61A LECTURE 25 – DECLARATIVE PROGRAMMING

Steven Tang and Eric Tzeng
August 6, 2013

Midterm grades up

- Class did well as a whole!
- `glookup -s test2`
- Regrades: Talk to your TA in person.

Albert keeps all of his top secret information in a binary tree. This prevents the layperson from reading his data. However, well trained computer scientists (such as you) can still access his information.

As a further layer of protection, Albert turns some of the nodes in his trees into `Eert` nodes. `Eert` nodes, which have `Tree` as their base class, are like normal `Tree` nodes, except they swap their left and right branches. (Albert settles for nothing less than the most advanced encryption techniques known to man.)

(a) **(3 pt)** Complete the `__init__` method for the `Eert` class **on the next page**. Make sure to use inheritance as much as possible. The `Eert` class should work as follows:

```
>>> e = Eert("61A account info",
...          Tree("Username: cs61a-te"),
...          Tree("Password: imsocool"))
>>> e.entry  # unchanged
"61A account info"
>>> e.left.entry  # swapped with right
"Password: imsocool"
>>> e.right.entry  # swapped with left
"Username: cs61a-te"
```

(b) **(5 pt)** Complete the definitions of the `decrypt` methods for both the `Tree` and `Eert` classes **on the next page**. When the `decrypt` method is invoked on a binary tree containing `Tree` and `Eert` nodes, it returns a copy of the binary tree, but with all `Eert` nodes replaced with `Tree` nodes. During this replacement, you should also swap the left and the right back to their proper positions! Here is a graphical representation of the process:

```
class Tree(object):

    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right

    def decrypt(self):
        "*** PART B ***"


class Eert(Tree):

    def __init__(self, entry, left=None, right=None):
        "*** PART A ***"

    def decrypt(self):
        "*** PART B ***"
```

Announcements

- Proj4 has been out
  - Due in 7 days – Start if you haven't!
  - Recursive art contest deadline one day **before** project is due
  - Future "homework" assignment will be to vote on your favorite submissions
- Final exam next Thursday
  - 7-10pm in 1 Pimentel
  - Any conflicts – notify us immediately
- Final exam review session this weekend
  - See Piazza Poll to vote on your time
  - Potential extra credit – more information later in the week

Laziness

Recall our previous sequence interface:

- A sequence has a finite, known length
- A sequence allows element selection for any element

In the cases we've seen so far, satisfying the sequence interface requires storing the entire sequence in a computer's memory

Problems?

- Infinite sequences - primes, positive integers
- Really large sequences - all Twitter tweets, votes in a presidential election

Streams

A stream is a recursive list with an *explicit* first element and a *lazily computed* rest-of-the-list

```
class Stream(Rlist):
    """A lazily computed recursive list."""
    def __init__(self, first,
                 compute_rest=lambda: Stream.empty):
        assert callable(compute_rest)
        self.first = first
        self._compute_rest = compute_rest
        self._rest = None
    @property
    def rest(self):
        """Return the rest of the stream, computing it if
        necessary."""
        if self._compute_rest is not None:
            self._rest = self._compute_rest()
            self._compute_rest = None
        return self._rest
```

"Please don't reference directly"

Integer Streams

An integer stream is a stream of consecutive integers

An integer stream starting at *k* consists of *k* and a function that returns the integer stream starting at *k+1*

```
def integer_stream(first=1):
    """Return a stream of consecutive integers, starting
    with first.

    >>> s = integer_stream(3)
    >>> s.first
    3
    >>> s.rest.first
    4
    """
    def compute_rest():
        return integer_stream(first+1)
    return Stream(first, compute_rest)
```

A Stream of Primes

The stream of integers not divisible by any *k* <= *n* is:

- The stream of integers not divisible by any *k* < *n*,
- Filtered to remove any element divisible by *n*
- This recurrence is called the *Sieve of Eratosthenes*

2, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, ~~9~~, ~~10~~, 11, ~~12~~, 13

```
def primes(istream):
    """Return a stream of primes, given a stream of
    consecutive integers."""
    def compute_rest():
        not_divisible = lambda x: x % istream.first != 0
        return primes(filter_stream(not_divisible,
                                    istream.rest))
    return Stream(istream.first, compute_rest)
```

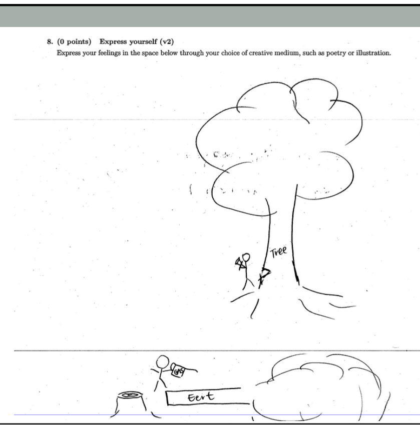
Try it

- Write a function `add_streams` that takes two streams and returns a new stream formed by summing corresponding elements in the argument streams. Stop when either of the streams ends.
- Bonus: see if you can use `add_streams` to define to define the Fibonacci stream!

Answers

```
def add_streams(s1, s2):
    if s1 is Stream.empty or s2 is Stream.empty:
        return Stream.empty
    return Stream(s1.first + s2.first,
                  lambda: add_streams(s1.rest, s2.rest))

fibs = Stream(0,
              lambda: Stream(1,
                             lambda: add_streams(fibs, fibs.rest)))
```

Short Break



Last “super-big” topic in course

- Lot to cover this lecture...
- We will continue this topic tomorrow as well
- Need to finish questions 1-4 on your Scheme project for Logic programming to work
- Bring your version of scheme to lab tomorrow!

Databases

A database is a collection of records (tuples) and an interface for adding, editing, and retrieving records

The Structured Query Language (SQL) is perhaps the most widely used programming language on Earth

```
SELECT * FROM toy_info WHERE color='yellow';
```

toy_id	toy	color	cost	weight
2	whiffleball	yellow	2.20	0.40
5	frisbee	yellow	1.50	0.20
10	yoyo	yellow	1.50	0.20

SQL is an example of a declarative programming language.

It separates *what* to compute from *how* it is computed

The language interpreter is free to compute the result in any way it deems appropriate

http://www.headfirstlabs.com/sql_hands_on/

Declarative Programming

The main characteristics of declarative languages:

- A "program" is a description of the desired solution
- The interpreter figures out how to generate such a solution

By contrast, in procedural languages such as Python & Scheme:

- A "program" is a description of procedures
- The interpreter carries out execution/evaluation rules

Building a universal problem solver is a difficult task

Declarative programming languages compromise by solving only a subset of all problems

They typically trade off data scale for problem complexity

The Logic Language

The *Logic* language is invented for this course

- Based on the Scheme project & ideas from Prolog
- Expressions are facts or queries, which contain relations
- Expressions and relations are both Scheme lists
- For example, **`(likes Albert dogs)`** is a relation
- Implementation fits on a single sheet of paper

Today's theme:

http://awhimsicalbohemian.typepad.com/.a/6a00e5538b84f3883301538dfa8f19970b-800wi

Simple Facts

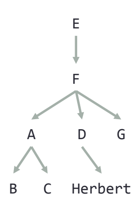
A simple fact expression in the *Logic* language declares a relation to be true

Let's say I want to track my many dogs' ancestry

Language Syntax:

- A relation is a Scheme list
- A fact expression is a Scheme list containing **`fact`** followed by one or more relations

```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
logic> (fact (parent fillmore delano))
logic> (fact (parent fillmore grover))
logic> (fact (parent eisenhower fillmore))
```



Relations are Not Procedure Calls

In *Logic*, a relation is not a call expression

- In Scheme, we write **`(abs -3)`** to call **`abs`** on **`-3`**
- In *Logic*, **`(abs -3 3)`** asserts that the **`abs`** of **`-3`** is **`3`**

For example, if we wanted to assert that **`1 + 2 = 3`**:

```
(add 1 2 3)
```

Why declare knowledge in this way? It will allow us to solve problems in two directions:

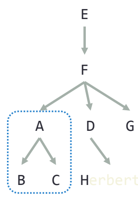
```
(add 1 2 _)
(add _ 2 3)
(add 1 _ 3)
(___ 1 2 3)
```

Queries

A query contains one or more relations. The *Logic* interpreter returns whether (and how) they are all simultaneously satisfied

Queries may contain variables: symbols starting with **?**

```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
logic> (fact (parent fillmore delano))
logic> (fact (parent fillmore grover))
logic> (fact (parent eisenhower fillmore))

logic> (query (parent abraham ?child))
Success!
child: barack
child: clinton
```

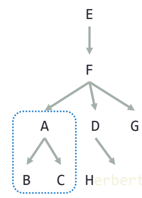


Queries

A query contains one or more relations. The *Logic* interpreter returns whether (and how) they are all simultaneously satisfied

Queries may contain variables: symbols starting with **?**

```
logic> (fact (parent delano herbert))
logic> (fact (parent abraham barack))
logic> (fact (parent abraham clinton))
logic> (fact (parent fillmore abraham))
logic> (fact (parent fillmore delano))
logic> (fact (parent fillmore grover))
logic> (fact (parent eisenhower fillmore))

logic> (query (parent ?who barack)
              (parent ?who clinton))
Success!
who: abraham
```



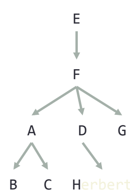
Compound Facts

A fact can include multiple relations and variables as well

```
(fact <conclusion> <hypothesis0> <hypothesis1> ... <hypothesisN>)
```

Means **<conclusion>** is true if all **<hypothesis$_K$>** are true

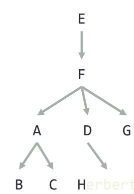
```
logic> (fact (child ?c ?p) (parent ?p ?c))

logic> (query (child herbert delano))
Success!

logic> (query (child eisenhower clinton))
Failure.

logic> (query (child ?child fillmore))
Success!
child: abraham
child: delano
child: grover
```



Recursive Facts

A fact is recursive if the same relation is mentioned in a hypothesis and the conclusion

```
logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

logic> (query (ancestor ?a herbert))
Success!
a: delano
a: fillmore
a: eisenhower

logic> (query (ancestor ?a barack)
              (ancestor ?a herbert))
Success!
a: fillmore
a: eisenhower
```



Searching to Satisfy Queries

The *Logic* interpreter performs a search in the space of relations for each query to find a satisfying assignment

```
logic> (query (ancestor ?a herbert))
Success!
a: delano
a: fillmore   ⇦
a: eisenhower

logic> (fact (parent delano herbert))
logic> (fact (parent fillmore delano))

logic> (fact (ancestor ?a ?y) (parent ?a ?y))
logic> (fact (ancestor ?a ?y) (parent ?a ?z) (ancestor ?z ?y))

(parent delano herbert)     ; (1), a simple fact
(ancestor delano herbert)   ; (2), from (1) and the 1st ancestor fact
(parent fillmore delano)    ; (3), a simple fact
(ancestor fillmore herbert) ; (4), from (2), (3), & the 2nd ancestor fact
```

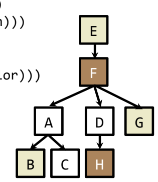
Hierarchical Facts

Relations can contain relations in addition to atoms

```
logic> (fact (dog (name abraham) (color white)))
logic> (fact (dog (name barack) (color tan)))
logic> (fact (dog (name clinton) (color white)))
logic> (fact (dog (name delano) (color white)))
logic> (fact (dog (name eisenhower) (color tan)))
logic> (fact (dog (name fillmore) (color brown)))
logic> (fact (dog (name grover) (color tan)))
logic> (fact (dog (name herbert) (color brown)))
```

Variables can refer to atoms or relations

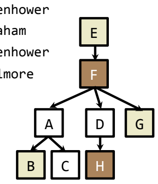
```
logic> (query (dog (name clinton) (color ?color)))
Success!
color: white

logic> (query (dog (name clinton) ?info))
Success!
info: (color white)
```



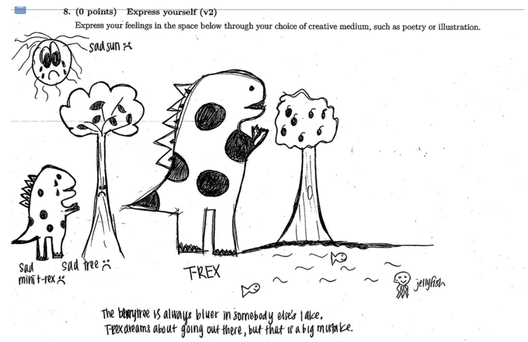
Example: Combining Multiple Data Sources

Which dogs have an ancestor of the same color?

```
logic> (query (dog (name ?name) (color ?color))
              (ancestor ?ancestor ?name)
              (dog (name ?ancestor) (color ?color)))
Success!
name: barack    color: tan     ancestor: eisenhower
name: clinton   color: white   ancestor: abraham
name: grover    color: tan     ancestor: eisenhower
name: herbert   color: brown   ancestor: fillmore
```



Break



Example: Appending Lists

Two lists append to form a third list if:

- The first list is empty and the second and third are the same

  `() (a b c) (a b c)`

- Both of the following hold:
  - List 1 and 3 have the same first element
  - The rest of list 1 and all of list 2 append to form the rest of list 3

  `(a b c) (d e f) (a b c d e f)`

```
logic> (fact (append-to-form () ?x ?x))

logic> (fact (append-to-form (?a . ?r) ?y (?a . ?z))
             (append-to-form ?r ?y ?z))
```

Logic Example: Anagrams

A permutation (i.e., anagram) of a list is:

- The empty list for an empty list
- The first element of the list inserted into an anagram of the rest of the list

Element | List | List with element

```
(fact (insert ?a ?r (?a . ?r)))

(fact (insert ?a (?b . ?r) (?b . ?s))
      (insert ?a      ?r        ?s))

(fact (anagram () ()))

(fact (anagram (?a . ?r) ?b)
      (insert   ?a   ?s  ?b)
      (anagram  ?r   ?s))
```

```
a|r t

 r t
ar t
 rat
 r ta

 t r
at r
 tar
 t ra
```

You try it out!

- Write facts to make double-elements work

logic> (query (double-elements (3 4) ?result))
Success!
result: (3 3 4 4)
logic> (query (double-elements ?start (4 4 5 5)))
Success!
start: (4 5)