CS61A Lecture 35
Amir Kamil
UC Berkeley
April 12, 2013

Announcements

- HW11 due next Wednesday
- Scheme project out

Dynamic Scope

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope).

Lexical scope: The parent of a frame is the environment in which a procedure was defined.

Dynamic scope: The parent of a frame is the environment in which a procedure was called.

(\texttt{define } \texttt{f (lambda (x) (+ x y))})
(\texttt{define } \texttt{g (lambda (x y) (f (+ x x)))})
(\texttt{g 3 7})

Lexical scope: The parent for \texttt{f}'s frame is the global frame.

Dynamic scope: The parent for \texttt{f}'s frame is \texttt{g}'s frame.

Error: unknown identifier: \texttt{y}

Functional Programming

All functions are pure functions.

- No re-assignment and no mutable data types.
- Name-value bindings are permanent.

Advantages of functional programming:

- The value of an expression is independent of the order in which sub-expressions are evaluated.
- Sub-expressions can safely be evaluated in parallel or lazily.
- Referential transparency: The value of an expression does not change when we substitute one of its sub-expression with the value of that sub-expression.

But... Can we make basic loops efficient?
Yes!

Iteration and Recursion

Reminder: Iteration is a special case of recursion.

Idea: The state of iteration can be passed as parameters.

\texttt{def factorial(n):}
  \texttt{if n == 0:}
    \texttt{return 1}
  \texttt{return n * factorial(n - 1)}

\texttt{def factorial(n, total):}
  \texttt{if n == 0:}
    \texttt{return total}
  \texttt{total = n - 1, total * n}

But this converted version still uses linear space in Python.
Tail Recursion

From the Revised Report on the Algorithmic Language Scheme:

"Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

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

def factorial(n, total):
  if n == 0:
    return total
  return factorial(n - 1, total * n)

Example: Length of a List

```
(define (length s)
  (if (null? s) 0
      (+ 1 (length (cdr s)))))
```

A call expression is not a tail call if more computation is still required in the calling procedure. Linear recursions can often be rewritten to use tail calls.

```
(define (length-tail s)
  (define (length-iter s n)
    (if (null? s) n
        (length-iter (cdr s) (+ 1 n))))
  (length-iter s 0))
```

Eval with Tail Call Optimization

The return value of the tail call is the return value of the current procedure call.

Therefore, tail calls shouldn't increase the environment size.

In the interpreter, recursive calls to `scheme_eval` for tail calls must instead be expressed iteratively.

Example: Reduce

```
(define (reduce procedure s start)
  (if (null? s)
      start
      (reduce procedure
                (cdr s)
                (procedure start (car s))))
```

Recurisve call is a tail call.

Other calls are not; constant space depends on `procedure`.

```
(reduce + '(3 4 5) 2) 120
(reduce (lambda (x y) (cons x y)) '(3 4 5) '(2)) (5 4 3 2)
```
Example: Map

```
(define (map procedure s)
  (define (map-iter procedure s m)
    (if (null? s) m
        (map-iter procedure
                   (cdr s)
                   (cons (procedure (car s)) m)))))
  (reverse (map-iter procedure s nil)))

(define (reverse s)
  (define (reverse-iter s r)
    (if (null? s) r
        (reverse-iter (cdr s)
                     (cons (car s) r)))))
  (reverse-iter s nil))
```

An Analogy: Programs Define Machines

Programs specify the logic of a computational device

![Diagram showing factorial calculation]

Interpreters are General Computing Machines

An interpreter can be parameterized to simulate any machine

```
5 120
```

```
(define (factorial n)
  (if (zero? n) 1 (* n (factorial (- n 1)))))
```

Our Scheme interpreter is a universal machine

A bridge between the data objects that are manipulated by our
programming language and the programming language itself
Internally, it is just a set of manipulation rules

Interpretation in Python

```
eval: Evaluates an expression in the current environment and returns the result. Doing so may affect the environment.
```

```
exec: Executes a statement in the current environment. Doing so may affect the environment.
```

```
eval('2 + 2')
```

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

```
import os
os.system('python <file>')
```

Directs the operating system to invoke a new instance of the Python interpreter.