Chapter 9 - Subprograms

Mark Snyder
Outline

1. Subprogram Basics
2. Parameter Passing Styles
3. Implementation Questions
4. Overloading and Generics
5. Closures
process abstraction

many nearly-identical names. procedures and functions are both kinds of subprograms.

- **subprogram**: collection of statements to run, with perhaps some arguments and returned value.
- **procedure**: like a big statement. (traditionally no return value)
- **function**: like a big expression (results in a value).
  - based on mathematical functions. supposedly no side-effects, but most languages allow then.
- **method**: object’s function that can be called upon request, accessing object’s state
header: “signature” or “prototype” containing some or all of:
  - name, kind of subprogram, formal parameters list, return type.

nesting of subprograms: started with Algol 60. Most languages disallow this.
parameters

- **formal parameter** ("parameter"): part of subprogram definition. Given name and possibly type
- **actual parameter** ("argument"): part of subprogram call. Given some sort of value.
- variable numbers of parameters: gives an array of any supplied args.
- matching: positional, keyword, default
  - positional: in order
  - keyword: by name, also via dictionary-passing
  - default: by omission and predefined backup value
Python Parameter Examples

Positional/Default/Keyword Params

def addThem (x=7,y=8,z=9):
    return x+y+z

print("#1: " , addThem())
print("#2: " , addThem(1))
print("#3: " , addThem(1,2))
print("#4: " , addThem(y=2,z=3,x=1))
print("#5: " , addThem(x=1,z=3))

see params.py
Python Example

def addThem(*vals):
    sum = 0
    for v in vals:
        sum += v
    return sum

uses

addThem() # 0
addThem(1,2,3,4) # 10
addThem(1.1, 3.4) # 4.5
xs = [1,2,3,4,5,6]
addThem(*xs) # 21

- vals is a tuple of all given arguments, in order
Polyvariadic Examples

Java Polyvariadic Example

```java
public class Polyvariadic {
    public static int numargs(Object... args){
        return args.length;
    }
    public static int numstrs (String... args) {
        return args.length;
    }
}
```

- see Polyvariadic.java
Implications the function can accept any number of parameters, with any names! Just pack them into a dictionary.

```
def reportThem(**params):
    for (key,val) in params.items():
        print(f"key={key}", f"val={val}")
```

```
d = {"x":1,"y":2}
reportThem(a=1,b=2,z=3)
reportThem(**d)
```
We get to feed lists or many values as the arguments, to functions that expect many or variadic args.

```python
various definitions
def f1 (*xs):
    for x in xs:
        print(x)
def f2 (**kps):
    for k in kps:
        print(kps[k])
def f3 (a,b,c):
    print(a,b,c)

uses
def main():
    f1(1,2,3)
    f2(a=1,b=2,c=3)
    nums = [1,2,3]
    f3(*nums)
    d = {"a":1,"b":2,"c":3}
    f3(**d)
```
What can be used for arguments?

what is allowed as a parameter? Whatever we allow, they are called “first class values”.

- primitive values?
- array or other structured values?
- addresses/references?
- types?
- functions? (not function calls)
parameter modes

- **in**: send in a value, but subprogram can’t affect the source’s version.
- **out**: let subprogram send value to caller. (like a named return value)
- **in-out**: both in and out through same parameter.
passing approaches: in

- **pass by value**: copy the actual value, send to subprogram.
  - recipient won’t/can’t affect the original.
  - takes time/space to copy. (linear to value size)

**example: Java primitives**

```java
public void noEffect(int x, int y) {
    // only local variables (params) modified
    x ++;
    y = x*100;
}
```
passing approaches: out

- **pass by result**: a sort of ‘named’ return value
  - no initial value sent to subprogram; used as local variable
  - last value locally stored is sent back to caller
  - what if two aliases are both sent? f(p1, p1)
  - what if one uses another? f(xs[p],p)

**example: C (see makeitfive.c)**

```c
void makeItFive (int* x){ *x = 5; }
int main(){
    int a = 3;
    makeItFive (&a);
    printf("a=%d\n",a);
}
```
C# allows out parameters with the `out` modifier:

```csharp
static void twoUpdates(out int x, out int y) {
    x = 100;
    y = 300;
}
```
pass by value-result: both in mode and out mode (“pass by copy”)
- all the pros/cons from both.
- local storage

implications
- copy of argument is used locally
- copy of last value is actively re-assigned at call site when the subprogram returns
passing approaches: in and out

pass by reference: copy the address of the value, send that to subprogram. (“pass by sharing”)

- in-out-mode (via access paths)
- recipient can affect the addressed value (but not the original address-copy!)
- constant time to copy the address.
- aliasing (between caller/callee; also between multiple params)

Examples

- C language: pointer parameters are effectively p.b.r
- Java: all non-primitive types are p.b.r.
pass by assignment: Python example: everything is a reference to an object (even for primitive types!).

- parameters are assigned the (reference) values from actual parameters, behaving like pass-by-reference
- except: many values are immutable, so the aliasing behavior is not observed.

```python
def f(vals, v):
    vals[v] = 99
v = -5
xs = [1,2,3]
x = 1
f(xs,x)
print (xs,x)  # "([1,99,3],1)"
```
passing approaches: in and out

**pass by name**: expression-argument isn’t evaluated until its usage is reached in executing the subprogram.

- Also: **re-evaluated** each time it is reached!
- allows for creating your own control structures!
- very odd to reason about; introduced in Algol 60, but largely not available to programmers now.
- implementation: a closure (or thunk)
Haskell and Pass-By-Name

Haskell uses a version of pass by name called **pass by need**.

- except, we have referential transparency! There’s no need to re-calculate the result that we’ve cached.
- this “memoized call-by-name” evaluates each parameter *at most once*

**a quick calculation**

```haskell
f a b = if a then (b,b,b,b,b,b,b) else 0
-- never evals fib
example1 = f False fib(10000)
-- still just evals once.
example2 = f True fib(10000)
```
Type-checking Parameters

- Parameters are usually given specific types
- Lang enforces these types, perhaps adding coercions
- In the original C, you could have un-checked parameters:

Old C example

```c
double fun (x,y)
  double x;
  int y;
  { return x+y; }

/*bad-but-allowed call: */
double rv = fun(150, 20.6);
/* your compiler might help with a cast */
```
can definitions be nested?
are local variables static or dynamic?
often, languages choose stack-dynamic.
  - supports recursion
  - but takes more space/time to execute
  - indirect addressing
  - no “C statics” behavior (history)
decisions, decisions...

- how are parameters passed? Can programmer choose?
- is type checking performed?
- generics or polymorphism allowed?
- what is the referencing environment? (are thunks/closures allowed?)
data flow

- what kinds of parameters are allowed (how is data passed to callee)
- what kinds of return values are allowed (how is data passed back to caller)
- aliasing behaviors
control flow

- caller’s evaluation is suspended, awaits return of control (and return values) from callee.
- single entry, multiple exits (creating threads)
- multiple entries, same return locations (e.g., longjmp calls)
- always return to caller
- return values allowed or not?
  - using out mode parameters
  - using return statements
organizing things

- place values, references, etc. on the stack or in registers
- are side effects allowed? (in mode helps reduce side effects)
- type checking: very important!
  - Python: variables don’t have types, only objects do. So parameters can’t be typechecked (!!!)
subprograms as parameters

- does it bring its own referencing environment?
  - what would a variable named \( x \) mean when the subprogram is called in this new location?

- shallow binding: use local env. when sub is actually executed (dynamic scoping).

- deep binding: use the env. from sub’s original definition (static scoping).

- ad hoc binding: use the env. from passer’s env. (…only exists to torture CS students on test days)
examples of various bindings

Python-esque code

```python
s="glob"
def f1(other):
    s="first"
    other()
def f2():
    print(s)
def main():
    s="main"
    f1(f2)
```

Notes

- shallow: f2 prints “first”
  - closest def’n of s when f2 was called
- deep: f2 prints “glob”
  - based on f2’s original static scope
- ad hoc: f2 prints “main”
  - closure of f2 code created at call of f1(f2); s=’’main’’ then.
Other subprogram-as-parameter approaches

- C langs: function pointers
- Haskell, Python: functions are first-class. Static scoping.

Haskell

```haskell
map (<10) [1..10]
map (map (+1)) xss
```

Python

```python
def inc(x):
    return x+1
def main():
    xs = [1,2,3]
    ys = map(inc,xs)
    print(list(ys))
```
Multi-dimensional Arrays

- pass by reference most likely - avoid copying large things
- must know sizes of dimensions.
  - separate compilation: compiler must account for sizes.

Examples

- **C dynamic arrays**: pass pointer, also extra params for array lengths of all dimensions
- **Ada**: dimensions can be part of the type (constrained arrays), or part of the declaration (unconstrained arrays)
- **Java**: dimension sizes are part of the structure (.length)
overloading

Allowing multiple subprograms with the same name.

- need different formal parameter lists (number or types of params).
- Ada also could differentiate by return type (unusual)
- provides one type of polymorphism: a named subprogram can operate over multiple types.
Various Kinds of Polymorphism

- **subtype polymorphism**: derived/extended types can behave the same as the parent/base type.
  - ex: OOP, subclasses
- **parametric polymorphism**: any type may be used for a parameter, because its value is never directly inspected by this function.
  - ex: Haskell type params; found in map, foldr, (++), etc.
  - ex: C++ templates
  - ex: Java Generics (must be classes).
- **ad hoc polymorphism**: definitions at various types are individually created; uses in these ways are acceptable.
  - ex: implementing Java interfaces gives us ad-hoc def’ns
  - ex: placing bounds on generic types in Java.
  - ex: type class instances in Haskell.
Operators are “special” (baked into language), so being allowed to override them is something of a privilege.

**Python**

Many operators are actually syntactic sugar for methods.

- `define __add__(self, other)` to overload the `+` operator
Overloading Haskell Operators

- All operators are just functions.
- If they are in a typeclass, they may be defined in an instance for some new type.

```
Num typeclass instance

data Complex = C Double Double deriving (Show, Eq)
instance Num Complex where
    (C r1 i1) + (C r2 i2) = C (r1+r2) (i1+i2)
    (C r1 i1) - (C r2 i2) = C (r1-r2) (i1-i2)
    (C r1 i1) * (C r2 i2) = C (r1*r2 - i1*i2)
                        (r1*i2 + i1*r2)
```

What is a closure?

**Closure**: both a subprogram and its referencing environment.

- a subprogram that’s passed around/called later.
- only needed when the subprogram can access variables at its own definition site that it won’t see at its call site, or that may otherwise have ended their lifetimes by the time it is called.
- goal: make the ref.env. as minimal as possible (don’t copy the entire set of accessible variables at that point!)

**Thunk**: just another name for a closure.

**Common approach**: Make an anonymous function.
Haskell’s **call by need** semantics means that each sub-expression is effectively a closure/thunk.

- Haskell: everything’s a Thunk.
  - every function call
  - every sub-expression
  - nothing is computed until needed, and even then, just as deep as necessary to get an answer.
def addMore ():
    more = 10
    # regardless of calling env., more==10 always.
    return (lambda a : a+more)

def main():
    more = 5
    f = addMore ()
    print (f(2))  #only uses more=10.
    more = 7
    print (f(100)) #only uses more=10.
Coroutines are subprograms that can yield and resume.
- like a function-level process that cooperates with other coroutines, to all yield/resume/yield/resume together.
- simulates concurrency

Python generator

```python
def fives():
    x = 0
    while x<100:
        yield x
        x += 5
    raise StopIteration()
```

Using a generator

```python
def main ():
    mults = fives()
    try :
        while True:
            print(next(mults))
    except StopIteration:
        pass
```