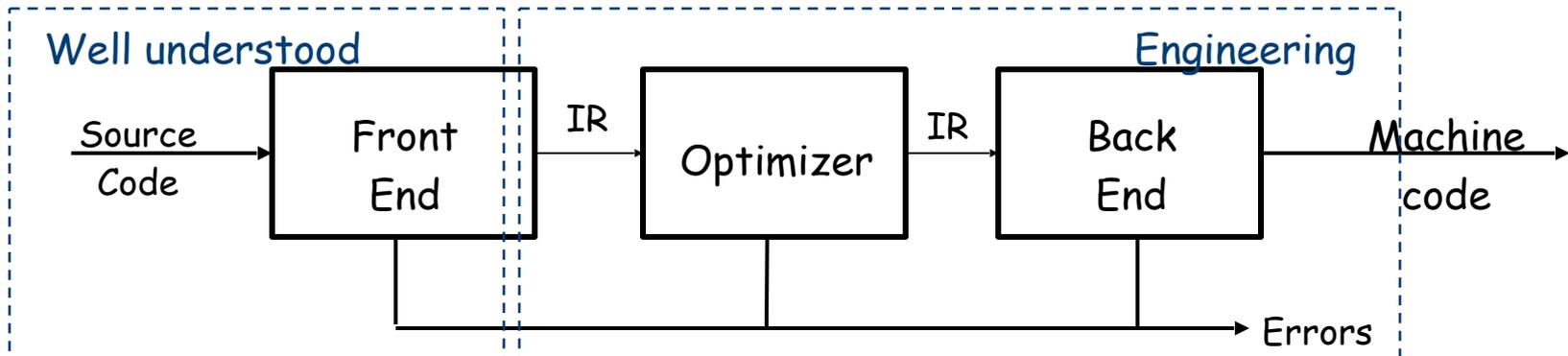# The Procedure Abstraction

Where are we?

The latter half of a compiler contains more open problems, more challenges, and more gray areas than the front half

- This is "compilation," as opposed to "parsing" or "translation"

Conceptual Overview

The compiler must provide, for each programming language construct, an implementation (or at least a strategy).

Those constructs fall into two major categories

- Individual statements (code shape)
- Procedures

We will look at procedures first, since they provide the surrounding context needed to implement statements

Object-oriented languages add some peculiar twists

Conceptual Overview

Procedures provide the central abstractions that make programming practical & large software systems possible

- Information hiding

- Distinct and separable name spaces

- Uniform interfaces

Hardware does little to support these abstractions

- Part of the compiler's job is to implement them
  — Compiler makes good on lies that we tell programmers

- Part of the compiler's job is to make it efficient
  — Role of code optimization

Practical Overview

The compiler must decide almost everything

- Location for each value (named and unnamed)

- Method for computing each result

  — For example, how should be translated a case statement?

- Compile-time versus runtime behaviour

  input(x);

  if x>3

  then foo(x);

  else fee(x);

All of these issues come to the forefront when we consider the implementation of procedures
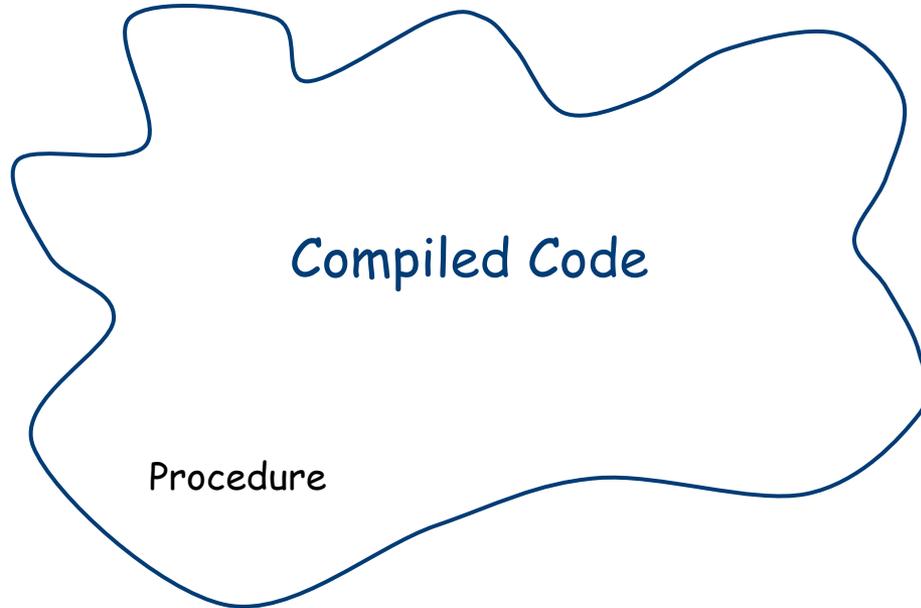
The Procedure Abstraction

Most of the tricky issues arise in implementing "procedures"

Issues

- Compile-time versus run-time behavior
- Assign storage for <u>everything</u> & map names to addresses
- Generate code to address any value
  — Compiler knows where some of them are
  — Compiler cannot know where others are
- Interfaces with other programs, other languages, & the OS
- Efficiency of implementation
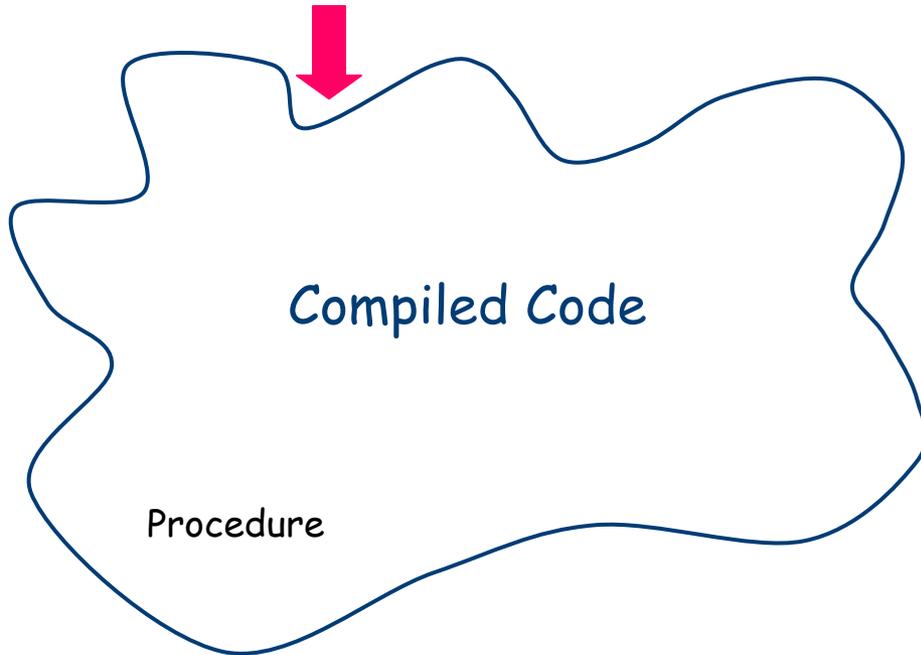
The Procedure & Its Three Abstractions

The compiler produces code for each procedure

Compiled Code

Procedure

The individual code bodies must fit together to form a working program

The Procedure & Its Three Abstractions
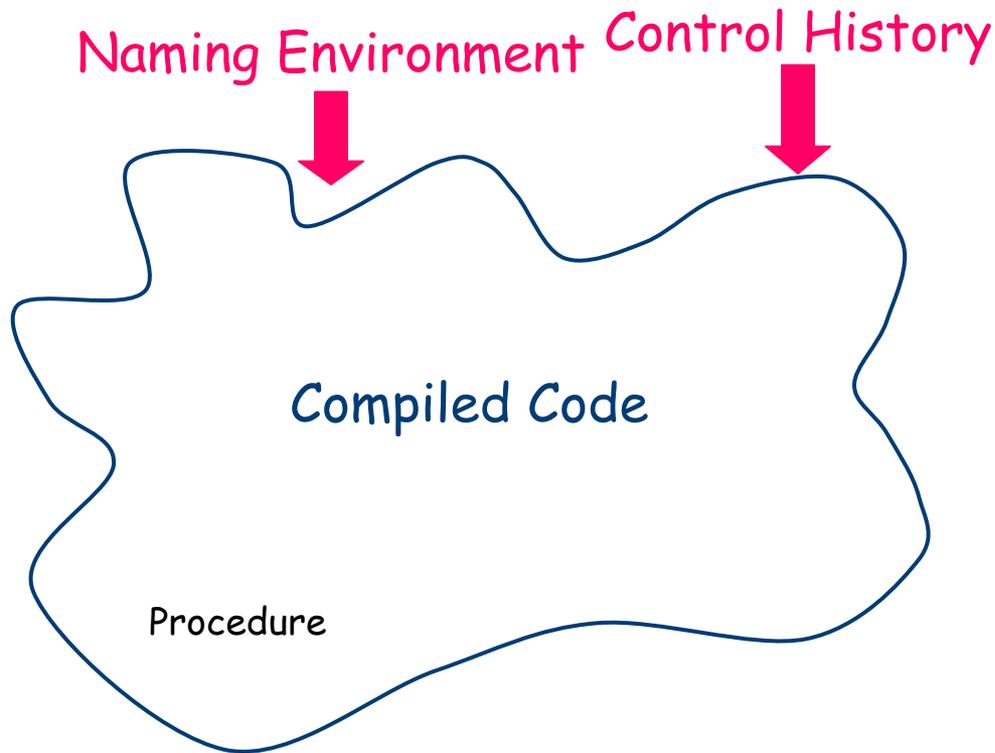
Naming Environment

Compiled Code

Procedure

"Naming" includes the ability to find and access objects in memory

Each procedure inherits a set of names

⇒ Variables, values, procedures, objects, locations, …

⇒ Clean slate for new names, "scoping" can hide other names

The Procedure & Its Three Abstractions

Naming Environment   Control History

Compiled Code

Procedure

Each procedure inherits a control history

⇒ Chain of calls that led to its invocation

⇒ Mechanism to return control to caller

The Procedure & Its Three Abstractions

Naming Environment   Control History

Compiled Code

System Services
(allocation, communication,
I/O, control, naming, …)
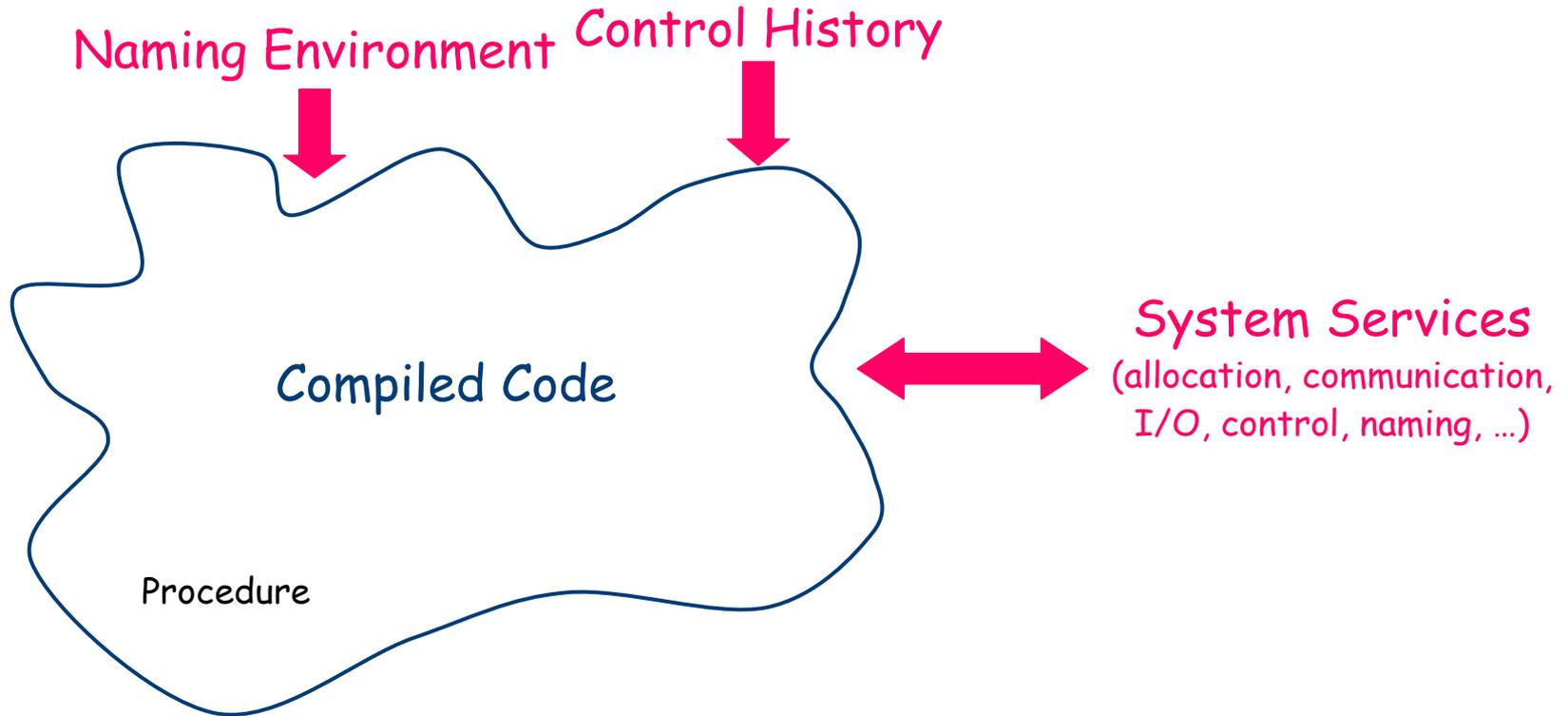
Procedure

Each procedure has access to external interfaces

⇒ Access by name, with parameters

⇒ Protection for both sides of the interface

The Procedure: Three Abstractions

- *Control* Abstraction
  - Well defined entries & exits
  - Mechanism to return control to caller
  - Some notion of parameterization (formal and actual parameters)

- Clean Name Space
  - Clean slate for writing locally visible names
  - Local names may obscure identical, non-local names
  - Local names cannot be seen outside

- External Interface
  - Access is by procedure name & parameters
  - Clear protection for both caller & callee
  - Invoked procedure can ignore calling context

Procedures permit a critical separation of concerns

The Procedure

Procedures allow us to use separate compilation

- Separate compilation allows us to build non-trivial programs

- Keeps compile times reasonable

- Lets multiple programmers collaborate

- Requires independent procedures

Without separate compilation, we would not build large systems

The procedure linkage convention (agreement that defines the actions to take to call a procedure)

- Ensures that each procedure inherits a valid run-time environment and that the callers environment is restored on return
  - The compiler must generate code to ensure this happens according to conventions established by the system

The Procedure   (More Abstract View)

A procedure is an abstract structure constructed via software

Underlying hardware directly supports little of the abstraction—it understands bits, bytes, integers, reals, & addresses, but not:

- Entries and exits

- Interfaces

- Call and return mechanisms

  — Typical machine supports the transfer of control (call and return) but not the rest of the calling sequence   (e.g., preserving context)

- Name space

- Nested scopes

All these are established by carefully-crafted mechanisms provided by compiler, run-time system, linker, loader, and OS;

Run Time versus Compile Time
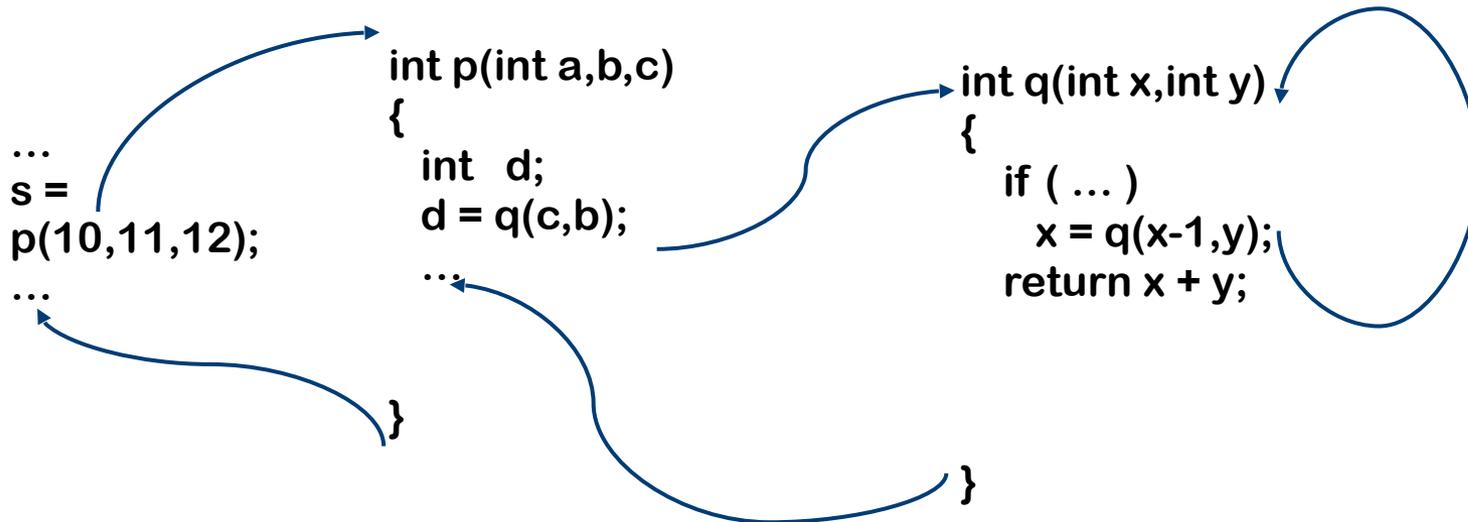
These concepts are often confusing

- Linkages (and code for procedure body) execute at run time

- Code for the linkage is emitted at compile time

- The linkage is designed long before either of these

The Procedure as a *Control Abstraction*

Procedures have well-defined control-flow

Invoked at a call site, with some set of actual parameters

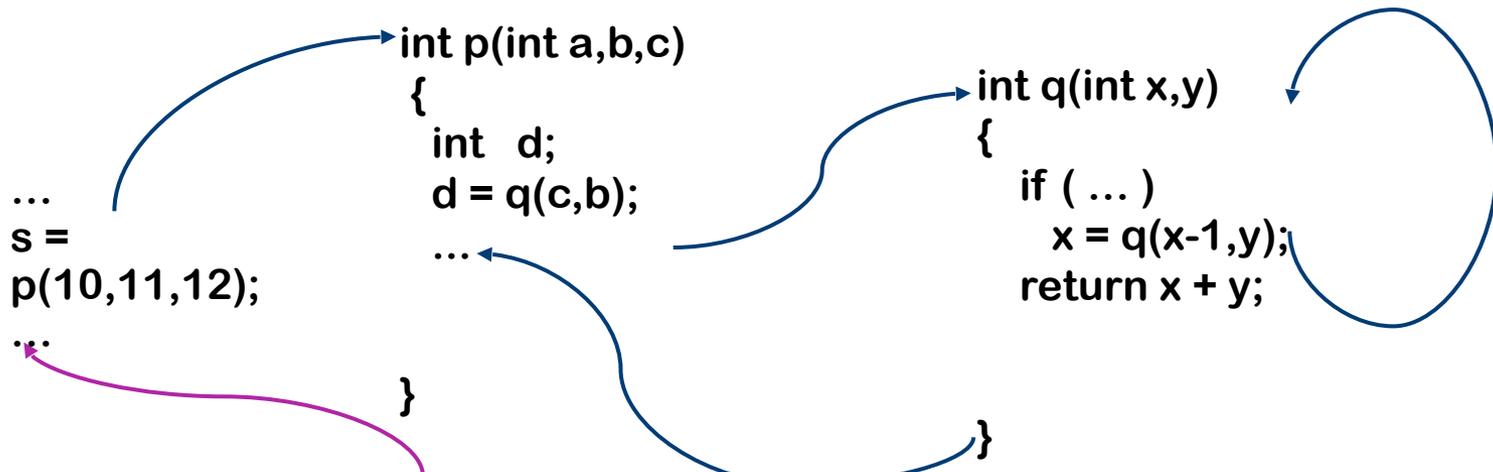- Control returns to call site, immediately after invocation


```
...
s =
p(10,11,12);
...
```

```
int p(int a,b,c)
{
  int  d;
  d = q(c,b);
  ...


}
```

```
int q(int x,int y)
{
  if ( ... )
     x = q(x-1,y);
  return x + y;


}
```

- Most languages allow recursion

The Procedure as a *Control Abstraction*

Implementing procedures with this behavior

- Requires code to save and restore a "return address"

- Must map actual parameters to formal parameters   (c→x, b→y)

- Must create storage for local variables

- p needs space for d

- where does this space go in recursive invocations?

```
int p(int a,b,c)
 {
   int  d;
   d = q(c,b);
...
s =                  ...
p(10,11,12);
...

                 }
```

```
int q(int x,y)
 {
    if ( … )
       x = q(x-1,y);
    return x + y;

            }
```
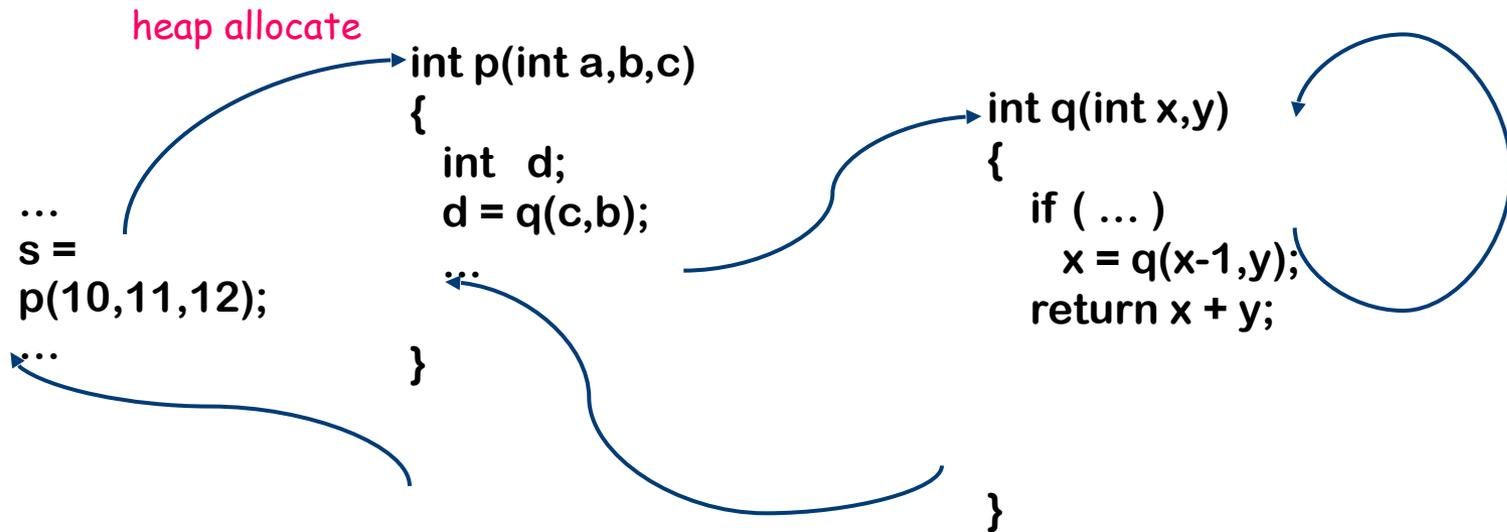
Compiler <u>emits</u> code that causes all this to happen at run time

The Procedure as a *Control Abstraction*

Implementing procedures with this behavior

- Must preserve p's state while q executes
  - recursion causes the real problem here

- Strategy: Create unique location for each procedure activation
  - In simple situations, can use a "stack" of memory blocks to hold local storage and return addresses   closures (procedure+runtime context) ⇒

heap allocate

```
int p(int a,b,c)
{
   int  d;
   d = q(c,b);
   ...

}
```

```
...
s =
p(10,11,12);
...
```

```
int q(int x,y)
{
    if ( … )
       x = q(x-1,y);
    return x + y;

}
```

Compiler <u>emits</u> code that causes all this to happen at run time

The Procedure as a Control Abstraction

In essence, the procedure linkage wraps around the unique code of each procedure to give it a uniform interface
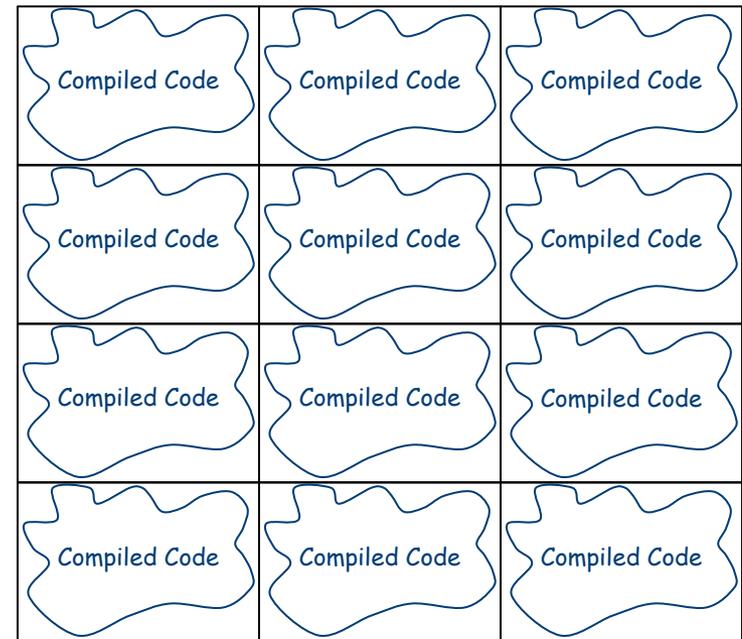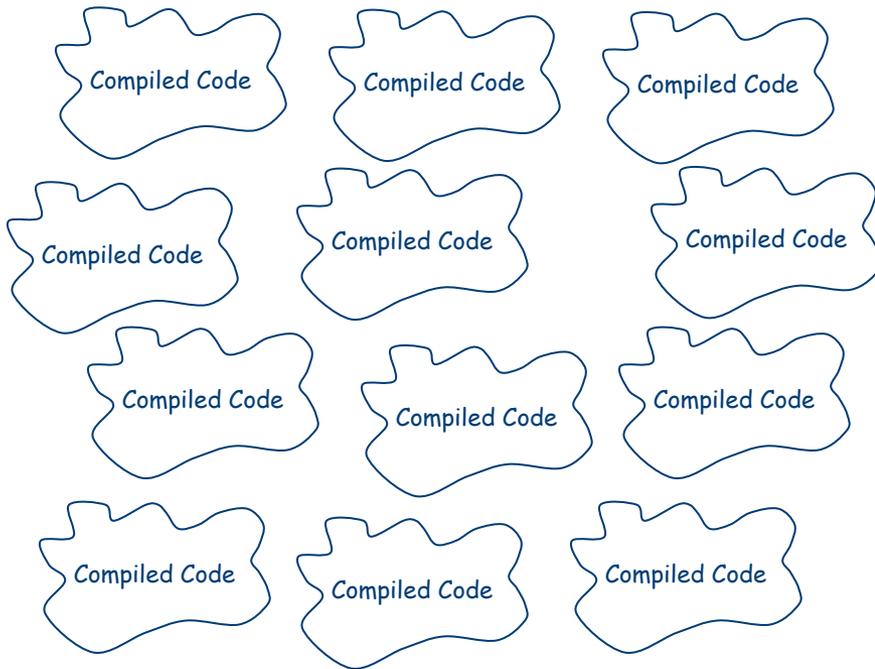
Similar to building a brick wall rather than a rock wall

The Procedure as a Name Space

Each procedure creates its own name space

- Any name can be declared locally

- Local names obscure identical non-local names

- Local names cannot be seen outside the procedure
  — Nested procedures are "inside" by definition

- We call this set of rules & conventions "lexical scoping"

Examples

- C has global, static, local, and block scopes   (Fortran-like)
  — Blocks can be nested, procedures cannot

- Scheme has global, procedure-wide, and nested scopes   (let)

The Procedure as a Name Space

Why introduce lexical scoping?

- Provides a compile-time mechanism for binding "free" variables
- Simplifies rules for naming & resolves conflicts
- Lets the programmer introduce "local" names with impunity

How can the compiler keep track of all those names?

The Problem

- At point p, which declaration of x is current?
- At run-time, where is the value of x that can be used?
- As parser goes in & out of scopes, how does it delete x?

The Answer

- The compiler must model the name space
- Lexically scoped symbol tables

Lexical vs Dynamical scoping

- Lexical scoping: each free variable is bound to the declaration for its name that is lexically closest to the use

  -The declaration always come from a scope that encloses the reference.

- Dynamic scoping: a free variable is bound to the variable by that name that was most recently created at runtime. Example LISP or as possibility Common LISP

Example with lexical scoping

```
procedure p {
     int a, b, c
     procedure q {
          int v, b, x, w
          procedure  r {
               int x, y, z

               ....
          }
          procedure s {
               int x, a, v

               ...
          }
          ... r ... s
     }
     ... q ...
}
```

```
B0: {

              int a, b, c
B1:           {

                   int v, b, x, w
B2:                {

                        int x, y, z

                        ....
                   }
B3:                {

                        int x, a, v

                        ...
                   }
                   ... r ... s
              }
          ... q ...

}
```

Where Do All These Variables Go?

Automatic & Local

- Keep them in the procedure activation record or in a register
- Automatic ⇒ lifetime matches procedure's lifetime

Static

- Procedure scope ⇒ storage area affixed with procedure name
- File scope ⇒ storage area affixed with file name
- Lifetime is entire execution

Global

- One or more named global data areas
- Lifetime is entire execution

Placing Run-time Data Structures

Classic Organization

Single Logical Address Space
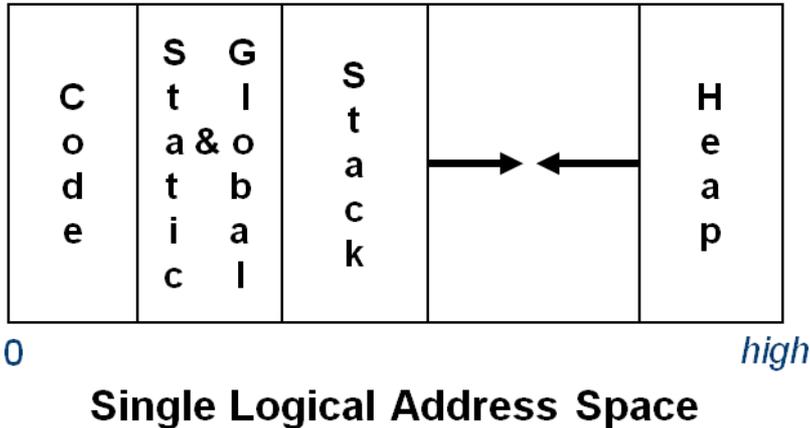
• Better utilization if stack & heap grow toward each other

• Code, static, & global data have known size

  – Use symbolic labels in the code

• Heap & stack both grow & shrink over time

• This is a <u>virtual</u> address space

How Does This Really Work?
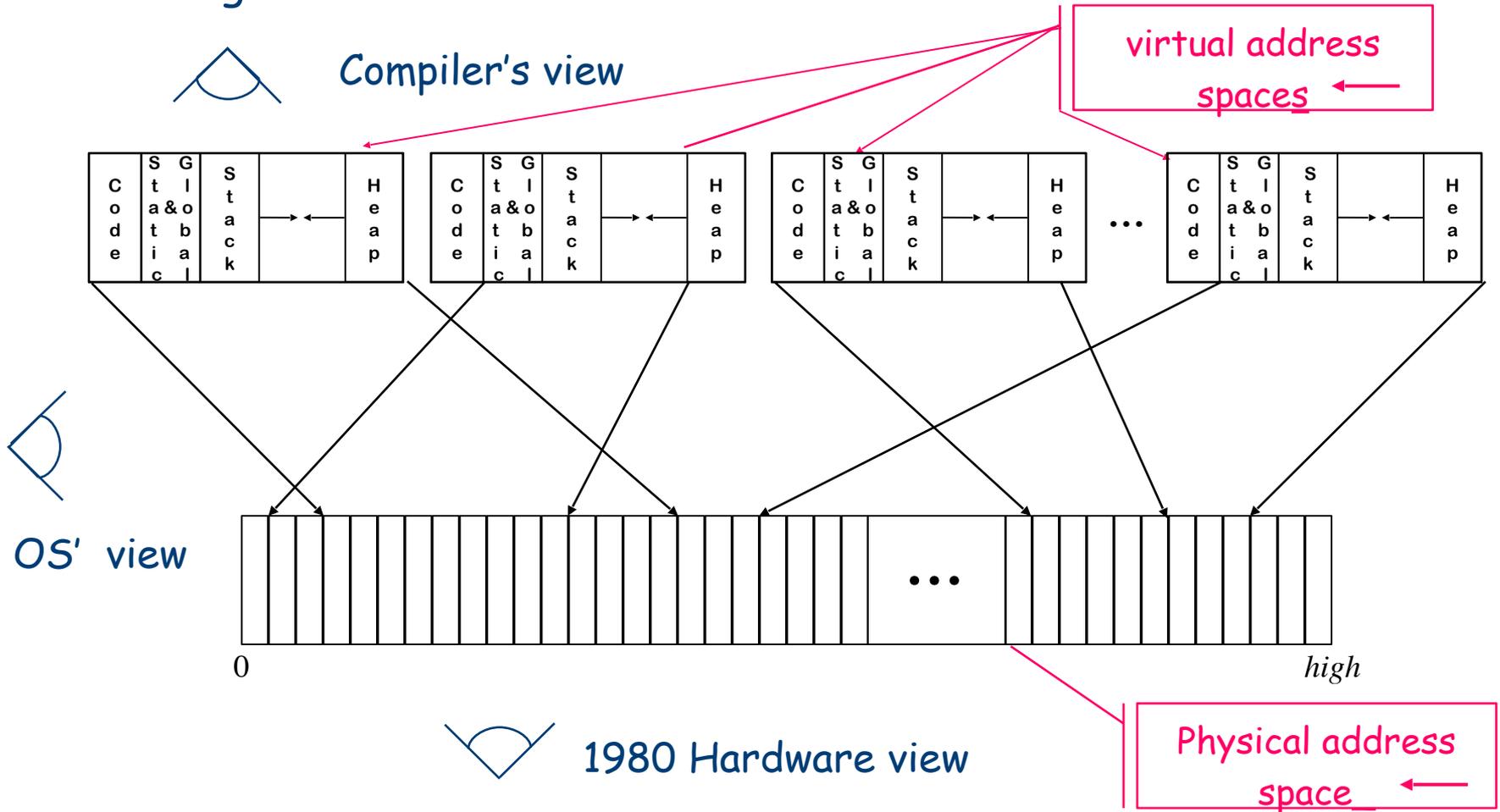
The Big Picture

Compiler's view

virtual address spaces ←

| Code | S t a t i c | G l o b a l & | S t a c k | → ← | H e a p | | Code | S t a t i c | G l o b a l & | S t a c k | → ← | H e a p | | Code | S t a t i c | G l o b a l & | S t a c k | → ← | H e a p | ... | Code | S t a t i c | G l o b a l & | S t a c k | → ← | H e a p |

OS' view

0   ...   *high*

1980 Hardware view

Physical address space__ ←

How Does This Really Work?

Of course, the "Hardware view" is no longer that simple
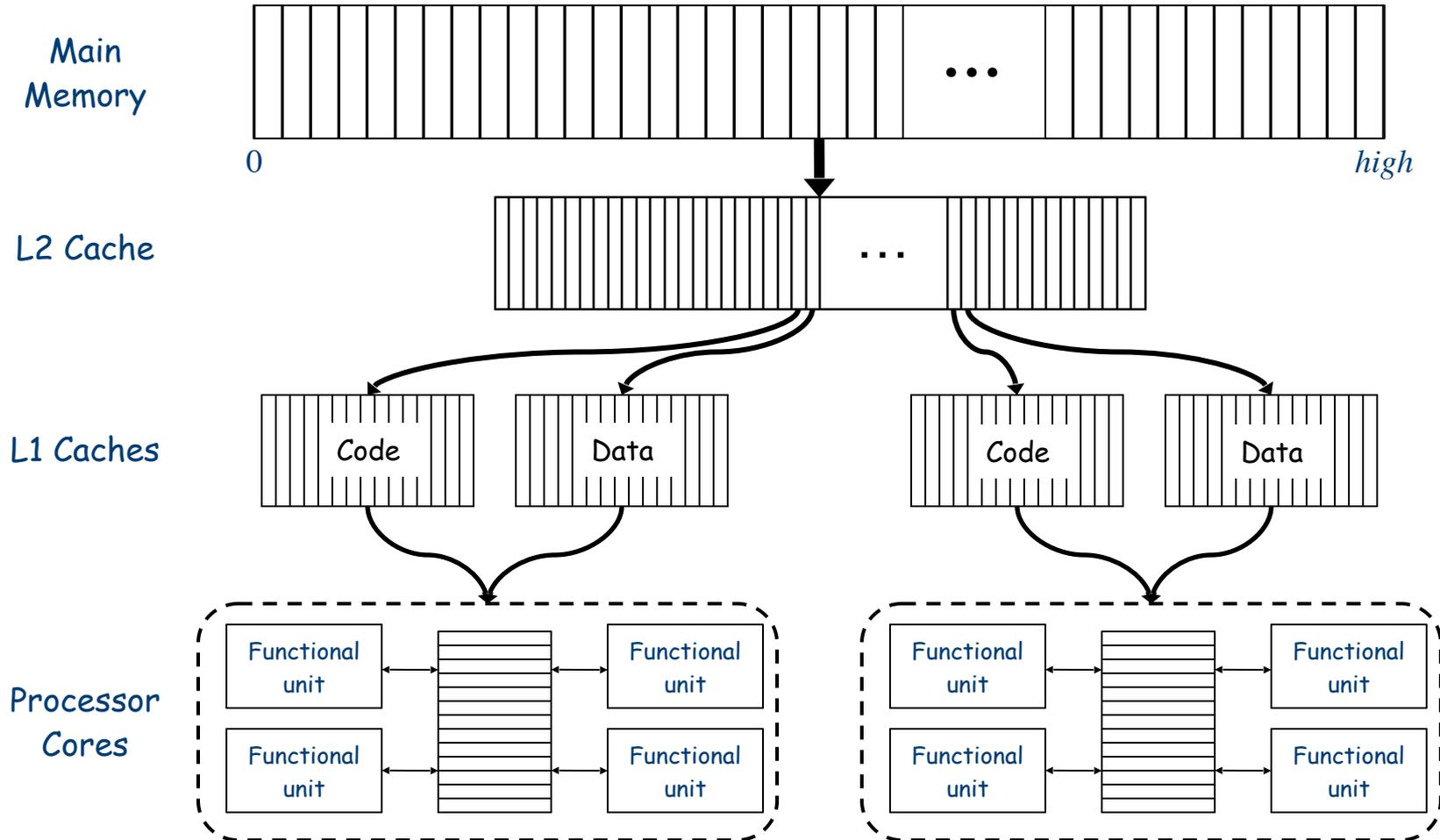
Where Do Local Variables Live?

A Simplistic model

- Allocate a data area for each distinct scope

What about recursion?

- Need a data area per invocation (or activation) of a scope
- We call this the scope's activation record
- The compiler can also store control information there!

More complex scheme

- One activation record (AR) per procedure instance
- All the procedure's scopes share a single AR (may share space)
- Static relationship between scopes in single procedure

Used this way, "static" means knowable at compile time (and, therefore, fixed).
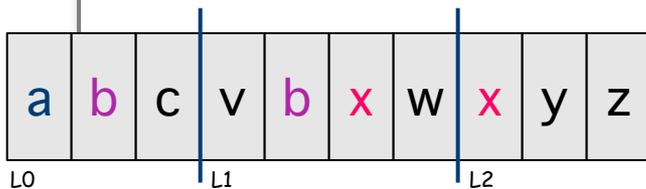
Translating Local Names

How does the compiler represent a specific instance of x ?

- Name is translated into a static coordinate
  - ‹level,offset› pair
  - "level" is lexical nesting level of the procedure
  - "offset" is unique within that scope
- Subsequent code will use the static coordinate to generate addresses and references
- "level" is a function of the table in which x is found
  - Stored in the entry for each x
- "offset" must be assigned and stored in the symbol table
  - Assigned at <u>compile time</u>
  - Known at <u>compile time</u>
  - Used to generate code that <u>executes</u> at <u>run-time</u>

Storage for Blocks within a Single Procedure

```
B0:  {
         int a, b, c
B1:      {
             int v, b, x, w
B2:          {
                 int x, y, z
                 ...
             }
B3:          {
                 int x, a, v
                 ...
             }
             ...
         }
         ...
}
```

a	b	c	v	b	x	w	x	y	z

L0   L1   L2

Storage in block B2

Fixed length data can always be at a constant offset from the beginning of a procedure

— In our example, the a declared at level 0 will always be the first data element, stored at byte 0 in the fixed-length data area

— The x declared at level 1 will always be the sixth data item, stored at byte 20 in the fixed data area

— The x declared at level 2 will always be the eighth data item, stored at byte 28 in the fixed data area

— But what about the a declared in the second block at level 2?

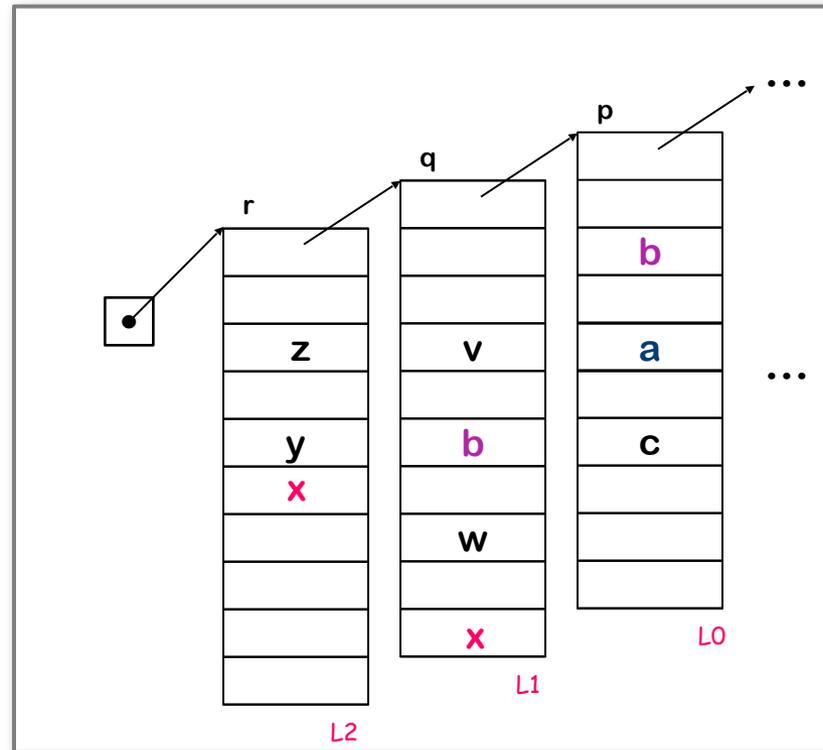# Lexically-scoped Symbol Tables

a | b | c | v | b | x | w | x | y | z

L0   L1   L2
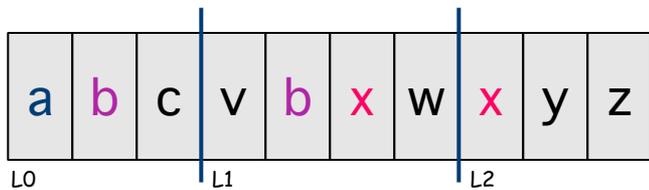
Storage in block B2

High-level idea

- Create a new table for each scope

- Chain them together for lookup

r
z
y
x
L2

q
v
b
w
x
L1

p
b
a
c
L0

...

...

Variable-length Data

```
B0: {   int a, b
        ...
        assign value to a
        ...
B1:     {  int v(a), b, x
           ...
B2:         {  int x, y(8)
               ...
            }
        }
     }
```

Arrays

→ If size is fixed at compile time, store in fixed-length data area

→ If size is variable, store descriptor in fixed length area, with pointer to variable length area

→ Variable-length data area is assigned at the end of the fixed length area for the block in which it is allocated (including all contained blocks)

| a | b | v | b | x | x | y(8) | v(a) |

Includes data for all fixed length objects in all blocks

Variable-length data
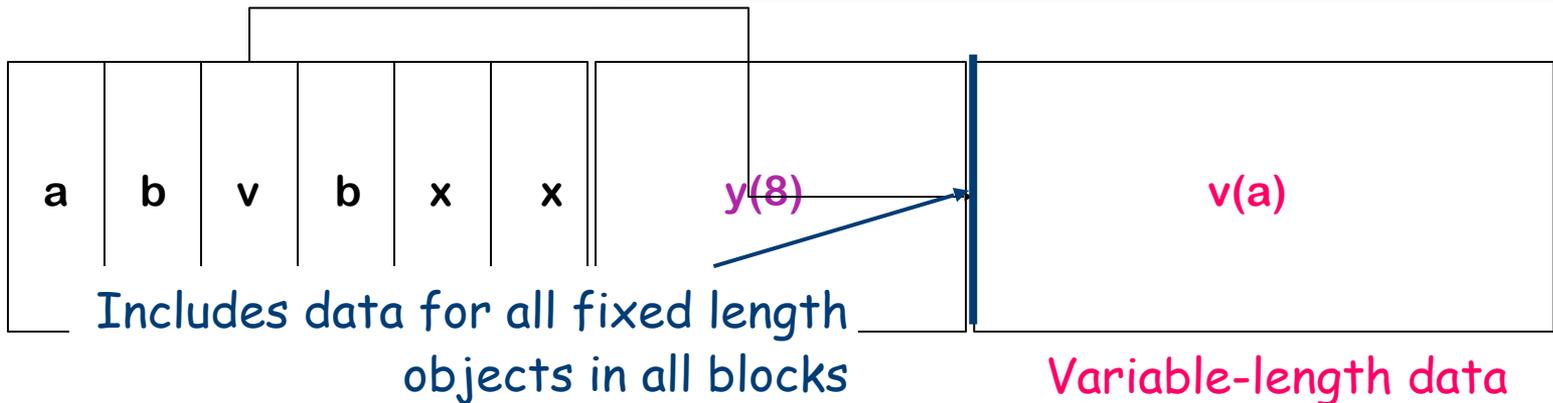
Activation Record Basics

parameters

register save area

ARP ≈ Activation Record Pointer

return value

return address

ARP

access links

in ILOC r_{arp}

caller's ARP

local variables

Space for parameters to the current routine

Saved register contents

If function, space for return value

Address to pass control

Help with non-local access

To restore caller's AR on a return

Space for local values & variables (including spills)

One AR for each invocation of a procedure

Activation Record Details

How does the compiler find the variables?

- They are at known offsets from the AR pointer

- The static coordinate leads to a "loadAI" operation

  — Level specifies an ARP, offset is the constant

Variable-length data

- If AR can be extended, put it above local variables

- Leave a pointer at a known offset from ARP

- Otherwise, put variable-length data on the heap

Initializing local variables

- Must generate explicit code to store the values

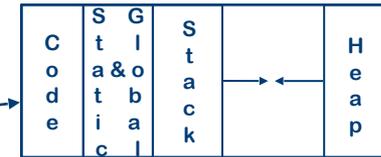- Among the procedure's first actions

Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, AND

- If code normally executes a "return"

⇒ Keep ARs on a stack

C,Pascal

C o d e	S t a t i c	G l o b a l	S t a c k		H e a p

- If a procedure can outlive its caller, OR

- If it can return an object that can reference its execution state

ML

⇒ ARs <u>must</u> be kept in the heap

- If a procedure makes no calls

⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap

Recap

Automatic	⇒	Lifetime matches procedure activation
Static	⇒	Lifetime may be as long as entire execution
Dynamic	⇒	Lifetime is under program control & not known at compile time

Where do variables live?

- Local & automatic ⇒ in procedure's activation record (AR)

- Static (@ any scope) ⇒ in a named static data area

- Dynamic (@ any scope) ⇒ on the heap

Variable length items?

- Put a descriptor in the "natural" location
- Allocate item at end of AR or in the heap

Represent variables by their static coordinates, <level,offset>

- Must map, at runtime, level into a data-area base address
- Must emit, at compile time, code to perform that mapping

Establishing Addressability

Must compute base addresses for each kind of data area

- Local variables
  - — Convert to static data coordinate and use ARP + offset

- Local variables of other procedures
  - — Convert to static coordinates
  - — Find appropriate ARP
  - — Use that ARP + offset

- Global & static variables

Use static coordinates

<l.o>

Must find the right AR

Need links to nameable ARs

Establishing Addressability

Two different ways:
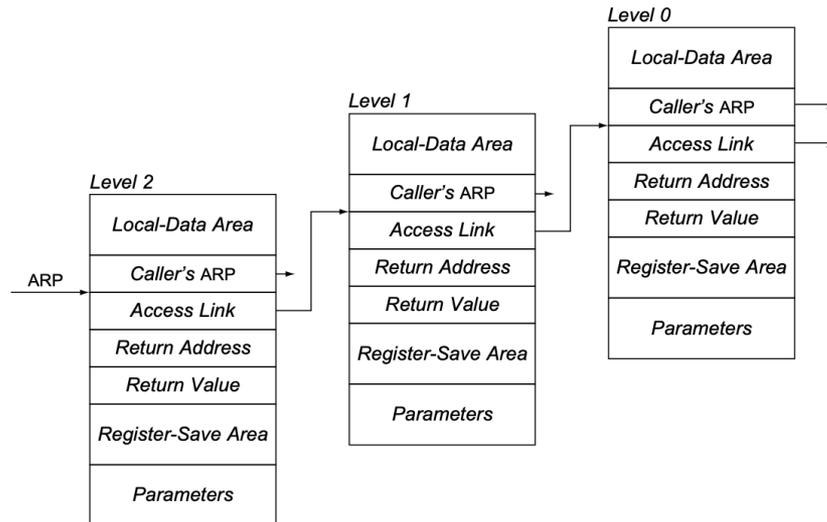
- Using a Address link technique

- Using a Display technique

Establishing Addressability

Using Access Links to Find an ARP for a Non-Local Variable

- Each AR has a pointer to AR of lexical ancestor

- Lexical ancestor need not be the caller

Some setup cost on each call

- Reference to <p,16> runs up access link chain to p
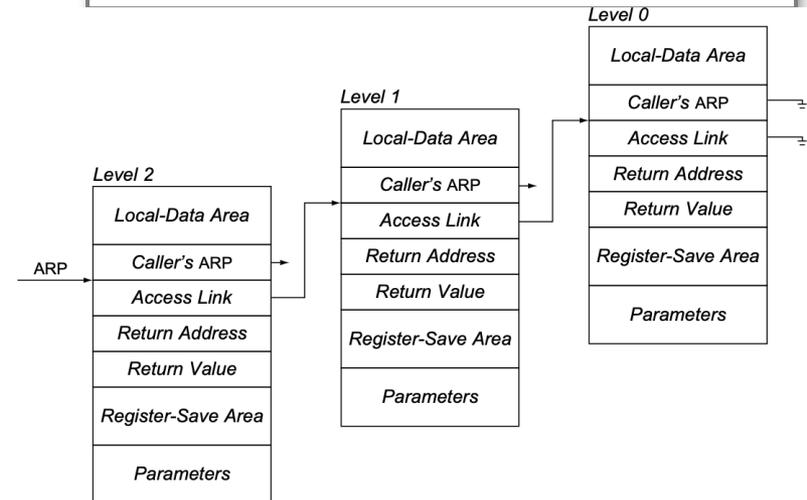
- Cost of access is proportional to lexical distance

Establishing Addressability

Using Access Links

SC	Generated Code
<2,8>	loadAI r_0,8 $\Rightarrow r_{10}$
<1,12>	loadAI r_0,-4 $\Rightarrow r_1$
	loadAI r_1,12 $\Rightarrow r_{10}$
<0,16>	loadAI r_0,-4 $\Rightarrow r_1$
	loadAI r_1,-4 $\Rightarrow r_1$
	loadAI r_1,16 $\Rightarrow r_{10}$

Assume
- Current lexical level is 2
- Access link is at ARP – 4
- ARP is in r_0

Level 0
Local-Data Area
Caller's ARP
Access Link
Return Address
Return Value
Register-Save Area
Parameters

Level 1
Local-Data Area
Caller's ARP
Access Link
Return Address
Return Value
Register-Save Area
Parameters

Level 2
Local-Data Area
Caller's ARP
Access Link
Return Address
Return Value
Register-Save Area
Parameters

ARP

Access cost varies with level

All accesses are relative to ARP (r_0)

Maintain access links

The compiler must add code to each procedure call that finds the appropriate ARP and stores in the activation record of the callee (at the position reserved for the access link)

Maintaining access link

For a caller at level p and a callee is defined at level q

- q=p+1 the callee is nested inside the caller
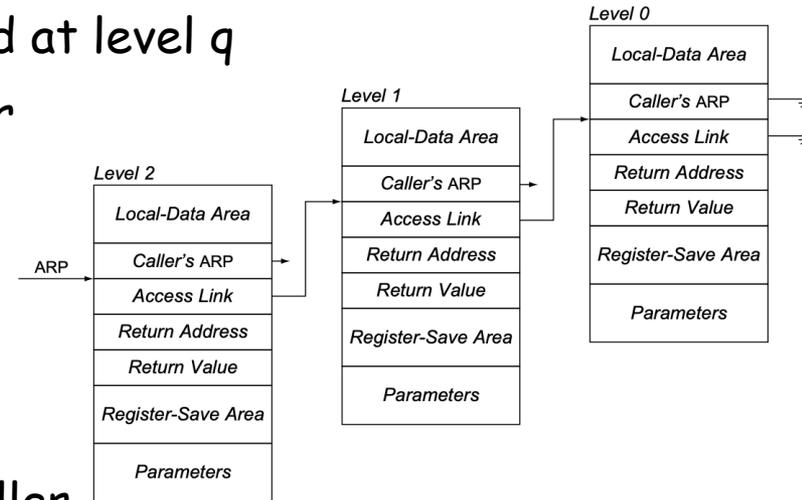   - → Callee use the current ARP as link

- q=p
    - → Callee copy the access link of the caller

- q<p

→ Find ARP for level q −1

→ Use that ARP as link

Cost of maintenance is proportional to lexical distance

The static and call chain do not coincide!

```
procedure main {
    procedure p1 { ... }
    procedure p2 {
        procedure q1 { ... }
        procedure q2 {
            procedure r1 { ... }
            procedure r2 {
                call p1; ... // call up from level 3 to level 1
            } // end of  r2
            call r2;     // call down from level 2 to level 3
        } //end of  q2
        call q2;          // call down from level 1 to level 2
    }    //end of  p2
    call p2;             // call down from level 0 to level 1
} // end of main
```
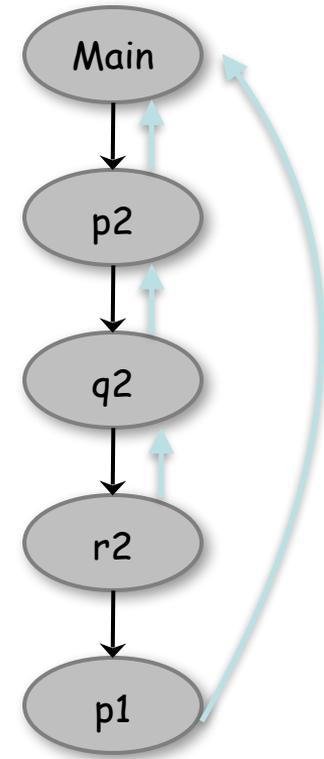

Call History
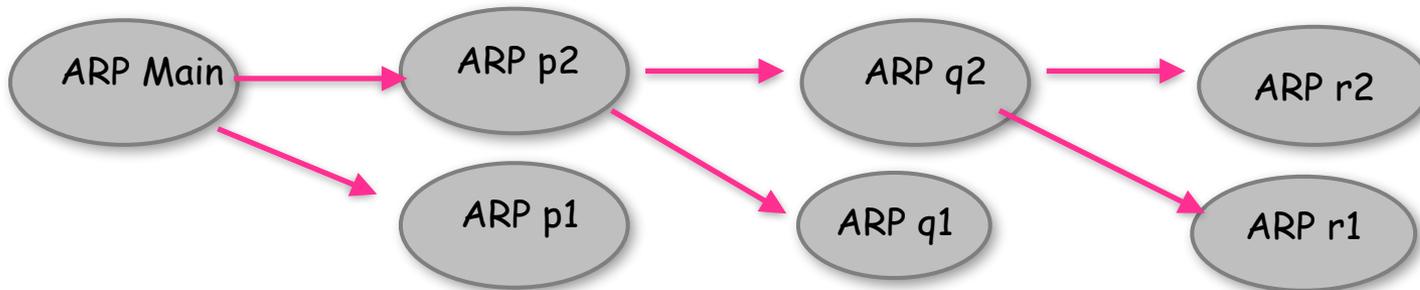
Establishing Addressability

Using a Display to Find an ARP for a Non-Local Variable
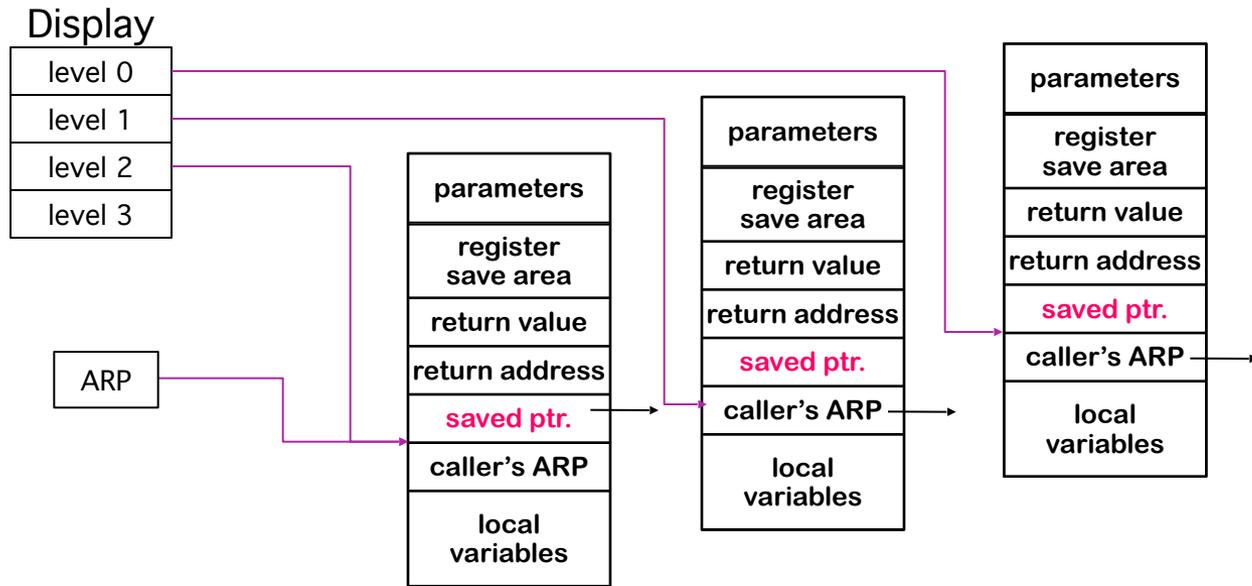
- Global array of pointer to nameable ARs
- Needed ARP is an array access away

Some setup
cost
on each call

- Reference to <p,16> looks up p's ARP in display & adds 16
- Cost of access is constant   (ARP + offset)

Establishing Addressability

Using a Display

SC	Generated Code	
‹2,8›	loadAI r_0,8	$\Rightarrow r_{10}$
‹1,12›	loadI _disp	$\Rightarrow r_1$
	loadAI r_1,4	$\Rightarrow r_1$
	loadAI r_1,12	$\Rightarrow r_{10}$
‹0,16›	loadI _disp	$\Rightarrow r_1$
	loadAI r_1,0	$\Rightarrow r_1$
	loadAI r_1,16	$\Rightarrow r_{10}$

Access costs are fixed

Address of display may consume a register

Assume

- Current lexical level is 2
- Display is at label _disp

Desired AR is at _disp + 4 x level

Display

level 0
level 1
level 2
level 3

Maintaining Display

Display

level 0
level 1
level 2
level 3

ARP

parameters
register save area
return value
return address
saved ptr.
caller's ARP
local variables

parameters
register save area
return value
return address
saved ptr.
caller's ARP
local variables

parameters
register save area
return value
return address
saved ptr.
caller's ARP
local variables

Maintaining access links

- On entry to level j
→ Save level j entry into AR (saved ptr field)
→ Store ARP in level j slot
- On exit from level j
→ Restore old level j entry

```
procedure main {
    procedure p1 { … }
    procedure p2 {
        procedure q1 { … }
        procedure q2 {
            procedure r1 { … }
            procedure r2 {
                call p1; … // call up from level 3 to level 1
            } // end of  r2
            call r2;      // call down from level 2 to level 3
        }  //end of  q2
        call q2;        // call down from level 1 to level 2
    }   //end of  p2
    call p2;          // call down from level 0 to level 1
}  // end of main
```

0	ARP main
1	ARP q2
2	ARP p2
3	ARP r2

0	ARP main
1	ARP p1

0	ARP main
1	ARP p1

```
procedure main {
    procedure p1 { ... }
    procedure p2 {
        procedure q1 { ... }
        procedure q2 {
            procedure r1 { ... }
            procedure r2 {
                call p1; ... // call up from level 3 to level 1
            } // end of  r2
            call r2;     // call down from level 2 to level 3
        } //end of  q2
        call q2;          // call down from level 1 to level 2
    }   //end of  p2
    call p2;              // call down from level 0 to level 1
}  // end of main
```

Establishing Addressability

Access Links Versus Display

- Each adds some overhead to each call

- Access links costs vary with level of reference
  - Overhead only incurred on references & calls
  - If ARs outlive the procedure, access links still work

- Display costs are fixed for all references
  - References & calls must load display address
  - Typically, this requires a register
  - Depends on ratio of non-local accesses to calls

For either scheme to work, the compiler must
insert code into each procedure call & return

Creating and Destroying Activation Records

All three parts of the procedure abstraction leave state
in the activation record

- How are ARs created and destroyed
  - Procedure call must allocate & initialize   (preserve caller's world)
  - Return must dismantle environment   (and restore caller's world)

- Caller & callee must collaborate on the problem
  - Caller alone knows some of the necessary state
    - → Return address, parameter values, access to other scopes
  - Callee alone knows the rest
    - → Size of local data area, registers it will use

Their collaboration takes the form of a <u>linkage convention</u>

Procedure Linkages

How do procedure calls actually work?

At compile time, callee may not be available for inspection

- Different calls may be in different compilation units
- All calls must use the same protocol

Compiler must use a standard sequence of operations

- Enforces control & data abstractions
- Divides responsibility between caller & callee

Usually a system-wide agreement, to allow interoperability

Saving Registers

Who saves the registers? Caller or callee?

- Arguments for saving on each side of the call
  - Caller knows which values are LIVE across the call
  - Callee knows which registers it will use

- Conventional wisdom: divide registers into three sets
  - Caller saves registers
    - → Caller targets values that short-LIVED value across the call
  - Callee saves registers
    - → Callee only uses these AFTER filling caller saved registers
  - Registers reserved for the linkage convention
    - → ARP, return address (if in a register), …

Where are they stored? In one of the ARs …

Procedure Linkages

Standard Procedure Linkage

procedure p

prolog
pre-call
post-return
epilog

procedure q
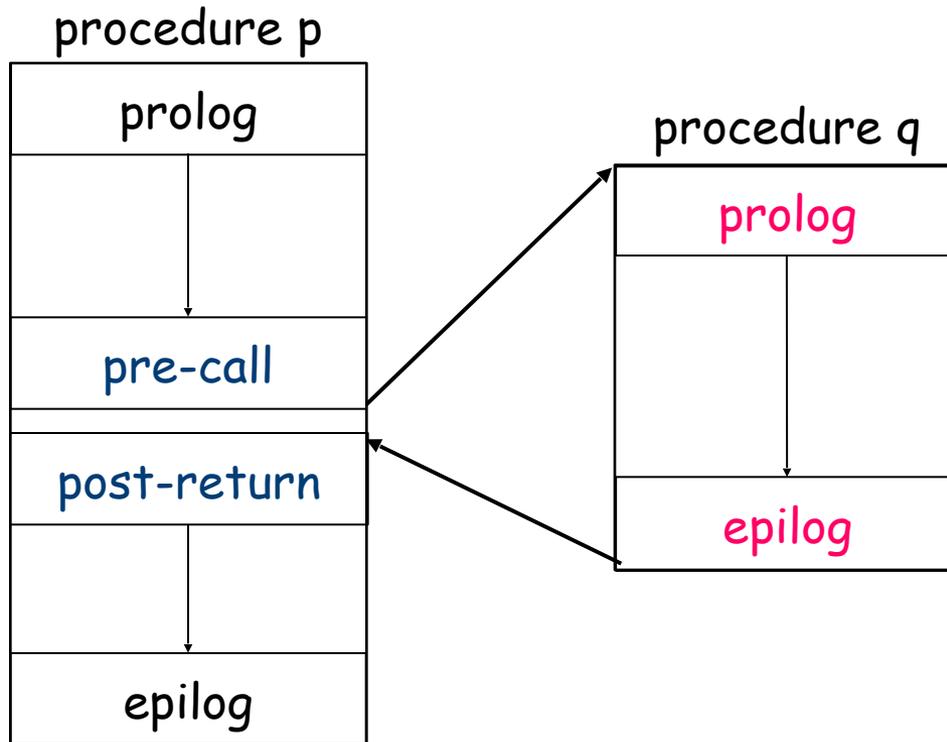
prolog
epilog

Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence

These are completely predictable from the call site ⟹ depend on the number & type of the actual parameters

Procedure Linkages

Pre-call Sequence

- **Starts** setting up callee's basic environment
- Evaluates formal parameters

The Details

- Allocate space for the callee's AR
- Evaluates each parameter & stores value or address
- Saves return address: caller's ARP into callee's AR
- If access links are used
  — Find appropriate lexical ancestor & copy into callee's AR
- Save any caller-save registers
  — Save into space in caller's AR
- Jump to address of callee's prolog code

Procedure Linkages

Post-return Sequence

- Undo the actions of the precall sequence

- Place any value back where it belongs

The Details

- Free the callee's AR

- Restore any caller-saved registers

- Restore any call-by-reference parameters to registers, if needed
  - Also copy back call-by-value/result parameters

- Continue execution after the call

Procedure Linkages

- Finish setting up callee's environment
- Preserve parts of caller's environment that will be disturbed

The Details

- Preserve any callee-saved registers
- If display is being used
  — Save display entry for current lexical level
  — Store current ARP into display for current lexical level
- Allocate space for local data
- Handle any local variable initializations

Procedure Linkages

Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

The Details

- Store return value
- Restore callee-saved registers
- Free space for local data, if necessary
- Load return address from AR
- Restore caller's ARP
- Jump to the return address

How is it realised? It depends on where the AR are...

If activation records are stored on the stack

- Easy to extend — simply bump top of stack pointer

- Caller & callee share responsibility
  - Caller can push parameters, space for registers, return value slot, return address, addressability info, & its own ARP
  - Callee can push space for local variables (fixed & variable size)

If activation records are stored on the heap

- Hard to extend

- Several options
  - Caller passes everything in registers; callee allocates & fills AR
  - Store parameters, return address, etc., in caller's AR !
  - Store callee's AR size in a defined static constant

Without recursion, activation records can be static

Communicating Between Procedures
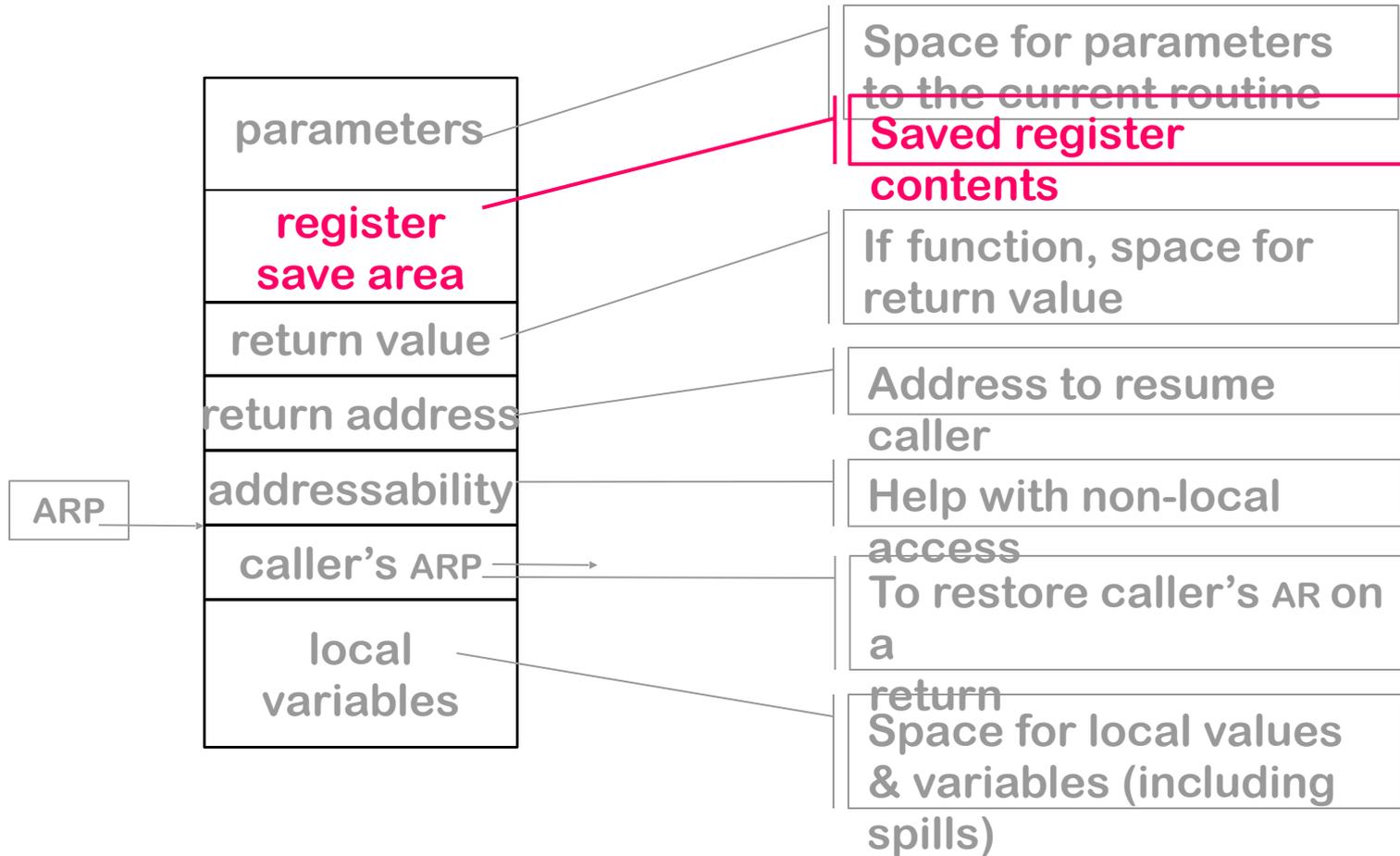
Most languages provide a parameter passing mechanism

⇒ Expression used at "call site" becomes variable in callee

Two common binding mechanisms

- Call-by-reference passes a pointer to actual parameter
  — Requires slot in the AR (for address of parameter)
  — Multiple names with the same address

- Call-by-value passes a copy of its value at time of call
  — Requires slot in the AR
  — Each name gets a unique location   (may have same value)
  — Arrays are mostly passed by reference, not value

call
fee(x,x,x);

Remember This Drawing?

parameters
register save area
return value
return address
addressability
caller's ARP
local variables

ARP →

Space for parameters to the current routine

Saved register contents

If function, space for return value

Address to resume caller

Help with non-local access

To restore caller's AR on a return

Space for local values & variables (including spills)

ARP ≈ Activation Record Pointer