Compilation

0368-3133 2014/15a

Lecture 7

Activation Records

Noam Rinetzky

Code generation
for procedure calls
(+ a few words on the runtime system)

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

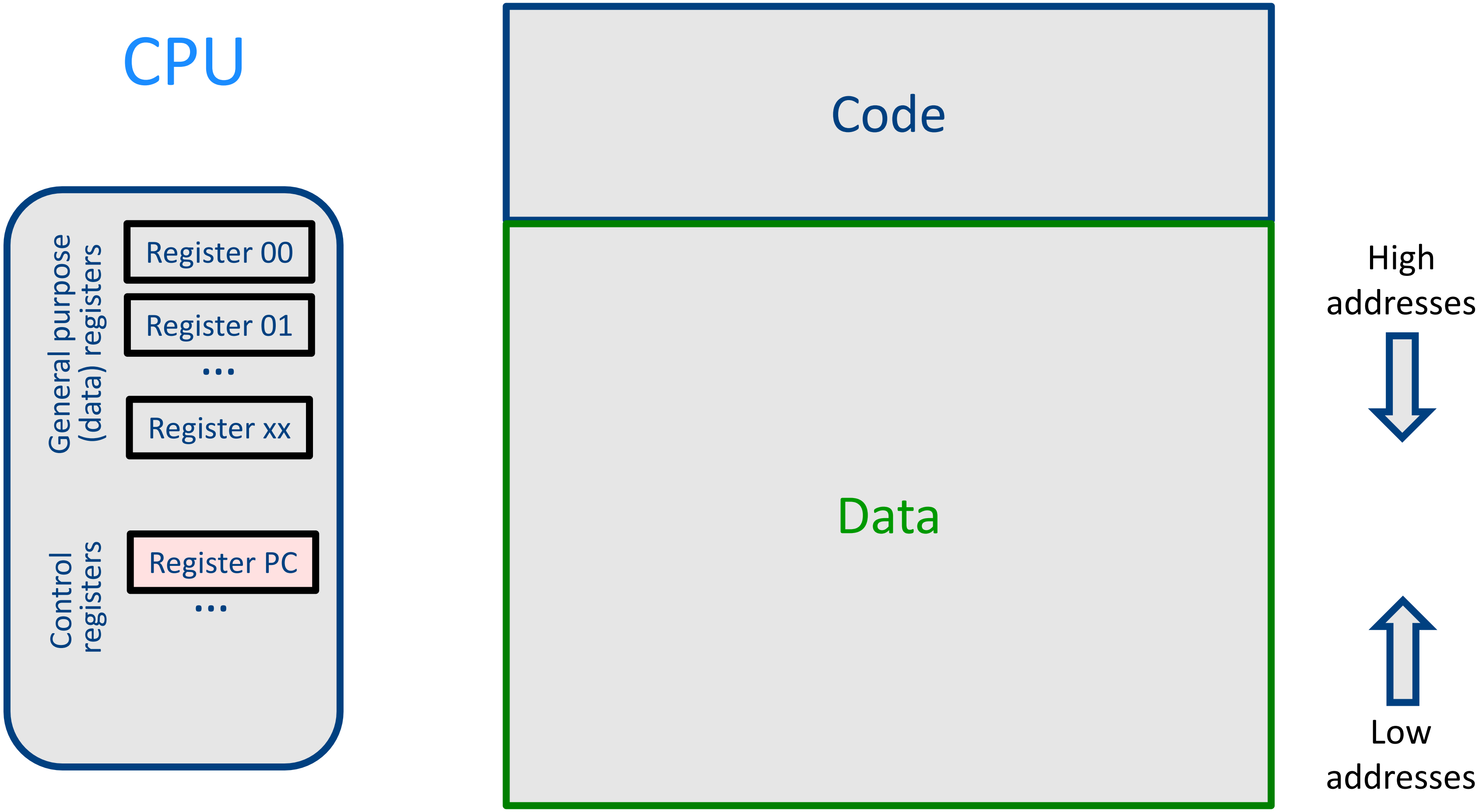
Calling Conventions

- In general, compiler can use any convention to handle procedures

- In practice, CPUs specify standards
        - Aka calling conventios
    - Allows for compiler interoperability
        - Libraries!

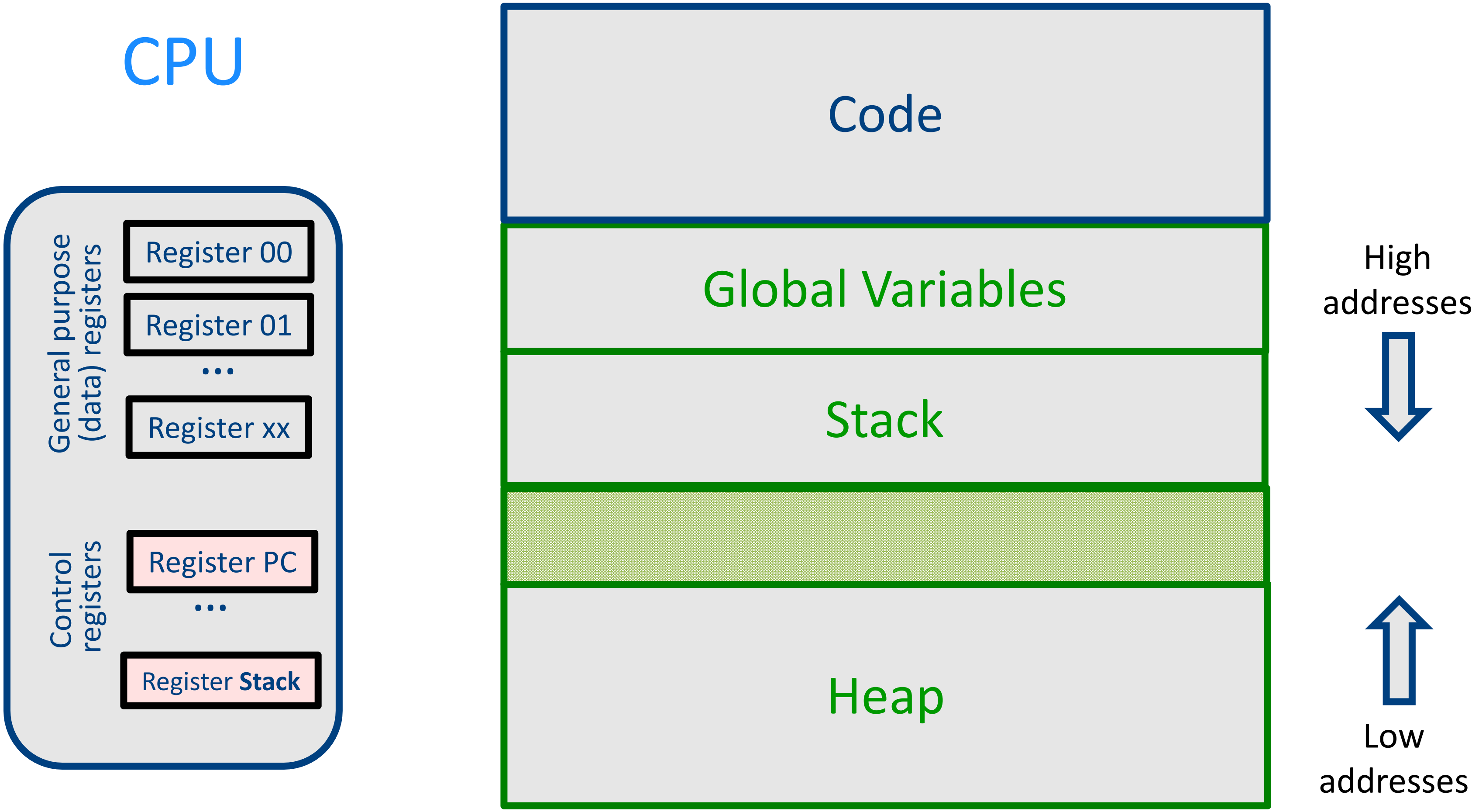
Abstract Register Machine

(High Level View)

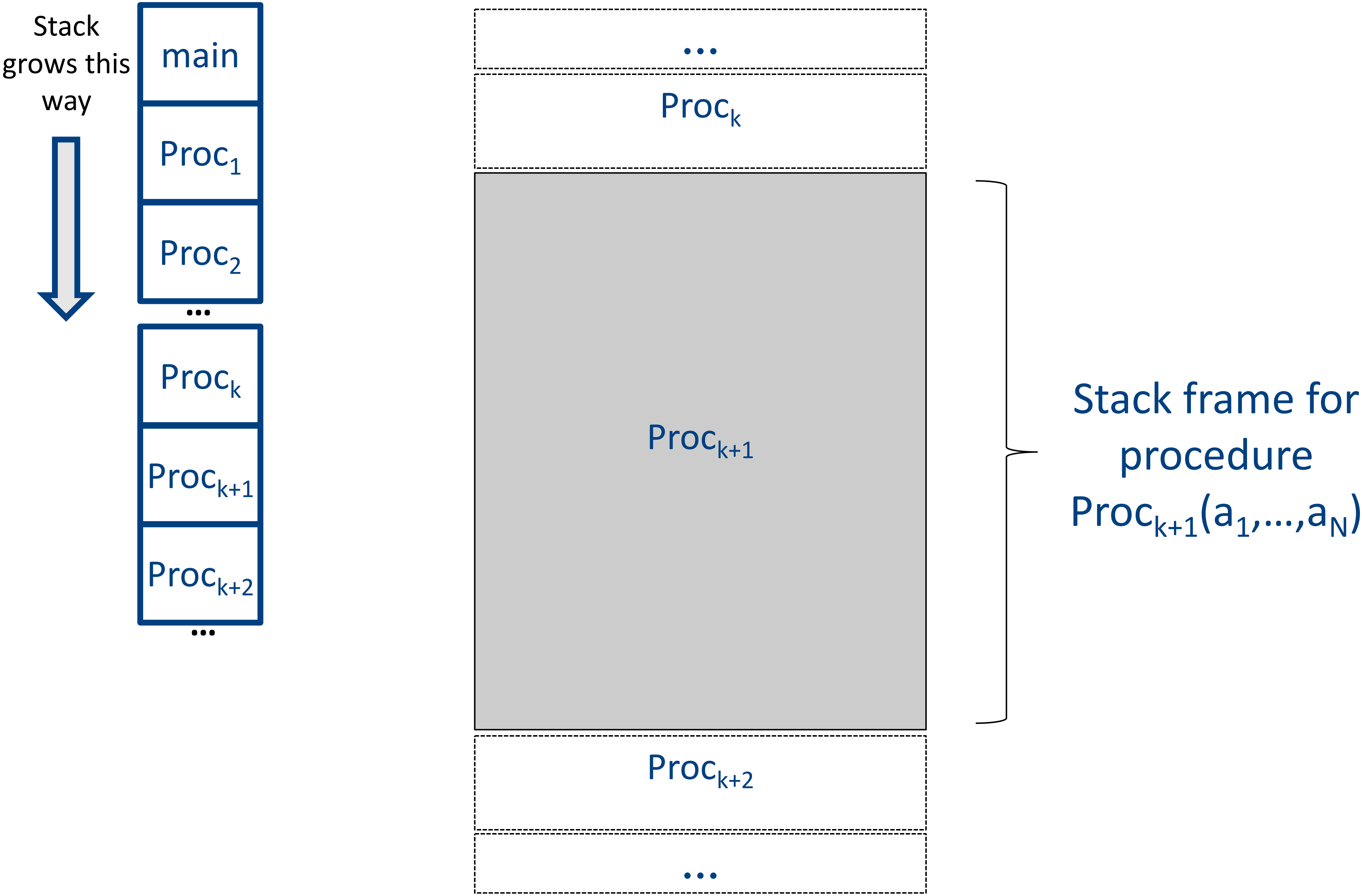


Abstract Register Machine

(High Level View)



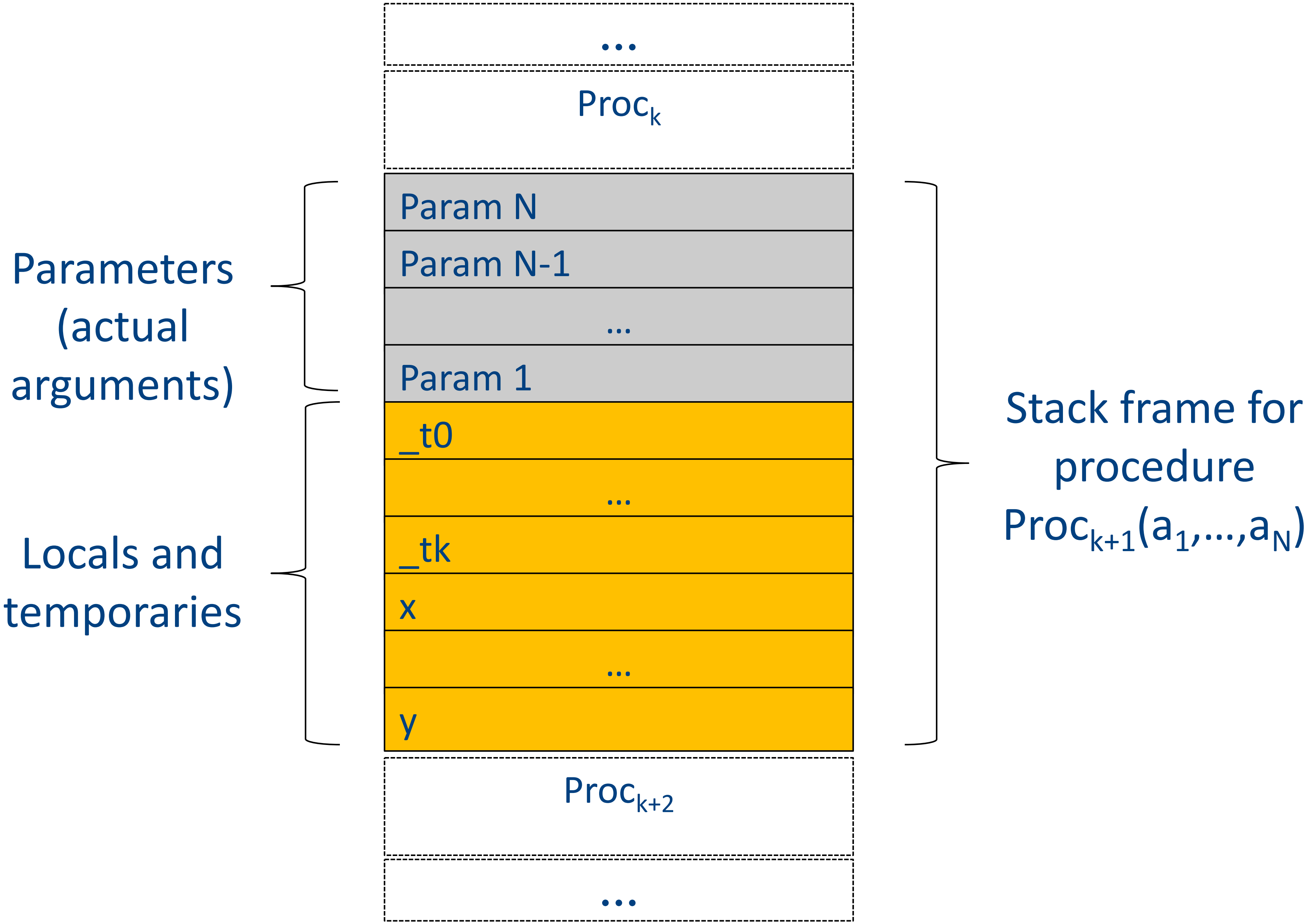
Abstract Activation Record Stack



Stack grows this way

Stack frame for procedure Proc$_{k+1}$(a$_1$,…,a$_N$)

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(a1,…,an);` looks like

  ```
  Push a1; … Push an;
  Call f;
  Pop x; // copy returned value
  ```

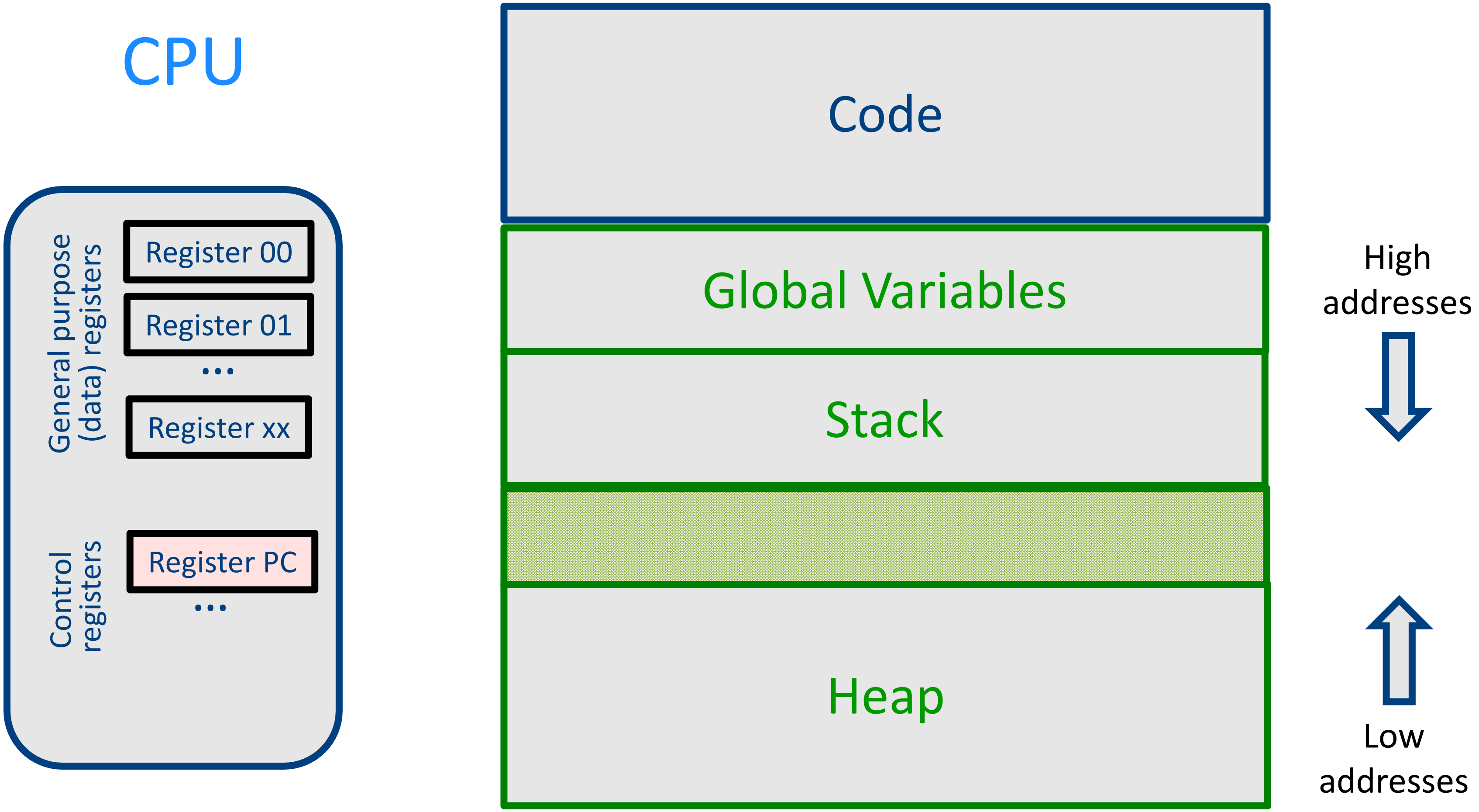
- Returning a value is done by pushing it to the stack (`return x;`)

  ```
  Push x;
  ```

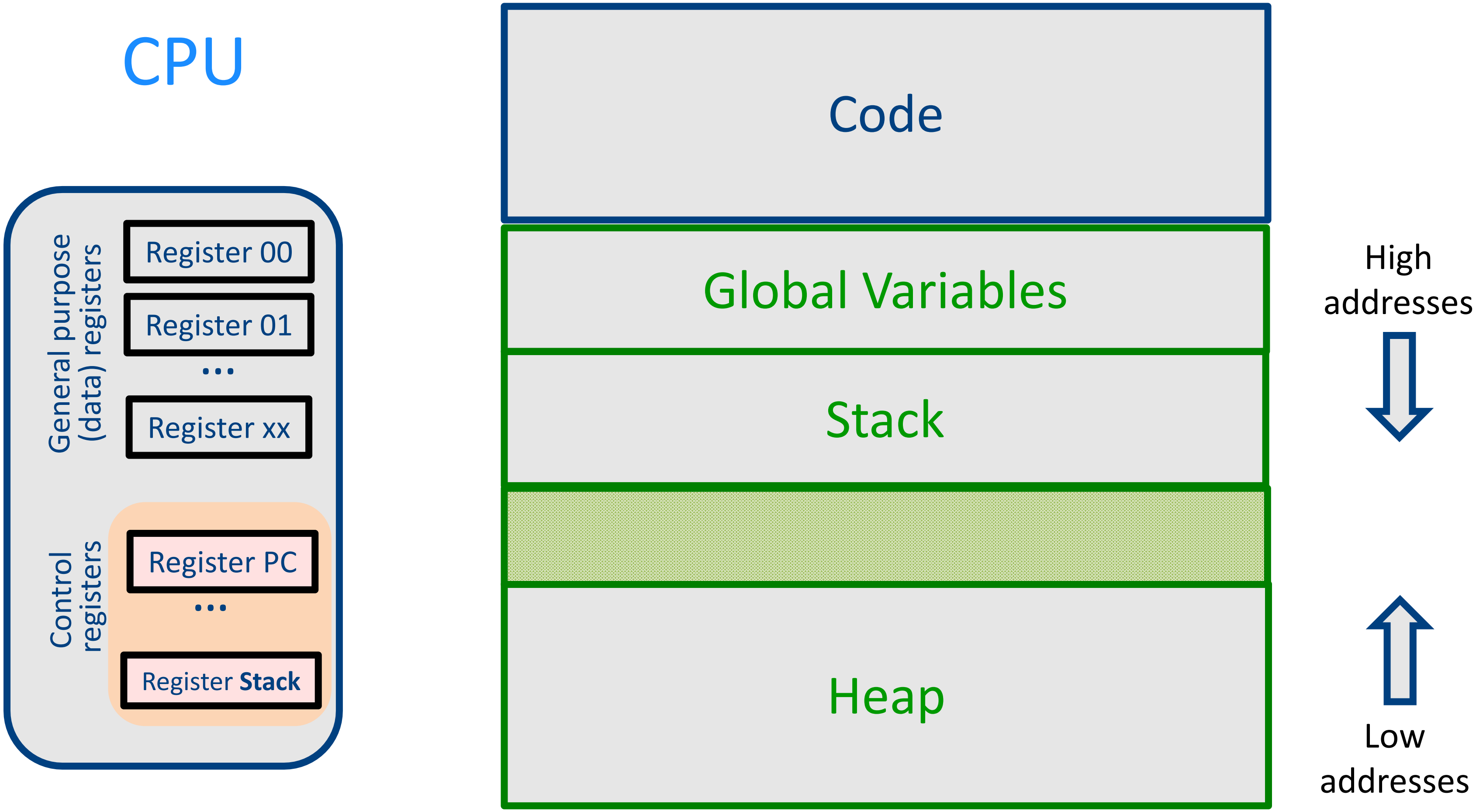
- Return control to caller (and roll up stack)

  ```
  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;
```

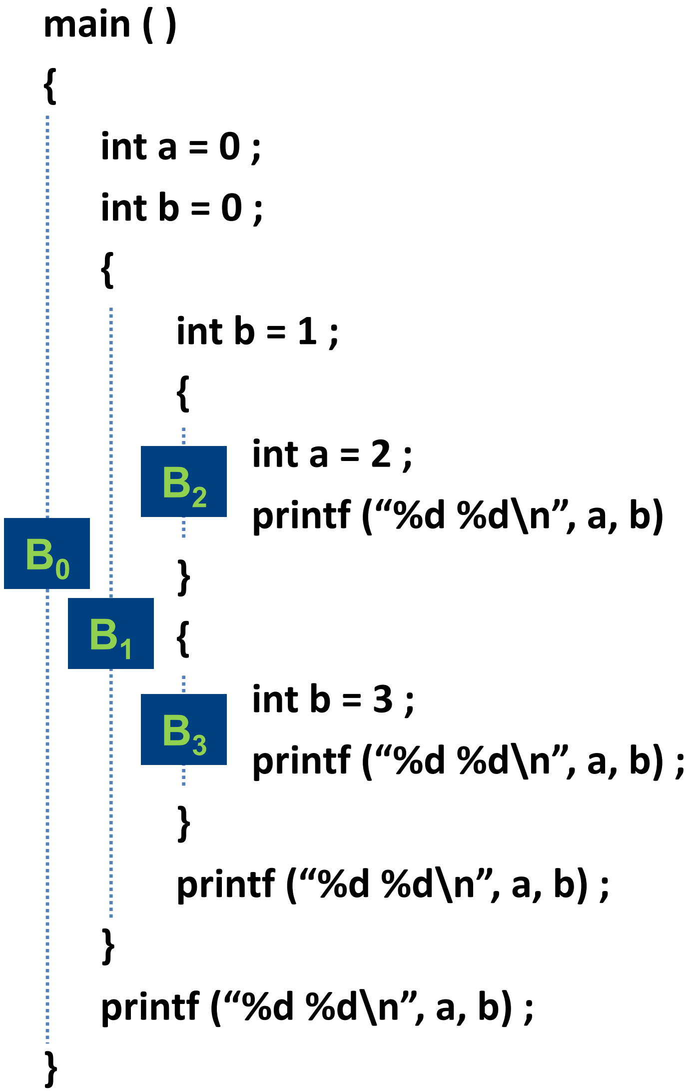
What Can We Do with Procedures?

- 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 ;
        {
            int a = 2 ;
            printf (“%d %d\n”, a, b)
        }
        {
            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?

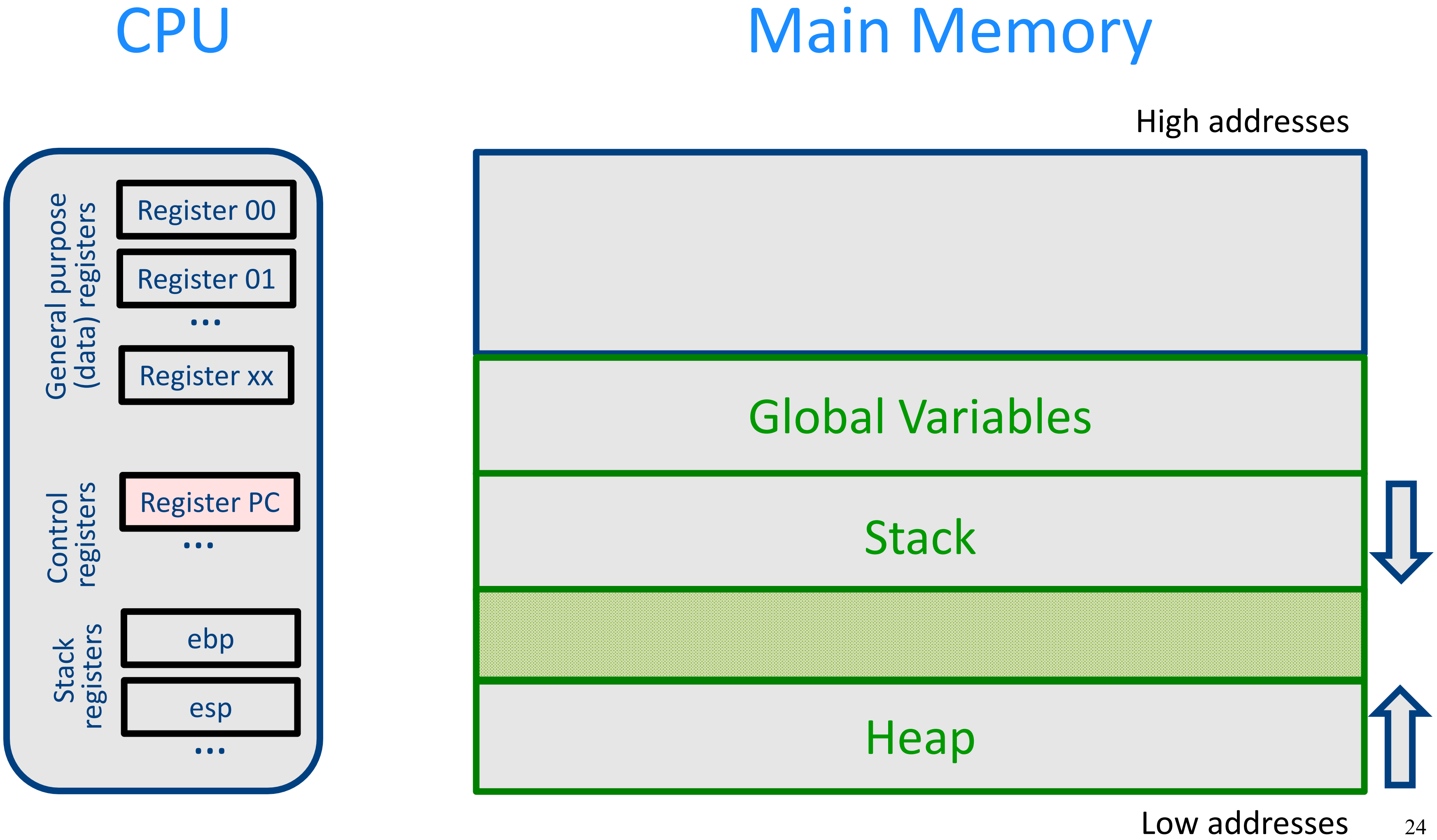
Compile-Time Information on Variables

- 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

General purpose (data) registers

Register 00

Register 01

…

Register xx

Control registers

Register PC

…

Stack registers

ebp

esp

…

Main Memory

High addresses

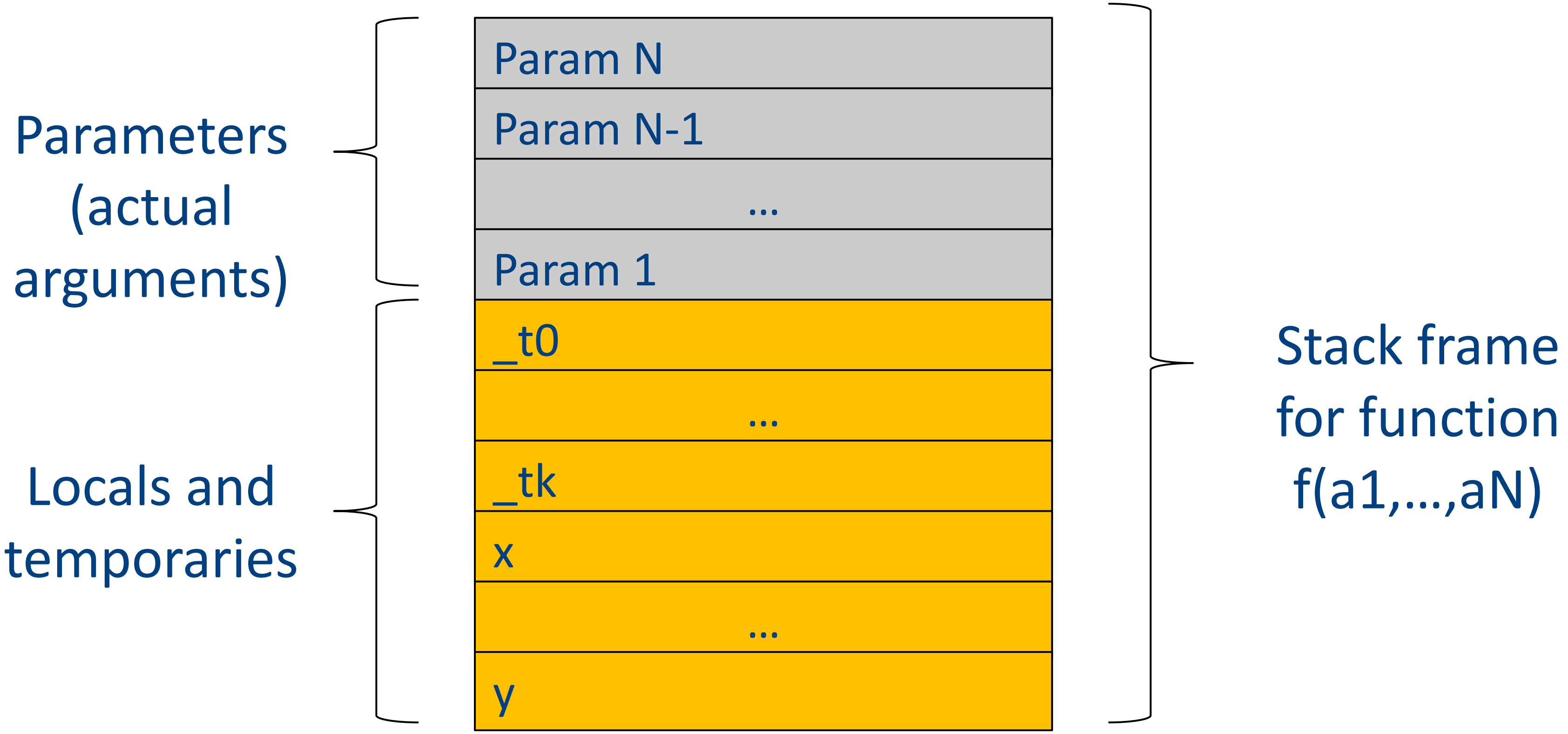
Global Variables

Stack

Heap

Low addresses

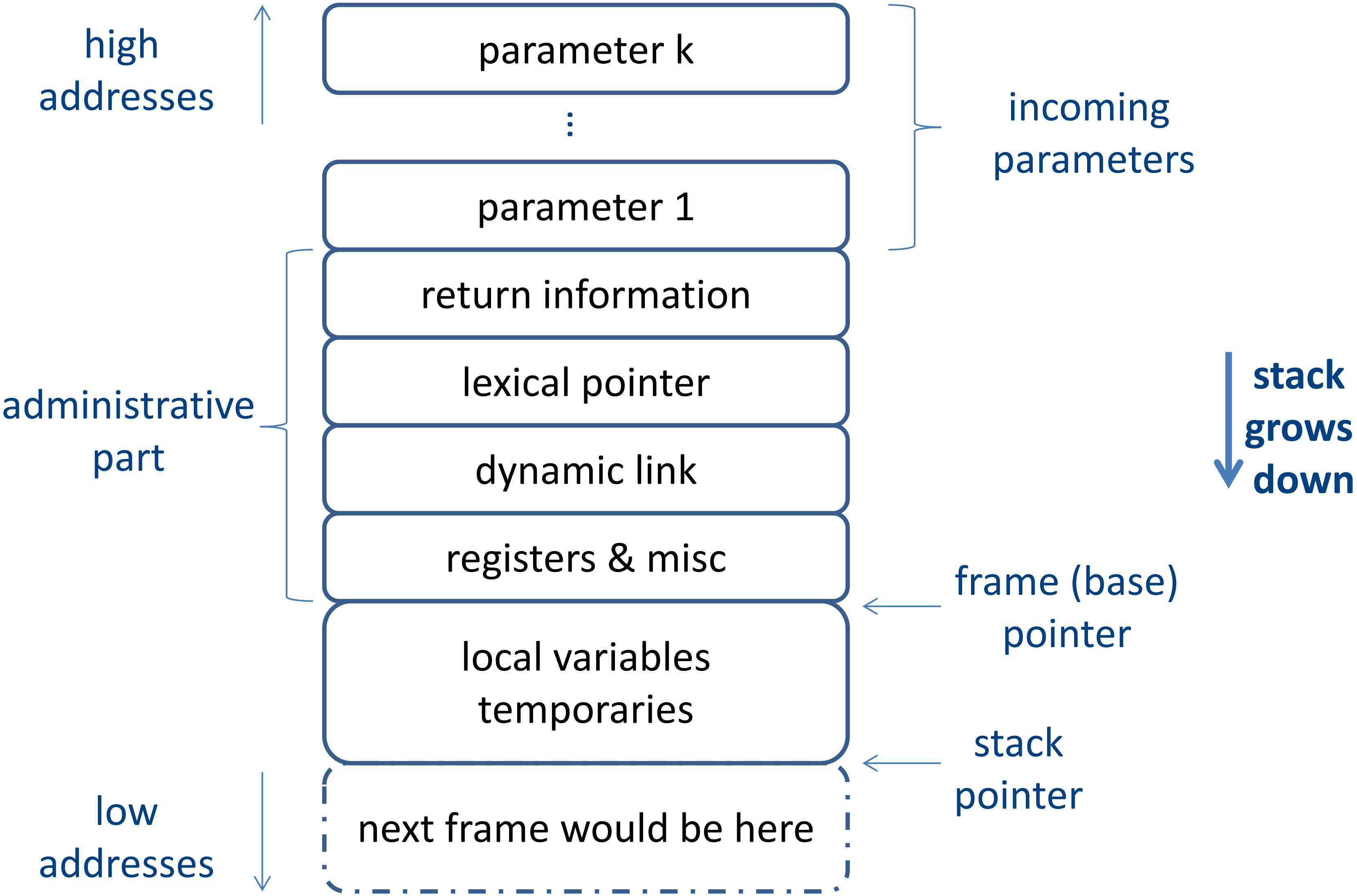
A Logical Stack Frame (Simplified)



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)



high
addresses

parameter k

⋮

parameter 1

incoming
parameters

administrative
part

return information

lexical pointer

dynamic link

registers & misc

**stack
grows
down**

frame (base)
pointer

local variables
temporaries

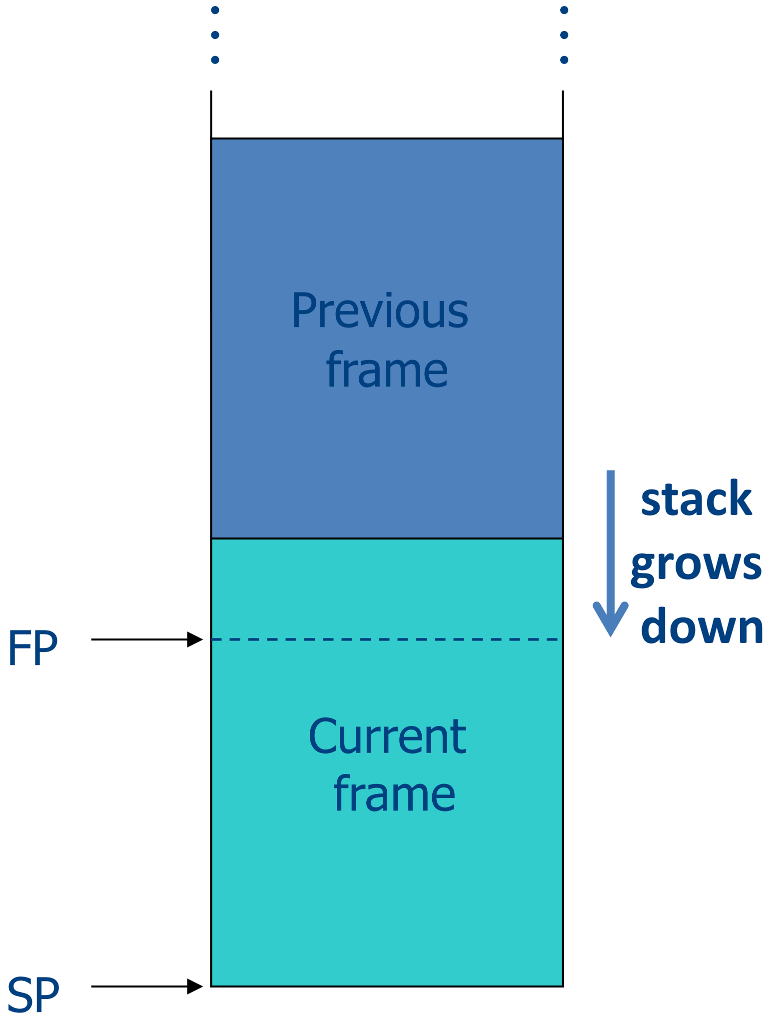
stack
pointer

low
addresses

next frame would be here

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;
 }
```

adminstrative
x1
y1
x2
x3
…

L-Values of Local Variables

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

Pentium Runtime Stack

Register	Usage
ESP	Stack pointer
EBP	Base pointer

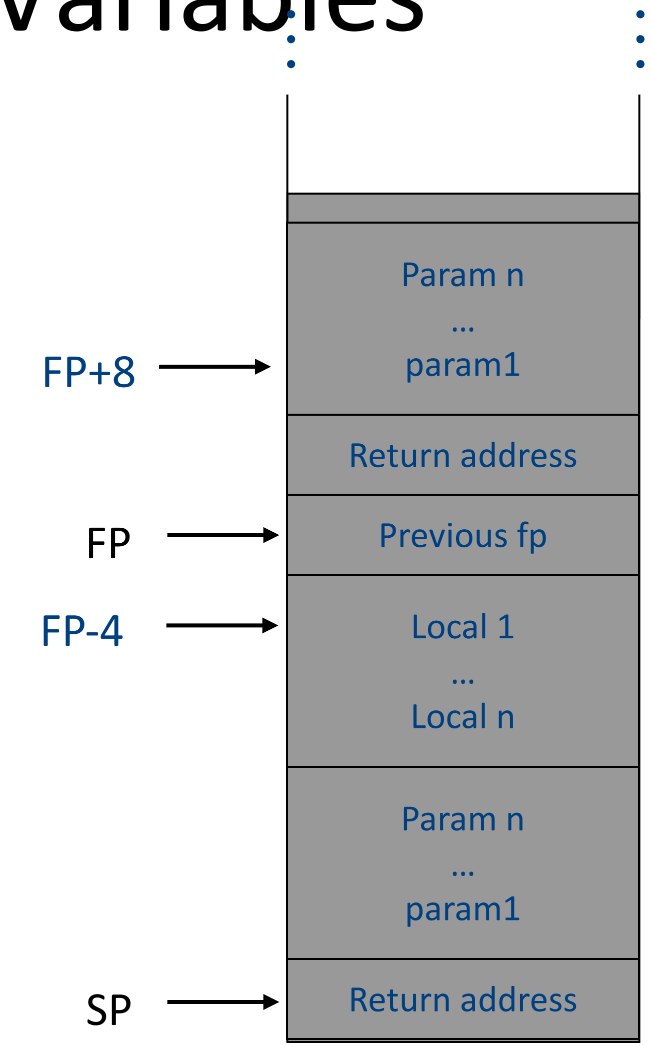
Pentium stack registers

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

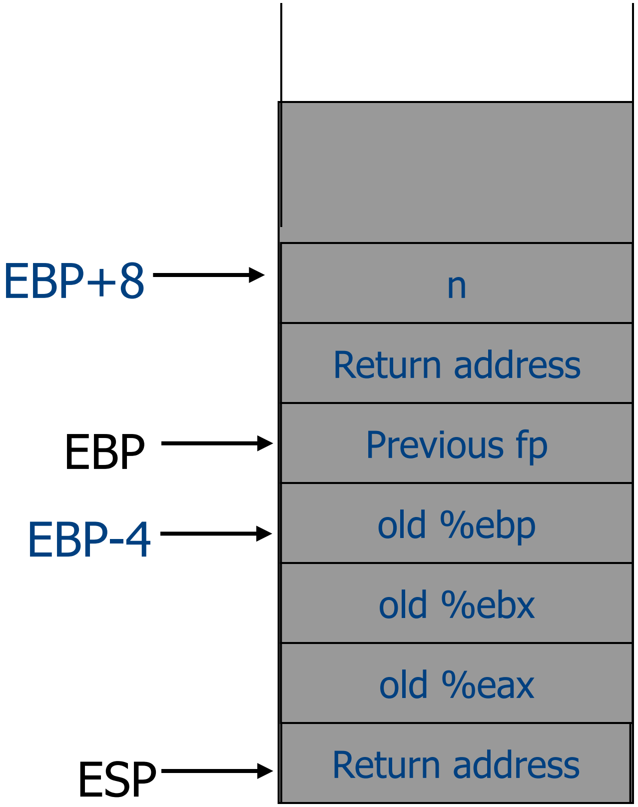
Accessing Stack Variables

- Use offset from FP (%ebp)
  - Remember: stack grows downwards
- Above FP = parameters
- Below FP = locals
- Examples
  - %ebp + 4 = return address
  - %ebp + 8 = first parameter
  - %ebp – 4 = 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)

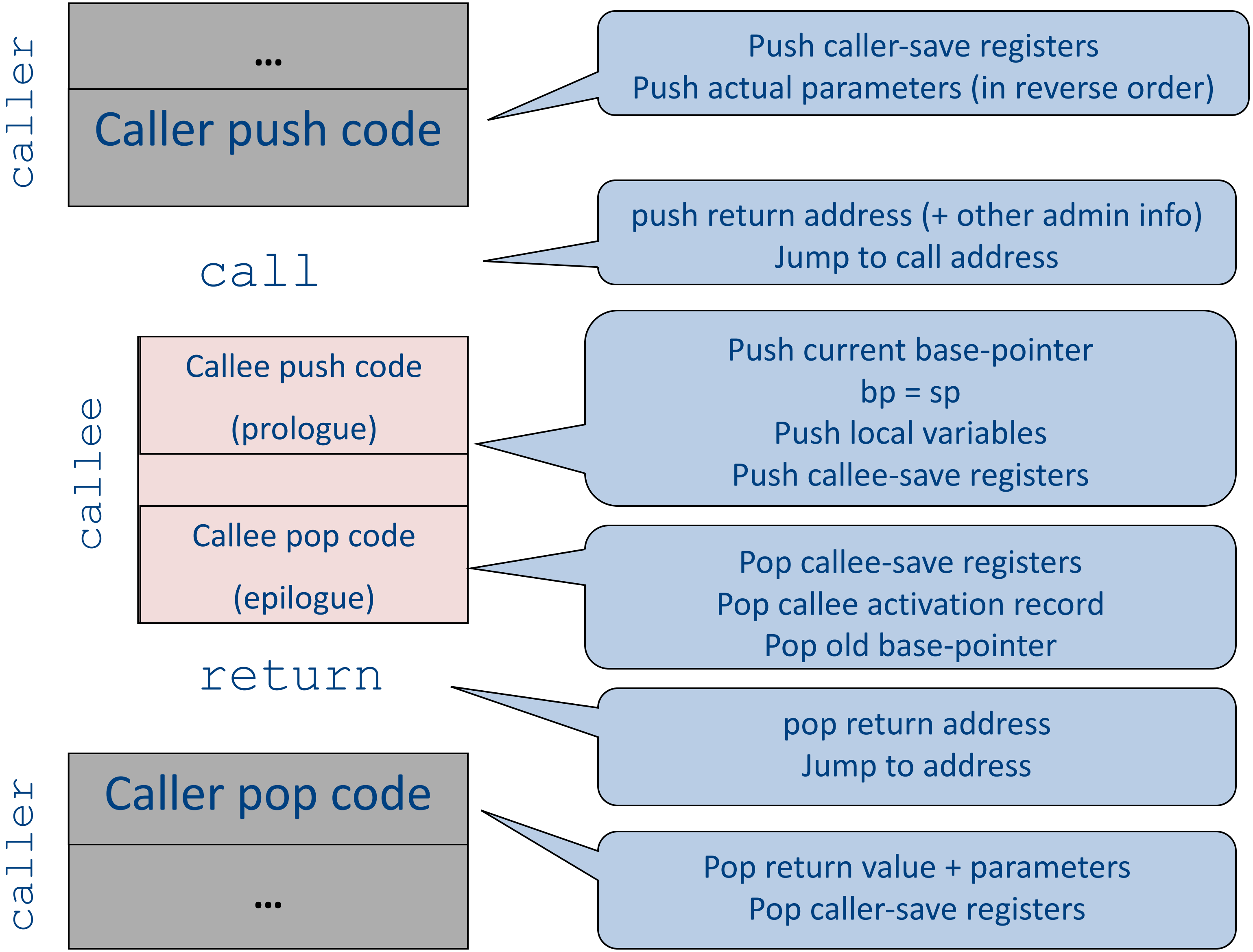
(disclaimer: real compiler can do better than that)

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



caller

…

Caller push code

Push caller-save registers
Push actual parameters (in reverse order)

`call`

push return address (+ other admin info)
Jump to call address

callee

Callee push code

(prologue)

Push current base-pointer
bp = sp
Push local variables
Push callee-save registers

Callee pop code

(epilogue)

Pop callee-save registers
Pop callee activation record
Pop old base-pointer

`return`

pop return address
Jump to address

caller

Caller pop code

…

Pop return value + parameters
Pop caller-save registers

“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)    {
        int b=a+1;
        f1();
        g1(b);
        return(b+2);
}
```

```
.global _foo
              Add_Constant -K, SP //allocate space for foo
              Store_Local  R5, -14(FP) // save R5
              Load_Reg  R5, R0; Add_Constant R5, 1
              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 (frame – unknown size)

- 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 → a → a → c → b → c → d

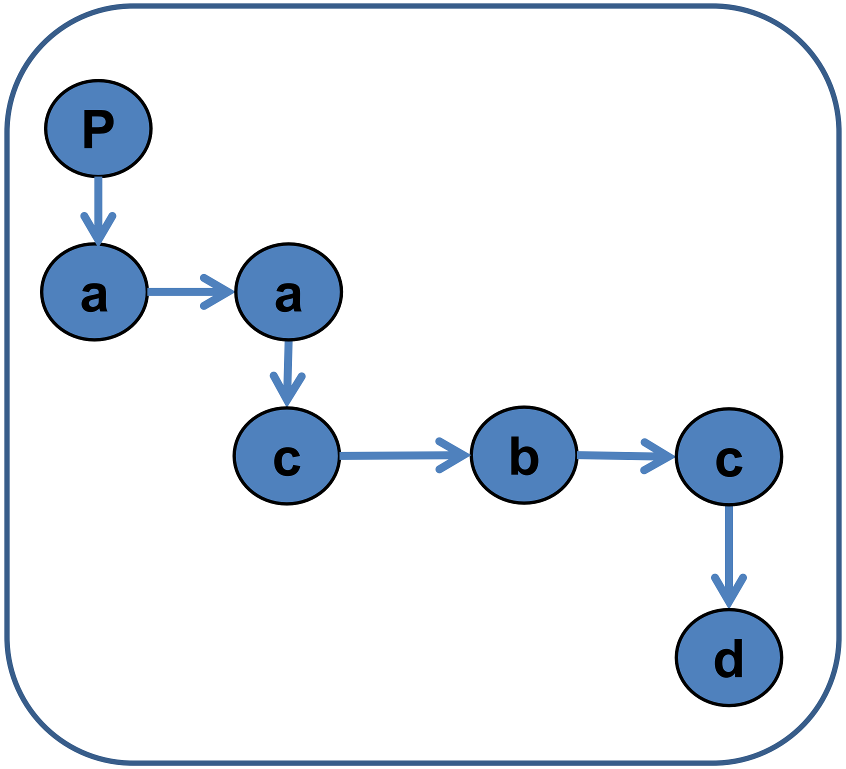
what are the addresses of variables "**x**," "**y**" and "**z**" in procedure d?

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?

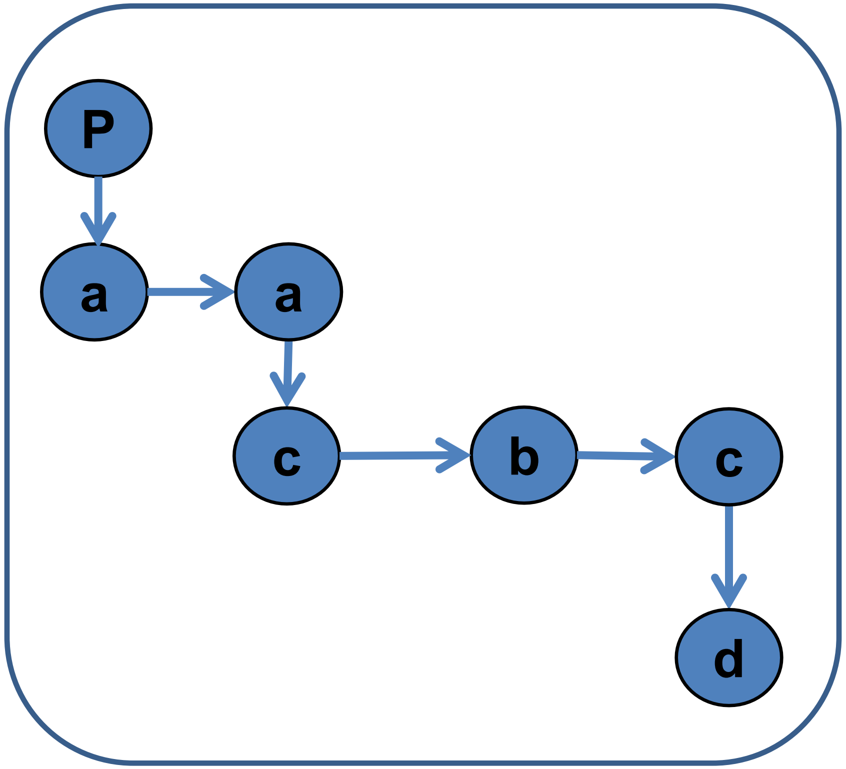
Possible call sequence:
p → a → a → c → b → c → d



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 → a → a → c → b → c → d



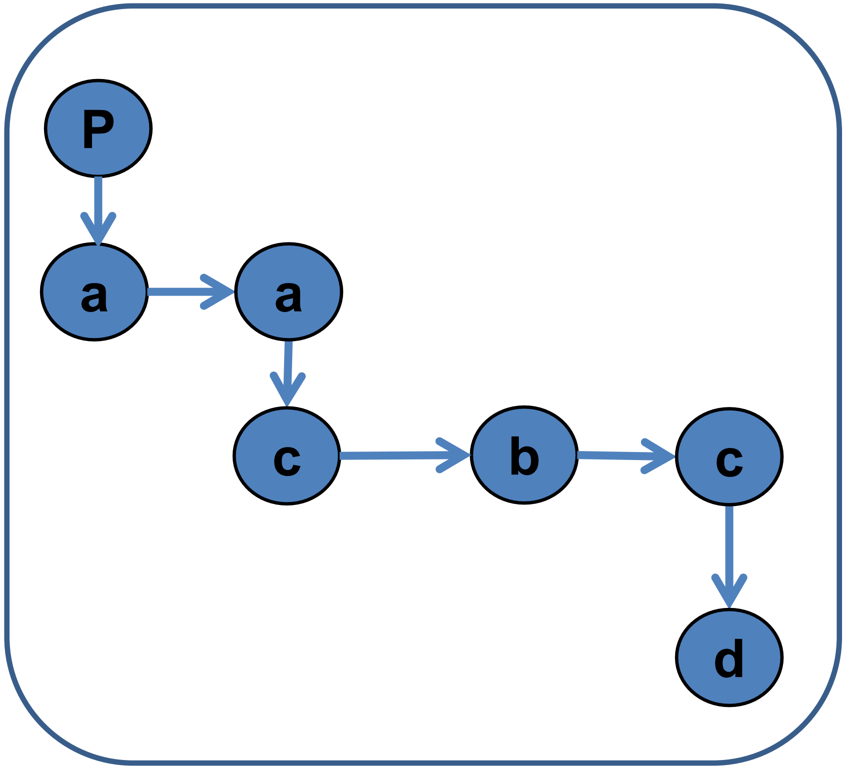
Nested Procedures

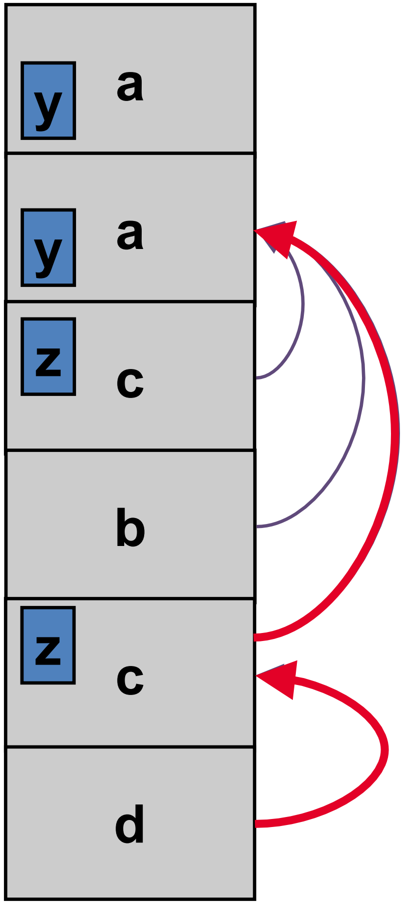
- problem: a routine may need to access variables of another routine that contains it statically
- solution: lexical pointer (a.k.a. access link) in the activation record
- lexical pointer points to the last activation record of the nesting level above it
  - in our example, lexical pointer of d points 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 → a → a → c → b → c → d

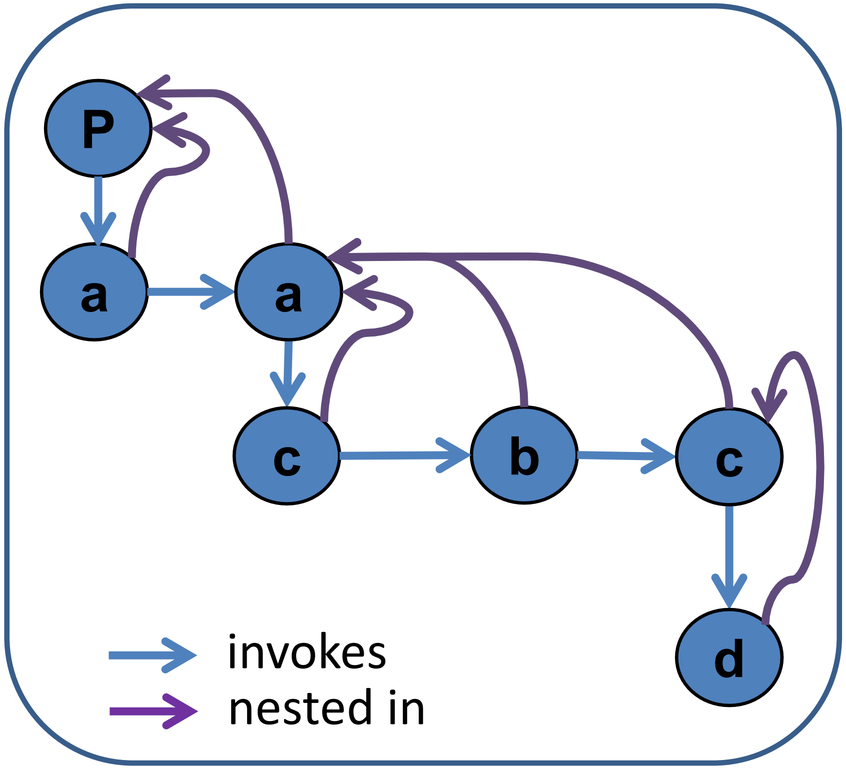


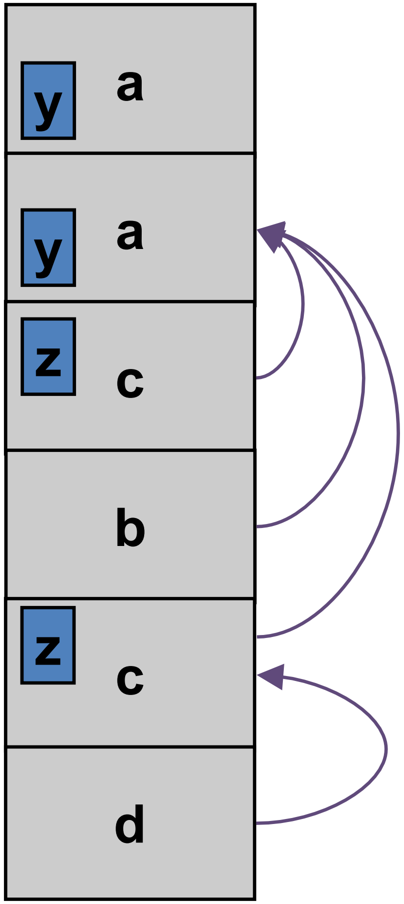


Lexical Pointers

```
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 → a → a → c → b → c → d



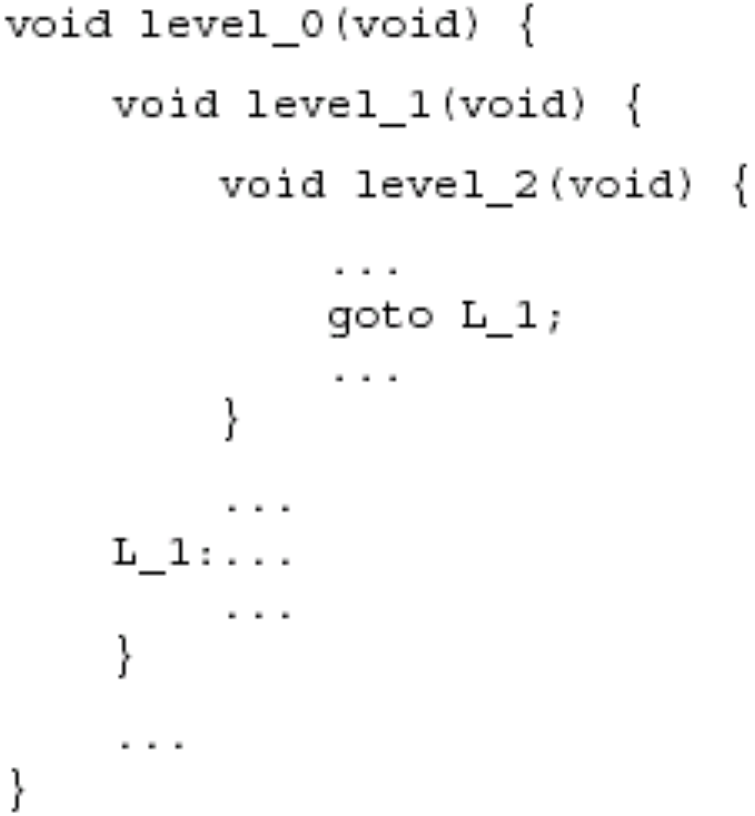


Activation Records: Remarks

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

Non-Local goto in C syntax

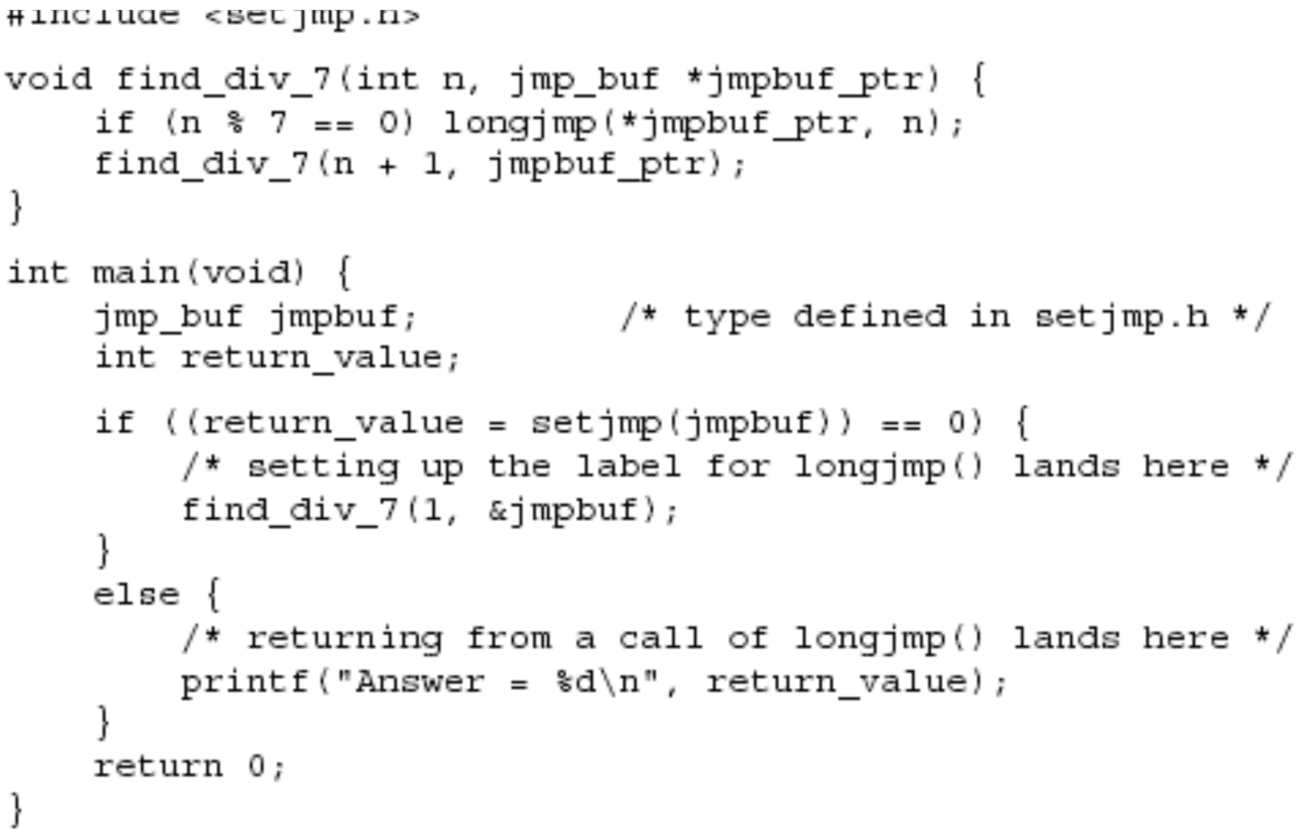


```
void level_0(void) {

    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;
}
```

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 varags)
  - 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 $\longleftarrow$sp-K M[sp+K+0] $\longleftarrow r_2$ $X_{157} \longleftarrow$r4 $X_{158} \longleftarrow$r5	save %sp, -K, %sp M[fp+68]$\longleftarrow i_0$ $X_{157} \longleftarrow i_1$ $X_{158} \longleftarrow i_2$

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 ;
}
```

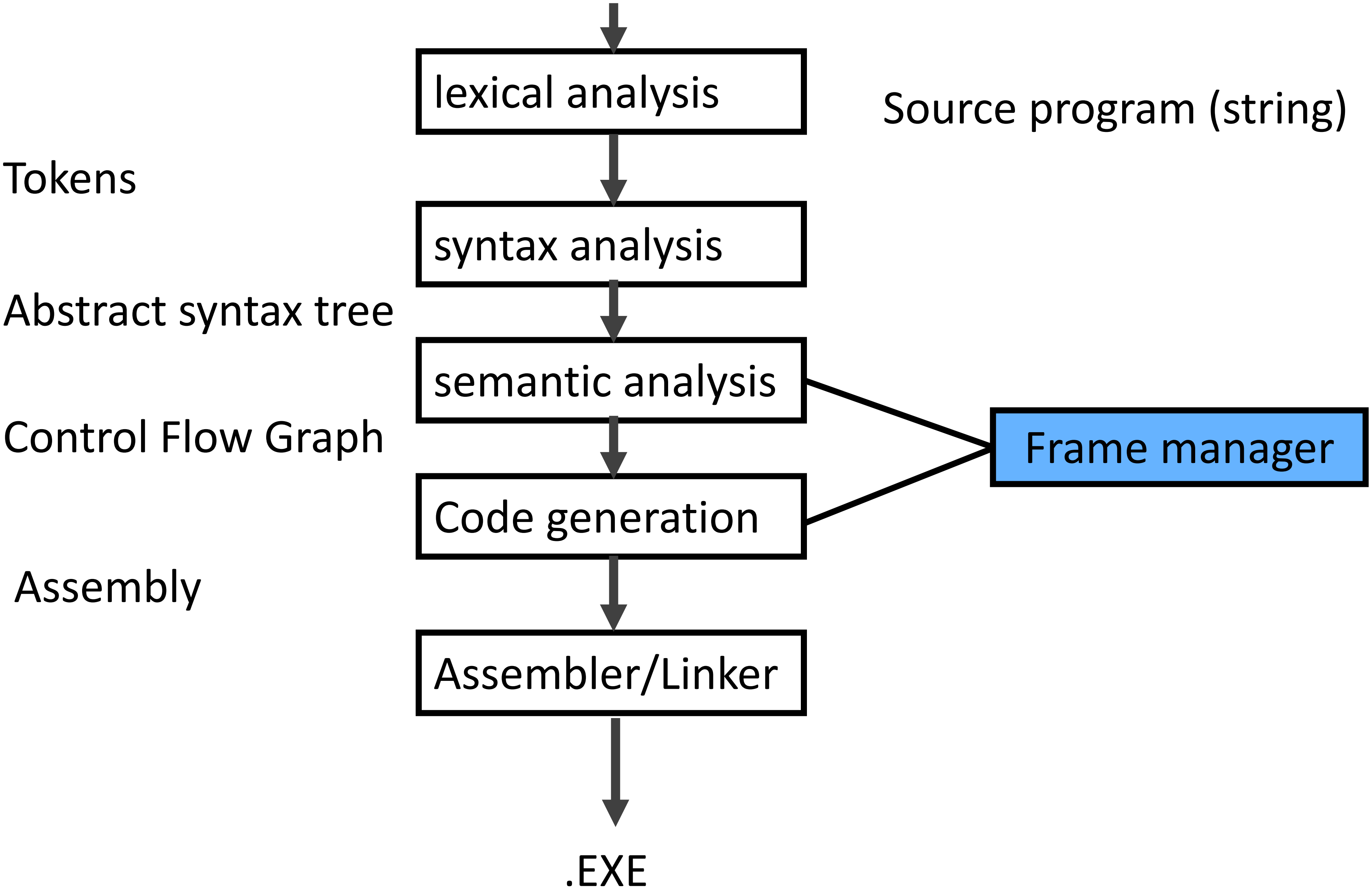
- Example 3: Dynamic allocation

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

Compiler Implementation

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

Basic Compiler Phases



Source program (string)

lexical analysis

Tokens

syntax analysis

Abstract syntax tree

semantic analysis

Control Flow Graph

Code generation

Frame manager

Assembly

Assembler/Linker

.EXE

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