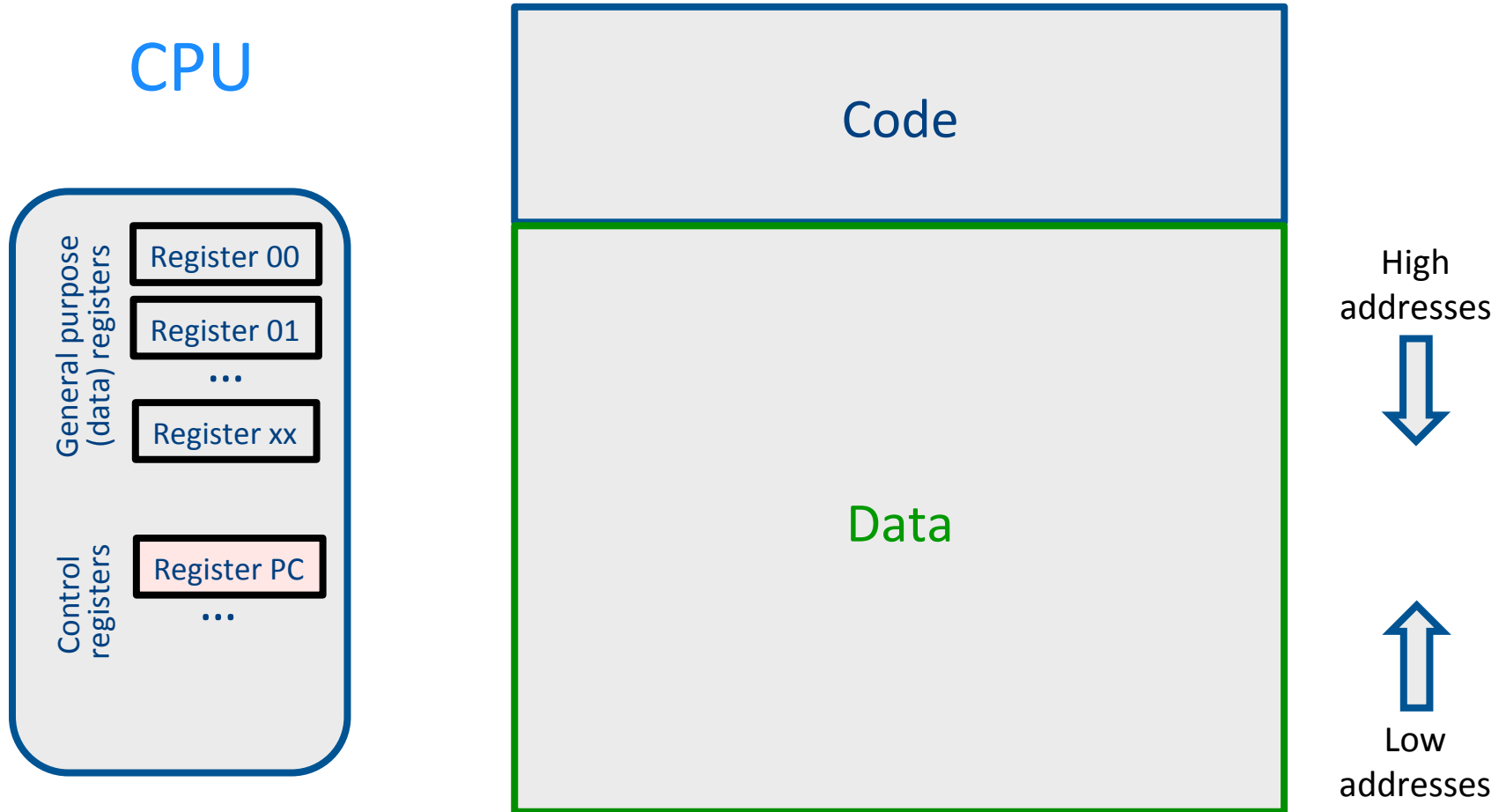
- **Stack**: a new computing environment
 - e.g., temporary memory for **local variables**
- Passing information into the new environment
 - **Parameters**
- **Transfer** of **control** to/from procedure
- Handling return values

Calling Conventions

- In general, compiler can use any convention to handle procedures
- In practice, CPUs specify standards
 - Aka calling conventios
 - Allows for compiler interoperability
 - Libraries!

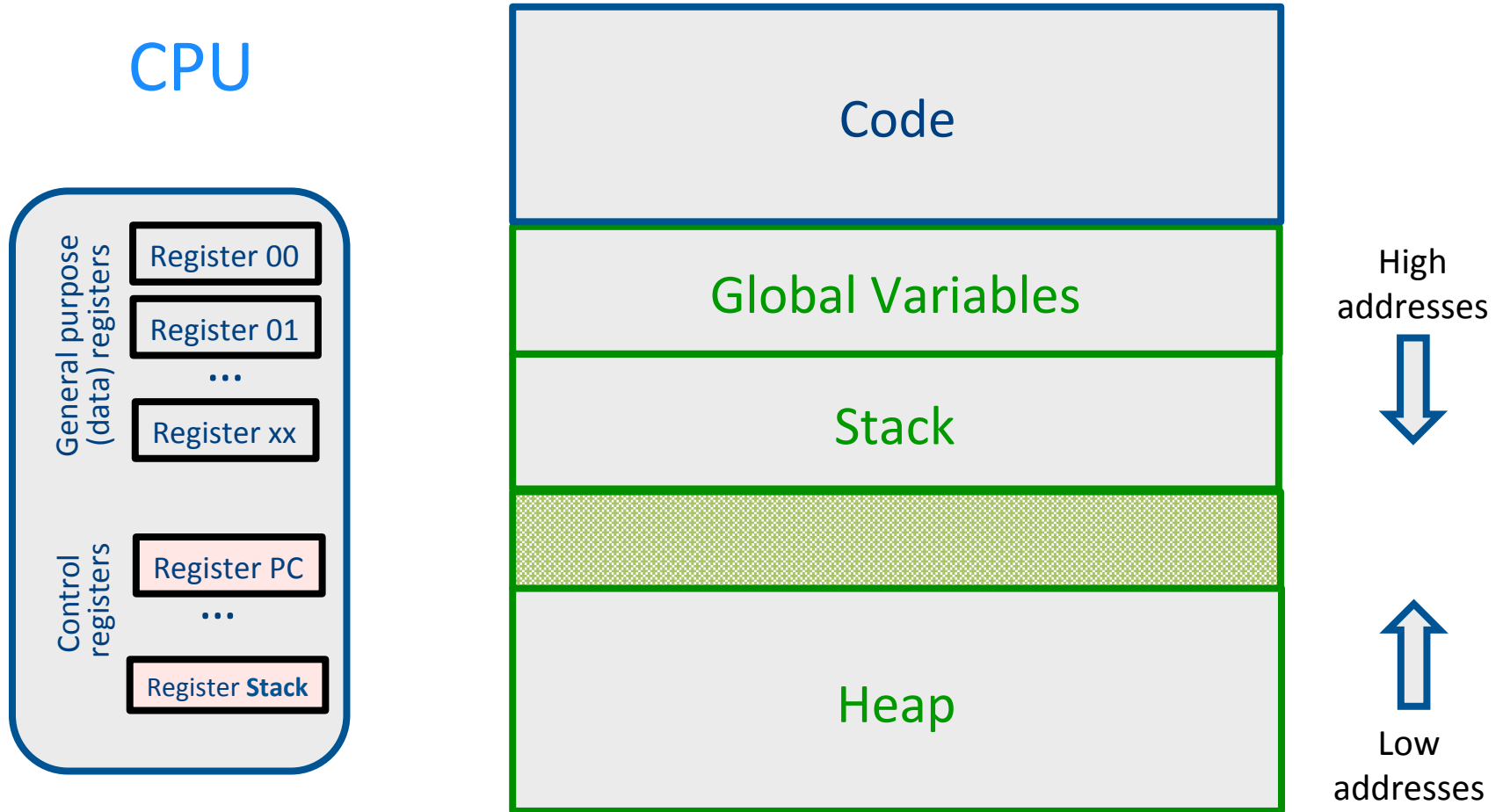
Abstract Register Machine

(High Level View)

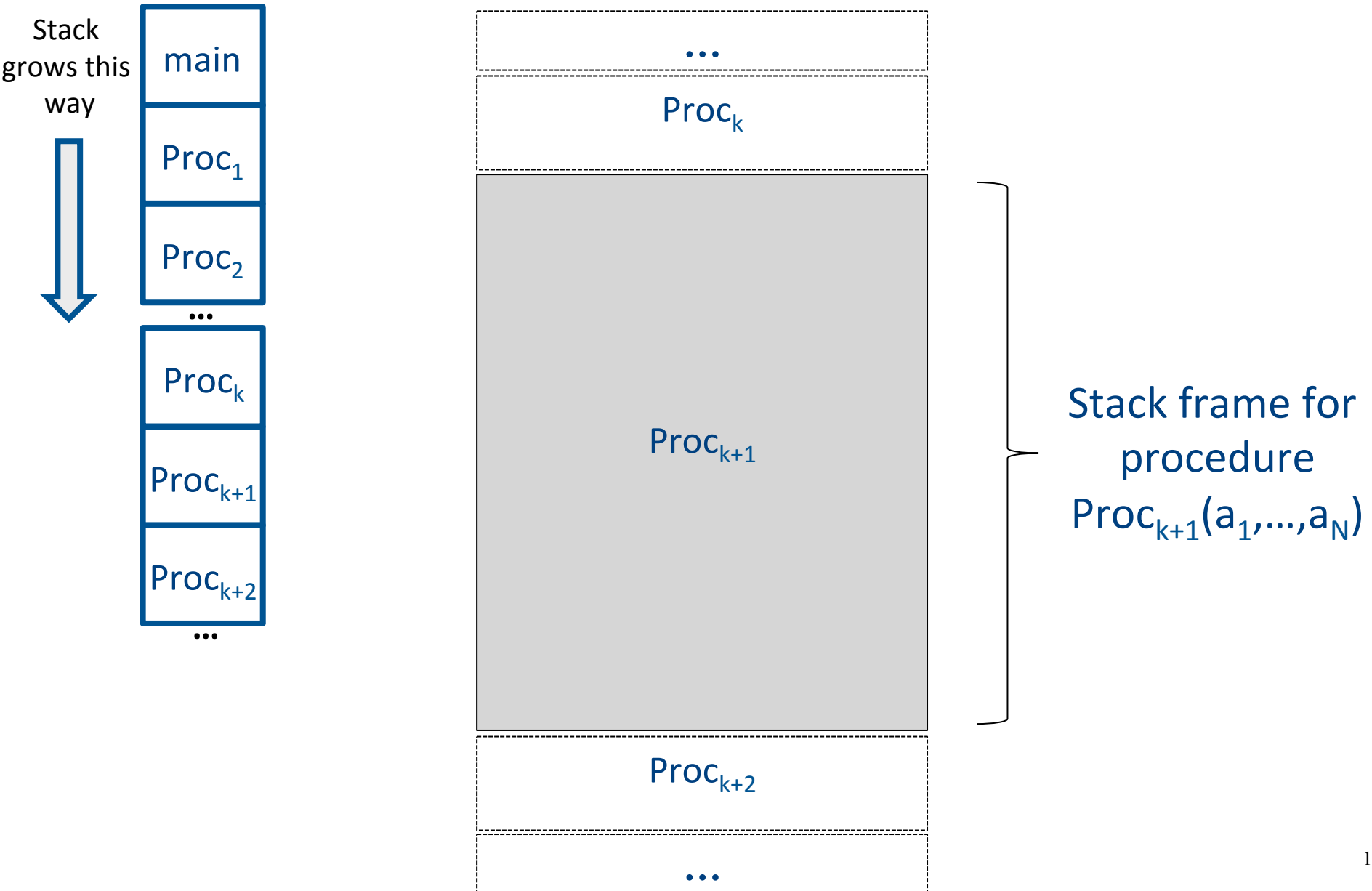


Abstract Register Machine

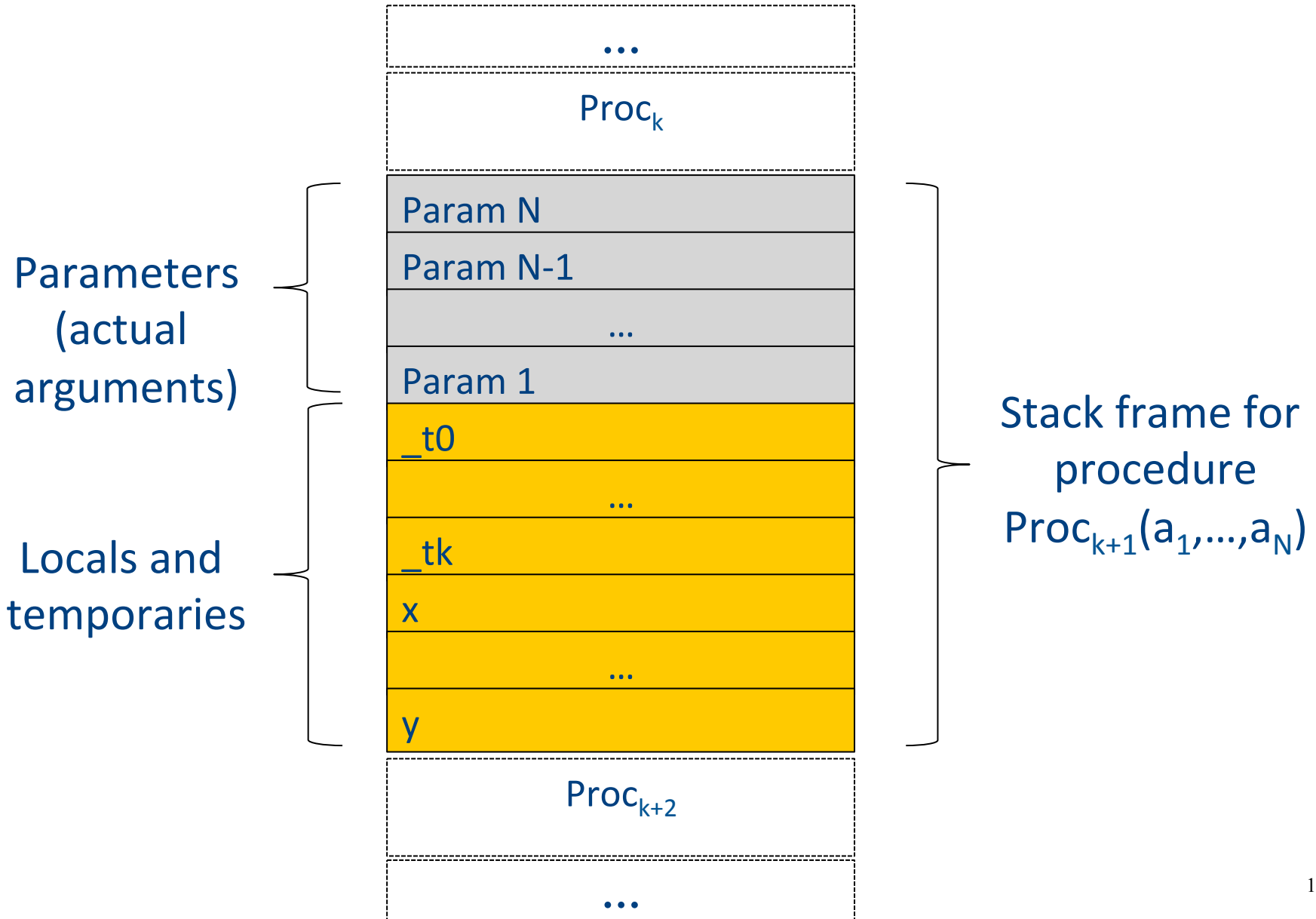
(High Level View)



Abstract Activation Record Stack



Abstract Stack Frame



Handling Procedures

- Store local variables/temporaries in a **stack**
- A function call instruction pushes arguments to stack and jumps to the function label

A statement **$x=f(a_1, \dots, a_n)$** ; looks like

Push a_1 ; ... Push a_n ;

Call f ;

Pop x ; // copy returned value

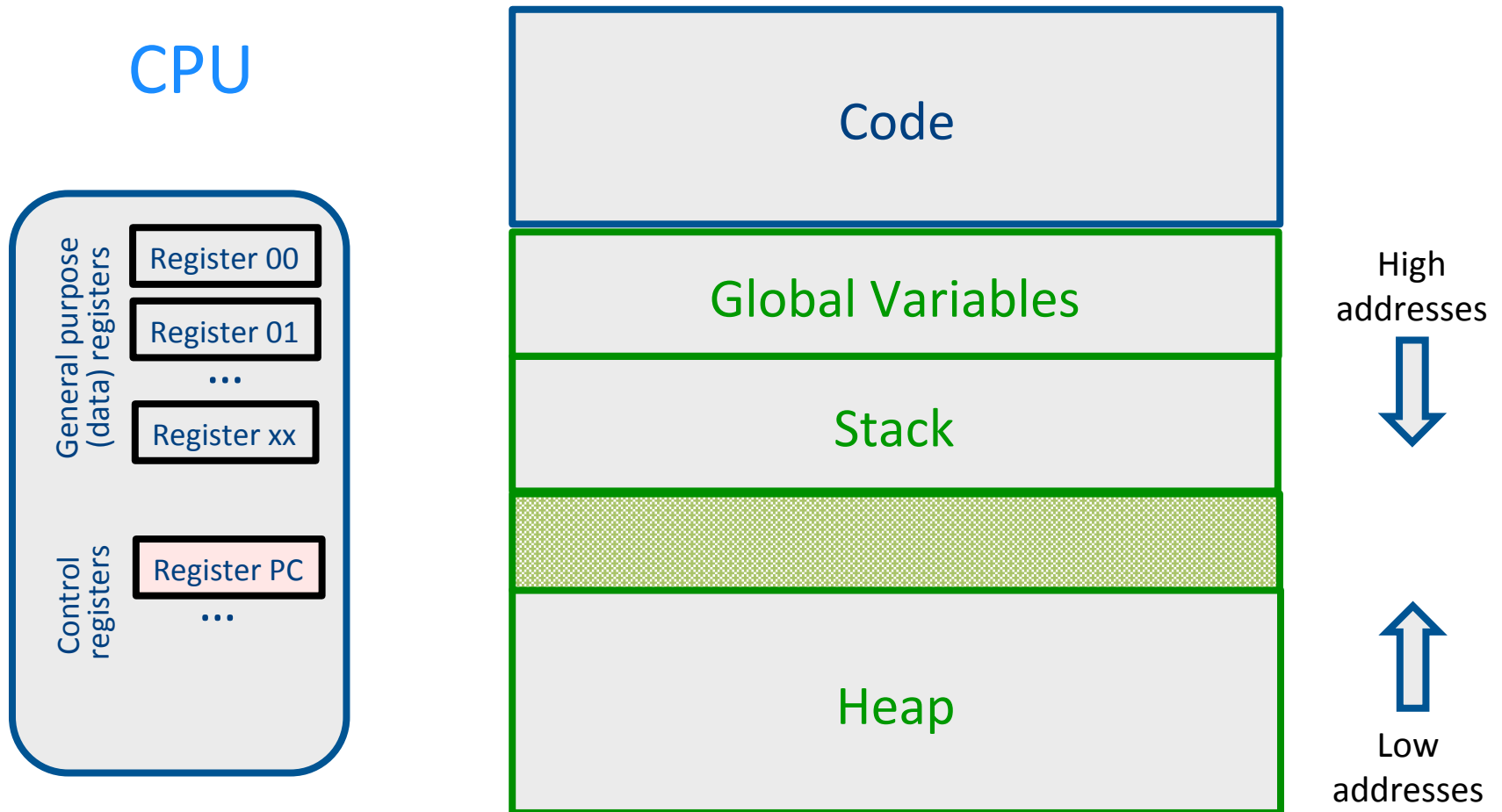
- Returning a value is done by pushing it to the stack (**return x ;**)

Push x ;

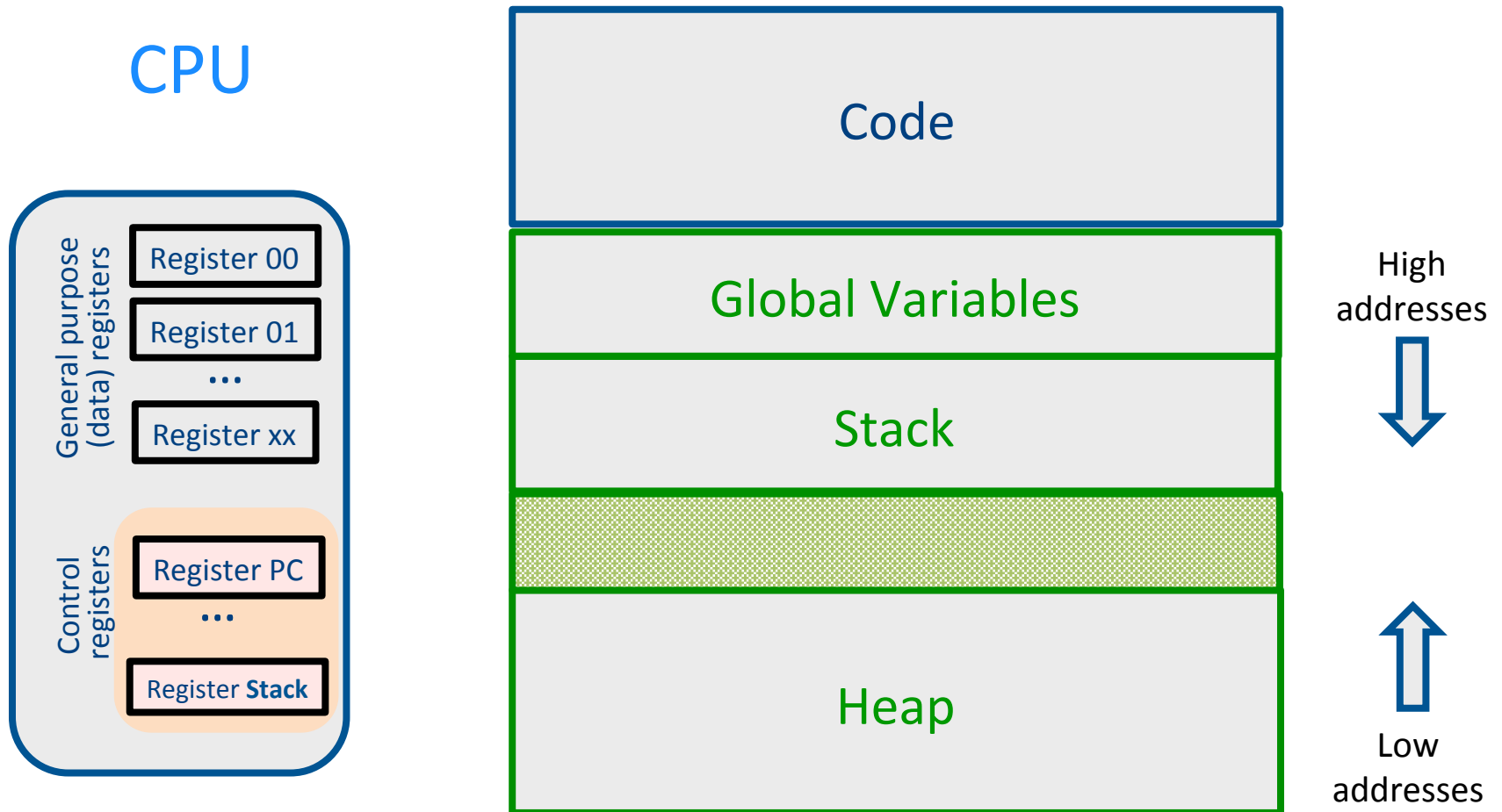
- Return control to caller (and roll up stack)

Return ;

Abstract Register Machine



Abstract Register Machine



Intro: Functions Example

```
int SimpleFn(int z) {
    int x, y;
    x = x * y * z;
    return x;
}

void main() {
    int w;
    w = SimpleFunction(137);
}
```

```
_SimpleFn:
    _t0 = x * y;
    _t1 = _t0 * z;
    x = _t1;
    Push x;
    Return;

main:
    _t0 = 137;
    Push _t0;
    Call _SimpleFn;
    Pop w;
```

What Can We Do with Procedures?

- Declarations & Definitions
- Call & Return
- Jumping out of procedures
- Passing & Returning procedures as parameters

Design Decisions

- Scoping rules
 - Static scoping vs. dynamic scoping
- Caller/callee conventions
 - Parameters
 - Who saves register values?
- Allocating space for local variables

Static (lexical) Scoping

```
main ( )
{
    int a = 0 ;
    int b = 0 ;
    {
        int b = 1 ;
        {
            B2 int a = 2 ;
            printf ("%d %d\n", a, b)
        }
        B1 {
            B3 int b = 3 ;
            printf ("%d %d\n", a, b) ;
        }
        printf ("%d %d\n", a, b) ;
    }
    printf ("%d %d\n", a, b) ;
}
```

a name refers to
its (closest)
enclosing **scope**

**known at
compile time**

Declaration	Scopes
a=0	B0,B1,B3
b=0	B0
b=1	B1,B2
a=2	B2
b=3	B3

Dynamic Scoping

- Each identifier is associated with a global stack of bindings
- When entering scope where identifier is declared
 - push declaration on identifier stack
- When exiting scope where identifier is declared
 - pop identifier stack
- **Evaluating the identifier in any context binds to the current top of stack**
- **Determined at runtime**

Example

```
int x = 42;
```

```
int f() { return x; }
```

```
int g() { int x = 1; return f(); }
```

```
int main() { return g(); }
```

- What value is returned from main?
- Static scoping?
- Dynamic scoping?

Why do we care?

- We need to generate code to access variables
- Static scoping
 - Identifier binding is known at compile time
 - “Address” of the variable is known at compile time
 - Assigning addresses to variables is part of code generation
 - No runtime errors of “access to undefined variable”
 - Can check types of variables

Variable addresses for static scoping: first attempt

```
int x = 42;
```

```
int f() { return x; }
```

```
int g() { int x = 1; return f(); }
```

```
int main() { return g(); }
```

identifier	address
x (global)	0x42
x (inside g)	0x73

Variable addresses for static scoping: first attempt

```
int a [11] ;

void quicksort(int m, int n) {
  int i;
  if (n > m) {
    i = partition(m, n);
    quicksort (m, i-1) ;
    quicksort (i+1, n) ;
  }

  main() {
  ...
  quicksort (1, 9) ;
}
```

**what is the address
of the variable “i” in
the procedure
quicksort?**

Compile-Time Information on Variables

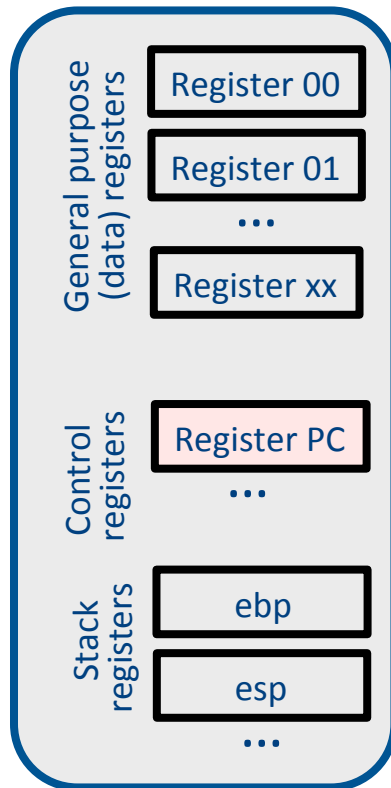
- Name
- Type
- Scope
 - when is it recognized
- Duration
 - Until when does its value exist
- Size
 - How many bytes are required at runtime
- Address
 - Fixed
 - Relative
 - Dynamic

Activation Record (Stack Frames)

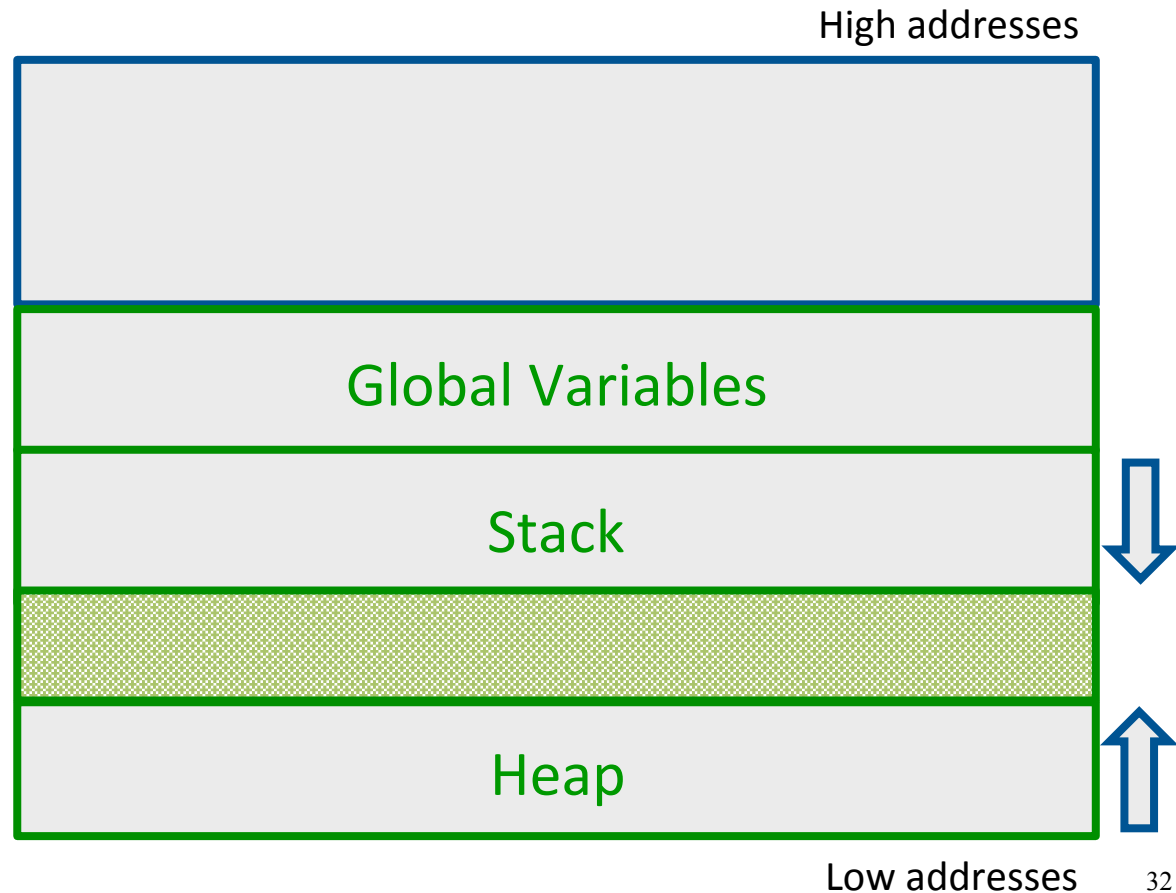
- separate space for each procedure **invocation**
- **managed at runtime**
 - **code for managing it generated by the compiler**
- **desired properties**
 - efficient allocation and deallocation
 - procedures are called frequently
 - variable size
 - different procedures may require different memory sizes

Semi-Abstract Register Machine

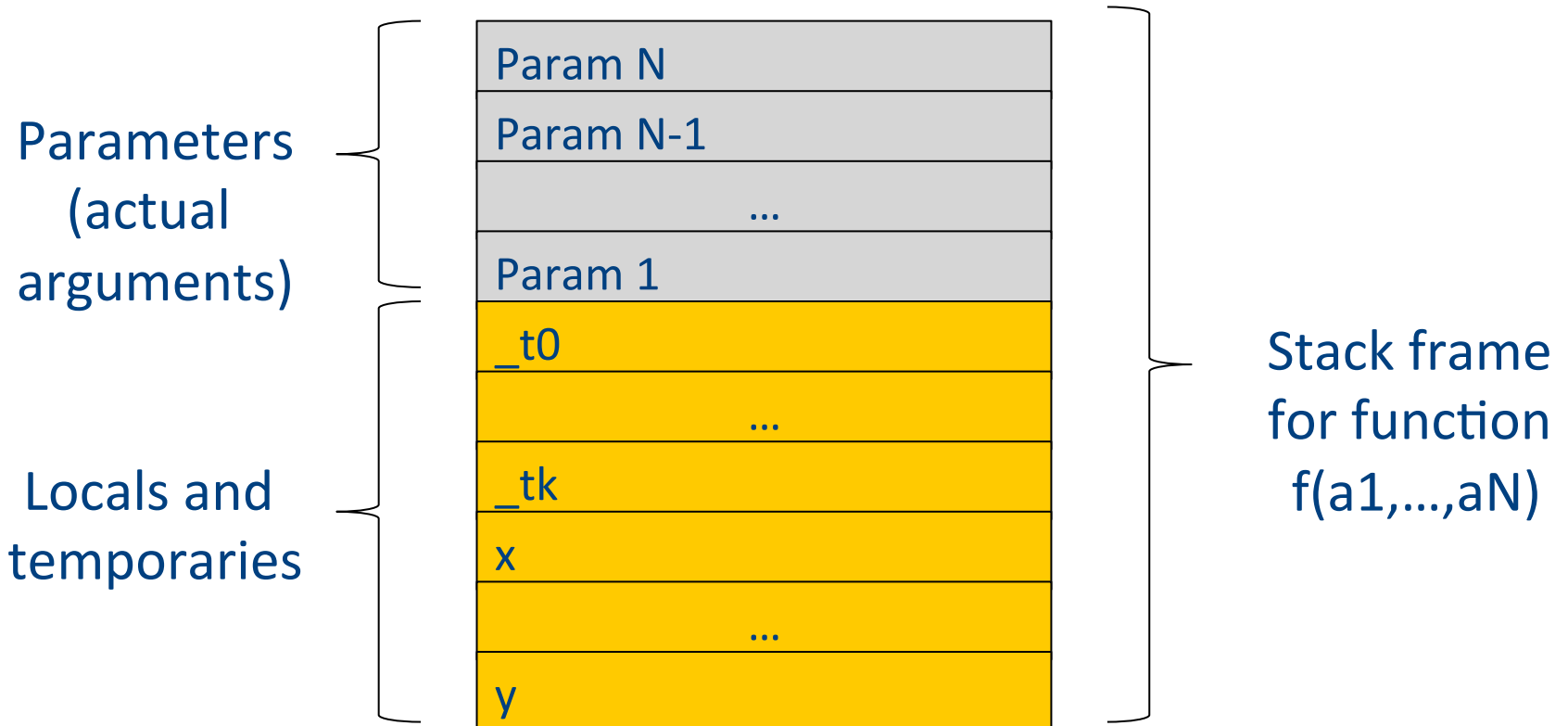
CPU



Main Memory



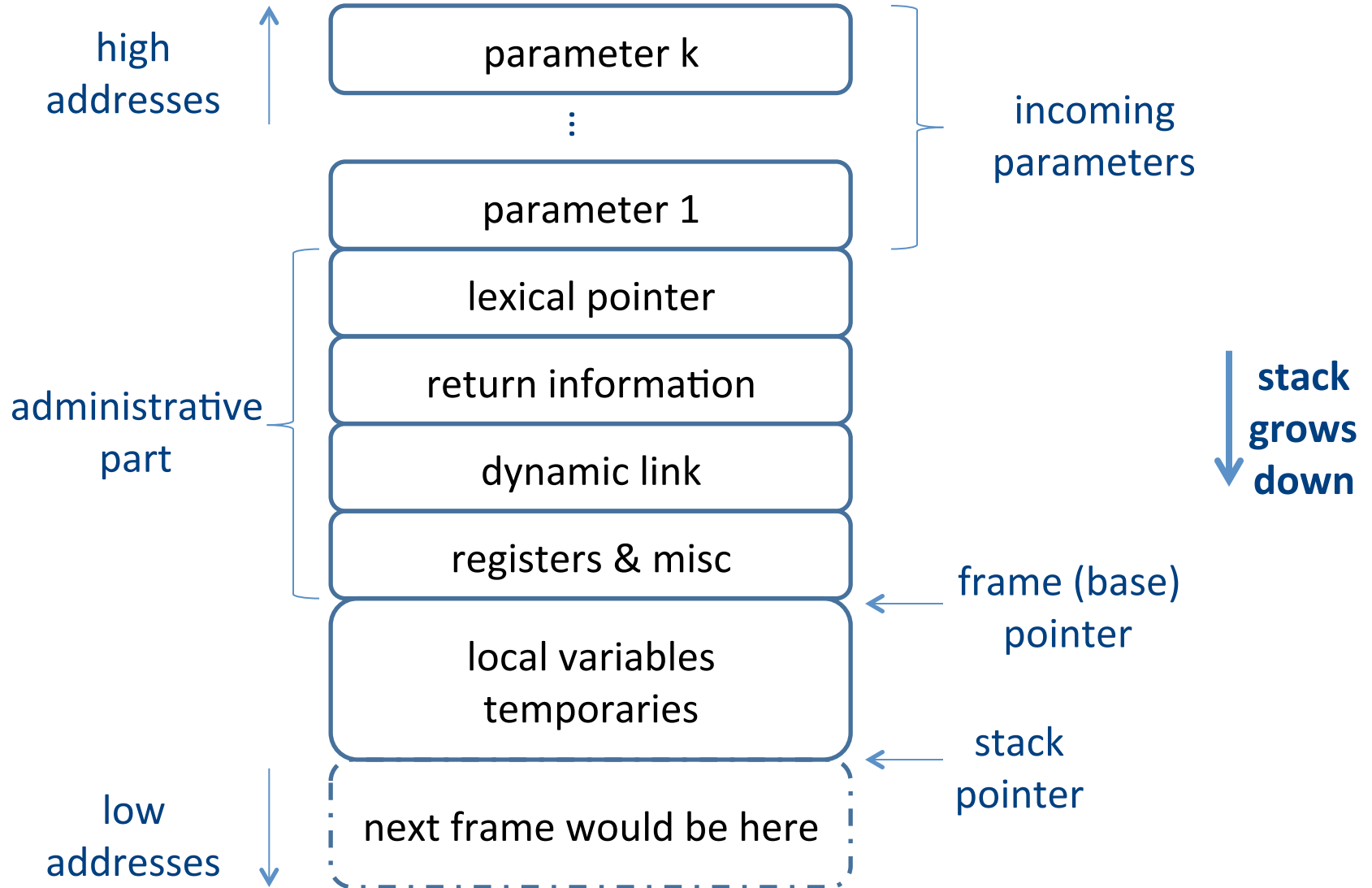
A Logical Stack Frame (Simplified)



Runtime Stack

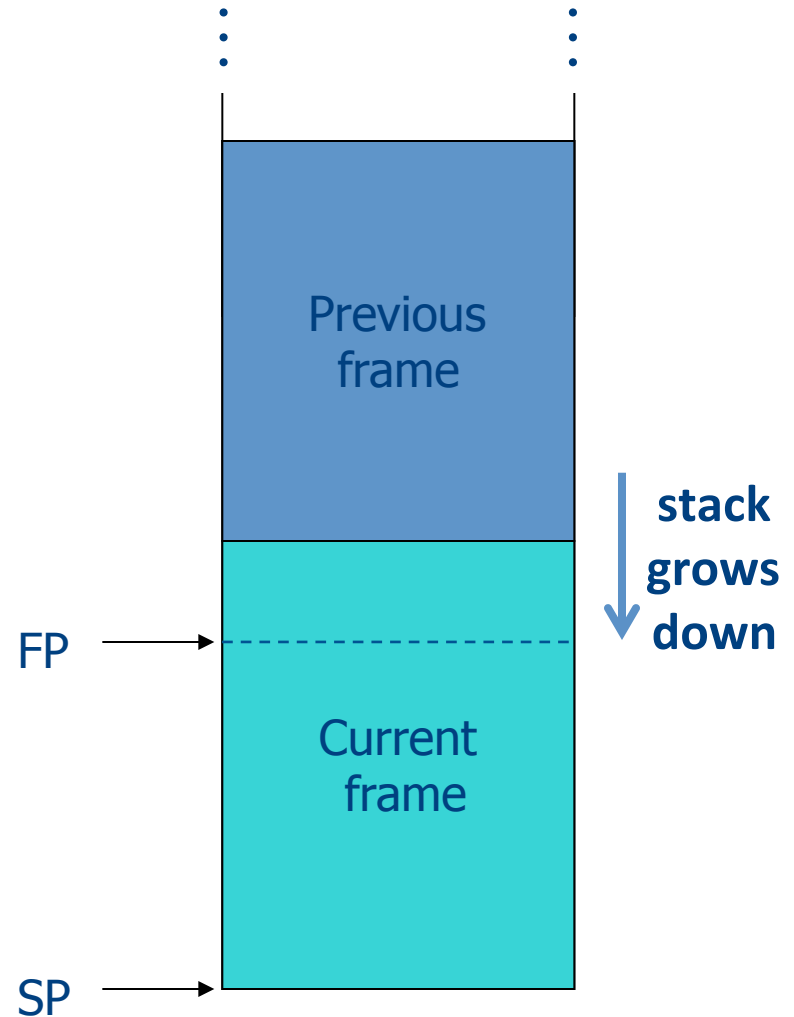
- Stack of activation records
- Call = push new activation record
- Return = pop activation record
- Only one “active” activation record – top of stack
- How do we handle recursion?

Activation Record (frame)



Runtime Stack

- SP – stack pointer
 - top of current frame
- FP – frame pointer
 - base of current frame
 - Sometimes called BP (base pointer)
 - Usually points to a “fixed” offset from the “start” of the frame



Code Blocks

- Programming language provide code blocks

```
void foo()  
{  
  int x = 8 ; y=9;//1  
  { int x = y * y ;//2 }  
  { int x = y * 7 ;//3}  
    x = y + 1;  
}
```

administrative
x1
y1
x2
x3
...

L-Values of Local Variables

- The offset in the stack is known at compile time
- $L\text{-val}(x) = FP + \text{offset}(x)$
- $x = 5 \Rightarrow$ Load_Constant 5, R3
Store R3, $\text{offset}(x)(FP)$

Pentium Runtime Stack

Register	Usage
ESP	Stack pointer
EBP	Base pointer

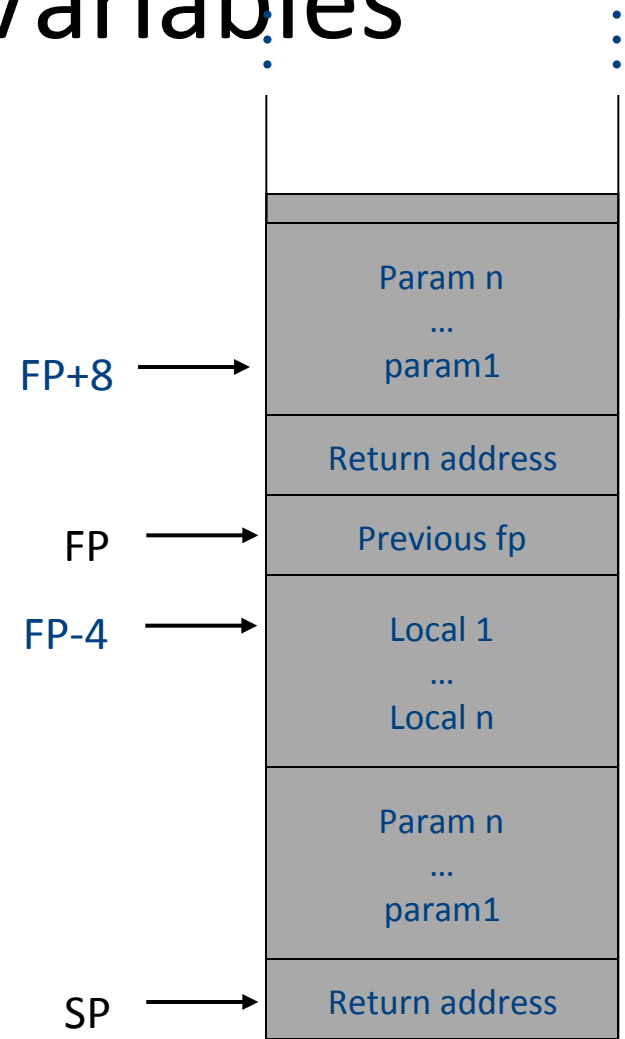
Pentium stack registers

Instruction	Usage
push, pusha,...	push on runtime stack
pop,popa,...	Base pointer
call	transfer control to called routine
return	transfer control back to caller

Pentium stack and call/ret instructions

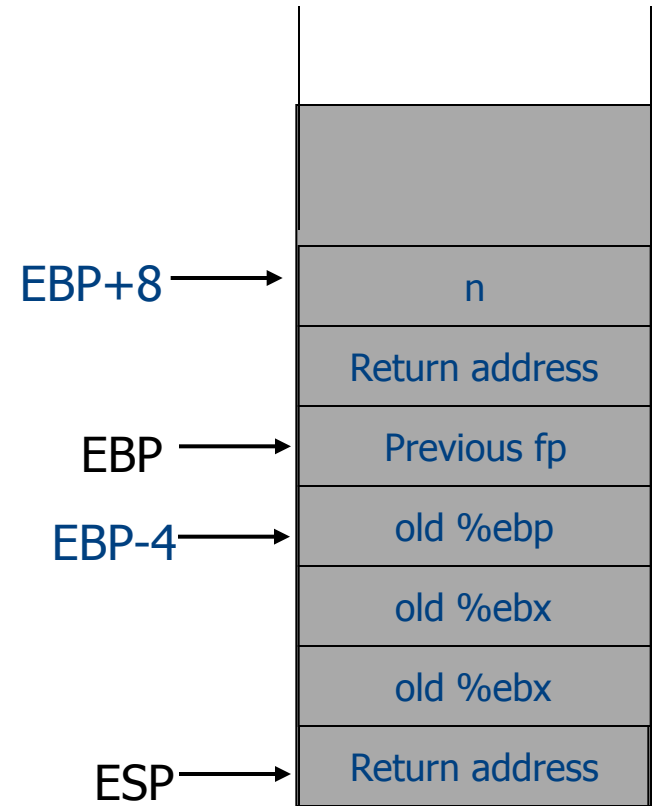
Accessing Stack Variables

- Use offset from FP (%ebp)
 - Remember: stack grows downwards
- Above FP = parameters
- Below FP = locals
- Examples
 - $\%ebp + 4 = \text{return address}$
 - $\%ebp + 8 = \text{first parameter}$
 - $\%ebp - 4 = \text{first local}$



Factorial – fact (int n)

```
fact:
pushl %ebp           # save ebp
movl %esp,%ebp      # ebp=esp
pushl %ebx          # save ebx
movl 8(%ebp),%ebx   # ebx = n
cmpl $1,%ebx        # n = 1 ?
jle .lresult        # then done
leal -1(%ebx),%eax  # eax = n-1
pushl %eax          #
call fact           # fact(n-1)
imull %ebx,%eax     # eax=retv*n
jmp .lreturn        #
.lresult:
movl $1,%eax        # retv
.lreturn:
movl -4(%ebp),%ebx  # restore ebx
movl %ebp,%esp      # restore esp
popl %ebp           # restore ebp
```



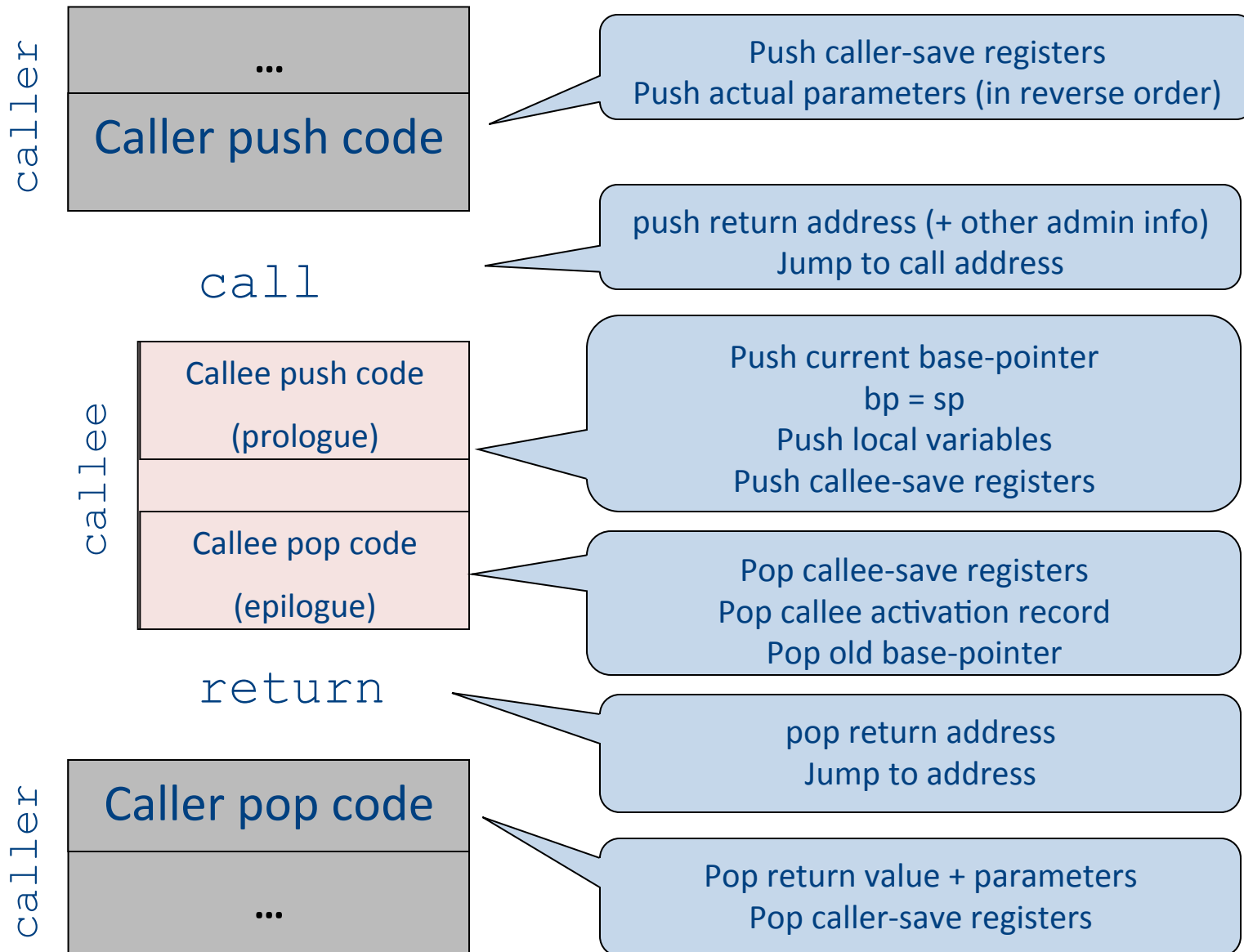
(stack in intermediate point)

(disclaimer: real compiler can do better than that)

Call Sequences

- The **processor does not save** the content of **registers** on procedure calls
- So who will?
 - Caller saves and restores registers
 - Callee saves and restores registers
 - But can also have both save/restore some registers

Call Sequences



“To Callee-save or to Caller-save?”

- Callee-saved registers need only be saved when callee modifies their value
- some heuristics and conventions are followed

Caller-Save and Callee-Save Registers

- Callee-Save Registers
 - Saved by the callee before modification
 - Values are automatically preserved across calls
- Caller-Save Registers
 - Saved (if needed) by the caller before calls
 - Values are not automatically preserved across calls
- Usually the architecture defines caller-save and callee-save registers
- Separate compilation
- Interoperability between code produced by different compilers/languages
- But compiler writers decide when to use caller/callee registers

Callee-Save Registers

- Saved by the callee before modification
- Usually at procedure prolog
- Restored at procedure epilog
- Hardware support may be available
- Values are automatically preserved across calls

```
int foo(int a) {           .global _foo
    int b=a+1;           Add_Constant -K, SP //allocate space for foo
    f1();                Store_Local R5, -14(FP) // save R5
    g1(b);               Load_Reg R5, R0; Add_Constant R5, 1
    return(b+2);        JSR f1 ; JSR g1;
                        Add_Constant R5, 2; Load_Reg R5, R0
                        Load_Local -14(FP), R5 // restore R5
                        Add_Constant K, SP; RTS // deallocate
}
```

Caller-Save Registers

- Saved by the caller before calls when needed
- Values are not automatically preserved across calls

```
void bar (int y) {  
    int x=y+1;  
    f2(x);  
    g2(2);  
    g2(8);  
}  
  
    .global _bar  
    Add_Constant -K, SP //allocate space for bar  
    Add_Constant R0, 1  
    JSR f2  
    Load_Constant 2, R0 ;    JSR g2;  
    Load_Constant 8, R0 ;    JSR g2  
    Add_Constant K, SP // deallocate space for bar  
    RTS
```

Parameter Passing

- 1960s
 - In memory
 - No recursion is allowed
- 1970s
 - In stack
- 1980s
 - In registers
 - First k parameters are passed in registers ($k=4$ or $k=6$)
 - Where is time saved?
- Most procedures are leaf procedures
- Interprocedural register allocation
- Many of the registers may be dead before another invocation
- Register windows are allocated in some architectures per call (e.g., sun Sparc)

Activation Records & Language Design

Compile-Time Information on Variables

- Name, type, size
- Address kind
 - Fixed (global)
 - Relative (local)
 - Dynamic (heap)
- Scope
 - when is it recognized
- Duration
 - Until when does its value exist

Scoping

```
int x = 42;
```

```
int f() { return x; }
```

```
int g() { int x = 1; return f(); }
```

```
int main() { return g(); }
```

- What value is returned from main?
- Static scoping?
- Dynamic scoping?

Nested Procedures

- For example – Pascal
- Any routine can have sub-routines
- Any sub-routine can access anything that is defined in its containing scope or inside the sub-routine itself
 - “non-local” variables

Example: Nested Procedures

```
program p() {  
  int x;  
  procedure a() {  
    int y;  
    [ procedure b() { ... c() ... };  
    [ procedure c() {  
      int z;  
      [ procedure d() {  
        y := x + z  
      };  
      ... b() ... d() ...  
    }  
    ... a() ... c() ...  
  }  
}  
a()  
}
```

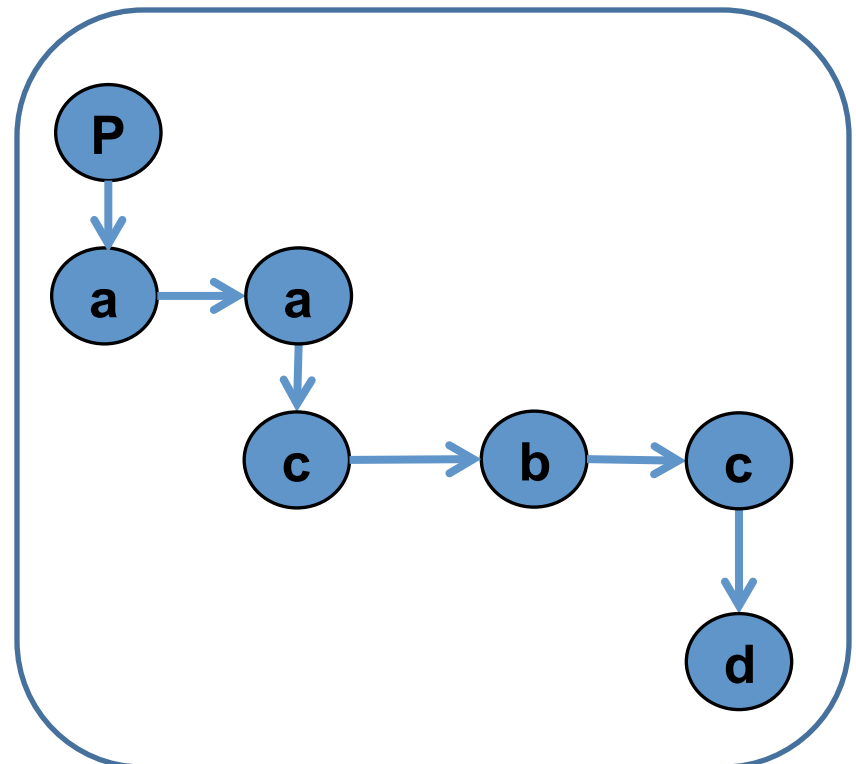
Possible call sequence:
 $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$

what are the addresses
of variables “**x**,” “**y**” and
“**z**” in procedure d?

Nested Procedures

- **can call a sibling, ancestor**
- when “c” uses (non-local) variables from “a”, which instance of “a” is it?
- how do you find the right activation record at runtime?

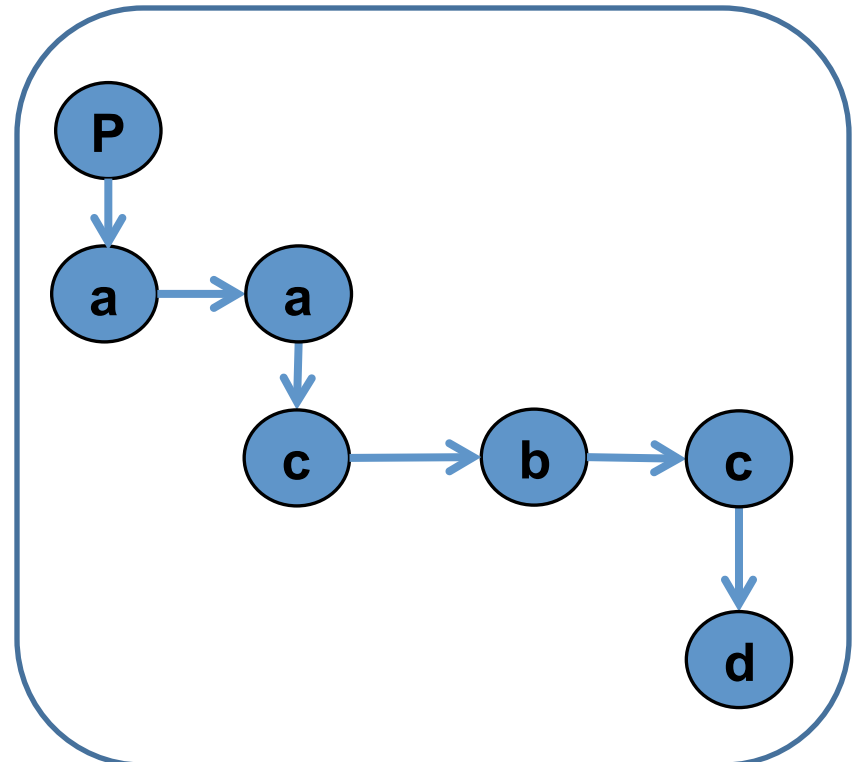
Possible call sequence:
 $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$



Nested Procedures

- goal: **find the closest routine in the stack from a given nesting level**
- if we reached the same routine in a sequence of calls
 - routine of level k uses variables of the same nesting level, it uses its own variables
 - if it uses variables of nesting level $j < k$ then it must be the last routine called at level j
- If a procedure is last at level j on the stack, then it must be ancestor of the current routine

Possible call sequence:
 $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$



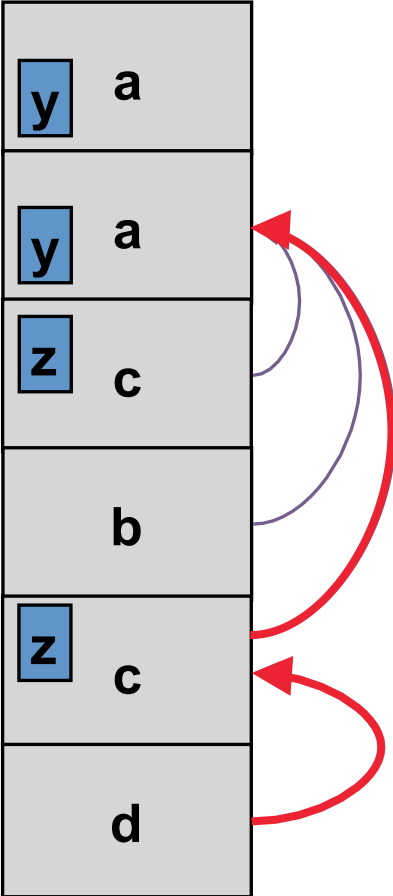
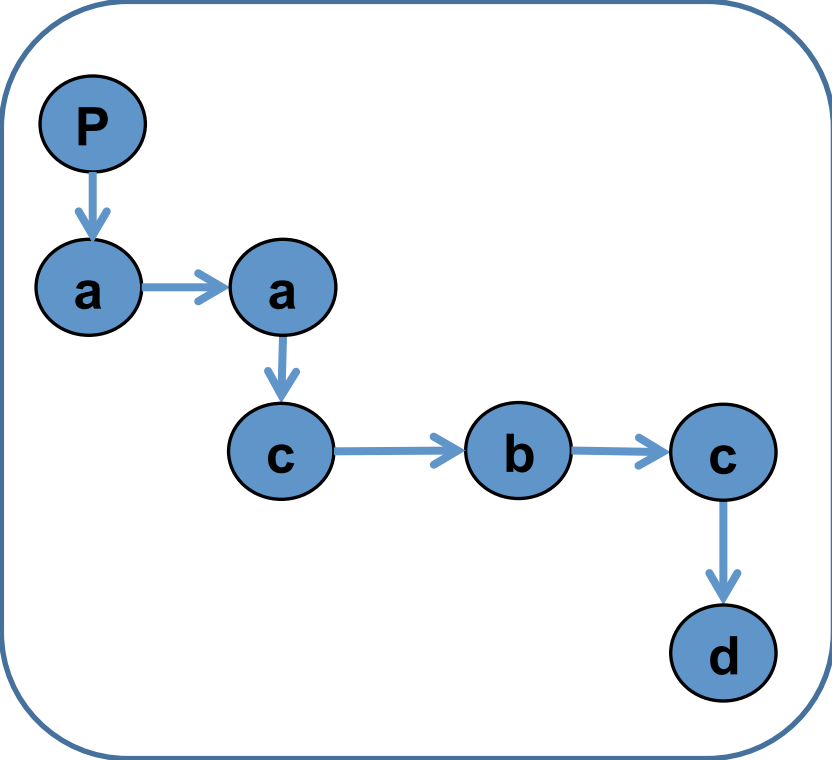
Nested Procedures

- problem: a routine may need to access variables of another routine that contains it statically
- solution: lexical pointer (a.k.a. access link) in the activation record
- lexical pointer points to the last activation record of the nesting level above it
 - in our example, lexical pointer of d points to activation records of c
- lexical pointers created at runtime
- number of links to be traversed is known at compile time

Lexical Pointers

```
program p() {  
  int x;  
  procedure a() {  
    int y;  
    [ procedure b() { c() };  
    procedure c() {  
      int z;  
      [ procedure d() {  
        y := x + z;  
      };  
      ... b() ... d() ...  
    }  
    ... a() ... c() ...  
  }  
  a()  
}
```

Possible call sequence:
 $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$



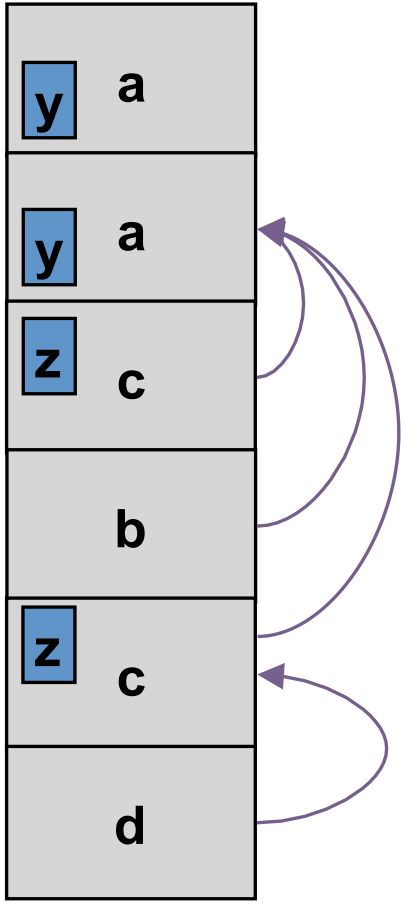
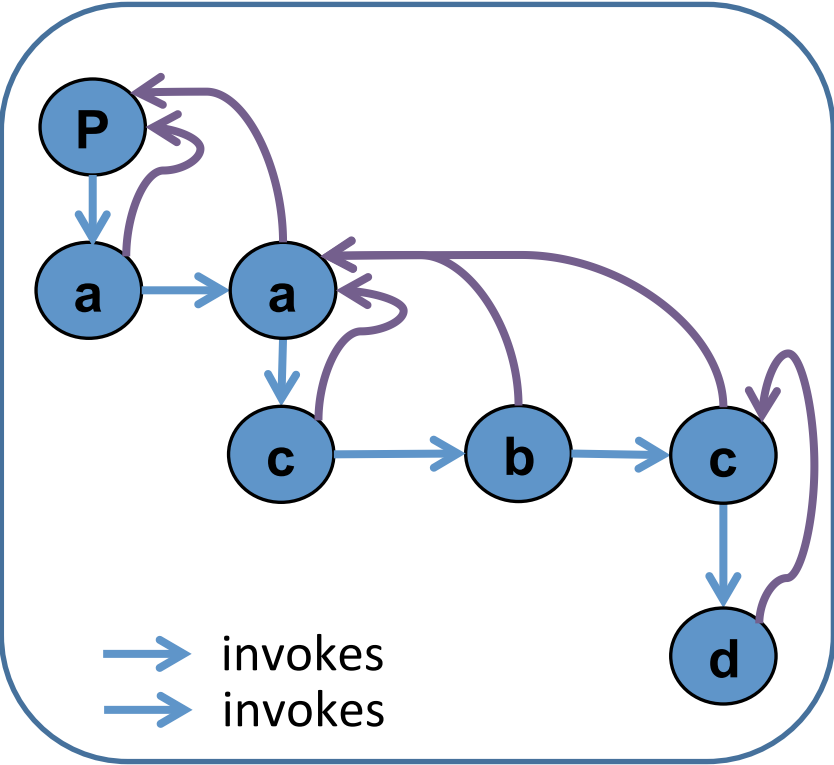
Lexical Pointers

```

program p() {
  int x;
  procedure a() {
    int y;
    [ procedure b() { c() };
      procedure c() {
        int z;
        [ procedure d() {
          y := x + z;
        };
        ... b() ... d() ...
      ]
    ... a() ... c() ...
  }
  a()
}

```

Possible call sequence:
 $p \rightarrow a \rightarrow a \rightarrow c \rightarrow b \rightarrow c \rightarrow d$



Activation Records: Remarks

Non-Local goto in C syntax

```
void level_0(void) {  
    void level_1(void) {  
        void level_2(void) {  
            ...  
            goto L_1;  
            ...  
        }  
        ...  
L_1: ...  
        ...  
    }  
    ...  
}
```

Non-local gotos in C

- `setjmp` remembers the current location and the stack frame
- `longjmp` jumps to the current location (popping many activation records)

Non-Local Transfer of Control in C

```
#include <setjmp.h>

void find_div_7(int n, jmp_buf *jmpbuf_ptr) {
    if (n % 7 == 0) longjmp(*jmpbuf_ptr, n);
    find_div_7(n + 1, jmpbuf_ptr);
}

int main(void) {
    jmp_buf jmpbuf;          /* type defined in setjmp.h */
    int return_value;

    if ((return_value = setjmp(jmpbuf)) == 0) {
        /* setting up the label for longjmp() lands here */
        find_div_7(1, &jmpbuf);
    }
    else {
        /* returning from a call of longjmp() lands here */
        printf("Answer = %d\n", return_value);
    }
    return 0;
}
```

Stack Frames

- Allocate a separate space for every procedure incarnation
- Relative addresses
- Provide a simple mean to achieve modularity
- Supports separate code generation of procedures
- Naturally supports recursion
- Efficient memory allocation policy
 - Low overhead
 - Hardware support may be available
- LIFO policy
- Not a pure stack
 - Non local references
 - Updated using arithmetic

The Frame Pointer

- The caller
 - the calling routine
- The callee
 - the called routine
- caller responsibilities:
 - Calculate arguments and save in the stack
 - Store lexical pointer
- call instruction:
 - $M[--SP] := RA$
 - $PC := \text{callee}$
- callee responsibilities:
 - $FP := SP$
 - $SP := SP - \text{frame-size}$
- Why use both SP and FP?

Variable Length Frame Size

- C allows allocating objects of unbounded size in the stack

```
void p() {  
    int i;  
    char *p;  
    scanf("%d", &i);  
    p = (char *) alloca(i*sizeof(int));  
}
```

- Some versions of Pascal allows conformant array value parameters

Limitations

- The compiler may be forced to store a value on a stack instead of registers
- The stack may not suffice to handle some language features

Frame-Resident Variables

- A variable x cannot be stored in register when:
 - x is passed by reference
 - Address of x is taken ($\&x$)
 - is addressed via pointer arithmetic on the stack-frame (C varargs)
 - x is accessed from a nested procedure
 - The value is too big to fit into a single register
 - The variable is an array
 - The register of x is needed for other purposes
 - Too many local variables
- An escape variable:
 - Passed by reference
 - Address is taken
 - Addressed via pointer arithmetic on the stack-frame
 - Accessed from a nested procedure

The Frames in Different Architectures

$g(x, y, z)$ where x escapes

	Pentium	MIPS	Sparc
x	InFrame(8)	InFrame(0)	InFrame(68)
y	InFrame(12)	InReg(X_{157})	InReg(X_{157})
z	InFrame(16)	InReg(X_{158})	InReg(X_{158})
View Change	$M[sp+0] \leftarrow fp$ $fp \leftarrow sp$ $sp \leftarrow sp-K$	$sp \leftarrow sp-K$ $M[sp+K+0] \leftarrow r_2$ $X_{157} \leftarrow r_4$ $X_{158} \leftarrow r_5$	$save\ \%sp,\ -K,\ \%sp$ $M[fp+68] \leftarrow i_0$ $X_{157} \leftarrow i_1$ $X_{158} \leftarrow i_2$

Limitations of Stack Frames

- A local variable of P cannot be stored in the activation record of P if its duration exceeds the duration of P
- Example 1: Static variables in C
(own variables in Algol)

```
void p(int x)
{
    static int y = 6 ;
    y += x;
}
```

- Example 2: Features of the C language

```
int * f()
{ int x ;
  return &x ;
}
```

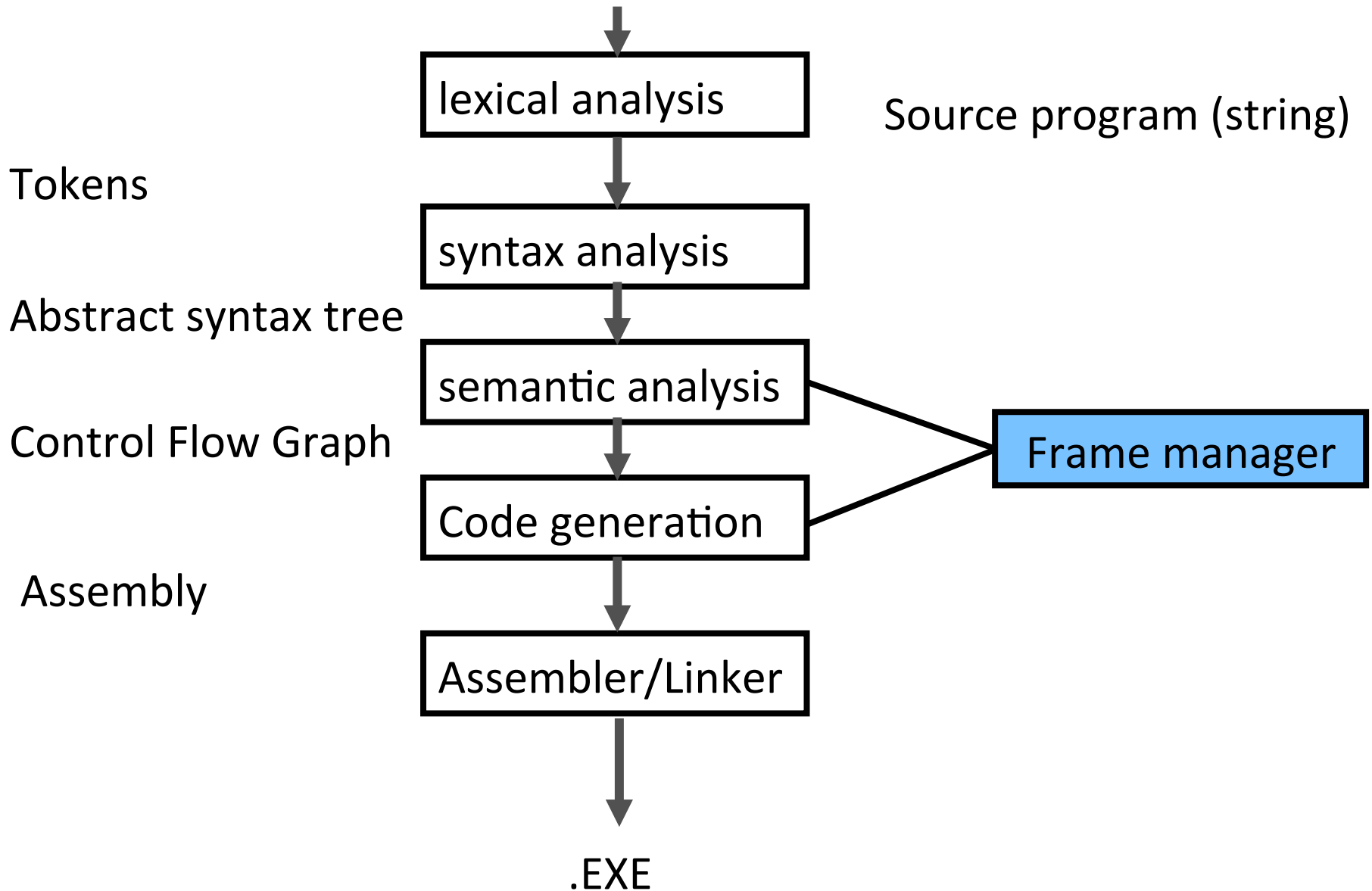
- Example 3: Dynamic allocation

```
int * f() { return (int *)
malloc(sizeof(int)); }
```

Compiler Implementation

- Hide machine dependent parts
- Hide language dependent part
- Use special modules

Basic Compiler Phases



Hidden in the frame ADT

- Word size
- The location of the formals
- Frame resident variables
- Machine instructions to implement “shift-of-view” (prologue/epilogue)
- The number of locals “allocated” so far
- The label in which the machine code starts

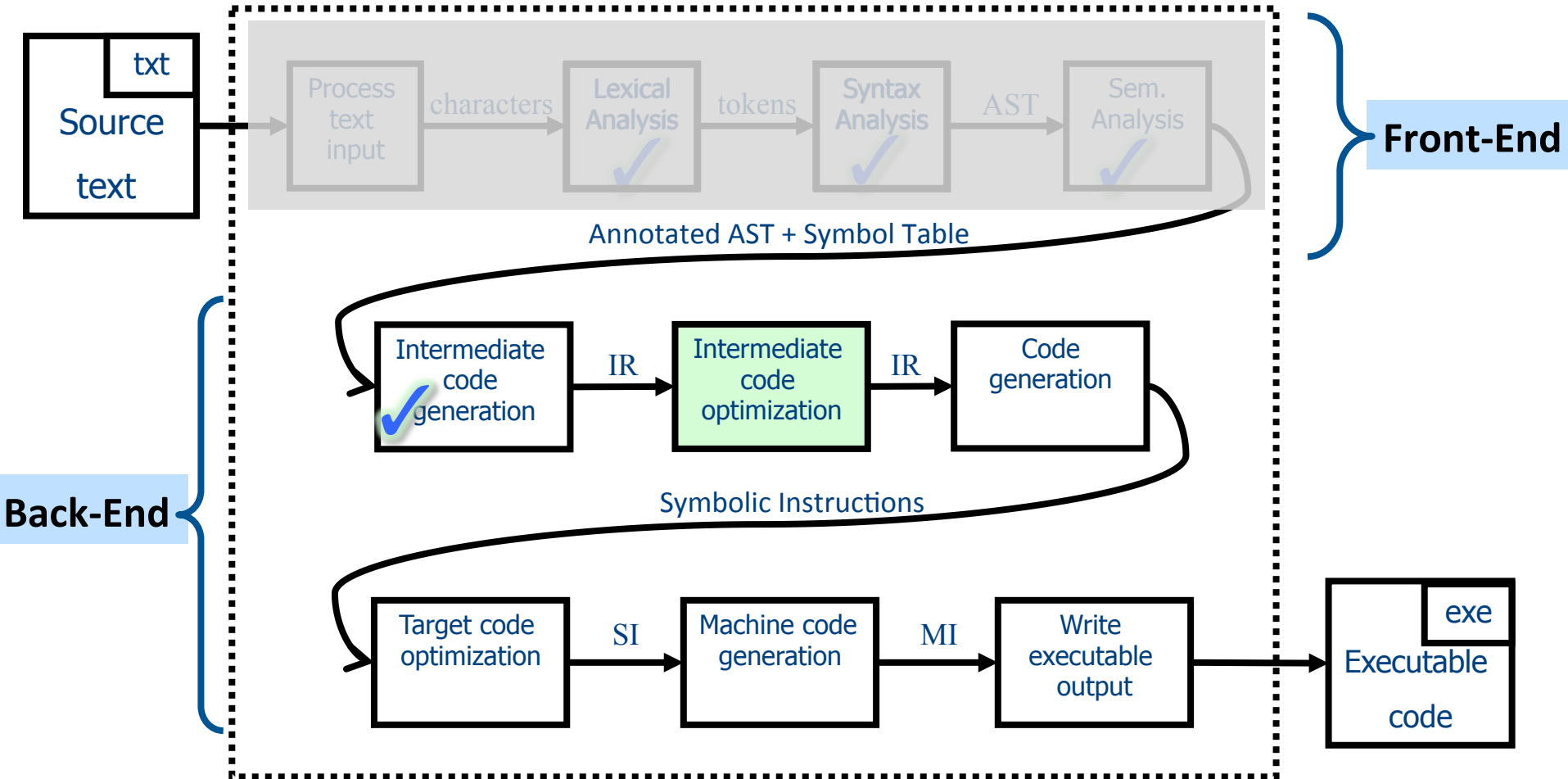
Activation Records: Summary

- compile time memory management for procedure data
- works well for data with well-scoped lifetime
 - deallocation when procedure returns

IR Optimization



Where we are



IR Optimization

- Making code better

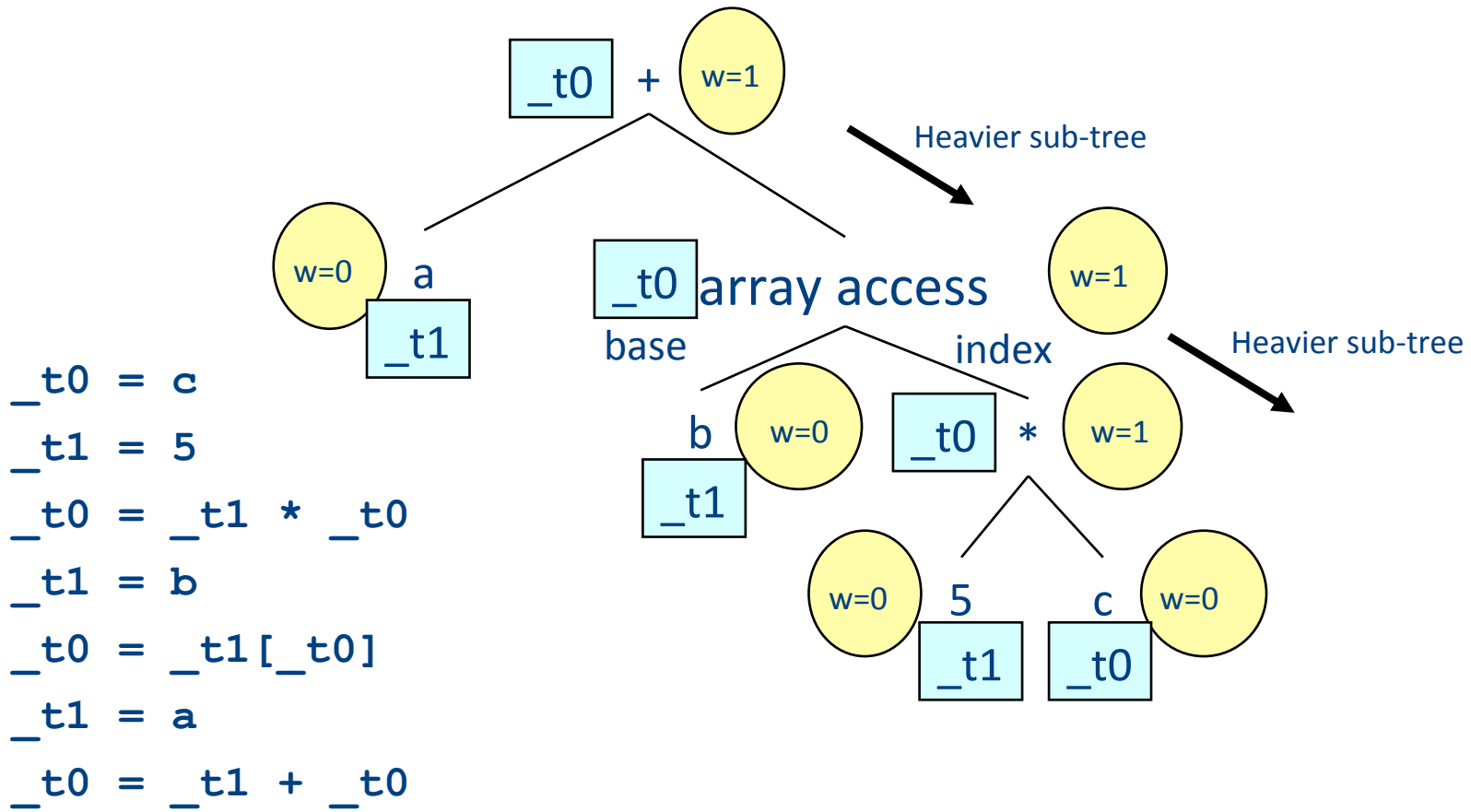
IR Optimization

- Making code “better”

“Optimized” evaluation

`_t0 = cgen(a+b[5*c])`

Phase 2: - use weights to decide on order of translation



But what about...

`a := 1 + 2;`

`y := a + b;`

`x := a + b + 8;`

`z := b + a;`

`a := a + 1;`

`w := a + b;`

Overview of IR optimization

- Formalisms and Terminology
 - Control-flow graphs
 - Basic blocks
- Local optimizations
 - Speeding up small pieces of a procedure
- Global optimizations
 - Speeding up procedure as a whole
- The dataflow framework
 - Defining and implementing a wide class of optimizations

Program Analysis

- In order to optimize a program, the compiler has to be able to reason about the properties of that program
- An analysis is called **sound** if it never asserts an incorrect fact about a program
- All the analyses we will discuss in this class are sound
 - *(Why?)*

Soundness

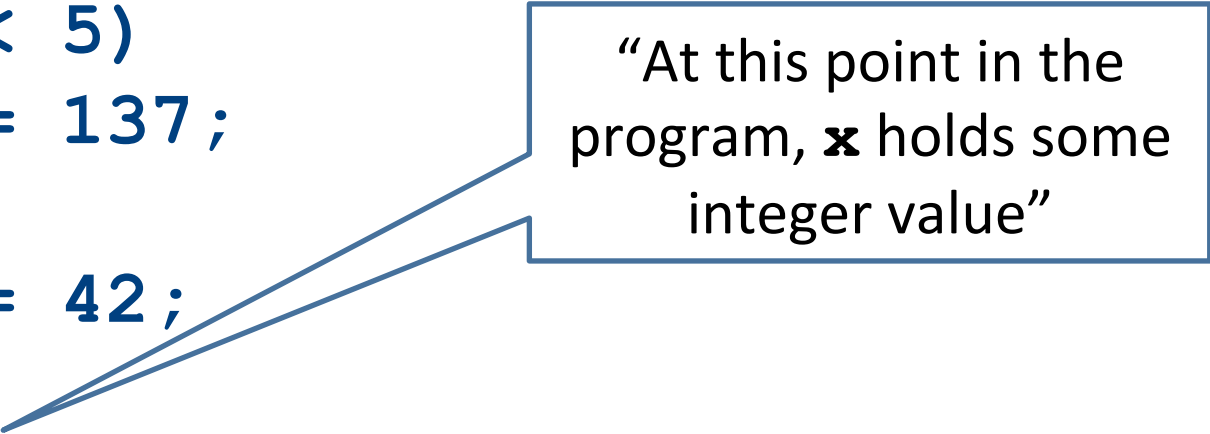
```
int x;
```

```
int y;
```

```
if (y < 5)  
    x = 137;
```

```
else  
    x = 42;
```

```
Print(x);
```



“At this point in the program, **x** holds some integer value”

Soundness

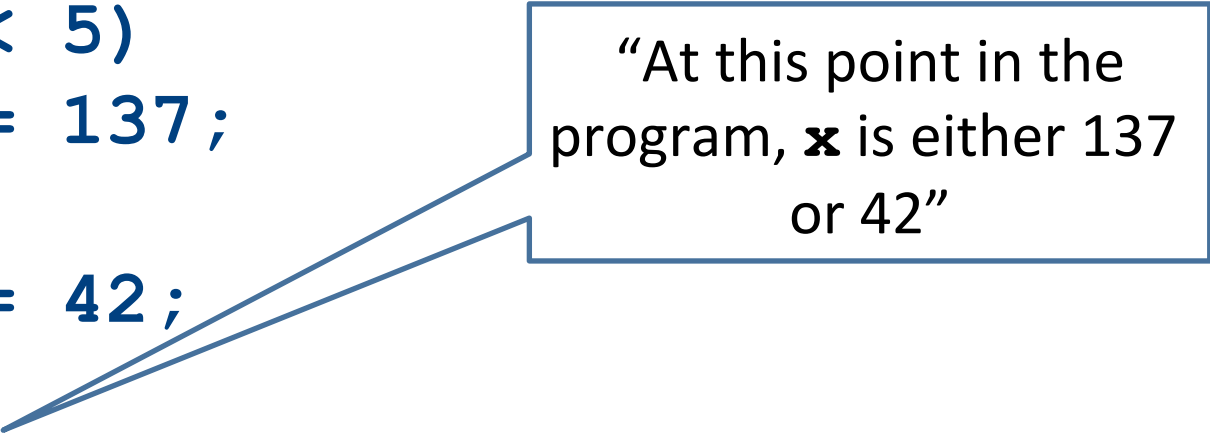
```
int x;
```

```
int y;
```

```
if (y < 5)  
    x = 137;
```

```
else  
    x = 42;
```

```
Print(x);
```



“At this point in the program, **x** is either 137 or 42”

Soundness

```
int x;
```

```
int y;
```

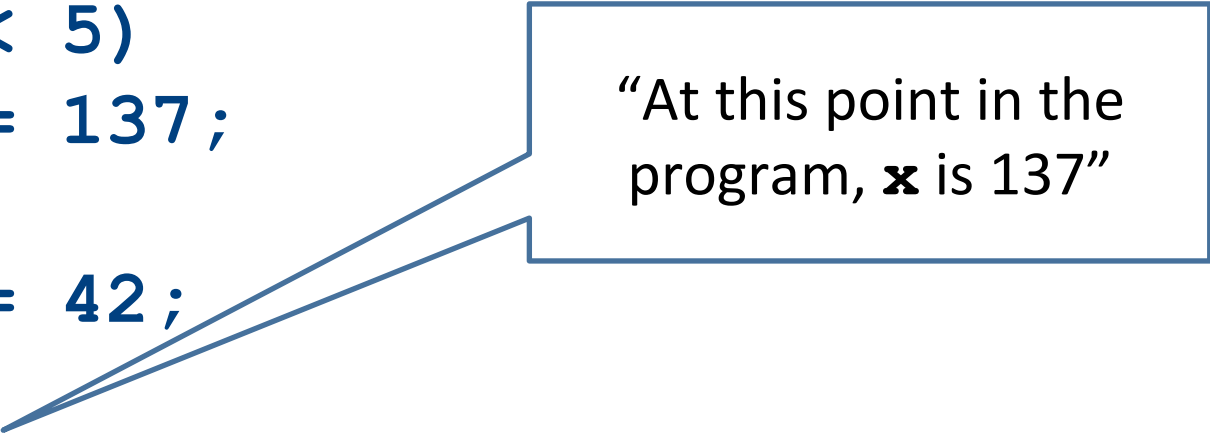
```
if (y < 5)
```

```
    x = 137;
```

```
else
```

```
    x = 42;
```

```
Print(x);
```



“At this point in the program, **x** is 137”

Soundness

```
int x;  
int y;  
  
if (y < 5)  
    x = 137;  
else  
    x = 42;  
  
Print(x);
```

“At this point in the program, **x** is either 137, 42, or 271”

Semantics-preserving optimizations

- An optimization is **semantics-preserving** if it does not alter the semantics of the original program
- Examples:
 - Eliminating unnecessary temporary variables
 - Computing values that are known statically at compile-time instead of runtime
 - Evaluating constant expressions outside of a loop instead of inside
- Non-examples:
 - Replacing bubble sort with quicksort (why?)
 - The optimizations we will consider in this class are all semantics-preserving

A formalism for IR optimization

- Every phase of the compiler uses some new abstraction:
 - Scanning uses regular expressions
 - Parsing uses CFGs
 - Semantic analysis uses proof systems and symbol tables
 - IR generation uses ASTs
- In optimization, we need a formalism that captures the structure of a program in a way amenable to optimization

Visualizing IR

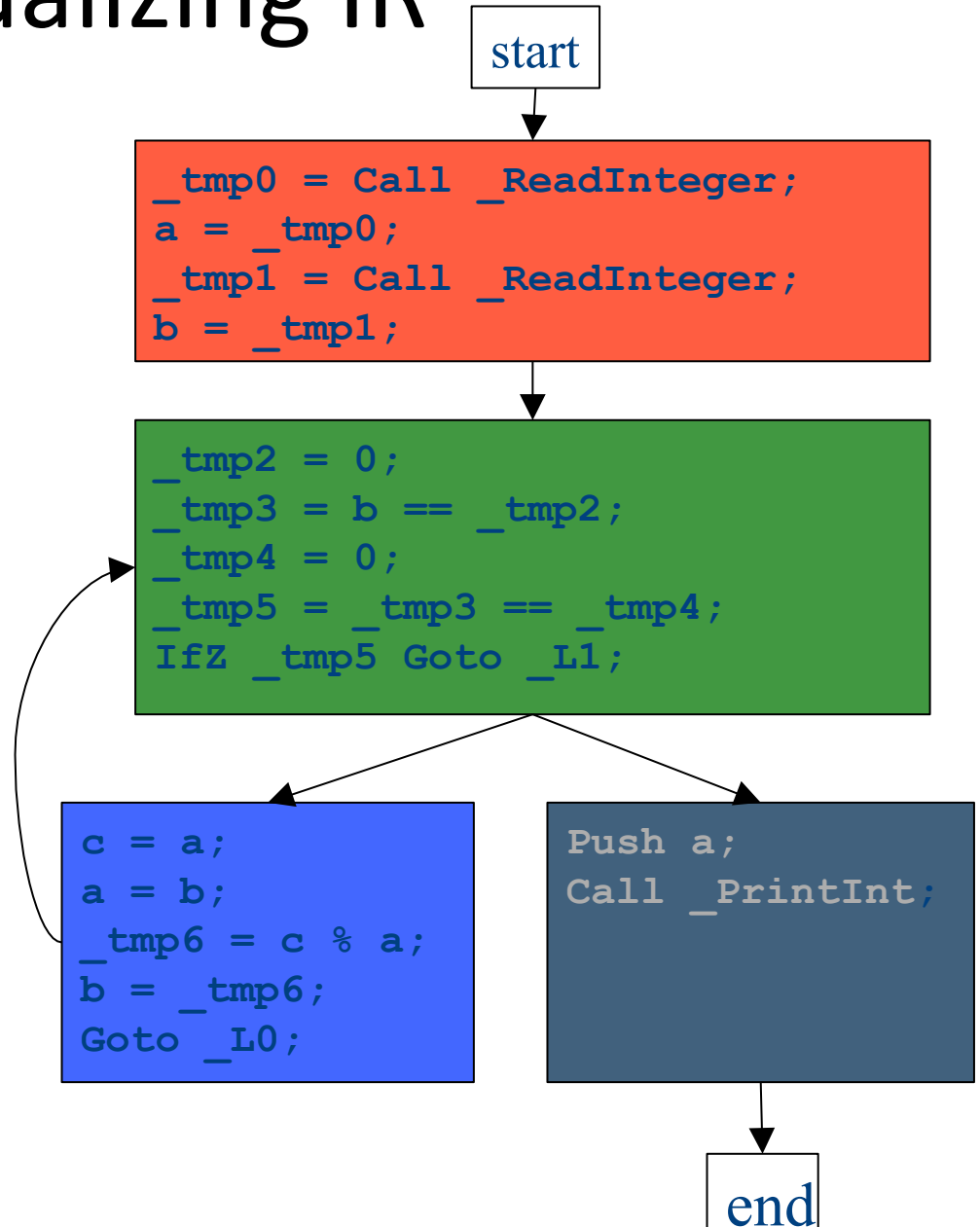
```
main:
    _tmp0 = Call _ReadInteger;
    a = _tmp0;
    _tmp1 = Call _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    Push a;
    Call _PrintInt;
```


Visualizing IR

```
main:
    _tmp0 = Call _ReadInteger;
    a = _tmp0;
    _tmp1 = Call _ReadInteger;
    b = _tmp1;
_L0:
    _tmp2 = 0;
    _tmp3 = b == _tmp2;
    _tmp4 = 0;
    _tmp5 = _tmp3 == _tmp4;
    IfZ _tmp5 Goto _L1;
    c = a;
    a = b;
    _tmp6 = c % a;
    b = _tmp6;
    Goto _L0;
_L1:
    Push a;
    Call _PrintInt;
```

Visualizing IR

```
main:  
  _tmp0 = Call _ReadInteger;  
  a = _tmp0;  
  _tmp1 = Call _ReadInteger;  
  b = _tmp1;  
_L0:  
  _tmp2 = 0;  
  _tmp3 = b == _tmp2;  
  _tmp4 = 0;  
  _tmp5 = _tmp3 == _tmp4;  
  IfZ _tmp5 Goto _L1;  
  c = a;  
  a = b;  
  _tmp6 = c % a;  
  b = _tmp6;  
  Goto _L0;  
_L1:  
  Push a;  
  Call _PrintInt;
```



Basic blocks

- A **basic block** is a sequence of IR instructions where
 - There is exactly one spot where control enters the sequence, which must be at the start of the sequence
 - There is exactly one spot where control leaves the sequence, which must be at the end of the sequence
- Informally, a sequence of instructions that always execute as a group

Control-Flow Graphs

- A **control-flow graph** (CFG) is a graph of the basic blocks in a function
- The term CFG is overloaded – from here on out, we'll mean “control-flow graph” and not “context free grammar”
- Each edge from one basic block to another indicates that control can flow from the end of the first block to the start of the second block
- There is a dedicated node for the start and end of a function

Types of optimizations

- An optimization is **local** if it works on just a single basic block
- An optimization is **global** if it works on an entire control-flow graph
- An optimization is **interprocedural** if it works across the control-flow graphs of multiple functions
 - We won't talk about this in this course

Basic blocks exercise

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```

```
START:  
    _t0 = 137;  
    y = _t0;  
    IfZ x Goto _L0;  
    t1 = y;  
    z = _t1;  
    Goto END:  
  
_L0:  
    _t2 = y;  
    x = _t2;  
  
END:
```

Divide the code into basic blocks

Control-flow graph exercise

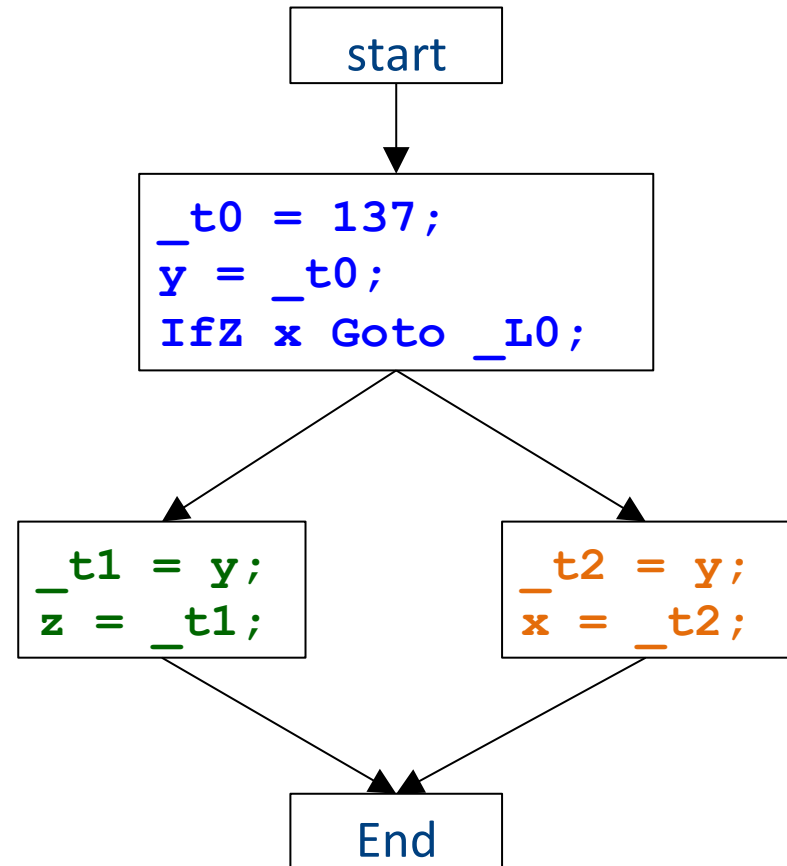
```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```

```
START:  
    _t0 = 137;  
    y = _t0;  
    IfZ x Goto _L0;  
    t1 = y;  
    z = _t1;  
    Goto END:  
  
_L0:  
    _t2 = y;  
    x = _t2;  
  
END:
```

Draw the control-flow graph

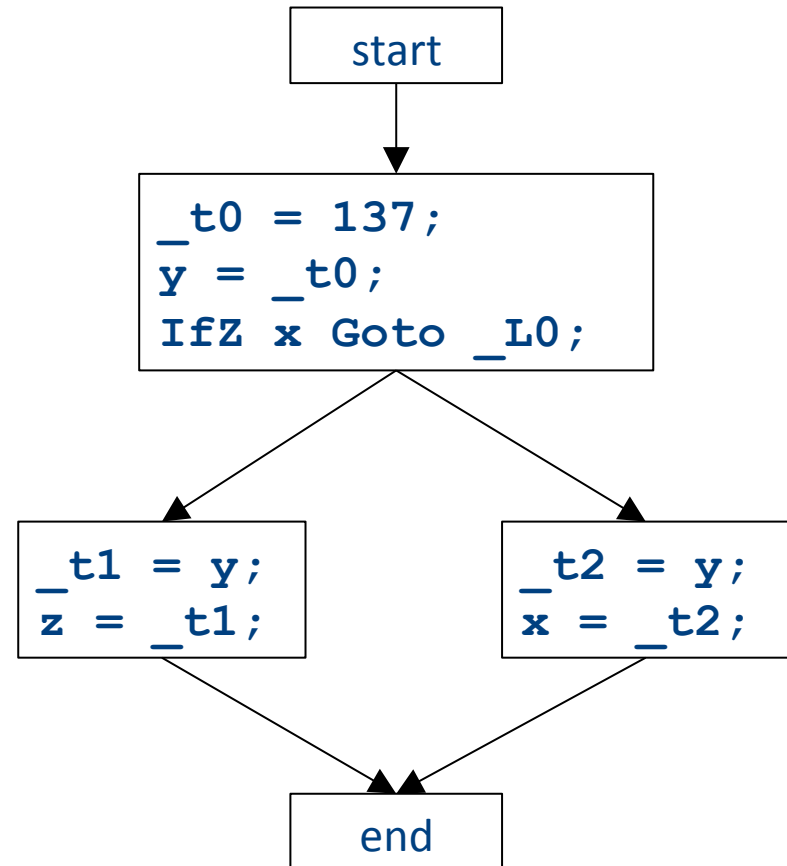
Control-flow graph exercise

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



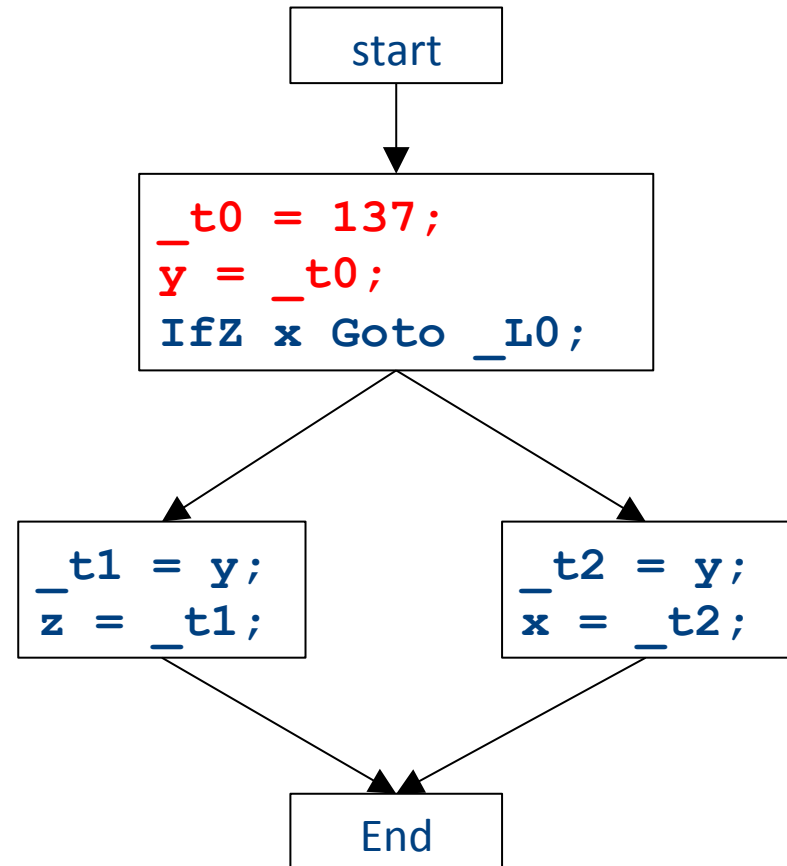
Local optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



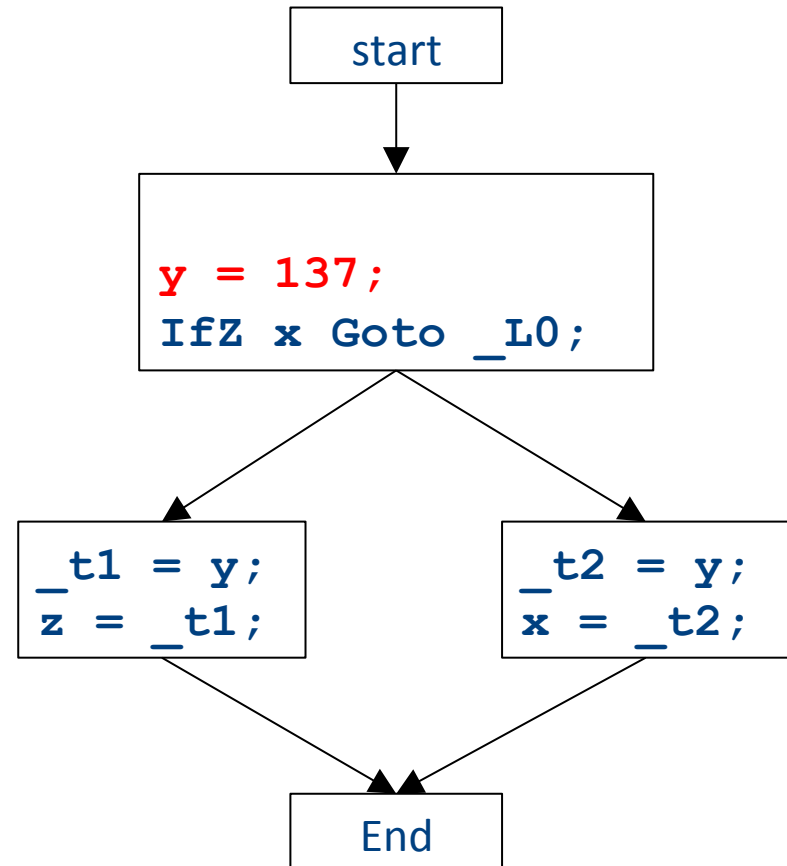
Local optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



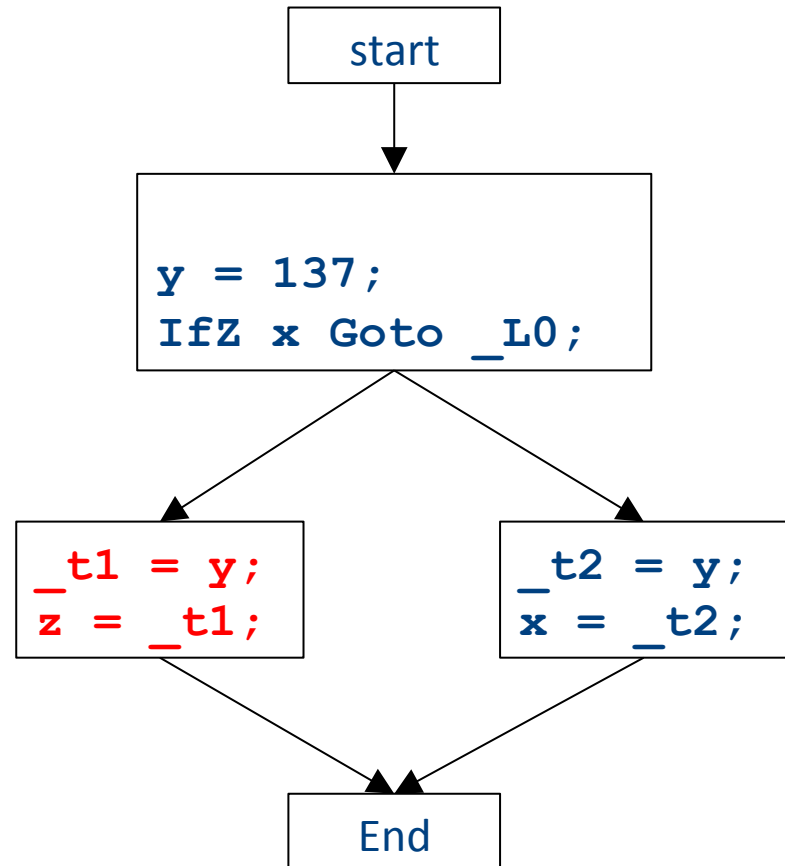
Local optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



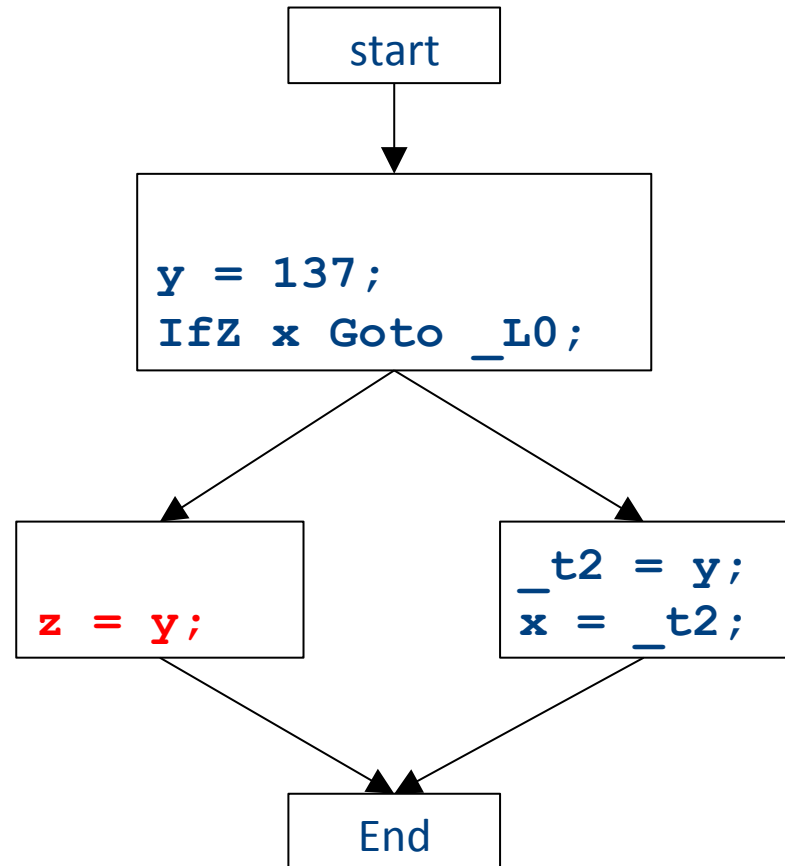
Local optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



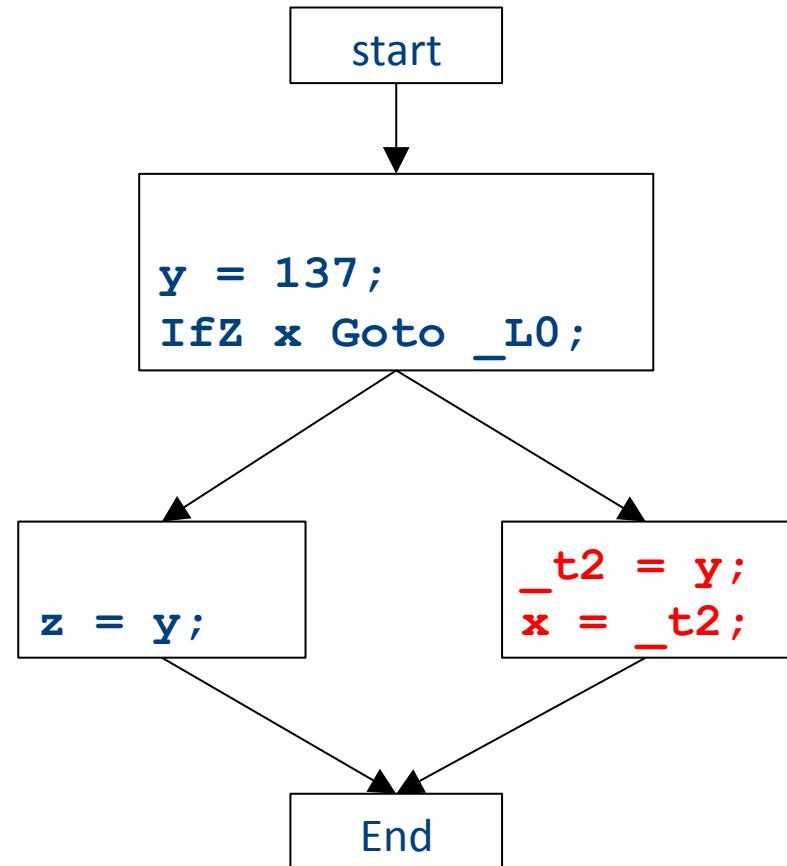
Local optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



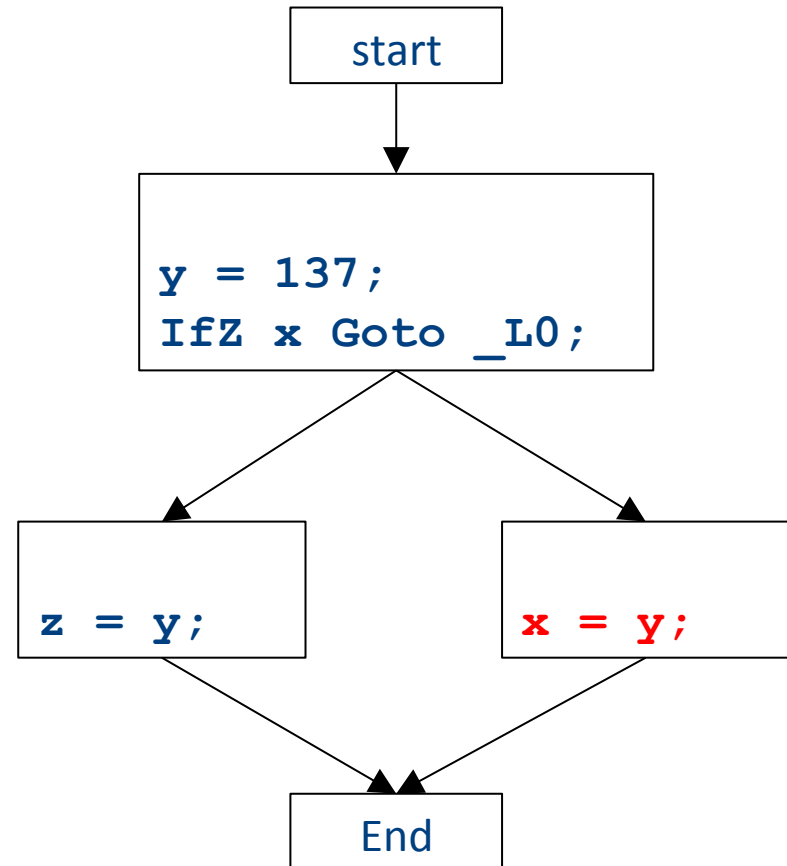
Local optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



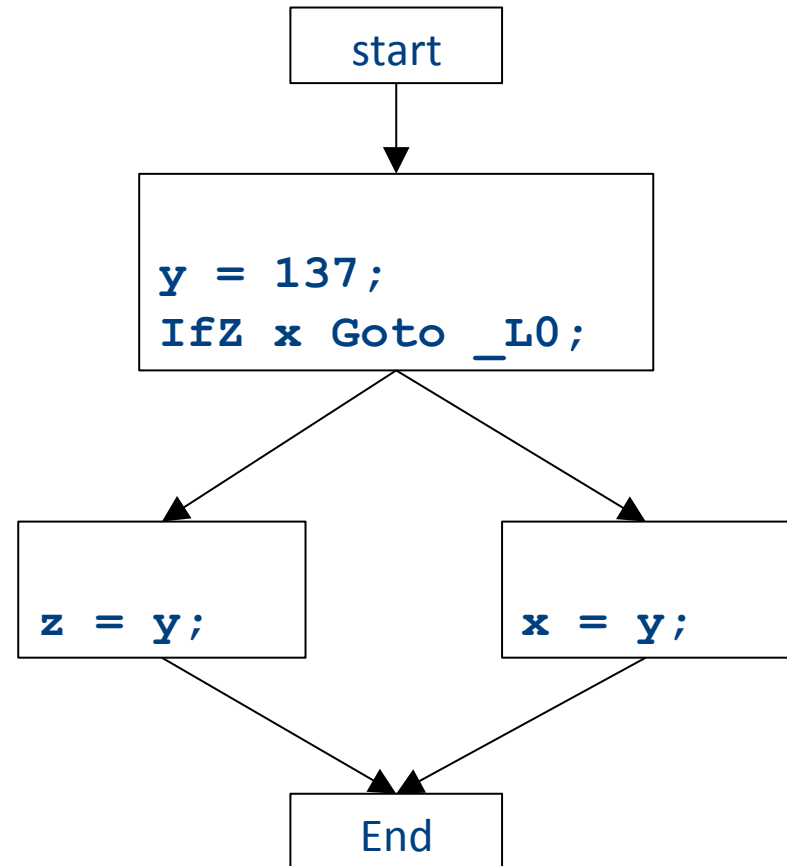
Local optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



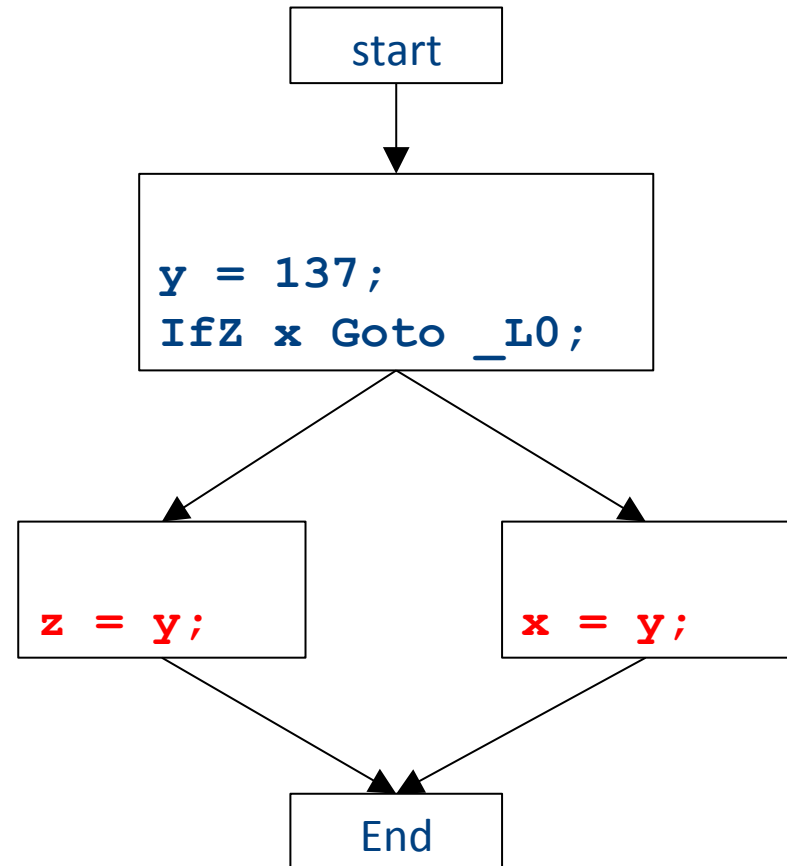
Global optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



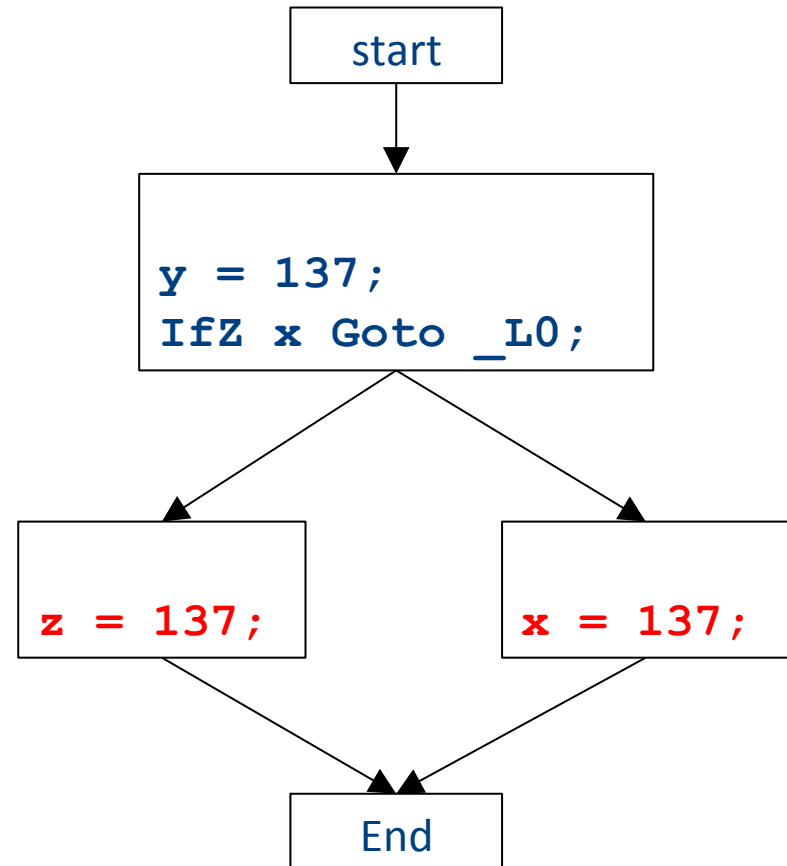
Global optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



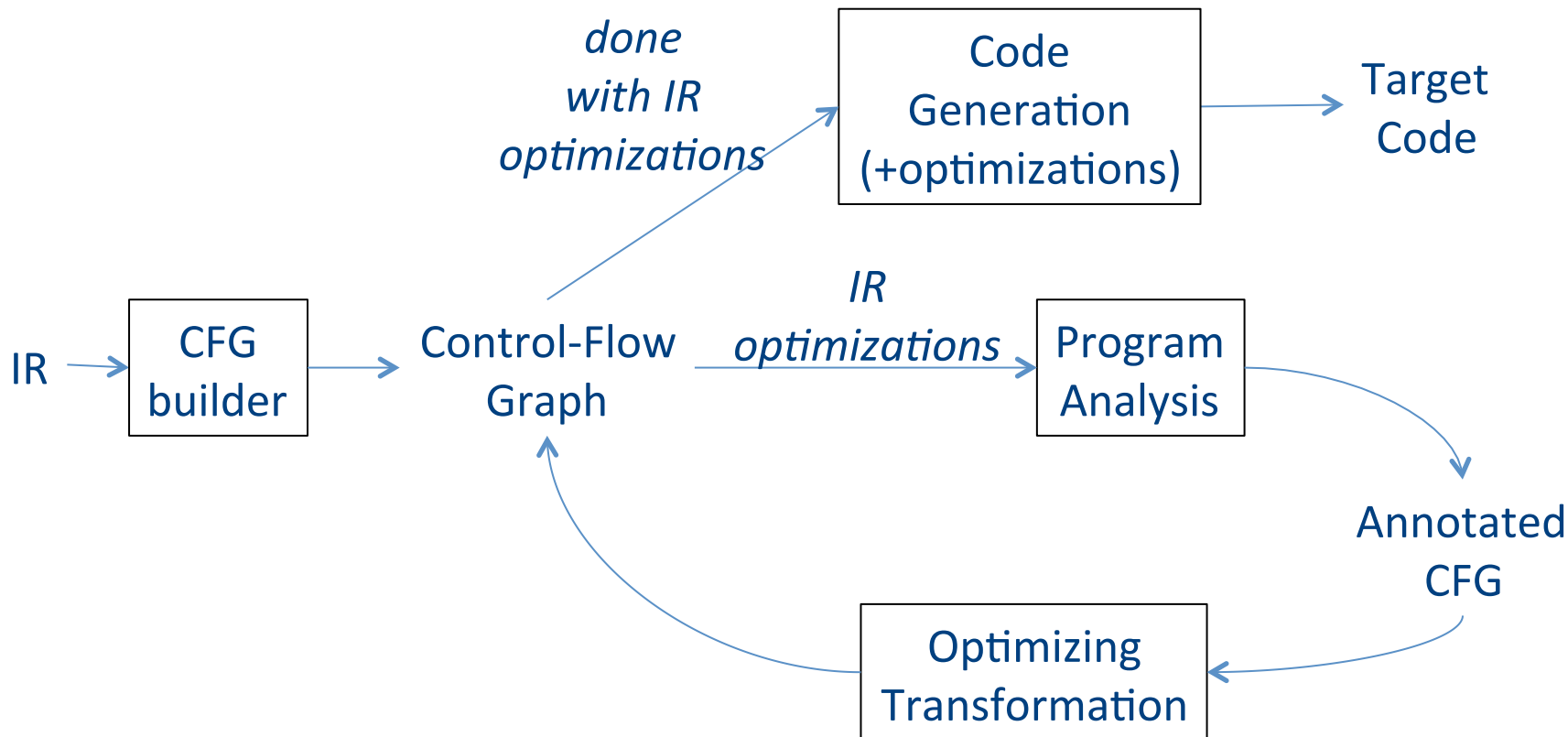
Global optimizations

```
int main() {  
    int x;  
    int y;  
    int z;  
  
    y = 137;  
    if (x == 0)  
        z = y;  
    else  
        x = y;  
}
```



Local Optimizations

Optimization path



Example

```
Object x;  
int a;  
int b;  
int c;
```

```
x = ne  
a = 4;  
c = a  
x.fn(a
```

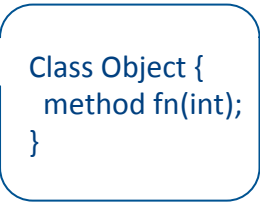
```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*( _tmp1) = _tmp2;
```

For brevity:
Simplified IR for procedure returns

```
+ b;  
c = _tmp4;  
_tmp5 = a + b;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Example

```
Object x;  
int a;  
int b;  
int c;
```



```
Class Object {  
    method fn(int);  
}
```

```
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = 4;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = a + b;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Example

```
Object x;  
int a;  
int b;  
int c;
```

Class Object {
 method fn(int);
}

```
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

For simplicity, ignore
Popping return value,
parameters etc.

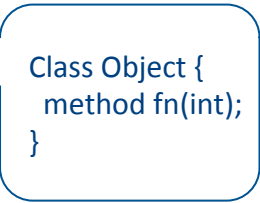
```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = 4;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = a + b;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Size of Object

Object Class

Example

```
Object x;  
int a;  
int b;  
int c;
```



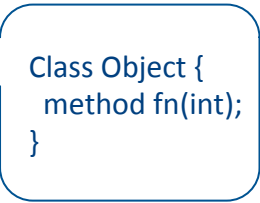
```
Class Object {  
  method fn(int);  
}
```

```
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = 4;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = a + b;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```


Example

```
Object x;  
int a;  
int b;  
int c;
```



```
Class Object {  
  method fn(int);  
}
```

```
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = 4;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = a + b;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Example

```
Object x;  
int a;  
int b;  
int c;
```

Class Object {
 method fn(int);
}

```
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

Points to ObjectC

Start of fn

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = 4;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = a + b;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Common Subexpression Elimination

- If we have two variable assignments
 $v1 = a \text{ op } b$
...
 $v2 = a \text{ op } b$
- and the values of $v1$, a , and b have not changed between the assignments, rewrite the code as
 $v1 = a \text{ op } b$
...
 $v2 = v1$
- Eliminates useless recalculation
- Paves the way for later optimizations

Common Subexpression Elimination

- If we have two variable assignments
v1 = a op b [or: v1 = a]
...
v2 = a op b [or: v2 = a]
- and the values of v1, a, and b have not changed between the assignments, rewrite the code as
v1 = a op b [or: v1 = a]
...
v2 = v1
- Eliminates useless recalculation
- Paves the way for later optimizations

Common subexpression elimination

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = 4;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = a + b;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Common subexpression elimination

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = 4;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = _tmp4;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Common subexpression elimination

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = 4;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = _tmp4;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Common subexpression elimination

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = _tmp4;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```


Common subexpression elimination

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = _tmp4;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Common subexpression elimination

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Copy Propagation

- If we have a variable assignment
 $v1 = v2$
then as long as $v1$ and $v2$ are not
reassigned, we can rewrite expressions of
the form
 $a = \dots v1 \dots$
as
 $a = \dots v2 \dots$
provided that such a rewrite is legal

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = _tmp2;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(x);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push x;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(_tmp1);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push _tmp1;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = a + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(_tmp1);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push _tmp1;  
Call _tmp7;
```


Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = _tmp3 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(_tmp1);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push _tmp1;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = _tmp3 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(_tmp1);  
_tmp7 = *(_tmp6);  
Push _tmp5;  
Push _tmp1;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = _tmp3 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(_tmp1);  
_tmp7 = *(_tmp6);  
Push c;  
Push _tmp1;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = _tmp3 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = *(_tmp1);  
_tmp7 = *(_tmp6);  
Push c;  
Push _tmp1;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = _tmp3 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = ObjectC;  
_tmp7 = *(_tmp6);  
Push c;  
Push _tmp1;  
Call _tmp7;
```

Is this transformation OK?
What do we need to know?

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = _tmp3 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = ObjectC;  
_tmp7 = *(_tmp6);  
Push c;  
Push _tmp1;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = _tmp3 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = ObjectC;  
_tmp7 = *(ObjectC);  
Push c;  
Push _tmp1;  
Call _tmp7;
```

Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp3;  
_tmp4 = _tmp3 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = ObjectC;  
_tmp7 = *(ObjectC);  
Push c;  
Push _tmp1;  
Call _tmp7;
```


Copy Propagation

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp0;  
_tmp4 = _tmp0 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = ObjectC;  
_tmp7 = *(ObjectC);  
Push c;  
Push _tmp1;  
Call _tmp7;
```

Dead Code Elimination

- An assignment to a variable v is called **dead** if the value of that assignment is never read anywhere
- **Dead code elimination** removes dead assignments from IR
- Determining whether an assignment is dead depends on what variable is being assigned to and when it's being assigned

Dead Code Elimination

```
Object x;  
int a;  
int b;  
int c;  
  
x = new Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp0;  
_tmp4 = _tmp0 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = ObjectC;  
_tmp7 = *(ObjectC);  
Push c;  
Push _tmp1;  
Call _tmp7;
```

Dead Code Elimination

```
Object x;  
int a;  
int b;  
int c;  
  
x = new  
Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

values
never
read

values
never
read

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
_tmp2 = ObjectC;  
*(_tmp1) = ObjectC;  
x = _tmp1;  
_tmp3 = _tmp0;  
a = _tmp0;  
_tmp4 = _tmp0 + b;  
c = _tmp4;  
_tmp5 = c;  
_tmp6 = ObjectC;  
_tmp7 = *(ObjectC);  
Push c;  
Push _tmp1;  
Call _tmp7;
```

Dead Code Elimination

```
Object x;  
int a;  
int b;  
int c;  
  
x = new  
Object;  
a = 4;  
c = a + b;  
x.fn(a + b);
```

```
_tmp0 = 4;  
Push _tmp0;  
_tmp1 = Call _Alloc;  
  
*(_tmp1) = ObjectC;  
  
_tmp4 = _tmp0 + b;  
c = _tmp4;  
  
_tmp7 = *(ObjectC);  
Push c;  
Push _tmp1;  
Call _tmp7;
```

Applying local optimizations

- The different optimizations we've seen so far all take care of just a small piece of the optimization
- Common subexpression elimination eliminates unnecessary statements
- Copy propagation helps identify dead code
- Dead code elimination removes statements that are no longer needed
- To get maximum effect, we may have to apply these optimizations numerous times

Applying local optimizations example

```
b = a * a;  
c = a * a;  
d = b + c;  
e = b + b;
```

Applying local optimizations example

```
b = a * a;  
c = a * a;  
d = b + c;  
e = b + b;
```

Which optimization should we apply here?

Applying local optimizations example

```
b = a * a;  
c = b;  
d = b + c;  
e = b + b;
```

Which optimization should we apply here?

Common sub-expression elimination

Applying local optimizations example

```
b = a * a;  
c = b;  
d = b + c;  
e = b + b;
```

Which optimization should we apply here?

Applying local optimizations example

```
b = a * a;  
c = b;  
d = b + b;  
e = b + b;
```

Which optimization should we apply here?

Copy propagation

Applying local optimizations example

```
b = a * a;  
c = b;  
d = b + b;  
e = b + b;
```

Which optimization should we apply here?

Applying local optimizations example

```
b = a * a;  
c = b;  
d = b + b;  
e = d;
```

Which optimization should we apply here?

Common sub-expression elimination (again)

Other types of local optimizations

- Arithmetic Simplification

- Replace “hard” operations with easier ones

- e.g. rewrite $\mathbf{x} = 4 * \mathbf{a};$ as $\mathbf{x} = \mathbf{a} \ll 2;$

- Constant Folding

- Evaluate expressions at compile-time if they have a constant value.

- e.g. rewrite $\mathbf{x} = 4 * 5;$ as $\mathbf{x} = 20;$

Optimizations and analyses

- Most optimizations are only possible given some analysis of the program's behavior
- In order to implement an optimization, we will talk about the corresponding program analyses

