STREAMS AND DECLARATIVE PROGRAMMING 12

COMPUTER SCIENCE 61A

November 20, 2014

1 Streams

A *stream* is a lazily-evaluated linked list. A stream's elements (except for the first element) are only computed when those values are needed.

```
class Stream:
    class empty:
        "An empty stream"
    empty = empty()

    def __init__(self, first, compute_rest=lambda: Stream.empty):
        assert callable(compute_rest), 'must be a function'
        self.first = first
        self._compute_rest = compute_rest

    @property
    def rest(self):
        """Return the rest, computing it if necessary."""
        if self._compute_rest is not None:
            self._rest = self._compute_rest()
            self._compute_rest = None
        return self._rest
```

A `Stream` instance is similar to a `Link` instance. Both have `first` and `rest` attributes. The rest of a `Link` is either a `Link` or `Link.empty`. Likewise, the rest of a `Stream` is either a `Stream` or `Stream.empty`.

However, instead of specifying all of the elements in `__init__`, we provide a function, `compute_rest`, that will be called to compute the remaining elements of the stream. Remember that the code in the function body is not evaluated until it is called, which lets us implement the desired evaluation behavior.

This implementation of streams also uses *memoization*. The first time a program asks a `Stream` for its `rest` field, the `Stream` code computes the required value using `compute_rest`, saves the resulting value, and then returns it. After that, every time the `rest` field is referenced, the stored value is simply returned.

Here is an example:

```
def make_integer_stream(first=1):
    def compute_rest():
        return make_integer_stream(first+1)
    return Stream(first, compute_rest)
```

Here, we start out with a stream whose first element is 1, and whose `compute_rest` function creates another stream. So when we do compute the `rest`, we get another stream whose first element is one greater than the previous element, and whose `compute_rest` creates another stream. Hence, we effectively get an infinite stream of integers, computed one at a time. This is almost like an infinite recursion, but one which can be viewed one step at a time, and so does not crash.

1.1 Questions

1. Suppose one wants to define a random infinite stream of numbers via the recursive definition: "a random infinite stream consists of a first random number, followed by a remaining random infinite stream." Consider an attempt to implement this via the code. Are there any problems with this? How can we fix this?

```
from random import random
random_stream = Stream(random(), lambda: random_stream)
```

Solution: The provided code will generate a single random number, and then produce an infinite stream which simply repeats that one number over and over. To fix this, we can make this into a function that returns a Stream:

```
def random_stream():
    return Stream(random(), random_stream)
```

2. Write a function `take_odd`, which takes in an infinite stream and returns a stream containing its odd indexed elements.

```
def take_odd(s):
```

Solution:

```
def take_odd(s):
    return Stream(s.rest.first, lambda: take_odd(s.rest.
       rest))
```

1.2 Extra Questions

1. Write a function `fib_stream` that creates an infinite stream of Fibonacci Numbers, using the `add_streams` function that was introduced in lab.

```
def fib_stream():
```

Solution:

```
def fib_stream():
    def compute_rest():
        return add_streams(fib_stream(),
                           fib_stream().rest)
    return Stream(0, lambda: Stream(1, compute_rest))
```

2. Write a function `seventh` that creates an infinite stream of the decimal expansion of dividing n by 7.

```
def seventh(n):
    """The decimal expansion of n divided by 7.

    >>> first_k(seventh(1), 10)
    [1, 4, 2, 8, 5, 7, 1, 4, 2, 8]
    """
```

Solution:

```
def seventh(n):
    """The decimal expansion of n divided by 7.
    >>> first_k(seventh(1), 10)
    [1, 4, 2, 8, 5, 7, 1, 4, 2, 8]
    """
    q, r = n*10 // 7, n*10 % 7
    return Stream(q, lambda: seventh(r))
```

1.3 Higher-Order Functions on Streams

Stream processing functions can be higher-order, abstracting a general computational process over streams. Take a look at `filter_stream`:

```
def filter_stream(filter_func, s):
    def make_filtered_rest():
        return filter_stream(filter_func, s.rest)

    if s is Stream.empty:
        return s
    elif filter_func(s.first):
        return Stream(s.first, make_filtered_rest)
    else:
        return filter_stream(filter_func, s.rest)
```

The Stream we create has as its `compute_rest` a function that "promises" to filter the rest of the Stream when called. So at any one point, the entire stream has not been filtered. Instead, only the part that has been referenced has been filtered.

1.4 Questions

1. What does the following Stream output? Try writing out the first few values of the stream to see the pattern.

```
def my_stream():
    def compute_rest():
        return add_streams(map_stream(double, my_stream()),
                           my_stream())
    return Stream(1, compute_rest)
```

Solution: Powers of 3: 1, 3, 9, 27, 81, ...

2. (Summer 2012 Final) What are the first five values in the following stream?

```
def my_stream():
    def compute_rest():
        return add_streams(filter_stream(lambda x: x % 2 == 0,
               my_stream()), map_stream(lambda x: x + 2,
                  my_stream()))
    return Stream(2, compute_rest)
```

Solution: 2, 6, 14, 30, 62

2 Declarative Programming

Over the semester, we have been using *imperative programming* – a programming style where code is written as a set of instructions for the computer. In this section, we introduce *declarative programming* – code that declares *what* we want, not *how* to compute it.

2.1 The SQL Language

SQL is an example of a declarative programming language. Statements do not describe computations directly, but instead describe the desired result of some computation. It is the role of the query interpreter of the database system to design and perform a computational process to produce such a result.

A table, also called a relation, has a fixed number of named and typed columns. Each row of a table represents a data record and has one value for each column. For example, we have a table named `records` that stores information about the employees at a small company[1]:

Name	Division	Title	Salary	Supervisor
Ben Bitdiddle	Computer	Wizard	60000	Oliver Warbucks
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
Cy D Fect	Computer	Programmer	35000	Ben Bitdiddle
Lem E Tweakit	Computer	Technician	25000	Ben Bitdiddle
Louis Reasoner	Computer	Programmer Trainee	30000	Alyssa P Hacker
Oliver Warbucks	Administration	Big Wheel	150000	Oliver Warbucks
DeWitt Aull	Administration	Secretary	25000	Oliver Warbucks
Eben Scrooge	Accounting	Chief Accountant	75000	Oliver Warbucks
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

2.2 Select statements

A select statement defines a new table either by listing the values in a single row, for example,

```
sqlite> select "Ben" as first, "Bitdiddle" as last;
Ben|Bitdiddle
```

[1]Example adapted from Structure and Interpretation of Computer Programs

Or, more commonly, we can project an existing table using a from clause, for example,

```
sqlite> select * from records where name = "Ben Bitdiddle";
Ben Bitdiddle|Computer|Wizard|60000|Oliver Warbucks
```

We can choose which columns to show, filter using a `where` clause, and sort with an `order by` clause.

```
select [columns] from [names] where [condition] order by [order]
```

The following statement lists the names and salaries of each employee under the accounting division, sorted in descending order by their salaries.

```
sqlite> select name, salary from records
   ...>   where division = "Accounting" order by -salary;
Eben Scrooge|75000
Robert Cratchet|18000
```

2.3 Questions

1. Write a query that outputs the names of employees that Oliver Warbucks directly supervises.

   Solution:

   ```
   select name from records where supervisor = "Oliver Warbucks";
   ```

2. Write a query that outputs all information about self-supervising employees.

   Solution:

   ```
   select * from records where name = supervisor;
   ```

3. Write a query that outputs the names of all employees with salary greater than 50000 in alphabetical order.

   Solution:

   ```
   select name from records where salary > 50000 order by name;
   ```

2.4 Joins

Suppose we have another table `meetings` which records the divisional meetings.

Division	Day	Time
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Thursday	1pm

Data are combined by joining multiple tables together into one, a fundamental operation in database systems. There are many methods of joining, all closely related, but we will focus on just one method in this class. When tables are joined, the resulting table contains a new row for each combination of rows in the input tables. If two tables are joined and the left table has m rows and the right table has n rows, then the joined table will have $m \cdot n$ rows. Joins are expressed in SQL by separating table names by commas in the `from` clause of a `select` statement.

```
sqlite> select name, day from records, meetings;
Alyssa P Hacker|Monday
...
Ben Bitdiddle|Monday
...
```

Tables may have overlapping column names, and so we need a method for disambiguating column names by table. A table may also be joined with itself, and so we need a method for disambiguating tables. To do so, SQL allows us to give aliases to tables within a `from` clause using the keyword `as` and to refer to a column within a particular table using a dot expression. In the example below we find the name and title of Louis Reasoner's supervisor.

```
sqlite> select b.name, b.title from records as a, records as b
   ...>    where a.name = "Louis Reasoner" and
   ...>          a.supervisor = b.name;
Alyssa P Hacker|Programmer
```

2.5 Questions

1. Write a query that creates a table with columns: employee, salary, supervisor and supervisor's salary, containing all supervisors who earn more than twice as much as the employee.

   Solution:

```
select e.name, e.salary, s.name, s.salary
  from records as e, records as s
  where e.supervisor = s.name and e.salary * 2 < s.salary;
```

2. Write a query that outputs the names of employees whose supervisor is in a different division.

Solution:

```
select e.name from records as e, records as s
  where e.supervisor = s.name and e.division != s.division;
```

3. Write a query that outputs the meeting days and times of all employees directly supervised by Oliver Warbucks.

Solution:

```
select m.day, m.time from records as r, meetings as m
  where r.division = m.division and
        r.supervisor = "Oliver Warbucks";
```

2.6 Extra Questions

1. A middle manager is a person who is both supervising someone and is supervised by someone different. Write a query that outputs the names of all middle managers.

Solution:

```
select b.name from records as a, records as b
  where a.supervisor = b.name and b.supervisor != b.name;
```

2. What is the output of the query in the previous part? Explain the output you get.

Solution:

```
Alyssa P Hacker
Ben Bitdiddle
Ben Bitdiddle
Ben Bitdiddle
```

There are multiple people with Ben Bitdiddle as supervisor, and joining tables together does not remove these duplicates.

3. Write a query that results in the names of all employees that have a meeting on the same day as their supervisor.

Solution:

```
select e.name from records as e, records as s, meetings as em, meetings
  where e.supervisor = s.name and em.day = sm.day and
        e.division = em.division and s.division = sm.division;
```