Subroutines and Control Abstraction

CSCI 3136

Principles of Programming Languages

Faculty of Computer Science

Dalhousie University

Winter 2012

Reading: Chapter 8

Abstractions as Program Building Blocks

Programming is about ***building abstractions***.

Subroutines are the main method to build ***control abstractions***.

The other form of abstraction we normally think about is ***data abstraction*** (next topic).

Basic Definitions

Subroutines are what we normally call

- *Functions*, if they return a value, or
- *Procedures*, if they do not and thus are called for their side effects.

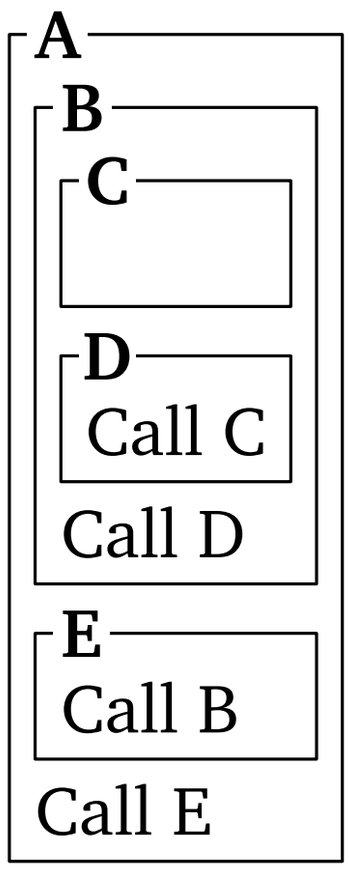
Subroutine parameters

- *Formal parameters* are the parameter names that appear in the subroutine declaration.
- *Actual parameters* or *arguments* are the values assigned to the formal parameters when the subroutine is called.

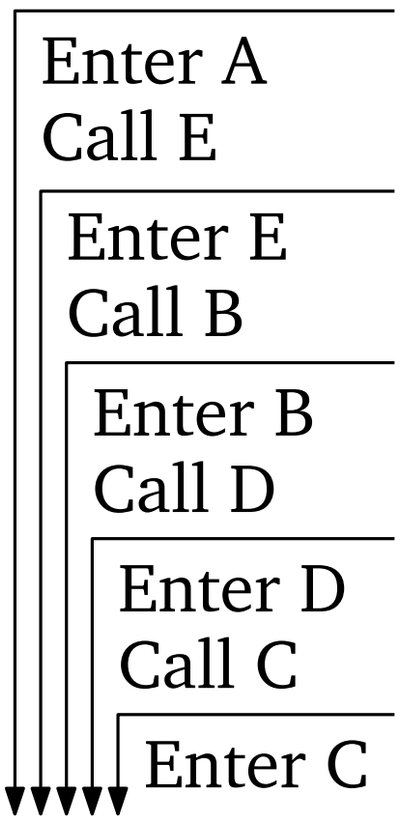
We already discussed stack-based allocation of *activation records* and their maintenance by caller and callee before, at the beginning, at the end, and after the subroutine call.

Static Chains and Dynamic Chains

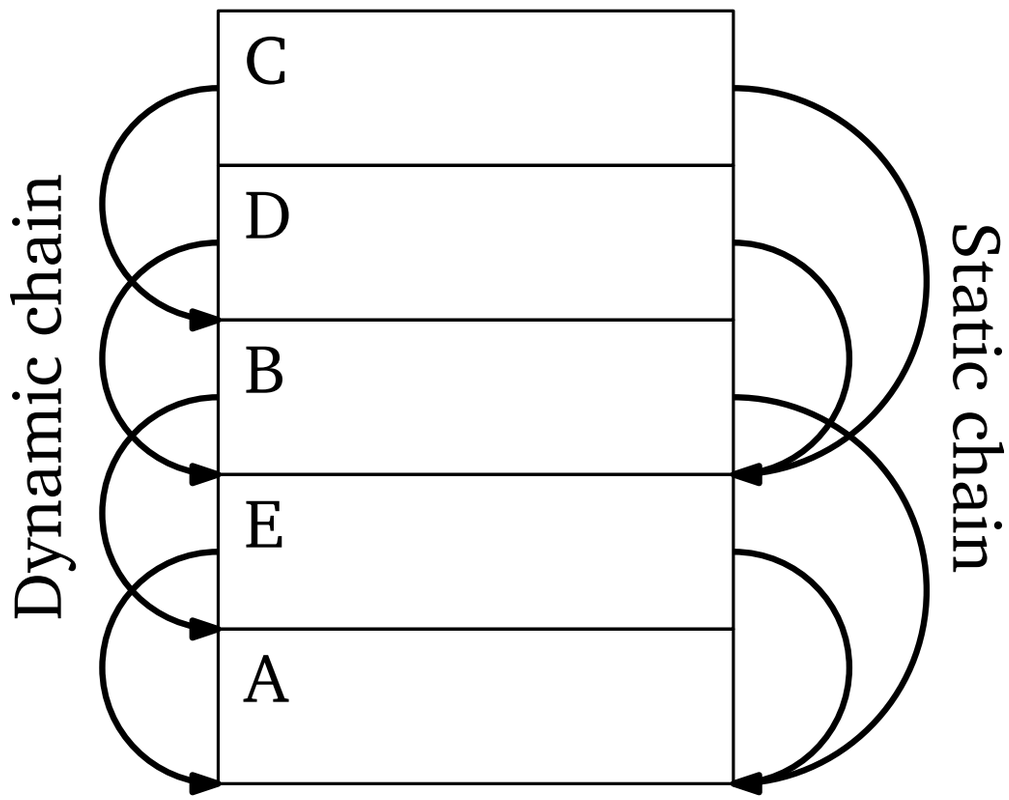
Source code



Program execution



Execution stack



Inline Expansion

Inline expansion replaces a subroutine call with the code of the subroutine.

Inline Expansion

Inline expansion replaces a subroutine call with the code of the subroutine.

Advantages

- Avoids overhead associated with subroutine calls ⇒ faster code.
- Encourages building abstractions in the form of many small subroutines.
- Related to but cleaner than macros.

Inline Expansion

Inline expansion replaces a subroutine call with the code of the subroutine.

Advantages

- Avoids overhead associated with subroutine calls ⇒ faster code.
- Encourages building abstractions in the form of many small subroutines.
- Related to but cleaner than macros.

Disadvantages

- Code bloating.
- Cannot be used for recursive subroutines.

Parameter Passing

Notation

`f(a, b, c)`	C, C++, Java, ...
`(f a b c)`	Lisp, Scheme
`a f: b fcont: c`	Smalltalk, Objective C
`a f b`	ML (infix notation)

Meaning

- Execute the named subroutine with its formal arguments bound to the provided actual arguments. How exactly?

Parameter Passing

Notation

`f(a, b, c)`	C, C++, Java, ...
`(f a b c)`	Lisp, Scheme
`a f: b fcont: c`	Smalltalk, Objective C
`a f b`	ML (infix notation)

Meaning

- Execute the named subroutine with its formal arguments bound to the provided actual arguments. How exactly?

Parameter passing modes

- By value
- By reference, by sharing
- By value/return

Semantics driven by implementation, not the other way around.

Parameter Passing Modes

Call by value

- A copy of the argument's value is passed
- Changes to the formal parameter do not affect the actual parameter

Call by reference

- The address of the argument is passed
- Formal parameter is an alias of actual parameter
- Changes to the formal parameter affect the actual parameter
- Actual parameter must be an ***l-value***

Similar or less common modes

- Call by value/return
- Call by sharing

Examples of Parameter Passing Modes (1)

FORTRAN

- All parameters are passed by reference
- Temporary variables are used to pass non-l-value expressions

Examples of Parameter Passing Modes (1)

FORTRAN

- All parameters are passed by reference
- Temporary variables are used to pass non-l-value expressions

Pascal

- Call by value is the default
- Keyword `var` before formal parameter switches to call by reference

  Example: `procedure sub(a : integer; var b : integer)`

Examples of Parameter Passing Modes (1)

FORTRAN

- All parameters are passed by reference
- Temporary variables are used to pass non-l-value expressions

Pascal

- Call by value is the default
- Keyword `var` before formal parameter switches to call by reference

  Example: `procedure sub(a : integer; var b : integer)`

C

- Call by value
- Arrays are passed by value, as pointers
- To simulate call by reference, pass a pointer

Examples of Parameter Passing Modes (2)

Smalltalk, Lisp, Clu, ML

- Reference model of variables ⇒ call by sharing
- Difference to call by reference: immutable objects may be passed by value

Examples of Parameter Passing Modes (2)

Smalltalk, Lisp, Clu, ML

- Reference model of variables ⇒ call by sharing
- Difference to call by reference: immutable objects may be passed by value

Ada

- `in` parameters: call by value
- `in out` parameters: call by reference
- `out` parameters: “call by result”

Examples of Parameter Passing Modes (2)

Smalltalk, Lisp, Clu, ML

- Reference model of variables ⇒ call by sharing
- Difference to call by reference: immutable objects may be passed by value

Ada

- `in` parameters: call by value
- `in out` parameters: call by reference
- `out` parameters: “call by result”

C++

- Same as C but with the addition of reference parameters:

```
void swap(int &a, int &b) { int t = a; a = b; b = t; }
```

- References can be declared `const`: efficiency of call by reference and safety of call by value

Examples of Parameter Passing Modes (3)

Java

- Call by value for primitive types
- Call by reference for compound types (objects)

Examples of Parameter Passing Modes (3)

Java

- Call by value for primitive types
- Call by reference for compound types (objects)

C#

- Call by value is the default
- `ref` and `out` keywords to force call by reference
- Distinction between call by value and call by reference made at data type level:
  - `struct` types are values.
  - `class` types are references.

Read-Only Parameters

A common practice in Pascal

- Large values are passed by reference for efficiency reasons
- High potential for bugs

Read-only parameters address this problem:

- Efficiency of call by reference
- Safety of call by value

Modula 3: `readonly` parameters

ANSI C, C++: `const` parameters

When using call by value, declaring a parameter `readonly` or `const` is pointless.

Some Uses of CONST in C++

Constant definition

```
const int buffersize = 512;
```

Read-only function parameter

```
void f(const int &i) { ... }
```

Immutable reference returned by a function (e.g., container interfaces)

```
const string &f() { ... }
```

Object method that cannot change the object (the only type of method that can be invoked on a `const` object)

```
int A::f(int i, string s) const { ... }
```

Subroutine Closures as Parameters

Functions as parameters and function return values require the passing of closures.

Languages that support this:

- Pascal
- Ada 95 (not Ada 83)
- All functional programming languages

Restricted passing of functions in C/C++ and FORTRAN

- Functions not allowed to nest (or not significantly in FORTRAN)
- No need for closures
- Pointers to subroutines suffice

Default (Optional) Parameters

Default (optional) parameters need not be specified by the caller.

If not specified, they take default values.

Ada

```
procedure put(item  : in integer;
              width : in field := 10 );
```

C++

```
void put(int item, int width = 10) { ... }
```

Implementation is trivial. How?

Named (Keyword) Parameters

Named (keyword) parameters need not appear in a fixed order.

- Good for documenting the purpose of parameters in a call.
- Necessary to utilize the full power of default parameters.

Ada

```
format_page(columns => 2,
            width   => 400,
            font    => Helvetica);
```

Implementation is once again trivial. How?

Variable Number of Arguments

C/C++ allow variable numbers of arguments:

```
#include <stdarg.h>

int printf1(char *format, ...)
{
  va_list args;
  va_start(args, format);
  char c = va_arg(args, char);
  ...
  va_end(args);
}
```

Java and C# provide similar facilities, in a typesafe but more restrictive manner.

Generic Subroutines and Modules (1)

Standard subroutines allow the same code to be applied to ***many different values***.

Generic subroutines can be applied to ***many different types***.

There are trade-offs involved in balancing the generality of the framework with type safety.

Generic Subroutines and Modules (1)

Standard subroutines allow the same code to be applied to ***many different values***.

Generic subroutines can be applied to ***many different types***.

There are trade-offs involved in balancing the generality of the framework with type safety.

Runtime type checks (Lisp)

```
(defun merge (a b)
  (cond ((null? a)
         b)
        ((null? b)
         a)
        ((< (car a) (car b))
         (cons (car a) (merge (cdr a) b)))
        (t
         (cons (car b) (merge a (cdr b))))))
```

Generic Subroutines and Modules (2)

Compile-time type checks upon instantiation (C++ templates)

```
class A {
  ...
  int f();
  ...
}

class B {
  // No method f
}

template <class T> class C {
  T data;
  ...
  int g() { return data.f(); }
}

C<A> a;  // OK
C<B> b;  // Error
```

Generic Subroutines and Modules (3)

Compile-time type checks upon declaration (Java)

```
public static <T extends Comparable<T>> void sort(T A[]) {
   ...
   if(A[i].compareTo(A[j]) >= 0) ...
   ...
}
...
Integer[] myArray = new Integer[50];
...
sort(myArray);
```

Haskell's concept of type classes is very similar to Java interfaces.

Exception Handling

An ***exception*** is an unexpected or abnormal condition arising during program execution.

They may be generated automatically in response to runtime errors or raised explicity in the program.

Typical semantics of exception handling
(Some (older) languages deviate from this)

- ***Exception handler*** lexically bound to a block of code.
- An exception raised in the block replaces the remaining code in the block with the code of the corresponding exception handler.
- If there is no matching handler, the subroutine exits and a handler is looked for in the calling subroutine.

Use of Exception Handlers

- Perform operations necessary to recover from the exception.
- Terminate the program gracefully, with a meaningful error message.
- Clean up resources allocated in the protected block before re-raising the exception.

Exception Support in Programming Languages

Representing exceptions

- Built-in exception type
- Object derived from an exception class
- Any kind of data can be raised as an exception

Exception Support in Programming Languages

Representing exceptions

- Built-in exception type
- Object derived from an exception class
- Any kind of data can be raised as an exception

Raising exceptions

- Automatically by the run-time system as a result of an abnormal condition (e.g., division by zero)
- `throw`/`raise` statement to raise exceptions manually

Exception Support in Programming Languages

Representing exceptions

- Built-in exception type
- Object derived from an exception class
- Any kind of data can be raised as an exception

Raising exceptions

- Automatically by the run-time system as a result of an abnormal condition (e.g., division by zero)
- `throw`/`raise` statement to raise exceptions manually

Where can exceptions be handled?

- Most languages allow exceptions to be handled locally and propagate unhandled exceptions up the dynamic chain.
- Clu does not allow exceptions to be handled locally. (How can you simulate local exception handlers?)
- PL/I's exception handling mechanism is similar to dynamic scoping.

Exception Support in Programming Languages

Representing exceptions

- Built-in exception type
- Object derived from an exception class
- Any kind of data can be raised as an exception

Raising exceptions

- Automatically by the run-time system as a result of an abnormal condition (e.g., division by zero)
- `throw`/`raise` statement to raise exceptions manually

Where can exceptions be handled?

- Most languages allow exceptions to be handled locally and propagate unhandled exceptions up the dynamic chain.
- Clu does not allow exceptions to be handled locally. (How can you simulate local exception handlers?)
- PL/I's exception handling mechanism is similar to dynamic scoping.

Some languages require exceptions thrown but not handled inside a subroutine to be declared as part of the subroutine definition.

Handling Exceptions Without Language Support

- "Invent" a value that can be used instead of a real value normally returned by the subroutine.
- Return an explicit "status" value to the caller. The caller needs to check this status.
- Rely on the caller to pass a closure to be called in case of an exception.

Exception Propagation

Exception handlers in the current scope are examined in order. The first one "matching" the exception is invoked.

If no matching handler is found, the subroutine exits, and the process is repeated in the caller.

Stack must be unwound (restored to the previous state) and any necessary clean-up needs to be performed (e.g., deallocation of heap objects, closing of file descriptors). Some languages provide support for this using constructs such as Java's `finally` clause.

Implementing Exception Handling (1)

A simple implementation

- Every subroutine/protected code block pushes its exception handler onto a handler stack.
- Exception handlers with multiple alternatives are implemented using if/then/else or switch statements in the handler.
- Every subroutine pushes a special exception handler onto the stack that is executed when control escapes the subroutine and performs all necessary clean-up operations.

Implementing Exception Handling (1)

A simple implementation

- Every subroutine/protected code block pushes its exception handler onto a handler stack.
- Exception handlers with multiple alternatives are implemented using if/then/else or switch statements in the handler.
- Every subroutine pushes a special exception handler onto the stack that is executed when control escapes the subroutine and performs all necessary clean-up operations.

This implementation is costly because it requires the manipulation of the handler stack for each subroutine call/return.

Implementing Exception Handling (2)

A faster implementation

- Store a global table mapping the memory addresses of code blocks to exception handlers (can be generated by compiler).
- When encountering an exception, perform binary search on the table using the program counter to locate the corresponding handler.

Implementing Exception Handling (2)

A faster implementation

- Store a global table mapping the memory addresses of code blocks to exception handlers (can be generated by compiler).
- When encountering an exception, perform binary search on the table using the program counter to locate the corresponding handler.

Comparison to simple mechanism

- Handling an exception is more costly (binary search), but exceptions are expected to be rare.
- In the absence of exceptions, the cost of this mechanism is zero!
- Cannot be used if the program consists of separately compiled units and the linker is not aware of this exception handling mechanism.

Implementing Exception Handling (2)

A faster implementation

- Store a global table mapping the memory addresses of code blocks to exception handlers (can be generated by compiler).
- When encountering an exception, perform binary search on the table using the program counter to locate the corresponding handler.

Comparison to simple mechanism

- Handling an exception is more costly (binary search), but exceptions are expected to be rare.
- In the absence of exceptions, the cost of this mechanism is zero!
- Cannot be used if the program consists of separately compiled units and the linker is not aware of this exception handling mechanism.

A hybrid approach

- Store a pointer to the appropriate table in each subroutine's stack frame

Exceptions in Java and C++

Java

- `throw` throws an exception.
- `try` encloses protected block.
- `catch` defines exception handler.
- `finally` defines block of clean-up code to be executed no matter what.
- Only `Throwable` objects can be thrown.
- Checked exceptions a function does not catch need to be declared.

```
try {
   ...
   throw ...
   ...
}
catch(SomeException e1) {
   ...
}
catch(SomeException e2) {
   ...
}
finally {
   ...
}
```

Exceptions in Java and C++

Java

- `throw` throws an exception.
- `try` encloses protected block.
- `catch` defines exception handler.
- `finally` defines block of clean-up code to be executed no matter what.
- Only `Throwable` objects can be thrown.
- Checked exceptions a function does not catch need to be declared.

```
try {
   ...
   throw ...
   ...
}
catch(SomeException e1) {
   ...
}
catch(SomeException e2) {
   ...
}
finally {
   ...
}
```

C++

- `throw`, `try`, and `catch` as in Java
- No `finally` block
- Any object can be thrown.
- Exception declarations on functions not required

Continuations in Scheme

Scheme does not support exceptions. However, it has a much more general construct that subsumes subroutines, coroutines, exceptions, . . . : ***continuations***.

A continuation is the "future" of the current computation, represented as the current stack content, referencing environment, register content, program counter, . . .

Continuations are first-class objects in Scheme: they can be passed as function arguments, returned as function results, and stored in data structures.

Call-With-Current-Continuation in Scheme

`(call-with-current-continuation f)` calls function `f` and passes the current continuation to `f` as an argument.

Simplest possible use: Escape procedure

- If `f` never makes use of the continuation it was passed as an argument, then everything works as if `f` had been invoked as `(f)`.
- If `f` invokes the continuation, then the program state is restored as if `f` had never been called.

Example: Look for the first negative number in a list

```
(call/cc (lambda (exit)
  (for-each (lambda (x)
    (if (negative? x)
      (exit x)))
    '(54 0 37 -3 245 19))
  #t))
```

$\Longrightarrow -3$

Call/CC for Exception Handling

```
(define list-length
  (lambda (obj)
    (call/cc (lambda (return)
      (letrec ((r (lambda (obj)
                    (cond ((null? obj) 0)
                          ((pair? obj) (+ 1 (r (cdr obj))))
                          (else        (return #f))))))
        (r obj))))))
```

```
(list-length '(1 2 3 4)) ⟹ 4
(list-length '(a b . c)) ⟹ #f
```

Setjmp/Longjmp Mechanism in C

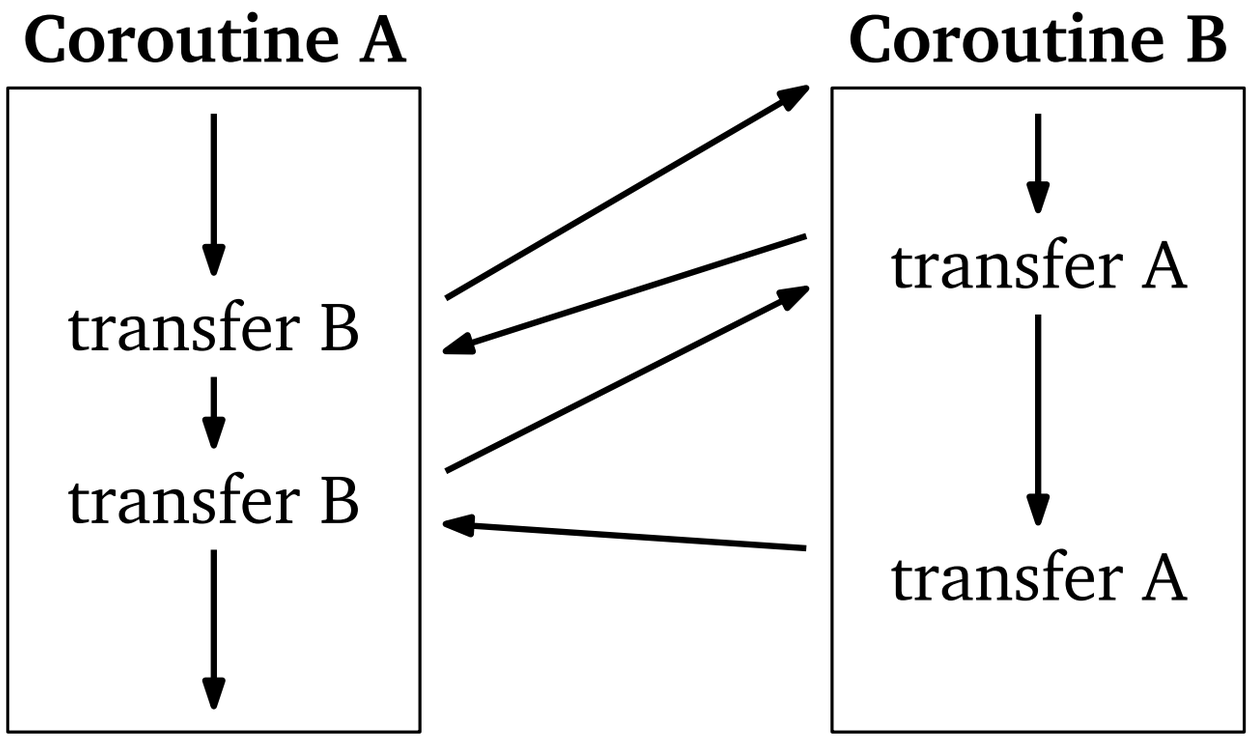
In C, `setjmp`/`longjmp` provide a limited form of continuations.

```
...
if(!setjmp(buffer)) {
  /* protected code */
} else {
  /* handler */
}
```

- First invocation of `setjmp` returns 0 and stores the current context (registers, stack pointer, ...) in the provided jump buffer.
- If no `longjmp` is performed on the buffer, the then-branch terminates as usual.
- If `longjmp` is invoked in the protected code, the `setjmp` returns for a second time, with a non-zero return value, and the handler in the else-branch is executed.

Coroutines

Coroutines are separate threads of execution that ***voluntarily*** transfer control to each other. (Contrast this with threads.)



Useful to implement iterators. (We have seen this already when talking about iteration.)

Managing Stack Space for Coroutines

Coroutines are “active” at the same time. Thus, they cannot use the same stack.

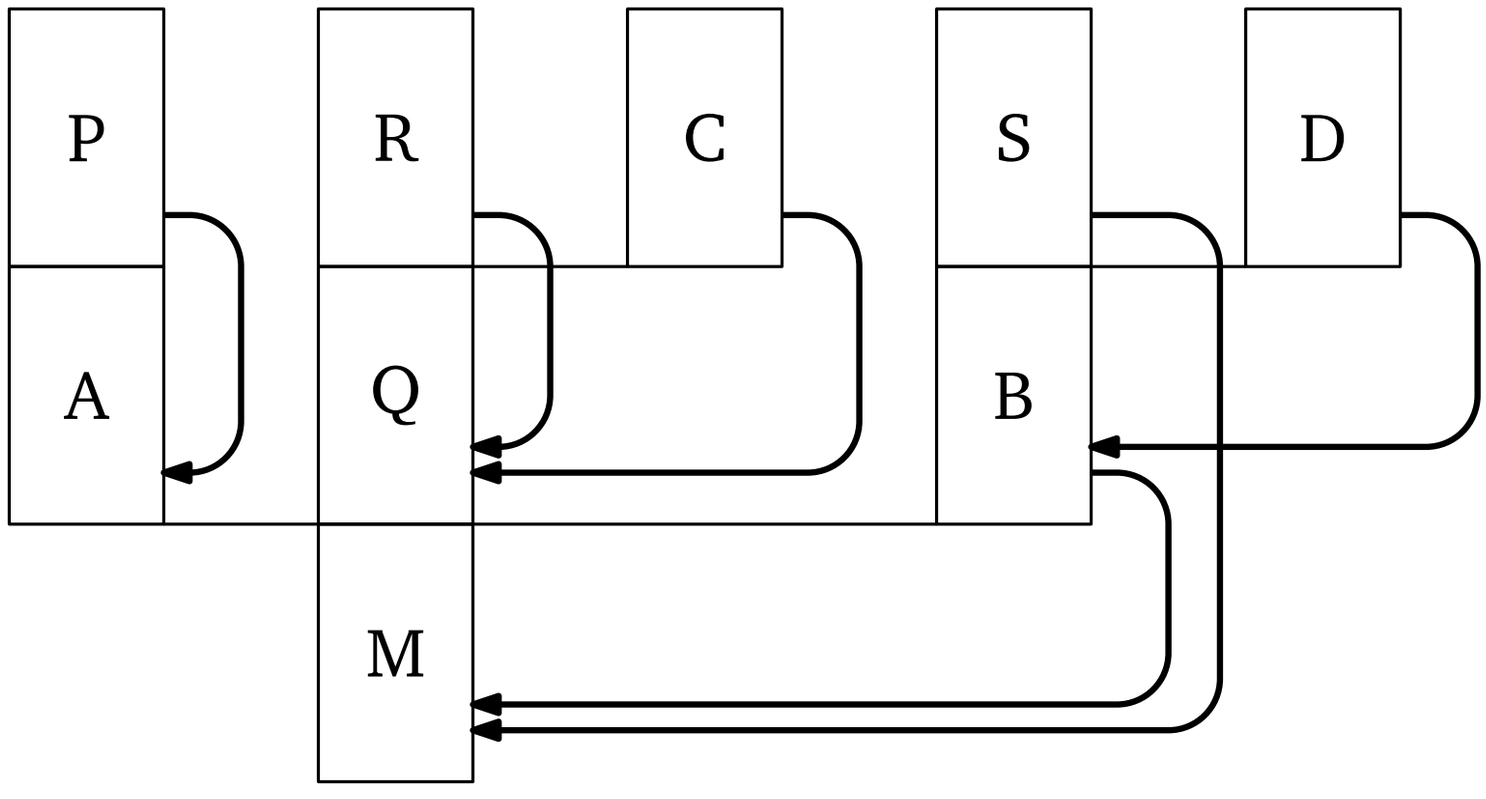
Some notion of stack is required to allow recursion within coroutines and support lexical scoping.

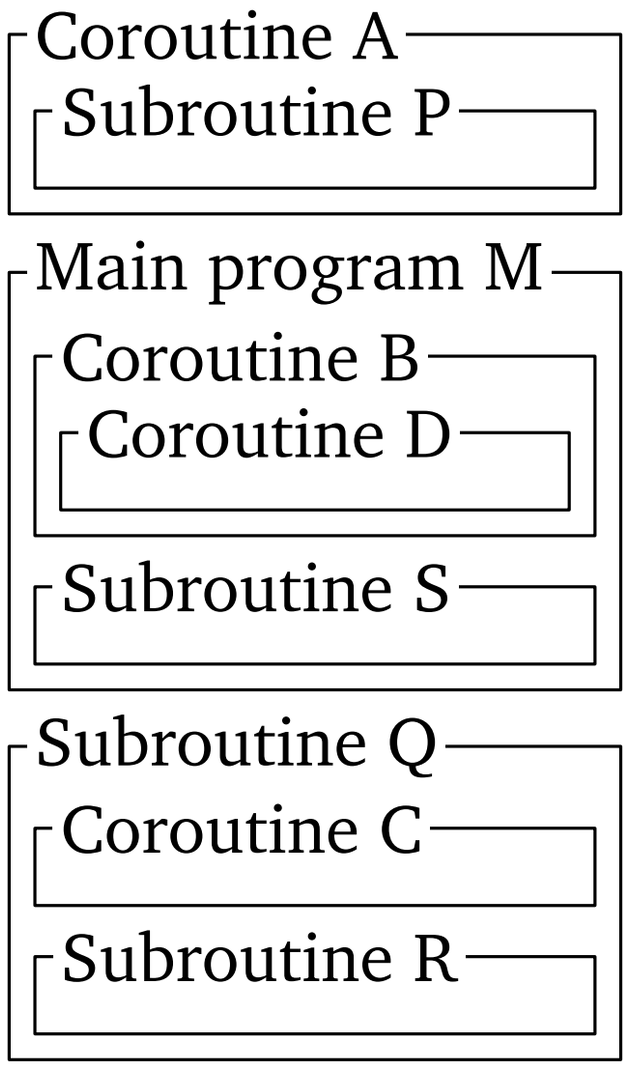
Managing Stack Space for Coroutines

Coroutines are “active” at the same time. Thus, they cannot use the same stack.

Some notion of stack is required to allow recursion within coroutines and support lexical scoping.

Solution: Cactus stack





Coroutines Using Continuations

```
(define (adder args)
  (let* ((val   (car args))
         (other (cdr args))
         (res   (call/cc
                  (lambda (c)
                    (other (cons val c))))))
    (if (< (car res) 100)
        (begin
          (display "Adder: ")
          (display (car res))
          (newline)
          (adder (cons (+ 1 (car res))
                       (cdr res)))))))
```

```
(define (multiplier args)
  (let* ((val   (car args))
         (other (cdr args))
         (res (call/cc
                (lambda (c)
                  (other (cons val c))))))
    (if (< (car res) 100)
        (begin
          (display "Multiplier: ")
          (display (car res))
          (newline)
          (multiplier (cons (* 2 (car res))
                            (cdr res)))))))
```

```
> (adder (cons 1 multiplier))
Adder: 1
Multiplier: 2
Adder: 4
Multiplier: 5
Adder: 10
Multiplier: 11
Adder: 22
Multiplier: 23
Adder: 46
Multiplier: 47
Adder: 94
Multiplier: 95
```