Computation Abstractions and Exception Handling

CSCI 3136: Principles of Programming Languages

Agenda

- Announcements
  - Assignment 9 is out, due July 30
  - Final exam, 1:00pm, Friday, August 2 in CHEB 170
  - In-class SRIs on Wednesday
- Readings: Read Chapter 6.6, 9
- Lecture Contents
  - Finish previous lecture
  - Motivation
  - Parameters and Arguments
  - Applicative and Normal Order Evaluation
  - Introduction to Exceptions
  - Languages Support
  - Exception Propagation
  - Exception Implementation
  - Examples

How are the Student Ratings of Instruction (SRI) used?

- ✓ Course and program ***(re) design.***
- ✓ ***Evaluation*** of teaching effectiveness.
- ✓ ***Promotion and tenure applications*** for instructors, and other personnel decisions.
- ✓ Preparation of supporting evidence for ***teaching awards and grants***.
- ✓ ***Quality assurance*** processes in the review and restructure of institutional, faculty, department and program goals.

How to complete the SRI

- Find the email in your Dal email account
  - Subject heading (depending on the system) is:
    - *Student Ratings of Instruction; or*
    - *Course Name and Number*
- Open the email and click on the link
  - Your course list should be visible
- Select the course for which you want to complete the evaluation
- Be sure to hit the SUBMIT button when you FINISH completing the form
- You may also SAVE and return to your work later

Motivation

- We take functions, procedures, methods, and subroutines for granted
- We learn them in 1st year and then use them
- How we use them is dictated by
  - Scope (Static or Dynamic) ✅
  - Binding (Deep or Shallow) ✅
  - Parameters (what the functions expect)
  - Arguments (what is passed in)
  - Evaluation (what takes place inside)
  - Return (L-values, R-Value, Composite, etc)

A Function Is …

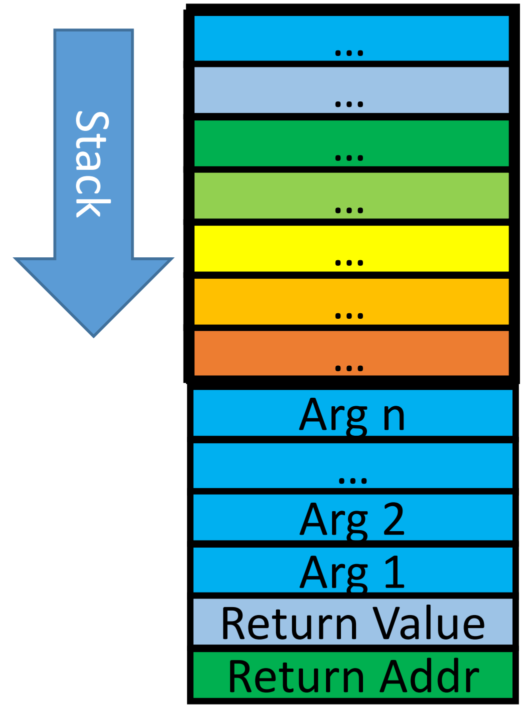
- Also known as a subroutine, procedure, method, etc.
- A piece of code that
  - Specifies what parameters it is expecting to be called with (0+)
  - Is called from somewhere in the program by the *caller* who provides arguments to be bound to parameters
  - When called, the *callee* performs a computation using the arguments provided by the caller
  - Optionally returns a value
  - Optionally generates side-effects
- Example:

```
def fact( n = 1 ):
  if n < 2:
    return 1
  else:
    return n * fact( n – 1 )

fact( 42 )
```

- What does this do? It returns a big number. ☺

Recall: Before the Call



Caller

- Pushes arguments on the stack
- Pushes a dummy return value (optional)
- Executes call instruction

  ```
  call foo
  ```

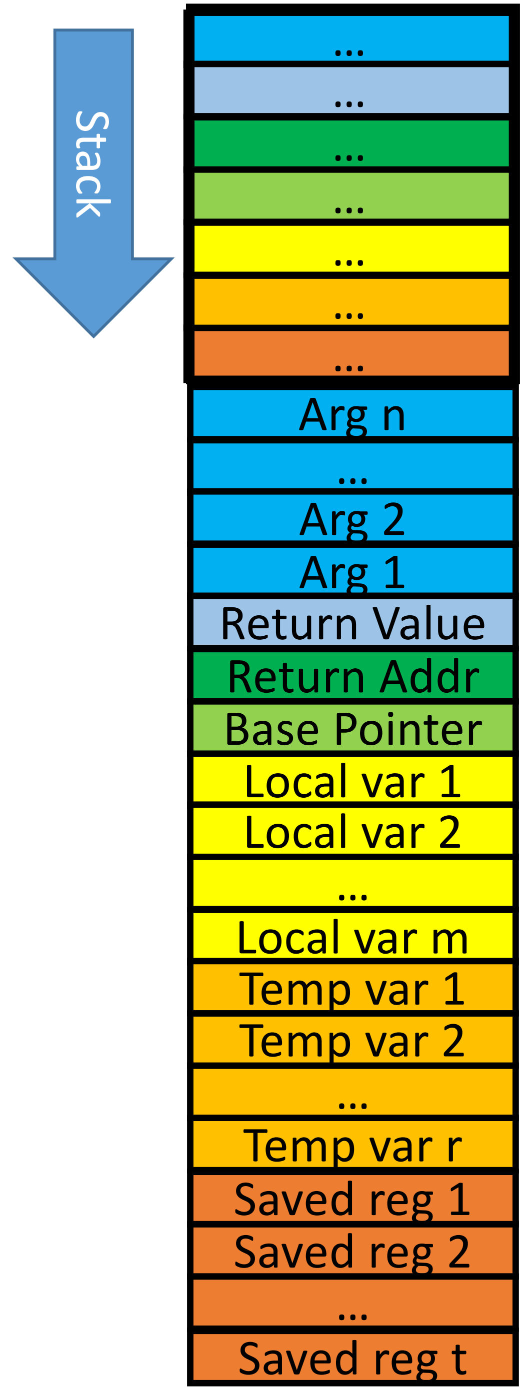
  - Pushes return address on stack
  - Jumps to subroutine (callee)

Recall: During the Call

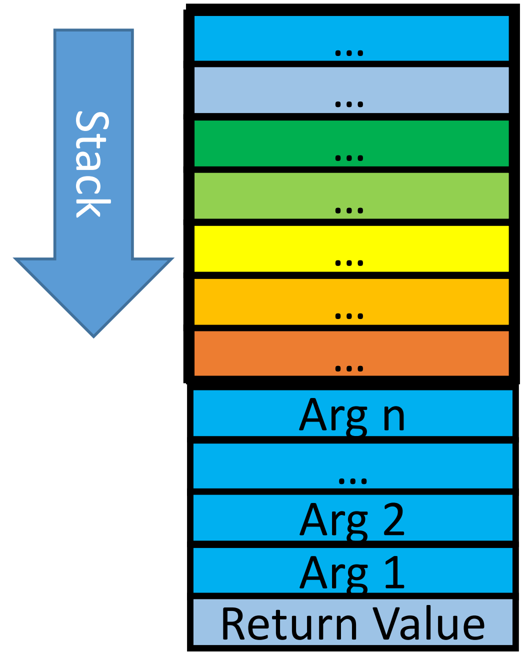
Callee

- Saves base pointer
- Allocates local variables
- Allocates temporary variables
- Saves registers
- Performs body of subroutine
- Restores registers
- Destroys local and temp variables
- Returns

  `ret`
  - Pops return address off stack
  - Jumps to return address



Recall: After the Call



Caller

- Pops arguments off the stack

Parameters and Arguments

- **Parameters** are the variables specified by the function declaration, which are visible during the call

  E.g. here is a function declaration in Swift

  ```
  func choose( m:Int, n:Int = 1 ) -> Int { …
  ```

  - Specifies two parameters: *m* and *n* of type Int
  - The second parameter has a default value

- **Arguments** are the values passed by the caller to the callee

  E.g. here is a function call in Swift

  ```
  marbles = choose( 42, 13 )
  ```

  - Passes two arguments to choose(): *42* and *13*
  - During the call, *m* is bound to *42* and *n* is bound to *13*

Parameter Modes

- Parameter modes is how the arguments are actually passed to the function
- The standard modes are
  - **Pass-by-value**: the r-value of the variable or expression is passed
    - Used by C
  - **Pass-by-reference**: the l-value of the variable or expression is passed
    - Used in Lisp, Smalltalk, Ruby, available in C++
- Other modes are:
  - **Pass-by-value/result:** pass by value, but copy the value from the parameter back to the variable that was passed
    - Used in Algol W, Ada (in out)
  - **Pass-by-sharing**: Similar to pass by reference. The object can be modified, but reference cannot
    - Used in Java for Composite types
  - **Read-only:** Are passed by reference but cannot be modified

Pass By Value

- The value of the expression is bound to the parameter (copied), and passed to the function
- Modifications to the value are not visible outside of the function

```
int increment( int val ) {
  val++;
  return val;
}

int i = 42;
int j = increment( i );
printf( “%d %d\n”, i, j );
```

- The output is: **`42 43`**

Pass By Reference

- The location (reference) of the value of the expression is bound to the parameter (copied), and passed to the function
- Modifications to the value are visible outside of the function

```
int increment( int& val ) {
  val++;
  return val;
}

int i = 42;
int j = increment( i );
printf( “%d %d\n”, i, j );
```

- The output is: **`43 43`**

Pass-by-Reference vs Pass-by-Value

- **Pass-by-value**
  - Easier to understand (fewer side-effects)
  - Efficient for primitive values (integer, floats, characters)
  - Changes are not propagated back from the call
  - Inefficient for large objects
- **Pass-by-reference**
  - Used to pass large or complex data structures
  - Changes to parameters are reflected in arguments
  - Greater care needs to be taken during recursion
- Question: When should arguments be evaluated?
  - At call?
  - At use?

Applicative and Normal Order of Evaluation

- Applicative-order evaluation
  - Arguments are evaluated before a subroutine call
  - Default in most programming languages
- Normal-order evaluation
  - Arguments are passed unevaluated to the subroutine
  - The subroutine evaluates them as needed
  - Useful for infinite or lazy data structures that are computed as needed
  - Examples: Macros in C, Haskell
  - Fine in functional languages
  - Problematic if there are side effects
    - What if argument is not always used?
  - Potential for inefficiency
    - What happens if argument is passed to other subroutines?

Return Values

- In most languages functions typically return r-values
  - A value that can be assigned to a variable or used in an expression
- Some languages, such as C++, allow functions to return l-values (locations of the value)
  - Seen in a previous lecture
- Return of l-values can be simulated in most languages
  - Using pointers in C
  - Returning references in Java
  - Etc.
- But … Sometimes it's hard to know what to return!

Exception Handling

- Things go wrong (bleep happens), we need to handle it gracefully
- *Exception* are unexpected or abnormal conditions during execution
  - Generated automatically in response to runtime errors
  - Raised explicitly by the program
- Exception handling is needed to
  - Perform operations necessary to recover from the exception
  - Terminate the program gracefully
  - Clean up resources allocated in the protected block
- Exception handling allows the programmer to
  - Specify what to do when an error occurs during program run-time
  - Separate the common path code from the error handling code

Exception Handling Syntax

- Syntax for catching and handling exceptions tends to be similar
- A **protected block** comprises 3 parts:
  - **try** : the common path code to be executed
  - **catch** : exception handlers for each exception to be caught
  - **finally** : an optional "clean-up" handler that always runs after the "try" regardless of whether an exception occurs
- Exception are **raised** (or thrown) by a raise (or throw) statement

```
raise Exception_1(…)
```

```
try {
  // common path
} catch ( Exception_1 e ) {
  // Exception 1 handler
} catch ( Exception_2 e ) {
  // Exception 2 handler
} ...
} else { // optional
  // default handler
} finally { // optional
  // clean up code
}
```

Exception Handling Semantics

- An exception handler is lexically bound to a block of code
- When an exception is raised in the block, search for a handler in present scope
- If there is no matching handler in present scope,
  - The scope is exited (may include block or subroutine)
  - A handler is searched for in the next scope

```
def main():
  try:
    parse()
  except MemoryError as p:
    print “Program too big!”

def parse():
  try:
    l = parseS()
    if lookahead() != None:
      raise ParseError( 'S' )
    eval_result( top_ref, l )
  except ParseError as p:
    print "Syntax Error”
  except EvalError as p:
    print "Evaluation Error”
```

Out of Memory

Language Support

- How are exceptions represented?
  - Built-in exception type (Python)
  - Object derived from an exception class (Java)
  - Any kind of data can be passed as part of an exception
- When are exceptions raised?
  - Automatically by the run-time system as a result of an abnormal condition

    e.g., division by zero, null dereference, out-of-bounds, etc
  - `throw` or `raise` statement to raise exceptions manually
- Where can exceptions be handled?
  - Most languages allow exceptions to be handled locally
  - Propagate unhandled exceptions up the dynamic chain.
    - Clu does not allow exceptions to be handled locally
- Some languages require exceptions that are thrown but not handled inside a subroutine be declared (Java)

Language Non-support

- Some languages do not support exceptions
  e.g., C
- Solution 1:
  - Reserve a return value to indicate an exception
- Solution 2:
  - Caller passes a closure (exception handler) to call
- Solution 3:
  - In C, signals and setjmp / longjmp can be used to simulate exceptions

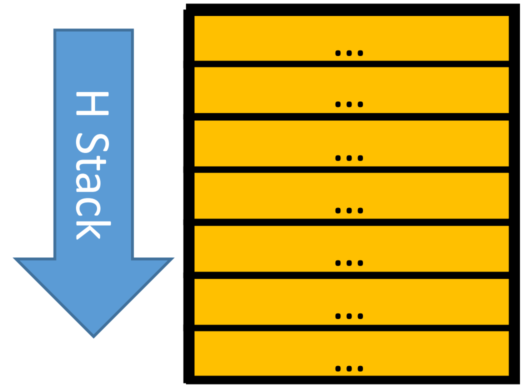
```
#include <setjmp.h>

int func(…) {
  static jmp_buf env;
  int i = setjmp(env);
  if( i == 0 ) {
    /* common path */
    ...
    /* exception 42 */
    longjmp(env, 42);
    ...
  } else if( i == 42 ) {
    /* handle exception 42 */
  } else if( i == ? ) {
    ...
```

Exception Implementations

- Options:
  - Simple (Pay as you go)
  - Location to Exception map (Pay on Exception)
  - Hybrid

Simple, Pay-as-You-Go Exception Handling



- Idea:
  - The program uses a second stack, called a Handler Stack (HS)
  - When a protected block is entered, a handler is pushed on the (HS)
    - Pointer to the handler code
    - Current stack frame (Program Stack)

      I.e., referencing environment

    Sound familiar?
    - An optional exit (finally) handler may also be pushed
  - If there are multiple exception handlers, these are implemented using an if/elseif/… construct in a single handler
  - When a protect block is exited, the handler is popped of the stack
- Simple implementation is costly because handler stack is manipulated on entry/exit of each protected block

Simple, Pay-as-You-Go Exception Handling

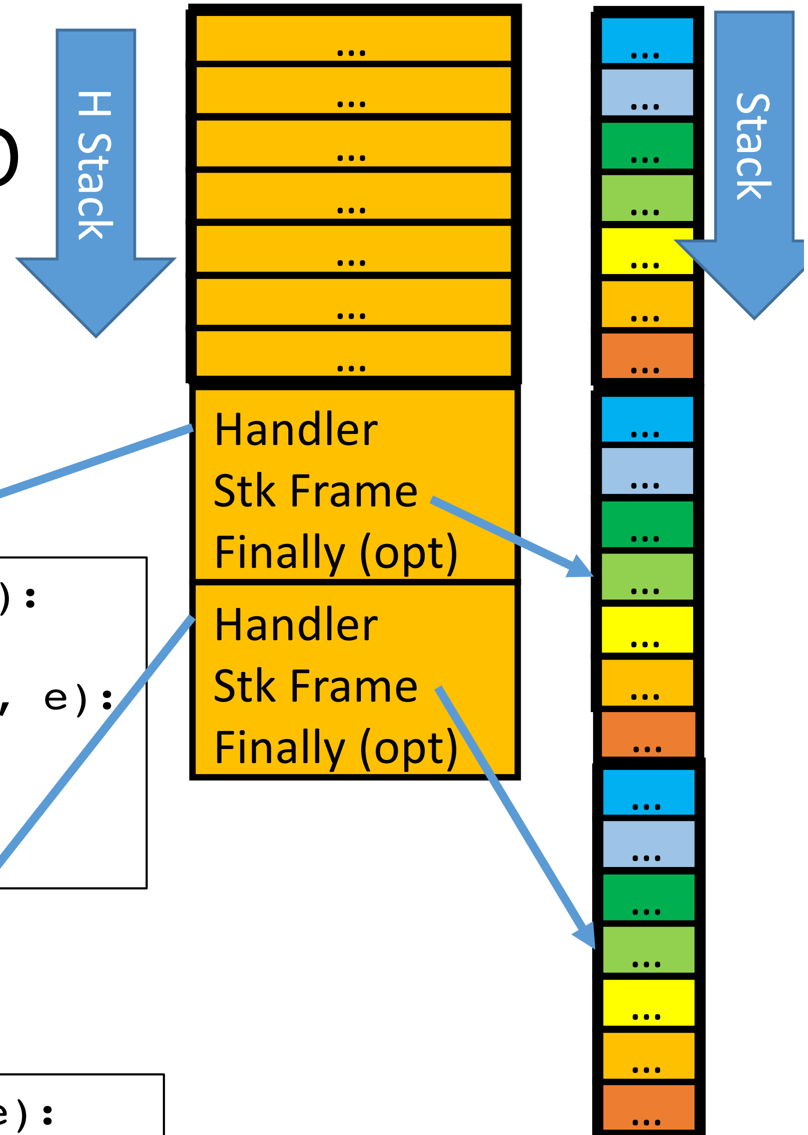
```
def foo():
  try:
    bar()
  except E1 as e:
    # E1 handler
  except E2 as e:
    # E2 handler
```

```
if isinstance(E1, e):
  # E1 handler
elsif isinstance(E2, e):
  # E2 handler
else:
  raise e
```

```
def bar():
  try:
    ...
  except E3 as e:
    # E3 handler
  except E4 as e:
    # E4 handler:
```

E1

```
if isinstance(E3, e):
  # E3 handler
elsif isinstance(E4, e):
  # E4 handler
else:
  raise e
```



Location to Exception Mapping

- A faster implementation (Pay on exception)
- Store a global map of code blocks (memory addresses) to handlers
    - Generated by compiler/linker
- On exception, index map with program counter to get handler
- Still need to keep track of stack frames
    - Each stack frame stores a pointer to most recent protected block

Pay on Exception Exception Handling

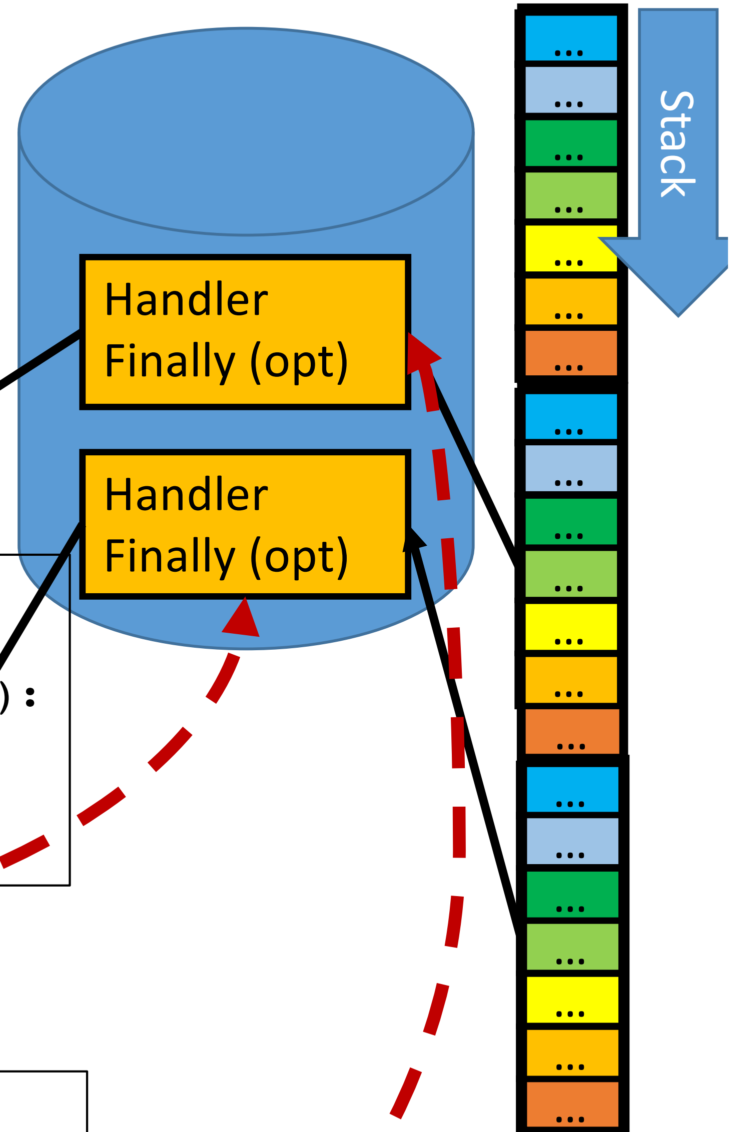
```
def foo():
  try:
    bar()
  except E1 as e:
    # E1 handler
  except E2 as e:
    # E2 handler
```

```
if isinstance(E1, e):
  # E1 handler
elsif isinstance(E2, e):
  # E2 handler
else:
  raise e
```

```
def bar():
  try:
    ...
  except E3 as e:
    # E3 handler
  except E4 as e:
    # E4 handler:
```

E1

```
if isinstance(E3, e):
  # E3 handler
elsif isinstance(E4, e):
  # E4 handler
else:
  raise e
```



Comparison of the 2 Approaches

- Location-based Exception handling
  - Handling an exception is more costly (search), but exceptions rare
  - No cost if no exceptions
  - Cannot be used if the program consists of separately compiled units and the linker is not aware of this exception handling mechanism
- Hybrid Approach:
  - Use a local map for each subroutine
  - Store a pointer to a local map in subroutines stack frame