EECS 390 – Lecture 7

Functions and Recursion

2/4/24

Agenda

- Parameter Passing
- Keyword, Default, and Variadic Arguments
- Recursion and Tail Recursion

Parameter Passing

- Arguments and parameters are a means of communication between a function and its caller
- A parameter may be used only for input, only for output, or for both
- Semantics of parameters determined by ***call mode*** of function
  - Call by value
  - Call by reference
  - Call by result
  - Call by value-result
  - Call by name

Call by Value

- A parameter represents a new variable in the frame of a function invocation
- Argument value is copied to parameter variable
- Parameter can only be used for input

```
void foo(int x) {
  x++;
  cout << x << endl;
}

int y = 3;
foo(y);                // prints 4
cout << y << endl;     // prints 3
```

Call by Reference

- Requires l-value as argument[1]
- Parameter name is bound to argument object
- Parameter can be used for input and output
- No separate storage for parameter

```
void swap(int &x, int &y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int x = 3, y = 4;
swap(x, y);         // x now 4, y now 3
```

[1]In C++, const l-value references can bind to r-values

Simulating Call by Reference

- Pointers can be used to simulate call by reference
- However, function is still call by value, since parameters correspond to new pointer variables

```
void swap(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int x = 3, y = 4;
swap(&x, &y);      // x now 4, y now 3
```

Call by Result

- Argument must be l-value
- Parameter is a new variable with its own storage
- Parameter is not initialized with argument value
- Upon return of the function, parameter value is copied to argument object
- Can only be used for output

```
void foo(result int x) {
  x = 3;
  ...
  x++;    // x is now 4
}

int y = 5;
foo(y);   // y is now 4
```

Call by Value-Result

- Combination of call by value and call by result
- Argument must be l-value
- Parameter is a new variable with storage, initialized with argument value
- Upon return, value of parameter is copied to argument object

```
int foo(v/r int x, v/r int y) {
  x++;
  return x - y;
}

int z = 3;
print(foo(z, z));    // prints 1
```

Again, not C++! Final value of z depends on whether it is copied from first or second parameter in the given language

Call by Name

- Any expression provided as argument
- Parameter name is replaced by argument expression everywhere in the body
- Expression computed when it is encountered in body
  - Often computed just the first time, then cached value used in subsequent references to the argument

```
void foo(name int a, name int b) {
  print(b);  // becomes print(++y)
  print(b);  // becomes print(++y)
}

int x = -1, y = 3;
foo(++x, ++y);  // prints 4, then 4 or 5 depending on
                // language semantics; y is now 4 or 5
print(x);  // prints -1 -- x is unchanged
```

!C++; Mutating expressions should not be passed by name, since behavior would depend on implementation details

Thunks

- In call by name, expression must be computed in its own environment

```
void bar(name int x) {
  int y = 3;
  print(x + y);  // becomes print(y + 1 + y)
}

int y = 1;
bar(y + 1);      // should print 5, not 7
```

- This is accomplished with a ***thunk***, a compiler-generated local function that packages the expression with its environment

Keyword Arguments

- In most languages, names are not specified for arguments when calling a function
  - Arguments are bound to parameters in order

```
void foo(int x, int y);
foo(3, 4);
```

- Some languages allow arguments to be passed to specific parameters, allowing them to be given in a different order and serving as documentation

```
def foo(x, y):
    print(x, y)

>>> foo(y = 3, x = 4)
4 3
```

Python also has positional-only (PEP 570) and keyword-only (PEP 3102) parameters.

Arguments in Swift

- Swift and Objective-C require argument names for most arguments, as well as that they are passed in the same order as the parameters

```
func greet(name: String, withGreeting: String) {
  print(withGreeting + " " + name)
}
greet(name: "world", withGreeting: "hello")
```

- Functions can specify separate internal and external names for a parameter

- Argument names used in function-overload resolution

```
func foo(a: Int) { ... }
func foo(b: Int) { ... }
foo(a: 3)
```

Default Arguments

- Some languages allow a function definition or declaration to provide a default argument for a parameter
- Allow a function to be called without an argument value for the parameter

```
void foo(int x, int y = 0);
foo(3);  // equivalent to foo(3, 0)
foo(3, 4);
```

- Parameters with default arguments generally have to be at the end of the parameter list
- Evaluation rules
  - Evaluated in definition environment in most languages
  - Most languages evaluate default argument each time the function is called

Python Default Arguments

- Python differs from most languages in that the default argument is evaluated only once at definition time

```
def foo(x, y=[]):
    y.append(x)
    print(y)

>>> foo(3)
[3]
>>> foo(4)
[3, 4]
```

Overloading as Alternative

- Some languages, such as Java, rely on function overloading to provide the same behavior as default arguments

```
static void foo(int x, int y) {
  System.out.println(x + y);
}

static void foo(int x) {
  foo(x, 0);
}
```

“Default” argument of `0`

Variadic Functions

- Functions that can be called with a variable number of arguments, also referred to as ***varargs***
- Arguments often packed into a container such as a tuple or array
- Arguments may be required to be of the same type, or can be of different types
- Example in Java:

```
static void print_all(String... args) {
  for (String s : args) {
    System.out.println(s);
  }
}

print_all("hello", "world");
```

All Strings, packaged into array

Java also allows an array to be passed into a variadic parameter.

Iteration and Recursion

- Iteration and recursion are just different tools for implementing the same algorithms
  - Iteration is actually equivalent to ***tail recursion***, which is recursion that does no work after the recursive call
- Example of transformation from iteration to recursion:

```
def fib(n):
    if n == 0: return 0
    prev, crnt = 0, 1
    for i in range(1, n):
        prev, crnt = crnt, prev + crnt
    return crnt
```

Local variables become parameters

```
def fib(n, prev=0, crnt=1, i=1):
    if n == 0: return 0
    if i == n: return crnt
    return fib(n, crnt, prev + crnt, i + 1)
```

Same termination condition

Iteration and Recursion

- Non-tail recursion may require an explicit data structure when converted to iteration
  - Takes the place of the implicit recursive call stack

```
def deep_sum(tree):
    if not tree: return 0
    return (tree.datum + deep_sum(tree.left)
               + deep_sum(tree.right))
```

Explicit stack

"Recursive case"

```
def deep_sum(tree):
    sum = 0
    stack = [tree]
    while stack:
        if tree := stack.pop():
            sum += tree.datum
            stack.append(tree.left)
            stack.append(tree.right)
    return sum
```

Activation Records and Recursion

- Recursion works on a machine since every function invocation gets its own activation record

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))
  )
)
```

Implicit Data in Activation Records

- An activation record includes implicit data needed by the function invocation
  - Storage for temporary values
  - Address where to place the return value
  - Address of caller's code and activation record
    - This is the caller's ***continuation*** (future topic)

- The set of implicit items can be determined statically

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))
  )
)
```

Space Usage of Factorial

- Computation of `factorial(n)` requires $n + 1$ invocations to be active at the same time

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))
  )
)
```

- Compare to iterative version in Python:

```
def factorial_iter(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

Alternate Definition of Factorial

- We can define another recursive version that:
  - Does no computation after the recursive call
  - Directly returns the result of the recursive call

```
(define (factorial-tail n)
  (factorial-tail-helper n 1)
)

(define (factorial-tail-helper n result)
  (if (= n 0)
      result
      (factorial-tail-helper (- n 1) (* result n))
  )
)
```

```
def factorial_iter(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

Tail-Call Optimization

- A call is a ***tail call*** if its caller directly returns the result without performing additional computation
- ***Tail-call optimization*** reuses the space for the caller's activation record for that of the tail call
- Some implicit data is also reused for the tail call:
  - Address where to place return value
  - Address of caller's code and activation record

```
(define (factorial-tail-helper n result)
  (if (= n 0)
      result
      (factorial-tail-helper (- n 1) (* result n))
  )
)

(display (factorial-tail-helper 4 1))
```

Tail-Call Optimization Failures

- Implicit computation, such as destructors, can prevent optimization

```
int sum(vector<int> values, int index,
        int partial_result = 0) {
  if (values.size() == index) { return 0; }
  return sum(values, index + 1,
             partial_result + values[index])
}
```

- Nested function definitions can prevent optimization