Compilation

0368-3133 (Semester A, 2013/14)

Lecture 9: Activation Records + Register Allocation

Noam Rinetzky

Slides credit: Roman Manevich, Mooly Sagiv and Eran Yahav

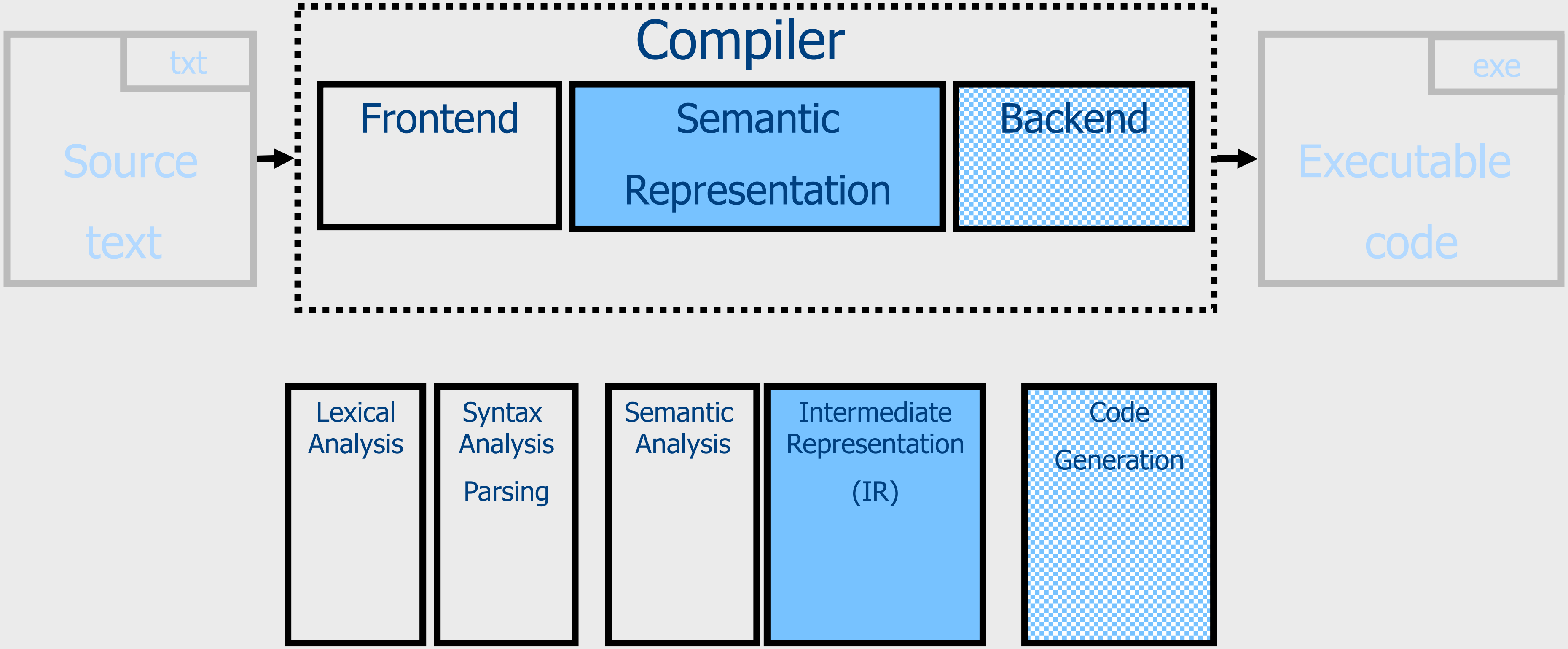
What is a Compiler?

"A compiler is a computer program that transforms source code written in a programming language (source language) into another language (target language).

The most common reason for wanting to transform source code is to create an executable program."

--Wikipedia

Conceptual Structure of a Compiler



From scanning to parsing

program text **((23 + 7) * x)**

Lexical Analyzer

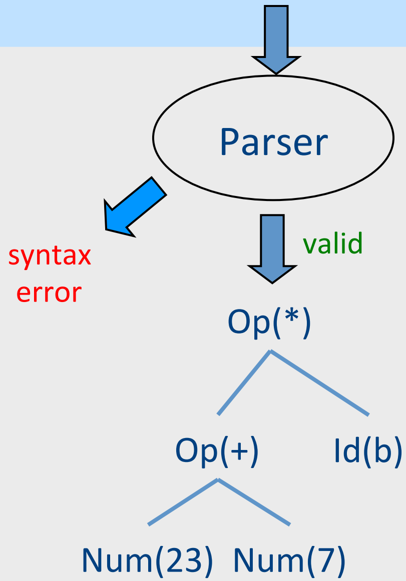
token stream

(	(	23	+	7	)	*	x	)
LP	LP	Num	OP	Num	RP	OP	Id	RP

Grammar:

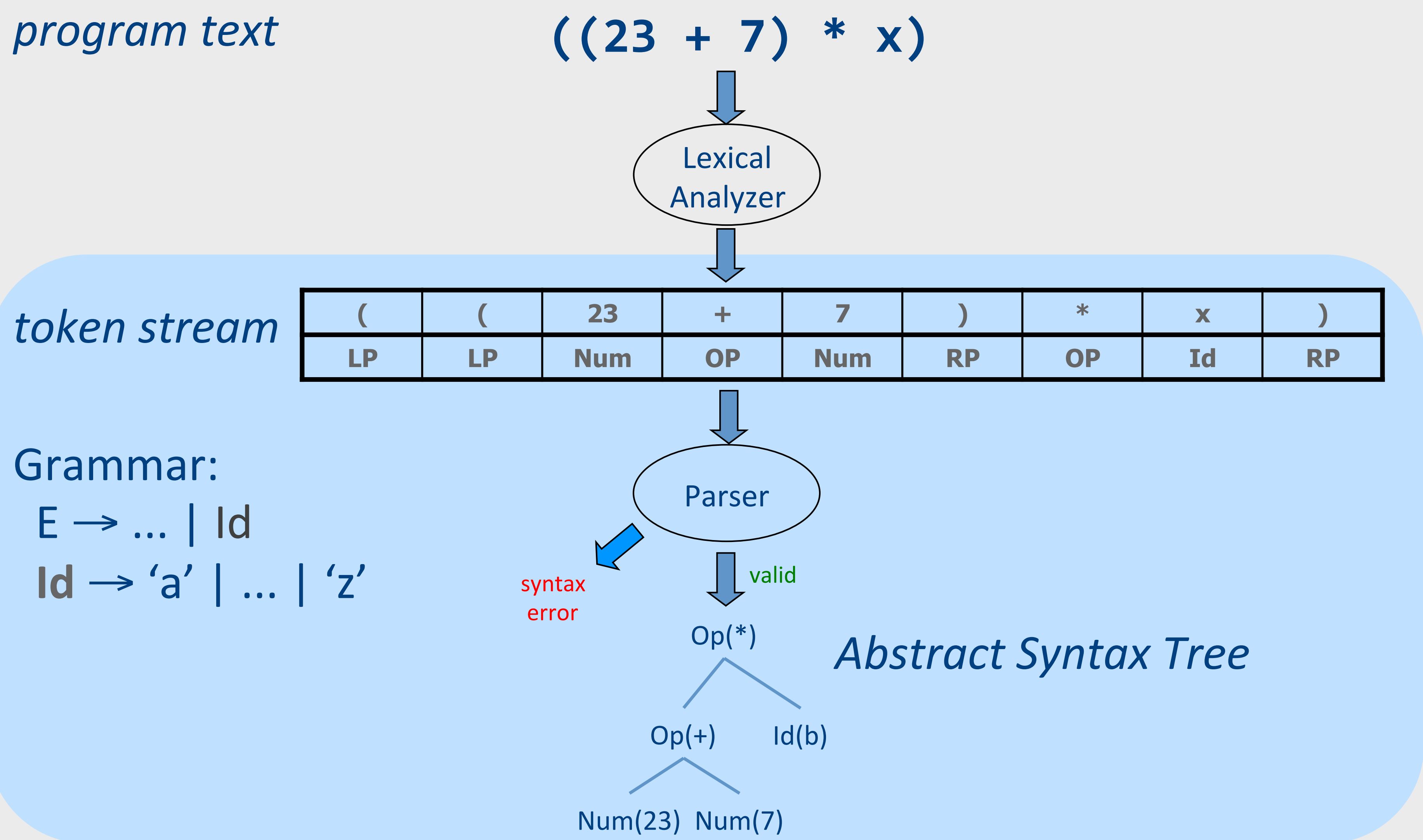
E → ... | Id

Id → ‘a’ | ... | ‘z’



Abstract Syntax Tree

From scanning to parsing



(	(	23	+	7	)	*	x	)
LP	LP	Num	OP	Num	RP	OP	Id	RP

Grammar:

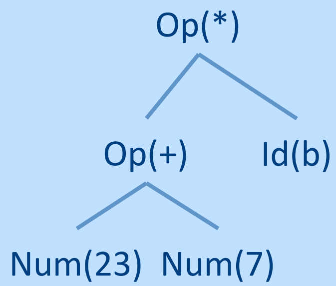
E → ... | Id

Id → ‘a’ | ... | ‘z’

Context Analysis

Type rules

$$\frac{E1 : int \qquad E2 : int}{E1 + E2 : int}$$



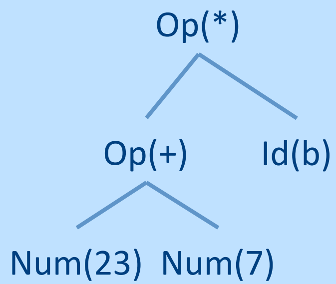
Abstract Syntax Tree

Semantic Error

Valid + Symbol Table

Code Generation

cgen
Frame Manager



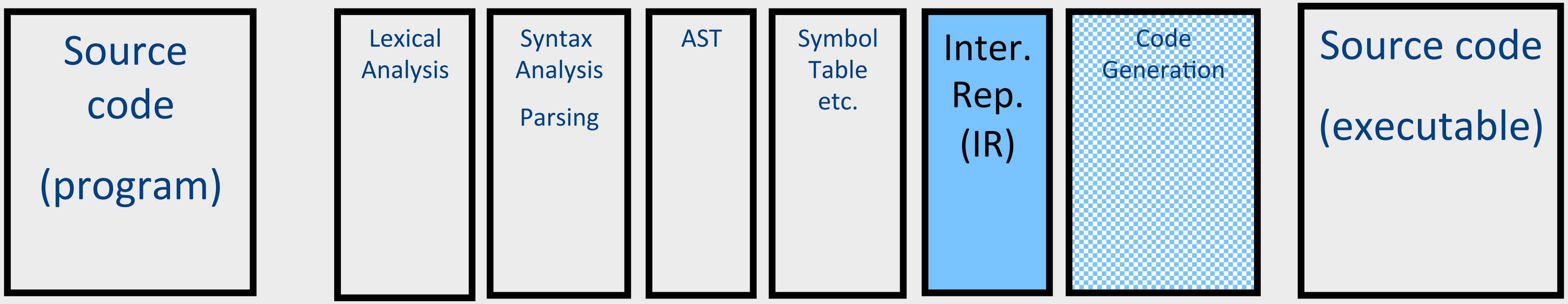
Valid Abstract Syntax Tree
Symbol Table

Verification (possible runtime)
Errors/Warnings

Intermediate Representation (IR)

input → Executable Code → output

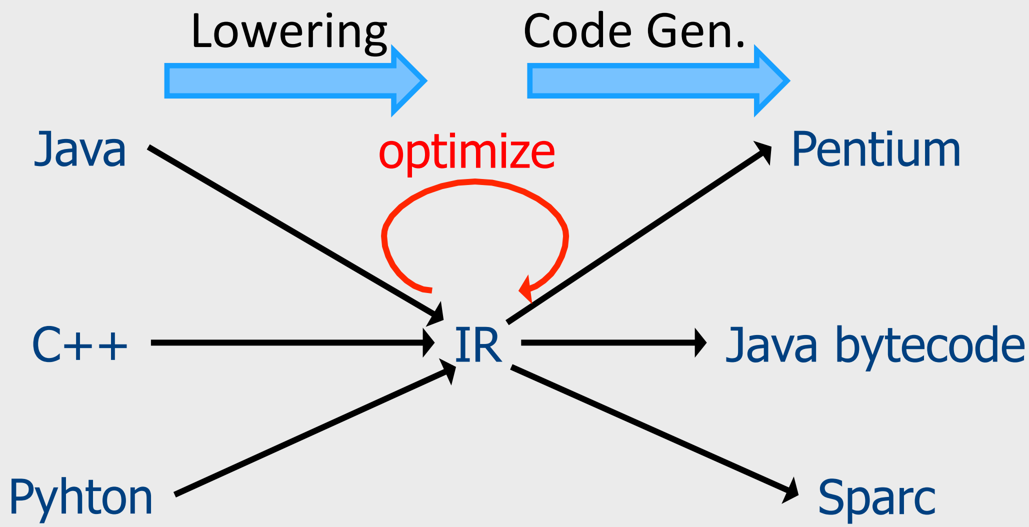
Code Generation: IR

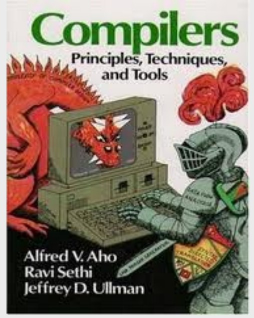


- Translating from abstract syntax (AST) to intermediate representation (IR)
  - Three-**A**ddress **C**ode
    - Primitive statements, control flow, procedure calls

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 node types (instructions)



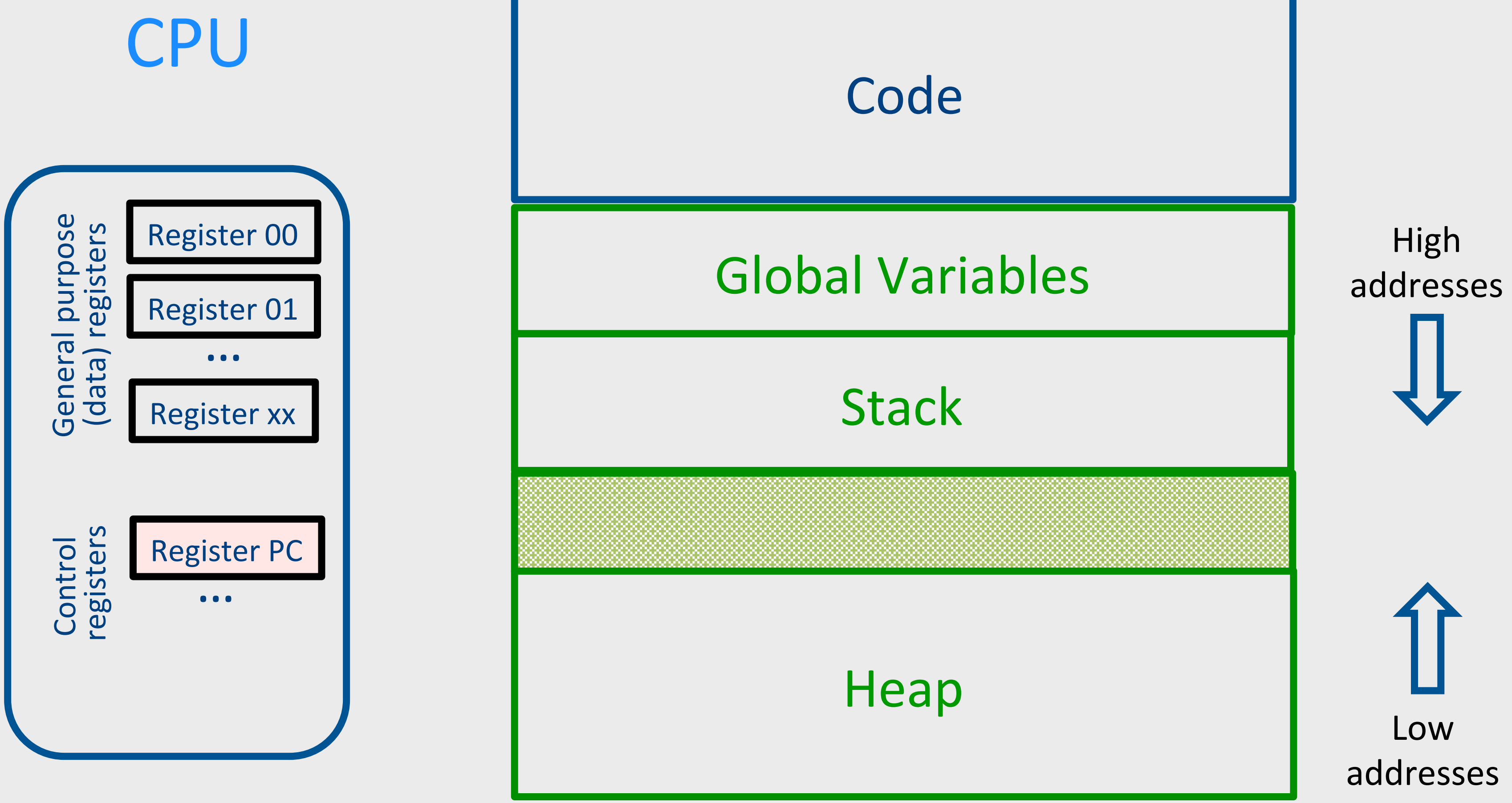


Chapter 8

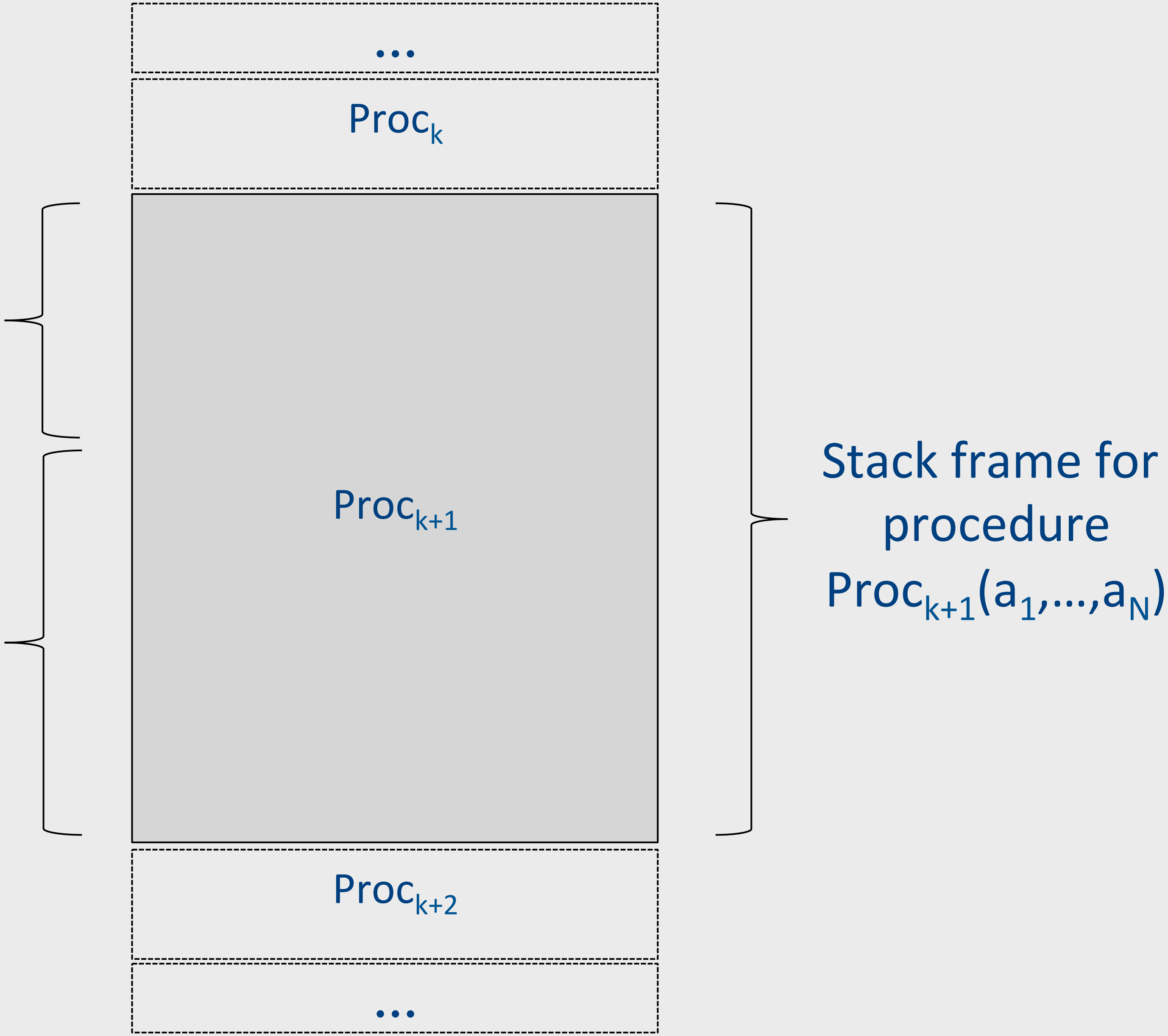
Three-Address Code IR

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

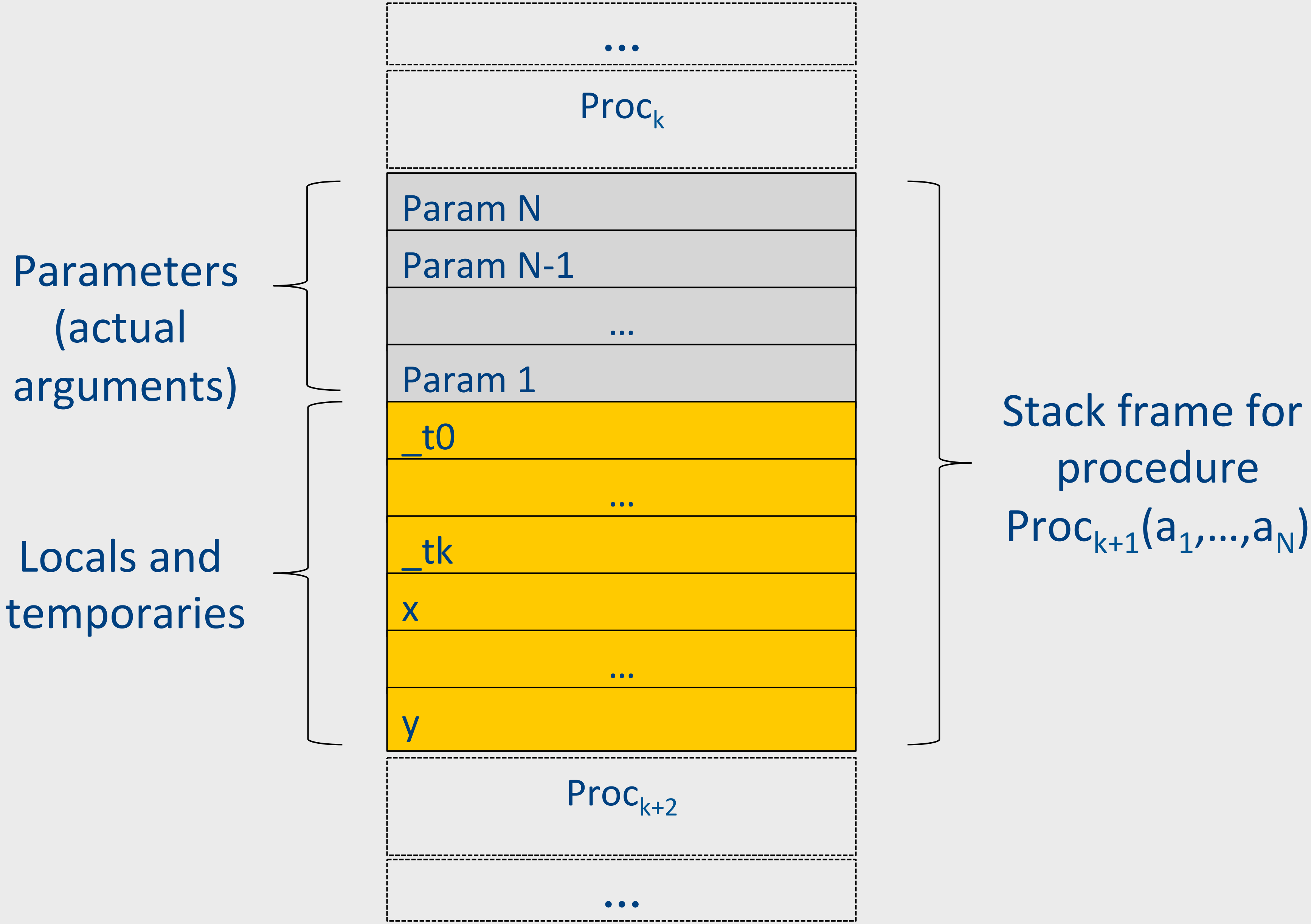
Abstract Register Machine



Abstract Activation Record Stack



Abstract Stack Frame



“Abstract” Code

- Memory load/store
  - Load: Memory → Register
  - Store: Register → Memory

Fixed number of Registers

- Operation: Between registers*
  - R1 = R2 + R3

TAC generation for expressions

- **cgen**(*atomic_expr)* directly generates TAC for atomic expressions
  - Constants, identifiers,…
- **cgen**(*compund_expr)* recursively generates TAC for compound expressions
  - binary operators, procedure calls, …
  - use temporary variables (**registers**) to store values of intermediate expressions

cgen example

cgen(5 + x) = {

Choose a new temporary *t*

Let t_1 = **cgen**(5)

Let t_2 = **cgen**(x)

Emit($t = t_1 + t_2$)

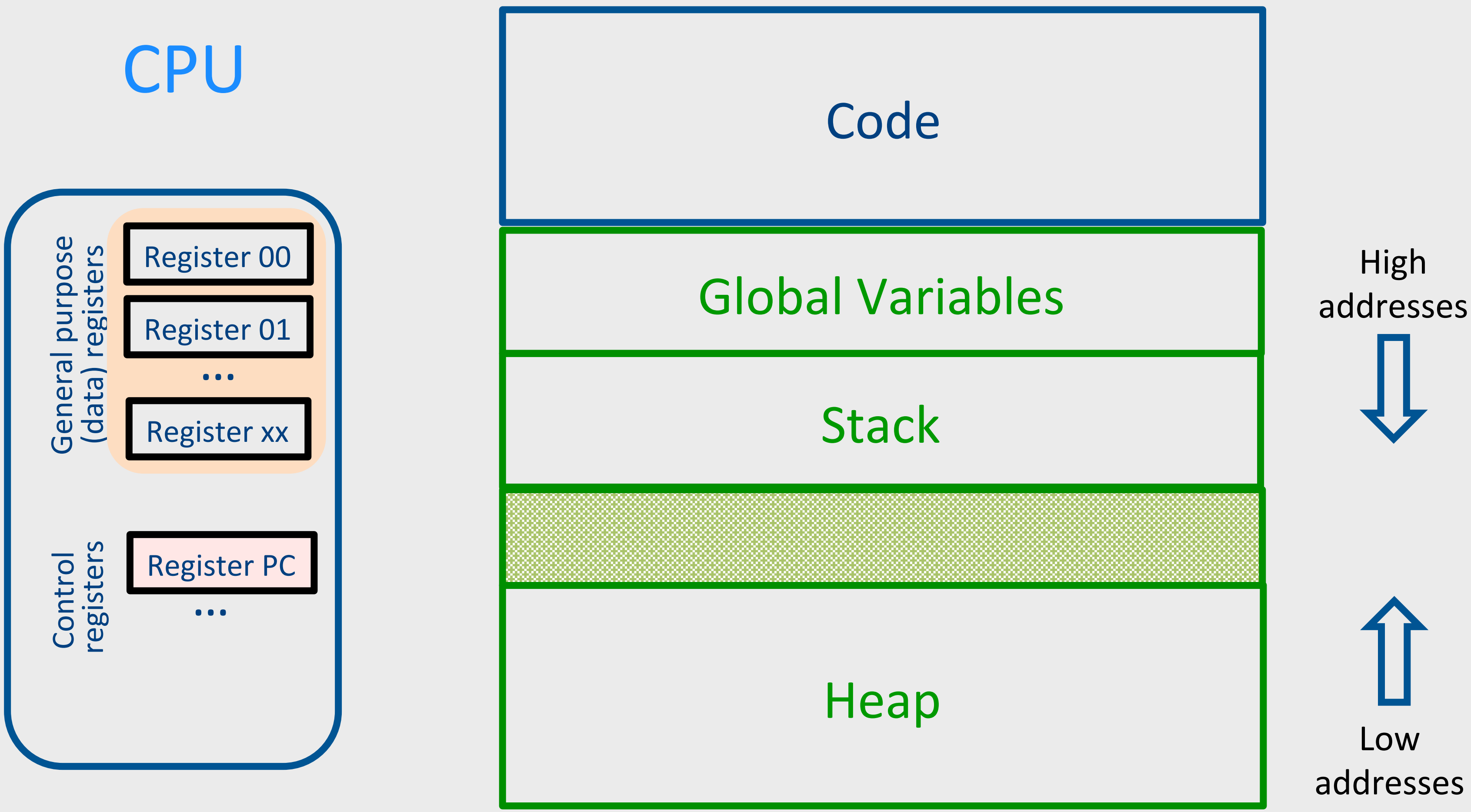
Return *t*

}

Naïve **cgen** for expressions

- Maintain a counter for temporaries in c
- Initially: c = 0
- **cgen**(e_1 *op* e_2) = {
  Let A = **cgen**(e_1)
  c = c + 1
  Let B = **cgen**(e_2)
  c = c + 1
  Emit(_tc = A *op* B;)
  Return _tc
  }

Abstract Register Machine



TAC Generation for Control Flow Statements

- Label introduction

  ```
  _label_name:
  ```

  Indicates a point in the code that can be jumped to
- Unconditional jump: go to instruction following label L

  ```
  Goto L;
  ```

- Conditional jump: test condition variable t;
  if 0, jump to label L

  ```
  IfZ t Goto L;
  ```

- Similarly : test condition variable t;
  if 1, jump to label L

  ```
  IfNZ t Goto L;
  ```

Control-flow example – conditions

```
int x;
int y;
int z;

if (x < y)
    z = x;
else
    z = y;
z = z * z;
```

```
        _t0 = x < y;
        IfZ _t0 Goto _L0;
        z = x;
        Goto _L1;
_L0:
        z = y;
_L1:
        z = z * z;
```

cgen for `if-then-else`

cgen(if (e) s_1 else s_2)

```
Let _t = cgen(e)
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: )
```

Interprocedural IR: Using a Stack

- Stack of activation records
  - One activation record per procedure invocation
- Call ➔ push new activation record
- Return ➔ pop activation record
- Only one “active” activation record
  - top of stack

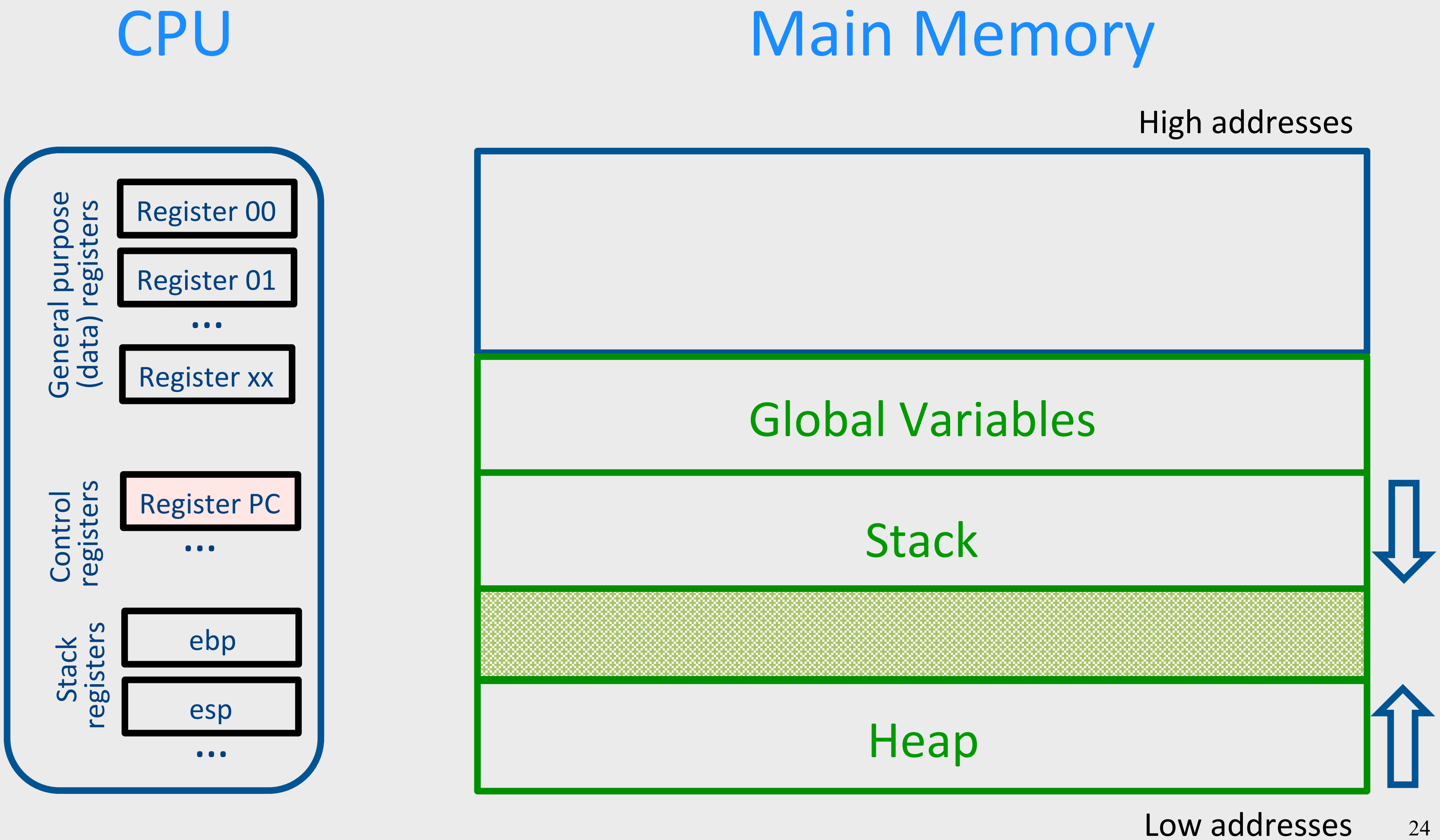
Mostly-Abstract Handling Procedures

- Store local parameters/variables/temporaries in the stack

- **procedure call** pushes arguments to stack and jumps to the function label
  - `x=f(a1,…,an)` ➔
    - `Push` a_1`;` … `Push` a_n`; return_address`
    - `Jump to first address of f;`
    - `Pop x;` // copy returned value

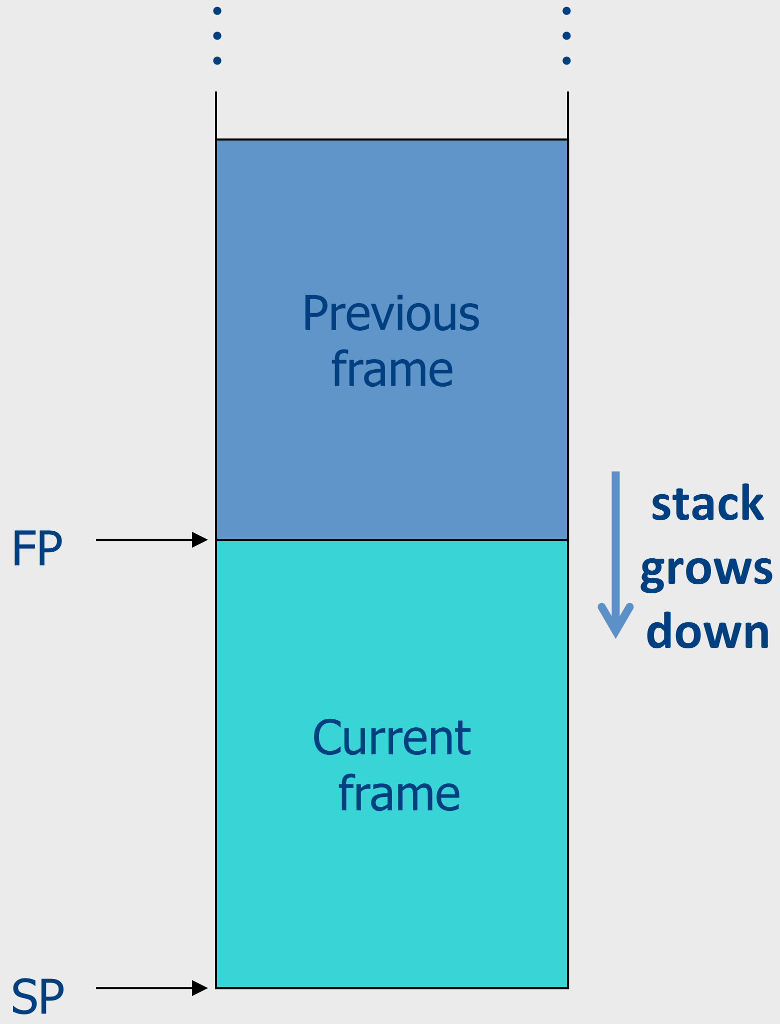
- **Procedure return** clean stack, store return value, return control to caller
  - `return x` ➔
    - `Pop` $temp_1, …, temp_k, var_1, …, var_m, param_1, …, param_n$
    - `Push x;`
    - `Jump to return_address`

Semi-Abstract Register Machine



Runtime Stack

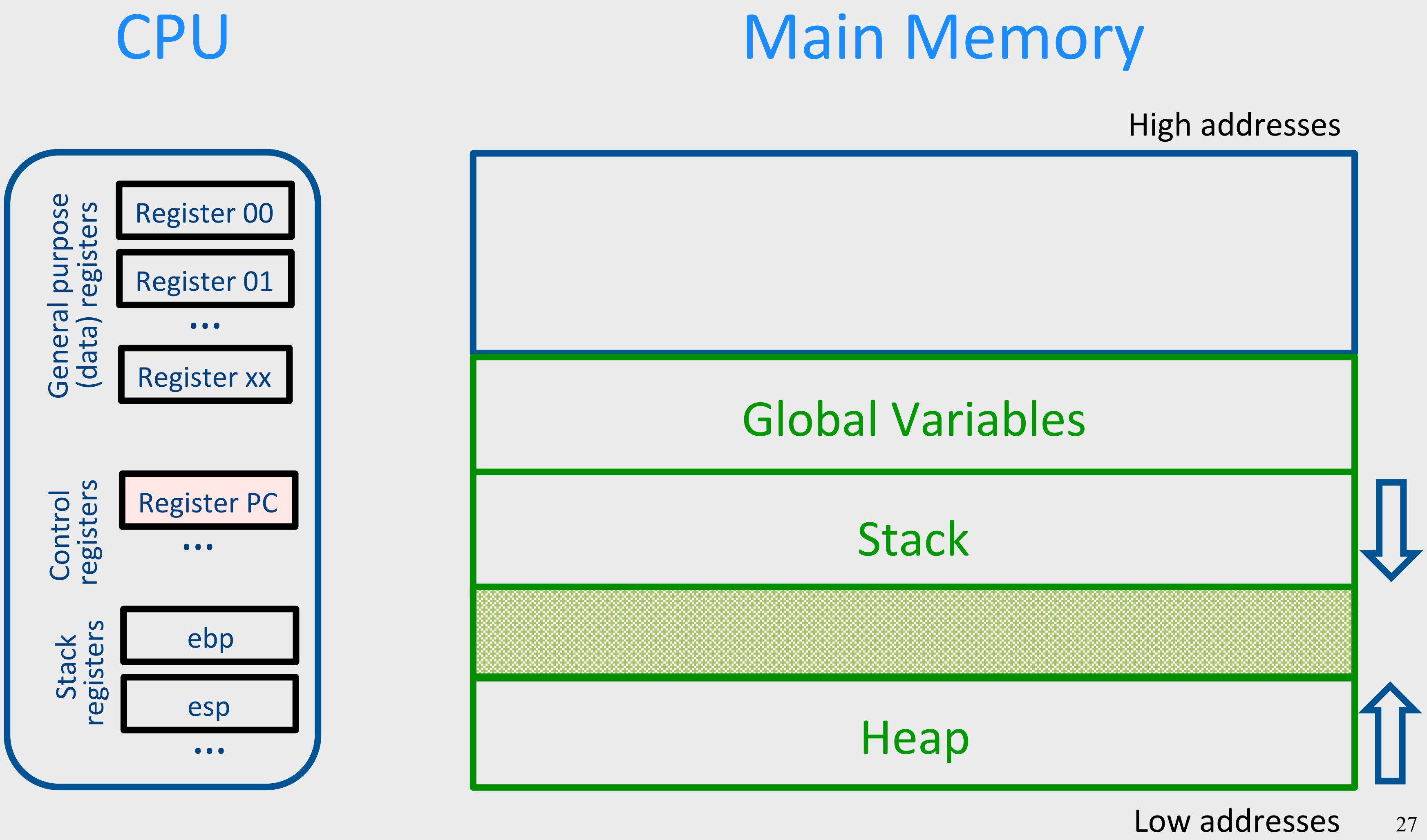
- SP – stack pointer
  – top of current frame
- FP – frame pointer
  – base of current frame
  - Sometimes called BP (base pointer)



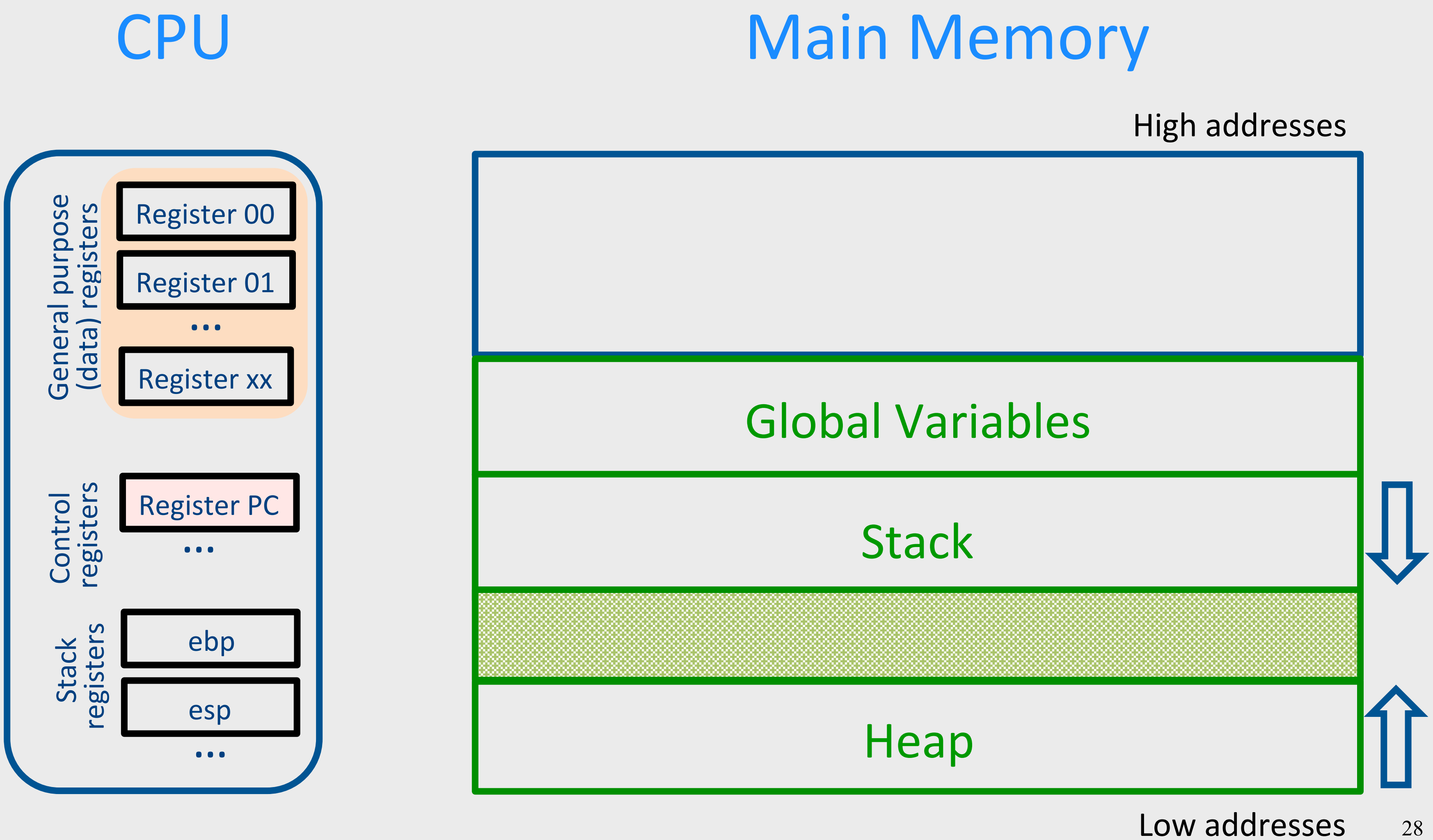
L-Values of Local Variables

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

What's the Problem with This Picture?



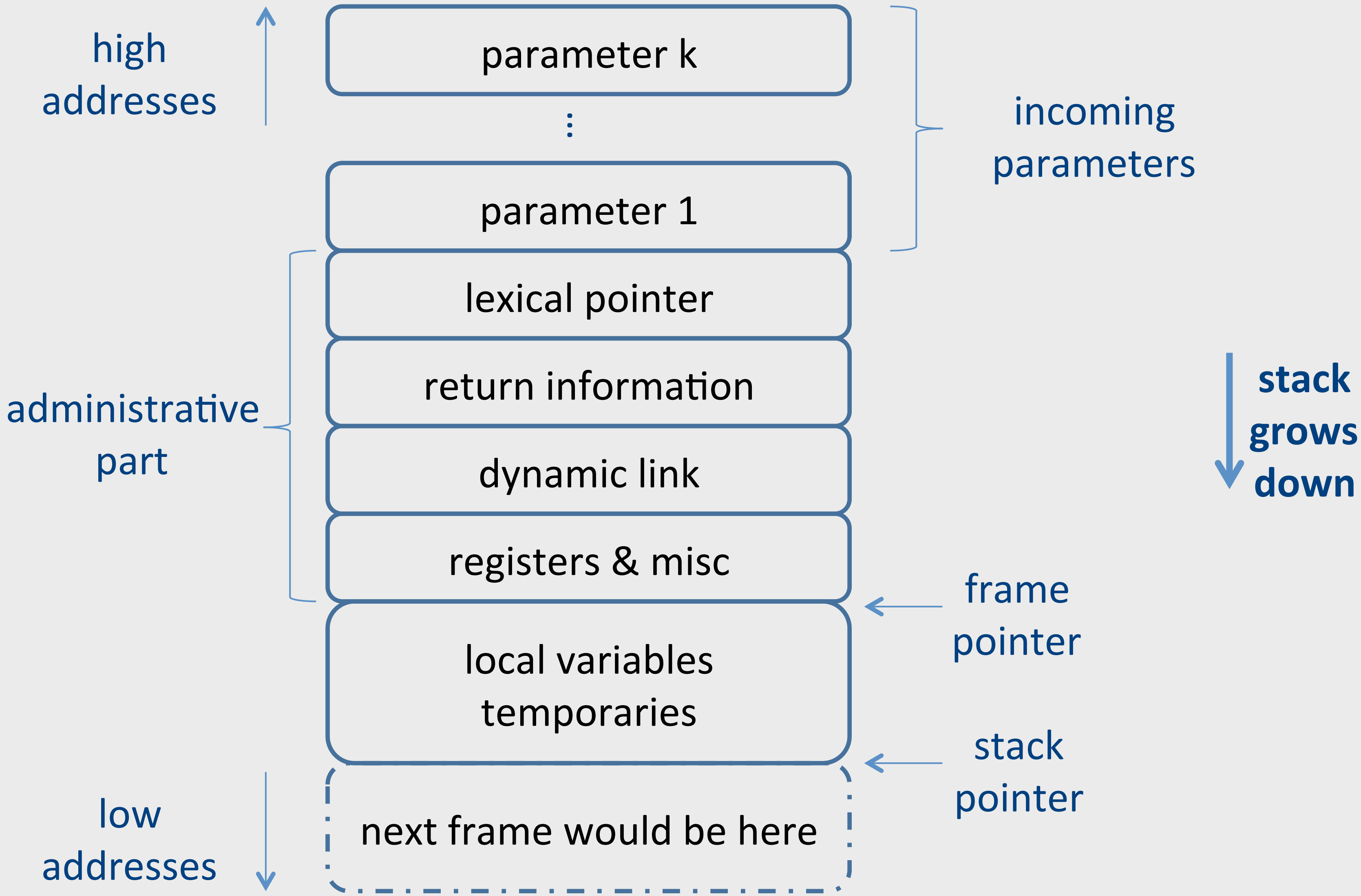
What's the Problem with This Picture?



Saving and Restoring Registers

- **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

Activation Record (Frame)



Pentium Runtime Stack

Register	Usage
ESP	Stack pointer
EBP	Base pointer

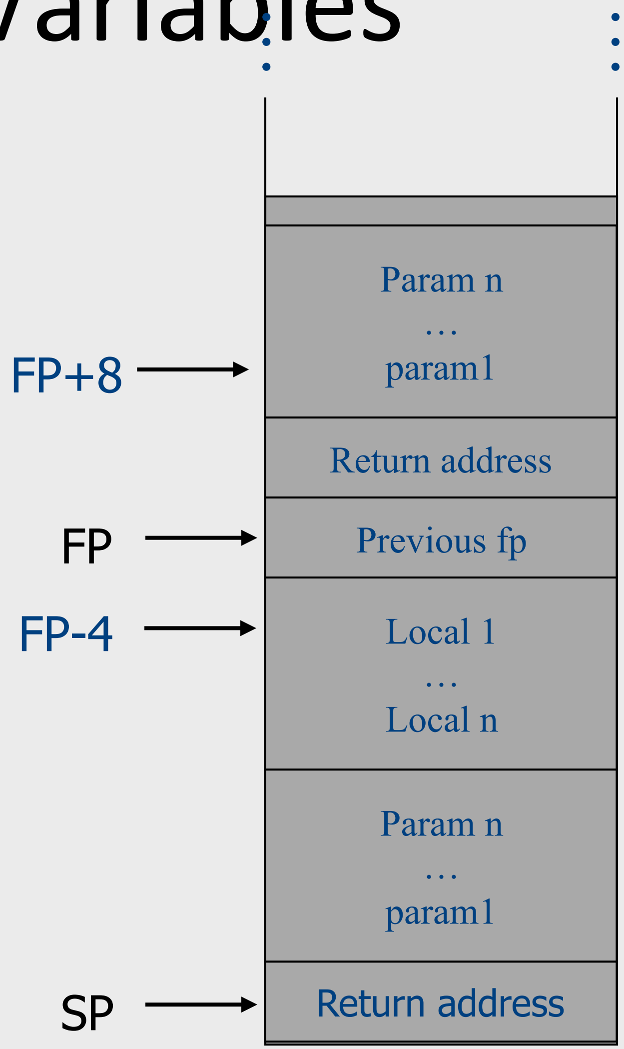
Pentium stack registers

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

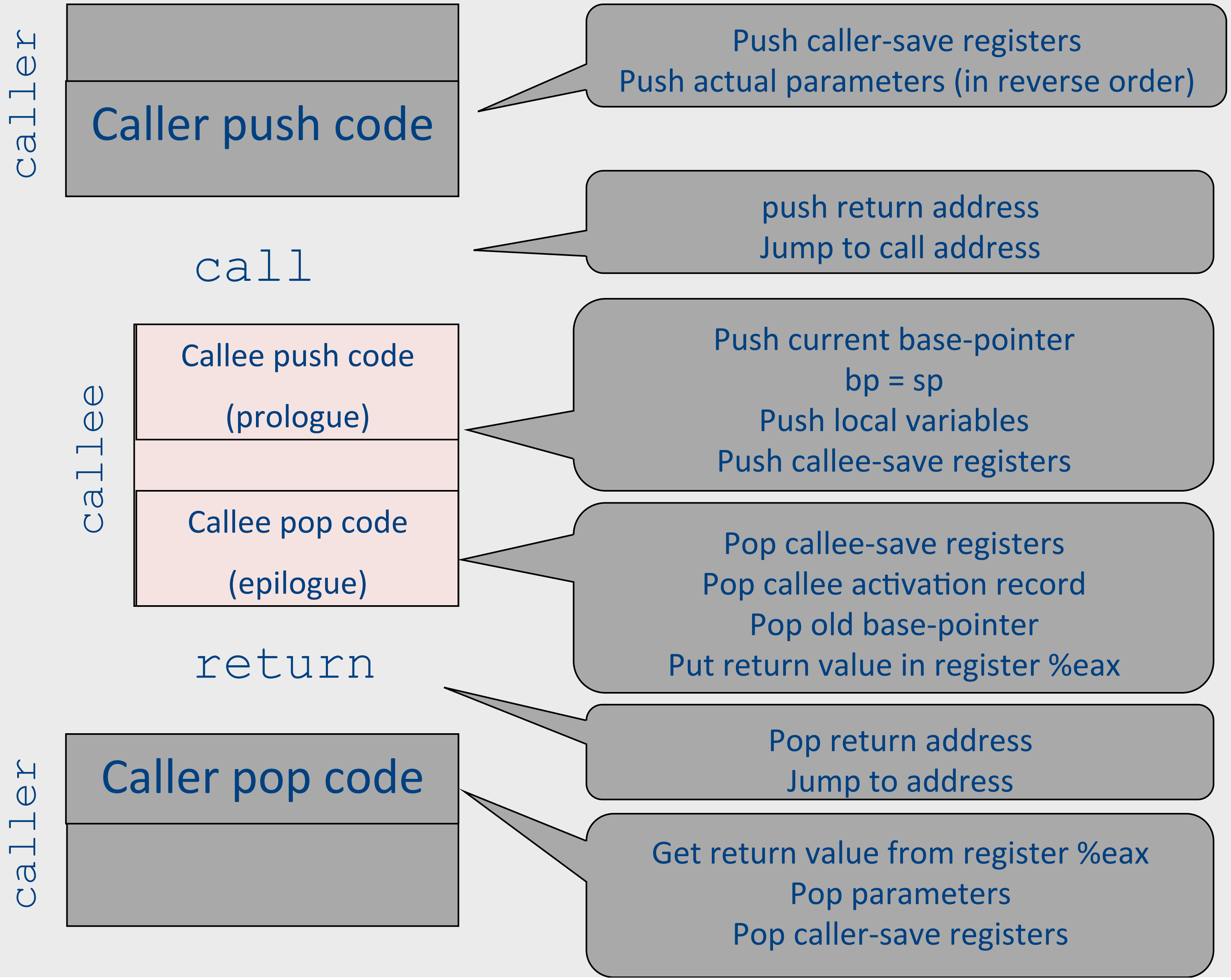
Pentium stack and call/ret instructions

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



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
- Architecture defines caller- & callee- save registers
  - Separate compilation
  - Interoperability between code produced by different compilers/languages
  - But compiler writers decide when to use calller/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

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) ;
}
```

B_0 B_1 B_2 B_3

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**

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?

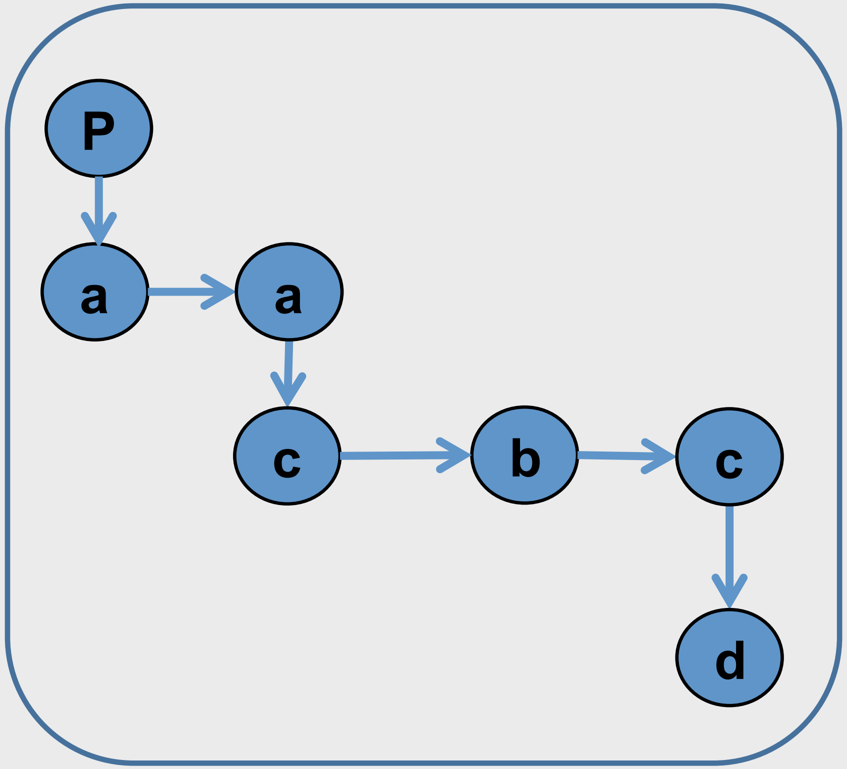
Nested Procedures

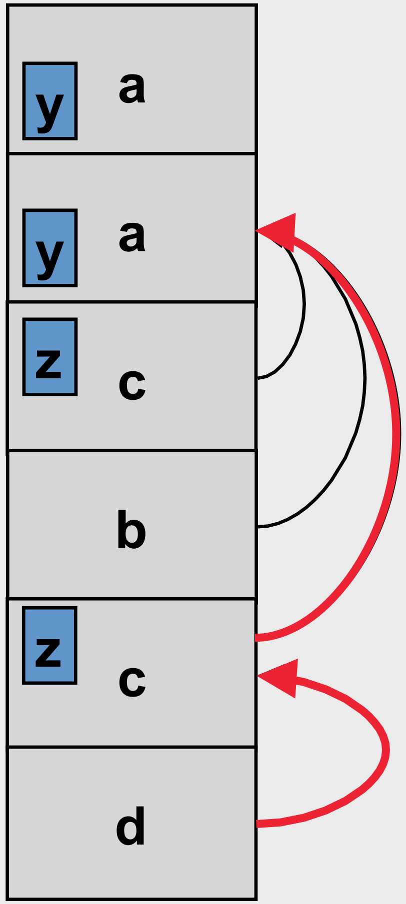
- For example – Pascal
- Any routine can have sub-routines
- Any sub-routine can access anything that is defined in its containing scope or inside the sub-routine itself
  - “non-local” variables

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





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

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 ;
}
```

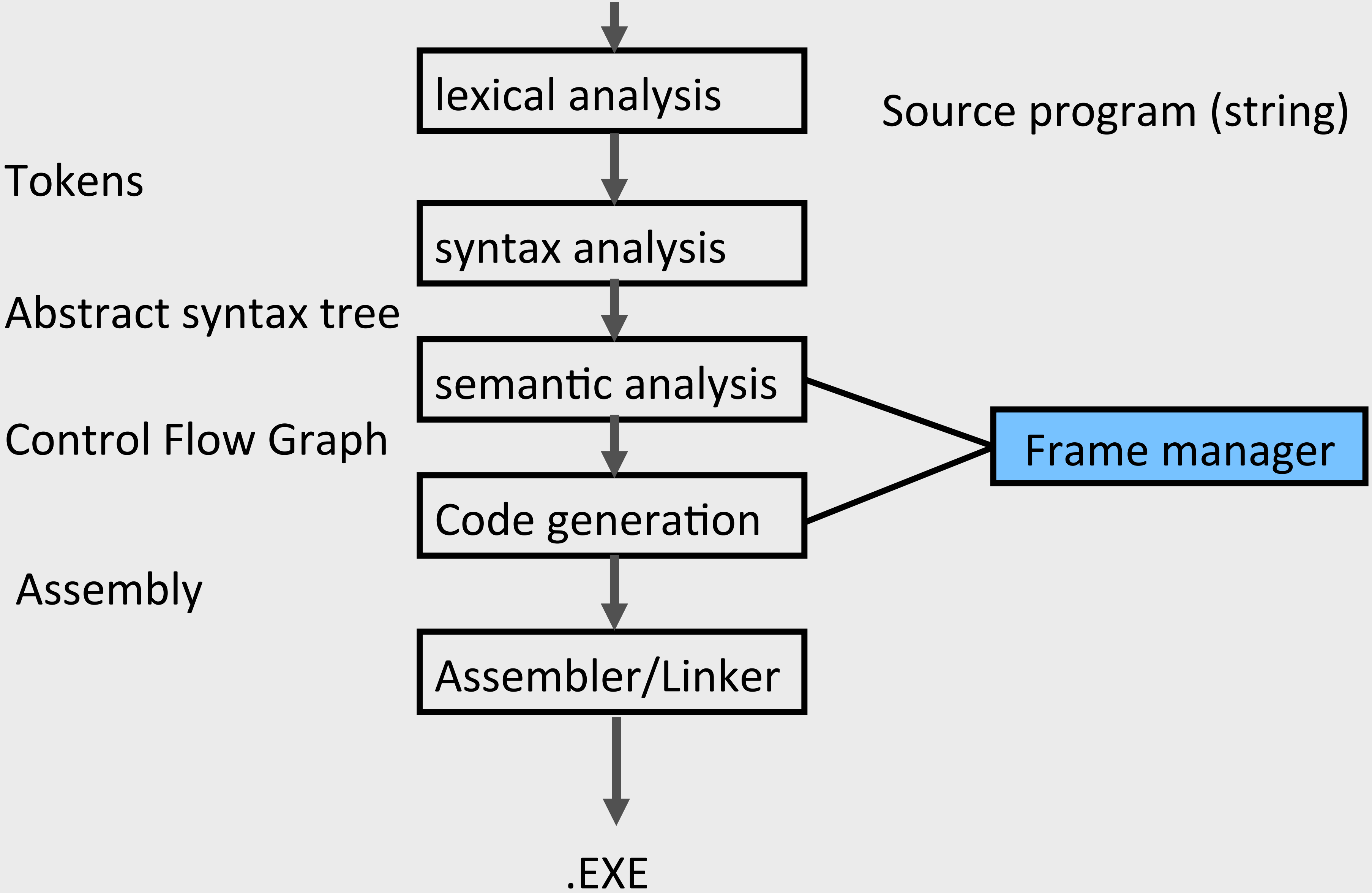
- Example 3: Dynamic allocation

```
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

Invocations to Frame

- “Allocate” a new frame
- “Allocate” new local variable
- Return the L-value of local variable
- Generate code for procedure invocation
- Generate prologue/epilogue
- Generate code for procedure return

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

Activation Records: Summary

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

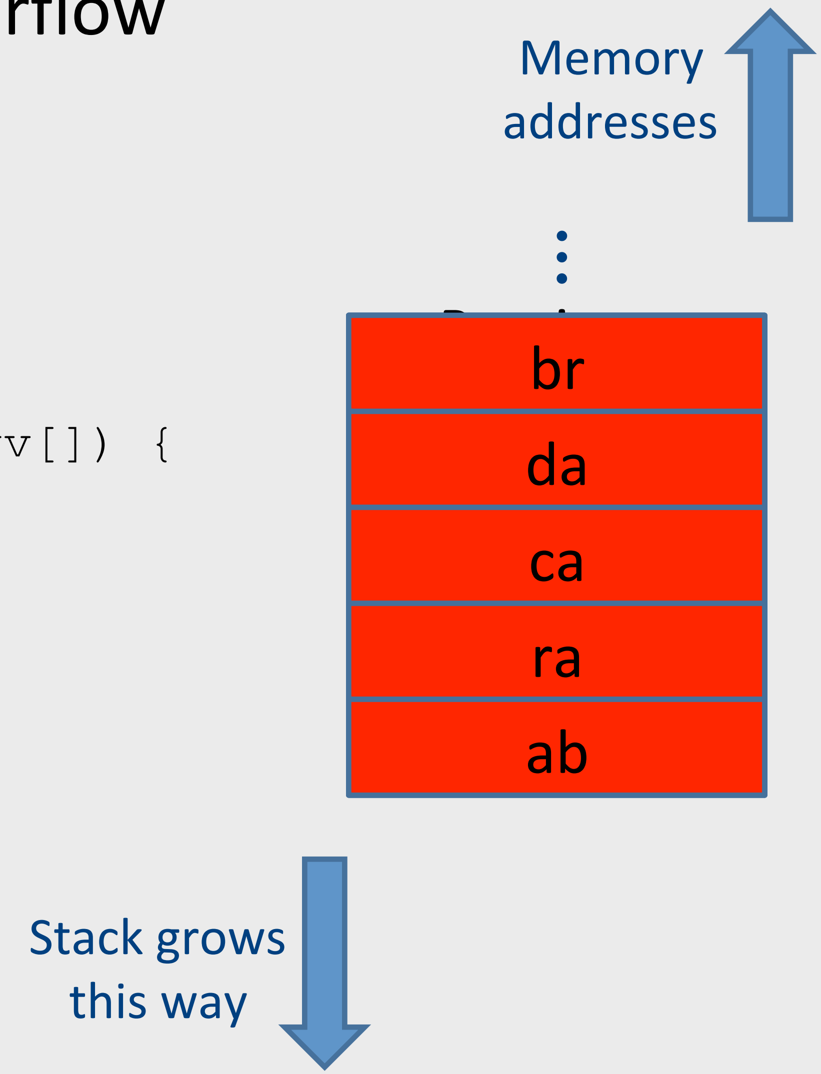
One More Thing*

Windows Exploit(s)

Buffer Overflow

```
void foo (char *x) {
  char buf[2];
  strcpy(buf, x);
}
int main (int argc, char *argv[]) {
  foo(argv[1]);
}

 ./a.out abracadabra
Segmentation fault
```



Buffer overflow

```
int check_authentication(char *password) {
  int auth_flag = 0;
  char password_buffer[16];

  strcpy(password_buffer, password);
  if(strcmp(password_buffer, "brillig") == 0)
        auth_flag = 1;
  if(strcmp(password_buffer, "outgrabe") == 0)
        auth_flag = 1;
   return auth_flag;
}
int main(int argc, char *argv[]) {   if(argc < 2) {
  printf("Usage: %s <password>\n", argv[0]); exit(0);      }
  if(check_authentication(argv[1])) {
     printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     printf("      Access Granted.\n");
     printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");   }
  else {
     printf("\nAccess Denied.\n");
   }
}
```

(source: "hacking – the art of exploitation, 2nd Ed")

Buffer overflow

```
 int check_authentication(char *password) {
  int auth_flag = 0;
  char password_buffer[16];

  strcpy(password_buffer, password);
  if(strcmp(password_buffer, "brillig") == 0)
          auth_flag = 1;
  if(strcmp(password_buffer, "outgrabe") == 0)
          auth_flag = 1;
   return auth_flag;
}
int main(int argc, char *argv[]) {   if(argc < 2) {
  printf("Usage: %s <password>\n", argv[0]); exit(0);     }
  if(check_authentication(argv[1])) {
     printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     printf("      Access Granted.\n");
     printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");   }
  else {
     printf("\nAccess Denied.\n");
   }
}
```

(source: "hacking – the art of exploitation, 2nd Ed")

Buffer overflow

```
int check_authentication(char *password) {
char password_buffer[16];
int auth_flag = 0;

  strcpy(password_buffer, password);
  if(strcmp(password_buffer, "brillig") == 0)
          auth_flag = 1;
  if(strcmp(password_buffer, "outgrabe") == 0)
          auth_flag = 1;
   return auth_flag;
}
int main(int argc, char *argv[]) {   if(argc < 2) {
  printf("Usage: %s <password>\n", argv[0]); exit(0);     }
  if(check_authentication(argv[1])) {
     printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
     printf("     Access Granted.\n");
     printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");   }
   else {
      printf("\nAccess Denied.\n");
   }
}
```


(source: "hacking – the art of exploitation, 2nd Ed")

Buffer overflow

```
0x08048529 <+69>:   movl   $0x8048647,(%esp)
0x08048530 <+76>:   call   0x8048394 <puts@plt>
0x08048535 <+81>:   movl   $0x8048664,(%esp)
0x0804853c <+88>:   call   0x8048394 <puts@plt>
0x08048541 <+93>:   movl   $0x804867a,(%esp)
0x08048548 <+100>:  call   0x8048394 <puts@plt>
0x0804854d <+105>:  jmp    0x804855b <main+119>
0x0804854f <+107>:  movl   $0x8048696,(%esp)
0x08048556 <+114>:  call   0x8048394 <puts@plt>
```

Example: Nested Procedures

```
program p;
var x: Integer;
procedure a
    var y: Integer;
    procedure b begin…b… end;
    function c
        var z: Integer;
        procedure d begin…d… end;
        begin…c…end;
    begin…a… end;
begin…p… end.
```

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

what is the address of variable "y" in procedure d?

The End

(Activation Records)