

CSC324: Principles of Programming Languages

Lecture 2

Wednesday 13 May, 2020

Last time we saw...

- ... how defining a **grammar** specifies the **syntax** of a programming language
- ... how applying **semantics** overtop the syntax allows us to construct **evaluation rules** for the language
- ... how to apply the above to the **lambda calculus**

The Lambda Calculus, pen-and-paper version...

```
<expr> = IDENTIFIER
```

```
| "(" "λ" IDENTIFIER "." expr ")" # function abstraction
```

```
| "(" expr expr ")" # application: applies 2nd expr to 1st
```

The Lambda Calculus, **Python** version...

```
<expr> = IDENTIFIER  
        | "lambda" IDENTIFIER ":" expr  # function abstraction  
        | "(" expr ")" "(" expr ")"    # application:
```

The Lambda Calculus, **Racket** version...

```
<expr> = IDENTIFIER
```

```
| "(" "λ" "(" IDENTIFIER ")" expr ")" # function abstraction
```

```
| "(" expr expr ")" # application: applies 2nd expr to 1st
```

The Lambda Calculus, **Haskell** version...

```
<expr> = IDENTIFIER
```

```
| "\" IDENTIFIER \"->\" expr    # function abstraction
```

```
| expr expr    # application: applies 2nd expr to 1st
```

Pen-and-Paper

```
<expr> = IDENTIFIER  
      | "(" "λ" IDENTIFIER "." expr ")" # function abstraction  
      | "(" expr expr ")" # application: applies 2nd to 1st
```

Python

```
<expr> = IDENTIFIER  
      | "lambda" IDENTIFIER ":" expr # abstraction  
      | expr "(" expr ")" # application
```

Racket

```
<expr> = IDENTIFIER  
      | "(" "λ" "(" IDENTIFIER ")" expr ")" # abstraction  
      | "(" expr expr ")" # application
```

Haskell

```
<expr> = IDENTIFIER  
      | "\" IDENTIFIER "->" expr # abstraction  
      | expr expr # application
```

What does it mean to "run" a program?

$$(\lambda x. x) y$$

Today:

What's missing from the previous lecture?

- name binding: giving a name to a variable
- "iteration" over simple datatypes

Name bindings

I think you have an intuitive notion of a name binding. A name binding is an expression that **associates the value of an evaluated expression to an identifier**, 1) *to be used in a subsequent expression*.

```
(venv) → csc324 git:(master) ipython
Python 3.7.6 (default, Dec 30 2019, 19:38:26)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.
```

```
In [1]: e = 2.71
```

```
In [2]: msg = "allo, there"
```

```
In [3]: e + len(msg)
```

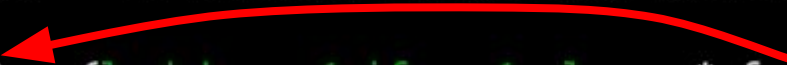
```
Out[3]: 14.71
```

Name bindings

I think you have an intuitive notion of a name binding. A name binding is an expression that **associates the value of an evaluated expression to an identifier**, 2) *to give a lambda expression a name so recursion can happen*

```
(venv) → csc324 git:(master) ✕ ipython
Python 3.7.6 (default, Dec 30 2019, 19:38:26)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: fact = (lambda n: 1 if n==1 else n * fact(n-1))
```



Global name bindings

```
(venv) → csc324 git:(master) x ipython
Python 3.7.6 (default, Dec 30 2019, 19:38:26)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help

In [1]: fact = (lambda n: 1 if n==1 else n * fact(n-1))

In [1]: e = 2.71

In [2]: msg = "allo, there"

In [3]: sum = e + len(msg)
```

We refer to these sorts of name bindings as "top-level" or "global" bindings.

Global name bindings in Racket

Global name bindings in Haskell

Extending our definition of a program

We've said that a program in the lambda calculus is an **expression**. Therefore:

```
<prog> = <expr>
```

```
<expr> = IDENTIFIER
```

```
| "(" "λ" IDENTIFIER "." expr ")" # function abstraction
```

```
| "(" expr expr ")" # application: applies 2nd expr to 1st
```

How do we incorporate top-level bindings into a program?

Extending our definition of a program

We've said that a program in the lambda calculus is an **expression**. Therefore:

```
<prog> = <binding>* <expr>
```

```
<binding> = "(" "define" IDENTIFIER <expr> ")"
```

```
<expr> = IDENTIFIER
```

```
| "(" "λ" IDENTIFIER "." expr ")" # function abstraction
```

```
| "(" expr expr ")" # application: applies 2nd expr to 1st
```


Variable bindings are **immutable**

...and this makes sense if you forget everything you know about programming and only rely on what you know from mathematics

Referential transparency

Definition: An identifier is said to be **referentially transparent** if the identifier can always be substituted for its value without changing the program's behaviour.

Example: any pure function is always referentially transparent. By contrast, an impure function like Python's `datetime.now()` function, are not referentially transparent (calling it one second later will produce a different value!)

Example: an assignment, like `x = x + 1`, is not referentially transparent.

Local name bindings

The previous name bindings were in *global* (or ***top-level***) scope. It is also desirable to bind a value to a name within an expression, such as a local variable inside a function definition.

Local name bindings with lambda

We saw how, for the lifetime of evaluating `(circle-area 10)`, the value of `r` is bound to 10

Can we use this to create local name bindings?

```
(define circle-area (lambda (r) (* pi r r)))  
  
(circle-area 10)
```

```
((λ (x y)
  (sqrt (+ (* x x) (* y y)))))
3 4)
```

```
((λ (x y)
  ((λ (xx yy)
    (sqrt (+ xx yy)))
    (* x x)
    (* y y))))
3 4)
```

Local name bindings with the let expression

This sort of construct is so useful that there is a special expression called *let* for this.

Local name bindings with the let expression

This sort of construct is so useful that there is a special expression called *let* for this.

```
((λ (x y)
  (let* ([xx (* x x)]
         [yy (* y y)])
    (sqrt (+ xx yy)))))
3 4)
```


Local name bindings with the let expression

This sort of construct is so useful that there is a special expression called *let* for this.

```
1 f x y = let xx = x*x  
2           yy = y*y  
3           in sqrt(xx + yy)  
4  
5
```

Does Python need local bindings?

- Not really! Python supports **local variables**, which are defined as statements within some scope
- However, let expressions are nice because they make it clear where a value is bound, which may actually be non-obvious in Python

Does Python need local bindings?

On line 10, we print the value of `is_even`, which is defined inside a while loop. Is this valid Python? If we wrote this in Java, would this compile?

If you were designing Python or Java, would you make the same choice?

```
1 def collatz(n):
2     count = 0
3     while n > 1:
4         count += 1
5         is_even = (n % 2 == 0)
6         if is_even:
7             n = n/2
8         else:
9             n = 3*n + 1
10    print("The final iteration's parity was " + ("even" if is_even else "odd"))
11    return count
12
```

"So if no values ever change, how do we do anything"

Excellent question!

So far, what we've studied makes programming in the functional style limiting. Without mutability, we can't construct a loop, and so any sort of interesting computation becomes impossible.

"So if no values ever change, how do we do anything"

Excellent question!

So far, what we've studied makes programming in the functional style limiting. Without mutability, we can't construct a loop, and so any sort of interesting computation becomes impossible.

...or does it?

Recall: the core problem is that we can bind a local name, but can't assign a subsequent value to it.

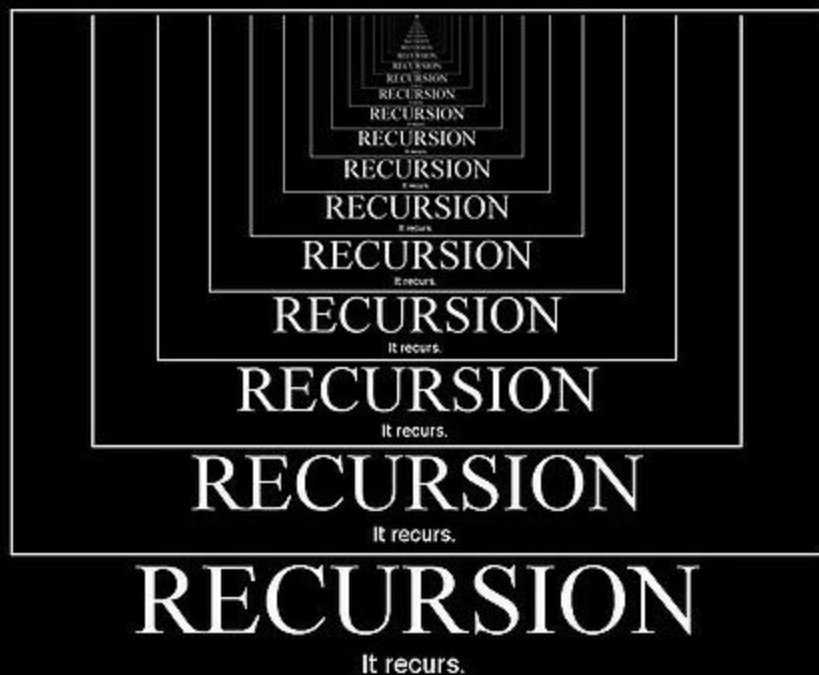
Also: recall the other way we implemented local name binding, with an inner **lambda expression...**

```
((λ (x y)
  (let* ([xx (* x x)]
         [yy (* y y)])
    (sqrt (+ xx yy)))))
3 4)
```

Here, we saw that we could give values to `xx` and `yy` by calling a function.

So, to change `xx` and `yy`, could we call the same function again with different values for those bindings?

```
((λ (x y)
  ((λ (xx yy)
    (sqrt (+ xx yy)))
   (* x x)
   (* y y))))
3 4)
```



From iteration to recursion

Here's a slow way to add numbers:

```
1 def slow_add(m, n):
2     while n > 0:
3         m += 1
4         n -= 1
5     return m
6
7 assert(slow_add(2,3) == 5)
8 assert(slow_add(6,0) == 6)
9 assert(slow_add(0,6) == 6)
10 print("all tests passed!")
11
```

From iteration to recursion

Here's a slow way to add numbers:

Here are the values of m and n each time we re-enter the while loop on line 2:

m	n
2	3
3	2
4	1
5	0

```
1 def slow_add(m, n):
2     while n > 0:
3         m += 1
4         n -= 1
5     return m
6
7 assert(slow_add(2,3) == 5)
8 assert(slow_add(6,0) == 6)
9 assert(slow_add(0,6) == 6)
10 print("all tests passed!")
```

From iteration to recursion

Here's a slow way to add numbers:

We see that the loop terminates when $n == 0$; what should we return?

When $n \neq 0$, what do we want to "mutate" m and n to?

```
1 def slow_add(m, n):  
2     while n > 0:  
3         m += 1  
4         n -= 1  
5     return m
```

```
6  
7 assert(slow_add(2,3) == 5)  
assert(slow_add(6,0) == 6)  
assert(slow_add(0,6) == 6)  
print("all tests passed!")
```

m	n
2	3
3	2
4	1
5	0

```
1 def slow_add(m, n):  
2     if n == 0:  
3         return ...  
4     return slow_add(..., ...)  
5  
6 assert(slow_add(2,3) == 5)  
7 assert(slow_add(6,0) == 6)  
8 assert(slow_add(0,6) == 6)  
9 print("all tests passed!")  
10  
11
```

From iteration to recursion

Here's a slow way to add numbers:

m	n
2	3
3	2
4	1
5	0

```
1 def slow_add(m, n):  
2     while n > 0:  
3         m += 1  
4         n -= 1  
5     return m
```

```
6  
7 assert(slow_add(2,3) == 5)  
assert(slow_add(6,0) == 6)  
assert(slow_add(0,6) == 6)  
print("all tests passed!")
```

```
1 def slow_add(m, n):  
2     if n == 0:  
3         return m  
4     return slow_add(m+1, n-1)  
5  
6 assert(slow_add(2,3) == 5)  
7 assert(slow_add(6,0) == 6)  
8 assert(slow_add(0,6) == 6)  
9 print("all tests passed!")
```

```
10  
11
```

From iteration to recursion (in Racket)

m	n
2	3
3	2
4	1
5	0

```
1 def slow_add(m, n):  
2     if n == 0:  
3         return m  
4     return slow_add(m+1, m-1)  
5  
6 assert(slow_add(2,3) == 5)  
7 assert(slow_add(6,0) == 6)  
8 assert(slow_add(0,6) == 6)  
9 print("all tests passed!")  
10  
11
```

From iteration to recursion

m	n
2	3
3	2
4	1
5	0

```
#lang racket
(require rackunit)

(define (slow-add m n)
  (if (= n 0)
      m
      (slow-add (+ m 1) (- n 1))))

(check-eq? (slow-add 5 5) 10)
(check-eq? (slow-add 0 6) 6)
(check-eq? (slow-add 6 0) 6)
```

From iteration to recursion

To transform an iterative operation on a piece of data into a recursive one:

- Identify the loop, and its *loop invariants*
- The base case of the recursive function is the loop's *termination condition*
- The loop variables, rather than being mutated, get re-evaluated as new values when passed back into the recursive function call

```
1 def slow_add(m, n):  
2     while n > 0:  
3         m += 1  
4         n -= 1  
5     return m  
6
```

```
1 def slow_add(m, n):  
2     if n == 0:  
3         return m  
4     return slow_add(m+1, m-1)  
5  
6 assert(slow_add(2,3) == 5)  
7 assert(slow_add(6,0) == 6)  
8 assert(slow_add(0,6) == 6)  
9 print("all tests passed!")  
10  
11
```

From iteration to recursion, round two

Here's a familiar-looking friend.

Let's try and convert it into a recursive function:

```
1 def fact(n):  
2     n_fact = 1  
3     while n >= 1:  
4         n_fact *= n  
5         n -= 1  
6     return n_fact  
7  
8 assert(fact(1) == 1)  
9 assert(fact(2) == 2)  
10 assert(fact(3) == 6)  
11 assert(fact(20) == 2432902008176640000)  
12 print("all tests passed!")  
13
```


From iteration to recursion, round two

However, we can't continue directly as before: we try to fill in our pieces but we realise that fact only takes one argument, but we needed a second one for n_fact...

```
1 def fact(n):  
2     n_fact = 1  
3     while n >= 1:  
4         n_fact *= n  
5         n -= 1  
6     return n_fact  
7
```

```
1 def fact(n):  
2     if n == 1:  
3         return ...  
4     return fact(n-1) # ??? what happened to n_fact?  
5  
6 assert(fact(1) == 1)  
7 assert(fact(2) == 2)  
8 assert(fact(3) == 6)  
9 assert(fact(20) == 2432902008176640000)  
10 print("all tests passed!")
```

From iteration to recursion, round two

We need to change fact to consume an extra **accumulating** parameter for n_fact.

It's common for the base case of functions that use accumulators to just return the accumulator itself.

```
1 def fact(n, n_fact):
2     if n == 1:
3         return n_fact
4     return fact(n-1, n * n_fact)
5
6 assert(fact(1) == 1)
7 assert(fact(2) == 2)
8 assert(fact(3) == 6)
9 assert(fact(20) == 2432902008176640000)
10 print("all tests passed!")
11
12
```

Are we done?

From iteration to recursion, round two

```
1 def fact(n, n_fact):
2     if n == 1:
3         return n_fact
4     return fact(n-1, n * n_fact)
5
6 assert(fact(1) == 1)
7 assert(fact(2) == 2)
8 assert(fact(3) == 6)
```

```
(venv) → csc324 git:(master) ✗ python /tmp/foo.py
```

```
Traceback (most recent call last):
```

```
File "/tmp/foo.py", line 6, in <module>
```

```
    assert(fact(1) == 1)
```

```
TypeError: fact() missing 1 required positional argument: 'n_fact'
```

From iteration to recursion, round two

Note that anyone who wants to call `inner_fact()` has to go through `fact()`; it is in some sense a **"private function"** of `fact()`.

```
1 def fact(n):
2     def inner_fact(n, n_fact):
3         if n == 1:
4             return n_fact
5         return inner_fact(n-1, n * n_fact)
6     return inner_fact(n, 1)
7
8 assert(fact(1) == 1)
9 assert(fact(2) == 2)
10 assert(fact(3) == 6)
11 assert(fact(20) == 2432902008176640000)
12 print("All tests passed!")
13
14
```

"Couldn't I just have...?"

In this case, yes! This is also a valid transformation into a recursive function. This avoids us having to create a helper `fact_iter()` function.

The steps on the previous slides, though, form a more general-purpose procedure.

```
1 def fact(n):  
2     if n == 1:  
3         return 1  
4     return n * fact(n-1)  
5  
6 assert(fact(1) == 1)  
7 assert(fact(2) == 2)  
8 assert(fact(3) == 6)  
9 assert(fact(20) == 2432902008176640000)  
10 print("All tests passed!")  
11  
12  
13
```