SUBROUTINES AND CONTROL ABSTRACTION

PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2018

Dalhousie University

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).

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

Subroutine

- Function if it returns a value
- Procedure if it does not and thus is called for its side effects

BASIC DEFINITIONS

Subroutine

- Function if it returns a value
- Procedure if it does not and thus is called for its side effects

Formal parameters of a subroutine

The parameter names that appear in the subroutine declaration

BASIC DEFINITIONS

Subroutine

- **Function** if it returns a value
- **Procedure** if it does not and thus is called for its side effects

Formal parameters of a subroutine

The parameter names that appear in the subroutine declaration

Actual parameters or arguments of a subroutine

The values bound to the formal parameters when the subroutine is called

Subroutine

- Function if it returns a value
- Procedure if it does not and thus is called for its side effects

Formal parameters of a subroutine

The parameter names that appear in the subroutine declaration

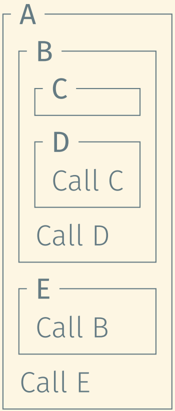
Actual parameters or arguments of a subroutine

The values bound to the formal parameters when the subroutine is called

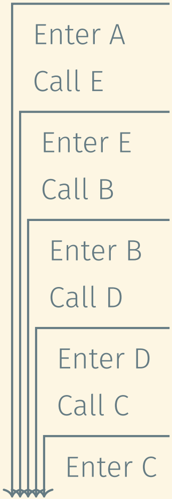
We already discussed activation records or (stack) frames as a means to manage the space for local variables allocated to each subroutine call.

STATIC CHAINS AND DYNAMIC CHAINS

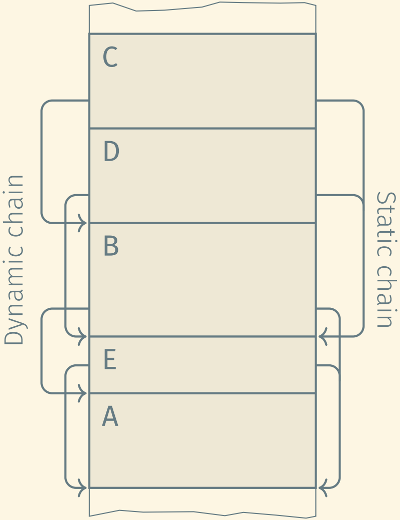
Source code



Program execution



Execution stack



ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

INLINE EXPANSION

Inline expansion

During compile time, the compiler replaces a subroutine call with the code of the subroutine.

INLINE EXPANSION

Inline expansion

During compile time, the compiler 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

During compile time, the compiler 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.
- Code profiling becomes more difficult.

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

PARAMETER PASSING

Notation:

`f(a, b, c)`	C, C++, Java, ...
`(f a b c)`	Lisp, Scheme
`a f: b fcont: c`	Smalltalk, Objective C
`f a b c`	Haskell, shell scripts

PARAMETER PASSING

Notation:

`f(a, b, c)`	C, C++, Java, …
`(f a b c)`	Lisp, Scheme
`a f: b fcont: c`	Smalltalk, Objective C
`f a b c`	Haskell, shell scripts

Meaning:

Execute the named subroutine with its formal arguments bound to the provided actual arguments.

Notation:

`f(a, b, c)`	C, C++, Java, ...
`(f a b c)`	Lisp, Scheme
`a f: b fcont: c`	Smalltalk, Objective C
`f a b c`	Haskell, shell scripts

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
`f a b c`	Haskell, shell scripts

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

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 the actual parameter.
- Changes to the formal parameter affect the actual parameter.
- The actual parameter must be an l-value.

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 poisters
- To simulate call by reference, pass a pointer

EXAMPLES OF PARAMETER PASSING MODES (2)

Smalltalk, Lisp, Clu, ML:

- Reference model of variables

$\Rightarrow$ Call by sharing: Object can be altered, just as with call by reference but the identity of the object cannot change.

EXAMPLES OF PARAMETER PASSING MODES (2)

Smalltalk, Lisp, Clu, ML:

- Reference model of variables

⇒ Call by sharing: Object can be altered, just as with call by reference but the identity of the object cannot change.

Ada:

- `in` parameters: Call by value
- `in out` parameters: Call by reference or call by value/return
- `out` parameters: "Call by result"

EXAMPLES OF PARAMETER PASSING MODES (2)

Smalltalk, Lisp, Clu, ML:

- Reference model of variables

⇒ Call by sharing: Object can be altered, just as with call by reference but the identity of the object cannot change.

Ada:

- `in` parameters: Call by value
- `in out` parameters: Call by reference or call by value/return
- `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, Python:

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

EXAMPLES OF PARAMETER PASSING MODES (3)

Java, Python:

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

C#:

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

A common practice in Pascal:

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

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

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

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.

Constant definition:

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

SOME USES OF CONST IN C++

Constant definition:

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

Read-only function parameter:

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

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() { ... }
```

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 { ... }
```

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

SUBROUTINE CLOSURES AS PARAMETERS

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

SUBROUTINE CLOSURES AS PARAMETERS

Function 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

SUBROUTINE CLOSURES AS PARAMETERS

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

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

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 : int field := 10);
```

C++:

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

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 : int field := 10);
```

C++:

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

Implementation is trivial. How?

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   => 4,
            font    => Helvetica);
```

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   => 4,
            font    => Helvetica);
```

Implementation is once again trivial. How?

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

VARIADIC SUBROUTINES (VARIABLE NUMBER OF ARGUMENTS)

C/C++/Python 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);
}
```

VARIADIC SUBROUTINES (VARIABLE NUMBER OF ARGUMENTS)

C/C++/Python 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.

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

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

Generic subroutines can be applied to many different types.

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

Generic subroutines can be applied to many different types.

There is a trade-off involved in balancing the generality of the framework with type safety.

Examples: Lisp, Scheme, Python, Ruby

```
(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: COMPILE-TIME TYPE CHECKS UPON INSTANTIATION

Example: 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
```

Examples:

- Java interfaces
- Haskell type classes

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

Integer[] myArray = new Integer[50];
sort(myArray);
```

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

Exception

Unexpected or abnormal condition arising during program execution

Exception

Unexpected or abnormal condition arising during program execution

Exceptions may be generated automatically in response to runtime errors or raised explicitly in the program.

EXCEPTION HANDLING

Exception

Unexpected or abnormal condition arising during program execution

Exceptions may be generated automatically in response to runtime errors or raised explicitly in the program.

Typical semantics of exception handling

- 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 anda handler is looked for in the calling subroutine.

EXCEPTION HANDLING

Exception

Unexpected or abnormal condition arising during program execution

Exceptions may be generated automatically in response to runtime errors or raised explicitly in the program.

Typical semantics of exception handling

- 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 anda handler is looked for in the calling subroutine.

Some (older) languages deviate from this.

- 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 (1)

Representing exceptions:

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

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 (2)

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 (2)

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 a 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 PROPAGATION

- Exception handlers in the current scope are examined in order. The first one that “matches” the exception is invoked.

EXCEPTION PROPAGATION

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

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

EXCEPTION PROPAGATION

- Exception handlers in the current scope are examined in order. The first one that "matches" the exception is invoked.
- If no matching handler is found, the subroutine exits, and the process is repeated in the caller.
- The 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.

A simple implementation:

IMPLEMENTING EXCEPTION HANDLING (1)

A simple implementation:

- 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 pushes a special exception handler onto the stack that is executed when control escapes the subroutine and performs all necessary clean-up operations.
- Every subroutine/protected code block pushes its exception handler onto a handler stack.

IMPLEMENTING EXCEPTION HANDLING (1)

A simple implementation:

- 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.
- 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.

IMPLEMENTING EXCEPTION HANDLING (1)

A simple implementation:

- 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.
- 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.

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

A faster implementation:

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).

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:

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 should be rare.

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 should be rare.
- In the absence of exceptions, the cost of this mechanism is zero!

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 should 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.

EXCEPTIONS IN JAVA AND C++

Java:

- `throw` throws an exception.
- `try` encloses a protected block.
- `catch` defines an exception handler.
- `finally` defines block of clean-up code to execute no matter what.
- Only `Throwable` objects can be thrown.
- Must declare uncaught checked exceptions.

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

EXCEPTIONS IN JAVA AND C++

Java:

- `throw` throws an exception.
- `try` encloses a protected block.
- `catch` defines an exception handler.
- `finally` defines block of clean-up code to execute no matter what.
- Only `Throwable` objects can be thrown.
- Must declare uncaught checked exceptions.

C++:

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

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

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

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

CONTINUATIONS IN SCHEME

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

A continuation is the “future” of the current computation, represented as

- Current stack content and referencing environment
- Current register content
- Current program counter
- ...

CONTINUATIONS IN SCHEME

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

A continuation is the "future" of the current computation, represented as

- Current stack content and referencing environment
- Current register content
- Current 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.

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 uses 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.

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 uses 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))
```

CALL/CC FOR EXCEPTION HANDLING

```
(define (list-length 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 */
}
```

- The 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, the `setjmp` returns for a second time, with a non-zero return value, and the handler in the else-branch is executed.

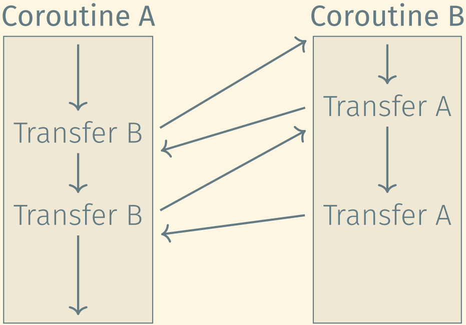
ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

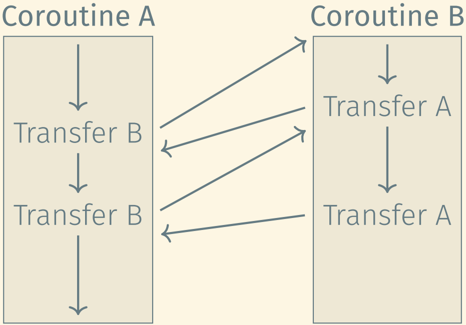
ROAD MAP

- Functions, procedures, and parameters
- Inline expansion
- Parameter passing modes
- Passing functions as arguments
- Default and named parameters
- Variadic subroutines
- Generic subroutines
- Exception handling
- Continuations
- Coroutines

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



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



Useful to implement generators, e.g., in Python

MANAGING STACK SPACE FOR COROUTINES

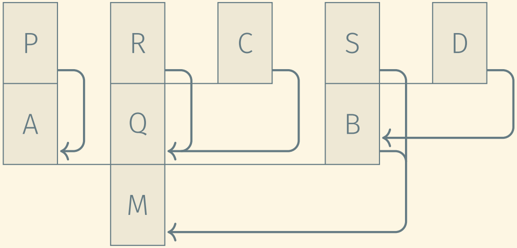
Coroutines are "active" at the same time. Thus, they cannot use the same stack.

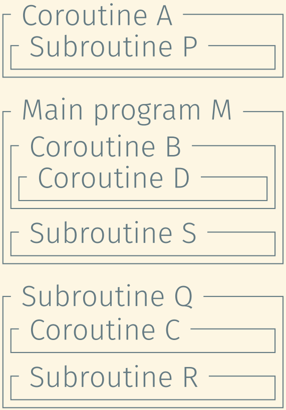
Some notion of stack is required to allow recursion within coroutines and support lexical scoping.

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





```
(define (adder args)
  (let* ((val   (car args))
         (other (cdr args))
         (res   (call/cc (lambda (c) (other (cons val c))))))
    (if (< (char 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
```

SUMMARY

- Subroutines are the main tool for building control abstractions.
- Parameter passing modes determine how subroutines interact with the outside world through their parameters.
- Exception handling is a mechanism to recover from abnormal situations in a program's execution.
- Exceptions should not be used for normal control flow! (Shame on you, Python!)
- Coroutines are elegant tools for implementing cooperative multi-threading.
- Continuations subsume subroutines, coroutines, exception handling, ...