6 Procedures

Chapter Contents

The Procedure Abstraction

The procedure is one of the **central abstractions** that underlie most modern programming languages

- important way for programmers to structure their software
- basic unit of work for most compilers

Procedures provide **three** critical abstractions

1. Procedure Call
2. Name Space
3. External Interface

Together, these abstractions allow the construction of **complex** and **nontrivial** programs.

The Procedure Abstraction

The **procedure call** abstraction is a standard mechanism to…

- invoke a procedure
- map parameters from the caller's name space to the callee's name space
- return control to the caller and resume execution at the point after the call
- return one or more values to the caller

Caller and Callee

In a procedure call, we refer to the procedure that is invoked as the **callee** and to the calling procedure as the **caller**.

The Procedure Abstraction

Each procedure creates a new and protected **name space**

- enable declaration of new names without concern for the surrounding context
- caller can map values and variables in its name space into the callee's name space
- procedure functions correctly and consistently when called from different contexts

Formal and Actual Parameters

A name declared as a parameter of some procedure `p` is a **formal parameter** of `p`. A value or variable passed as a parameter to `p` at a specific call site is an **actual parameter** of the call.

The Procedure Abstraction

Procedures define the critical **interfaces** among the parts of large software systems

The **linkage convention** is an agreement between the compiler and operating system that defines rules

- that map names to values and locations,
- that preserve the caller's runtime environment, and
- that create the callee's environment

It creates a context in which a programmer can safely invoke code written by others.

Without a linkage convention, both the programmer and the compiler would need **detailed knowledge** about the implementation of the callee at each procedure call.

Procedure Calls

In the following, we focus on Algol-like languages that have a simple and clear call/return discipline

- a procedure call transfers control from the call site in the caller to the **start** of the callee
- on exit from the callee, control returns to the point **immediately following** the call site

If the callee invokes **other** procedures, they are activated in the same way.

Activation

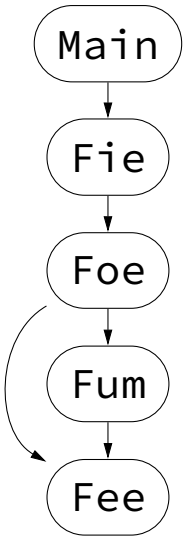
A call to a procedure activates it. Thus, we call an instance of its execution an activation.

Procedure Calls

```
1  program Main(input, output);
2     var x, y, z: integer;
3
4     procedure Fee;
5        var x: integer;
6     begin { Fee }
7        x := 1;
8        y := x * 2 + 1
9     end;
10
11    procedure Fie;
12       var y: real;
13
14       procedure Foe;
15          var z: real;
16
17          procedure Fum;
18             var y: real;
19          begin { Fum }
20             x := 1.25 * z;
21             Fee;
22             writeln('x = ',x)
23          end;
24
25       begin { Foe }
26          z := 1;
27          Fee;
28          Fum
29       end;
30
31    begin { Fie }
32       Foe;
33       writeln('x = ',x)
34    end;
35
36 begin { Main }
37    x := 0;
38    Fie
39 end.
```

Procedure Calls

The **call graph** shows the set of potential calls among the procedures. Executing `Main` can result in two calls to `Fee`: one from `Foe` and another from `Fum`.



1. `Main` calls `Fie`
2. `Fie` calls `Foe`
3. `Foe` calls `Fee`
4. `Fee` returns to `Foe`
5. `Foe` calls `Fum`
6. `Fum` calls `Fee`
7. `Fee` returns to `Fum`
8. `Fum` returns to `Foe`
9. `Foe` returns to `Fie`
10. `Fie` returns to `Main`

The **execution history** shows that both calls occur at runtime.

Procedure Calls

When the compiler generates code for calls and returns, that code must **preserve** enough information so that calls and returns operate correctly.

The call and return behavior of Algol-like languages can be modelled with a **stack**

- when a caller calls a callee, it **pushes** the return address in the caller onto the stack
- when the callee returns, it **pops** that address off the stack and jumps to the address

Return Address

When `p` calls `q`, the address in `p` where execution should continue after `p`'s return is called its **return address**.

Name Space

In most procedural languages, a complete program will contain multiple **name spaces**.

A name space…

- maps a set of names to a set of values and procedures over a range of statements
- may inherit some names from other name spaces
- can create names that are inaccessible outside the name space

Name space rules give the programmer control over access to information.

Scope

In an Algol-like language, **scope** refers to a name space. The term is often used in discussions of the visibility of names.

Name Spaces of Algol-like Languages

Most programming languages inherit many of the conventions that were defined for Algol 60. This is particularly true of the rules that govern the visibility of names.

While Algol 58 introduced the fundamental notion of the **compound statement**, Algol 60 was the first language implementing **nested function** definitions with **lexical scope**.

Lexical Scope

Scopes that nest in the order that they are encountered in the program are often called **lexical scopes**.

Nested Lexical Scopes

Most Algol-like languages allow the programmer to **nest** scopes inside one another.

The limits of a scope are marked by specific terminal symbols

- Pascal demarcated scopes with a `begin` at the start and an `end` at the finish
- C uses curly braces, { and }, to begin and end a block, its notion of a scope

Note Each procedure also defines a new scope that covers its entire definition. The programmer can declare new variables and, in some languages, procedures in this scope.

With (nested) lexical scopes a compiler must decide which use of name refers to which declaration. To do so, it uses the **scoping rules** defined by the programming language.

Nested Lexical Scopes

Lexical Scoping

The most common scoping discipline is called **lexical scoping**. It follows a simple general principle.

In a given scope, each name refers to its lexically closest declaration.

Example If `s` is used in the current scope, it refers to the `s` declared in the current scope. If none exists, it refers to the declaration of `s` that occurs in the **closest enclosing** scope.

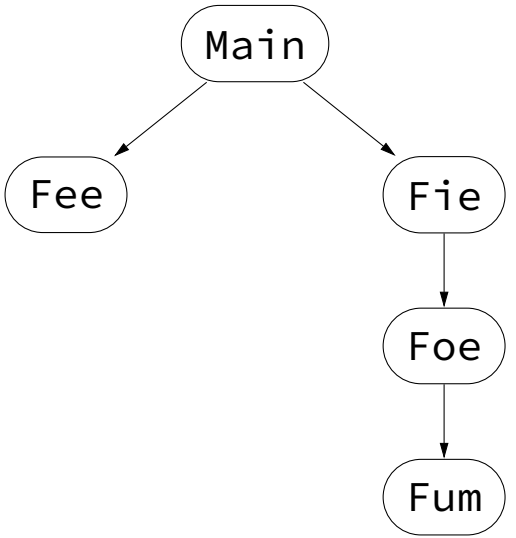
Note The outermost scope contains **global variables**.

Nested Lexical Scopes

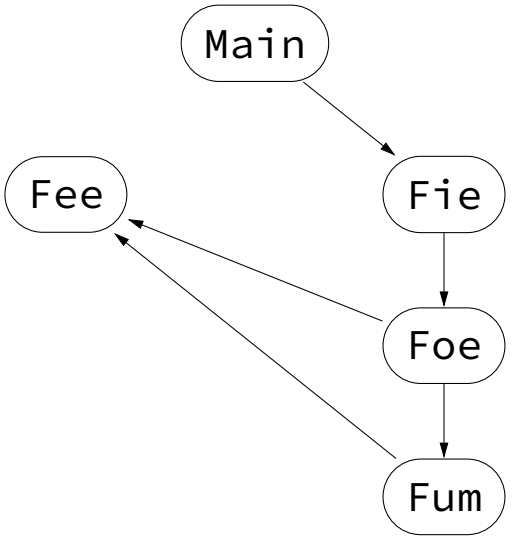
```
1  program Main0(input, output);
2     var x1, y1, z1: integer;
3
4     procedure Fee1;
5        var x2: integer;
6     begin { Fee1 }
7        x2 := 1;
8        y1 := x2 * 2 + 1
9     end;
10
11    procedure Fie1;
12       var y2: real;
13
14       procedure Foe2;
15          var z3: real;
16
17          procedure Fum3
18             var y4: real;
19          begin { Fum3 }
20             x1 := 1.25 * z3;
21             Fee1;
22             writeln('x = ',x1)
23          end;
24
25       begin { Foe2 }
26          z3 := 1;
27          Fee1;
28          Fum3
29       end;
30
31    begin { Fie1 }
32       Foe2;
33       writeln('x = ',x1)
34    end;
35
36 begin { Main0 }
37    x1 := 0;
38    Fie1
39 end.
```

Nested Lexical Scopes

Nesting Relationships



Call Graph



Nested Lexical Scopes

The listing on Slide 280 shows each name with a subscript that indicates its **level** number. Names declared in a procedure are **one** level deeper than the level of the procedure name.

To represent names in a lexically scoped language, the compiler can use the **static coordinate** for each name.

Static Coordinate

The static coordinate is a pair $\langle \mathrm{l}, \mathrm{o} \rangle$, where l is the name's lexical nesting level and o is the its offset in the data area for level l.

To obtain l, the front end uses a **lexically scoped** symbol table. The offset o should be stored with the name and its level in the symbol table.

Nested Lexical Scopes

Example The table below shows the static coordinate for each variable name in each procedure of the code on Slide 280.

Scope	x	y	z
`Main`	$\langle 1,0 \rangle$	$\langle 1,4 \rangle$	$\langle 1,8 \rangle$
`Fee`	$\langle 2,0 \rangle$	$\langle 1,4 \rangle$	$\langle 1,8 \rangle$
`Fie`	$\langle 1,0 \rangle$	$\langle 2,0 \rangle$	$\langle 1,8 \rangle$
`Foe`	$\langle 1,0 \rangle$	$\langle 2,0 \rangle$	$\langle 3,0 \rangle$
`Fum`	$\langle 1,0 \rangle$	$\langle 4,0 \rangle$	$\langle 3,0 \rangle$

Scope Rules across Various Languages

Scoping rules vary idiosyncratically from programming language to programming language.

FORTRAN

- single global scope that groups variables into a “common block”
- each procedure or function has a local scope for parameters, variables, and labels
- `save` statement to give a local variable the lifetime of a global variable

ANSI C

- global scope for procedure names and global variables
- each procedure has a local scope for parameters, variables, and labels
- blocks also define a separate local scope and they can be nested
- file-level scope includes `static` names that are not enclosed in a procedure

Runtime Structures to Support Algol-like Languages

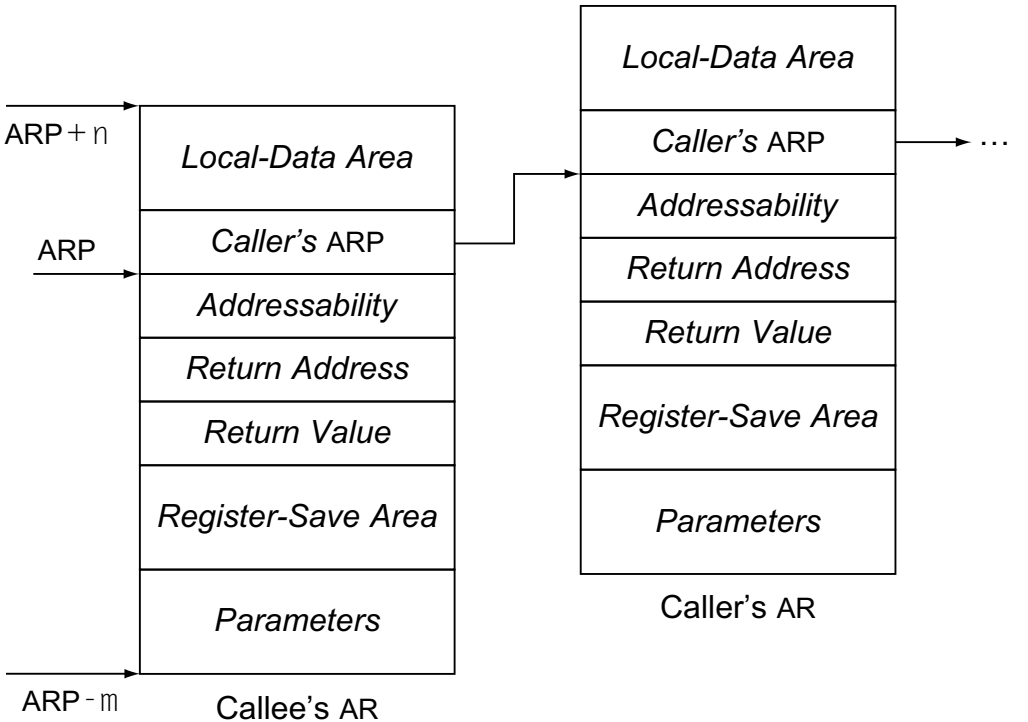
To realize **procedure calls** and **scoped name spaces**, runtime structures are required. The key data structure to implement both abstractions is the **activation record** (AR).

In principle, **each procedure call** gives rise to a new AR

- stores the **return address** where the callee can find it
- maps the **actual parameters** to the formal parameter names known by the callee
- reserves storage space for **local variables** declared in the callee's scope
- provides **other information** needed to connect the callee to the surrounding program

If **multiple instances** of a procedure are active, *e.g.*, recursive calls, each has its own AR.

Runtime Structures to Support Algol-like Languages



Runtime Structures to Support Algol-like Languages

The entire AR is addressed through an **activation record pointer** (ARP), with various fields in the AR found at positive and negative offsets from the ARP.

The ARP always points to a designated location in the AR

- **fixed-length fields** are stored in the center, where compiled code can access those items at fixed offsets from the ARP
- **variable-length fields** whose sizes may change from one invocation to another, *e.g.*, parameters and local data, are stored at either end

Question

Is this a reasonable layout for the activation record?

Runtime Structures to Support Algol-like Languages

Because procedures typically access their AR frequently, most compilers **dedicate** a hardware register to hold the ARP of the current procedure.

Examples

- SPARC Assembly uses registers `%fp` (frame pointer) and `%sp` (stack pointer) to demarcate the activation record (or stack frame)
- the corresponding registers in X86-64 Assembly are called `%rbp` (base pointer) and `%rsp` (stack pointer)
- in ILOC, we refer to this dedicated register as r_{arp}

Local Storage

The compiler needs to **reserve space for local data** items in the AR

- assign each such item an appropriately sized area
- record its offset from the ARP in the symbol table

Together with its lexical level the item's offset becomes the its static coordinate.

If the compiler **cannot know the size** of a local variable at compile time

1. leave space for a pointer to the actual data or to a descriptor for an array
2. arranges to allocate the actual storage elsewhere at runtime
3. fill the reserved slot with the address of the dynamically allocated memory

Local Storage

If the source language allows the program to specify an initial value for a variable, the compiler must arrange for that **initialization** to occur.

Static and Global Variables

- lifetime independent of any procedure
- data can be inserted directly into the appropriate locations by the loader
- for example, use a **data section** (`.data`) in assembly code

Local Variables

- must be initialized at runtime as a procedure may be invoked multiple times
- generate instructions that store the initial values to the appropriate locations

Local Storage

When p calls q, one of them must **save the register values** that p needs after the call

- it may be necessary to save all the register values or a subset may suffice
- on return to p, these saved values must be restored
- saved registers are stored in the AR of either p or q, or both

If the **callee saves a register**, its value is stored in register save area of the callee's AR. Similarly, if the **caller saves a register**, its value is stored in the caller's register save area.

Calling Convention

Who saves what is defined by the **calling convention** used by the operating system. Solaris, Linux, FreeBSD, and macOS follow the System V AMD64 ABI calling convention. Windows follows the Microsoft x64 calling convention.

Local Storage

Example The table below shows which registers are saved by the caller and the callee, respectively, when following the System V AMD64 ABI calling convention.

Register	Purpose	Saved by
`%rax`	return value	–
`%rbx`	scratch	callee
`%rcx`	4th parameter	–
`%rdx`	3rd parameter	–
`%rsi`	2nd parameter	–
`%rdi`	1st parameter	–
`%rbp`	base pointer	callee
`%rsp`	stack pointer	callee
`%r8`	5th parameter	–
`%r9`	6th parameter	–
`%r10`	scratch	caller
`%r11`	scratch	caller
`%r12`	scratch	callee
`%r13`	scratch	callee
`%r14`	scratch	callee
`%r15`	scratch	callee

Allocating Activation Records

When p calls q at runtime, the code that implements the call must allocate an AR for q and initialize it with the appropriate values.

Fields of AR are stored in **memory**

- caller needs to have access to callee's AR to store parameters, return address, *etc*.
- this forces allocation of callee's AR into caller, who might not know the local data size

Values of AR passed by **registers**

- allocation of the ar can be performed in the callee, who knows the local data size
- callee may store into its ar some of the values passed in registers after allocation

The compiler can allocate ARs on the **stack** or on the **heap**. In certain cases, it can allocate a single **static** AR for a procedure or **coalesce** ARs of a set of procedures.

Allocating Activation Records

If procedure calls and returns are balanced, they follow a last-in, first-out (LIFO) discipline. Since the ARs also follow this LIFO ordering, they can be allocated on the **stack**.

Keeping activation records on the stack has several advantages

- inexpensive allocation and deallocation: simply move the top-of-stack pointer
- separation of concerns between caller and callee
  - caller allocates space for address parameters, return address, *etc*.
  - callee can extend AR to include local data area and variable-sized objects
- debugger can walk the stack to produce a snapshot of the currently active procedures

Example Pascal, C, and Java are typically implemented with stack-allocated ARs

Allocating Activation Records

Certain language features prevent ARs from being allocated on the stack

- if a procedure can outlive its caller
- if a procedure can return an object that includes references to its local variables

In these situations, ARs can be kept on the **heap**. With heap-allocated ARs, variable-size objects can be allocated as separate objects on the heap.

- explicit deallocation: procedure return code frees the AR and variable-size objects
- implicit deallocation: garbage collector frees them when they are no longer useful

Example Implementations of Scheme and ML typically use heap-allocated ARs

Allocating Activation Records

A **leaf procedure** is a procedure that calls no other procedures. The compiler can allocate ARs for leaf procedures **statically**. This eliminates the runtime costs of AR allocation.

Example If the calling convention requires the caller to save its own registers, then the AR of a leaf procedure needs no register save area.

Under certain circumstances the compiler can do even better! Assuming that callees cannot outlive callers, **only one** leaf procedure can be active at any point during execution.

- allocate a **single** static AR for use by **all** leaf procedures
- AR must be large enough to accommodate any of the program's leaf procedures
- static variables of all leaf procedures can be laid out together in that single AR

Using a single static AR for leaf procedures reduces the space overhead of separate static ARs for each leaf procedure.

Allocating Activation Records

If the compiler discovers a set of procedures that are always invoked in a fixed sequence, it may be able to **coalesce** their ARs and save allocation costs.

Example If a call from `p` to `q` always results in calls to `r` and `s`, the compiler may find it profitable to allocate the ARs for `q`, `r`, and `s` at the same time.

In practice, this optimization is **limited** by the following factors

- separate compilation
- function-valued parameters

Communicating Values Between Procedures

A procedure **encapsulates** common operations relative to a small set of names. To make this encapsulation work, the compiler needs to **bind these names to values** correctly.

Names in a procedure can refer to the following values

- parameter values
- return value
- values of global variables
- values of local variables in an enclosing procedures

Passing Parameters

Parameter binding plays a critical role in our ability to write abstract, modular code

- procedure can be written without knowledge of the contexts in which it will be called
- procedure can be called from many contexts without exposing its internal details

Most modern programming languages use one of two conventions for mapping actual parameters to formal parameters: **call-by-value** and **call-by-reference**.

Call by Value

Call by Value

Call-by-value is a convention where the caller evaluates the actual parameters and passes their values to the callee.

Any modification of a value parameter in the callee is not visible in the caller.

Call-by-value parameter passing

- caller **copies** the value of an actual parameter into the **appropriate location** for the corresponding formal parameter, *i.e.*, a register or a parameter slot in the callee's AR
- **only one name** refers to that value: the name of the formal parameter
- its value is an **initial condition** determined by evaluating the actual parameter
- if the callee **changes the value**, the change is visible inside the callee, but not outside

Call by Value

```
1  int fee(int x, int y) {
2     x = 2 * x;
3     y = x + y;
4     return y;
5  }
6
7  c = fee(2,3);
8  a = 2;
9  b = 3;
10 c = fee(a,b);
11 a = 2;
12 b = 3;
13 c = fee(a,a);
```

The three invocations produce the following results when invoked using call-by-value parameter binding.

Call by Value	a		b		Return Value
	in	out	in	out	
fee(2,3)	–	–	–	–	7
fee(a,b)	2	2	3	3	7
fee(a,a)	2	2	3	3	6

With call by value, the binding is simple and intuitive.

Call by Reference

Call by Reference

Call-by-reference is a convention where the caller passes an address for the formal parameter to the callee.

If the actual parameter is a variable (rather than an expression), then changing the formal's value also changes the actual's value.

Call-by-reference parameter passing

- caller stores a **pointer** in the register or AR slot for each parameter
- in the callee, each call-by-reference formal parameter needs a **level of indirection**

Call by Reference

Dealing with different kinds of actual parameters

- **variables**: pass the variable's address to the callee
- **expressions**
  1. caller evaluates the expression
  2. stores the result in the local data area of its own AR
  3. passes a pointer to that result to the callee
- **constants**: should be treated as expressions to avoid being changed by the callee

Note Some languages forbid passing expressions and constants as actual parameters to call-by-reference formal parameters.

Call by Reference

Call by reference differs from call by value in **two critical** ways

1. any redefinition of a reference formal parameter is reflected in the corresponding actual parameter
2. any reference formal parameter might be bound to a variable that is accessible by another name inside the callee

Aliasing

When two names can refer to the same location, they are said to be **aliases**. Aliasing can create counterintuitive behavior.

Call by Reference

```
1  int fee(int* x, int* y) {
2     *x = 2 * *x;
3     *y = *x + *y;
4     return *y;
5  }
6
7  c = fee(2,3);
8  a = 2;
9  b = 3;
10 c = fee(&a,&b);
11 a = 2;
12 b = 3;
13 c = fee(&a,&a);
```

With (simulated) call-by-reference parameter passing, the example produces different results.

Call by Value	a		b		Return Value
	in	**out**	**in**	**out**	
`fee(2,3)`	*does not compile in C*				
`fee(a,b)`	2	4	3	7	7
`fee(a,a)`	2	8	3	3	8

The third call causes `x` and `y` to refer to the same location, and thus, the same value.

Space for Parameters

The size of the representation for a parameter has an impact on the cost of procedure calls.

Scalar Values (*e.g.*, variables or pointers)

- stored in registers or in the parameter area of the callee's AR
- cost per parameter is small for both call-by-value and call-by-reference parameters

Large Values (*e.g.*, arrays, records, or structures)

- copying call-by-value parameters will add significant cost to the procedure call
- some languages allow the implementation to pass such parameters by reference
- others include constructs to specify that passing a particular parameter by reference is acceptable, *e.g.*, `const` in C

Returning Values

Because the return value, by definition, is used after the callee terminates, it needs storage **outside** the callee's AR.

If the return value is **small** and has a **fixed size**

- store return value in a **designated register**
- store return value in the **caller's AR**
  1. caller allocates space for return value in its own AR
  2. caller stores a pointer to that space in the return slot of its own AR
  3. callee uses ARP to caller's AR load the pointer

If the caller **cannot know the size** of the returned value

1. callee allocates space for return value, possibly on the heap
2. callee stores the pointer in the return-value slot of the caller's AR
3. caller must free the space allocated by the callee

Establishing Addressability

Apart from binding parameters and passing the return value, the compiler must ensure that each procedure can **generate an address** for each variable that it needs to reference.

In general, the address calculation consists of **two** portions

- finding the **base address** of the data area of the scope that contains the value
- finding the correct **offset** within that data area for the value

Data Area and Base Address

The memory region that holds the data for a specific scope is called its **data area**.

The address of the start of a data area is often called a **base address**. The base address can either be static, *i.e.*, known at compile time, or dynmaic, *i.e.*, only known at runtime.

Variables with Static Base Addresses

Compilers typically arrange for global and static data areas to have **static base addresses**.

The strategy to generate an address for such a variable is simple

- compute the data area's base address into a register
- add its offset to the base address

Compiler attaches a **symbolic**, **assembly-level label** for the variable to the data area.

Dpending on the instruction set, the label might be used in a **load immediate** operation or it might evaluate to an address which can be loaded into a register with a **move** operation.

The label becomes a **relocatable symbol** for the assembler and the loader, which convert it into a runtime virtual address.

Variables with Static Base Addresses

The compiler constructs the label for a base address by mangling the name

- add a prefix, a suffix, or both to the original name
- use characters that are legal in the assembly code but not in the source language

Name Mangling

The process of constructing a unique string from a source-language name is called **name mangling**.

Variables with Static Base Addresses

```
1    .data                          # begin of the global data area
2 _msg:
3    .ascii  "Hello,␣world!\n"
4    .text                          # begin of the program text
5    .globl _main                   # export main
6 _main:
7    pushq %rbp                     # save current base pointer
8    movq  %rsp, %rbp               # move base pointer to stack pointer
9    leaq  _msg(%rip), %rdi         # load value from static base address
10   xorq  %rax, %rax               # zero out %rax (varargs = 0)
11   call  _printf                  # call printf
12   popq  %rbp                     # restore base pointer
13   retq                           # return
```

Note Global constant can also be loaded with `movq _msg@GOTPCREL(%rip), %rdi`.

Variables with Dynamic Base Addresses

Local variables declared within a procedure are typically stored in the procedure's AR

- they have **dynamic base addresses**
- compiler needs a mechanism to find the addresses of various ARs

Fortunately, lexical scoping rules limit the set of ARs that can be accessed from any point in the code to the **current AR** and the **ARs of lexically enclosing procedures**.

Local Variable of the Current Procedure

Accessing a **local variable of the current procedure** is trivial: its base address is simply the address of the current AR, which is stored in the ARP.

- add offset to the ARP and use the result as the value's address
- most processors provide efficient support for these common operations

In some cases, a value is not stored at a **constant** offset from the ARP.

- value might reside in a register, *i.e.*, loads and stores are not needed
- variable might have an unpredictable or changing size
  1. store it in an area reserved for variable-size objects in AR or on the heap
  2. reserve space in AR for a pointer the variable's actual location
  3. generate one additional load to access the variable

Local Variables of Other Procedures

To access a **local variable of some enclosing lexical scope**, the compiler must build runtime data structures that can map a **static coordinate** into a **runtime address**.

Access Links or Static Links

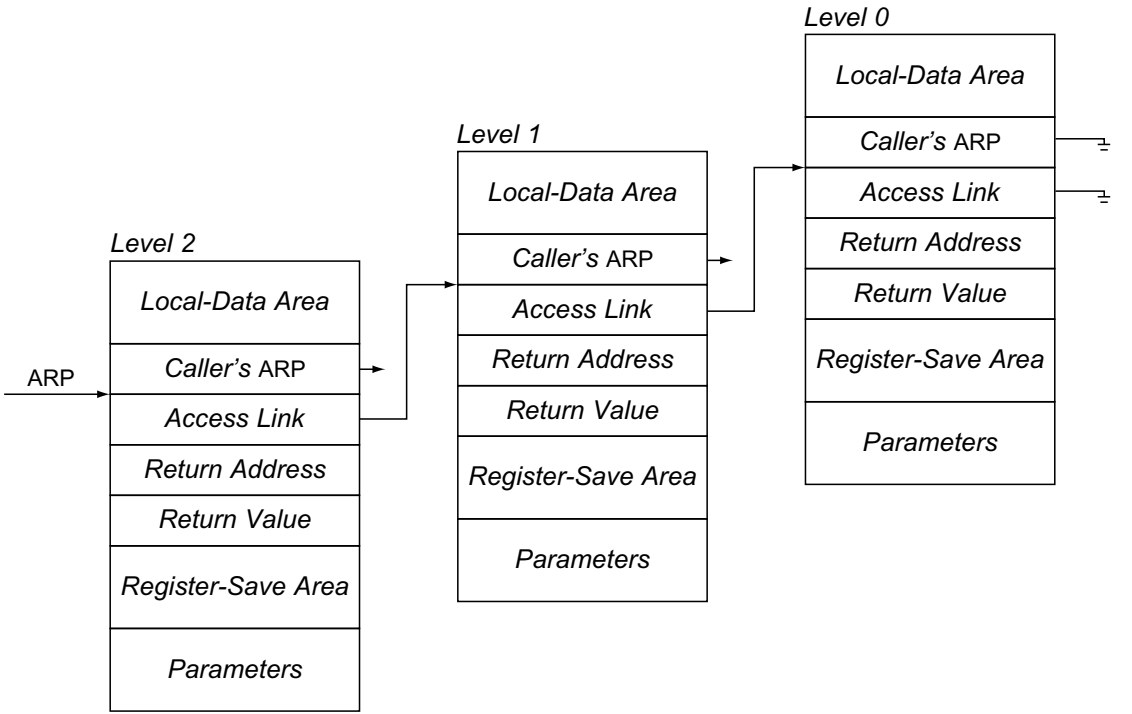
- extend each AR with a pointer to the AR of its **immediate** lexical ancestor
- emit code to walk the chain of links and to find the appropriate ARP
- load value with its offset (from static coordinate) relative to this ARP

Global Display

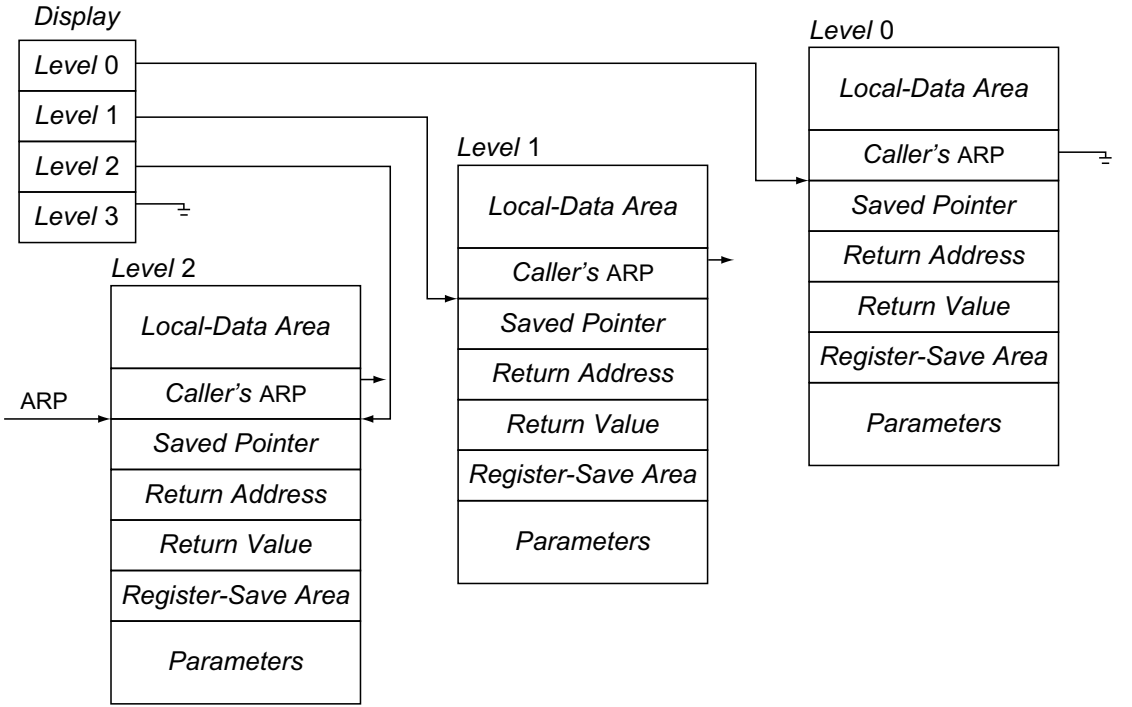
- a global array stores ARP of the **most recent** procedure activation at each lexical level
- local variables of other procedures are referenced indirectly through the display

While the cost of nonlocal access is fixed with a global display, the compiler must insert code where needed to maintain the values in the display.

Access Links



Global Display



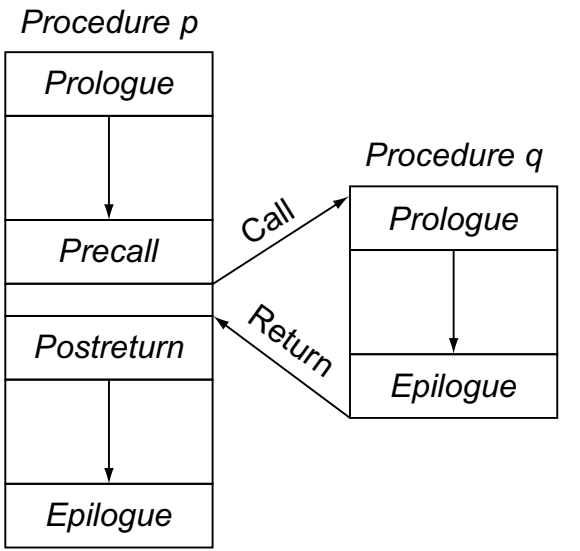
Standardized Linkages

The **procedure linkage** is a contract between the compiler, the operating system, and the target machine

- assigns responsibility for naming, resource allocation, addressability, and protection
- ensures interoperability of code emitted by one compiler with code from other sources

Pieces of a standard procedure linkage

- each procedure has a **prologue sequence** and an **epilogue sequence**
- each call site includes both a **precall sequence** and a **postreturn sequence**



System V AMD64 ABI

Mapping Actual to Formal Parameters

- integer arguments (including pointers) are placed in the registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`, in that order
- floating point arguments are placed in the registers `%xmm0`-`%xmm7`, in that order.
- parameter values in excess of the available registers are pushed onto the stack
- if the function takes a variable number of arguments (like `printf`) then the `%eax` register must be set to the number of floating point arguments

Saving Registers

- the callee may use any registers, but it must restore the values of the registers `%rbx`, `%rbp`, `%rsp`, and `%r12`-`%r15`, if it changes them.

Returning Values

- the return value of a call is placed in `%rax`