

# Computer Science 160

## Translation of Programming Languages

Instructor: Christopher Kruegel

---

# Procedures

# The Procedure Abstraction

UC Santa Barbara

---

- What is a Procedure:  
An abstraction (illusion) provided by programming languages
  - What does the idea of procedure provide?
    - Control abstraction: transfer of control between caller and callee
    - Namespace (scope): caller and callee have different namespaces
    - External interface: passing arguments between caller and callee
  - Why have procedures?
    - establishes a separation of concerns (an important software engineering principle)
    - enables modular design
    - enables separate compilation
-

# Why do should I care about procedures?

UC Santa Barbara

---

- This is a compilers class, not a programming languages class, right?
    - A programming language has the notion of procedures
    - The architecture (or whatever is going to run our output code) does not have any notion of a procedure, of control abstraction, or even of separate name spaces
    - It is up to the compiler to take the input program (that has procedures) and translate it into output code that the runs exactly the same (even if the underlying machine has no notion of a procedure)
    - In essence, the compiler has to implement procedures. So, we better know what it is that we have to implement
-

# The Procedure Abstraction

UC Santa Barbara

---

- There are three major things that procedures provide
  - Control abstraction
    - Well defined entries and exits
    - Mechanism to return control to caller
    - Mapping a set of parameters (arguments) from caller's name space to callee's name space
  - Name space (scope)
    - Declare locally visible names
    - Local names may hide identical, non-local names
    - Local names cannot be seen outside
  - External Interface
    - Access is by procedure name and parameters
    - Protection for both caller and callee
-

# The Procedure Abstraction

UC Santa Barbara

---

Procedures are the key to building large systems

- Requires system-wide cooperation
    - Broad agreement on memory layout, calling sequences
    - Involves architecture, operating system, and compiler
  - Establishes the need for a private context
    - Create private storage for each procedure invocation
    - Encapsulate information about control flow and data abstractions
  - Provides shared access to system-wide facilities
    - Storage management
-

# The Procedure Abstraction

UC Santa Barbara

---

Procedures allow us to use separate compilation

- Separate compilation allows us to build non-trivial programs
- Keeps compile times reasonable
- Lets multiple programmers collaborate
- Without separate compilation, we would not be able to build large systems

The *procedure linkage convention*

- Ensures that each procedure inherits a valid run-time environment and that it restores one for its caller
  - Highly machine dependent
  - Establishes division of responsibility and separation of concerns
-

# The Procedure Abstraction

UC Santa Barbara

---

- Underlying hardware supports little of this abstraction
  - Procedures create named variables – Hardware understands a linear array of memory locations (one big array)
  - Procedures have nested scopes – Hardware understands a linear array of memory locations (one big array)
  - Procedures require call/return mechanisms – Hardware simply advances its program counter through some sequence of instructions or branches (like a GOTO statement)
- Compiler translates procedure abstraction to something that hardware can understand (with some help from operating system and little help from hardware)

*One view is that computer science is simply the art of realizing successive layers of abstraction*

---

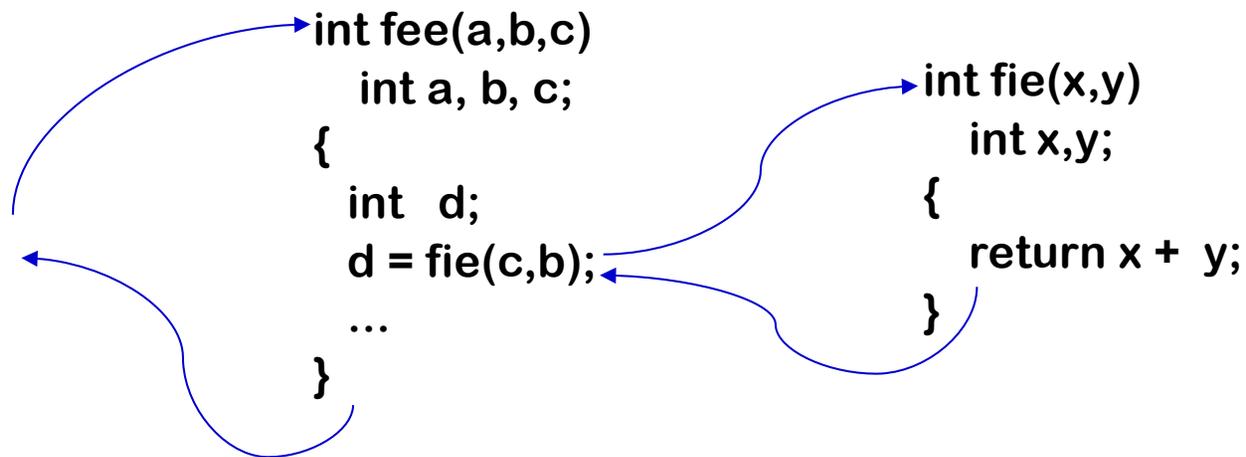
# The Procedure as a Control Abstraction

UC Santa Barbara

Procedures have well-defined control-flow behavior

The Algol-60 procedure call

- Invoked at a call site, with some set of *actual parameters*
- Control returns to call site, immediately after invocation



- Most languages allow recursion (Fortran does not)

# The Procedure as a Control Abstraction

UC Santa Barbara

---

Implementing procedures with this behavior

- Requires code to save a “return address” and use it at “return”
- Must create unique location for each procedure activation
  - Can use a “stack” of return addresses
  - Sometimes, this stack is implemented in hardware
- Need to map *actual parameters* to *formal parameters* ( $c \rightarrow x, b \rightarrow y$ )
- Must create storage for local variables (and, maybe, parameters)
  - fee needs space for d (and, maybe, a, b, c)
- Must preserve fee’ s state while fie executes

*Compiler must emit code that causes all this to happen at run-time*

---

# The Procedure as a Name Space

UC Santa Barbara

---

Each procedure creates its own name space (scope)

- Names can be declared locally
- Local names hide identical -- but non-local -- names
- Local names cannot be seen outside the procedure
  - Nested procedures are “inside” by definition
- We call this set of rules and conventions “lexical scoping”

Examples

- C has global, static, local, and block scopes
    - Blocks can be nested, procedures cannot
  - Pascal has nested procedures (CSimple as well)
-

# Same Names in Programs

UC Santa Barbara

- Multiple **variables** can have the **same names** in a program

```
int a = 4;
{
    int a = 3;
    int b = 3;
    print(a, b);
}
print(a);
```

```
int add(int x, int y)
{return x + y;}
int subtract(int x, int y)
{return x - y;}
```

- Multiple **functions** can have the **same names** in a program.
  - e.g., function overloading in C++

```
void print(int i) {cout << " Here is int " << i << endl; }
void print(double f) { cout << " Here is float " << f << endl;}
void print(char const *c) {cout << " Here is char* " << c << endl;}
```

# Name Space (Scope)

UC Santa Barbara

- **The scope of a declaration**
  - The part of a program where the name of a declaration is visible
  - This means it is only legal to refer to the identifier within its scope. Here, identifier refers to function or variable name.

```
int a = 4;
{
    int a = 3;
    int b = 3;
    print(a, b);
}
print(a, b);
```

Any error?

# Nested Block and Inheritance

UC Santa Barbara

- Block:
  - a syntactic unit with declarations and statements
  - may require memory allocation during execution (once or several times)
- Block structure (nesting):
  - a block can have inner blocks (recursively)
  - declarations in a block are visible also in the inner blocks

```
int a = 4;
{
    int b = 3;
    print(a, b);
}
```

```
int a = 4;
{
    int a = 3;
    int b = 3;
    print(a, b);
}
```

- **Shadowing** occurs when an identifier declared within a given scope has the same name as an identifier declared in an outer scope.

# Lexical Scoping vs. Dynamic Scoping

UC Santa Barbara

---

- Lexical Scoping: Scoping is determined by the program text (static)
    - The scope of a declaration in a block  $B$  includes  $B$
    - If a variable  $x$  is not declared in  $B$ , then occurrence of  $x$  in  $B$  is in the scope of the declaration of  $x$  in enclosing block  $B'$  if
      - $B'$  has a declaration of  $x$
      - $B'$  is more closely nested around  $B$  than any other block with declaration of  $x$
    - Lexical scoping is used in Algol-like languages such as Pascal, C
  - Dynamic Scoping: Scoping is determined by the run-time behavior
    - A variable that is not declared in the current scope is bound to the variable by that name that was most recently created at *run-time*
    - Dynamic scoping is used in some forms of LISP
-

# Example: Lexical vs. Dynamic Scoping

UC Santa Barbara

```
program dynamic(input, output)
  var r : real;
  procedure show;
    begin write( r ) end;
  procedure small;
    var r : real;
    begin r := 0.125; show end;
begin
  r := 0.25
  show; small; writeln;
  show; small; writeln
end.
```

With lexical scoping, the output is:

```
0.250 0.250
0.250 0.250
```

If dynamic scoping was used, then the output will be:

```
0.250 0.125
0.250 0.125
```

# Scoping Rules for C

UC Santa Barbara

---

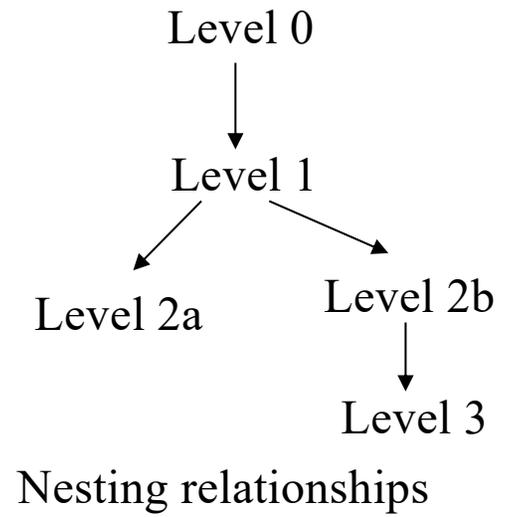
- A name can have global scope, all declarations of the same global name refer to a single instance.
  - A name can have file-wide scope (`static`), such names are visible to every procedure in the file containing the declaration.
  - A name can be declared locally in a procedure, then the scope of the name is the procedure body.
  - A name can be declared within a block (denoted by a pair of curly braces). Such a variable is only visible inside the declaring block.
-

# Example in C

UC Santa Barbara

```
static int w;                /* level 0 */
int x;
void example (int a, int b); /* level 1 */
{
    int c;
    {
        int b, z;           /* level 2a */
        ...
    }
    {
        int a, x;           /* level 2b */
        ...
        {
            int c, x;       /* level 3 */
            b = a + b + c + x;
        }
    }
}
```

Level	Names
0	w, x, example
1	a, b, c
2a	b, z
2b	a, x
3	c, x



At any scope level, a variable is bound to the declaration that is in that scope level. If there is no declaration at that scope level, then it is bound to the declaration that is in its closest ancestor

# Pascal: Scoping

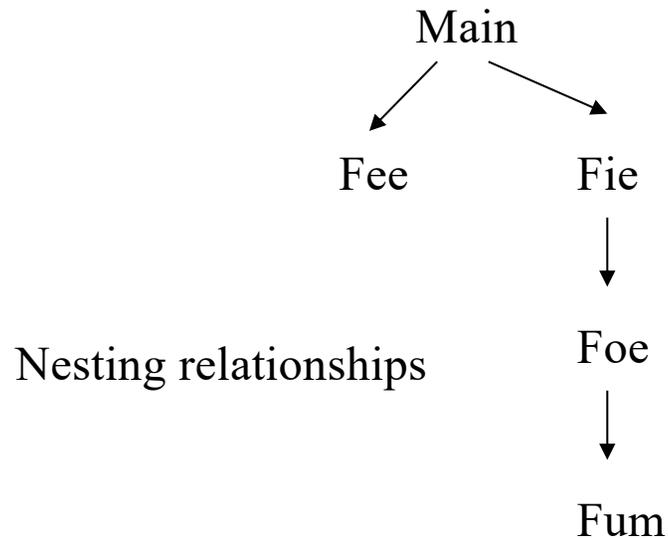
UC Santa Barbara

```

program Main(input, output);
  var x, y, z: integer;
  procedure Fee;
    var x: integer;
    begin { Fee }
      x := 1;
      y := x * 2 + 1
    end;
  procedure Fie;
    var y: real;
    procedure Foe;
      var z: real;
      procedure Fum;
        var y: real;
        begin { Fum }
          x := 1.25 * z;
          Fee;
        end;
      begin { Foe }
        z := 1;
        Fee;
        Fum
      end;
    begin { Fie }
      Foe;
    end;
  begin { Main }
    x := 0;
    Fie
  end.
  
```

Scope	x	y	z
Main	Main.x	Main.y	Main.z
Fee	Fee.x	Main.y	Main.z
Fie	Main.x	Fie.y	Main.z
Foe	Main.x	Fie.y	Foe.z
Fum	Main.x	Fum.y	Foe.z

Main.z

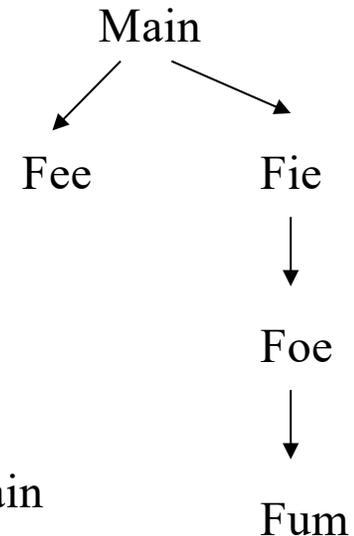


# Pascal: Scoping

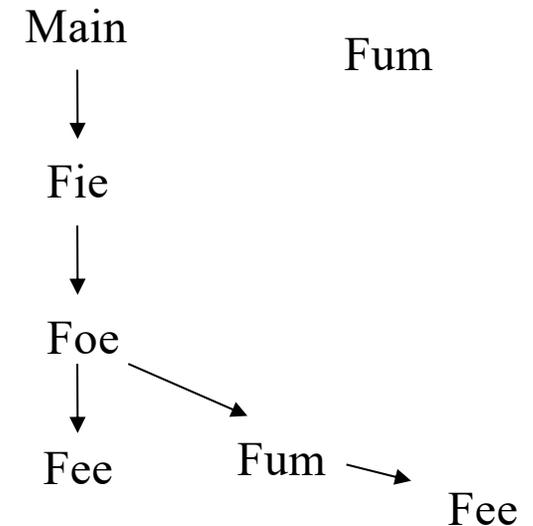
UC Santa Barbara

```
program Main(input, output);
  var x, y, z: integer;
  procedure Fee;
    var x: integer;
    begin { Fee }
      x := 1;
      y := x * 2 + 1
    end;
  procedure Fie;
    var y: real;
    procedure Foe;
      var z: real;
      procedure Fum;
        var y: real;
        begin { Fum }
          x := 1.25 * z;
          Fee;
        end;
      begin { Foe }
        z := 1;
        Fee;
        Fum
      end;
    begin { Fie }
      Foe;
    end;
  begin { Main }
    x := 0;
    Fie
  end.
```

Nesting relationships  
(static, lexical scoping)



**Activation Tree** shows the transfer of control between different procedures  
(dynamic scoping uses this)



# Scoping Rules

UC Santa Barbara

---

- Java
    - public classes are in the global scope
    - a class contains fields (data) and methods (procedures),
    - methods and fields in public classes can be declared public which makes them accessible to methods in classes of other packages
    - fields and methods in a class are accessible to methods in all other classes of the same package, unless they are explicitly declared private
    - classes can nest inside other classes
-

# Scoping Rules

UC Santa Barbara

---

- Fortran
    - a global scope that holds procedure names and common block names (correspond to global variables)
    - a series of local scopes, one per procedure
    - a local variable has a lifetime that matches the invocation of the procedure (but programmers can override this rule using save statement which preserves the value of the variable across procedure calls)
-

# Illegal Shadowing

UC Santa Barbara

- In most languages, it is illegal to declare two identifiers with the same name if they are in the same scope. Here identifiers refer to a function or variable name.
- In some languages (e.g., Java), it is illegal to shadow local variables.

```
public static void foo () {  
    int i;  
    for (int i = 0; i < 5; i++) // illegal to redeclare i  
        System.out.println(i);  
}
```

- Making this illegal help prevent potential bugs

# The Procedure as a Name Space

UC Santa Barbara

---

Why introduce lexical scoping?

- Provides a mechanism for binding “free” variables
- Simplifies rules for naming and resolves conflicts

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  $x$  found?
- As parser goes in and out of scopes, how does it delete  $x$ ?

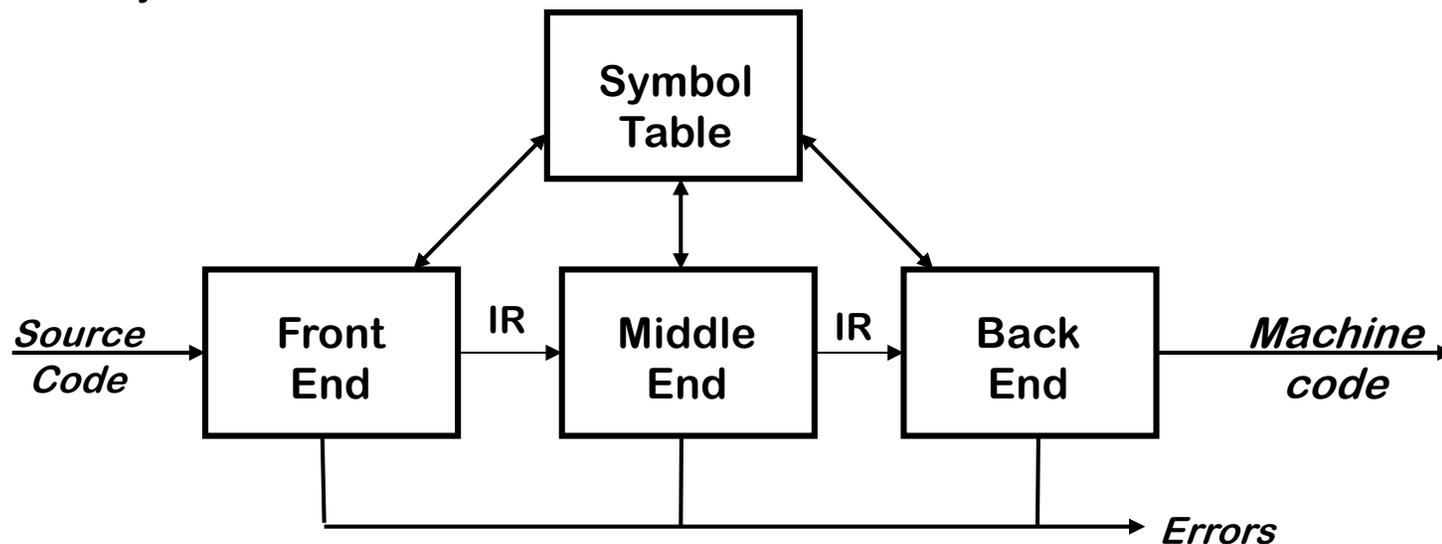
The Answer

- Use lexically-scoped symbol tables
-

# Symbol Tables

UC Santa Barbara

- How can we manage these name spaces at compile-time?
  - Symbol Tables



- Symbol table is used to keep information about names
  - type, scope, memory location, ...
- We can rediscover this information every time we need it by traversing the intermediate representation (AST), but using a symbol table is more efficient

# Symbol Tables

UC Santa Barbara

---

- What are the entries in the symbol table?
    - variable names
    - procedure and function names
    - defined constants
    - labels
    - keywords (may also be entered if the lexical analyzer does not take care of them)
-

# Symbol Tables

UC Santa Barbara

---

- What information should be kept?
    - textual name
    - data type
    - for arrays dimension, upper and lower bounds for each dimension
    - for records or structures list of fields and their information
    - for functions and procedures number and type of arguments
    - storage information (memory location: base address and offset)
    - scope
  - Entries are not uniform, different entries may require different information
  - Information is entered as it becomes available during different phases of the compilation
-

# Symbol Tables

UC Santa Barbara

---

- Basic symbol table interface:
    - Insert names (identifiers) and their attributes to the symbol table
      - $\text{Insert}(\text{name}, \text{record})$
    - Lookup names and their attributes
      - $\text{LookUp}(\text{name})$
  - A simple (not very efficient) implementation: Unordered list
    - use an array or linked list
    - enter the names in the order they are encountered
    - Insert  $O(1)$  complexity if multiple entries are allowed. If we have to check that an entry was not entered before, then insert will be  $O(N)$
    - LookUp  $O(N)$  time
    - Simple and compact, but too slow in practice
-

# Hash Tables

UC Santa Barbara

---

- Distribute the entries to  $S$  buckets using a hash function on names
  - Hash function should be efficient and distribute entries uniformly
  - Open hashing: Linked lists for the buckets which have collisions
    - Cost for lookup should be  $O(N / S)$  assuming that computing the hash function takes constant time
    - If  $S \approx N$ , then cost of lookup is  $O(1)$
  - Open addressing: rehash when there is a collision
    - Requires more space since it stores every entry in the table, however rehash chains are short and the lookup is faster if the hash functions have good distributions
-

# Lexically-Scoped Symbol Tables

UC Santa Barbara

---

## The problem

- The compiler needs a distinct record for each declaration
- Nested lexical scopes allow for duplicate declarations

Solution: store the scope information in the symbol table

## The interface

- `Insert(name,level)` – creates record for *name* at *level*
  - `Lookup(name,level)` – returns pointer or index
  - `OpenScope()` – increments the current level and creates a new symbol table for that level
  - `CloseScope()` – changes current level pointer so that it points at the table for the scope surrounding the current level, and then decrements the current level
-

# Example in C

UC Santa Barbara

```
static int w;          /* level 0 */
int x;
void example (int a, int b); /* level 1 */
{
  int c;
  {
    int b, z;          /* level 2a */
    ...
  }
  {
    int a, x;          /* level 2b */
    ...
    {
      int c, x;        /* level 3 */
      b = a + b + c + x;
    }
  }
}
```

Generated sequence of calls:

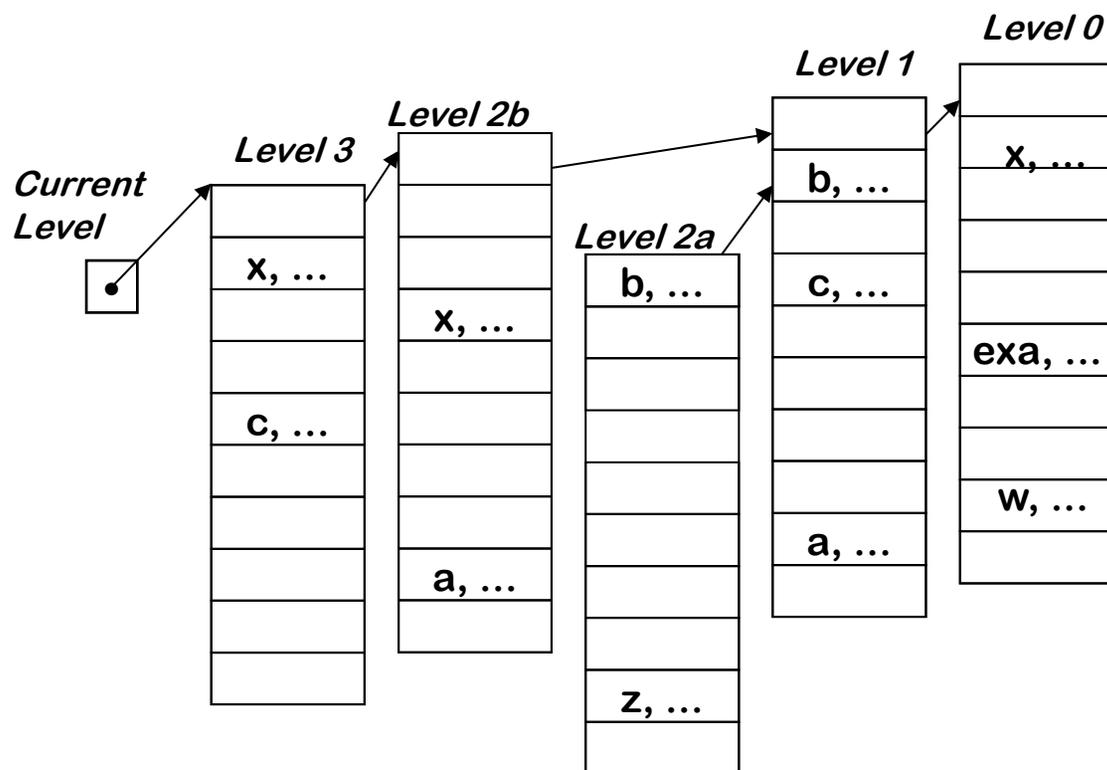
```
OpenScope()
Insert(a)
Insert(b)
Insert(c)
OpenScope()
Insert(b)
Insert(z)
CloseScope()
OpenScope()
Insert(a)
Insert(x)
OpenScope()
Insert(c)
Insert(x)
Lookup(b)
LookUp(a)
LookUp(b)
LookUp(c)
LookUp(x)
CloseScope()
CloseScope()
CloseScope()
```

# Lexically-Scoped Symbol Tables

UC Santa Barbara

## High-level idea

- Create a new table for each scope
- Chain them together for lookup



- *Insert()* inserts at the current level
- *LookUp()* walks chain of tables and returns first occurrence of name
- *OpenScope()* creates a new table, connects it to the current level and updates the current level to point to the new table
- *CloseScope()* removes the table which is the top table in the chain

# What about run-time?

UC Santa Barbara

---

- The symbol tables are how we store information about the variables, scope, procedures at compile-time
  - When we are compiling the program we build up a lexically-scoped symbol table and we use information in the compiler to help us compile a program
    - But when we compile the program, how do we implement procedures?
    - How can we track scope at run-time?
  - We need to compile a special data structure into the program to help us keep track of procedures.
    - We can then push and pop these data structures when procedure is called
    - This data structure is called an activation record
-

# Control and Data

UC Santa Barbara

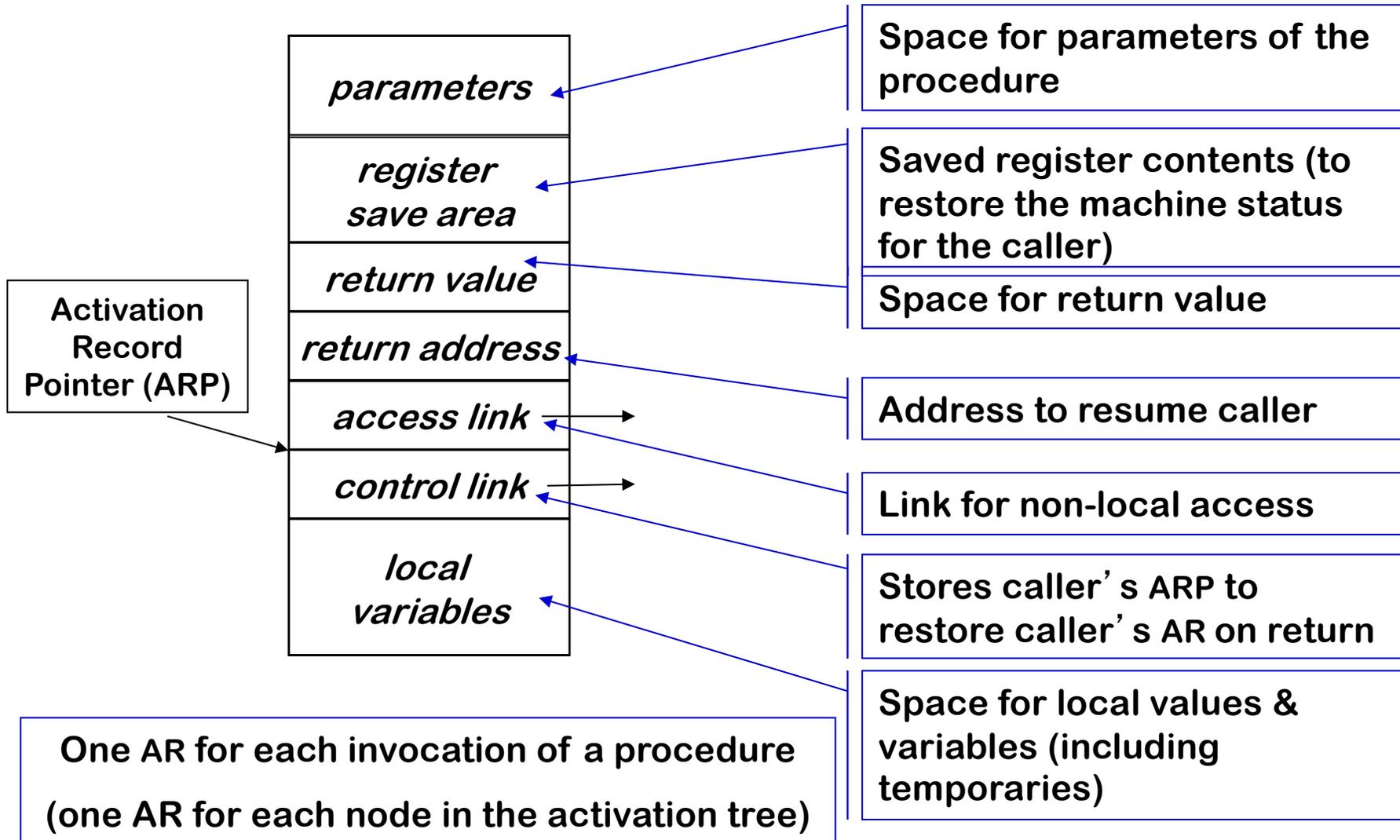
---

To deal with procedures we need to keep track of two basic things

- Transfer of control between procedures: What is the return address
    - Where is the context for the caller? We need a *control link* which will point to the context of the caller
    - What is the return address? In languages such as C and Pascal callee cannot outlive its caller, this means that we can store the return addresses on a stack
  - Where is the data:
    - Where is the local data for the procedure? Again we can use a stack
    - Where is the non-local data? If we can have nested scopes (as in Pascal) then we need an *access link* which will point to the next lexical scope
  - Where do we keep this information: in an *Activation Record*
-

# Activation Record Basics

UC Santa Barbara



# Where Do We Store the Activation Record?

UC Santa Barbara

Where do activation records live?

- If lifetime of AR matches lifetime of invocation *AND*  
If code normally executes a “return” (this is the case with C)  
⇒ Keep ARs on a stack
- If a procedure can outlive its caller *OR*  
If it can return an object that can reference its execution state  
⇒ ARs must be kept in the heap
- If a procedure makes no calls *OR*  
If language does not allow recursion  
⇒ AR can be allocated statically

# Variables in the Activation Record

UC Santa Barbara

How do we find the variables in the activation record at run-time?

- They are at known offsets from the ARP (activation record pointer)
- Access variables by offset (offset is constant, assume ARP is stored in R0):

MOV offset(R0), R1

← This loads the value of  
the variable to register R1

MOV R1, offset(R0)

← This stores the value of  
R1 to the variable

Variable-length data

- If AR can be extended, put variable-length data below local variables
- Leave a pointer at a known offset from ARP
- Otherwise, put variable-length data on the heap

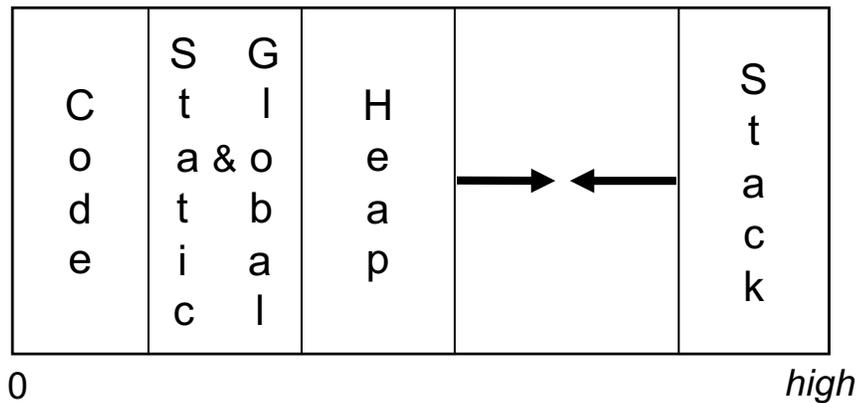
Initializing local variables

- Must generate explicit code to store the values which will be executed in each activation
- Among the procedure's first actions

# Placing Run-time Data Structures

UC Santa Barbara

## Classic Organization



Single Logical Address Space

- Better utilization if stack and heap grow toward each other
- Uses address space, not allocated memory

- Code, static, and global data have known size
  - > Use symbolic labels in the code
- Heap and stack both grow and shrink over time
- This is a virtual address space, it is relocatable

# Memory Allocation

UC Santa Barbara

---

## Local to procedure

- keep them in the AR or in a register
- lifetime matches procedure's lifetime

## For static and global variables

- Lifetime is entire execution, can be stored statically
- Generate assembly language labels for the base addresses (relocatable code)

## Static

- Procedure scope  $\Rightarrow$  mangle procedure name to generate a label
- File scope  $\Rightarrow$  mangle file name to generate a label

## Global

- One or more named global data areas
  - One per variable, or per file, or per program, ...
-

# Establishing Addressability

UC Santa Barbara

Must create base addresses

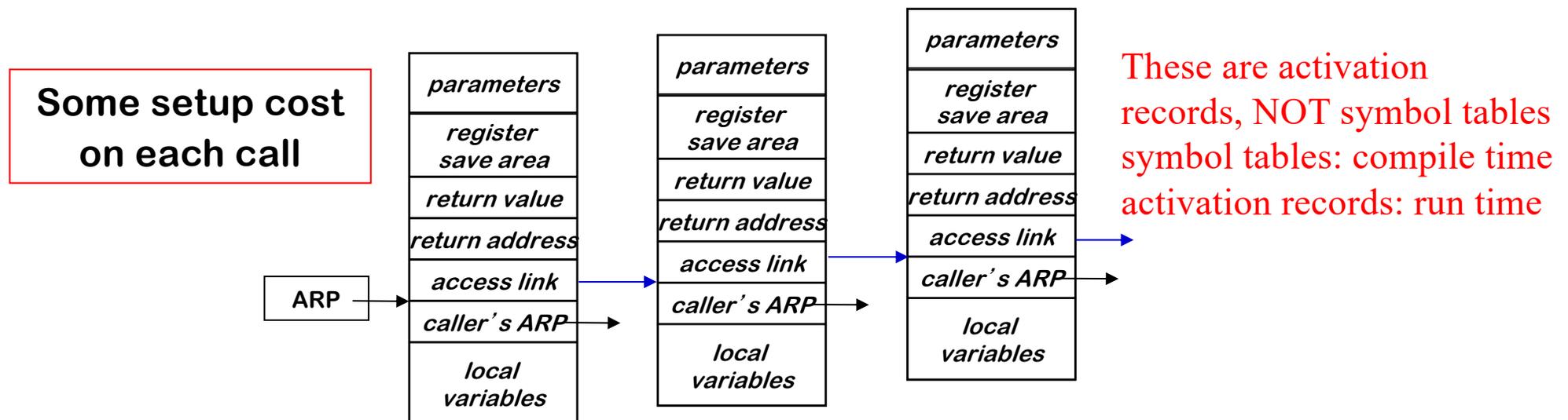
- Global and static variables
    - Construct a label by mangling names (*i.e.*, `&_fee`)
  - Local variables
    - Convert to static distance coordinate and use `ARP + offset`
    - `ARP` becomes the base address
  - Local variables of other procedures
    - Convert to static distance coordinate
    - Find appropriate `ARP`
    - Use that `ARP + offset`
- } Must find the right AR  
} Need links to other ARS

# Establishing Addressability

UC Santa Barbara

## Using access links

- Each AR has a pointer to AR of lexical ancestor
- In some languages lexical ancestor need not be the caller



- Reference to  $p$  runs up access link chain until  $p$  is found
- Cost of access is proportional to lexical distance

# Parameter Passing

UC Santa Barbara

---

- *Call-by-value* (used in C)
    - formal parameter has its distinct storage
    - caller copies the value of the actual parameter to the appropriate parameter slot in callee' s AR
    - do not restore on return
    - arrays, structures, strings are problem
  - *Call-by-reference* (used in PL/I )
    - caller stores a pointer in the AR slot for each parameter
    - if the actual parameter is a variable caller stores its address
    - if the actual parameter is an expression, caller evaluates the expression, stores its results in its own AR, and stores a pointer to that results in the appropriate parameter slot in callee' s AR
-

# Procedure Linkages

UC Santa Barbara

---

How do procedure calls actually work?

- At compile time, caller may not know the callee's code and vice versa
  - Different calls may be in different compilation units (for example they could be in different files)
  - Compiler may not know system code from user code (printf or my\_printf)
  - All calls must use the same protocol

Compiler must use a standard sequence of operations

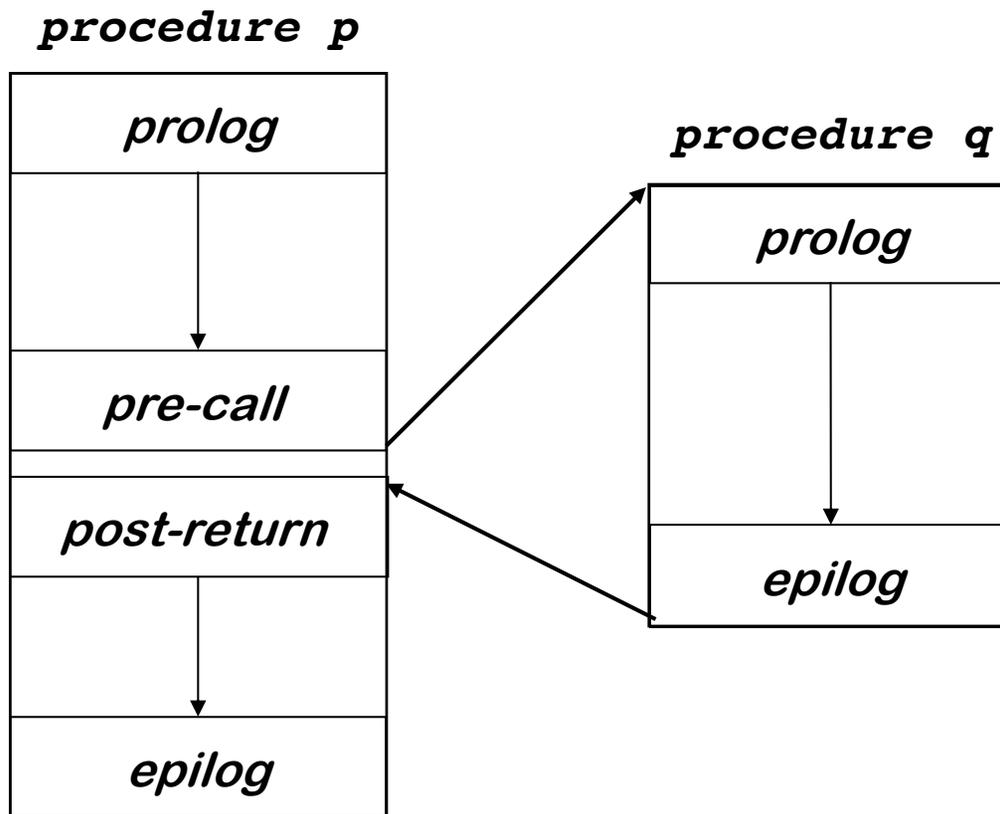
- Enforces control and data abstractions
- Divides responsibility between caller and callee

Usually a system-wide agreement (for interoperability)

---

# Procedure Linkages

## Standard procedure linkage



Procedure has

- standard prolog
- standard epilog

Each call involves a

- pre-call sequence
- post-return sequence

These operations are completely predictable from the call site  $\Rightarrow$  depend on the number and type of the actual parameters

# Procedure Linkages

UC Santa Barbara

---

## Pre-call Sequence

- Sets up callee's basic AR
- Helps preserve caller's environment

## The details

- Allocate space for the callee's AR
  - Evaluates each parameter and stores value or address
  - Saves return address into callee's AR
  - (Possible) Saves caller's ARP into callee's AR (control link)
  - If access links are used
    - Find appropriate lexical ancestor and 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

UC Santa Barbara

---

## Post-return Sequence

- Finish restoring caller's environment
- Place any value back where it belongs

## The details

- Copy return value from callee's AR, if necessary
  - Free the callee's AR
  - Restore any caller-save registers
  - Restore any call-by-reference parameters to registers, if needed
  - Continue execution after the call
-

# Procedure Linkages

UC Santa Barbara

## Prolog Code

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

## The Details

- Preserve any callee-save registers
- If display is being used
  - Save display entry for current lexical level
  - Store current AR into display for current lexical level
- Allocate space for local data
  - Easiest scenario is to extend the AR
- Find any static data areas referenced in the callee (load the base address to a register)
- Handle any local variable initializations

**With heap allocated AR, may need to use a separate heap object for local variables**

# Procedure Linkages

UC Santa Barbara

## Eplilog Code

- Finish the business of the callee
- Start restoring the caller's environment

**If ARs are stack allocated, this may not be necessary. (Caller can reset top of stack to its pre-call value.)**

## The Details

- Store return value? No, this happens on the return statement
- Restore callee-save registers
- Free space for local data, if necessary
- Load return address from AR
- Restore caller's ARP
- Jump to the return address