MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Lecture Notes: Feb. 5

The PCAP framework for controlling complexity

Some simple Python procedures

```
def square(x):
    return x*x

def average(a,b):
    return (a + b) / 2.0

def meanSquare(a,b):
    return average(square(a), square(b))
```

Hero of Alexandria's algorithm for computing square roots:

To compute an approximation to the square root of x:

1. Let g be a guess for the answer
2. Compute an improved guess by taking the average of g and x/g
3. Keep improving the guess until its good enough.

A procedure for computing square roots:

```
def goodEnough(guess, x):
    return abs(x-square(guess)) < .00001

def improve(guess,x):
    return average(guess, x/guess)

def sqrtIter(guess,x):
    while not(goodEnough(guess,x)):
        guess=improve(guess,x)
    return guess

def sqrt(x):
    return sqrtIter(1.0,x)
```

Another version of the square root procedure, which uses block structure

```
def sqrt(x):
    def goodEnough(guess):
        return abs(x-square(guess)) < .00001
    def improve(guess):
        return average(guess, x/guess)
    def iter(guess):
        while not(goodEnough(guess)):
            guess=improve(guess)
        return guess
    return iter(1.0)
```

Computing powers, b^e

```
def expt(b,e):
    if e==0:
        return 1
    else:
        return b*expt(b,e-1)
```

This results in a **linear time process**

Fast exponentiation:

```
def fastexp(b,e):
    if e == 0:
        return 1
    elif e % 2 == 1:
        return b * fastexp(b,e-1)
    else:
        return square(fastexp(b,e/2))
```

This results in a **logarithmic time process**

A procedure for evaluating polynomials. (Uses list comprehension.)

```
def evalPoly(p,x):
    m=len(p)
    d=m-1
    return sum([p[i] * x**(d-i) for i in range(m)])
```

Evaluating polynomials with Horner's rule

```
def horner(p,x):
    result = 0
    for coeff in p:
        result = coeff + x*result
    return result
```

Recap of the PCAP framework (to continue next week)

	Procedures	Data
Primitives	$+, *, /, ==$	numbers, strings
Means of combination	if, while, $3*(4+7)$, list comprehension	lists
Means of abstraction	def	??
Capturing common patterns	??	??

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Course notes for Week 2

1 Higher-order functions

Another element of functional programming style is the idea of functions as *first-class objects*. That means that we can treat functions or procedures in much the same way we treat numbers or strings in our programs: we can pass them as arguments to other procedures and return them as the results from procedures. This will let us capture important common patterns of abstraction and will also be an important element of object-oriented programming.

We've been talking about the fundamental principles of software engineering as being modularity and abstraction. But another important principle is laziness! Don't ever do twice what you could do only once.[1] This standardly means writing a procedure whenever you are going to do the same computation more than once. In this section, we'll explore ways of using procedures to capture common patterns in ways that might be new to you.

What if we find that we're often wanting to perform the same procedure twice on an argument? That is, we seem to keep writing `square(square(x))`. If it were always the same procedure we were applying twice, we could just write a new function

```
def squaretwice(x):
    return square(square(x))
```

But what if it's different functions? The usual strategies for abstraction seem like they won't work.

In the functional programming style, we treat functions as first-class objects. So, for example, we can do:

```
>>> m = square
>>> m(7)
49
```

And so we can write a procedure that consumes a function as an argument. So, now we could write:

```
def doTwice(f, x):
    return f(f(x))
```

This is cool, because we can apply it to any function and argument. So, if we wanted to square twice, we could do:

[1]Okay, so the main reason behind this rule isn't laziness. It's that if you write the same thing more than once, it will make it harder to write, read, debug, and modify your code reliably.

```
>>> doTwice(square,2)
16
```

Python gives us a way to define functions without naming them. The expression `lambda y: y + y` denotes a function of a single variable; in this case it is a function that takes numbers as input, and doubles them. In Python, `lambda` doesn't require a `return` expression, and it can only be used with a single expression; you can't have `if` or `for` inside a `lambda` expression. If you need to put those in a function, you have to name that function explicitly.

Another way to apply a procedure multiple times is this:

```
def doTwiceMaker(f):
    return lambda x: f(f(x))
```

This is a procedure that *returns a procedure*! If you'd rather not use `lambda`, you could write it this way:

```
def doTwiceMaker(f):
    def twoF(x):
        return f(f(x))
    return twoF
```

Now, to use `doTwiceMaker`, we could do:

```
>>> twoSquare = doTwiceMaker(square)
>>> twoSquare(2)
16
>>> doTwiceMaker(square)(2)
16
```

A somewhat deeper example of capturing common patterns is sums. Mathematicians have invented a notation for writing sums of series, such as

$$\sum_{i=1}^{100} i \quad \text{or} \quad \sum_{i=1}^{100} i^2 \ .$$

Here's one that gives us a way to compute π:

$$\pi^2/8 = \sum_{i=1,3,5,\ldots} \frac{1}{i^2} \ .$$

It would be easy enough to write a procedure to compute any one of them. But even better is to write a higher-order procedure that allows us to compute any of them, simply:

```
def summation(low, high, f, next):
    s = 0
    x = low
    while x <= high:
        s = s + f(x)
        x = next(x)
    return s
```

This procedure takes integers specifying the lower and upper bounds of the index of the sum, the function of the index that is supposed to be added up, and a function that computes the next value of the index from the previous one. This last feature actually makes this notation more expressive than the usual mathematical notation, because you can specify any function you'd like for incrementing the index. Now, given that definition for sum, we can do all the special cases we described in mathematical notatation above:

```
def sumint(low,high):
    return summation(low, high, lambda x: x, lambda x: x+1)

def sumsquares(low,high):
    return summation(low, high, lambda x: x**2, lambda x: x+1)

def piSum(low,high):
    return summation(low, high, lambda x: 1.0/x**2, lambda x: x+2)

>>> (8 * piSum(1, 1000000))**0.5
3.1415920169700393
```

Now, we can use this to build an even cooler higher-order procedure. You've seen the idea of approximating an integral using a sum. We can express it easily in Python, using our `sum` higher-order function, as

```
def integral(f, a, b):
    dx = 0.0001
    return dx * summation(a, b, f, lambda(x): x + dx)

>>> integral(lambda x: x**3, 0, 1)
0.2500500024999337
```

We'll do one more example of a very powerful higher-order procedure. In the previous chapter, we saw an iterative procedure for computing square roots. We can see that procedure as a special case of a more general process of computing the *fixed-point* of a function. The fixed point of a function f, called f^*, is defined as the value such that $f^* = f(f^*)$. Fixed points can be computed by starting at an initial value v_0, and iteratively applying f: $f^* = f(f(f(f(f(\ldots f(v_0))))))$. A function may have many fixed points, and which one you'll get may depend on the v_0 you start with.

We can capture this idea in Python with:

```
def closeEnough(g1,g2):
    return abs(g1-g2)<.0001
def fixedPoint(f,guess):
    next=f(guess)
    while not closeEnough(guess, next):
        guess=next
        next=f(next)
    return next
```

And now, we can use this to write the square root procedure much more compactly:

```
def sqrt(x):
    return fixedPoint(lambda g: average(g,x/g), 1.0)
```

Another way to think of square roots is that to compute the square root of x, we need to find the y that solves the equation

$$x = y^2 \ ,$$

or, equivalently,

$$y^2 - x = 0 \ .$$

We can try to solve this equation using Newton's method for finding roots, which is another instance of a fixed-point computation.[2] In order to find a solution to the equation $f(x) = 0$, we can find a fixed point of a different function, g, where

$$g(x) = x - \frac{f(x)}{Df(x)} \ .$$

The first step toward turning this into Python code is to write a procedure for computing derivatives. We'll do it by approximation here, though there are algorithms that can compute the derivative of a function analytically, for some kinds of functions (that is, they know things like that the derivative of x^3 is $3x^2$.)

```
dx = 1.e-4
def deriv(f):
    return lambda x:(f(x+dx)-f(x))/dx

>>> deriv(square)
<function <lambda> at 0x00B96AB0>

>>> deriv(square)(10)
20.000099999890608
```

Now, we can describe Newton's method as an instance of our `fixedPoint` higher-order procedure.

```
def newtonsMethod(f,firstGuess):
    return fixedPoint(lambda x: x - f(x)/deriv(f)(x), firstGuess)
```

How do we know it is going to terminate? The answer is, that we don't really. There are theorems that state that if the derivative is continuous at the root and if you start with an initial guess that's close enough, then it will converge to the root, guaranteeing that eventually the guess will be close enough to the desired value. To guard against runaway fixed point calculations, we might put in a maximum number of iterations:

```
def fixedPoint(f, guess, maxIterations = 200):
    next=f(guess)
    count = 0
    while not closeEnough(guess, next) and count < maxIterations:
```

[2] Actually, this is Raphson's improvement on Newton's method, so it's often called the Newton-Raphson method.

```
        guess=next
        next=f(next)
        count = count + 1
    if count == maxIterations:
        print "fixedPoint terminated without desired convergence"
    return next
```

The `maxIterations = 200` is a handy thing in Python: an optional argument. If you supply a third value when you call this procedure, it will be used as the value of `maxIterations`; otherwise, `200` will be used.

Now, having defined `fixedPoint` as a black box and used it to define `newtonsMethod`, we can use `newtonsMethod` as a black box and use it to define `sqrt`:

```
def sqrt(x):
    return newtonsMethod(lambda y:y**2 - x, 1.0)
```

So, now, we've shown you two different ways to compute square root as an iterative process: directly as a fixed point, and indirectly using Newton's method as a solution to an equation. Is one of these methods better? Not necessarily. They're roughly equivalent in terms of computational efficiency. But we've articulated the ideas in very general terms, which means we'll be able to re-use them in the future.

We're thinking more efficiently. We're thinking about these computations in terms of more and more general methods, and we're expressing these general methods themselves as procedures in our computer language. We're capturing common patterns as things that we can manipulate and name. The importance of this is that if you can name something, it becomes an idea you can use. You have power over it. You will often hear that there are any ways to compute the same thing. That's true, but we are emphasizing a different point: There are many ways of expressing the same computation. It's the ability to choose the appropriate mode of expression that distinguishes the master programmer from the novice.

1.1 Map

What if, instead of adding 1 to every element of a list, you wanted to divide it by 2? You could write a special-purpose procedure:

```
def halveElements(list):
    if list == []:
        return []
    else:
        return [list[0]/2.0] + halveElements(list[1:])
```

First, you might wonder why we divided by `2.0` rather than `2`. The answer is that Python, by default, given two integers does integer division. So `1/2` is equal to 0. Watch out for this, and if you don't want integer division, make sure that you divide by a float.

So, back to the main topic: `halveElements` works just fine, but it's really only a tiny variation on `incrementElements1`. We'd rather be lazy, and apply the principles of modularity and abstraction so we don't have to do the work more than once. So, instead, we write a generic procedure, called `map`, that will take a procedure as an argument, apply it to every element of a list, and return a list made up of the results of each application of the procedure.

```
def map(func, list):
    if list == []:
        return []
    else:
        return [func(list[0])] + map(func, list[1:])
```

Now, we can use it to define `halveElements`:

```
def halveElements(list):
    return map(lambda x: x/2.0, list)
```

It's generally considered more in the Python style (or "Pythonic") to use list comprehensions than map, but it's good to know we could have implemented this if we had needed it. For completeness here's how to do the same thing with a list comprehension:

```
def halveElements1(list):
    return [x/2.0 for x in list]
```

1.2 Reduce

Another cool thing you can do with higher-order programming is to use the `reduce` function. Reduce takes a binary function and a list, and returns a single value, which is obtained by repeatedly applying the binary function to pairs of elements in the list. So, if the list contains elements $x_1 \dots x_n$, and the function is f, the result will be $f(\dots, f(f(x_1, x_2), x_3), \dots, x_n)$.

This would let us write another version of `addList`:

```
def addList7(list):
    return reduce(add, list)
```

Note that we have to define the `add` function, as

```
def add(x, y):  return x + y
```

or import it from the `operator` package, by doing

```
from operator import add
```

You can also use `reduce` to concatenate a list of lists. Remembering that the addition operation on two lists concatenates them, we have this possibly suprising result:

```
>>> reduce(add, [[1, 2, 3],[4, 5],[6],[]])
[1, 2, 3, 4, 5, 6]

>>>addList7([[1, 2, 3],[4, 5],[6],[]])
[1, 2, 3, 4, 5, 6]
```

1.2.1 Filter

Yet another handy higher-order-function is `filter`. Given a list and a function of a single argument, it returns a new list containing all of the elements of the original list for which the function returns True. So, for example,

```
>>> filter(lambda x: x % 3 == 0, range(100))
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51,
54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

returns all of the multiples of 3 less than 100.

This is also done nicely with a list comprehension:

```
>>> [x for x in range(100) if x%3 == 0]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51,
54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

1.3 Conclusion

The combination of `map`, `reduce`, and `filter` is a very powerful computational paradigm, which lets you say what you want to compute without specifying the details of the order in which it needs to be done. Google uses this paradigm for a lot of their computations, which allows them to be easily distributed over large numbers of CPUs.

We'll find that it's useful to use these and other patterns involving higher-order functions in many situations, especially in building up new PCAP systems of our own.

2 On Style

Software engineering courses often provide very rigid guidelines on the style of programming, specifying the appropriate amount of indentation, or what to capitalize, or whether to use underscores in variable names. Those things can be useful for uniformity and readability of code, specially when a lot of people are working on a project. But they are mostly arbitrary: a style is chosen for consistency and according to some person's aesthetic preferences.

There are other matters of style that seem, to us, to be more fundamental, because they directly affect the readability or efficiency of the code.

- **Avoid recalculation of the same value.** You should compute it once and assign it to a variable instead; otherwise, if you have a bug in the calculation (or you want to change the program), you will have to change it multiple times. It is also inefficient.
- **Avoid repetition of a pattern of computation.** You should use a function instead, again to avoid having to change or debug the same basic code multiple times.
- **Avoid numeric indexing.** You should use destructuring if possible, since it is much easier to read the code and therefore easier to get right and to modify later.

- **Avoid excessive numeric constants.** You should name the constants, since it is much easier to read the code and therefore easier to get right and to modify later.

Here are some examples of simple procedures that exhibit various flaws. We'll talk about what makes them problematic.

Normalize a vector Let's imagine we want to normalize a vector of three values; that is to compute a new vector of three values, such that its length is 1. Here is our first attempt; it is a procedure that takes as input a list of three numbers, and returns a list of three numbers:

```
def normalize3(v):
    return [v[0]/math.sqrt(v[0]**2+v[1]**2+v[2]**2),
            v[1]/math.sqrt(v[0]**2+v[1]**2+v[2]**2),
            v[2]/math.sqrt(v[0]**2+v[1]**2+v[2]**2)]
```

This is correct, but it looks pretty complicated. Let's start by noticing that we're recalculating the denominator three times, and instead save the value in a variable.

```
def normalize3(v):
    magv = math.sqrt(v[0]**2+v[1]**2+v[2]**2)
    return [v[0]/magv,v[2]/magv,v[3]/magv]
```

Now, we can see a repeated pattern, of going through and dividing each element by `magv`. Also, we observe that the computation of the magnitude of a vector is a useful and understandable operation in its own right, and should probably be put in its own procedure. That leads to this procedure:

```
def mag(v):
    return math.sqrt(sum([vi**2 for vi in v]))

def normalize3(v):
    return [vi/mag(v) for vi in v]
```

This is especially nice, because now, in fact, it applies not just to vectors of length three. So, it's both shorter and more general than what we started with. But, one of our original problems has snuck back in: we're recomputing `mag(v)` once for each element of the vector. So, finally, here's a version that we're very happy with:

```
def mag(v):
    return math.sqrt(sum([vi**2 for vi in v]))

def normalize3(v):
    magv = mag(v)
    return [vi/magv for vi in v]
```

Perimeter of a polygon Now, let's consider the problem of computing the perimeter of a polygon. The input is a list of vertices, encoded as a list of lists of two numbers (such as `[[1, 2], [3.4, 7.6], [-4.4, 3]]`). Here is our first attempt:

```
def perim(vertices):
    result = 0
    for i in range(len(vertices)-1):
        result = result + math.sqrt((vertices[i][0]-vertices[i+1][0])**2 + \
                                    (vertices[i][1]-vertices[i+1][1])**2)
    return result + math.sqrt((vertices[-1][0]-vertices[0][0])**2 + \
                                    (vertices[-1][1]-vertices[0][1])**2)
```

Again, this works, but it ain't pretty. The main problem is that someone reading the code doesn't immediately see what all that subtraction and squaring is about. We can fix this by defining another procedure:

```
def perim(vertices):
    result = 0
    for i in range(len(vertices)-1):
        result = result + pointDist(vertices[i],vertices[i+1])
    return result + pointDist(vertices[-1],vertices[0])

def pointDist(p1,p2):
    return math.sqrt(sum([(p1[i] - p2[i])**2 for i in range(len(p1))]))
```

Now, we've defined a new procedure `pointDist`, which computes the Euclidean distance between two points. And, in fact, we've written it generally enough to work on points of any dimension (not just two). Just for fun, here's another way to compute the distance, which some people would prefer and others would not.

```
def pointDist(p1,p2):
    return math.sqrt(sum([(c1 - c2)**2 for (c1, c2) in zip(p1, p2)]))
```

For this to make sense, you have to understand `zip`. Here's an example of how it works:

```
> zip([1, 2, 3],[4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
```

Bank transfer What if we have two values, representing bank accounts, and want to transfer an amount of money `amt` between them? Assume that a bank account is represented as a list of values, such as `['Alyssa', 8300343.03, 0.05]`, meaning that `'Alyssa'` has a bank balance of $8,300,343.03, and has to pay a 5-cent fee for every bank transaction. We might write this procedure as follows. It moves the amount from one balance to the other, and subtracts the transaction fee from each account.

```
def transfer(a1, a2, amt):
    a1[1] = a1[1] - amt - a1[2]
    a2[1] = a2[1] + amt - a2[2]
```

To understand what it's doing, you really have to read the code at a detailed level. Furthermore, it's easy to get the variable names and subscripts wrong.

Here's another version that abstracts away the common idea of a deposit (which can be positive or negative) into a procedure, and uses it twice:

```
def transfer(a1, a2, amt):
    deposit(a1, -amt)
    deposit(a2, amt)

def deposit(a, amt):
    a[1] = a[1] + amt - a[2]
```

Now, `transfer` looks pretty clear, but `deposit` could still use some work. In particular, the use of numeric indices to get the components out of the bank account definition is a bit cryptic (and easy to get wrong).[3]

```
def deposit(a, amt):
    (name, balance, fee) = a
    a[1] = balance + amt - fee
```

Here, we've used a destructuring assignment statement to give names to the components of the account. Unfortunately, when we want to change an element of the list representing the account, we still have to index it explicitly. Given that we have to use explicit indices, this approach in which we name them might be better.

```
acctName = 0
acctBalance = 1
acctFee = 2
def deposit(a, amt):
    a[acctBalance] = a[acctBalance] + amt - a[acctFee]
```

Strive, in your programming, to make your code as simple, clear, and direct as possible. Occasionally, the simple and clear approach will be too inefficient, and you'll have to do something more complicated. In such cases, you should still start with something clear and simple, and in the end, you can use it as documentation.

[3]We'll see other approaches to this when we start to look at object-oriented programming. But it's important to apply basic principles of naming and clarity no matter whether you're using assembly language or Java.

14. Introduction to higher-order functions

Higher-order functions is another key area in the functional programming paradigm; Perhaps the most important at all. In this chapter we will explore this exiting area, and we will give a number of web-related examples.

14.1. Higher-order functions

Lecture 4 - slide 2

The idea of higher-order functions is of central importance for the functional programming paradigm. As we shall see on this and the following pages, this stems from the fact that higher-order functions can be further generalized by accepting functions as parameters. In addition, higher-order functions may act as function generators, because they allow functions to be returned as the result from other functions.

Let us first define the concepts of higher-order functions and higher-order languages.

A *higher-order function* accepts functions as arguments and is able to return a function as its result

A *higher-order language* supports higher-order functions and allows functions to be constituents of data structures

When some functions are 'higher-order' others are bound to be 'lower-order'. What, exactly, do we mean by the 'order of functions'. This is explained in below.

- The order of data
    - **Order 0**: Non function data
    - **Order 1**: Functions with domain and range of order 0
    - **Order 2**: Functions with domain and range of order 1
    - **Order *k***: Functions with domain and range of order *k-1*

Order 0 data have nothing to do with functions. Numbers, lists, and characters are example of such data.

Data of order 1 are functions which work on 'ordinary' order 0 data. Thus order 1 data are the functions we have been concerned with until now.

Data of order 2 - and higher - are example of the functions that have our interest in this lecture.

With this understanding, we can define higher-order functions more precisely.

Functions of order *i*, *i* >= 2, are called higher-order functions

14.2. Some simple and general higher-order functions

Lecture 4 - slide 3

It is time to look at some examples of higher-order functions. We start with a number of simple ones.

The `flip` function is given in two versions below. `flip` takes a function as input, which is returned with reversed parameters, cf. Program 14.1.

The first version of `flip` uses the shallow syntactic form, discussed in Section 8.12. The one in ??? uses the raw lambda expression, also at the outer level.

```
(define (flip f)
  (lambda (x y)
    (f y x)))
```

Program 14.1 *The function flip changes the order of it's parameters. The function takes a function of two parameters, and returns another function of two parameters. The only difference between the input and output function of flip is the ordering of their parameters.*

The read expression in Program 14.1 and ??? are the values returned from the function flip.

```
(define flip
  (lambda (f)
    (lambda (x y)
      (f y x))))
```

Program 14.2 *An alternative formulation of flip without use of the sugared define syntax.*

The function `negate`, as shown in Program 14.3, takes a predicate `p` as parameter. `negate` returns the negated predicate. Thus, if `(p x)` is true, then `((negate p) x)` is false.

```
(define (negate p)
  (lambda (x)
    (if (p x) #f #t)))
```

Program 14.3 *The function negate negates a predicate. Thus, negate takes a predicate function (boolean function) as parameter and returns the negated predicate. The resulting negated predicate returns true whenever the input predicate returns false, and vise versa.*

The function `compose` in Program 14.4 is the classical function composition operator, known by all high school students as 'f o g'

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
```

Program 14.4 *The function compose composes two functions which both are assumed to take a single argument. The resulting function composed of* `f` *and* `g` *returns* `f(g(x))`*, or in Lisp* `(f (g x))`*, given the input* `x` *. The* `compose` *function from the general LAML library accepts two or more parameters, and as such it is more general than the compose function shown here.*

Exercise 4.1. *Using* `flip`*,* `negate`*, and* `compose`

Define and play with the functions `flip`, `negate`, and `compose`, as they are defined on this page .

Define, for instance, a flipped `cons` function and a flipped minus function.

Define the function `odd?` in terms of `even?` and `negate`.

Finally, compose two HTML mirror functions, such as `b` and `em`, to a new function.

Be sure you understand your results.

14.3. Linear search in lists

Lecture 4 - slide 4

Let us program a simple, but useful higher-order function which searches a list by linear search. The function `find-in-list`, shown in Program 14.5 takes a predicate `pred` and a list `lst` as parameters. This predicate is applied on the elements in the list. The first element which satisfy the predicate is returned.

Search criterias can be passed as predicates to linear search functions

```
;; A simple linear list search function.
;; Return the first element which satisfies the predicate pred.
;; If no such element is found, return #f.
(define (find-in-list pred lst)
  (cond ((null? lst) #f)
        ((pred (car lst)) (car lst))
        (else (find-in-list pred (cdr lst)))))
```

Program 14.5 *A linear list search function. A predicate accepts as input an element in the list, and it returns either true (#t) or false (#f). If the predicate holds (if it returns true), we have found what we searched for. The predicate* `pred` *is passed as the first parameter to* `find-in-list`*. As it is emphasized in blue color, the predicate is applied on the elements of the list. The first successful application (an application with true result) terminates the search, and the element is returned. If the first case in the conditional succeeds (the brown condition) we have visited all elements in the list, and we conclude that the element looked for is not there. In that case we return false.*

The dialogue below shows examples of linear list search with `find-in-list`.

```
1> (define hair-colors (pair-up '(ib per ann) '("black" "green" "pink")))

2> hair-colors
((ib . "black") (per . "green") (ann . "pink"))

3> (find-in-list (lambda (ass) (eq? (car ass) 'per)) hair-colors)
(per . "green")

4> (find-in-list (lambda (ass) (equal? (cdr ass) "pink")) hair-colors)
(ann . "pink")

5> (find-in-list (lambda (ass) (equal? (cdr ass) "yellow")) hair-colors)
#f

6> (let ((pink-person
          (find-in-list
            (lambda (ass) (equal? (cdr ass) "pink")) hair-colors)))
    (if pink-person (car pink-person) #f))
ann
```

Program 14.6 *A sample interaction using* `find-in-list`*. We define a simple association list which relates persons (symbols) and hair colors (strings). The third interaction searches for per's entry in the list. The fourth interaction searches for a person with pink hair color. In the fifth interaction nothing is found, because no person has yellow hair color. In the sixth interaction we illustrate the convenience of boolean convention in Scheme: everything but #f counts as true. From a traditional typing point of view the* `let` *expression is problematic, because it can return either a person (a symbol) or a boolean value. Notice however, from a pragmatic point of view, how useful this is.*

Exercise 4.2. *Linear string search*

Lists in Scheme are linear linked structures, which makes it necessary to apply linear search techniques.

Strings are also linear structures, but based on arrays instead of lists. Thus, strings can be linearly searched, but it is also possible to access strings randomly, and more efficiently.

First, design a function which searches a string linearly, in the same way as `find-in-list`. Will you just replicate the parameters from `find-in-list`, or will you prefer something different?

Next program your function in Scheme.

Exercise 4.3. *Index in list*

It is sometimes useful to know *where in a list* a certain element occurs, if it occurs at all. Program the function `index-in-list-by-predicate` which searches for a given element. The comparion between the given element and the elements in the list is controlled by a comparison parameter to `index-in-list-by-predicate`. The function should return the list position of the match (first element is number 0), or #f if no match is found.

Some examples will help us understand the function:

```
(index-in-list-by-predicate '(a b c c b a) 'c eq?) => 2

(index-in-list-by-predicate '(a b c c b a) 'x eq?) => #f

(index-in-list-by-predicate '(two 2 "two") 2
  (lambda (x y) (and (number? x) (number? y) (= x y)))) => 1
```

Be aware if your function is tail recursive.

Exercise 4.4. *Binary search in sorted vectors*

Linear search, as illustrated by other exercises, is not efficient. It is often attractive to organize data in a sorted vector, and to do binary search in the vector.

This exercise is meant to illustrate a real-life higher-order function, generalized with several parameters that are functions themselves.

Program a function `binary-search-in-vector`, with the following signature:

```
(binary-search-in-vector v el sel el-eq? el-leq?)
```

`v` is the sorted vector. `el` is the element to search for. If `v-el` is an element in the vector, the actual comparison is done between `el` and `(sel v-el)`. Thus, the function `sel` is used as a selector on vector elements. Equality between elements is done by the `el-eq?` function. Thus, `(el-eq? (sel x) (sel y))` makes sense on elements x and y in the vector. The ordering of elements in the vector is defined by the `el-leq?` function. `(el-leq? (sel x) (sel y))` makes sense on elements x and y in the vector.

The call `(binary-search-in-vector v el sel el-eq? el-leq?)` searches the vector via binary search and it returns an element `el-v` from the vector which satisfies (el-eq? (sel el-v) el). If no such element can be located, it returns #f.

Here are some examples, with elements being cons pairs:

```
(binary-search-in-vector '#( (2 . x) (4 . y) (5 . z) (7 . i) (9 . c)
                             (11 . c)) 7 car = <=)    =>
(7 . i)

(binary-search-in-vector '#( (2 . x) (4 . y) (5 . z) (7 . i) (9 . c)
                             (11 . c)) 2 car = <=)    =>
(2 . x)
```

```
(binary-search-in-vector '#( (2 . x) (4 . y) (5 . z) (7 . i) (9 . c)
                             (11 . c)) 10 car = <=)   =>
#f
```

Be sure to program a tail recursive solution.

14.4. Generation of list selectors

Lecture 4 - slide 5

The function `find-in-list` took a function as parameter. In this section we will give an example of a higher-order function which returns a function as result.

It is attractive to *generate* generalizations of the list selector functions `car`, `cadr`, etc

The function `make-selector-function` generates a list selector function which returns element number *n* from a list. It should be noticed that the first element in a list is counted as number one. This is contrary to the convention of the function `list-ref` and other similar Scheme function, which counts the first element in a list as number zero. This explains the `(- n 1)` expression in Program 14.7.

```
(define (make-selector-function n)
  (lambda (lst) (list-ref lst (- n 1))))
```

Program 14.7 *A simple version of the* `make-selector-function` *function.*

In the web version of the material (slide view or annotated slide view) you will find yet another version of the function `make-selector-function`, which provides for better error messages, in case element number *n* does not exist in the list. We have taken it out of this version because of its size and format.

The dialogue below shows examples of definitions and uses of list selector functions generated by `make-selector-function`.

```
1> (define first (make-selector-function 1 "first"))

2> (first '(a b c))
a

3> (first '())
The selector function first: The list () is is too short for selection.
It must have at least 1 elements.
>

4> (define (make-person-record firstname lastname department)
      (list 'person-record firstname lastname department))
```

```
5> (define person-record
      (make-person-record "Kurt" "Normark" "Computer Science"))

6> (define first-name-of (make-selector-function 2 "first-name-of"))

7> (define last-name-of (make-selector-function 3 "last-name-of"))

8> (last-name-of person-record)
"Normark"

9> (first-name-of person-record)
"Kurt"
```

Program 14.8 *Examples usages of the function make-selector-function. In interaction 1 through 3 we demonstrate generation and use of the first function. Next we outline how to define accessors of data structures, which are represented as lists. In reality, we are dealing with list-based record structures. In my every day programming, such list structures are quite common. It is therefore immensely important, to access data abstractly (via name accessors, instead of via the position in the list (car, cadr, etc). In this context, the make-selector-function comes in handy.*

14.5. References

[-] Foldoc: higher order function
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=higher+order+function

15. Mapping and filtering

In this chapter we will focus on higher-order functions that work on lists. It turns out that the appropriate combinations of these make it possible to solve a variety of different list processing problems.

15.1. Classical higher-order functions: Overview

Lecture 4 - slide 7

We start with an overview of the classical higher-order functions on lists, not just mapping and filtering, but also including reduction and zipping functions which we cover in subsequent sections.

There exists a few higher-order functions via which a wide variety of problems can be solved by simple combinations

- Overview:
    - **Mapping**: Application of a function on all elements in a list
    - **Filtering**: Collection of elements from a list which satisfy a particular condition
    - **Accumulation**: Pair wise combination of the elements of a list to a value of another type
    - **Zipping**: Combination of two lists to a single list

The functions mentioned above represent abstractions of *algorithmic patterns* in the functional paradigm

The idea of patterns has been boosted in the recent years, not least in the area of object-oriented programming. The classical higher-order list functions encode recursive patterns on the recursive data type list. As a contrast to many patterns in the object-oriented paradigm, the patterns encoded by `map`, `filter`, and others, can be programmed directly. Thus, the algorithmic patterns we study here are not design patterns. Rather, they are programming patterns for the practical functional programmer.

15.2. Mapping

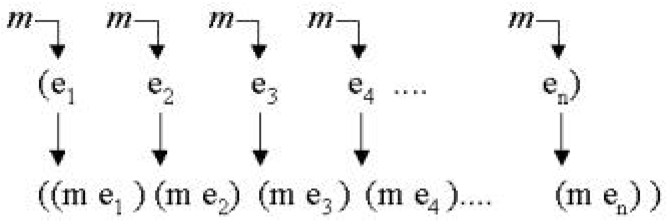
Lecture 4 - slide 8

The idea of mapping is to apply a function on each element of a list, hereby collecting the list of the function applications

A mapping function applies a function on each element of a list and returns the list of these applications

The function `map` is an essential Scheme function

The idea of mapping is illustrated below.



m
$(e_1 \quad e_2 \quad e_3 \quad e_4 \; .... \quad e_n)$
$((m\ e_1)\ (m\ e_2)\ (m\ e_3)\ (m\ e_4)....\ (m\ e_n))$

Figure 15.1 *Mapping a function m on a list. m is applied on every element, and the list of these applications is returned.*

15.3. The mapping function

Lecture 4 - slide 9

It is now time to study the implementation of the mapping function. We program a function called `mymap` in order not to redefine Scheme's own mapping function (a standard function in all Scheme implementations).

```
(define (mymap f lst)
  (if (null? lst)
      '()
       (cons (f (car lst))
             (mymap f (cdr lst)))))
```

Program 15.1 *An implementation of map. This is not a good implementation because the recursive call is not a tail call. We leave it as an exercise to make a memory efficient implementation with tail recursion - see the exercise below.*

Exercise 4.5. *Iterative mapping function*

In contrast to the function `mymap` on this page , write an iterative mapping function which is tail recursive.

Test your function against `mymap` on this page, and against the native `map` function of your Scheme system.

Exercise 4.6. *Table exercise: transposing, row elimination, and column elimination.*

In an earlier section we have shown the application of some very useful table manipulation functions. Now it is time to program these functions, and to study them in further details.

Program the functions `transpose`, `eliminate-row`, and `eliminate-column`, as they have been illustrated earlier. As one of the success criteria of this exercise, you should attempt to use higher-order functions as much and well as possible in your solutions.

Hint: Program a higher-order function, `(eliminate-element n)`. The function should return a function which eliminates element number n from a list.

15.4. Examples of mapping

Lecture 4 - slide 10

We will now study a number of examples.

Expresion	**Value**
`(map string? (list 1 'en "en" 2 'to "to"))`	`(#f #f #t #f #f #t)`
`(map (lambda (x) (* 2 x)) (list 10 20 30 40))`	`(20 40 60 80)`
`(ul (map (compose li b (lambda (x) (font-color red x))) (list "a" "b" "c") ) )`	(ul (map (compose li (compose b (lambda (x) (font-color red x)))) (list "a" "b" "c")))
Same as above	`(ul (map (compose li (compose b (lambda (x) (font-color red x)))) (list "a" "b" "c") ) )`

Table 15.1 *In the first row we map the* `string?` *predicate on a list of atoms (number, symbols, and strings). This reveals (in terms of boolean values) which of the elements that are strings. In the second row of the table, we map a 'multiply with 2' function on a list of numbers. The third row is more interesting. Here we map the composition of* `li` *,* `b` *, and red font coloring on the elements a, b, and c. When passed to the HTML mirror function* `ul` *, this makes an unordered list with red and bold items. Notice that the* `compose` *function used in the example is a higher-order function that can compose two or more functions. The function* `compose` *from* `lib/general.scm` *is such a function. Notice also that the HTML mirror function* `ul` *receives a list, not a string. The fifth and final row illustrates the raw HTML output, instead of the nicer rendering of the unordered list, which we used in the third row.*

15.5. Filtering

Lecture 4 - slide 11

As the name indicates, the `filter` function is good for examining elements of a list for a certain property. Only elements which possess the property are allowed through the filter.

A filtering function applies a predicate (boolean function) on every element of a list. Only elements on which the predicate returns true are returned from the filtering function.

The function `filter` is not an essential Scheme function - but is part of the LAML general library

The figure below illustrates the filtering idea.

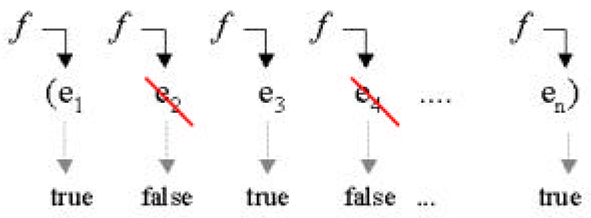


Figure 15.2 *Filtering a list with a predicate f. The resulting list is the subset of the elements which satisfy f (the elements on which f returns true).*

15.6. The filtering function

Lecture 4 - slide 12

The next item on the agenda is an implementation of `filter` .

For practical purposes it is important to have a memory efficient `filter` function

As a consequence of the observation above, we now program a tail recursive version of `filter`. Notice that it is the function `filter-help`, which does the real filtering job.

```
(define (filter pred lst)
  (reverse (filter-help pred lst '())))

(define (filter-help pred lst res)
  (cond ((null? lst) res)
        ((pred (car lst))
           (filter-help pred (cdr lst)  (cons (car lst) res)))
        (else
           (filter-help pred (cdr lst)  res))))
```

Program 15.2 *An implementation of filter which is memory efficient. If the predicate holds on an element of the list (the red fragment) we include the element in the result (the brown fragment). If not (the green fragment), we drop the element from the result (the purple fragment).*

Exercise 4.7. *A straightforward filter function*

The `filter` function illustrated in the material is memory efficient, using tail recursion.

Take a moment here to implement the straightforward recursive filtering function, which isn't tail recursive.

15.7. Examples of filtering

Lecture 4 - slide 13

As we did for mapping, we will also here study a number of examples. As before, we arrange the examples in a table where the example expressions are shown to the left, and their values to the right.

Expression	**Value**
`(filter even? '(1 2 3 4 5))`	`(2 4)`
`(filter (negate even?) '(1 2 3 4 5))`	`(1 3 5)`
`(ol (map li (filter string? (list 1 'a "First" "Second" 3))))`	1. First 2. Second
Same as above	`<ol><li>First</li> <li>Second</li></ol>`

Table 15.2 *In the first row we filter the first five natural numbers with the* `even?` *predicate. In the second row, we filter the same list of numbers with the* `odd?` *predicate. Rather than using the name odd? we form it by calculating* `(negate even?)` *. We have seen the higher-order function* `negate` *earlier in this lecture. The third and final example illustrates the filtering of a list of atoms with the* `string?` *predicate. Only strings pass the filter, and the resulting list of strings is rendered in an ordered list by means of the mirror function of the ol HTML element.*

15.8. References

[-] Foldoc: filter
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=filter

[-] The LAML general library
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/man/general.html

[-] Foldoc: map
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=map

16. Reduction and zipping

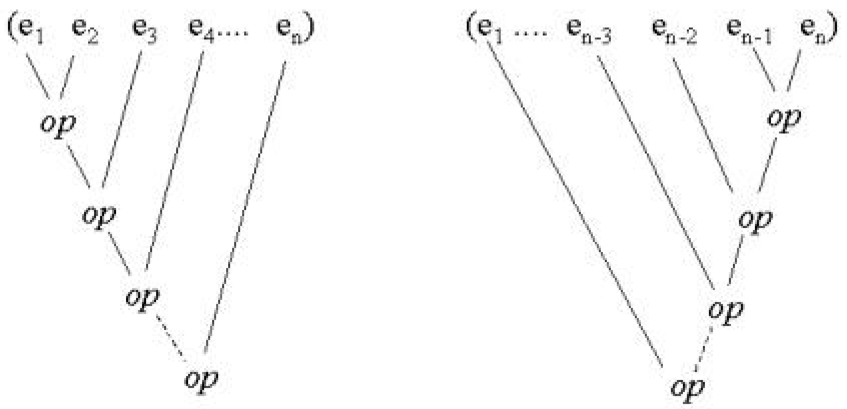
The reduction and zipping functions work on lists, like map and filter from Chapter 15.

16.1. Reduction

Lecture 4 - slide 15

List reduction is useful when we need somehow to 'boil down' a list to a 'single value'. The boiling is done with a binary function, as illustrated in Figure 16.1.

Reduction of a list by means of a binary operator transforms the list to a value in the range of the binary operator.



$(e_1\ e_2\ e_3\ e_4 \ldots\ e_n)$
op
op
op
op
$(e_1 \ldots\ e_{n-3}\ e_{n-2}\ e_{n-1}\ e_n)$
op
op
op
op

Figure 16.1 *Left and right reduction of a list. Left reduction is - quite naturally - shown to the left, and right reduction to the right.*

There is no natural value for reduction of the empty list. Therefore we assume as a precondition that the list is non-empty.

The intuitive idea of reduction will probably be more clear when we meet examples in Table 16.1 below.

Examples of left and right reduction are given in the table below. Be sure to understand the difference between left and right reduction, when the function, with which we reduce, is not commutative.

Expression	Value
`(reduce-left - '(1 2 3 4 5))`	`-13`
`(reduce-right - '(1 2 3 4 5))`	`3`
`(reduce-left string-append (list "The" " " "End"))`	`"The End"`
`(reduce-left append (list (list 1 2 3) (list 'a 'b 'c)))`	`(1 2 3 a b c)`

Table 16.1 *Examples of reductions. The - left reduction of the list corresponds to calculating the expression (- (- (- (- 1 2) 3) 4) 5). The - right reduction of the list corresponds to calculating the expression (- 1 (- 2 (- 3 (- 4 5)))).*

16.2. The reduction functions

Lecture 4 - slide 16

We will now implement the reduction functions introduced above in Section 16.1. Both right reduction and left reduction will be implemented, not least because they together illustrate a good point about iterative and tail recursive processing of lists. The explanations of this is found in the captions of Program 16.1 and Program 16.2.

The function `reduce-right` is a straightforward recursive function
The function `reduce-left` is a straightforward iterative function

```
(define (reduce-right f lst)
  (if (null? (cdr lst))
      (car lst)
      (f (car lst)
         (reduce-right f (cdr lst)))))
```

Program 16.1 *The function reduce-right. Notice the fit between the composition of the list and the recursive pattern of the right reduction.*

```
(define (reduce-left f lst)
  (reduce-help-left f (cdr lst) (car lst)))

(define (reduce-help-left f lst res)
  (if (null? lst)
      res
      (reduce-help-left f (cdr lst) (f res (car lst)))))
```

Program 16.2 *The function reduce-left. There is a misfit between left reduction and the recursive composition of the list with heads and tails. However, an iterative process where we successively combine e1 and e2 (giving r1), r1 and e3 etc., is straightforward. As we have seen several times, this can be done by a tail recursive function, here reduce-help-left.*

In summary, right reduction is easy to program with a recursive function. The reason is that we can reduce the problem to `(f (car lst) X)`, where `X` a right reduction of `(cdr lst)` with `f`. The right reduction of `(cdr lst)` is smaller problem than the original problem, and therefore we eventually meet the case where the list is trivial (in this case, a single element list).

The left reduction combines the elements one after the other, iteratively. First we calculate `(f (car el) (cadr el))`, provided that the list is of length 2 or longer. Let us call this value `Y`. Next `(f Y (caddr el))` is calculated, and so on in an iterative way. We could easily program this with a simple loop control structure, like a for loop.

16.3. Accumulation

Lecture 4 - slide 17

In this section we introduce a variation of reduction, which allows us also to reduce the empty list. We chose to use the word *accumulation* for this variant.

It is not satisfactory that we cannot reduce the empty list

We remedy the problem by passing an extra parameter to the reduction functions

We call this variant of the reduction functions for *accumulation*

It also turns out that the accumulation function is slightly more useful than `reduce-left` and `reduce-right` from Section 16.2. The reason is that we control the type of the parameter `init` to `accumulate-right` in Program 16.3. Because of that, the signature of the accumulate function becomes more versatile than the signatures of `reduce-left` and `reduce-right`. Honestly, this is not easy to spot in Scheme, whereas in languages like Haskell and ML, it would have been more obvious.

Below we show the function `accumulate-right`, which performs right accumulation. In contrast to `reduce-right` from Program 16.1`accumulate-right` also handles the extreme case of the empty list. If the list is empty, we use the explicitly passed `init` value as the result.

```
(define (accumulate-right f init lst)
  (if (null? lst)
      init
      (f (car lst) (accumulate-right f init (cdr lst)))))
```

Program 16.3 *The function accumulate-right. The recursive pattern is similar to the pattern of reduce-right.*

The table below shows a few examples of right accumulation, in the sense introduced above.

Expression	Value
`(accumulate-right - 0 '())`	`0`
`(accumulate-right - 0 '(1 2 3 4 5))`	`3`
`(accumulate-right append '() (list (list 1 2 3) (list 'a 'b 'c)))`	`(1 2 3 a b c)`

Table 16.2 *Examples of right accumulations. The first row illustrates that we can accumulate the empty list. The second and third rows are similar to the second and third rows in Table 15.1.*

In relation to web programming we most often append accumulate lists and strings

`accumulate-right` is part of the general LAML library

Due to their deficiencies, the reduction functions are not used in LAML

16.4. Zipping

Lecture 4 - slide 18

The zipping function is named after a zipper, as known from pants and shirts. The image below shows the intuition behind a list zipper.

Two equally long lists can be pair wise composed to a single list by means of *zipping* them

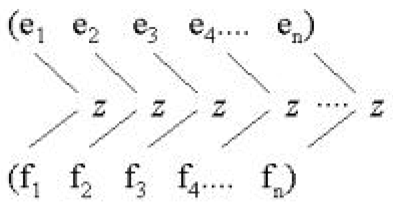

(e_1 e_2 e_3 e_4.... e_n)
z z z z .... z
(f_1 f_2 f_3 f_4.... f_n)

Figure 16.2 *Zipping two lists with the function `z`. The head of the resulting list is (`z` `e` $_i$ `f` $_i$), where the element `e` $_i$ comes from the first list, and `f` $_i$ comes from the other.*

We implement the zipping function in the following section.

16.5. The zipping function

Lecture 4 - slide 19

The `zip` function in Program 16.4 takes two lists, which are combined element for element. As a precondition, it is assumed that both input list have the same size.

```
(define (zip z lst1 lst2)
  (if (null? lst1)
      '()
      (cons
        (z (car lst1) (car lst2))
        (zip z (cdr lst1) (cdr lst2)))))
```

Program 16.4 *The function zip.*

Below we show examples of zipping with the `zip` function. For comparison, we also show an example that involves `string-merge`, which we discussed in Section 11.7.

Expression	**Value**
`(zip cons '(1 2 3) '(a b c))`	`((1 . a) (2 . b) (3 . c))`
`(apply string-append (zip string-append '("Rip" "Rap" "Rup") '(", " ", and " "")))`	`"Rip, Rap, and Rup"`
`(string-merge '("Rip" "Rap" "Rup") '(", " ", and "))`	`"Rip, Rap, and Rup"`

Table 16.3 *Examples of zipping.*

`Zip` is similar to the function `string-merge` from the LAML general library

However, `string-merge` handles lists of strings non-equal lengths, and it concatenates the zipped results

17. Currying

Currying is an idea, which is important in contemporary functional programming languages, such as Haskell. In Scheme, however, the idea is less attractive, due to the parenthesized notation of function calls.

Despite of this, we will discuss the idea of currying in Scheme via some higher-order functions like `curry` and `uncurry`. We will also study some ad hoc currying of Scheme functions, which has turned out to be useful for practical HTML authoring purposes, not least when we are dealing with tables.

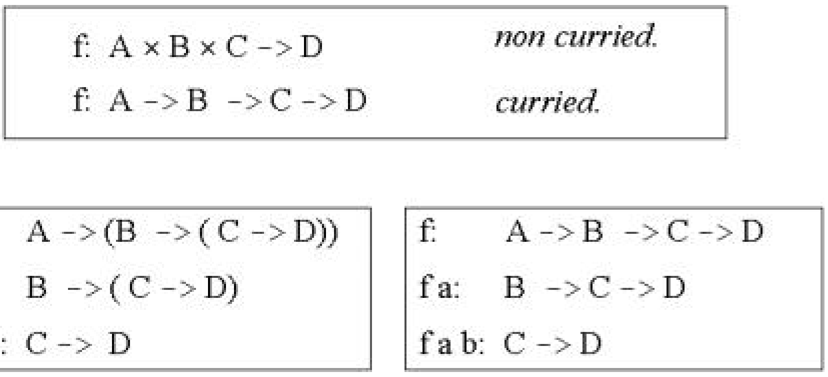
17.1. The idea of currying

Lecture 4 - slide 21

Currying is the idea of interpreting an arbitrary function to be of one parameter, which returns a possibly intermediate function, which can be used further on in a calculation.

Currying allows us to understand every function as taking at most one parameter. Currying can be seen as a way of generating intermediate functions which accept additional parameters to complete a calculation

The illustration below shows what happens to function signatures (parameter profiles) when we introduce currying.



f: A × B × C -> D *non curried.*
f: A -> B -> C -> D *curried.*

f: A -> (B -> (C -> D))
f a: B -> (C -> D)
f a b: C -> D

f: A -> B -> C -> D
f a: B -> C -> D
f a b: C -> D

Figure 17.1 *The signatures of curried functions. In the upper frame we show the signature of a function f, which takes three parameters. The frames below show the signature when f is curried. In the literature, the notation shown to the bottom right is most common. The frame to the left shows how to parse the notation (the symbol -> associates to the right).*

Currying and Scheme is not related to each other. Currying must be integrated at a more basic level to be elegant and useful

17.2. Currying in Scheme

Lecture 4 - slide 22

Despite the observations from above, we can explore and play with currying in Scheme. We will not, however, claim that it comes out as elegant as, for instance, in Haskell.

It is possible to generate curried functions in Scheme.

But the parenthesis notation of Lisp does not fit very well with the idea of currying

The function `curry2` generates a curried version of a function, which accepts two parameters. The curried version takes one parameter at a time. Similarly, `curry3` generates a curried version of a function that takes three parameters.

The functions `uncurry2` and `uncurry3` are the inverse functions.

It is worth a consideration if we can generalize `curry2` and `curry3` to a generation of `curry`*n* via a higher-order function `curry`, which takes *n* as parameter. We will leave that as an open question.

```
(define (curry2 f)
  (lambda(x)
    (lambda(y)
      (f x y))))

(define (curry3 f)
  (lambda(x)
    (lambda(y)
      (lambda(z)
       (f x y z)))))

(define (uncurry2 f)
  (lambda (x y)
    ((f x) y)))

(define (uncurry3 f)
  (lambda (x y z)
    (((f x) y) z)))
```

Program 17.1 *Generation of curried and uncurried functions in Scheme.*

Exercise 4.8. *Playing with curried functions in Scheme*

Try out the functions `curry2` and `curry3` on a number of different functions.

You can, for instance, use then curry functions on plus (+) and map.

Demonstrate, by a practical example, that the uncurry functions and the curry functions are inverse to each other.

17.3. Examples of currying

Lecture 4 - slide 23

Let us here show a couple of examples of the curry functions from Section 17.2.

Curried functions are very useful building blocks in the functional paradigm In particular, curried functions are adequate for mapping and filtering purposes

The function font-1 is assumed to take three parameters. The font size (an integer), a color (in some particular representation that we do not care about here) and a text string on which to apply the font information. We show a possible implementation of font-1 in terms of the font mirror function in Program 17.2.

Expression	**Value**
`(font-1 4 red "Large red text")`	Large red text

`(define curried-font-1 (curry3 font-1))`
`(define large-font (curried-font-1 5))`
`((large-font blue) "Very large blue text")`
	Very large blue text

`(define small-brown-font ((curried-font-1 2) brown))`
`(small-brown-font "Small brown text")`
	Small brown text

`(define large-green-font ((curried-font-1 5) green))`
`(list-to-string (map large-green-font (list "The" "End")) " ")`
	The End

Table 17.1 *Examples of currying in Scheme.*

```
(define (font-1 size color txt)
  (font 'size (as-string size)
        'color (rgb-color-encoding color)
        txt))
```

Program 17.2 *A possible implementation of font-1 in terms of the font HTML mirror function.*

17.4. Ad hoc currying in Scheme (1)

Lecture 4 - slide 24

In some situations we would wish that the `map` function, and similar functions, were curried in Scheme. But we cannot generate an *f*- mapper by evaluating the expression `(map f)`. We get an error message which tells us that `map` requires at least two parameters.

In this section we will remedy this problem by a pragmatic, ad hoc currying made via use of a simple higher-order function we call `curry-generalized`.

It is possible to achieve 'the currying effect' by generalizing functions, which requires two or more parameters, to only require a single parameter

In order to motivate ourselves, we will study a couple of attempts to apply a curried mapping function.

Expression	**Value**
`(map li (list "one" "two" "three"))`	`("<li>one</li>" "<li>two</li>" "<li>three</li>")`
`(define li-mapper (map li))`	*map: expects at least 2 arguments, given 1*
`(define li-mapper ((curry2 map) li))` `(li-mapper (list "one" "two" "three"))`	`("<li>one</li>" "<li>two</li>" "<li>three</li>")`

Table 17.2 *A legal mapping and an impossible attempt to curry the mapping function. The last example shows an application of curry2 to achieve the wanted effect, but as it appears, the solution is not very elegant.*

In Program 17.3 we program the function `curry-generalized`. It returns a function that generalizes the parameter `f`. If we pass a single parameter to the resulting function, the value of the red lambda expression is returned. If we pass more than one parameter to the resulting function, `f` is just applied in the normal way.

```
(define (curry-generalized f)
  (lambda rest
    (cond ((= (length rest) 1)
             (lambda lst (apply f (cons (car rest) lst))))
          ((>= (length rest) 2)
             (apply f (cons (car rest) (cdr rest)))))))
```

Program 17.3 *The function curry-generalized. This is a higher-order function which generalizes the function passed as parameter to* `curry-generalized`*. The generalization provides for just passing a single parameter to* `f`*, in the vein of currying.*

The blue expression aggregates the parameters - done in this way to be compatible with the inner parts of the red expression. In a simpler version `(cons (car rest) (cdr rest))` would be replace by `rest`.

In the next section we see an example of curry generalizing the `map` function.

17.5. Ad hoc currying in Scheme (2)

Lecture 4 - slide 25

We may now redefine `map` to `(curry-generalized map)`. However, we usually bind the curry generalized mapping function to another name, such as `gmap` (for generalized `map`).

This section shows an example, where we generate a `li` mapper, by `(gmap li)`.

Expression	Value

`(define gmap (curry-generalized map))`
`(define li-mapper (gmap li))`
`(li-mapper (list "one" "two" "three"))`
	`("<li>one</li>"`
	` "<li>two</li>"`
	` "<li>three</li>")`

`(gmap li (list "one" "two" "three"))`	`("<li>one</li>"`
	` "<li>two</li>"`
	` "<li>three</li>")`

Table 17.3 *Examples of curry generalization of map. Using `curry-generalized` it is possible to make a `li`-mapper in an elegant and satisfactory way. The last row in the table shows that `gmap` can be used instead of `map`. Thus, `gmap` can in all respect be a substitution for `map`, and we may chose to redefine the name `map` to the value of `(curry-generalized map)`.*

If we redefine map to `(curry-generalized map)`, the new mapping function can be used instead of the old one in all respects. In addition, `(map f)` now makes sense; `(map f)` returns a function, namely an *f mapper*. Thus `((map li) "one" "two" "three")` does also make sense, and it gives the result shown in one of value cells to the right of Table 17.3.

17.6. References

[-] Foldoc: curried function
http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?query=curried+function

18. Web related higher-order functions

We finish our coverage of higher-order functions with a number of examples from the web domain.

18.1. HTML mirror generation

Lecture 4 - slide 27

In this section we will, in a principled way, show how to generate simple HTML mirror functions in Scheme. Please notice that the HTML mirror functions in LAML are more sophisticated and elaborate than the ones discussed here.

There are three different cases to consider: double tag elements, single tag elements, and tags that can be both single and double.

A well-known tag, that can be both single and double is the p tag.

The higher-order functions `generate-double-tag-function` and `generate-single-tag-function` are the top level functions. They rely on a couple of other functions, which we program in Program 18.2 - Program 18.4.

```
(define (generate-double-tag-function tag-name)
  (lambda (contents . attributes)
    (double-tag tag-name contents attributes)))

(define (generate-single-tag-function tag-name)
  (lambda attributes
    (single-tag tag-name attributes)))
```

Program 18.1 *The two higher-order functions for the HTML mirror generation. This version corresponds to the an earlier version of LAML's HTML mirror.*

```
(define (single-tag name attributes)
 (start-tag name attributes))

(define (double-tag name contents attributes)
 (string-append (start-tag name attributes)
                (as-string contents)
                (end-tag name)))
```

Program 18.2 *Functions that generate single and double tags.*

The functions `start-tag` and `end-tag` are used in Program 18.2 and implemented in Program 18.3.

```
(define (start-tag kind attributes)
  (if (null? attributes)
      (string-append "<" kind ">")
      (let ((html-attributes (linearize-attributes attributes)))
         (string-append "<" kind " " html-attributes " >"))))

(define (end-tag kind)
  (string-append "</" kind ">"))
```

Program 18.3 *Functions that generate individual single and double tags.*

The missing aspect at this point is the attribute handling stuff. It is made in Program 18.4.

```
(define (linearize-attributes attr-list)
  (string-append
    (linearize-attributes-1
      (reverse attr-list) "" (length attr-list))))

(define (linearize-attributes-1 attr-list res-string lgt)
  (cond ((null? attr-list) res-string)
        ((>= lgt 2)
          (linearize-attributes-1
           (cddr attr-list)
           (string-append
            (linearize-attribute-pair
             (car attr-list) (cadr attr-list)) " " res-string)
           (- lgt 2)))
        ((< lgt 2)
          (error "The attribute list must have even length"))))

(define (linearize-attribute-pair val attr)
  (string-append (as-string attr)
                 " = " (string-it (as-string val))))
```

Program 18.4 *Functions for attribute linearization. The parameter attr-list is a property list.*

Recall that property lists, as passed to the function `linearize-attributes` in Program 18.4 have been discussed in Section 6.6.

There are several things to notice relative to LAML. First, the HTML mirror in LAML does not generate strings, but an internal representation akin to abstract syntax trees.

Second, the string concatenation done in Program 18.1 through Program 18.4, where a lot of small strings are aggregated, generates a lot of 'garbage strings'. The way this is handled (by the `render` functions in LAML) is more efficient, because we write string parts directly into a stream (or into a large, pre-allocated string).

You will find more details about LAML in Chapter 25 and subsequent chapters.

18.2. HTML mirror usage examples

Lecture 4 - slide 28

Let us now use the HTML mirror generation functions, which we prepared via `generate-double-tag-function` and `generate-single-tag-function` in Section 18.1.

The example assumes loading of `laml.scm` and the function `map-concat`, which concatenates the result of a map application.

The real mirrors use implicit (string) concatenation

As noticed above, there some differences between the real LAML mirror functions and the ones programmed in Section 18.1. The functions from above require string appending of such constituents as the three `tr` element instances in the table; This is inconvenient. Also, the mirror functions from above require that each double element gets exactly one content string followed by a number of attributes. The real LAML mirror functions accept pieces of contents and attributes in arbitrary order (thus, in some sense generalizing the XML conventions where the attributes come before the contents inside the start tag). Finally, there is no kind of contents nor attribute validation in the mirror functions from above. The LAML mirror functions validate both the contents and the attributes relative to the XML Document Type Definition (DTD).

Expression	**Value**

(let* ((functions
         (map generate-double-tag-
function
              (list "table" "td"
"tr")))
       (table (car functions))
       (td (cadr functions))
       (tr (caddr functions)))
 (table
  (string-append
   (tr
     (map-concat td (list "c1" "c2"
"c3"))
     'bgcolor "#ff0000")
   (tr
     (map-concat td (list "c4" "c5"
"c6")))
   (tr
     (map-concat td (list "c7" "c8"
"c9"))))
  'border 3))
	<table border="3">
	   <tr bgcolor="#ff0000">
	     <td> c1 </td>
	     <td> c2 </td>
	     <td> c3 </td>
	   </tr>
	   <tr>
	     <td> c4 </td>
	     <td> c5 </td>
	     <td> c6 </td>
	   </tr>
	   <tr>
	     <td> c7 </td>
	     <td> c8 </td>
	     <td> c9 </td>
	   </tr>
	</table>

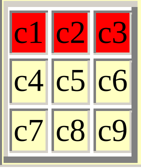
Same as above	 c1 c2 c3 c4 c5 c6 c7 c8 c9

Table 18.1 *An example usage of the simple HTML mirror which we programmed on the previous page. The bottom example shows, as in earlier similar tables, the HTML rendering of the constructed table. The* `map-concat` *function used in the example is defined in the general LAML library as* `(define (map-concat f lst) (apply string-append (map f lst)))`. *In order to actually evaluate the expression you should load* `laml.scm` *of the LAML distribution first.*

To show the differences between the simple mirror from Section 18.1 and the real mirror we will show the same example using the XHTML mirror functions in Section 18.3.

18.3. Making tables with the real mirror

Lecture 4 - slide 29

The real mirror provide for more elegance than the simple mirror illustrated above Here we will use the XHTML1.0 transitional mirror

In the example below there is no need to string append the tr forms, and there is no need to use a special string appending mapping function, like `map-concat` from Table 18.1. Attributes can appear before, within, or after the textual content. This makes the HTML mirror expression simpler and less clumsy. The rendering result is, however, the same.

Expression	**Rendered value**

`(table`
`  'border 3`
`   (tr`
`     (map td (list "c1" "c2" "c3"))`
`     'bgcolor "#ff0000")`
`   (tr`
`     (map td (list "c4" "c5" "c6")))`
`   (tr`
`     (map td (list "c7" "c8" "c9")))`
`)`
	`<table border = "3">`
	`  <tr bgcolor = "#ff0000">`
	`   <td>c1</td>`
	`   <td>c2</td>`
	`   <td>c3</td>`
	`  </tr>`
	`  <tr>`
	`   <td>c4</td>`
	`   <td>c5</td>`
	`   <td>c6</td>`
	`  </tr>`
	`  <tr>`
	`    <td>c7</td>`
	`    <td>c8</td>`
	`    <td>c9</td>`
	`  </tr>`
	`</table>`

Same as above	c1 c2 c3 c4 c5 c6 c7 c8 c9

Table 18.2 *A XHTML mirror expression with a table corresponding to the table shown on the previous page and the corresponding HTML fragment. Notice the absence of string concatenation. Also notice that the border attribute is given before the first tr element. The border attribute could as well appear after the tr elements, or in between them.*

You might think, that the example above also could be HTML4.01. But, not quite, in fact. In HTML4.01 there need to be a `tbody` (table body) form in between the `tr` instances and the `table` instance. Without this extract level, the table expression will not be valid. Try it yourself! It is easy.

[How, you may ask. In Emacs do `M-x set-interactive-laml-mirror-library` and enter `html-4.01`. Then do `M-x run-laml-interactively`. Copy the table expression from above, and try it out. You can shift to XHTML1.0 by `M-x set-interactive-laml-mirror-library` and asking for `xhtml-1.0-transitional`, for instance. Then redo `M-x run-laml-interactively`. Be sure to use `xml-render` on the result of `(table ...)` to make a textual rendering.]

18.4. Tables with higher-order functions

Lecture 4 - slide 30

In the context of higher-order functions there are even better ways to deal with tables than the one shown in Table 18.2 from Section 18.3.

The table expression in the last line in Table 18.3 shows how.

Instead of explicit composition of `td` and `tr` elements we can use a mapping to apply `tr` to rows and `td` to elements

Expression	**Value**

`(define rows`
`  '(("This" "is" "first" "row")`
`    ("This" "is" "second" "row")`
`    ("This" "is" "third" "row")`
`    ("This" "is" "fourth" "row"))`
`)`
 	
`(table 'border 5`
`    (gmap (compose tr (gmap td))`
`rows))`
	This | is | first | row
	This | is | second | row
	This | is | third | row
	This | is | fourth | row

The rendered value:

This	is	first	row
This	is	second	row
This	is	third	row
This	is	fourth	row

Table 18.3 *In the table expression we map - at the outer level - a composition of `tr` and a `td`-mapper. The `td`-mapper is made by `(gmap td)`.*

Recall that we already have discussed the ad hoc currying, which is involved in `gmap`, cf. the discussion in Section 17.4.

The last example illustrates that `(gmap td)` is a useful building block, which can be composed with other functions.

The last example depends on the fact that the HTML mirror functions accept lists of elements and attributes.

You should consult Chapter 26 to learn about the exact parameter passing rules of the HTML mirror functions in LAML.

18.5. HTML element modifications

Lecture 4 - slide 31

It is often useful in some context to bind an attribute of a HTML mirror function (or a number of attributes) to some fixed value(s). This can be done by the higher-order function `modify-element`, which we discuss below.

The idea behind the function `modify-element` is to perform an a priori binding of some attributes and some of the contents of a mirror function.

The function `modify-element` is simple. First notice that it accepts a function, namely the `element` parameter. It also returns a function; In effect, it returns `element` with `attributes-and-contents` appended to the parameters of the modified element. As another possibility, we could have prepended it.

```
(define (modify-element element . attributes-and-contents)
  (lambda parameters
   (apply element
    (append parameters attributes-and-contents))))
```

Program 18.5 *The function modify-element.*

In the table below we illustrate three examples where `td`, `ol`, and `ul` are modified with a priori bindings of selected attributes.

Expression	**Value**

`(define td1`
` (modify-element td`
`  'bgcolor (rgb-color-list red)`
`  'width 50))`
 	
`(table 'border 5`
`  (map (compose tr (gmap td1)) rows))`
	This | is | first | row
	This | is | second | row
	This | is | third | row
	This | is | fourth | row

`(define ol1`
`  (modify-element ol 'type "A"))`
 	
`(ol1`
` (map`
`   (compose li as-string)`
`   (number-interval 1 10)))`
	A. 1
	B. 2
	C. 3
	D. 4
	E. 5
	F. 6
	G. 7
	H. 8
	I. 9
	J. 10

`(define ul1`
`  (modify-element ul 'type "square"))`
 	
`(ul1`
` (map`
`  (compose li as-string)`
`  (number-interval 1 10)))`
	▪ 1
	▪ 2
	▪ 3
	▪ 4
	▪ 5
	▪ 6
	▪ 7
	▪ 8
	▪ 9
	▪ 10

Table 18.4 *Examples of element modification using the function* ***`modify-element`***.

LAML supports two related, but more advanced functions called `xml-in-laml-parametrization` and `xml-in-laml-abstraction`. The first of these is intended to transform an 'old style function' to a function with XML-in-LAML parameter conventions, as explained in Chapter 26. The second function is useful to generate functions with XML-in-LAML parameter conventions in general.

18.6. The function simple-html-table

Lecture 4 - slide 32

We will now show show an implementation of the function `simple-html-table`

In an earlier exercise - 'restructuring of lists' - we have used the function `simple-html-table`

We will now show how it can be implemented

```
(define simple-html-table
 (lambda (column-widht list-of-rows)
  (let ((gmap (curry-generalized map))
        (td-width
          (modify-element td 'width
                          (as-string column-widht))))
    (table
      'border 1
      (tbody
       (gmap (compose tr (gmap td-width)) list-of-rows))))))
```

Program 18.6 *The function simple-html-table. Locally we bind* `gmap` *to the curry generalized map function. We also create a specialized version of* `td`*, which includes a* `width` *attribute the value of which is passed as parameter to* `simple-html-table`*. In the body of the* `let` *construct we create the table in the same way as we have seen earlier in this lecture.*

18.7. The XHTML mirror in LAML

Lecture 4 - slide 33

In order to illustrate the data, on which the HTML mirrors in LAML rely, the web edition of the material includes a huge table with the content model and attribute details of each of the 77 XHTML1.0 strict elements.

LAML supports an exact mirror of the 77 XHTML1.0 strict elements as well as the other XHTML variants

The LAML HTML mirror libraries are based on a parsed representation of the HTML DTD (Document Type Definition). The table below is automatically generated from the same data structure.

The table is too large to be included in the paper version of the material. Please take a look in the corresponding part of the web material to consult the table.

18.8. Generation of a leq predicate from enumeration

Lecture 4 - slide 34

As the last example related to higher-order functions we show the function `generate-leq`, see Program 18.7.

The idea is to generate a boolean 'less than or equal' (leq) function based on an explicit enumeration order, which is given as input to the function `generate-leq`. A number of technicalities are involved. You should read the details in Program 18.7 to grasp these details.

In some contexts we wish to specify a number of clauses in an arbitrary order

For presentational clarity, we often want to ensure that the clauses are presented in a particular order

Here we want to generate a leq predicate from an enumeration of the desired order

```
;; Generate a less than or equal predicate from the
;; enumeration-order. If p is the generated predicate,
;; (p x y) is true if and only if (selector x) comes before
;; (or at the same position) as (selector y) in the
;; enumeration-order. Thus, (selector x) is assumed to give a
;; value in enumeration-order. Comparison with elements in the
;; enumeration-list is done with eq?
(define (generate-leq enumeration-order selector)
  (lambda (x y)
     ; x and y supposed to be elements in enumeration order
     (let ((x-index (list-index (selector x) enumeration-order))
           (y-index (list-index (selector y) enumeration-order)))
       (<= x-index y-index))))

; A helping function of generate-leq.
; Return the position of e in lst. First is 1
; compare with eq?
; if e is not member of lst return (+ 1 (length lst))
(define (list-index e lst)
 (cond ((null? lst) 1)
       ((eq? (car lst) e) 1)
       (else (+ 1 (list-index e (cdr lst))))))
```

Program 18.7 *The functions* ***`generate-leq`*** *and the helping function* ***`list-index`*** .

The table below shows a very simple example, in which we use `simple-leq?`, which is generated by the higher-order function `generate-leq` from Program 18.7.

Expression	Value
`(define simple-leq? (generate-leq '(z a c b y x) id-1))` `(sort-list '(a x y z c c b a) simple-leq?)`	`(z a a c c b y x)`

Table 18.5 *A simple example of an application of generate-leq.*

The fragment in Program 18.8 gives a more realistic example of the use of generated 'less than or equal' functions. In Program 18.9 we show how the desired sorting of `manual-page` subelements is achieved.

```
(manual-page
 (form '(show-table rows))
 (title "show-table")
 (description "Presents the table, in terms of rows")
 (parameter "row" "a list of elements")
 (pre-condition "Must be placed before the begin-notes clause")
 (misc "Internally, sets the variable lecture-id")
 (result "returns an HTML string")
)
```

Program 18.8 *A hypothetical manual page clause. Before we present the clauses of the manual page we want to ensure, that they appear in a particular order, say title, form, description, pre-condition, result, and misc. In this example we will illustrate how to obtain such an ordering in an elegant manner.*

```
(define (syntactic-form name)
  (lambda subclauses (cons name subclauses)))

(define form (syntactic-form 'form))
(define title (syntactic-form 'title))
(define description (syntactic-form 'description))
(define parameter (syntactic-form 'parameter))
(define pre-condition (syntactic-form 'pre-condition))
(define misc (syntactic-form 'misc))
(define result (syntactic-form 'result))

(define (manual-page . clauses)
 (let ((clause-leq?
         (generate-leq
           '(title form description
             pre-condition result misc)
           first))
      )
  (let ((sorted-clauses (sort-list clauses clause-leq?)))
    (present-clauses sorted-clauses))))
```

Program 18.9 *An application of* `generate-leq` *which sorts the manual clauses.*

18.9. References

[-] The XHTML1.0 frameset validating mirror
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/xml-in-laml/mirrors/man/xhtml10-frameset-mirror.html

[-] The XHTML1.0 transitional validating mirror
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/xml-in-laml/mirrors/man/xhtml10-transitional-mirror.html

[-] The XHTML1.0 strict validating mirror
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/xml-in-laml/mirrors/man/xhtml10-strict-mirror.html

[-] The HTML4.01 transitional validating mirror
http://www.cs.auc.dk/~normark/scheme/distribution/laml/lib/html4.01-transitional-validating/man/surface.html

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Lecture Notes: Feb. 12

Capturing Common Patterns with Higher-Order Procedures

Three procedures for computing sums

```
def sumint(low,high):
    s=0
    x=low
    while x <= high:
        s = s + x
        x = x + 1
    return s

def sumsquares(low,high):
    s=0
    x=low
    while x <= high:
        s = s + x**2
        x = x + 1
    return s
```

Approximation to $\pi^2/8$

```
def piSum(low,high):
    s=0
    x=low
    while x < high:
        s = s + 1.0/x**2
        x = x + 2
    return s
```

The general idea of summation, expressed as a procedure that captures the common pattern: $\sum_a^b f$:

```
def summation(low,high,f,next):
    s=0
    x=low
    while x <= high:
        s = s + f(x)
        x = next(x)
    return s
```

The sumint procedure, expressed as a general sum

```
def sumint(low,high):
    def identity(x): return x
    def add1(x): return x+1
    return summation(low,high,identity,add1)
```

The same three sums, expressed in terms of the general idea of summation, using `lambda` to avoid having to name the internal procedures:

```
def sumsquares(low,high):
    return summation(
         low,
         high,
         lambda x: x**2,
         lambda x: x+1
         )

def sumsquares(low,high):
    return summation(
         low,
         high,
         lambda x: x**2,
         lambda x: x+1
         )

def piSum(low,high):
    return summation(low,
         high,
         lambda x: 1.0/x**2,
         lambda x: x+2
         )
```

Expressing a general method of finding a fixed point of a function `f`:

```
def fixedPoint(f,firstGuess):
    def close(g1,g2):
        return abs(g1-g2)<.0001
    def iter(guess,next):
        while True:
            if close(guess, next):
                return next
            else:
                guess=next
                next=f(next)
    return iter(firstGuess,f(firstGuess))
```

Then we can compute square roots as fixed points:

```
def sqrt(x):
    def average(a,b): return (a+b)/2.0
    return fixedPoint(lambda g: average(g,x/g),1.0)
```

Four procedures for computing the sum of $f(x) = x\sqrt{x}$ for all the numbers in a list. They all do the same computation, but are expressed differently.

```
def sumf1(p):
    result = 0
    i = 0
    while i < len(p):
        result = result + p[i]*sqrt(p[i])
        i = i + 1
    return result
```

```
def sumf2(p):
    result = 0
    for x in p:
        result = result + x*sqrt(x)
    return result
```

```
def sumf3(p):
    return reduce(
        add,
        [x*sqrt(x) for x in p]
        )
```

```
def sumf4(p):
    return reduce(
        add,
        map(lambda x: x*sqrt(x),p)
        )
```

Computing derivatives: Given a function f, the derivative Df is another function. Therefore D *itself* is a function whose value is a function:

```
def deriv(f):
    dx=0.0001
    return lambda x:(f(x+dx)-f(x))/dx
```

We can write this equivalently, without using `lambda`:

```
def deriv(f):
    dx=0.0001
    def d(x):
        return (f(x+dx)-f(x))/dx
    return d
```

In either case, if we apply `deriv` to a procedure, the result is another procedure, that we can then apply to a number, e.g.,

```
>>> deriv(square)(10)
```

This returns 20 (approximately) because the derivative of $x \mapsto x^2$ is $x \mapsto 2x$.

Once we can express derivative, we can express Newton's method:

```
def newtonsMethod(f,firstGuess):
    return fixedPoint(
        lambda x: x - f(x)/deriv(f)(x),
        firstGuess)
```

and we can express computing square roots as an application of Newton's method:

```
def sqrt(x):
    return newtonsMethod(
        lambda y: y**2 - x,
        1.0)
```

The general method of iterative improvement, expressed as a procedure:

```
def iterativeImprove(goodEnough,improve,start):
     result = start
     while not goodEnough(result):
         result = improve(result)
     resturn result
```

Rights and privileges of first-class citizens in programming languages (Christopher Strachey)

- May be named by variables
- May be passed as arguments to procedures
- May be returned as results of procedures
- May be included in data structures

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Pset Addendeum for Week 2, Issued: Thursday, Feb. 14

On page 10, the definition of `selectAction` is missing its last line. It should be:

```
def selectAction(actionSet, default):
    actionList = [a for a in actionSet]
    if len(actionList) > 0:
        return actionList[0]
    else:
        print "No legal action!!"
        return default
```

Question 22 (Revised):
Write `prioritizedAndNDB`. Test it outside of soar, on the example above, and other examples of your construction. Here's how you might go about testing:

```
def b1(sensors):
    return set([1, 2, 3])
def b2(sensors):
    return set([2, 3, 4])
def b3(sensors):
    return set([1, 2])
def b4(sensors):
    return set([1, 4])
prioritizedAndNDB([b1, b2, b3, b4])
```

Hint: In thinking about how to write `prioritizedAndNDB` remember that in this week's tutor problems you wrote a procedure `fullintersection`. That doesn't solve the problem, but it could be a useful building block. Note that you can't just copy that code wholesale because it used a different representation of sets.

Python set type

There are some things to know and watch out for in Python's `set` type.

- To make a set from a list of items, do `set([1, 2, 3])`.
- The intersection of sets `a` and `b` is `a & b`.
- The union of sets `a` and `b` is `a | b`.
- The set difference between sets `a` and `b` is `a - b`.
- The number of elements in a set `a` is `len(a)`.
- To add an element `x` to set `s`, you can do `s.add(x)`. Be careful, though, because this changes `s`. To make a whole new set you can do `s | set([x])`.
- **Don't use `and` and `or` to operate on sets!!**. These are Boolean operators, and they treat 0, False, and None all as false values and everything else as true values. For clarity, you shouldn't apply them to anything but Booleans.

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Course notes for Week 3

Basic Object-Oriented Programming and State Machines

1 Introduction

Here is our familiar framework for thinking about primitives and means of combination, abstraction, and capturing common patterns. In this lecture, we'll add ideas for abstracting and capturing common patterns in data structures, and ultimately achieving even greater abstraction and modularity by abstracting over data structures combined with the methods that operate on them.

	Procedures	Data
Primitives	`+`, `*`, `==`	numbers, strings
Means of combination	`if`, `while`, `f(g(x))`	lists, dictionaries, objects
Means of abstraction	`def`	abstract data types, classes
Means of capturing common patterns	higher-order procedures	generic functions, classes

Let's start with an example. We've looked at three different implementations of sets:

- as lists;
- as functions; and
- as whatever Python uses in its built-in data type.

What is in common between these implementations is that they included the same set of abstract operations, but with different details about how the data was stored and how the various operations were computed. The good thing about thinking about sets in the abstract is that we can build something more complex (such as non-deterministic behaviors) on top of an implementation of set operations, without caring too much about how those set operations are ultimately implemented. We'll call such a thing an *abstract data type*. It's abstract, in that you can use it without knowing the implementation details. We used Python sets by reading their documentation, but without having any idea how they're actually implemented in Python.

Using ADTs is good because it

1. allows us to concentrate on the high-level aspects of the job we're trying to do; and
2. allows us or someone else to change the underlying implementation of sets (perhaps to make it faster or more memory-efficient) without having to change any of the code that we've implemented that uses sets.

An *abstract data type* (ADT) is a collection of related operations that can be performed on a set of data. The documentation of an ADT specifies its *contract*: a promised set of relationships among the inputs and outputs of the functions involved. For instance, part of the contract of a `set` ADT would be that the `union` of two sets, `s1` and `s2`, would `contain` all and only elements that are `contained` in `s1` or `s2`. ADTs are often used to describe generic structures like sets or dictionaries; they can also be used to describe more particular structures like bank accounts or telephone-directory entries.

A set ADT might include the following operations:

`makeSet`	:	*list* → *set*
`contains`	:	(*item*, *set*) → *Boolean*
`isSubset`	:	(*set*, *set*) → *Boolean*
`intersection`	:	(*set*, *set*) → *set*
`union`	:	(*set*, *set*) → *set*
`len`	:	*set* → *int*
`contents`	:	*set* → *list*

A bank-account ADT might include the following operations:

`makeBankAccount`	:	(*float*, *float*, *string*, *string*) → *account*
`balance`	:	*account* → *number*
`owner`	:	*account* → *string*
`creditLimit`	:	*account* → *number*
`deposit`	:	*account* → `None`

Operations like `makeSet` are often called *constructors*: they make a new instance of the ADT. Operatons like `balance` and `contents` are called *selectors*: they select out and return some piece of the data associated with the ADT. The `deposit` operation is special: it doesn't return a value, but it does *change* (sometimes we say *side-effect*) values stored inside the object (in this case, it will change the balance of the bank account, but we don't know exactly how it will do it, because we don't know how the balance is represented internally).

Different computer languages offer different degrees of support for using ADTs. Even in the most primitive language, you can write your code in a modular way, to try to preserve abstraction. But modern object-oriented languages offer built-in facilities to make this style easy to use.

2 Execution Model

This is repeated from section 3 of week 1 notes as a review; we will rely on it in the next section.

In order to really understand Python's object-oriented programming facilities, we have to start by understanding how it uses *environments* to store information, during and between procedure calls.

2.1 Environments

The first thing we have to understand is the idea of *binding environments* (we'll often just call them *environments*; they are also called *namespaces* and *scopes*). An environment is a stored mapping between names and entities in a program. The entities can be all kinds of things: numbers, strings, lists, procedures, objects, etc. In Python, the names are strings and environments are actually dictionaries, which we've already experimented with.

Environments are used to determine values associated with the names in your program. There are two operations you can do to an environment: add a binding, and look up a name. You do these things implcitly all the time in programs you write. Consider a file containing

```
a = 5
print a
```

The first statement, `a = 5`, creates a binding of the name `a` to the value 5. The second statement prints something. First, to decide that it needs to print, it looks up `print` and finds an associated built-in procedure. Then, to decide what to print, it evaluates the associated expression. In this case, the expression is a name, and it is evaluated by looking up the name in the environment and returning the value it is bound to (or generating an error if the name is not bound).

In Python, there are environments associated with each module (file) and one called `__builtin__` that has all the procedures that are built into Python. If you do

```
>>> import __builtin__
>>> dir(__builtin__)
```

you'll see a long list of names of things (like `'sum'`), which are built into Python, and whose names are defined in the builtin module. You don't have to type `import __builtin__`; as we'll see below, you always get access to those bindings. You can try importing `math` and looking to see what names are bound there.

Another operation that creates a new environment is a function call. In this example,

```
def f(x):
    print x
>>> f(7)
```

when the function `f` is called with argument 7, a new *local* environment is constructed, in which the name `x` is bound to the value 7.

So, what happens when Python actually tries to evaluate `print x`? It takes the symbol `x` and has to try to figure out what it means. It starts by looking in the *local* environment, which is the one defined by the innermost function call. So, in the case above, it would look it up and find the value 7 and return it.

Now, consider this case:

```
def f(a):
    def g(x):
        print x, a
        return x + a
    return g(7)
>>> f(6)
```

What happens when it's time to evaluate `print x, a`? First, we have to think of the environments. The first call, `f(6)` establishes an environment in which `a` is bound to 6. Then the call `g(7)` establishes another environment in which `x` is bound to 7. So, when needs to print `x` it looks in the local environment and finds that it has value 7. Now, it looks for `a`, but doesn't find it in the local environment. So, it looks to see if it has it available in an *enclosing environment*; an environment that was enclosing this procedure *when it was defined*. In this case, the environment associated with the call to `f` is enclosing, and it has a binding for `a`, so it prints `6` for `a`. So, what does `f(6)` return? 13.

You can think of every environment actually consisting of two things: (1) a dictionary that maps names to values and (2) an enclosing environment.

If you write a file that looks like this:

```
b = 3
def f(a):
    def g(x):
        print x, a, b
        return x + a + b
    return g(7)

>>> f(6)
7 6 3
16
```

When you evaluate `b`, it won't be able to find it in the local environment, or in an enclosing environment created by another procedure definition. So, it will look in the *global environment*. The name global is a little bit misleading; it means the environment associated with the file. So, it will find a binding for `b` there, and use the value 2.

One way to remember how Python looks up names is the LEGB rule: it looks, in order, in the *Local*, then the *Enclosing*, then the *Global*, then the *Builtin* environments, to find a value for a name. As soon as it succeeds in finding a binding, it returns that one.

Bindings are also created when you execute an `import` statement. If you execute

```
import math
```

Then the `math` module is loaded and the name `math` is bound, in the current environment, to the math module. No other names are added to the current environment, and if you want to refer to internal names in that module, you have to qualify them, as in `math.sqrt`. If you execute

```
from math import sqrt
```

then, the `math` module is loaded, and the name `sqrt` is bound, in the current environment, to whatever the the name `sqrt` is bound to in the `math` module. But note that if you do this, the name `math` isn't bound to anything, and you can't access any other procedures in the `math` module.

Another thing that creates a binding is the definition of a function: that creates a binding of the function's name, in the environment in which it was created, to the actual function.

Finally, bindings are created by `for` statements and list comprehensions; so, for example,

```
for element in listOfThings:
    print element
```

creates successive bindings for the name `element` to the elements in `listOfThings`.

Figure 2 shows the state of the environments when the print statement in the example code shown in figure 1 is executed. Names are first looked for in the local scope, then its enclosing scope (if there were more than one enclosing scope, it would continue looking up the chain of enclosing scopes), and then in the global scope.

```
a = 1
b = 2
c = 3
def d(a):
    c = 5
    from math import pi
    def e(x):
        for i in range(b):
            print a, b, c, x, i, pi, d, e
    e(100)

>>> d(1000)
1000 2 5 100 0 3.14159265359 <function d at 0x4e3f0> <function e at 0x4ecf0>
1000 2 5 100 1 3.14159265359 <function d at 0x4e3f0> <function e at 0x4ecf0>
```

Figure 1: Code and what it prints

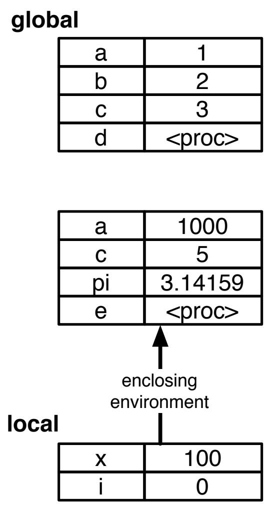


Figure 2: Binding environments that were in place when the first print statement in figure 1 was executed.

Local versus global references There is an important subtlety in the way names are handled in handled in the environment created by a procedure call. When a name that is not bound in the local environment is referred to, then it is looked up in the enclosing, global, and built-in environments. So, as we've seen, it is fine to have

```
a = 2
def b():
    print a
```

When a name is assigned in a local environment, a new binding is created for it. So, it is fine to have

```
a = 2
def b():
    a = 3
    c = 4
    print a, c
```

Both assignments cause new bindings to be made in the local environment, and it is those bindings that are used to supply values in the print statement.

But here is a code fragment that causes trouble:

```
a = 3
def b():
    a = a + 1
    print a
```

It seems completely reasonable, and you might expect it to print 4. But, instead, it generates an error if we try to call `b`. What's going on?? It all has to do with when Python decides to add a binding to the local environment. When it sees this procedure definition, it sees that the name `a` is assigned to, and so, at the very beginning, it puts an entry for `a` in the local environment. Now, when it's time to execute the statement

```
a = a + 1
```

it starts by evaluating the expression on the right hand side: `a + 1`. When it tries to look up the name `a` in the local environment, it finds that it has been added to the environment, but hasn't yet had a value specified. So it generates an error.

We can still write code to do what we intended (write a procedure that increments a number named in the global environment), by using the `global` declaration:

```
a = 3
def b():
    global a
    a = a + 1
    print a
>>> b()
4
>>> b()
5
```

The statement `global a` asks that a new binding for `a` *not* be made in the local environment. Now, all references to `a` are to the binding in the global environment, and it works as expected. In Python, we can only make assignments to names in the local scope or in the global scope, but not to names in an enclosing scope. So, for example,

```
def outer():
    def inner():
        a = a + 1
    a = 0
    inner()
```

In this example, we get an error, because Python has made a new binding for `a` in the environment for the call to `inner`. We'd really like for `inner` to be able to see and modify the `a` that belongs to the environment for `outer`, but there's no way to arrange this.

3 Object-oriented programming

Object-oriented programming (OOP, to its friends) helps us with a lot of aspects of abstraction in programming; this week, we'll just look at how it helps with constructing ADTs. The OOP facilities in Python are quite lightweight and flexible.

A *class* is a collection of procedures (and sometimes data) attached to names in an environment, which is meant to represent a generic *type* of object, like a set or a bank account. It typically contains the procedure definitions that allow it to fulfill the specifications of an ADT.

An *object* is also a collection of procedures and data, attached to names in an environment, but it is intended to represent a particular instance of a class, such as the set containing 1 and 2, or Leslie's bank account. When we want to make a particular bank account, we can create an object that is an *instance* of the bank account class. Instances (objects) are environments whose "enclosing" environment is the class of which they are instances. So, an object has access to all of the values defined in its class, but it can be specialized for the the particular instance it is intended to represent.

Here's a *very* simple class, and a little demonstration of how it can be used.

```
class SimpleThing:
    a = 6

>>> x = SimpleThing()
>>> x
<__main__.SimpleThing instance at 0x85468>
>>> x.a
6
>>> y = SimpleThing()
>>> y.a
6
>>> y.a = 10
>>> y.a
10
>>> x.a
6
>>> SimpleThing.a
6
```

To define a class, you start with a `class` statement, and then a set of indented assignments and definitions. Each assignment to a new name makes a new variable binding within the class. Whenever you define a class, you get a *constructor*, which will make a new instance of the class.

In our example above, the constructor is `SimpleThing()`.[1] When we make a new instance of `SimpleThing`, we get an object. We can look at the value of attribute `a` of the object `x` by writing `x.a`. An object is an environment, and this is the syntax for looking up name `a` in environment `x`. There is no binding for `a` in `x`, so it looks in its enclosing environment, which is the environment of the class `SimpleThing`, and finds a binding for `a` there, and returns 6.

If we make another instance, `y`, of the class, and assign a value to its attribute `a`, that makes a fresh binding for `a` in `y`, and doesn't change the original binding of `SimpleThing.a`.

Some of you may have experience with Java, which is much more rigid about what you can do with objects than Python is. In Python, you can add attributes to objects on the fly. So, we could continue the previous example with:

```
>>> x.newAttribute = "hi"
```

and there would be no problem.

Here's another example to illustrate the definition and use of *methods*, which are procedures whose first argument is the object, and that can be accessed via the object.

```
class Square:
    dim = 6

    def getArea (self):
        return self.dim * self.dim

    def setArea (self, area):
        self.dim = area**0.5
```

This class is meant to represent a square. Squares need to store, or remember, their dimension, so we make an attribute for it, and assign it initially to be 6 (we'll be smarter about this in the next example). Now, we define a method `getArea` that is intended to return the area of the square. There are a couple of interesting things going on here.

Like all methods, `getArea` has an argument, `self`, which will stand for the object that this method is supposed to operate on.[2] Now, remembering that objects are environments, the way we can find the dimension of the square is by looking up the name `dim` in this square's environment, which was passed into this method as the object `self`.

We define another method, `setArea`, which will set the area of the square to a given value. In order to change the square's area, we have to compute a new dimension and store it in the `dim` attribute of the square object.

Now, we can experiment with instances of class `Square`.

```
>>> s = Square()
>>> s.getArea()
36
>>> Square.getArea(s)
```

[1] A note on style. It is useful to adopt some conventions for naming things, just to help your programs be more readable. We've used the convention that variables and procedure names start with lower case letters and that class names start with upper case letters. And we try to be consistent about using something called "camel caps" for writing compound words, which is to write a compound name with the successiveWordsCapitalized. An alternative is_to_use_underscores.

[2] The argument doesn't have to be named `self`, but this is a standard convention.

```
36
>>> s.dim
6
>>> s.setArea(100)
>>> s.dim
10.0
```

We make a new instance using the constructor, and ask for its area by writing `s.getArea()`. This is the standard syntax for calling a method of an object, but it's a little bit confusing because its argument list doesn't really seem to match up with the method's definition (which had one argument). A style that is less convenient, but perhaps easier to understand, is this: `Square.getArea(s)`. Remembering that a class is also an environment, with a bunch of definitions in it, we can see that it starts with the class environment `Square` and looks up the name `getArea`. This gives us a procedure of one argument, as we defined it, and then we call that procedure on the object `s`. It is fine to use this syntax, if you prefer, but you'll probably find the `s.getArea()` version to be more convenient. One way to think of it is as asking the object `s` to perform its `getArea` method on itself.

Here's a version of the square class that has a special `initialization` method.

```
class Square1:
    def __init__(self, initialDim):
        self.dim = initialDim

    def getArea (self):
        return self.dim * self.dim

    def setArea (self, area):
        self.dim = area**0.5

    def __str__(self):
        return "Square of dim " + str(self.dim)
```

Whenever the constructor for a class is called, Python looks to see if there is a method called `__init__` and calls it, with the newly constructed object as the first argument and the rest of the arguments from the constructor added on. So, we could make two new squares by doing

```
>>> s1 = Square1(10)
>>> s1.dim
10
>>> s1.getArea()
100
>>> s2 = Square1(100)
>>> s2.getArea()
10000
>>> print s1
Square of dim 10
```

Now, instead of having an attribute `dim` defined at the class level, we create it inside the initialization method. The initialization method is a method like any other; it just has a special name. Note that it's crucial that we write `self.dim = initialDim` here, and not just `dim = initialDim`. All the usual rules about environments apply here. If we wrote `dim = initialDim`, it would make a variable in the method's local environment, called `dim`, but that variable would only exist during the execution of the `__init__` procedure. To make a new attribute of the object, it needs to be stored in the environment associated with the object, which we access through `self`.

Our class `Square1` has another special method, `__str__`. This is the method that Python calls on an object whenever it needs to find a printable name for it. By default, it prints something like `<__main__.Square1 instance at 0x830a8>`, but for debugging, that can be pretty uninformative. By defining a special version of that method for our class of objects, we can make it so when we try to print an instance of our class we get something like `Square of dim 10` instead. We've used the Python procedure `str` to get a string representation of the value of `self.dim`. You can call `str` on any entity in Python, and it will give you a more or less useful string representation of it. Of course, now, for `s1`, it would return `'Square of dim 10'`. Pretty cool.

Okay. Now we can go back to running the bank.

```
class Account:
    def __init__(self, initialBalance):
        self.currentBalance = initialBalance
    def balance(self):
        return self.currentBalance
    def deposit(self, amount):
        self.currentBalance = self.currentBalance + amount
    def creditLimit(self):
        return min(self.currentBalance * 0.5, 10000000)

>>> a = Account(100)
>>> b = Account(1000000)

>>> Account.balance(a)
100
>>> a.balance()
100
>>> Account.deposit(a, 100)
>>> a.deposit(100)
>>> a.balance()
300
>>> b.balance()
1000000
```

We've made an `Account` class that maintains a balance as state. There are methods for returning the balance, for making a deposit, and for returning the credit limit. These methods hide the details of the internal representation of the object entirely, and each object encapsulates the state it needs.

Here's the definition of a class representing the `set` ADT. We've called it `Sset` so it won't clash with Python's `set`.

```
class Sset:
    def __init__(self, items):
        self.contents = items

    def contains(self, x):
        return x in self.contents

    def add(self, x):
        self.contents.append(x)
    def intersection(self, otherSet):
        return [x for x in self.contents if otherSet.contains(x)]

    def union(self, otherSet):
        return self.contents + otherSet.elements()

    def elements(self):
        return self.contents

    def __str__(self):
        return str(self.contents)
    def __repr__(self):
        return str(self.contents)
```

Note that we've included both a `__str__` and a `__repr__` method. The Python documentation has this to say on the subject:

> "The str() function is meant to return representations of values which are fairly human-readable, while repr() is meant to generate representations which can be read by the interpreter (or will force a SyntaxError if there is not equivalent syntax). For objects which don't have a particular representation for human consumption, str() will return the same value as repr()."

When you do `print x`, then `str(x)` will be printed. When you just cause the Python shell to evaluate `x` and print the result, then it will print `repr(x)`. When in doubt, it's usually handy to define both.

Here are some examples of using `Sset`:

```
>>> s1 = Sset([1, 2, 3])
>>> s2 = Sset([3, 4, 5, 6, 7])
>>> s3 = Sset([4, 6, 8, 10])
>>> print s1.intersection(s2)
[3]
>>> print Sset.intersection(s1, s2)
[3]
>>> print s1.elements()
[1, 2, 3]
>>> s1.add(57)
>>> print s1.elements()
[1, 2, 3, 57]
```

4 State Machines

Both because they are a very important idea in electrical engineering, computer science, and a variety of other fields, and because they're a good domain for exercising OOP, we'll turn our attention to *state machines*.

A state machine (SM) is characterized by:

- a set of *states*, S,

- a set of *inputs*, I, also called the *input vocabulary*,
- a set of *outputs*, O, also called the *output vocabulary*,
- a *transition function*, t, that indicates, for every state and input pair, what the next state will be,
- an *output function*, o, that indicates, for every state, what output to produce in that state, and
- an *initial state*, s_0, that is the element of S that the machine starts in.

Together, these functions describe a discrete-time transition system, that can be thought of as performing a *transduction*: it takes in a (potentially infinite) sequence of inputs and generates a (potentially infinite) sequence of outputs.

Let's start by considering a simple example, where:

$$\begin{aligned}
S &= \textit{integers} \\
I &= \{u, d\} \\
O &= \textit{integers} \\
t(s,i) &= \begin{cases} s+1 & \text{if } i = u \\ s-1 & \text{if } i = d \end{cases} \\
o(s) &= s \\
s_0 &= 0
\end{aligned}$$

This machine can count up and down. It starts in state 0. Now, if it gets input u, it goes to state 1; if it gets u again, it goes to state 2. If it gets d, it goes back down to 1, and so on. In this case, the output is always the same as the state (because the output function o is the identity). If we were to feed it an input sequence of $u, u, u, d, d, u, u,$, it would generate the output sequence $0, 1, 2, 3, 2, 1, 2, 3$.

Delay An even simpler machine just takes the input and passes it through to the output. No state machine can be an instant pass-through, though, so the kth element of the input sequence will be the $k+1$st element of the output sequence. Here's the machine definition, formally:

$$\begin{aligned}
S &= \textit{anything} \\
I &= \textit{anything} \\
O &= \textit{anything} \\
t(s,i) &= i \\
o(s) &= s \\
s_0 &= 0
\end{aligned}$$

Given an input sequence $i_0, i_1, i_2, \ldots$, this machine will produce an output sequence $0, i_0, i_1, i_2, \ldots$. The initial 0 comes because it has to be able to produce an output before it has even seen an input, and that output is produced based on the initial state, which is 0. This very simple building block will come in handy for us later on.

Running Sum Here is a machine whose output is the sum of all the inputs it has ever seen.

$$\begin{aligned}
S &= \textit{numbers} \\
I &= \textit{numbers}
\end{aligned}$$

$$\begin{aligned} O &= \textit{numbers} \\ t(s,i) &= s+i \\ o(s) &= s \\ s_0 &= 0 \end{aligned}$$

Given input sequence $1, 5, 3$, it will generate an output sequence $0, 1, 6, 9$.

Language acceptor Here is a machine whose output is 1 if the input string adheres to a simple pattern, and 0 otherwise. In this case, the pattern has to be $a, b, c, a, b, c, a, b, c, \ldots$.

$$\begin{aligned} S &= \{0, 1, 2, 3\} \\ I &= \{a, b\} \\ O &= \{0, 1\} \\ t(s,i) &= \begin{cases} 1 & \text{if } s = 0, i = a \\ 2 & \text{if } s = 1, i = b \\ 0 & \text{if } s = 2, i = c \\ 3 & \text{otherwise} \end{cases} \\ o(s) &= \begin{cases} 0 & \text{if } s = 3 \\ 1 & \text{otherwise} \end{cases} \\ s_0 &= 0 \end{aligned}$$

Elevator As a final example, we'll make a state-machine model of a crippled elevator, which never actually changes floors. All we can ask the elevator to do is open or close its doors, or do nothing. So, the possible inputs to this machine are `commandOpen`, `commandClose`, and `noCommand`. The elevator doors don't open and close instantaneously, so we model the elevator as having four possible states: `opened`, `closing`, `closed`, and `opening`. These correspond to the doors being fully open, starting to close, being fully closed, and starting to open. Finally, the machine can generate three possible outputs, which give some useful information about the state of the elevator. If the doors are closed, the output is `sensorClosed`; if they are open, the output is `sensorOpened`; and otherwise the output is `noSensor`.

State machines with a finite (small) number of states are often diagrammed using *state diagrams*; figure 3 shows the transition and output functions for our elevator model. The circles represent states. The bold label in the circle is the state name; the other entry is the output from that state. The arcs indicate transitions. The labels on the arcs are one or more inputs that lead to those state transitions.

In the `closed` state, if the elevator is commanded to open, it goes into the `opening` state and the output is `noSensor`. In the `opening` state, the `commandOpen` or `noCommand` input causes a transition to the `opened` state and the output of `sensorOpened`. In the `opening` state, the `commandClose` input causes a transition to the `closing` state. The remaining transitions and outputs can be read from the state diagram.

The transition function for the machine should indicate what state the machine transitions to as a result of each of the inputs and so it captures the arcs of the machine. In general, we have to specify the effect of *every* input in *every* state. The output function is simpler; it just indicates the output for every state.

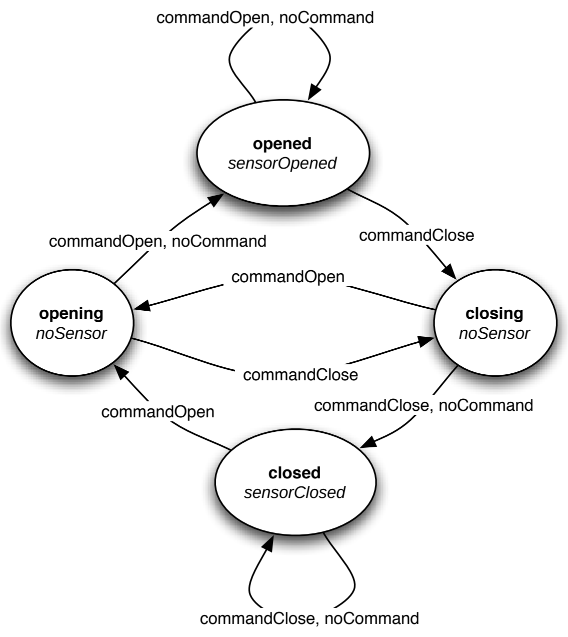


Figure 3: State diagram for a very simple elevator. Inspired by a figure from the Wikipedia article: Finite state machine

Here is the formal description of the elevator machine:

$$\begin{aligned}
\mathsf{S} &= \{\mathit{opened}, \mathit{closing}, \mathit{closed}, \mathit{opening}\} \\
\mathsf{I} &= \{\mathit{commandOpen}, \mathit{commandClose}, \mathit{noCommand}\} \\
\mathsf{O} &= \{\mathit{sensorOpened}, \mathit{sensorClosed}, \mathit{noSensor}\} \\
s_0 &= \mathit{closed} \\
t(s,i) &= \mathit{table1} \\
o(s) &= \mathit{table2}
\end{aligned}$$

The transition and output functions are most conveniently described using tables. The transition function is:

s	i	$t(s,i)$
opened	*commandOpen*	*opened*
opened	*noCommand*	*opened*
opened	*commandClose*	*closing*
closing	*commandOpen*	*opening*
closing	*noCommand*	*closed*
closing	*commandClose*	*closed*
closed	*commandOpen*	*opening*
closed	*noCommand*	*closed*
closed	*commandClose*	*closed*
opening	*commandOpen*	*opened*
opening	*noCommand*	*opened*
opening	*commandClose*	*closing*

The output function is:

s	o(s)
opened	*sensorOpened*
closing	*noSensor*
closed	*sensorClosed*
opening	*noSensor*

To help think about this machine, assume the elevator starts in the `closed` state, and try to predict the sequence of states and outputs that would result from the following sequence of inputs: *commandOpen*, *commandClose*, *noCommand*, *commandOpen*.

Composition Now we know how to define primitive state machines. Next week, we'll see how to apply our PCAP ideas to state machines, by developing a set of state-machine combinators, that will allow us to put primitive state machines together to make more complex machines.

Knuth on Elevator Controllers *Donald E. Knuth is a computer scientist who is famous for, among other things, his series of textbooks (as well as for TEX, the typesetting system we use to make all of our handouts), and a variety of other contributions to theoretical computer science.*

> "It is perhaps significant to note that although the author had used the elevator system for years and thought he knew it well, it wasn't until he attempted to write this section that he realized there were quite a few facts about the elevator's system of choosing directions that he did not know. He went back to experiment with the elevator six separate times, each time believing he had finally achieved a complete understanding of its *modus operandi.* (Now he is reluctant to ride it for fear some new facet of its operation will appear, contradicting the algorithms given.) We often fail to realize how little we know about a thing until we attempt to simulate it on a computer."
>
> *The Art of Computer Programming, Donald E., Knuth, Vol 1. page 295. On the elevator system in the Mathematics Building at Cal Tech. First published in 1968*

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment 3, Issued: Tuesday, February 19

To do this week

...before your lab on February 21 or 22

1. Read the notes on state machines (section 4 of the course notes).
2. Do the on-line tutor problems in section PS.4.2.
3. **Attend lecture: at one of the following places and times:**
   - **Thursday, Feb 21, 10AM - 11AM, in 32-123**
   - **Thursday, Feb 21, 2PM - 3PM, in 32-123**
   - **Friday, Feb 22, 10AM - 11AM, in 32-155**

...in your lab on February 21 or 22

1. **Hand in assignment 2 (software and design labs, stapled together) and the week 1 exploration, if you've done it (separately)**
2. Work through the numbered questions on the lab.

...before the start of your software lab on February 26 or 27

1. Do the on-line tutor problems in section PS.4.3.
2. Turn in written answers to all numbered questions in this lab.

Lab for Week 3

Do this lab on your own.

On the lab laptops, be sure you do:

```
athrun 6.01 update
```

so that you can get the `Desktop/6.01/lab3` directory which has the file `PrimitiveSM.py`.

The work for this week is a substantial problem set using simple object-oriented programming, with two parts: one on state machines and one on polynomials. During software lab, we want to get you started on each of these sections, so we'll ask you to spend an hour working on the first part of the state machine assignment, and then to switch and spend an hour working on the polynomial part. We don't expect you to finish either part during lab.

Simple State Machines

You have read about state machines in the course notes already. Now, it's time to build a state-machine class. The only operation we really need to do on a state machine is to ask it to take a step. Taking a step means that we give it an input; it makes a transition to a new state, depending on the input and the old state; and then it returns an output.

Here is the `PrimitiveSM` class. We call it primitive not because its knuckles drag on the ground, but because we'll be exploring ways to make composite SMs in the following week. The `__init__` method takes the transition and output functions as input. It also needs to know what state it should be in when it starts; for generality that we will take advantage of next week, we'll provide a function that can be called to generate an initial state. The types of the arguments to `__init__` are:

$$\begin{aligned} \texttt{transitionfn} &: (\textit{state}, \textit{input}) \rightarrow \textit{state} \\ \texttt{outputfn} &: \textit{state} \rightarrow \textit{output} \\ \texttt{initfn} &: () \rightarrow \textit{state} \end{aligned}$$

The return type is `PrimitiveSM`.

```
class PrimitiveSM:
    def __init__(self, transitionfn, outputfn, initfn):
        self.transitionFunction = transitionfn
        self.outputFunction = outputfn
        self.currentState = initfn()

    def step(self, input=None):
        self.currentState = self.transitionFunction(self.currentState, input)
        return self.currentOutput()

    def currentOutput(self):
        return self.outputFunction(self.currentState)
```

The `step` method takes an input for the state machine; but since we will sometimes be interested in state machines that don't have an input, we've supplied a *default* argument of `None`. This means that when we call it, `step` can either be given zero or one arguments; if it is given zero, then it will act as if it was given one argument, `None`.

Once we have this class definition, we can construct a simple state machine instance. We'll make `tickTock`, which alternates between outputs of `True` and `False`.

```
tickTock = PrimitiveSM(lambda s, i: not s,
                       lambda s: s,
                       lambda : True)
```

Now type

```
>>> tickTock.step()
```

a few times and see what happens.

To save typing, we've defined a procedure that will run a state machine with no input for some pre-specified number of steps:

```
def run(sm, n = 10):
    print sm.currentOutput()
    for i in range(n):
        print sm.step()
```

Try using this to run the `tickTock` machine. The argument `n = 10` means that you can specify a second argument to say how many steps the machine should be run, but if you don't it will default to 10.

Question 1: Make a new state machine that counts. Its first output should be 0, then 1, then 2, etc. It should ignore its input. Test it using `run`.

Question 2: Make a state machine that ignores its input and generates this sequence: $0, 1, 2, 3, 0, 1, 2, 3, \ldots$. Test it appropriately.

Question 3: Write a procedure that takes an input n and outputs a state machine that ignores its input and generates the sequence $0, 1, \ldots, n-1, 0, 1, \ldots, n-1, 0, \ldots$. Test it appropriately.

Question 4: Write a procedure `transduce` that takes as an argument a state machine and a list, and runs the state machine for a number of steps equal to the length of the list, feeding each element of the list in as input to the state machine, and returning the list of outputs. The first element of the list of outputs should be the output the machine generates based on the initial state (see how `run` calls `sm.currentOutput()` to get the first output). The length of the output list should be one more than the length of the input list (because we're interested in the initial output).

Question 5: Make a new state machine that simply delays its input by one time step. So, if you give it an input sequence of `[3, 4, 2, 5]`, its output sequence should be `[init, 3, 4, 2, 5]`, where `init = 0`. Test it with `transduce`.

Question 6: Write a procedure `makeDelayMachine` that takes a single argument `init`. It should return a state machine like the one you created above, but which outputs `init` as the first element.

Question 7: Write a procedure `makeDoubleDelayMachine` that takes two arguments, `init1` and `init2`. It should return a state machine that outputs `init1` as the first output, and then outputs `init2`, and then outputs its first input, and then its second, input, etc. So, for example, if you gave it an input sequence of `[3, 4, 2, 5]`, it should generate an output sequence of `[init1, init2, 3, 4, 2]`. (Hint: let the state be a list or tuple containing the two most recent inputs).

Question 8: Make a new state machine whose output is the average of all of the inputs it has seen so far. There are many ways to do this. Try to find a way that doesn't involve storing all the inputs you have ever seen in the state. Test it appropriately.

Question 9: A *set-reset* flip-flop is a simple finite-state component that is (or, at least was) used in the design and construction of computers. The machine has two states, 0 and 1. The output is the same as the state. It has two separate input variables, called S and R for *set* and *reset*, each of which can take on values 0 or 1. If the *set* input is equal to 1, then the flip-flop should change to state 1. If the *reset* input is equal to 1, then the flip-flop should change to state 0. If neither input is equal to 1, then it should stay in whatever state it's currently in. If both inputs are equal to 1, then you can do whatever you want to (it's supposed to be an illegal input condition). Let it start in state 0. Make and test an SR flip-flop.

Question 10: The Fibonacci numbers are the sequence $1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$. Each number in the sequence is the sum of the previous two numbers. Make a state machine that takes no input, but generates the Fibonacci numbers as output.

After an hour working on state machines, please switch to the next section, even if you're not done, so you can get familiar with both ideas while we're all in lab.

Polynomial Class

We have already seen that we can represent a polynomial as a list of coefficients starting with the highest-order term. For example, the polynomial $x^4 - 7x^3 + 10x^2 - 4x + 6$ would be represented as the list `[1, -7, 10, -4, 6]`. We have seen a couple of methods for evaluating a polynomial represented this way at a particular value for x. We are going to extend our ability to work with polynomials, by defining a polynomial class, and providing operations for performing algebraic operations on polynomials. It's good practice with object-oriented programming, and it will be useful to us in a couple of weeks.

Write a class called `Polynomial` that supports evaluation and addition on polynomials. You can structure it any way you'd like, but it should be correct and be beautiful. It should support, at least, an interaction like the following. We suggest converting all of the coefficients to floats, to forestall any future problems with integer division.

```
>>> p1 = Polynomial([3, 2, 1])
>>> print p1
poly[3.0, 2.0, 1.0]
```

The constructor accepts a list of coefficients, highest-order first, so this represents the polynomial $3x^2 + 2x + 1$.

```
>>> p2 = Polynomial([100, 200])
>>> print p1.add(p2)
poly[3.0, 102.0, 201.0]
>>> print p1 + p2
poly[3.0, 102.0, 201.0]
```

Use a `__str__` method to make the polynomials print out nicely (see the course notes for more information about this). Limit the precision with which the coefficients are printed out to make it more readable. You can, for example, convert a number `n` to a string with 4 digits after the decimal place, using scientific notation if necessary, using this expression:

```
"%.5g" % n
```

Question 11: Define the basic parts of your class, with an `__init__` method and a `__str__` method.

Question 12: Implement the `add` method for your class. **Be sure that performing the operation `p1 + p2` *does not change* the original value of `p1` or `p2`.**

A cool thing about Python is that you can *overload* the arithmetic operators. So, for example, if you add the following method to your `Polynomial` class

```
    def __add__(self, v):
        return self.add(v)
```

or, if you prefer,

```
    def __add__(self, v):
        return Polynomial.add(self, v)
```

then you can do

```
>>> print Polynomial([3, 2, 1]) + Polynomial([100, 200])
poly[3.0, 102.0, 201.0]
```

Exactly what gets printed as a result of this statement depends on how you've defined your `__str__` procedure; this is just an example. Now, you can also ask for the value of the polynomial at a particular `x`:

```
>>> p1.val(1)
6.0
>>> p1.val(10)
321.0
>>> (p1 + p2).val(10)
1521.0
```

Question 13: Add the `__add__` method to your class.

Question 14: Add the `val` method to your class, which evaluates the polynomial for the specified value. Use one of the rules we implemented in software lab 1.

Extend your `Polynomial` class to support multiplication.

```
>>> p1 = Polynomial([3, 2, 1])
>>> p2 = Polynomial([100, 200])
>>> print p1 * p1
poly[9.0, 12.0, 10.0, 4.0, 1.0]
>>> print p1 * p2
poly[300.0, 800.0, 500.0, 200.0]
>>> print (p1 * p1) + p1
poly[9.0, 12.0, 13.0, 6.0, 2.0]
```

The tricky part of the implementation is how to implement the multiplication of polynomials, including polynomials of different numbers of coefficients. Here are a few hints about polynomial multiplication. The length of the resulting polynomial is the sum of the lengths of the input polynomials minus 1. So, the product of two polynomials of length 3 is a polynomial of length 5. The key observation is that the $\mathtt{k}^{\mathtt{th}}$ coefficient is the result of adding up the products of all the pairs of coefficients whose indices add up to `k`. Be sure you understand the results in the example above.

As for beauty, try to do things as simply as possible. Don't do anything twice. If you need some extra procedures to help you do your work, you can put them in the same file as your class definition, but outside the class (so, put them at the end of the file, with no indentation). For instance, we found it useful to write helper functions to: add two lists of numbers of the same length, and to extend a list of numbers to a particular length by adding zeros at the front.

Question 15: Add the `mul` and `__mul__` methods to your `Polynomial` class. Show that they work, and that the results can be added and multiplied.

Now, add to your polynomial class the ability to find roots. Here's what we'd like you to do in the different cases:

- If the polynomial is linear, return the single real root.
- If it is quadratic, return both roots, whether or not they are complex.

- If it is higher order, then return one root, using Newton's method (with a complex guess, but returning a real number if the result is nearly real), as we did in software lab 2.

So, now, you might get something like:

```
>>> p1.roots()
[(-0.33333333333333331+0.47140452079103173j),
 (-0.33333333333333331-0.47140452079103173j)]
>>> p3 = Polynomial([3, 2, -1])
>>> p3.roots()
[0.33333333333333331, -1.0]
>>> (p1 * p2).roots()
Looking for a single root with Newton's method
-1.9999999999282596
```

Newton's method might fail to find a root; change `fixedPoint` so that it only runs for some maximum number of iterations, and then terminates even if it hasn't found a solution.

Question 16: Demonstrate your root finding for polynomials with 2, 3, and 4 coefficients. Show that, in the quadratic case, you can find both real and complex roots.

Writeup: Due at the beginning of your lab on February 26 or 27

Please hand in your solutions and test cases to all of the questions in this handout.

Increment, Twice

```
def makeIncr(init):
    return PrimitiveSM(lambda s, i: i+1,
                       lambda s: s,
                       lambda : init)

class Incr:
    def __init__(self, initialValue):
        self.value = initialValue
    def step(self, input):
        self.value = input + 1
        return self.currentOutput()
    def currentOutput(self):
        return self.value
```

In lecture, there was an inconsistency, which led me to say the transition function of the PrimitiveSM was wrong. In fact, the error was elsewhere, and is corrected by the red text

Generic Superclass

```
class SM:
    def run(self, n = 10):
        result = [self.currentOutput()]
        for i in range(10):
            result.append(self.step())
        return result

class Incr (SM):
    def __init__(self, initialValue): …
    def step(self, input=None): …
    def currentOutput(self): …
```

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment 4, Issued: Tuesday, Feb. 26

Overview of this week's work

In software lab

- Work through the software lab.
- Submit whatever work you have finished at the end of the lab into the tutor.

Before the start of your design lab on Feb 28 or 29

- Read the class notes and review the lecture handout.
- Do the on-line tutor problems in section PS.5.2.
- Read the entire description of the design lab, so that you will be ready to work on it when you get to lab.

In design lab

- Take the nanoquiz in the first 15 minutes; don't be late.
- Work through the design lab with a partner, and take good notes on the results of your work.

At the beginning of your next software lab on Mar 4 or 5

- Submit online tutor problems in section PS.5.3.
- Submit written solutions to software lab, and to all the design lab questions. All written work must conform to the homework guidelines on the web page.

- **You will need SoaR in this software lab, so if you don't have SoaR installed on your own machine, use a lab laptop or Athena station.**
- **Get the lab4 files via `athrun 6.01 update` or from the home page. We have given you several that have `Skeleton` in the name. You should make a copy of those files and rename them to remove the `Skeleton`. You will get failing `imports` otherwise.**

Software Lab: Combinations of state machines

In class we discussed three methods of making new state machines out of old ones: serial composition, parallel composition, and feedback composition. Below (and in the file `SimpleSMSkeleton.py`) is the skeleton of the `SerialSM` class. We've provided the constructor method, which take state machines as input and construct a new, composite state machine.

```
class SerialSM:
    def __init__(self, sm1, sm2):
        self.m1 = sm1
        self.m2 = sm2

    # Be careful to keep the timing consistent.  The input to m2 has
    # to be the current output of m1, not the output after it is
    # stepped.
    def step(self, input=None):
        # Your code here

    def currentOutput(self):
        # Your code here
```

Question 1: Write the `step` and `currentOutput` methods for a serial SM. Test it by combining two `incr` machines (the definition is in `SimpleSMSkeleton.py`). Draw a picture showing how the machines are connected. First, figure out what the result *ought* to be by filling out a table like this one (remember that, by definition, $output_1 = input_2$:

step	$input_1$	$state_1$	$output_1$	$state_2$	$output_2$
0					
1					
2					
3					

Then test to be sure your machine is doing the right thing.

Question 2: Compare the results of composing a `sum` machine (defined in `SimpleSMSkeleton.py`), which outputs the sum of all of its inputs so far with an `incr` machine. Predict what the result ought to be by filling out a table like this one:

step	input_1	state_1	output_1	state_2	output_2
0					
1					
2					
3					

Then be sure you're getting the right result.

Feedback The following class defines a feedback state machine. Its `__init__` method takes a state machine, `sm`, and returns a state machine with no inputs, which consists of `sm` with its output fed back to its input. We'll define the output of the feedback machine to be the same as the output of `sm`.

```
class FeedbackSM:
    def __init__(self, sm):
        self.m = sm

    def step(self, input=None):
        return self.m.step(self.m.currentOutput())

    def currentOutput(self):
        return self.m.currentOutput()
```

Now, we can use this to couple two machines together, with the outputs of one machine serving as the inputs to the other, and vice versa.

```
def simulatorSM(m1, m2):
    return FeedbackSM(SerialSM(m1, m2))
```

Question 3: In the code file, you'll find the definition of `makeIncr`, which takes an initial state as an argument, and then thereafter updates its state to be its input plus 1. If we make the following coupled machine, we get a potentially surprising string of outputs:

```
>>> fizz = simulatorSM(makeIncr(10), makeIncr(100))
>>> run(fizz)
100
11
102
13
104
15
106
17
108
19
110
```

Explain why this happens. Draw a picture of the objects involved in this machine and say exactly what state variables they contain. Fill in a table describing the inputs, states, and outputs of each component machine at each step.

Question 4: Imagine a robot driving straight toward a wall. We would like to use a state machine to model this system, where the state is the distance from the robot to the wall in meters (pick an initial distance for your initial state) and the input is the robot's current forward velocity in meters per second. Assume that each state transition corresponds to the passage of 0.2 seconds. Write a primitive state machine to model this system. Hint: the distance should go down as the machine steps.

Question 5: Now, let's think of the controller that this robot might be executing. Assume the robot would like to stop at some distance $\mathtt{d}_{Desired}$ from the wall. The controller can be modeled as a state machine that receives, as input, the current distance to the wall and generates, as output, a velocity. For now, assume there's no inertia involved, and so the controller can choose any new velocity it wants, independent of its previous velocity. In particular, let's assume it selects a new velocity that is equal to $\mathtt{k}(\mathtt{d} - \mathtt{d}_{Desired})$, where `d` is the current distance to the wall, $\mathtt{d}_{Desired}$ is the desired distance to the wall, and `k` is a "gain" constant. Write a primitive state machine to model this controller.

Question 6: Now, use the `simulatorSM` function to put these two machines together. Run the coupled machine. Try a large value of `k` and a small value of `k`. What happens? (By setting `SimpleSM.verbose` to `True`, you can get `PrimitiveSM.step` to print out information about each step of each primitive machine.)

Question 7: You can plot data from inside soar. If all you want to do is plot, the best thing is to make a brain with only a setup method, and put your plotting commands in there. Then, when you run the brain, you'll see your data plotted in a window. The `SimpleSMBrainSkeleton.py` file defines a `setup` function that will plot some data.

We have modified `run`, in the `SimpleSMSkeleton.py` file, so that it returns a list of all the outputs generated by the machine. So, you can just copy the `setup` and `step` definitions from `SimpleSMBrainSkeleton.py` file to the file with your state-machine definitions, call `run` inside of `setup` to get some data, and then plot it using `graphDiscrete`. If you select that file as a brain in soar, it will run your `setup` method.

Produce plots of the distance between the robot and the wall, over time, with two interestingly different values of `k`. Note that the graphing windows have a Save button that lets you save the graphs to a Postscript (.ps) file.

Now, check your understanding with the following question:

Question 8: Ralph Reticulatus correctly implements `SerialSM.step`, and then makes a common mistake, and tries to run the following code:

```
rm = makeSumSM()
ralph = SerialSM(rm, rm)
>>> transduce(ralph, [1]*10)
[0, 0, 2, 6, 14, 30, 62, 126, 254, 510, 1022]
```

(Note that this result depends on how, precisely, Ralph wrote his `SerialSM.step` method. There are multiple ways to write that method correctly that will result in different outputs in this case; but none will generate the desired sequence.)

Rita Rocksnake recognized Ralph's problem, and reminded him that if we were working with two bank accounts, he'd make two different instances of the `Account` class. Rita wrote this code, instead:

```
rita = SerialSM(makeSumSM(),makeSumSM())
transduce(rita, [1]*10)
[0, 0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

She got the intended result. Explain why. Draw a picture of all of the objects in both Ralph and Rita's attempts, and say exactly what say what state variables they contain.

Software Exploration: New primitive state machines

This is worth 2 out of 10 points for the Week 4 exploration.

In the basic work of this lab, we approached the serial, parallel, and feedback composition of machines by defining new classes of state machines, in such a way that, for example, an instance of a serial composition state machine contained instances of the two original state machines, asked them to store their state internally, and simply asked them to step when necessary.

That approach works well when the goal is only to be able to run the composite state machines. However, later in the course, we will find that state machines can be used as models of an environment, for example, to plan a sequence of actions by trying out different action sequences in the model and seeing which ones perform the best. In order to do that, we need to know the transition function for the composite state machine as a single function.

So, in this exploration, we will pursue another strategy for composing state machines, in which we provide a function `makeSerialSM` that takes as input two *primitive* state machines and returns another *primitive* state machine. That means, it must construct new transition, output, and initialization functions that describe the operation of the entire serial state machine, and use them as arguments to the constructor of the `primitiveSM` class. To write such a function, you have to start by thinking carefully about what the state of the composite machine will be.

Exploration 1: Write the function `makeSerialSM` of type

$$(\mathit{primitiveSM}, \mathit{primitiveSM}) \rightarrow \mathit{primitiveSM}$$

Exploration 2: Write the function `makeParallelSM` of type

$$(\mathit{primitiveSM}, \mathit{primitiveSM}) \rightarrow \mathit{primitiveSM}$$

Exploration 3: Write the function `makeFeedbackSM` of type

$$(\mathit{primitiveSM}, \mathit{input}) \rightarrow \mathit{primitiveSM}$$

Exploration 4: Demonstrate these composition methods using various primitive machines that you constructed in the earlier parts of the lab.

Design Lab: Combinations of terminating behaviors

In lab 2, we explored a system for making primitive behaviors and combining them. In that case, our behaviors were all operating at once, and their outputs were expressed as sets of actions. It's possible to do a lot of different things this way, but many people are frustrated by the inability to ask their robot to do things sequentially, like drive up to the wall and then turn left. In this lab, we'll develop a new system of behavior definition and combination, which is yet another instance of the PCAP idea, that allows sequential combination of behaviors.

Terminating state machines

So far, the state machines that we have looked at all run forever, performing a transduction from a potentially infinite stream of inputs to a potentially infinite stream of outputs. Sometimes our work feels like that too! But sometimes, jobs actually get done, and we move on to new tasks.

Our first step will be to build on our original state-machine abstract data type, to construct a *terminating state machine* ADT. In addition to providing the `step` and `currentOutput` methods that regular SM's provided, terminating state machines have to provide two more methods:

- `done`: () → `Boolean`, which is `True` when the machine has terminated; and
- `reset`: () → (), which resets the machine back to its original initial state.

In other words, a terminating state machine is one where we can ask if it is "done." A machine is said to have *terminated* when its `done` method returns `True`. The way the machine decides if it has terminated is by computing a function of the machine state, called `terminationFunction`.

We can take advantage of the object-oriented programming facility of *inheritance* to build terminating state machines. Rather than redefining a whole new class of primitive terminating state machines, we can make it be a subclass of the `PrimitiveSM` class. Here's how it goes:

```
class PrimitiveTSM(PrimitiveSM):
    def __init__(self, transitionfn, outputfn, initStatefn,
                 terminationfn = lambda x: False):
        self.terminationFunction = terminationfn
        self.initialStateFunction = initStatefn
        PrimitiveSM.__init__(self, transitionfn, outputfn, initStatefn)

    def done(self):
        return self.terminationFunction(self.currentState)

    def reset(self):
        self.currentState = self.initialStateFunction()
```

Here, we had to redefine the `__init__` method to do some additional work. It has a new argument, `terminationfn`, which defaults to be a function that always returns `False`; so, the first thing it does is to store that function. Then, it also stores the `initStatefn` argument, because it will need it to implement the `reset` method. Finally, it calls the `__init__` method of the `PrimitiveSM` class to do the rest of the work.

We don't need to change the `step` or `currentOutput` methods from the way they work on `PrimitiveSM`s, so we inherit them from `PrimitiveSM` and don't need to say anything about them.

Then, we define the `done` method, which simply calls the `terminationFunction` function on `self.currentState`. Who defined `self.currentState`? The `__init__` method of `PrimitiveSM`.

And we define the `reset` method to reset the current state to whatever value is returned by `self.initialStateFunction`.

Sequential combination

Now, we'd like to implement some means of combining these machines so we can make more complex overall sequential structures. How can we make this happen?

We'd like to define a new class of terminating state machines, called `SequentialTSM`, that takes as an initialization argument a list of terminating state machines. Running the sequence amounts to running the first machine until it terminates, then the second machine until it terminates, and so on.

To do this the sequence machine keeps track of which component machine it is currently running, and keeps calling that machine's associated `step` function until it has terminated; then starts running the next machine in the sequence, and so on. The overall sequence machine terminates on the step after the last component machine terminates.

The code below contains all but the `step` and `done` methods for the class, which you should complete. Our `reset` method actually sets the initial state, so be sure to reset any component SM before beginning to step it.

```
class SequentialTSM():
    def __init__(self, smList):
        self.smList = smList
        self.reset()

    def reset(self):
        self.counter = 0
        self.smList[self.counter].reset()

    def currentOutput(self):
        if not self.done():
            return self.smList[self.counter].currentOutput()
```

Here are some simple state machines and combinations to use when you're testing your work, below.

```
def makeCharTSM(c):
    return PrimitiveTSM(lambda s, i: True,      # transition
                        lambda s: c,            # output
                        lambda: False,          # initial state
                        lambda s: s)            # done
def makeTextSequenceTSM(str):
    return SequentialTSM([makeCharTSM(c) for c in str])
```

The `makeCharTSM` procedure takes a character `c` and returns a terminating state machine whose state is either `True` or `False`. If the state is `True`, then the machine is done. The initial state is `False`, and one step will change the state to `True`. The output is the character `c`. So, this machine runs for one step and generates the output sequence [`c`].

The `makeTextSequenceTSM` procedure takes a string as an argument and returns a `SequentialTSM` that is made up of individual primitive TSMs that output a single character. So, if you `run` the machine `makeTextSequenceTSM('abcd')`, the output sequence should be `['a', 'b', 'c', 'd']`. (Note that we have provided a new version of `run`, that runs a machine until it terminates (or until a limit is reached, whichever comes first).

Use these procedures to test your `SequentialTSM` and `RepeatTSM` classes, below.

Question 9: Write the `step` and `done` methods of the `SequentialTSM` class. Be sure that each call to `step` of the sequential machine results in exactly one call to `step` of some component machine. The sequential machine should terminate one step after the last of its component machines does (that is, if the `done` method of the last component machine first returns `True` on step n, then the sequential machine should first return `True` on step $n + 1$).

Question 10: Sometimes we'd just like to repeat the same thing over and over again. Define a new class `RepeatTSM`, whose constructor takes a terminating state machine and a positive integer `k`, and executes that state machine until done, `k` times. Provide all of the methods required of a terminating state machine.

Checkpoint 1: To be completed by 45 minutes after the beginning of lab

Terminating behaviors

We can use our new sequencing abilities to get the robot to perform behaviors in sequence. We'll concentrate, this week, on building deterministic behaviors that specify completely how the robot is to be controlled; but we'll combine them by building sequential combinations of them.

A *terminating behavior* is a TSM that transduces a sequence of sensor inputs (each of which is our usual list of sonar readings and a pose) into a sequence of actions. Here's the basic brain structure for using a terminating behavior.

```
def setup():
    robot.behavior = yourTSMHere
    robot.behavior.reset()

def step():
    if robot.behavior.done():
        doAction(stop)
    else:
        doAction(robot.behavior.step(collectSensors()))
```

If you defined a new terminating behavior class, like `DoTheMacarena`, then you'd have `DoTheMacarena()` in the code above in place of `yourTSMHere`.

In the `setup` function, we make whatever calls are necessary to construct the terminating state machine that will generate the robot's behavior. We store that machine in the variable `robot.behavior`. You can think of `robot` as an object that will persist across invocations of `setup` and `step` in the robot brain, and will let us store the state of our behavior (which is stored in the state of a TSM).

In the `step` function, we check to see if the behavior is done, and, if so, we stop. Otherwise, we collect a fresh set of sensor values, feed them as input into the behavior machine's `step` method, get an action as an output from that machine, and issue the associated command.

Notice that the brain and the behavior each have their own `step` method, and that these are different. This is an example of generic functions, implemented with the aid of object-oriented programming. In this implementation, the brain's `step` method calls the behavior's `step` method as long as the behavior is not `done`.[1]

[1] Are you wondering why we refer to the brain's `step` as a *method*, when you don't see any class definition here?

Let's start by building some basic terminating behaviors for the robot. Here's a somewhat stupid[2] behavior, to illustrate how it might work. Imagine that we wanted to define a new primitive terminating behavior object, called `GoForwardUntilDeltaX`, which moves the robot forward until its x coordinate has increased by some value `deltaX`. We might write code like this:

```
class GoForwardUntilDeltaX:
    def __init__ (self, deltaX):
        self.deltaX = deltaX

    def reset(self):
        (sonars, pose) = collectSensors()
        self.initialX = pose[0]
        self.currentX = pose[0]

    def step(self, sensors):
        (sonars, pose) = sensors
        self.currentX = pose[0]
        return self.currentOutput()

    def self.currentOutput():
        if self.done():
            return stop
        else:
            return go

    def done(self):
        return self.currentX > self.initialX + self.deltaX
```

The initializer for the class just remembers the delta value. When the behavior is reset (which must be done before it is run for the first time), it looks at the robot's current pose, and the initial x coordinate, both as `self.initialX`, which will remain the same until the behavior is reset again, and as `self.currentX`, which will be updated on each step. On each step, the behavior selects the x coordinate of the pose from the sensor readings and remembers it. Then it generates an appropriate output, by calling its own `currentOutput` method. This is a behavior, so the output should be an action. The `currentOutput` method checks to see if the behavior is done; if so, it returns `stop` and otherwise it returns `go`. The `done` method returns the value `True` when the x coordinate of the current pose is greater than the x coordinate of the initial pose by at least `self.deltaX`.

Implement the following primitive terminating behaviors and test them by themselves using the brain code found in `TSMBrainSkeleton.py`. That file also contains some helper functions that you might find useful.

When you're testing terminating behaviors, you'll need to reload the brain inside SoaR once a behavior has terminated. Why? Because the program has to be run again from the beginning; if it just continues from where it was, the behavior might still think it's done.

There really is a `Brain` class with `step` and `setup` methods, but the class definition is buried inside the implementation of SoaR. There's also a `robot` object that the brain uses in order to store attributes. That's what lets the `setup` procedure store something as `robot.behavior`, which can be referenced by the brain's `step`. You can use `robot` for storing anything that you like, and it will be local to the particular instance of the brain that SoaR is running.

[2] Why stupid? Because it's hard to imagine why it would be useful to move in whatever direction the robot is pointed until its global x coordinate has increased by some amount. It's entirely possible that, given the heading of the robot, x will decrease or remain unchanged as the robot moves forward.

Question 11: Define a class `TBDrive`, with an `__init__` method that takes as an argument a distance `d` to move. It should move forward a fixed distance, `d`, from the robot's position at the time the `reset` method is called, along the current heading, and then terminate. The class should be a terminating behavior in the sense that it provides `step`, `currentOutput`, `reset`, and `done` methods. *Caveat:* Remember that the robot might be moving at an arbitrary angle with respect to its global coordinate system.

Question 12: Define a class `TBTurn`, with an `__init__` method that takes as an argument an angle `a` to rotate through. It should be a terminating behavior, as above. It should rotate `a` radians from the robot's heading at the time the `reset` method is called, and then terminate. *Caveat:* Remember that `pose()` returns an angle between 0 and 2π, so finding the difference between two angles is not trivial. We have provided a helper in `TSMBrainSkeleton.py` function you may find useful.

Question 13: Define a class `TBForwardUntilBlocked` that is a terminating behavior that moves forward until it is blocked in front, and then terminates.

Question 14: Test these behaviors in the simulator, and take note of their performance.

Think carefully about an appropriate termination condition to use in each case. **Don't start implementing these until you've talked with your LA about your plan!**

Checkpoint 2: To be completed by 1.5 hours after the beginning of lab

Hip to be Square

Question 15: Now, use sequencing and your new primitives to make the actual robot drive in a one meter square. Explain how your program works. Draw a diagram of the different class instances involved.

Question 16: Execute your program for driving in a square repeatedly. Describe your results. Think about a simple way of measuring and describing the accuracy of your program running on the robot.

Checkpoint 3: To be completed by 2 hours after the beginning of lab

Safety

We have now constructed a useful framework for building sequential behaviors on top of a substrate that still affords frequent reading of and reaction to sensor readings. This means that, while carrying out a sequence of behaviors, the robot can react to surprises. For now, we'll show how it can change its low-level behavior in reaction to sensor values, without changing the actual sequence of high-level steps it is taking. Later in the class, we'll develop a stream-based method of sequential programming that affords a great deal of flexibility in the high-level sequencing, as well.

Question 17: Write a class `TBDriveSafely` that has the basic `TBDrive` terminating behavior as a superclass, and that overrides one of the superclass methods so that the robot will sit and wait rather than run into an obstacle. Note that it shouldn't terminate when it's blocked. Test this on the robot and describe your experiments.

Checkpoint 4: To be completed by 2.5 hours after the beginning of lab

More means of combination

Implement one more means of combination for terminating behaviors. Here are some that we have thought of:

- **TBIf**: take a condition and two terminating behaviors. Test the condition when this behavior is reset (not created!), and then execute the first behavior if the condition was true and the second if it was false.
- **TBWhile**: take a condition and a terminating behavior. When the behavior is reset, evaluate the condition. If it's false, terminate. If not, execute the terminating behavior, then test the condition again, etc.
- **TBParallelFun**: take a terminating behavior and a function, and call the function on every step of the terminating behavior.

Question 18: Use all of your means of combination to make an interesting robot behavior (the robot macarena, for example). Run this on the robot and describe the results.

Checkpoint 5: To be completed by the end of lab

Concepts covered in this lab

- Encapsulation of state into objects can provide useful abstraction.
- There are many different frameworks for abstraction and combination, and it's important to choose or design one suited to your problem.
- It can be hard to get repeatable behavior from a physical device.

Homework due in your lab on March 4 or 5

1. Log in to the online tutor and complete the second half of this week's tutor problems.
2. Hand in your code and test cases from the software lab.
3. Write up and hand in the answers to all the numbered question in this handout, including the following thought questions:

Question 19: Suggest how you could arrange it so that an FSM could actually produce outputs in the world, for example, send commands to a robot, or actually run the motor on an elevator door.

Question 20: How reliable was your program to drive in a square? What do you think was the main contributing source to the errors? What are some strategies for reducing the error?

Question 21: We have now seen two different frameworks for making primitive robot behaviors and combining them: the parallel (utility-based) and sequential (terminating behaviors) approaches. Consider a robot for operating in a household, doing chores. Give examples of situations in which each type of behavior combination would be appropriate.

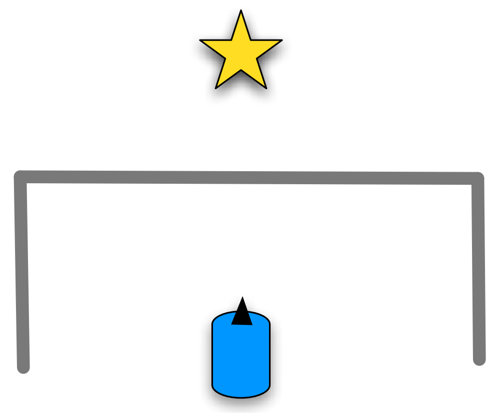


Figure 1: Robot in a cul-de-sac.

Exploration: Bug Algorithm

This is worth 8 out of 10 points for the Week 4 exploration.

You should have found that your program for moving forward is of of limited utility in the presence of obstacles. That is, that your robot could find an obstacle directly between itself and the goal that it is trying to reach. Some kinds of blockages can be escaped by simple behaviors (consider the goal as an attractive force and the sonar measurements as repulsive forces, add up all the forces and move along the resultant force), but a robot trapped in a cul-de-sac, as shown in figure 1, will have to be smarter to reach its goal.

One effective method for getting out of such jams is the “Bug 2” algorithm[3]. The idea is illustrated in figure 2. Here’s the algorithm:

- Compute the line L from start to goal
- Follow that line until an obstacle is encountered
- Follow the obstacle (in either direction) until you encounter the L line again, *closer to the goal*
- Leave the obstacle and continue toward the goal

Note that in the Bug2 algorithm the robot’s motion is not just a function of the sensors, it depends on the line L which was defined when the robot started. So, it requires remembering values between invocations of the `step` procedure. If you want to use a variable to store a value between invocations of the `step` procedure, you can extend your brain to something like this:

```
#code that is called once at startup
def setup():
    print "start"
    robot.x = 0

#code that is called about 10 times a second
def step():
    robot.x = robot.x + 1
    print "Step ", robot.x
```

[3]It is often better, but sometimes much worse than another algorithm called “Bug 1”. For more info see `http://www.cs.jhu.edu/~hager/Teaching/cs336/Notes/Chap2-Bug-Alg.pdf`

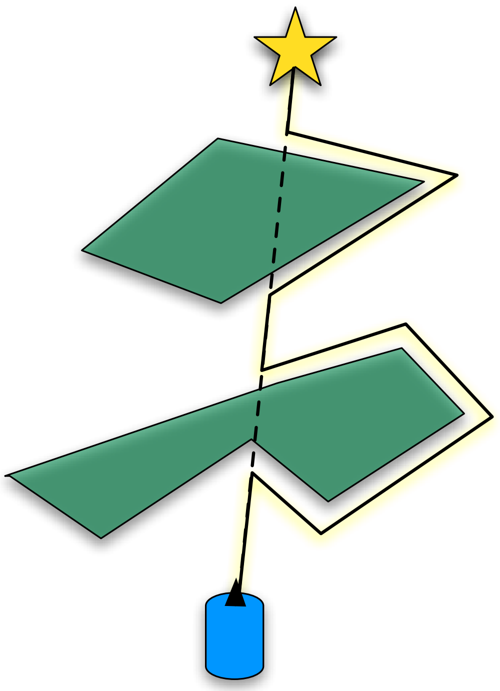


Figure 2: The Bug2 algorithm.

When implementing Bug2, you can define the line L in the `setup` procedure and then the basic behavior in the `step` procedure. In later weeks, we will develop more elaborate algorithms that depend on *state*, which encapsulates the robot's history.

A comment on representing lines. In high school, they probably taught you to represent the line between two points (x_1, y_1) and (x_2, y_2) in the form $y = mx + b$ where m is the slope $y_2 - y_1/x_2 - x_1$ and b is the y-intercept (what's the expression for that?). You can use m and b to represent the line L, you can test whether an (x, y) position is on the line by checking whether it (nearly) satisfies[4] the equation for the line. You should note that the $y = mx + b$ is not really a very good representation for lines; it cannot represent vertical lines, since their slope is infinite. A better representation is $ax + by + c = 0$. You can stick with $y = mx + b$ for now, but you should check for the vertical case and possibly change one of the endpoints a little bit.

The bug algorithm has some state in it. Think about how you could use the terminating sequential behavior framework (or state machines, or some other systematic way of using state) to structure your algorithm.

To simplify testing and debugging in the simulator, we recommend adding this definition to your file

```
def cheatPose():
    return app.soar.output.abspose.get()
```

and using `cheatPose()` instead of `pose()` to determine where the robot is. If you do this, then when you drag the robot around the simulator window with the mouse, it will magically know the

[4]Even in the simulator you cannot expect that the position of the robot will fall exactly on the line.

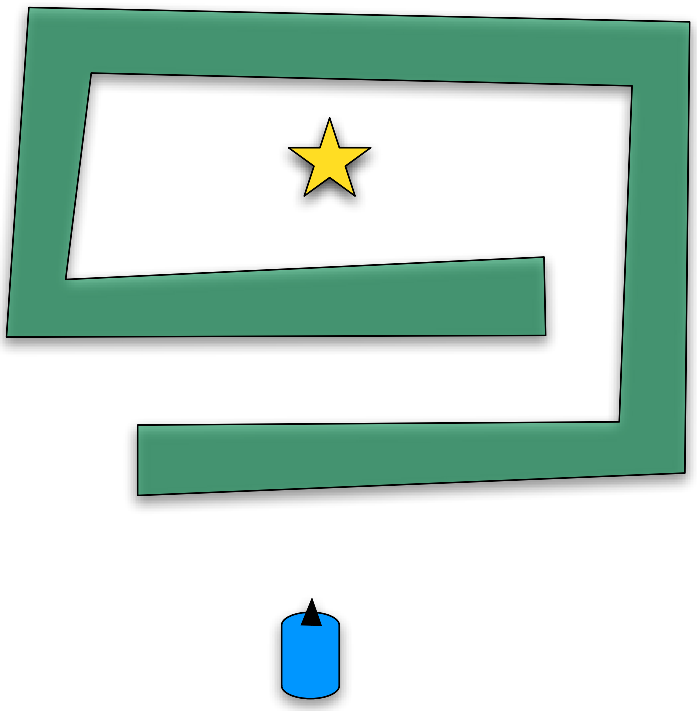


Figure 3: What would Bug2 do in this domain without the *closer to goal* test?

true pose. (On the real robot, of course, if you were to pick it up and put it down, it would have no idea that it had moved.)

Exploration 5: Implement Bug2 in the SoaR simulator. Test it on the `BugTest` worlds in the `ps4` distribution.

Exploration 6: Explain why Bug2 would be hard to implement on the real robots.

Exploration 7: If we hadn't added the *closer to the goal* requirement in the Bug 2 algorithm, what would it have done on the environment in figure 3? Try it with the robot turning to the right when it hits the obstacle, and with it turning left.

Exploration 8: Integrate your implementation of Bug2 into your program to drive in a square.

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment 4 Addendum

Clarifications for Question 9

- Before any component machine is stepped the first time, it must be reset.
- You can assume that each machine in the sequence, after it is reset, will have to be stepped at least once before it is done.
- Be sure that your `step` method returns an output.
- The `done` method may be called many times, and shouldn't change the state of the system (that is, it shouldn't advance the counter or call `step` on any component machines).
- When you call `run` on a text sequence machine, don't worry if the very first character is repeated twice, or if there is an extra `None` at the end. (You should be able to trace through the code, though, and understand why it's happening).
- Write your `done` method first; think about how you're going to know when the machine as a whole is done.
- Here's a way to think about what has to happen in your `step` method. Let `c` be the counter that is keeping track of which component machine is being executed, and m[`c`] be the `c`th component machine. We want to be sure that, if the whole machine isn't yet done, that we make *exactly* one call to `step` on a component machine.
  - If the composite machine is done, just return
  - Else, if m[c] isn't done, then step it.
  - Else (m[c] is done)
    - Increment `c`
    - If m[c] is a machine in our list (`c` isn't too big)
      - Reset m[c]
      - Step m[c]
  - Return the current output

Clarifications for Question 10, 11

- In the example code for `GoForwardUntilXLimit`, the line `def self.currentOutput():` should be `def currentOutput(self):`
- These behaviors are given sensor readings as input and should generate actions as outputs. They shouldn't call motor command. And the only place you should need to call *collectSensors()* or `pose()` is in the `reset` method.

Clarifications for Question 12

- Put your `TBDrive` and `TBTurn` classes in the brain file.
- Be sure to load a simulator before you load your brain.

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Assignment 4, Issued: Tuesday, Feb. 26: Revised

Revised Version

Overview of this week's work

In software lab

- Work through the software lab.
- Submit whatever work you have finished at the end of the lab into the tutor.

Before the start of your design lab on Feb 28 or 29

- Read the class notes and review the lecture handout.
- Do the on-line tutor problems in section PS.5.2.
- Read the entire description of the design lab, so that you will be ready to work on it when you get to lab.

In design lab

- Take the nanoquiz in the first 15 minutes; don't be late.
- Work through the design lab with a partner, and take good notes on the results of your work.

At the beginning of your next software lab on Mar 4 or 5

- Submit online tutor problems in section PS.5.3.
- Submit written solutions to questions 1 through 14 as well as 17 and 18. All written work must be done individually, and conform to the homework guidelines on the web page.

- **You will need SoaR in this software lab, so if you don't have SoaR installed on your own machine, use a lab laptop or Athena station.**
- **Get the lab4 files via `athrun 6.01 update` or from the home page. We have given you several that have `Skeleton` in the name. You should make a copy of those files and rename them to remove the `Skeleton`. You will get failing `imports` otherwise.**

Software Lab: Combinations of state machines

In class we discussed three methods of making new state machines out of old ones: serial composition, parallel composition, and feedback composition. Below (and in the file `SimpleSMSkeleton.py`) is the skeleton of the `SerialSM` class. We've provided the constructor method, which take state machines as input and construct a new, composite state machine.

```
class SerialSM:
    def __init__(self, sm1, sm2):
        self.m1 = sm1
        self.m2 = sm2

    # Be careful to keep the timing consistent.  The input to m2 has
    # to be the current output of m1, not the output after it is
    # stepped.
    def step(self, input=None):
        # Your code here

    def currentOutput(self):
        # Your code here
```

Question 1: Write the `step` and `currentOutput` methods for a serial SM. Test it by combining two `incr` machines (the definition is in `SimpleSMSkeleton.py`). Draw a picture showing how the machines are connected. First, figure out what the result *ought* to be by filling out a table like this one (remember that, by definition, $output_1 = input_2$:

step	$input_1$	$state_1$	$output_1$	$state_2$	$output_2$
0					
1					
2					
3					

Then test to be sure your machine is doing the right thing.

Question 2: Compare the results of composing a `sum` machine (defined in `SimpleSMSkeleton.py`), which outputs the sum of all of its inputs so far with an `incr` machine. Predict what the result ought to be by filling out a table like this one:

step	input_1	state_1	output_1	state_2	output_2
0					
1					
2					
3					

Then be sure you're getting the right result.

Feedback The following class defines a feedback state machine. Its `__init__` method takes a state machine, `sm`, and returns a state machine with no inputs, which consists of `sm` with its output fed back to its input. We'll define the output of the feedback machine to be the same as the output of `sm`.

```
class FeedbackSM:
    def __init__(self, sm):
        self.m = sm

    def step(self, input=None):
        return self.m.step(self.m.currentOutput())

    def currentOutput(self):
        return self.m.currentOutput()
```

Now, we can use this to couple two machines together, with the outputs of one machine serving as the inputs to the other, and vice versa.

```
def simulatorSM(m1, m2):
    return FeedbackSM(SerialSM(m1, m2))
```

Question 3: In the code file, you'll find the definition of `makeIncr`, which takes an initial state as an argument, and then thereafter updates its state to be its input plus 1. If we make the following coupled machine, we get a potentially surprising string of outputs:

```
>>> fizz = simulatorSM(makeIncr(10), makeIncr(100))
>>> run(fizz)
100
11
102
13
104
15
106
17
108
19
110
```

Explain why this happens. Draw a picture of the objects involved in this machine and say exactly what state variables they contain. Fill in a table describing the inputs, states, and outputs of each component machine at each step.

Question 4: Imagine a robot driving straight toward a wall. We would like to use a state machine to model this system, where the state is the distance from the robot to the wall in meters (pick an initial distance for your initial state) and the input is the robot's current forward velocity in meters per second. Assume that each state transition corresponds to the passage of 0.2 seconds. Write a primitive state machine to model this system. Hint: the distance should go down as the machine steps.

Question 5: Now, let's think of the controller that this robot might be executing. Assume the robot would like to stop at some distance $\mathtt{d}_{Desired}$ from the wall. The controller can be modeled as a state machine that receives, as input, the current distance to the wall and generates, as output, a velocity. For now, assume there's no inertia involved, and so the controller can choose any new velocity it wants, independent of its previous velocity. In particular, let's assume it selects a new velocity that is equal to $\mathtt{k}(\mathtt{d} - \mathtt{d}_{Desired})$, where `d` is the current distance to the wall, $\mathtt{d}_{Desired}$ is the desired distance to the wall, and `k` is a "gain" constant. Write a primitive state machine to model this controller.

Question 6: Now, use the `simulatorSM` function to put these two machines together. Run the coupled machine. Try a large value of `k` and a small value of `k`. What happens? (By setting `SimpleSM.verbose` to `True`, you can get `PrimitiveSM.step` to print out information about each step of each primitive machine.)

Question 7: You can plot data from inside soar. If all you want to do is plot, the best thing is to make a brain with only a setup method, and put your plotting commands in there. Then, when you run the brain, you'll see your data plotted in a window. The `SimpleSMBrainSkeleton.py` file defines a `setup` function that will plot some data.

We have modified `run`, in the `SimpleSMSkeleton.py` file, so that it returns a list of all the outputs generated by the machine. So, you can just copy the `setup` and `step` definitions from `SimpleSMBrainSkeleton.py` file to the file with your state-machine definitions, call `run` inside of `setup` to get some data, and then plot it using `graphDiscrete`. If you select that file as a brain in soar, it will run your `setup` method.

Produce plots of the distance between the robot and the wall, over time, with two interestingly different values of `k`. Note that the graphing windows have a Save button that lets you save the graphs to a Postscript (.ps) file.

Now, check your understanding with the following question:

Question 8: Ralph Reticulatus correctly implements `SerialSM.step`, and then makes a common mistake, and tries to run the following code:

```
rm = makeSumSM()
ralph = SerialSM(rm, rm)
>>> transduce(ralph, [1]*10)
[0, 0, 2, 6, 14, 30, 62, 126, 254, 510, 1022]
```

(Note that this result depends on how, precisely, Ralph wrote his `SerialSM.step` method. There are multiple ways to write that method correctly that will result in different outputs in this case; but none will generate the desired sequence.)

Rita Rocksnake recognized Ralph's problem, and reminded him that if we were working with two bank accounts, he'd make two different instances of the `Account` class. Rita wrote this code, instead:

```
rita = SerialSM(makeSumSM(),makeSumSM())
transduce(rita, [1]*10)
[0, 0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

She got the intended result. Explain why. Draw a picture of all of the objects in both Ralph and Rita's attempts, and say exactly what say what state variables they contain.

Software Exploration: New primitive state machines

This is worth 2 out of 10 points for the Week 4 exploration.

In the basic work of this lab, we approached the serial, parallel, and feedback composition of machines by defining new classes of state machines, in such a way that, for example, an instance of a serial composition state machine contained instances of the two original state machines, asked them to store their state internally, and simply asked them to step when necessary.

That approach works well when the goal is only to be able to run the composite state machines. However, later in the course, we will find that state machines can be used as models of an environment, for example, to plan a sequence of actions by trying out different action sequences in the model and seeing which ones perform the best. In order to do that, we need to know the transition function for the composite state machine as a single function.

So, in this exploration, we will pursue another strategy for composing state machines, in which we provide a function `makeSerialSM` that takes as input two *primitive* state machines and returns another *primitive* state machine. That means, it must construct new transition, output, and initialization functions that describe the operation of the entire serial state machine, and use them as arguments to the constructor of the `primitiveSM` class. To write such a function, you have to start by thinking carefully about what the state of the composite machine will be.

Exploration 1: Write the function `makeSerialSM` of type

$$(\mathit{primitiveSM}, \mathit{primitiveSM}) \rightarrow \mathit{primitiveSM}$$

Exploration 2: Write the function `makeParallelSM` of type

$$(\mathit{primitiveSM}, \mathit{primitiveSM}) \rightarrow \mathit{primitiveSM}$$

Exploration 3: Write the function `makeFeedbackSM` of type

$$\mathit{primitiveSM} \rightarrow \mathit{primitiveSM}$$

Exploration 4: Demonstrate these composition methods using various primitive machines that you constructed in the earlier parts of the lab.

Design Lab: Combinations of terminating behaviors

In lab 2, we explored a system for making primitive behaviors and combining them. In that case, our behaviors were all operating at once, and their outputs were expressed as sets of actions. It's possible to do a lot of different things this way, but many people are frustrated by the inability to ask their robot to do things sequentially, like drive up to the wall and then turn left. In this lab, we'll develop a new system of behavior definition and combination, which is yet another instance of the PCAP idea, that allows sequential combination of behaviors.

Terminating state machines

So far, the state machines that we have looked at all run forever, performing a transduction from a potentially infinite stream of inputs to a potentially infinite stream of outputs. Sometimes our work feels like that too! But sometimes, jobs actually get done, and we move on to new tasks.

Our first step will be to build on our original state-machine abstract data type, to construct a *terminating state machine* ADT. In addition to providing the `step` and `currentOutput` methods that regular SM's provided, terminating state machines have to provide two more methods:

- `done`: $() \rightarrow \mathtt{Boolean}$, which is `True` when the machine has terminated; and
- `reset`: $() \rightarrow ()$, which resets the machine back to its original initial state.

In other words, a terminating state machine is one where we can ask if it is "done." A machine is said to have *terminated* when its `done` method returns `True`. The way the machine decides if it has terminated is by computing a function of the machine state, called `terminationFunction`.

We can take advantage of the object-oriented programming facility of *inheritance* to build terminating state machines. Rather than redefining a whole new class of primitive terminating state machines, we can make it be a subclass of the `PrimitiveSM` class. Here's how it goes:

```
class PrimitiveTSM(PrimitiveSM):
    def __init__(self, transitionfn, outputfn, initStatefn,
                 terminationfn = lambda x: False):
        self.terminationFunction = terminationfn
        self.initialStateFunction = initStatefn
        PrimitiveSM.__init__(self, transitionfn, outputfn, initStatefn)

    def done(self):
        return self.terminationFunction(self.currentState)

    def reset(self):
        self.currentState = self.initialStateFunction()
```

Here, we had to redefine the `__init__` method to do some additional work. It has a new argument, `terminationfn`, which defaults to be a function that always returns `False`; so, the first thing it does is to store that function. Then, it also stores the `initStatefn` argument, because it will need it to implement the `reset` method. Finally, it calls the `__init__` method of the `PrimitiveSM` class to do the rest of the work.

We don't need to change the `step` or `currentOutput` methods from the way they work on `PrimitiveSM`s, so we inherit them from `PrimitiveSM` and don't need to say anything about them.

Then, we define the `done` method, which simply calls the `terminationFunction` function on `self.currentState`. Who defined `self.currentState`? The `__init__` method of `PrimitiveSM`.

And we define the `reset` method to reset the current state to whatever value is returned by `self.initialStateFunction`.

Sequential combination

Now, we'd like to implement some means of combining these machines so we can make more complex overall sequential structures. How can we make this happen?

We'd like to define a new class of terminating state machines, called `SequentialTSM`, that takes as an initialization argument a list of terminating state machines. Running the sequence amounts to running the first machine until it terminates, then the second machine until it terminates, and so on.

To do this the sequence machine keeps track of which component machine it is currently running, and keeps calling that machine's associated `step` function until it has terminated; then starts running the next machine in the sequence, and so on. The overall sequence machine terminates on the step after the last component machine terminates.

The code below contains all but the `step` and `done` methods for the class, which you should complete. Our `reset` method actually sets the initial state, so be sure to reset any component SM before beginning to step it.

```
class SequentialTSM():
    def __init__(self, smList):
        self.smList = smList
        self.reset()

    def reset(self):
        self.counter = 0
        self.smList[self.counter].reset()

    def currentOutput(self):
        if not self.done():
            return self.smList[self.counter].currentOutput()
```

Here are some simple state machines and combinations to use when you're testing your work, below.

```
def makeCharTSM(c):
    return PrimitiveTSM(lambda s, i: True,      # transition
                        lambda s: c,            # output
                        lambda: False,          # initial state
                        lambda s: s)            # done
def makeTextSequenceTSM(str):
    return SequentialTSM([makeCharTSM(c) for c in str])
```

The `makeCharTSM` procedure takes a character `c` and returns a terminating state machine whose state is either `True` or `False`. If the state is `True`, then the machine is done. The initial state is `False`, and one step will change the state to `True`. The output is the character `c`. So, this machine runs for one step and generates the output sequence [`c`].

The `makeTextSequenceTSM` procedure takes a string as an argument and returns a `SequentialTSM` that is made up of individual primitive TSMs that output a single character. So, if you `run` the machine `makeTextSequenceTSM('abcd')`, the output sequence should be `['a', 'b', 'c', 'd']`. (Note that we have provided a new version of `run`, that runs a machine until it terminates (or until a limit is reached, whichever comes first).

Use these procedures to test your `SequentialTSM` class, below.

Question 9: Write the `step` and `done` methods of the `SequentialTSM` class. Be sure that each call to `step` of the sequential machine results in exactly one call to `step` of some component machine. The sequential machine should terminate one step after the last of its component machines does (that is, if the `done` method of the last component machine first returns `True` on step n, then the sequential machine should first return `True` on step $n + 1$).

Checkpoint 1: To be completed by 45 minutes after the beginning of lab

Terminating behaviors

We can use our new sequencing abilities to get the robot to perform behaviors in sequence. We'll concentrate, this week, on building deterministic behaviors that specify completely how the robot is to be controlled; but we'll combine them by building sequential combinations of them.

A *terminating behavior* is a TSM that transduces a sequence of sensor inputs (each of which is our usual list of sonar readings and a pose) into a sequence of actions. Here's the basic brain structure for using a terminating behavior.

```
def setup():
    robot.behavior = yourTSMHere
    robot.behavior.reset()

def step():
    if robot.behavior.done():
        doAction(stop)
    else:
        doAction(robot.behavior.step(collectSensors()))
```

If you defined a new terminating behavior class, like `DoTheMacarena`, then you'd have `DoTheMacarena()` in the code above in place of `yourTSMHere`.

In the `setup` function, we make whatever calls are necessary to construct the terminating state machine that will generate the robot's behavior. We store that machine in the variable `robot.behavior`. You can think of `robot` as an object that will persist across invocations of `setup` and `step` in the robot brain, and will let us store the state of our behavior (which is stored in the state of a TSM).

In the `step` function, we check to see if the behavior is done, and, if so, we stop. Otherwise, we collect a fresh set of sensor values, feed them as input into the behavior machine's `step` method, get an action as an output from that machine, and issue the associated command.

Notice that the brain and the behavior each have their own `step` method, and that these are different. This is an example of generic functions, implemented with the aid of object-oriented programming. In this implementation, the brain's `step` method calls the behavior's `step` method as long as the behavior is not `done`.[1]

[1]Are you wondering why we refer to the brain's `step` as a *method*, when you don't see any class definition here? There really is a `Brain` class with `step` and `setup` methods, but the class definition is buried inside the implementation of SoaR. There's also a `robot` object that the brain uses in order to store attributes. That's what lets the `setup` procedure store something as `robot.behavior`, which can be referenced by the brain's `step`. You can use `robot` for storing anything that you like, and it will be local to the particular instance of the brain that SoaR is running.

Let's start by building some basic terminating behaviors for the robot. Here's a somewhat stupid[2] behavior, to illustrate how it might work. Imagine that we wanted to define a new primitive terminating behavior object, called `GoForwardUntilDeltaX`, which moves the robot forward until its x coordinate has increased by some value `deltaX`. We might write code like this:

```
class GoForwardUntilDeltaX:
    def __init__ (self, deltaX):
        self.deltaX = deltaX

    def reset(self):
        (sonars, pose) = collectSensors()
        self.initialX = pose[0]
        self.currentX = pose[0]

    def step(self, sensors):
        (sonars, pose) = sensors
        self.currentX = pose[0]
        return self.currentOutput()

    def self.currentOutput():
        if self.done():
            return stop
        else:
            return go

    def done(self):
        return self.currentX > self.initialX + self.deltaX
```

The initializer for the class just remembers the delta value. When the behavior is reset (which must be done before it is run for the first time), it looks at the robot's current pose, and the initial x coordinate, both as `self.initialX`, which will remain the same until the behavior is reset again, and as `self.currentX`, which will be updated on each step. On each step, the behavior selects the x coordinate of the pose from the sensor readings and remembers it. Then it generates an appropriate output, by calling its own `currentOutput` method. This is a behavior, so the output should be an action. The `currentOutput` method checks to see if the behavior is done; if so, it returns `stop` and otherwise it returns `go`. The `done` method returns the value `True` when the x coordinate of the current pose is greater than the x coordinate of the initial pose by at least `self.deltaX`.

Implement the following primitive terminating behaviors and test them by themselves using the brain code found in `TSMBrainSkeleton.py`. That file also contains some helper functions that you might find useful.

When you're testing terminating behaviors, you'll need to reload the brain inside SoaR once a behavior has terminated. Why? Because the program has to be run again from the beginning; if it just continues from where it was, the behavior might still think it's done.

[2]Why stupid? Because it's hard to imagine why it would be useful to move in whatever direction the robot is pointed until its global x coordinate has increased by some amount. It's entirely possible that, given the heading of the robot, x will decrease or remain unchanged as the robot moves forward.

Question 10: Define a class `TBDrive`, with an `__init__` method that takes as an argument a distance `d` to move. It should move forward a fixed distance, `d`, from the robot's position at the time the `reset` method is called, along the current heading, and then terminate. The class should be a terminating behavior in the sense that it provides `step`, `currentOutput`, `reset`, and `done` methods. *Caveat:* Remember that the robot might be moving at an arbitrary angle with respect to its global coordinate system.

Question 11: Define a class `TBTurn`, with an `__init__` method that takes as an argument an angle `a` to rotate through. It should be a terminating behavior, as above. It should rotate `a` radians from the robot's heading at the time the `reset` method is called, and then terminate. *Caveat:* Remember that `pose()` returns an angle between 0 and 2π, so finding the difference between two angles is not trivial. We have provided a helper in `TSMBrainSkeleton.py` function you may find useful.

Question 12: Test these behaviors in the simulator, and take note of their performance.

Think carefully about an appropriate termination condition to use in each case. **Don't start implementing these until you've talked with your LA about your plan!**

Checkpoint 2: To be completed by 1.5 hours after the beginning of lab

Hip to be Square

Question 13: Now, use sequencing and your new primitives to make the actual robot drive in a one meter square. Explain how your program works. Draw a diagram of the different class instances involved.

Question 14: Execute your program for driving in a square repeatedly. Describe your results. Think about a simple way of measuring and describing the accuracy of your program running on the robot.

Checkpoint 3: To be completed by 2 hours after the beginning of lab

At this point, if you have not completed questions 4 through 7, switch to doing them.

If you have completed them, you may continue on to do questions 15 and 16, which will count for 4 exploration points. You may do this **or** the "Bug Algorithm" exploration at the end of the lab (which is considerably more difficult), but may not get credit for both.

Safety

We have now constructed a useful framework for building sequential behaviors on top of a substrate that still affords frequent reading of and reaction to sensor readings. This means that, while carrying out a sequence of behaviors, the robot can react to surprises. For now, we'll show how it can change its low-level behavior in reaction to sensor values, without changing the actual sequence of high-level steps it is taking. Later in the class, we'll develop a stream-based method of sequential programming that affords a great deal of flexibility in the high-level sequencing, as well.

Question 15: Write a class `TBDriveSafely` that has the basic `TBDrive` terminating behavior as a superclass, and that overrides one of the superclass methods so that the robot will sit and wait rather than run into an obstacle. Note that it shouldn't terminate when it's blocked. Test this on the robot and describe your experiments.

More means of combination

Implement two more means of combination for terminating behaviors. Here are some that we have thought of:

- **RepeatTSM**: take a terminating state machine and a positive integer `k`. Executes that state machine until done, `k` times.
- **IfTSM**: take a condition and two terminating behaviors. Test the condition when this behavior is reset (not created!), and then execute the first behavior if the condition was true and the second if it was false.
- **WhileTSM**: take a condition and a terminating behavior. When the behavior is reset, evaluate the condition. If it's false, terminate. If not, execute the terminating behavior, then test the condition again, etc.
- **ParallelFunTSM**: take a terminating behavior and a function, and call the function on every step of the terminating behavior.

And here's another possibly interesting behavior

- **ForwardUntilBlockedTB** that is a terminating behavior that moves forward until it is blocked in front, and then terminates.

Question 16: Use all of your means of combination to make an interesting robot behavior (the robot macarena, for example). Run this on the robot and describe the results.

Concepts covered in this lab

- Encapsulation of state into objects can provide useful abstraction.
- There are many different frameworks for abstraction and combination, and it's important to choose or design one suited to your problem.
- It can be hard to get repeatable behavior from a physical device.

Homework due in your lab on March 4 or 5

1. Log in to the online tutor and complete the second half of this week's tutor problems.
2. Write up and hand in the answers to questions 1 through 14, as well as the following thought questions.

Question 17: How reliable was your program to drive in a square? What do you think was the main contributing source to the errors? What are some strategies for reducing the error?

Question 18: We have now seen two different frameworks for making primitive robot behaviors and combining them: the constraint (set-based) and sequential (terminating behaviors) approaches. Consider a robot for operating in a household, doing chores. Give examples of situations in which each type of behavior combination would be appropriate.

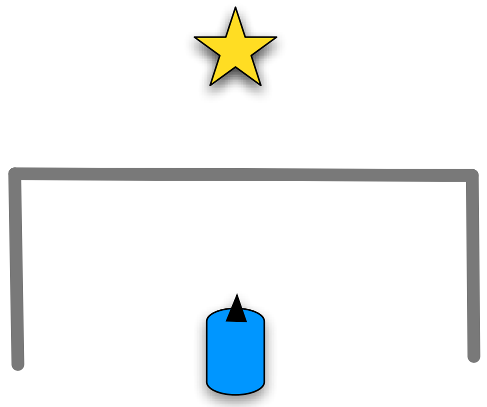


Figure 1: Robot in a cul-de-sac.

Exploration: Bug Algorithm

This is worth 8 out of 10 points for the Week 4 exploration.

You should have found that your program for moving forward is of of limited utility in the presence of obstacles. That is, that your robot could find an obstacle directly between itself and the goal that it is trying to reach. Some kinds of blockages can be escaped by simple behaviors (consider the goal as an attractive force and the sonar measurements as repulsive forces, add up all the forces and move along the resultant force), but a robot trapped in a cul-de-sac, as shown in figure 1, will have to be smarter to reach its goal.

One effective method for getting out of such jams is the “Bug 2” algorithm[3]. The idea is illustrated in figure 2. Here’s the algorithm:

- Compute the line L from start to goal
- Follow that line until an obstacle is encountered
- Follow the obstacle (in either direction) until you encounter the L line again, *closer to the goal*
- Leave the obstacle and continue toward the goal

Note that in the Bug2 algorithm the robot’s motion is not just a function of the sensors, it depends on the line L which was defined when the robot started. So, it requires remembering values between invocations of the `step` procedure. If you want to use a variable to store a value between invocations of the `step` procedure, you can extend your brain to something like this:

```
#code that is called once at startup
def setup():
    print "start"
    robot.x = 0

#code that is called about 10 times a second
def step():
    robot.x = robot.x + 1
    print "Step ", robot.x
```

[3]It is often better, but sometimes much worse than another algorithm called “Bug 1”. For more info see `http://www.cs.jhu.edu/~hager/Teaching/cs336/Notes/Chap2-Bug-Alg.pdf`

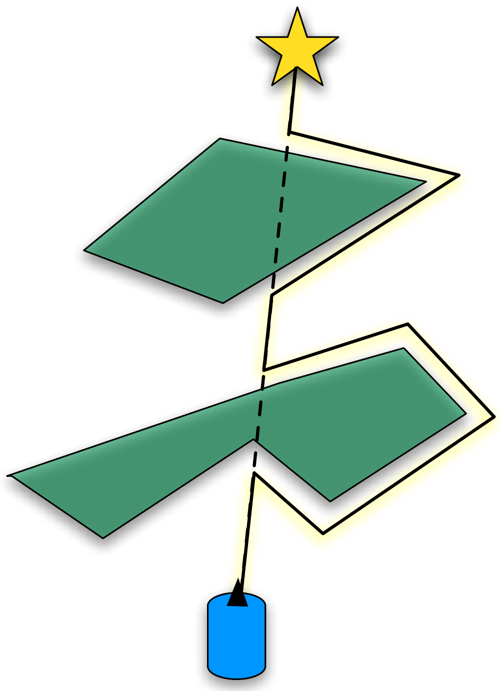


Figure 2: The Bug2 algorithm.

When implementing Bug2, you can define the line L in the `setup` procedure and then the basic behavior in the `step` procedure. In later weeks, we will develop more elaborate algorithms that depend on *state*, which encapsulates the robot's history.

A comment on representing lines. In high school, they probably taught you to represent the line between two points (x_1, y_1) and (x_2, y_2) in the form $y = mx + b$ where m is the slope $y_2 - y_1 / x_2 - x_1$ and b is the y-intercept (what's the expression for that?). You can use m and b to represent the line L, you can test whether an (x, y) position is on the line by checking whether it (nearly) satisfies[4] the equation for the line. You should note that the $y = mx + b$ is not really a very good representation for lines; it cannot represent vertical lines, since their slope is infinite. A better representation is $ax + by + c = 0$. You can stick with $y = mx + b$ for now, but you should check for the vertical case and possibly change one of the endpoints a little bit.

The bug algorithm has some state in it. Think about how you could use the terminating sequential behavior framework (or state machines, or some other systematic way of using state) to structure your algorithm.

To simplify testing and debugging in the simulator, we recommend adding this definition to your file

```
def cheatPose():
    return app.soar.output.abspose.get()
```

and using `cheatPose()` instead of `pose()` to determine where the robot is. If you do this, then when you drag the robot around the simulator window with the mouse, it will magically know the

[4]Even in the simulator you cannot expect that the position of the robot will fall exactly on the line.

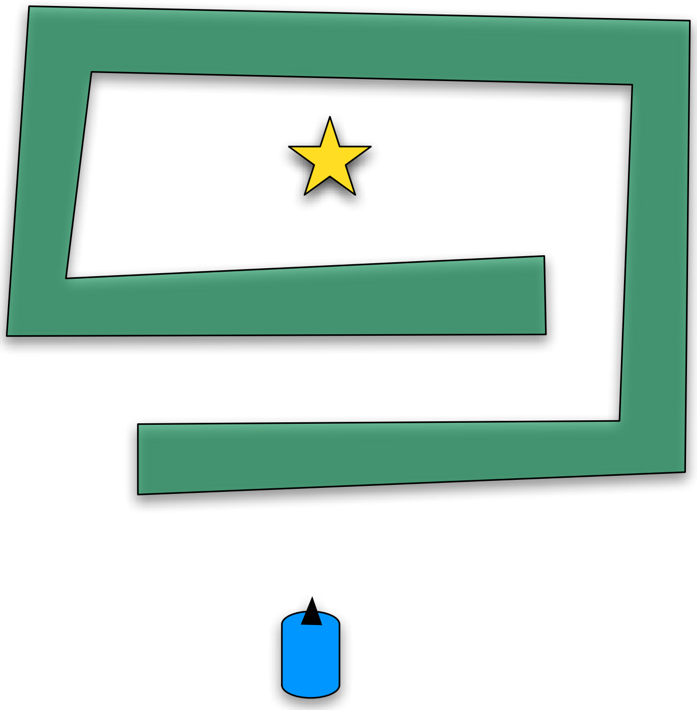


Figure 3: What would Bug2 do in this domain without the *closer to goal* test?

true pose. (On the real robot, of course, if you were to pick it up and put it down, it would have no idea that it had moved.)

Exploration 5: Implement Bug2 in the SoaR simulator. Test it on the `BugTest` worlds in the `ps4` distribution.

Exploration 6: Explain why Bug2 would be hard to implement on the real robots.

Exploration 7: If we hadn't added the *closer to the goal* requirement in the Bug 2 algorithm, what would it have done on the environment in figure 3? Try it with the robot turning to the right when it hits the obstacle, and with it turning left.

Exploration 8: Integrate your implementation of Bug2 into your program to drive in a square.

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Course notes for Week 4

1 State, objects, and abstraction

Last week we introduced object-oriented programming, motivating classes because they provide a convenient way of organizing the procedures and data associated with an abstract data type. This week, we'll look at some other important abstractions strategies for computer programs and see how OOP can help us with them, as well.

In the context of our table and the PCAP framework, we will pay special attention to generic functions and inheritance in OOP, which give us methods for capturing common patterns in data (and the procedures that operate on that data).

	Procedures	Data
Primitives	`+`, `*`, `==`	numbers, strings
Means of combination	`if`, `while`, `f(g(x))`	lists, dictionaries, objects
Means of abstraction	`def`	abstract data types, classes
Means of capturing common patterns	higher-order procedures	generic functions, inheritance

1.1 Objects and state

We saw, last week, how to define a convenient bank-account object that implements an ADT for a bank account.

```
class Account:
    def __init__(self, initialBalance):
        self.currentBalance = initialBalance
    def balance(self):
        return self.currentBalance
    def deposit(self, amount):
        self.currentBalance = self.currentBalance + amount
    def creditLimit(self):
        return min(self.currentBalance * 0.5, 10000000)
```

We can use this ADT to create and manage several bank accounts at once:

```
>>> a = Account(100)
>>> b = Account(1000000)

>>> Account.balance(a)
100
>>> a.balance()
100
>>> Account.deposit(a, 100)
>>> a.deposit(100)
>>> a.balance()
300
>>> b.balance()
1000000
>>> a.balance()
300
```

The `Account` class contains the procedures that are common to all bank accounts; the individual objects contain state in the values associated with the names in their environments. That state is *persistent*, in the sense that it exists for the lifetime of the program that is running, and doesn't disappear when a particular method call is over.

1.2 Generic functions

Now, imagine that the bank we're running is getting bigger, and we want to have several different kinds of accounts. Now there is a monthly fee just to have the account, and the credit limit depends on the account type. Let's see how it would work to use dictionaries to store the data for our bank accounts. Here's a new data structure and two constructors for the different kinds of accounts.

```
def makePremierAccount(balance, rate, owner, ssn):
    return {"balance": balance,
            "interestRate": rate,
            "owner": owner,
            "ssn": ssn,
            "type": "Premier"}

def makeEconomyAccount(balance, rate, owner, ssn):
    return {"balance": balance,
            "interestRate": rate,
            "owner": owner,
            "ssn": ssn,
            "type": "Economy"}

a5 = makePremierAccount(3021835.97, .0003, "Susan Squeeze", "558421212")
a6 = makeEconomyAccount(3.22, .00000001, "Carl Constrictor", "555121348")
```

The procedures for depositing and getting the balance would be the same for both kinds of accounts. But how would we get the credit limit? We could have separate procedures for getting the credit limit for each different kind of account:

```
def creditLimitEconomy(account):
    return min(account['balance']*0.5, 20.00)
def creditLimitPremier(account):
    return min(account['balance']*1.5, 10000000)

>>> creditLimitPremier(a5)
4532753.9550000001
>>> creditLimitEconomy(a6)
1.6100000000000001
```

But doing this means that, no matter what you're doing with this account, you have to be conscious of what kind of account it is. It would be nicer if we could treat the account generically. We can, by writing one procedure that does different things depending on the account type. This is called a *generic* function.

```
def creditLimit(account):
    if account["type"] == "Economy":
        return min(balance*0.5, 20.00)
    elif account["type"] == "Premier":
        return min(balance*1.5, 10000000)
    else:
        return min(balance*0.5, 10000000)

>>> creditLimit(a5)
4532753.9550000001
>>> creditLimit(a6)
1.6100000000000001
```

In this example, we had to do what is known as *type dispatching*; that is, we had to explicitly check the type of the account being passed in and then do the appropriate operation. We'll see later in this lecture that Python has the ability to do this for us automatically.

1.3 Classes and inheritance

If we wanted to define another type of account as a Python class, we could do it this way:

```
class PremierAccount:
    def __init__(self, initialBalance):
        self.currentBalance = initialBalance
    def balance(self):
        return self.currentBalance
    def deposit(self, amount):
        self.currentBalance = self.currentBalance + amount
    def creditLimit(self):
        return min(self.currentBalance * 1.5, 10000000)
>>> c = PremierAccount(1000)
>>> c.creditLimit()
1500.0
```

This will let people with premier accounts have larger credit limits. And, the nice thing is that we can ask for its credit limit without knowing what kind of an account it is, so we see that objects support generic functions, as we spoke about them earlier.

However, this solution is still not satisfactory. In order to make a premier account, we had to repeat a lot of the same definitions as we had in the basic account class. That violates our fundamental principle of laziness: never do twice what you could do once; instead, abstract and reuse.

Inheritance lets us make a new class that's like an old class, but with some parts overridden or new parts added. When defining a class, you can actually specify an argument, which is another class. You are saying that this new class should be exactly like the *parent class* or *superclass*, but with certain definitions added or overridden. So, for example, we can say

```
class PremierAccount(Account):
    def creditLimit(self):
        return min(self.currentBalance * 1.5, 10000000)
```

```
class EconomyAccount(Account):
    def creditLimit(self):
        return min(self.currentBalance*0.5, 20.00)

>>> a = Account(100)
>>> b = PremierAccount(100)
>>> c = EconomyAccount(100)
>>> a.creditLimit()
100.0
>>> b.creditLimit()
150.0
>>> c.creditLimit()
20.0
```

This is like generic functions! But we don't have to define the whole thing at once. We can add pieces and parts as we define new types of accounts. And we automatically inherit the methods of our superclass (including `__init__`). So we still know how to make deposits into a premier account:

```
>>> b.deposit(100)
>>> b.balance()
200
```

This is actually implemented by setting the subclass's enclosing environment to be the superclass's environment. Figure 1 shows the environments after we've executed all of the account-related statements above. You can see that each class and each instance is an environment, and that superclasses enclose subclasses and classes enclose their instances. So, as a consequence of the rules for looking up names in environments, when we ask a `PremierAccount` instance for its `creditLimit` method, it is found in the enclosing class environment; but when we look for the `deposit` method, it is found in the superclass's environment. Once we know how the environment structure is set up, the details of the look-up process are the ones we know from the rest of Python.

The fact that objects know what class they were derived from allows us to ask each object to do the operations appropriate to it, without having to take specific notice of what class it is. Procedures that can operate on objects of different types or classes without explicitly taking their types or classes into account are called *polymorphic*. Polymorphism is a very powerful method for capturing common patterns.

There is a lot more to learn and understand about object-oriented programming; we have just seen the bare basics. But here's a summary of how the object-oriented features of Python help us achieve useful software-engineering mechanisms.

1. **Data structure**: Objects contain a dictionary mapping attribute names to values.
2. **Abstract data types**: Methods provide abstraction of implementation details.
3. **State**: Object attributes are persistent.
4. **Generic functions**: Method names are looked up in object's environment
5. **Inheritance**: Can easily make new related classes, with added or overridden attributes.

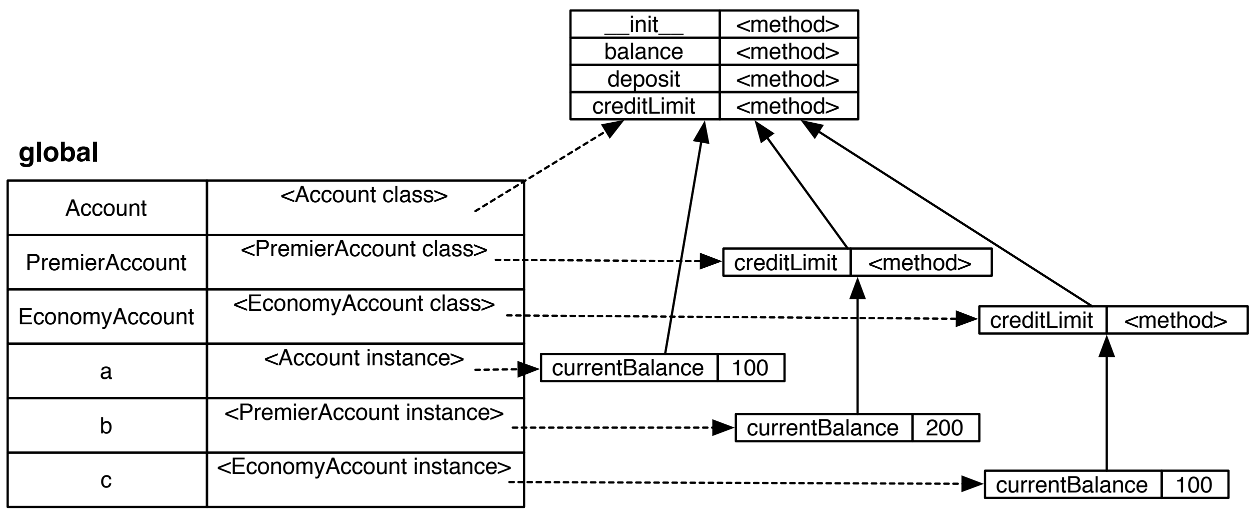


Figure 1: The `Account` class, two subclasses, and three instances. Dashed arrows show the environments associated with instances and classes. Solid arrows show enclosing environments.

2 Combination and abstraction of state machines

Last week, we studied the definition of a primitive state machine, and saw a number of examples. State machines are useful for a wide variety of problems, but specifying them using state transition tables or even more general functions ends up being quite tedious. Ultimately, we'll want to build large state-machine descriptions compositionally: by specifying primitive machines and then combining them into more complex systems. We'll start, here, by looking at ways of combining state machines.

2.1 Machine Composition

We can apply our PCAP (primitive, combination, abstraction, pattern) methodology here, to build more complex SMs out of simpler ones. Figure 2 sketches three types of SM composition: serial, parallel, and feedback.

2.1.1 Serial composition

In serial composition, we take two machines and use the output of the first one as the input to the second. The result is a new composite machine, whose input vocabulary is the input vocabulary of the first machine and whose output vocabulary is the output vocabulary of the second machine. It is, of course, crucial that the output vocabulary of the first machine be the same as the input vocabulary of the second machine.

Recalling the *delay* machine from last week, let's see what happens if we make the serial composition of two delay machines. Let m_1 be a delay machine with initial value $init_1$ and m_2 be a delay machine with initial value $init_2$. Then $serialCompose(m_1, m_2)$ is a new state machine, constructed by making the output of m_1 be the input of m_2. Now, imagine we feed a sequence of values,

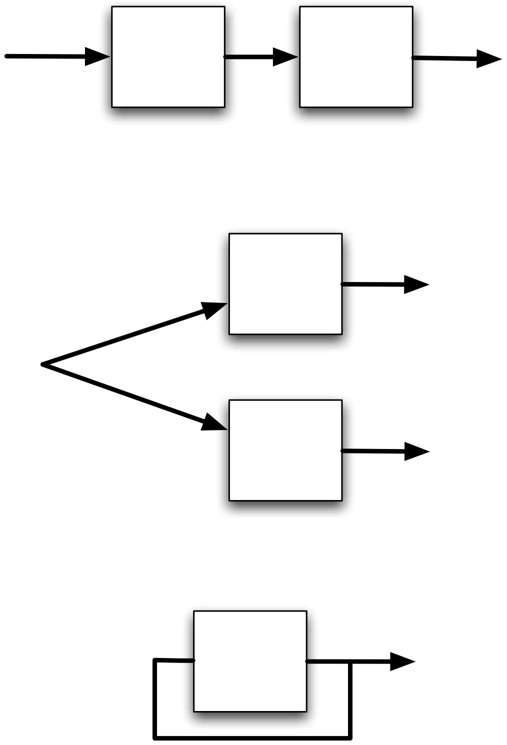


Figure 2: Serial, parallel, and feedback compositions of state machines.

$3, 2, 5, 6$, into the composite machine, m. What will come out? Let's try to understand this by making a table of the states and values at different times:

time	m_1 input	m_1 state	m_1 output	m_2 input	m_2 state	m_2 output
0	3	$init_1$	$init_1$	$init_1$	$init_2$	$init_2$
1	2	3	3	3	$init_1$	$init_1$
2	5	2	2	2	3	3
3	6	5	5	5	2	2
4	—	6	6	6	5	5

Another way to think about state machines and composition is as follows. Let the input to m_1 at time t be called $I_1[t]$ and the output of m_1 at time t be called $O_1[t]$. Then, we can describe the workings of the machine in terms of an equation:

$$O_1[t] = I_1[t-1] \quad \text{, for all values of } t > 0;$$

that is, that the output value at some time t is equal to the input value at the previous time step. You can see that in the table above. The same relation holds for the input and output of m_2:

$$O_2[t] = I_2[t-1] \quad \text{for all values of } t > 0.$$

Now, since we have connected the output of m_1 to the input of m_2, we also have that $I_2[t] = O_1[t]$ for all values of t. This lets us make the following derivation:

$$O_2[t] \;=\; I_2[t-1]$$

$$\begin{aligned} &= O_1[t-1] \\ &= I_1[t-2] \end{aligned}$$

This makes it clear that we have built a "delay by two" machine, by serially composing two single delay machines.

As with all of our systems of combination, we will be able to form the serial composition not only of two primitive machines, but of any two machines that we can make, through any set of compositions of primitive machines.

2.1.2 Parallel composition

In parallel composition, we take two machines and run them "side by side". They both take the same input, and the output of the composite machine is the pair of outputs of the individual machines. The result is a new composite machine, whose input vocabulary is the same as the input vocabulary of the component machines and whose output vocabulary is pairs of elements, the first from the output vocabulary of the first machine and the second from the output vocabulary of the second machine.

2.1.3 Feedback composition

Another important means of combination that we will make much use of later is the feedback combinator, in which the output of a machine is fed back to be the input of the same machine at the next step. The first value that is fed back is the output associated with the initial state of the machine which is being operated upon. It is crucial that the input and output vocabularies of the machine that is being operated on are the same (because the output at step t will be the input at step $t+1$). Because we have fed the output back to the input, this machine doesn't consume any inputs; but we will treat the feedback value as an output of this machine.

Here is an example of using feedback to make a machine that counts.

We can start with a simple machine, an incrementer, that takes a number as input and returns that same number plus 1 as the output. It has very limited memory. Here is its formal description:

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \\ O &= \textit{numbers} \\ t(s,i) &= i+1 \\ o(s) &= s \\ s_0 &= 0 \end{aligned}$$

So, if you feed this machine the stream of inputs $4, 9, 2, 7$, you will get the stream of outputs $0, 5, 10, 3, 8$. As with all of our state machines, the output depends on the input from the previous time step, so it does have memory in that sense, but it doesn't remember much. We can us it to build a machine with persistent memory by feeding the output of the incrementer back to be its input.

To make a counter, which *does* need memory, we do a feedback operation on the incrementer, connecting its output up to its input. More formally, if the input to the incrementer at time t

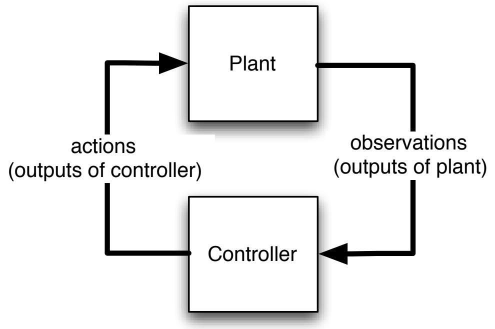


Figure 3: Two machines connected together, as for a simulator.

is $I[t]$ and the output of the incrementer at time i is $O[t]$, then the definition of the incrementer implies that $O[t] = 1 + I[t-1]$. Now, doing the feedback operation means that $I[t] = O[t]$, so that $O[t] = 1 + O[t-1]$. This is a recursive relationship, which bottoms out at $O[0]$, which is the output associated with the initial state of the original machine. We'll be spending time over the next few weeks studying machines whose behavior is defined by equations of this basic kind.

2.2 Plants and controllers

One common situation in which we combine machines is to simulate the effects of coupling a controller and a so-called "plant". A plant is a factory or other external environment that we might wish to control. In this case, we connect two state machines so that the output of the plant (typically thought of as some sensory observations) is input to the controller, and the output of the controller (typically thought of as some actions) is input to the plant. This is shown schematically in figure 3. For example, that's what happens when you build a Soar brain that interacts with the robot: the robot (and the world it is operating in) is the "plant" and the brain is the controller. We can build a coupled machine by first connecting the machines serially and then using feedback on that combination.

As a concrete example, let's think about a robot driving straight toward a wall. It has a distance sensor that allows it to observe the distance to the wall at time t, $d[t]$, and it desires to stop at some distance $d_{desired}$. The robot can execute velocity commands, and we program it to use this rule to set its velocity at time t, based on its most recent sensor reading:

$$v[t] = K(d_{desired} - d[t-1]) \ .$$

This controller can also be described as a state machine, whose input sequence is the values of d and whose output sequence is the values of v.

$$\begin{aligned} S &= \textit{numbers} \\ I &= \textit{numbers} \\ O &= \textit{numbers} \\ t(s,i) &= K(d_{desired} - i) \\ o(s) &= s \\ s_0 &= d_{init} \end{aligned}$$

Now, we can think about the "plant"; that is, the relationship between the robot and the world. The distance of the robot to the wall changes at each time step depending on the robot's forward velocity and the length of the time steps. Let δT be the length of time between velocity commands issued by the robot. Then we can describe the world with the equation:

$$d[t] = d[t-1] - \delta T v[t-1] \ .$$

This system can be described as a state machine, whose input sequence is the values of the robot's velocity, v, and whose output sequence is the values of its distance to the wall, d.

Finally, we can couple these two systems, as for a simulator, to get a single state machine with no inputs. We can observe the sequence of internal values of d and v to understand how the system is behaving.

State machines are such a general formalism, that a huge class of discrete-time systems can be described as state machines. The system of defining primitive machines and combinations gives us one discipline for describing complex systems. It will turn out that there are some systems that are conveniently defined using this discipline, but that for other kinds of systems, other disciplines would be more natural. As you encounter complex engineering problems, your job is to find the PCAP system that is appropriate for them, and if one doesn't exist already, invent one.

State machines are such a general class of systems that although it is a useful framework for implementing systems, we cannot generally analyze the behavior of state machines. That is, we can't make much in the way of generic predictions about their future behavior, except by running them to see what will happen.

Next week, we will look at a restricted class of state machines, whose state is representable as a bounded history of their previous states and previous inputs, and whose output is a linear function of those states and inputs. This is a *much* smaller class of systems than all state machines, but it is nonetheless very powerful. The important lesson will be that restricting the form of the models we are using will allow us to make stronger claims about their behavior.