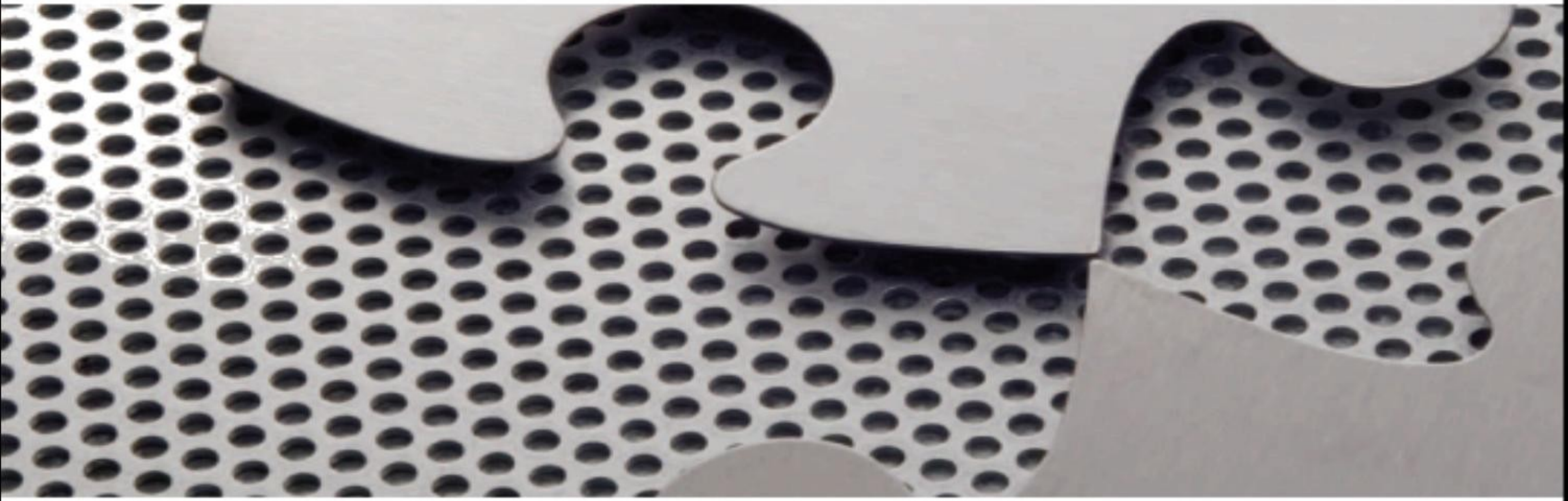
Programming Languages Third Edition



Chapter 10

Control II – Procedures and Environments

Objectives

- Understand the nature of procedure definition and activation
- Understand procedure semantics
- Learn parameter-passing mechanisms
- Understand procedure environments, activations, and allocation
- Understand dynamic memory management

Objectives (cont'd.)

- Understand the relationship between exception handling and environments
- Learn to process parameter modes in TinyAda

Introduction

- Procedures and functions: blocks whose execution is deferred and whose interfaces are clearly specified
- Many languages make strong syntactic distinctions between functions and procedures
- Can make a case for a significant semantic distinction as well:
  - Functions should produce a value only have no side effects
  - Procedures produce no values and operate by producing side effects

Introduction (cont'd.)

- Procedure calls are therefore statements, while function calls are expressions
- Most languages do not enforce semantic distinctions
  - Functions can produce side effects as well as return values
  - Procedures may produce values through their parameters while causing no side effects
- We will not make a significant distinction between them since most languages do not

Introduction (cont'd.)

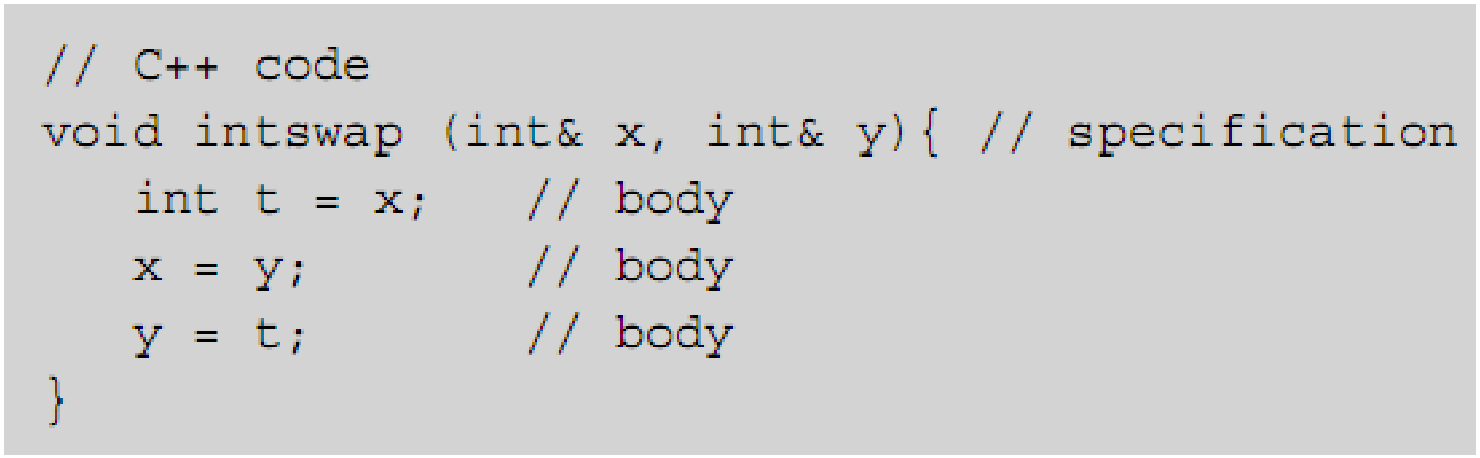
- Functional languages generalize the notion of a function
  - Functions are first-class data objects themselves
- Functions require dynamic memory management, including garbage collection
- **Activation record**: the collection of data needed to maintain a single execution of a procedure

Procedure Definition and Activation

- **Procedure**: a mechanism for abstracting a group of actions or computations
- **Body** of the procedure: the group of actions
- Procedure name: represents the body
- A procedure is defined by providing a **specification** (or **interface**) and a body
- **Specification**: includes the procedure name, types and names of the formal parameters, and the return value type (if any)

Procedure Definition and Activation (cont'd.)

- Example: in C++ code



```
// C++ code
void intswap (int& x, int& y){ // specification
   int t = x;    // body
   x = y;        // body
   y = t;        // body
}
```

- In some languages, a procedure specification can be separated from its body
  - Example:

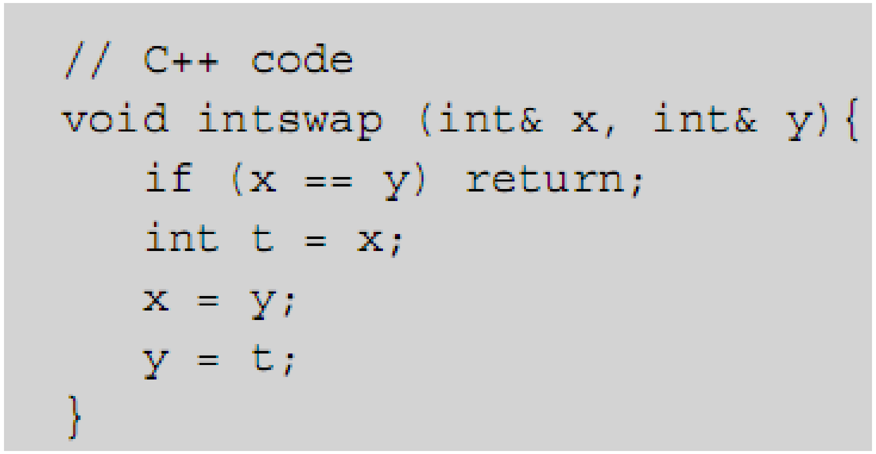
```
void intswap (int&, int&); // specification only
```

Procedure Definition and Activation (cont'd.)

- You **call** (or **activate**) a procedure by stating its name and providing arguments to the call which correspond to its formal parameters
  - Example: `intswap (a, b);`
- A call to a procedure transfers control to the beginning of the body of the called procedure (the **callee**)
- When execution reaches the end of the body, control is returned to the **caller**
  - May be returned before the end of the body by using a **return-statement**

Procedure Definition and Activation (cont'd.)

- Example:



```
// C++ code
void intswap (int& x, int& y){
    if (x == y) return;
    int t = x;
    x = y;
    y = t;
}
```

- In FORTRAN, procedures are called **subroutines**
- **Functions**: appear in expressions and compute **returned values**

Procedure Definition and Activation (cont'd.)

- A function may or may not change its parameters and nonlocal variables
- In C and C++, all procedures are implicitly functions
  - Those that do not return values are declared `void`
- In Ada and FORTRAN, different keywords are used for procedures and functions
- Some languages allow only functions
  - All procedures must have return values
  - This is done in functional languages

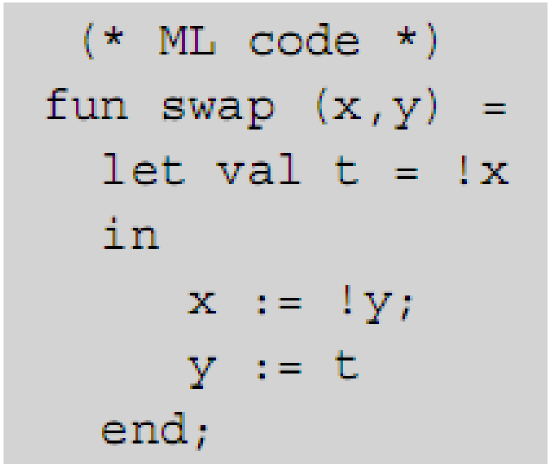
Procedure Definition and Activation (cont'd.)

```
-- Ada procedure
procedure swap ( x,y: in out integer) is
  t: integer;
begin
  if (x = y) then return;
  end if;
  t := x;
  x := y;
  y := t;
end swap;

-- Ada function
function max ( x,y: integer ) return integer is
begin
   if (x > y) then return x;
   else return y;
   end if;
end max;
```

Procedure Definition and Activation (cont'd.)

- In ML, procedure and function declarations are written in a form similar to constant declarations



```
  (* ML code *)
fun swap (x,y) =
  let val t = !x
  in
     x := !y;
     y := t
  end;
```

- A procedure declaration creates a **constant procedure value** and associates a symbolic name with that value

Procedure Definition and Activation (cont'd.)

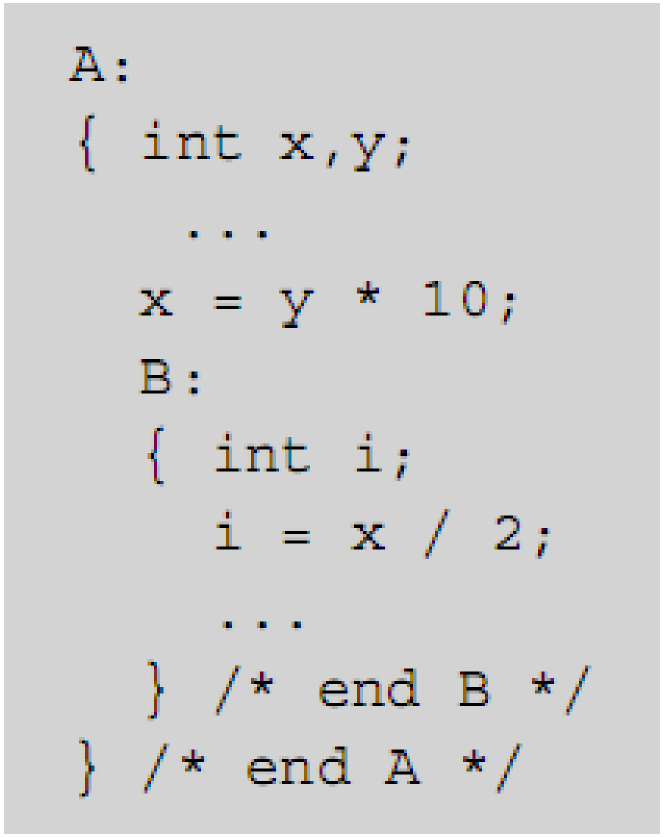
- A procedure communicates with the rest of the program through its parameters and also through **nonlocal references** (references to variables outside the procedure body)
- **Scope rules** that establish the meanings of nonlocal references were covered in Chapter 7

Procedure Semantics

- Semantically, a procedure is a block whose declaration is separated from its execution
- The **environment** determines memory allocation and maintains the meaning of names during execution
  - Memory allocated for local objects of a block are called the **activation record** (or **stack frame**)
  - The block is said to be **activated** as it executes
- When a block is entered during execution, control transfers into the block’s activation

Procedure Semantics (cont'd.)

- Example: in C code
  - Blocks A and B are executed as they are encountered
- When a block is exited, control transfers back to the surrounding block, and the activation record of the exiting block is released



```
A:
{ int x,y;
   ...
  x = y * 10;
  B:
  { int i;
    i = x / 2;
    ...
  } /* end B */
} /* end A */
```

Procedure Semantics (cont'd.)

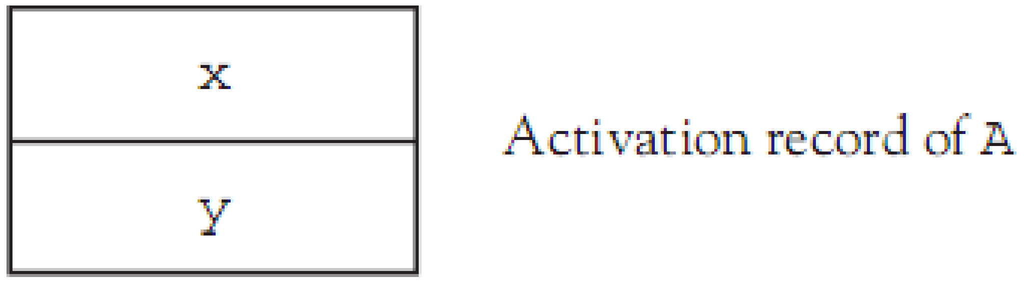


Figure 10.1 The activation record for block A

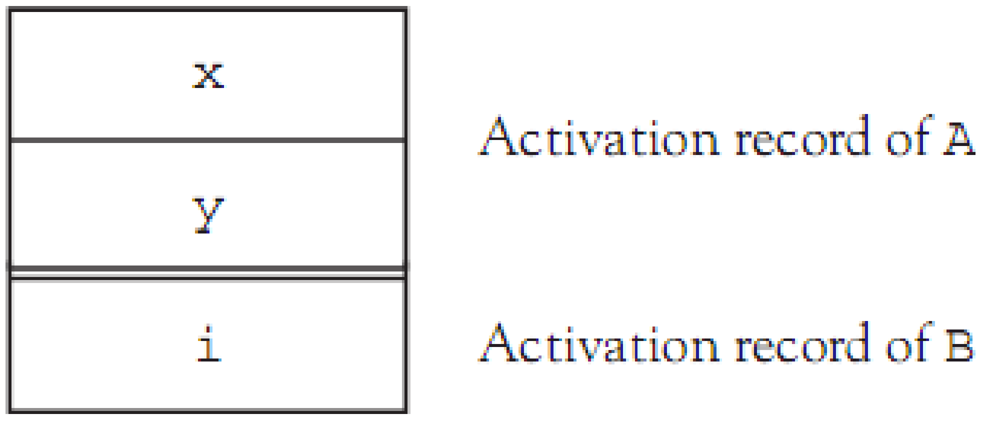
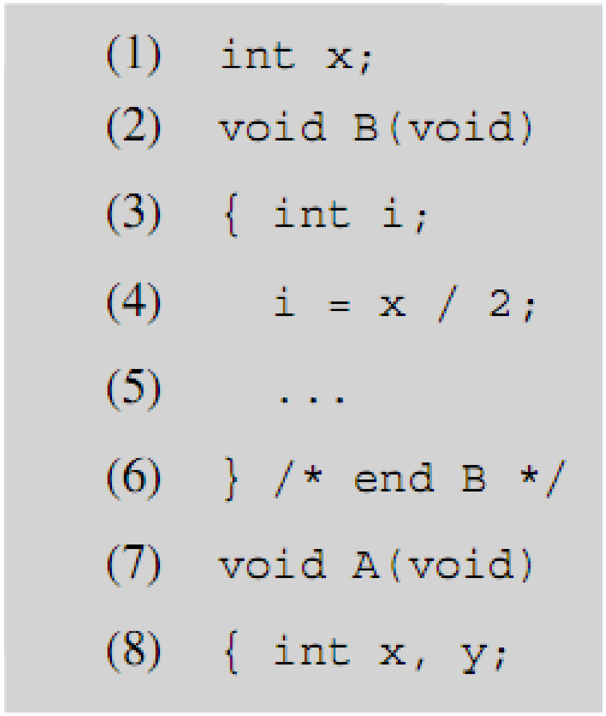


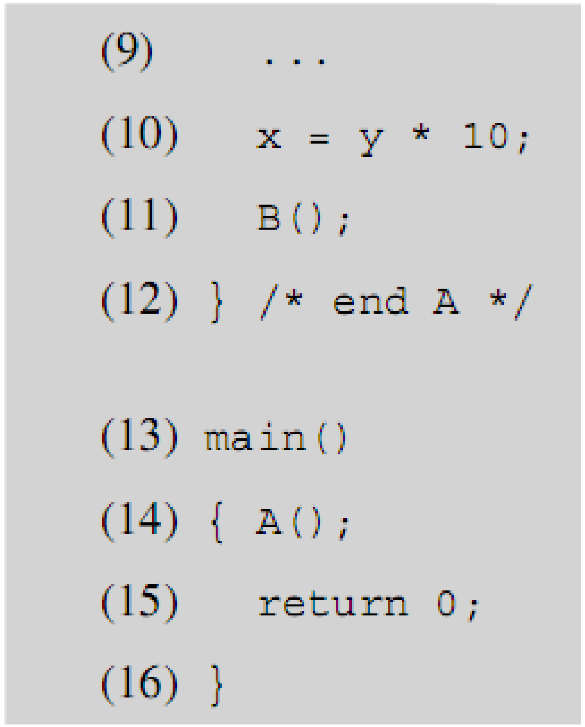
Figure 10.2 The activation records for blocks A and B

Procedure Semantics (cont'd.)

- Example: B is a procedure called from within A



```
int x;
void B(void)
{ int i;
  i = x / 2;
  ...
} /* end B */
void A(void)
{ int x, y;
```



```
  ...
  x = y * 10;
  B();
} /* end A */

main()
{ A();
  return 0;
}
```

Procedure Semantics (cont’d.)

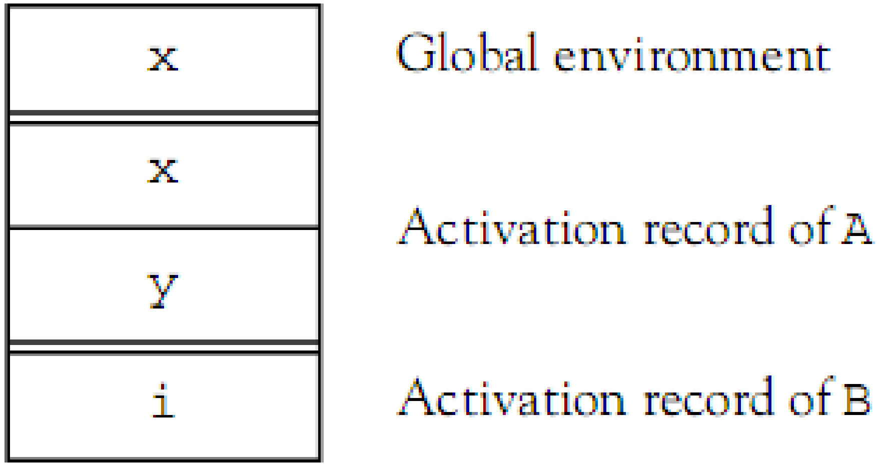


Figure 10.3 The activation records of the global environment, block A, and a procedure call of B

Procedure Semantics (cont'd.)

- **Defining environment** (or **static environment**) of B is the global environment
- **Calling environment** (or **dynamic environment**) of B is the activation record of A
- For blocks that are not procedures, the defining and calling environments are always the same
- A procedure can have any number of calling environments during which it will retain the same defining environment

Procedure Semantics (cont'd.)

- A nonprocedure block communicates with its surrounding block via nonlocal references
  - Lexical scoping allows it to access all variables in the surrounding block that are not redeclared in its own declarations
- A procedure block can only communicate with its defining block via references to nonlocal variables
  - It has no way of directly accessing the variables in its calling environment
  - It communicates with its calling environment through its **parameters**

Procedure Semantics (cont'd.)

- **Parameter list** is declared with the definition of the procedure
  - Parameters do not take on any value until they are replaced by arguments when the procedure is called
- Parameters are also called **formal parameters**, while arguments are called **actual parameters**
- One could make the case that procedures should communicate only using their parameters and should never use or change a nonlocal variable
  - To avoid dependencies (use) or side effects (change)

Procedure Semantics (cont'd.)

- While this is a good rule for variables, it is not good for functions and constants
- **Closed form**: procedures that depend only on parameters and fixed language features
- **Closure**: the code of a function together with a representation of its defining environment
  - Can be used to resolve all outstanding nonlocal references relative to the body of the function
  - Runtime environment must compute closures for all functions when needed

Parameter-Passing Mechanisms

- The nature of the bindings of arguments to parameters affects the semantics of procedure calls
- Languages differ significantly in the kinds of parameter-passing mechanisms available and the range of permissible implementation effects
- Four mechanisms will be discussed:
  - Pass by value
  - Pass by reference
  - Pass by value-result
  - Pass by name

Pass by Value

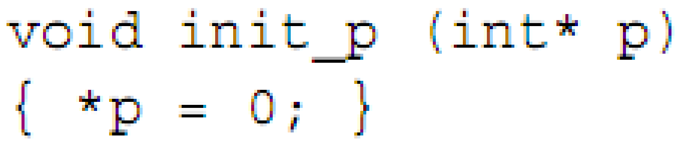
- **Pass by value**: most common mechanism for parameter passing
  - Arguments are evaluated at time of call, and their values become the values of the parameters
- In the simplest form of pass by value, value parameters behave as constant values during execution of the procedure
- Pass by value: a process in which all parameters in the procedure body are replaced by the corresponding argument values

Pass by Value (cont'd.)

- This mechanism is usually the only mechanism used in functional languages
- It is the default mechanism in C++ and Pascal and essentially the only mechanism in C and Java
- They use a slightly different interpretation of pass by value
  - Parameters are viewed as local variables of the procedure, with initial values given by argument values
  - Value parameters may be assigned but cause no changes outside the procedure

Pass by Value (cont'd.)

- In Ada, `in` parameters may not be assigned
- Pass by value does <u>not</u> imply that changes outside the procedure cannot occur through the use of parameters
  - A pointer or reference type parameter contains an address as its value, and this can be used to change memory outside the procedure
- Example: in C code



```
void init_p (int* p)
{ *p = 0; }
```

Pass by Value (cont'd.)

- Note: assigning directly to the pointer parameter does not change the argument outside the procedure

```
void init_ptr (int* p)

{ p = (int*) malloc(sizeof(int)); /* error - has no effect! */
}
```

- In some languages, certain values are implicitly pointers or references
  - Example: in C, arrays are implicitly pointers, so an array value parameter can be used to change values stored in the array

Pass by Value (cont’d.)

- In Java, object types are implicitly pointers, so an object parameter can be used to change its data
  - Direct assignments to parameters are not allowed

Pass by Reference

- **Pass by reference**: passes the location of the variable, making the parameter an **alias** for the argument
  - Any changes to the parameter occur to the argument as well
- Is the default for FORTRAN
  - Can be specified in C++ by using an ampersand (&) after the data type
  - Can be specified in Pascal by using the `var` keyword before the variable name

Pass by Reference (cont’d.)

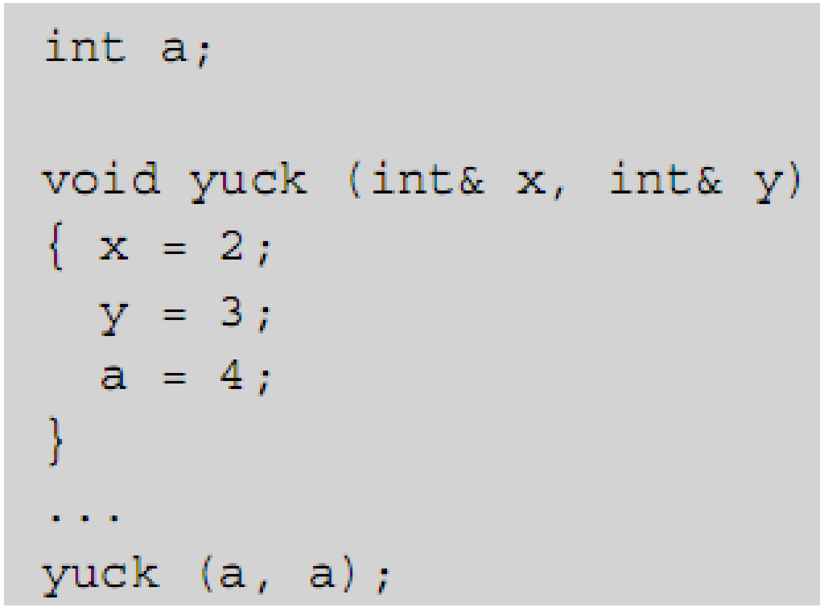
- Example:
  - After a call to `inc(a)` the value of a has increased by 1 so that a side effect has occurred

```
void inc(int& x) // C++
{ x++ ; }

procedure inc(var x: integer); (* Pascal *)
begin
  x :=   x + 1;
end;
```

Pass by Reference (cont'd.)

- Example: multiple aliasing is also possible
  - Inside procedure yuck after the call, x, y, and a all refer to the same variable, namely a



```
int a;

void yuck (int& x, int& y)
{ x = 2;
  y = 3;
  a = 4;
}
...
yuck (a, a);
```

Pass by Reference (cont'd.)

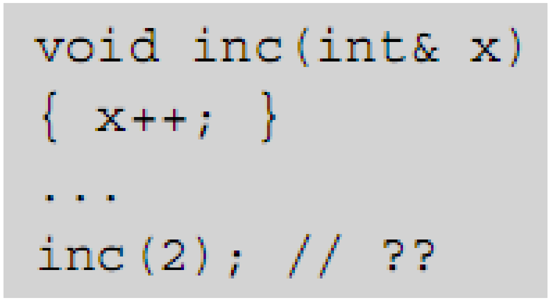
- Can achieve pass by reference in C by passing a reference or location explicitly as a pointer

```
void inc (int* x) /*C imitation of pass by reference */
{ (*x)++; /* adds 1 to *x */ }
...
int a;
...
inc(&a); /* pass the address of a to the inc function */
```

  – Note the necessity of explicitly taking the address of variable a and then explicitly dereferencing it again in the body of `inc`

Pass by Reference (cont'd.)

- How should reference arguments that are not variables be handled?
- Example: in C++ code



```
void inc(int& x)
{ x++; }
...
inc(2); // ??
```

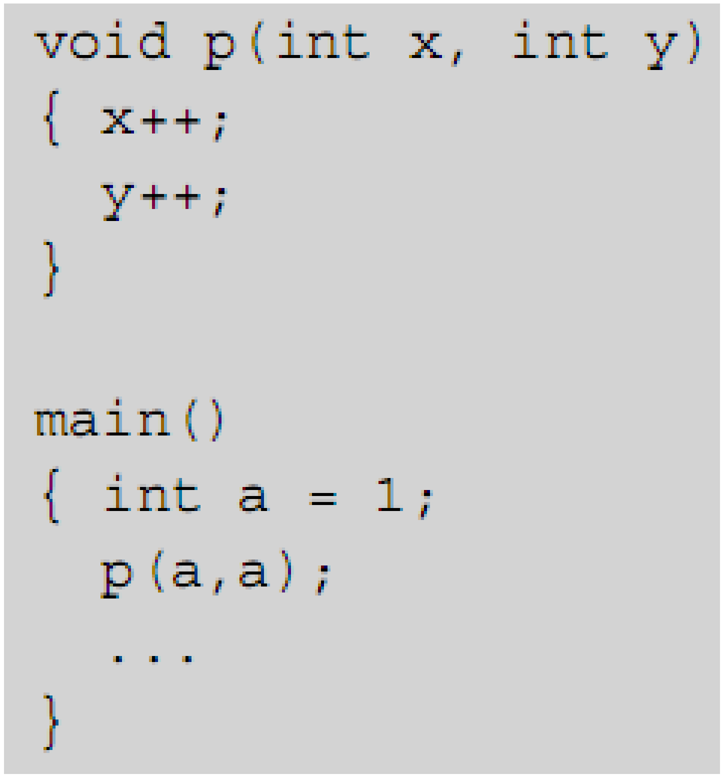
  - FORTRAN creates a temporary integer location, initializes it to the value 2, then applies the `inc` function
  - This is an error in C++ and Pascal

Pass by Value-Result

- **Pass by value-result**:
  - Value of the argument is copied and used in the procedure
  - Final value of the parameter is copied back out to the argument location when the procedure exits
  - Also called **copy-in**, **copy-out**, or **copy-restore**
- Pass by value-result can only be distinguished from pass by reference when using aliasing

Pass by Value-Result (cont'd.)

- Example: in C code
  - If pass by reference, a has value 3 after p is called
  - If pass by value-result, a has value 2 after p is called



```
void p(int x, int y)
{ x++;
  y++;
}

main()
{ int a = 1;
  p(a,a);
  ...
}
```

Pass by Value-Result (cont'd.)

- Issues that must be addressed include:
  - Order in which results are copied back to arguments
  - Whether locations of arguments are calculated only on entry and stored, or are recalculated on exit
- Another option is the **pass by result** mechanism:
  - There is no incoming value, only an outgoing one

Pass by Name and Delayed Evaluation

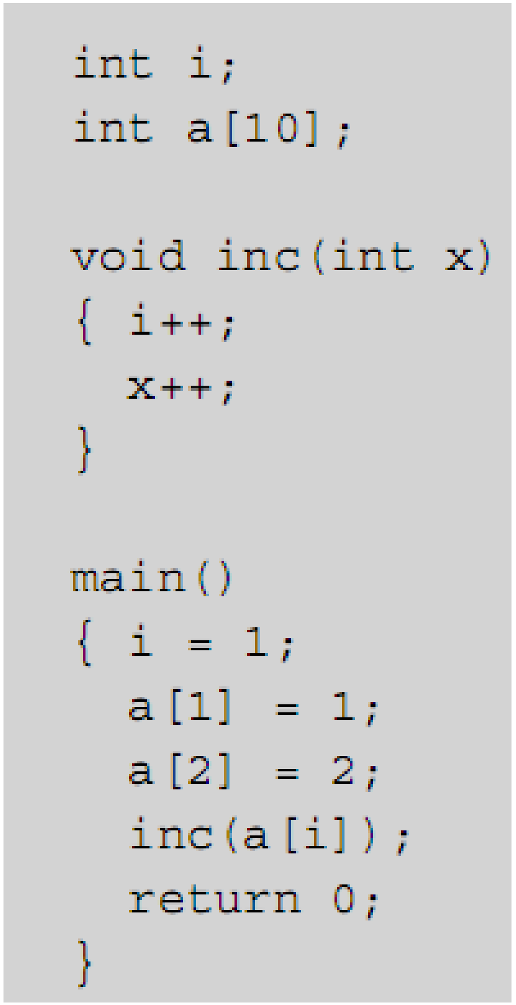
- **Pass by name**: introduced in Algol60
  - Intended as a kind of advanced inlining process for procedures
  - Is essentially equivalent to normal order delayed evaluation
  - Is difficult to implement and has complex interactions with other language constructs, especially arrays and assignment
- It should be understood as a basis for the lazy evaluation studied in Chapter 3

Pass by Name and Delayed Evaluation (cont'd.)

- In pass by name, the argument is not evaluated until its actual use as a parameter in the called procedure
  - The name of the argument replaces the name of the parameter to which it corresponds
- The text of an argument at the point of call is viewed as a function, which is evaluated every time the corresponding parameter name is reached in the procedure
  - However, the argument is always evaluated in the environment of the caller

Pass by Name and Delayed Evaluation (cont'd.)

- Example: in C code
  - The result of this code is to set `a[2]` to 3, leaving `a[1]` unchanged



```
int i;
int a[10];

void inc(int x)
{ i++;
  x++;
}

main()
{ i = 1;
  a[1] = 1;
  a[2] = 2;
  inc(a[i]);
  return 0;
}
```

```
#include <stdio.h>
int i;

int p(int y)
{ int j = y;
  i++;
  return j+y;
}

void q(void)
{ int j = 2;
  i = 0;
  printf("%d\n", p(i + j));
}

main()
{ q();
  return 0;
}
```

Figure 10.4 Pass by name example (in C syntax)

Pass by Name and Delayed Evaluation (cont'd.)

- Historically, the interpretation of pass by name arguments as functions to be evaluated when the procedure was called referred to the arguments as **thunks**
- Pass by name is problematic when side effects are desired
- Can exploit call by name in certain circumstances
  - **Jensen's device**: uses pass by name to apply an operation to an entire array

Pass by Name and Delayed Evaluation (cont'd.)

- Example: Jensen's device in C code:

```
int sum (int a, int index, int size)
{ int temp = 0;
  for (index = 0; index < size; index++)  temp += a;
  return temp;
}
```

  - If a and `index` are pass by name parameters, this code will compute the sum of all elements `x[0]` through `x[9]`:

```
int x[10], i, xtotal;
...
xtotal = sum(x[i],i,10);
```

Parameter-Passing Mechanism vs. Parameter Specification

- Ada has two notions of parameter communication, `in` parameters and `out` parameters
  - Any parameter can be declared `in`, `out`, or `in out`
  - `in` parameter represents an incoming value only
  - `out` parameter specifies an outgoing value only
  - `in out` parameter specifies both incoming and outgoing value
- Any parameter implementation can be used, as long as the appropriate values are communicated properly
  - An `in` value on entry and an `out` value on exit

Parameter-Passing Mechanism vs. Parameter Specification (cont'd.)

- Any program that violates these protocols is **erroneous**
  - `in` parameter cannot legally be assigned a new value
  - `out` parameter cannot legally be used by the procedure
- A translator can prevent many violations of parameter specifications

Type Checking of Parameters

- In strongly typed languages, procedure calls must be checked to ensure that arguments agree in type and number with the specified parameters
- This means that:
  - Procedures may not have a variable number of parameters
  - Rules must be stated for type compatibility between parameters and arguments
- For pass by reference, parameters usually must have the same type
  - This can be relaxed for pass by value

Procedure Environments, Activations, and Allocation

- The environment for a block-structured language with lexical scope can be maintained in a stack-based fashion
  - Activation record is created on the environment stack when a block is entered, and released when the block is exited
- Same structure can be extended to procedure activations in which the defining and calling environments differ
- **Closure** is necessary to resolve nonlocal references

Procedure Environments, Activations, and Allocation (cont'd.)

- Must understand this execution model to fully understand the behavior of programs
  - Semantics of procedure calls are embedded in this model
- A completely stack-based environment is not adequate to deal with procedure variables and dynamic creation of procedures
  - Languages with these facilities (particularly functional languages) must use a more complex fully dynamic environment with garbage collection

Fully Static Environments

- In Fortran77, all memory allocation can be performed at load time
- All variable locations are fixed throughout program execution
- Function and procedure definitions cannot be nested
  - All procedures/functions are global
- Recursion is not allowed
- All information associated with a function or subroutine can be statically allocated

Fully Static Environments (cont'd.)

- Each procedure or function has a fixed activation record containing space for local variables and parameters
- Global variables are defined by COMMON statements
  - They are determined by pointers to a common area

Fully Static Environments (cont'd.)

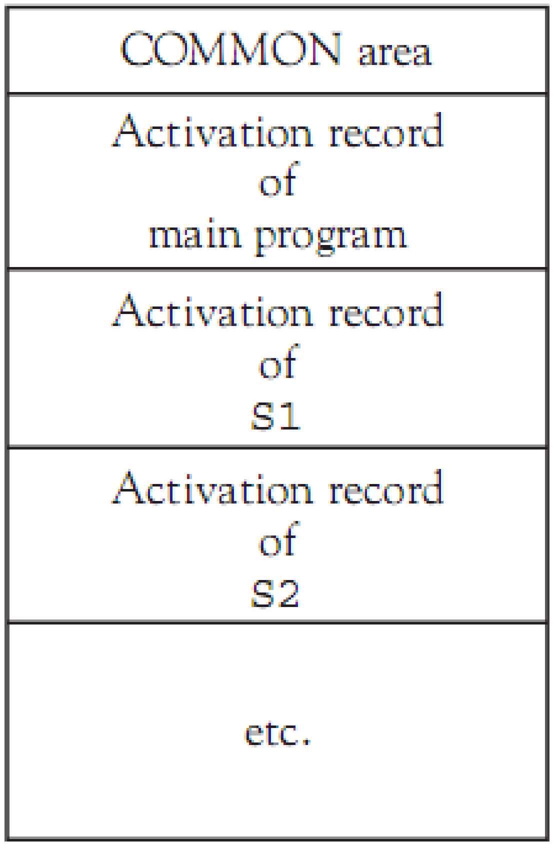


Figure 10.5 The runtime environment of a FORTRAN program with subprograms S1 and S2

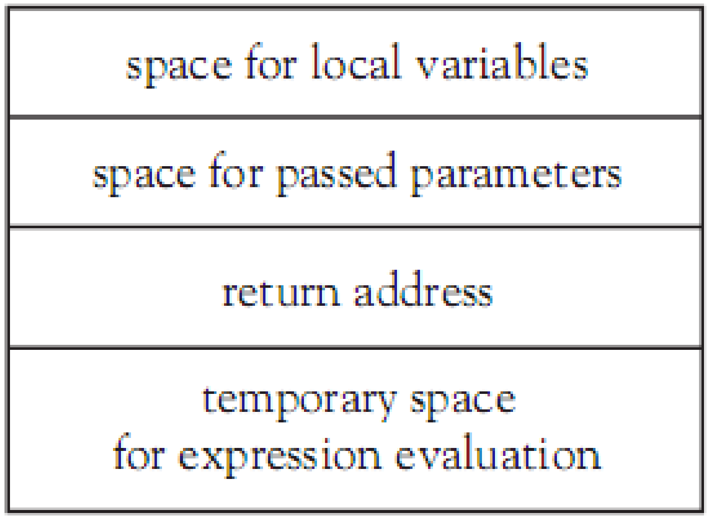
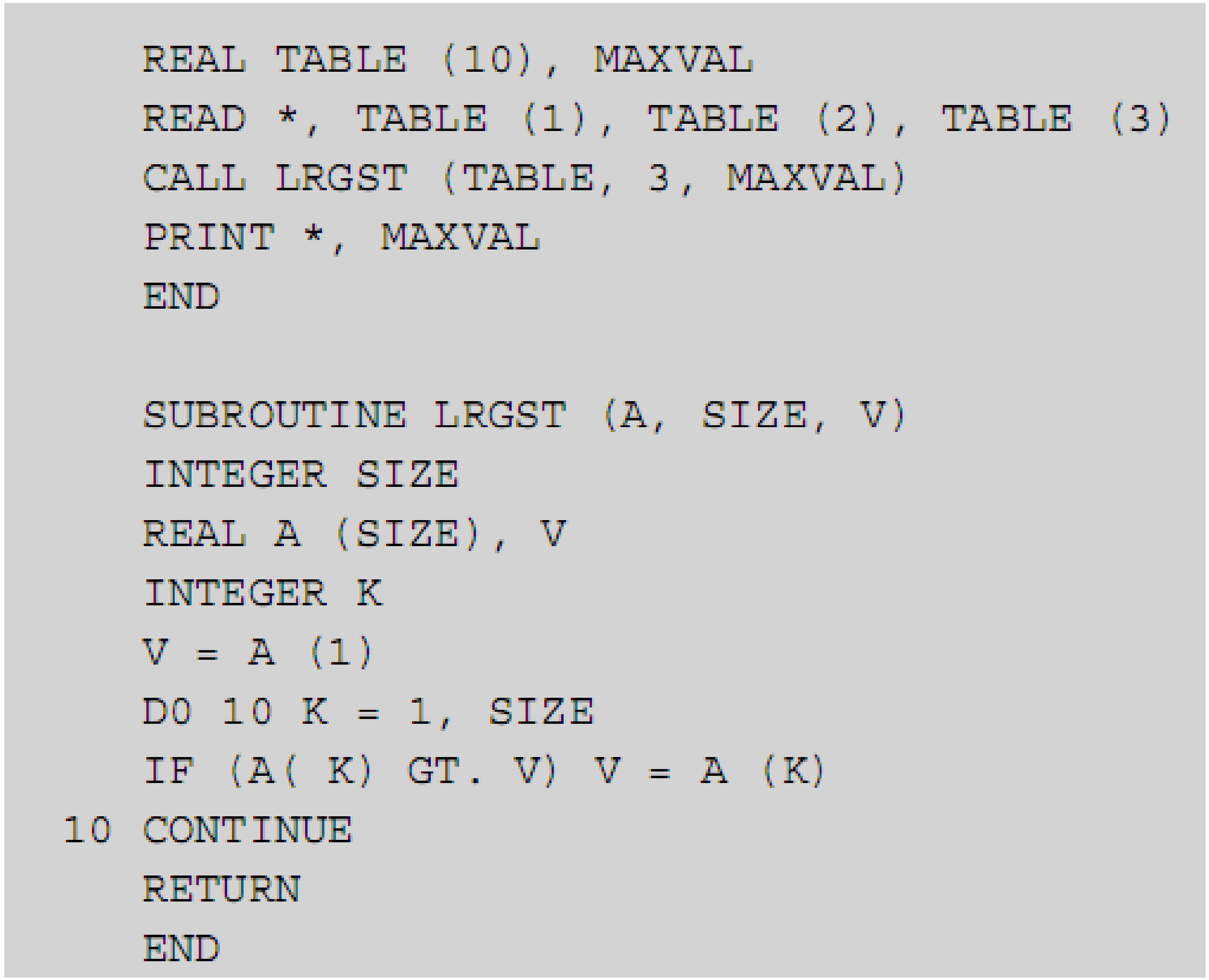

space for local variables
space for passed parameters
return address
temporary space for expression evaluation

Figure 10.6 The component areas of an activation record

Fully Static Environments (cont'd.)



```
   REAL TABLE (10), MAXVAL
   READ *, TABLE (1), TABLE (2), TABLE (3)
   CALL LRGST (TABLE, 3, MAXVAL)
   PRINT *, MAXVAL
   END

   SUBROUTINE LRGST (A, SIZE, V)
   INTEGER SIZE
   REAL A (SIZE), V
   INTEGER K
   V = A (1)
   D0 10 K = 1, SIZE
   IF (A( K) GT. V) V = A (K)
10 CONTINUE
   RETURN
   END
```

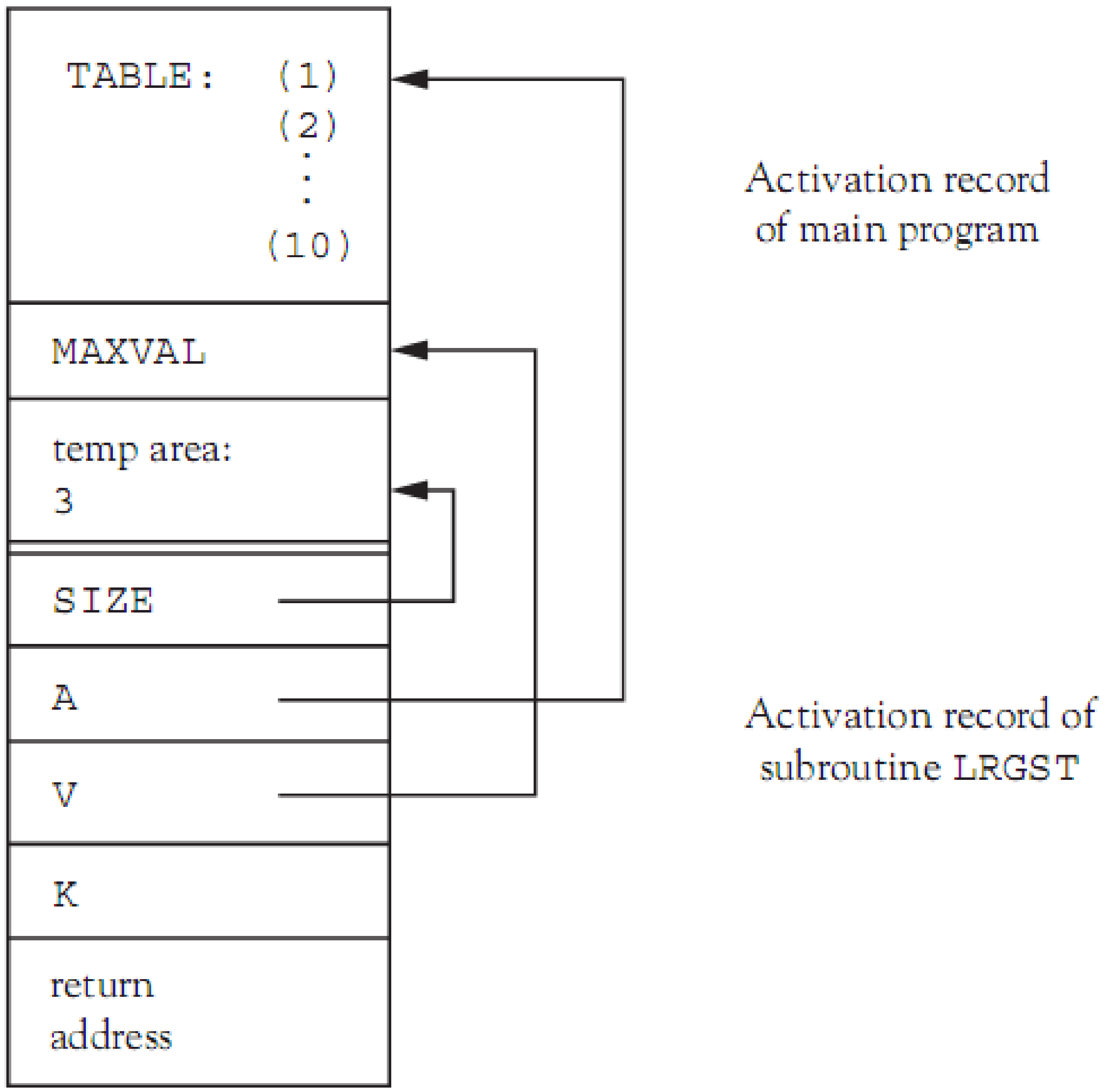


TABLE: (1)
(2)
(10)
MAXVAL
temp area:
3
SIZE
A
V
K
return address
Activation record of main program
Activation record of subroutine LRGST

Figure 10.7 The runtime environment immediately after a call to the subroutine LRGST

Stack-Based Runtime Environments

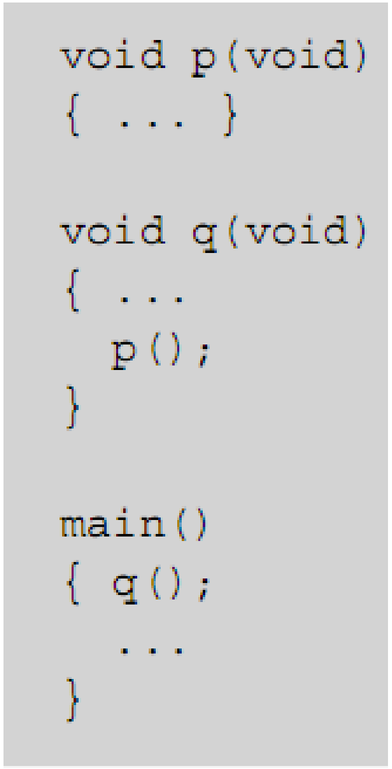
- In a block-structured language with recursion, activations of procedure blocks cannot be allocated statically
  - A procedure may be called again before its previous activation is exited, so a new activation must be created on each procedure entry
- A pointer to the current activation is required, since each procedure has no fixed location for its activation record
  - Usually kept in a register called the **environment pointer**, or **ep**

Stack-Based Runtime Environments (cont'd.)

- Must also maintain a pointer to the activation record of the block from which the current activation was entered
  - If a procedure call, this is the activation of the caller
- The ep must be restored to point to the previous activation
  - Previous location's pointer is called the **control link** (or **dynamic link**)

Stack-Based Runtime Environments (cont'd.)

- Example: in C code



```
void p(void)
{ ... }

void q(void)
{ ...
  p();
}

main()
{ q();
  ...
}
```

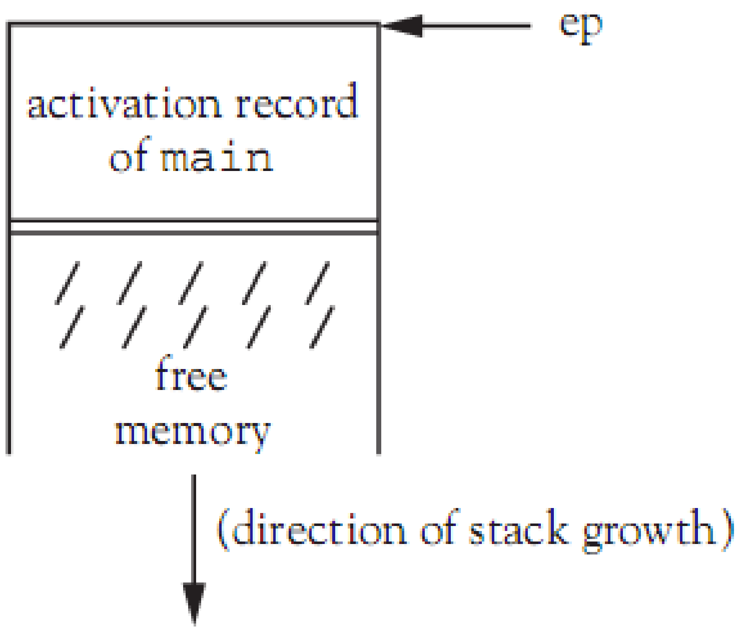


Figure 10.8 The runtime environment after main begins executing

Stack-Based Runtime Environments (cont'd.)

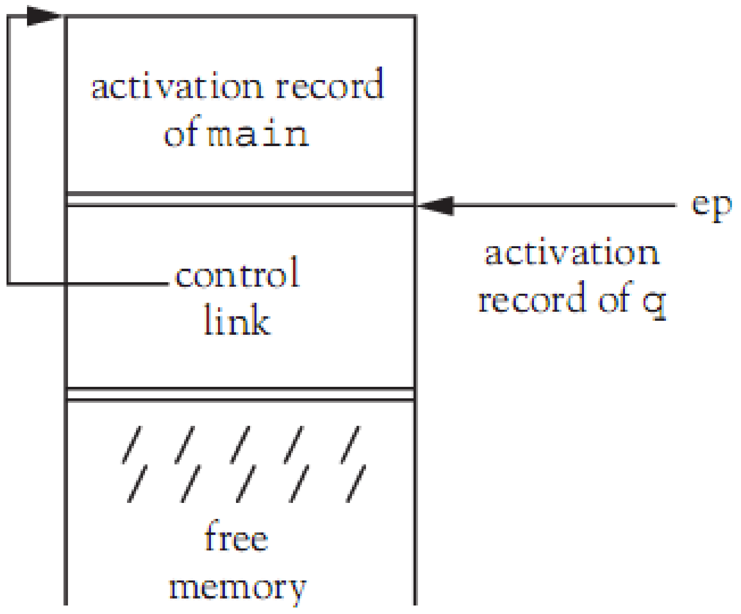


Figure 10.9 The runtime environment after q begins executing

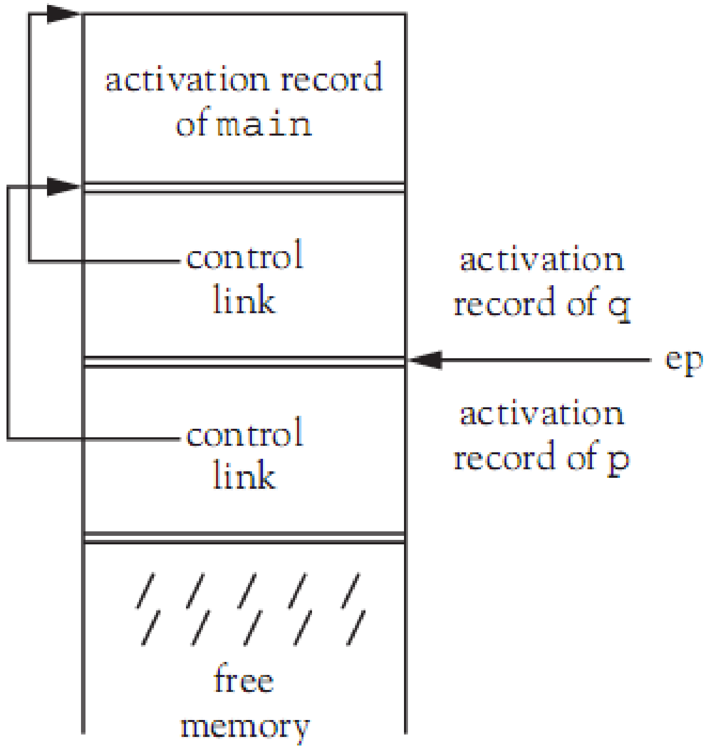


Figure10.10 The runtime environment after p is called within q

Stack-Based Runtime Environments (cont'd.)

- The fields in each activation record need to contain this information:

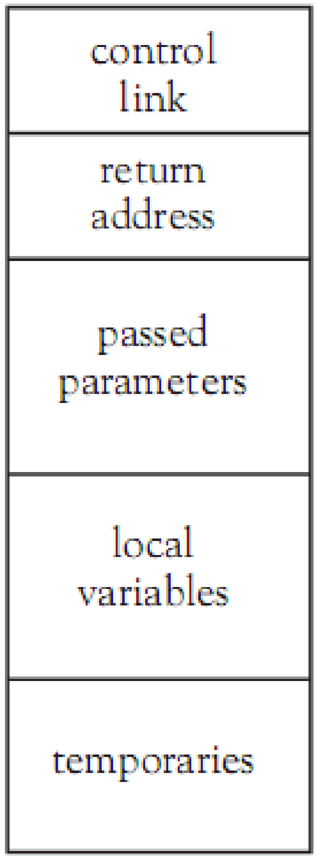


Figure 10.11 The component areas of an activation record with a control link

Stack-Based Runtime Environments (cont'd.)

- Local variables are allocated in the current activation record, in the same order each time, because they are static
- Each variable is thus allocated the same position in the activation record relative to the beginning of the record
  - This is called the **offset** of the local variable

Stack-Based Runtime Environments (cont'd.)

```
int x;

void p( int y)
{ int i = x;
  char c;
  ...
}
void q ( int a)
{ int x;
  ...
  p(1);
}

main()
{ q(2);
  return 0;
}
```

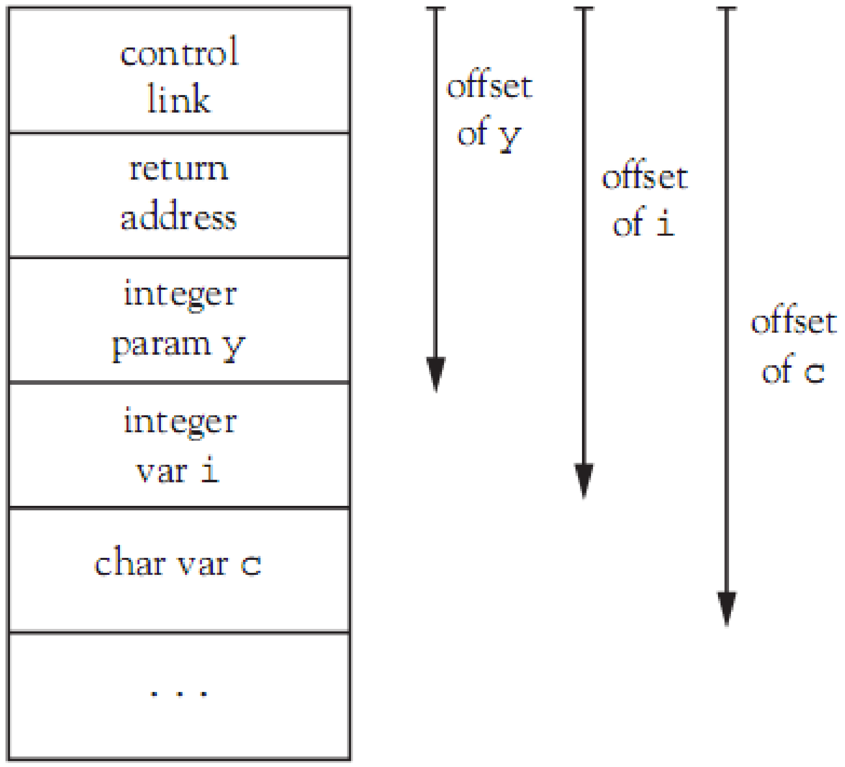


Figure 10.12 An activation record of p showing the offsets of the parameter and temporary variables

Stack-Based Runtime Environments (cont'd.)

- Since procedures cannot be nested in FORTRAN or C, nonlocal references outside a procedure are actually global and are allocated statically
  - No additional structures are required in the activation stack
- When nested procedures are permitted, nonlocal references to local variables are permissible in a surrounding procedure scope

Stack-Based Runtime Environments (cont'd.)

```
procedure q is
  x: integer;

  procedure p (y: integer) is
    i: integer := x;
  begin
    ...
  end p;

  procedure r is
  x: float;
  begin
    p(1);
    ...
  end r;

begin
  r;
end q;
```

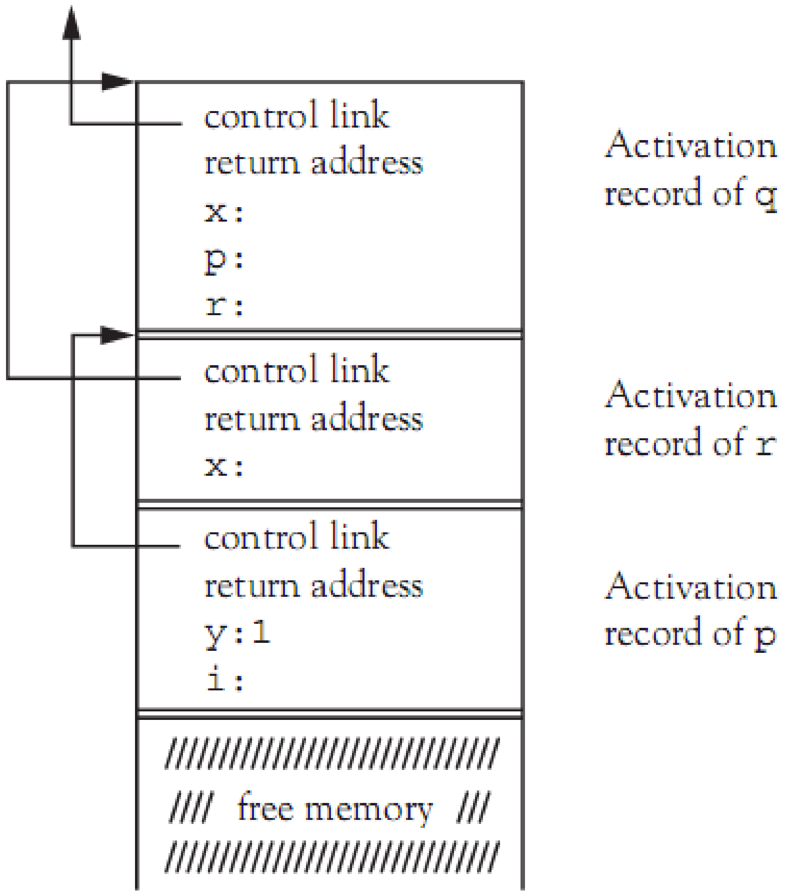


Figure 10.13 The runtime environment with activations of p, r, and q

Stack-Based Runtime Environments (cont’d.)

- To achieve lexical scope, a procedure must maintain a link to its **lexical** or **defining environment**
  - This link is called the **access link** (or **static link**)
- Each activation record needs an access link field
- When blocks are deeply nested, it may be necessary to follow multiple access links to find a nonlocal reference

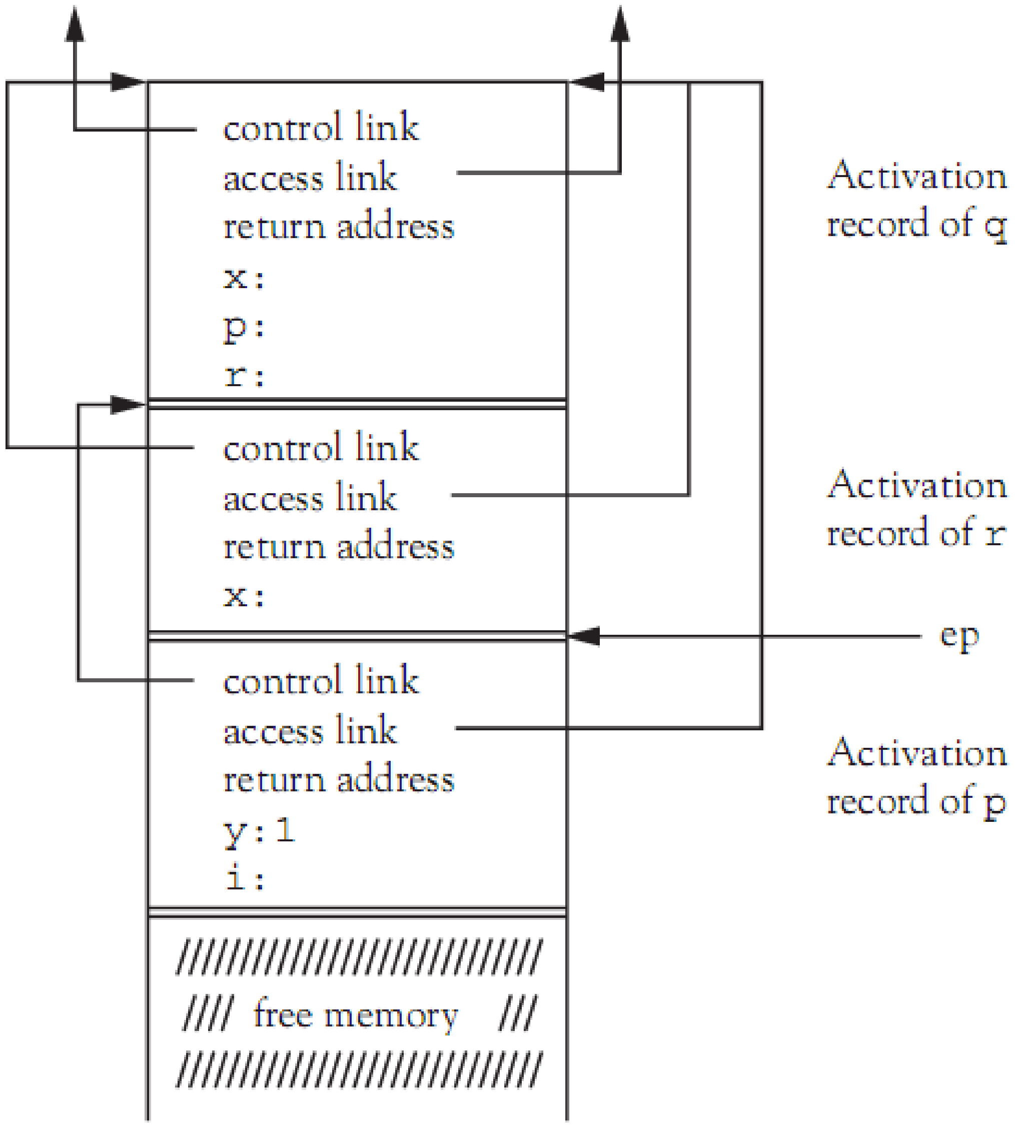
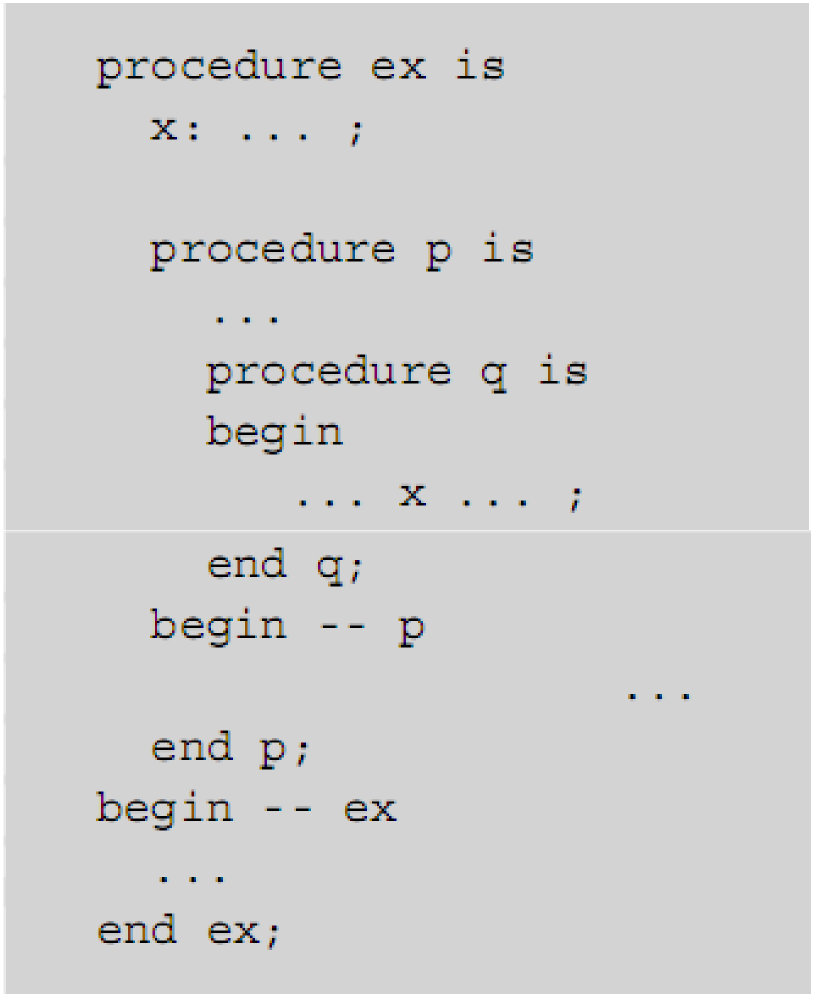


Figure 10.14 The runtime environment with activations of p, r, and q with access links

Stack-Based Runtime Environments (cont'd.)

- Example: in Ada code
  - To access x from inside q, must follow the access link in the activation record of q
  - Then use the access link to the activation record of p
  - Then follow the access link to the global environment



```
procedure ex is
  x: ... ;

  procedure p is
    ...
    procedure q is
    begin
       ... x ... ;
    end q;
  begin -- p
                           ...
  end p;
begin -- ex
   ...
end ex;
```

Stack-Based Runtime Environments (cont’d.)

- This process is called **access chaining**
- The number of access links that must be followed corresponds to the difference in nesting levels (or **nesting depth**) between the accessing environment and the defining environment of the variable being accessed
- Closure (the procedure code together with the mechanism for resolving nonlocal references) is significantly more complex

Stack-Based Runtime Environments (cont'd.)

- Closure requires two pointers:
  - Code or instruction pointer (`ip`)
  - Access link, or environment pointer of its defining environment (`ep`)
- Closure will be denoted by `<ep, ip>`

```
with Text_IO; use Text_IO;
with Ada.Integer_Text_IO;
use Ada.Integer_Text_IO;

procedure lastex is

  procedure p(n: integer) is

    procedure show is
    begin
      if n > 0 then p(n - 1);

      end if;

      put(n);
      new_line;
    end show;
  begin -- p
    show;
  end p;
begin -- lastex
  p(1);
end lastex;
```

Figure 10.15 An Ada program with nested procedures

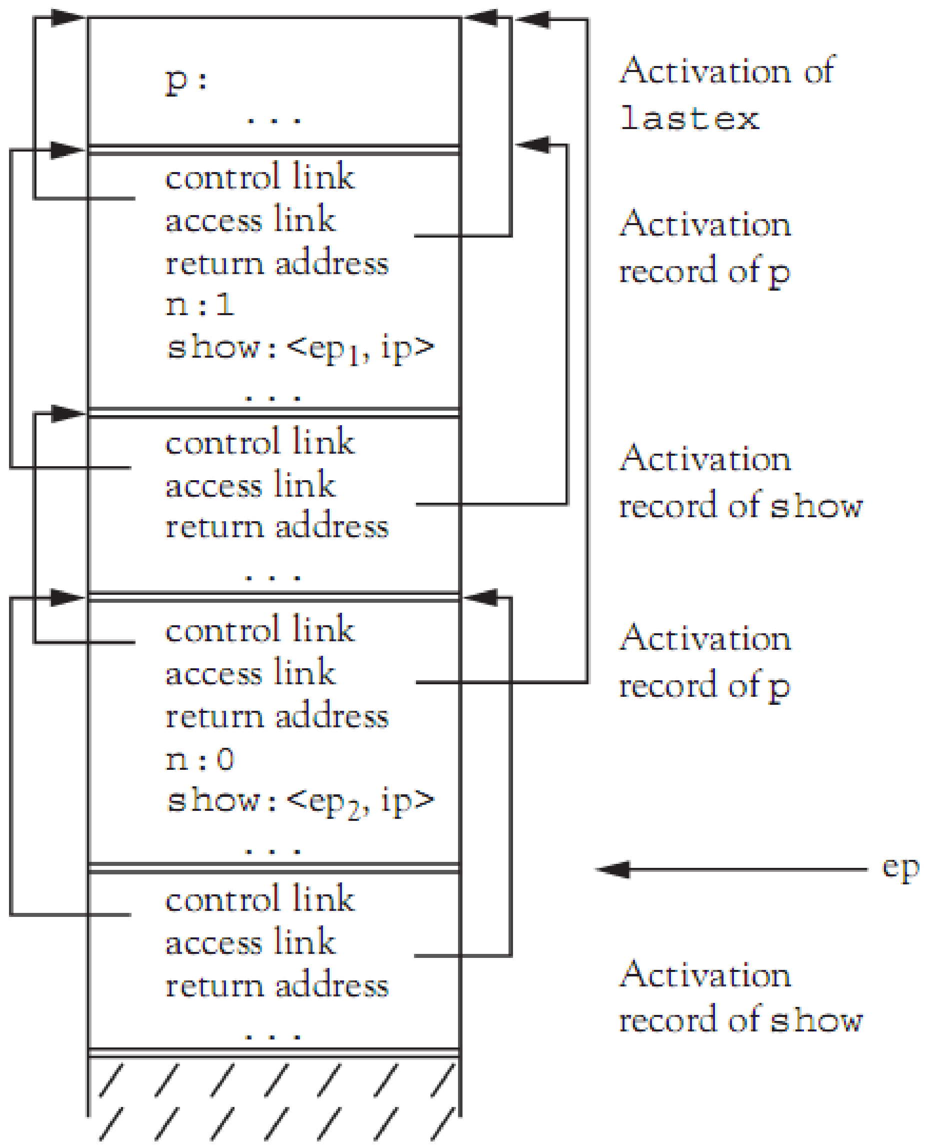


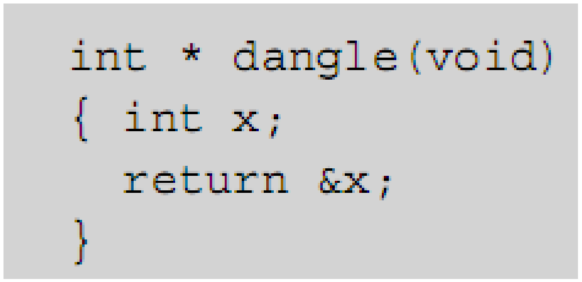
Figure 10.16 Environment of `lastex` during the second call to `show` (line 8 of Figure 10.15)

Dynamically Computed Procedures & Fully Dynamic Environments

- The use of `<ep, ip>` closures for procedures is suitable even for languages with procedures that are passed as parameters, as long as these parameters are value parameters
  - The procedure passed as a parameter is passed as an `<ep, ip>` pair
  - Its access link is the `ep` part of the closure
- This approach is used in Ada and Pascal
- A stack-based environment does have limitations

Dynamically Computed Procedures & Fully Dynamic Environments (cont'd.)

- Any procedure that can return a pointer to a local object will result in a dangling reference when the procedure is exited
- Example: in C code



```
int * dangle(void)
{ int x;
  return &x;
}
```

  – The assignment `addr = dangle()` will cause `addr` to point to an unsafe location in the activation stack

Dynamically Computed Procedures & Fully Dynamic Environments (cont'd.)

- Java does not allow this
- Ada95 makes it an error by stating the **Access-type Lifetime Rule**:
  - An attribute `x'access` yielding a result belonging to an access type `T` is only allowed if `x` can remain in existence at least as long as `T`
- If procedures can be created dynamically and returned from other procedures, they become first-class values
  - This flexibility is usually desired in a functional language

Dynamically Computed Procedures & Fully Dynamic Environments (cont'd.)

- A stack-based environment cannot be used, since the closure of a locally defined procedure will have an ep pointing to the current activation record
- If this closure is available outside the activation of the creating procedure, the ep will point to an activation record that no longer exists

```
type WithdrawProc is
  access function (x:integer) return integer;

InsufficientFunds: exception;

function makeNewBalance (initBalance: integer)
   return WithDrawProc
is
   currentBalance: integer;

   function withdraw (amt: integer) return integer is
   begin
     if amt <= currentBalance then
       currentBalance := currentBalance - amt;
     else
       raise InsufficientFunds;
     end if;
     return currentBalance;
   end withdraw;

begin
   currentBalance := initBalance;
   return withdraw'access;
end makeNewBalance;
```

Dynamically Computed Procedures & Fully Dynamic Environments (cont'd.)

- After executing the following code:

```
withdraw1, withdraw2: WithdrawProc;
withdraw1 := makeNewBalance(500);
withdraw2 := makeNewBalance(100);
```

  – If we now execute this code:

```
newBalance1 := withdraw1(100);
newBalance2 := withdraw2(50);
```

  – We should get 400 for `newBalance1` and 50 for `newBalance2`
  – If the two instances of the local `currentBalance` variable have disappeared from the environment, these calls will not work

Dynamically Computed Procedures & Fully Dynamic Environments (cont'd.)

- In LISP, functions and procedures are first-class values
  - No nongeneralities or nonorthogonalities should exist
- **Fully dynamic** environment: activation records are deleted only when they can no longer be reached from within the executing program
- Must be able to reclaim unreachable storage
  - Two methods for this are **reference counts** and **garbage collection**

Dynamically Computed Procedures & Fully Dynamic Environments (cont’d.)

- The structure of activation records becomes treelike instead of stacklike
  - Control links to the calling environment no longer necessarily point to the immediately preceding activation
- This is the model under which Scheme and other functional languages execute

Dynamically Computed Procedures & Fully Dynamic Environments (cont'd.)

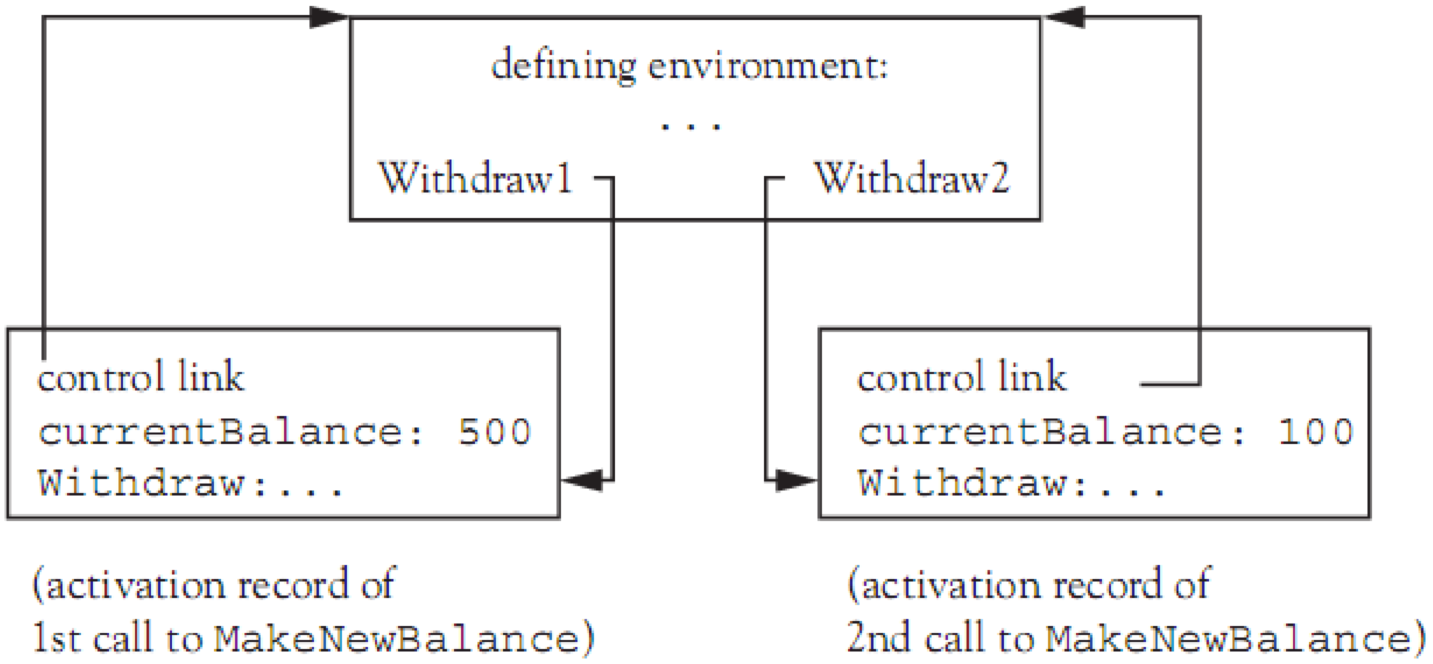


Figure 10.17 A tree-like runtime environment for a language with first-class procedures

Dynamic Memory Management

- In a typical imperative language such as C, automatic allocation and deallocation of storage occurs only for activation records on a stack
- Explicit dynamic allocation and use of pointers is available under manual programmer control using a memory **heap** separate from the stack
  - Automatic garbage collection is desirable for the heap
- Any language that does not apply significant restrictions to the use of procedures and functions must provide automatic garbage collection

Dynamic Memory Management (cont'd.)

- Two categories of automatic memory management:
  - **Reclamation** of storage no longer used (previously called **garbage collection**)
  - **Maintenance** of free space available for allocation

Maintaining Free Space

- The free space in a contiguous block of memory provided to an executing program is maintained by using a list of free blocks
  - Can be done via a linked list

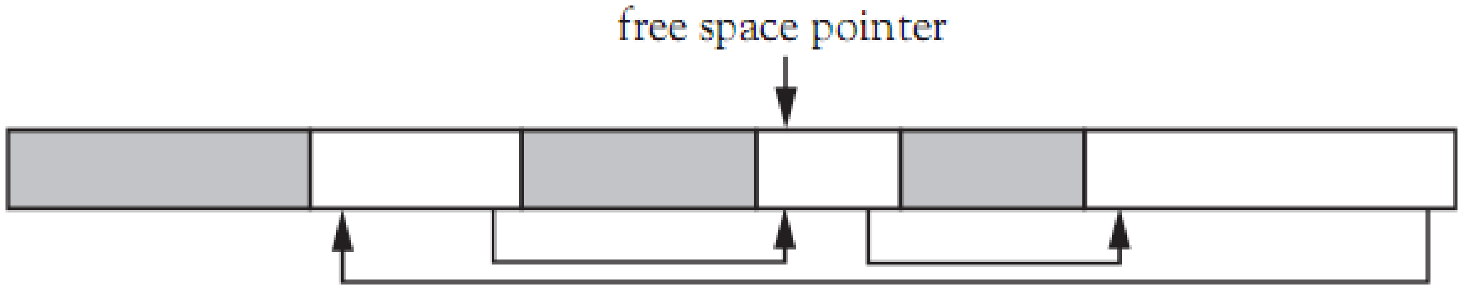


Figure 10.18 The free space represented as a linked list

Maintaining Free Space (cont'd.)

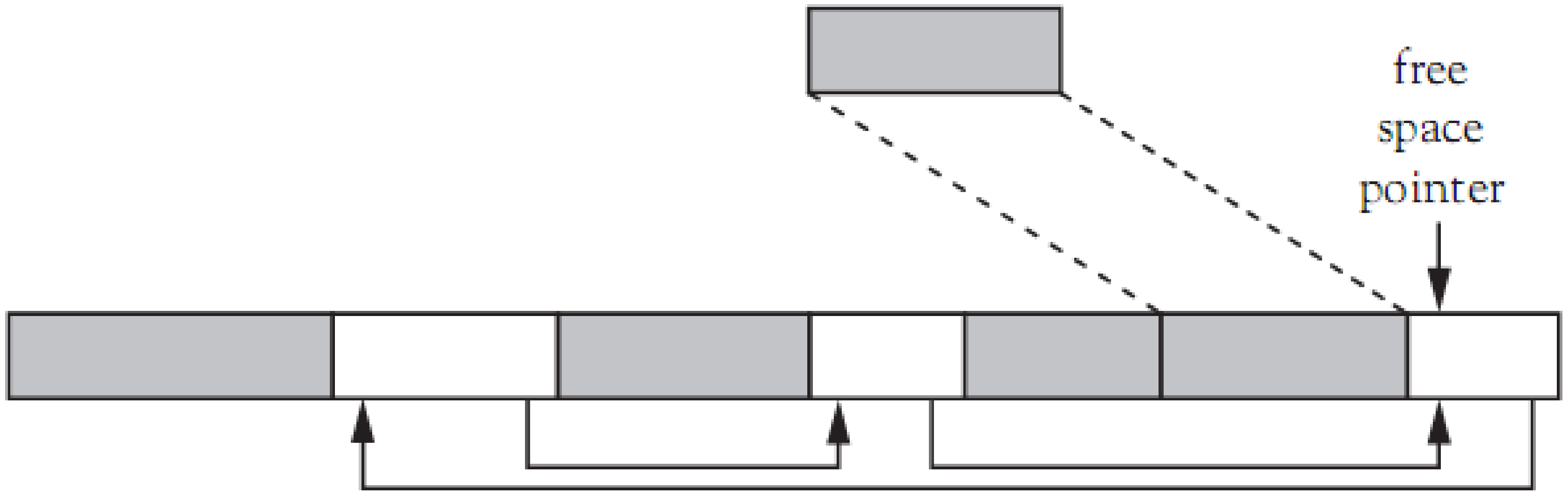


Figure 10.19 Allocating storage from the free space

Maintaining Free Space (cont'd.)

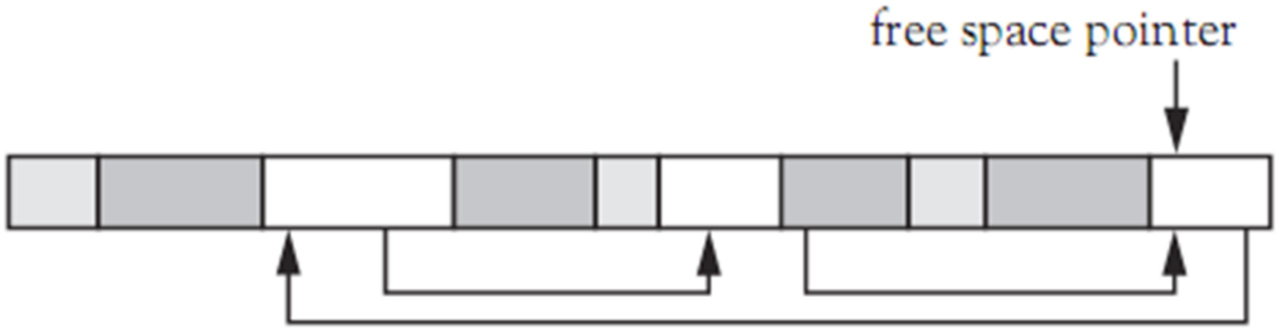


Figure 10.20 Returning storage to the free list

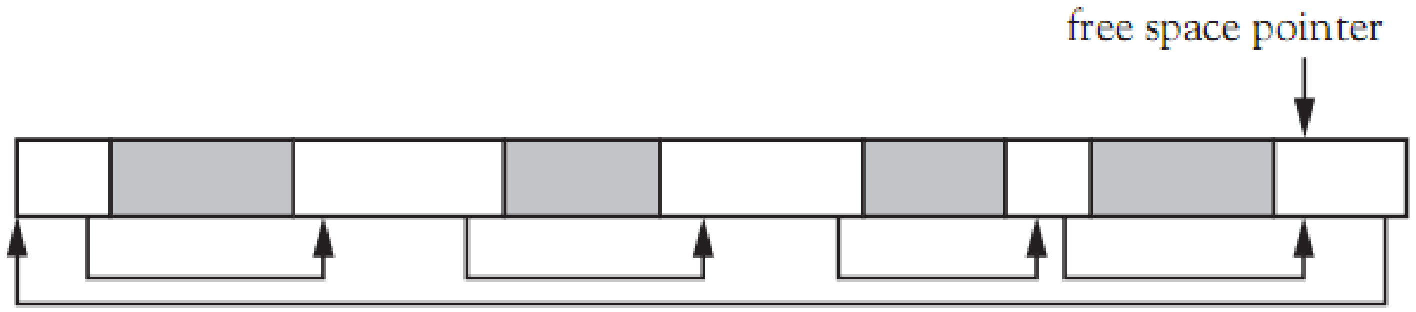


Figure 10.21 The free list after the return of storage

Maintaining Free Space (cont'd.)

- **Coalescing**: process of joining immediately adjacent blocks of free memory to form the largest contiguous block of free memory
- A free list can become **fragmented**
  - This may cause the allocation of a large block to fail
- Memory must occasionally be **compacted** by moving all free blocks together and coalescing them into one block
- Storage compaction involves considerable overhead

Maintaining Free Space (cont’d.)

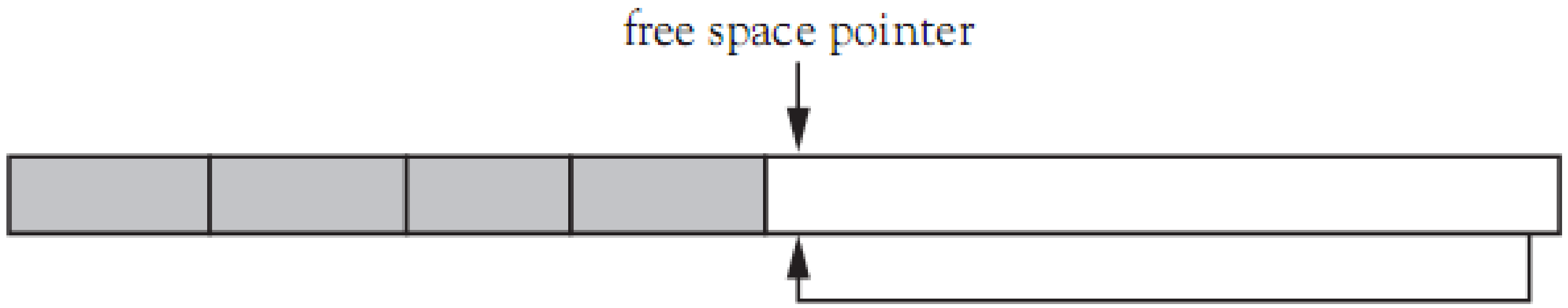


Figure 10.22 The free list after compaction

Reclamation of Storage

- Two main methods for recognizing when a block of storage is no longer referenced:
  - Reference counting
  - Mark and sweep
- **Reference counting**: an eager method of reclamation
  - Each block of allocated storage contains a field that stores the count of references to that block from other blocks
  - When count drops to 0, the block can be returned to the free list

Reclamation of Storage (cont'd.)

- Drawbacks of reference counting:
  - Extra memory required to keep the counts
  - Effort to maintain the counts is fairly large, because reference counts must be decremented recursively
  - Circular references can cause unreferenced memory to never be deallocated

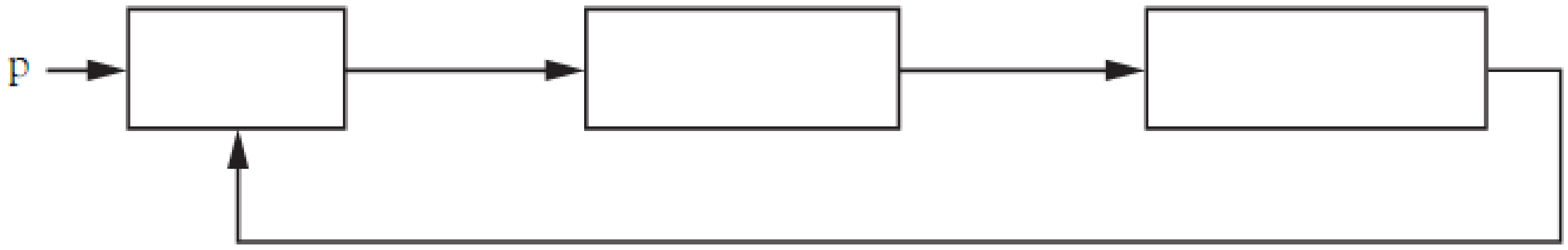


Figure 10.23 A circular linked list

Reclamation of Storage (cont'd.)

- **Mark and sweep**: a lazy method that puts off reclaiming any storage until the allocator runs out of space
  - It looks for all storage that can be referenced
  - Moves all unreferenced storage back to the free list
- Mark and sweep occurs in two passes:
  - First pass follows all pointers recursively and marks each block of storage reached
  - Second pass sweeps linearly through memory, returning unmarked blocks to the free list

Reclamation of Storage (cont’d.)

- Drawback of mark and sweep:
  - Double pass through memory causes a significant delay in processing, which can occur every few minutes
- An alternative is to split available memory into two halves and allocate storage only from one half at a time
  - During marking pass, reached blocks are copied to the other half
  - Called **stop and copy**
  - Does little to improve processing delays

Reclamation of Storage (cont'd.)

- **Generational garbage collection**: invented in 1980s
  - Adds a permanent storage area to the prior scheme
  - Allocated objects that survive long enough are copied into permanent space and never deallocated during subsequent reclamations
  - Reduces the amount of memory to be searched
- This method works very well, especially with a virtual memory system

Exception Handling and Environments

- Raising and handling exceptions are similar to procedure calls and can be implemented in similar ways
- There are some differences, however:
  - An activation cannot be created on the stack for raising an exception, since the stack may be unwound when searching for a handler
  - A handler must be found and called dynamically
  - Exception type information must be retained since the handler actions are based on the exception type, not the exception value

Exception Handling and Environments (cont'd.)

- When an exception is raised, no activation record is created
  - The exception object and its type information is stored in a known location, such as a register
  - A jump is performed to generic code that looks for a handler
  - Exit code is called if a handler is not found
  - The return address for the successful handling of an exception must also be stored in a known location
- This process handles the first difference

Exception Handling and Environments (cont'd.)

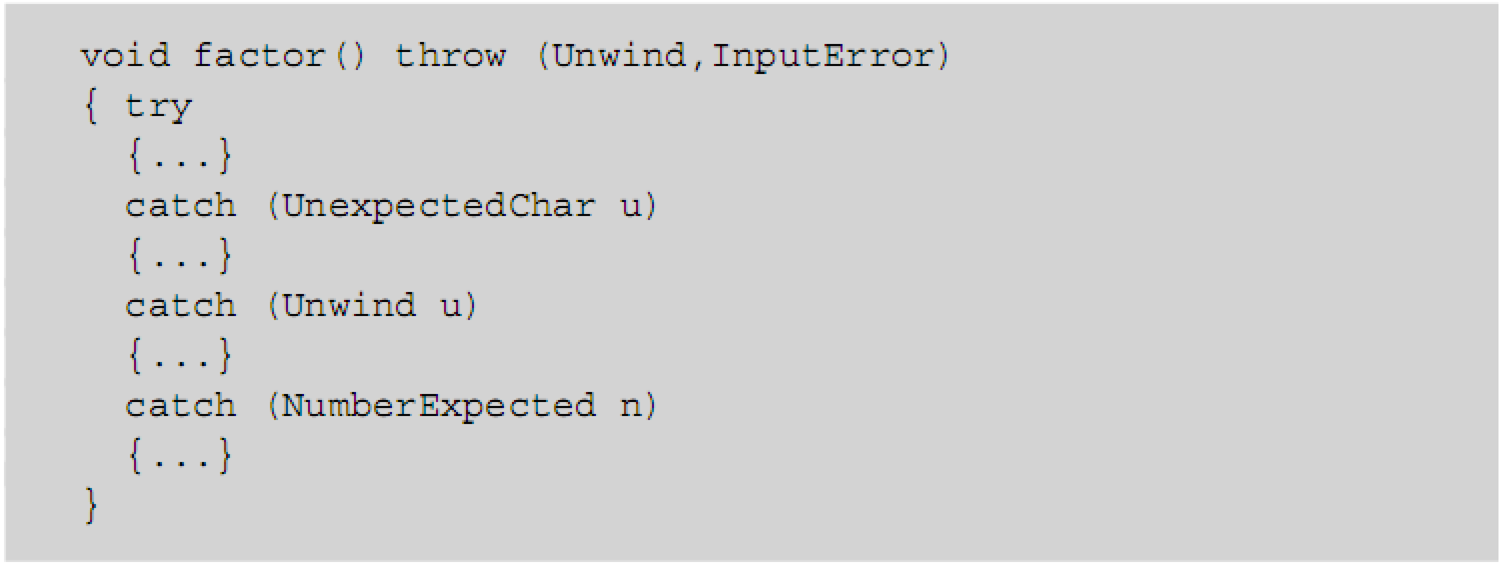
- To deal with the second difference, pointers to handlers must be kept on some kind of stack
  - When a procedure that has an associated handler is entered, a pointer to the handler must be recorded on the stack
  - When exited, the handler pointer must be removed from the stack
- To implement this stack directly, it must be maintained either on the heap or elsewhere in its own memory area (not on the runtime stack)

Exception Handling and Environments (cont'd.)

- The third difference relates to how to record the needed type information without adding overhead in the exception structures themselves
  - A lookup table might be used
- The basic implementation of handlers is relatively straightforward
  - Collect all handler code for a particular block into a single handler implemented as a switch statement
- Main problem is that maintenance of the handler stack causes a potentially significant runtime penalty

Exception Handling and Environments (cont'd.)

- Example:



```
void factor() throw (Unwind,InputError)
{ try
    {...}
    catch (UnexpectedChar u)
    {...}
    catch (Unwind u)
    {...}
    catch (NumberExpected n)
    {...}
}
```

Case Study: Processing Parameter Modes in TinyAda

- In TinyAda:
  - An identifier can be a parameter
  - Parameters are subject to the same restrictions as variables
  - Neither parameters nor variables can appear in static expressions
- Parameters can be further specified in terms of the roles they play, called parameter modes
  - Modes allow us to apply a new set of constraints during semantic analysis

The Syntax and Semantics of Parameter Modes

- Syntax rules for TinyAda’s parameter modes:

```
parameterSpecification = identifierList ":" mode <type>name

mode = [ "in" ] | "in" "out" | "out"
```

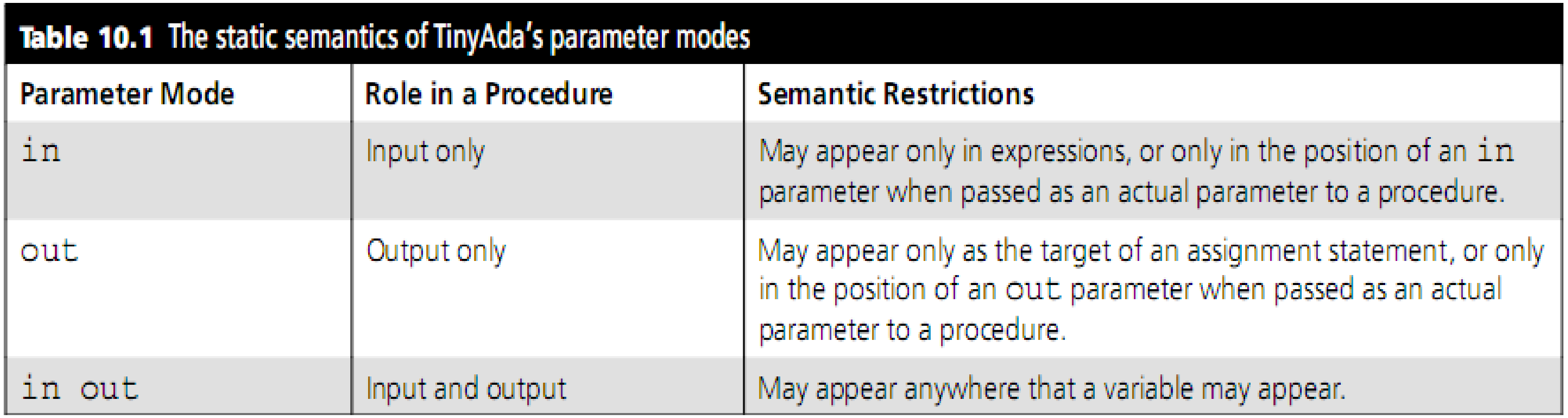


Table 10.1 The static semantics of TinyAda’s parameter modes

Parameter Mode	Role in a Procedure	Semantic Restrictions
`in`	Input only	May appear only in expressions, or only in the position of an `in` parameter when passed as an actual parameter to a procedure.
`out`	Output only	May appear only as the target of an assignment statement, or only in the position of an `out` parameter when passed as an actual parameter to a procedure.
`in out`	Input and output	May appear anywhere that a variable may appear.

Processing Parameter Declarations

- The `SymbolEntry` class is modified to include a mode field
  - Possible values are `SymbolEntry.IN`, `SymbolEntry.OUT`, and `SymbolEntry.IN_OUT`

Processing Parameter References

- If a name in an expression is a parameter, then its mode cannot be OUT
- If a name in the target of an assignment statement is a parameter, then its mode cannot be IN
- The number and types of actual parameters must be matched against the procedure’s formal parameters
  - The mode must also now be examined to apply the new constraints