CS61A Lecture 26
Logic Programming

Jom Magrotker
UC Berkeley EECS
August 1, 2012

Computer Science in the News

Sing for the win

A new app game developed by a Cambridge student challenges people to sing the right note at the right time in order to smash down a wall and advance to the next level - surreptitiously engaging them with basic music theory at the same time.

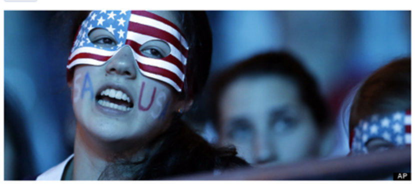
http://www.cam.ac.uk/research/news/sing-for-the-win/

Computer Science in the News

LA City Workers' Olympics Viewing Is Causing A Municipal Computer Meltdown

08/01/12 01:55 AM ET AP

Like 157 people like this. Be the first of your friends.



http://www.huffingtonpost.com/2012/08/01/la-city-workers-olympics-_n_1726878.html

---

Today

- Review: Streams
- Logic programming
- PyGic

Review: Streams

What are the first five elements of the stream

```
s = Stream(1,
           lambda:
           Stream(3,
                  lambda:
                  mul_streams(s,
                              s.rest)))
```

Review: Streams

The stream we have to determine

s 1 3

s 1 3
s.rest 3

The streams it depends on

Review: Streams

```
s          1  3  3
s                1  3  3
s.rest           3  3
```

When you add to the main stream, update all the other streams that depend on it!

Review: Streams

```
s          1  3  3  9
s                1  3  3  9
s.rest           3  3  9
```


Review: Streams

```
s          1  3  3  9  27 ...
s                1  3  3  9
s.rest           3  3  9
```


Imperative Programming

In most of our programs so far, we have described *how to* compute a certain value or to perform a certain task.

```
def abs(x):
    if x >= 0:
        return x
    return -x
```

- `def abs(x):` — To find the absolute value of x...
- `if x >= 0:` — ... Check if x is greater than, or equal to, zero.
- `return x` — If so, return the number itself.
- `return -x` — Otherwise, return the negative of the number.

Imperative Programming

We found an *algorithm* to solve a problem, and we wrote the corresponding program that told the computer to follow a series of steps.

This is known as ***imperative programming***.

The Problem with Imperative Programming

Sometimes, we do not know exactly ***how to*** solve a problem, but we do know ***what*** the solution ***is***.

For example, what is the square root of a number y?
It is a number x such that $x^2 = y$.

The Problem with Imperative Programming

The definition of a square root does not tell us ***how*** to find it, but it does tell us ***what*** it is.

Of course, we have many methods to find square roots: we can use Newton's method, or we can square all numbers smaller than y.

But, we would still be telling the computer ***how*** to solve the problem!

13

The Problem with Imperative Programming

Problem:

Can we express our problems in a way similar to how they were defined?

Can we provide a set of definitions and descriptions of ***what*** we want, and leave it to the computer to deduce ***how*** to solve it?

http://i0.kym-cdn.com/photos/images/original/000/234/765/b7e.jpg

14

Announcements

- Homework 13 due **Saturday, August 4**.
  - Includes Py, streams, iterators, and generators
  - Also includes the Project 4 contest.
- Project 4 due **Tuesday, August 7**.
  - Partnered project, in two parts.
  - Twelve questions, so *please start early*!
  - Two extra credit questions.
- De-stress potluck on **Thursday, August 2** from **7pm to 10pm** in the **Wozniak Lounge** (Soda, 4th floor).
  - Food and games.
  - Come and leave when you want.

15

Announcements: Final

- Final is **Thursday, August 9**.
  - *Where*? 1 Pimentel.
  - *When*? 6PM to 9PM.
  - *How much*? *All* of the material in the course, from June 18 to August 8, will be tested.
- Closed book and closed electronic devices.
- One 8.5" x 11" 'cheat sheet' allowed.
- No group portion.
- We will get back to you today if you have conflicts and have told us. If you haven't told us yet, please *let us know*.

16

Declarative Programming

New paradigm!

In ***declarative programming***, we describe ***what*** the properties of the required solution are, and the computer discovers how to find the solution.

For example, we give a computer the definition of the square root of a number, and it discovers how to find the square root of the number.

17

Logic Programming

In general, declarative programming is very hard and is an active field of research.

Logic programming is a type of declarative programming that uses mathematical logic and logical inference to solve a problem.

18

Logic Programming

One of the most commonly used logic programming languages is Prolog.

SWI Prolog

Prolog was designed in 1972 by Alain Colmerauer for use in natural language processing.

We will use and study a very simplified Python-based version of Prolog.

http://upload.wikimedia.org/wikipedia/commons/f/f8/Swipl.png

PyGic

PyGic is a logic programming language, with design inspired by Prolog, but with a Python-like syntax.

Jon coded it and is pretty proud of it. ☺

Stephen Martinis and Eric Kim also helped.

We will use it to explore some of the principles behind logic programming.

PyGic: Facts

We will start by stating the ***facts*** that we know.

```
P?> fact likes(jon, ice_cream)
Yes.
P?> fact likes(tom, ice_cream)
Yes.
P?> fact likes(jon, coffee)
Yes.
P?> fact father(james, harry)
Yes.
P?> fact mother(lily, harry)
Yes.
P?> fact father(harry, albus_severus)
Yes.
P?> fact father(harry, james_sirius)
Yes.
```

The interpreter will only respond with `Yes.` or `No.`

The `Yes.` here indicates that the interpreter has accepted our fact.

PyGic: Facts

A *fact* allows us to tell the computer what we know to be true, and what it can build up from.

It *looks* like a function, but ***it is not.***

A fact establishes a ***relation*** between different objects. It does *not* compute a value, whereas a function would.

PyGic: Queries and Variables

We can now perform simple *queries*.

```
P?> likes(jon, ?what)
Yes.
?what = ice_cream
```

Variables are denoted with a question mark at the front.

Variables indicate things we do not know. Here, for example, we do not know what Jon likes.

PyGic: Queries and Variables

We can now perform simple *queries*.

```
P?> likes(jon, ?what)
Yes.
?what = ice_cream
```

The interpreter is able to find a match among the facts we gave it!

It finds that if `?what` were replaced by `ice_cream`, it gets a fact.

PyGic: Queries and Variables

```
P?> more?
Yes.
?what = coffee
P?> more?
No.
```

Are there any more things that Jon likes?

The interpreter finds another match!

Are there any more things that Jon likes?

The interpreter cannot find any more matches.

25

PyGic: Queries and Variables

We assert facts and the interpreter stores them in a database.

When we perform simple queries, we use variables to denote values that we do not know.

The interpreter then *matches* the query against the facts that it knows. If it finds a match, it shows the values for the variables in that match.

26

PyGic: Queries and Variables

The interpreter can only match against the facts that it knows. It *cannot* match against facts that have not been established.

Jon could like other things, but the interpreter does not know about these things.

This is the ***closed world assumption***: the interpreter only knows what is given to it.

27

PyGic: Queries and Variables

The interpreter finds all possible matches to your query, and you can have as many variables as you will need in your query.

For example, what would the interpreter respond to the following query?

```
P?> likes(?who, ?what)
```

28

PyGic: Unification

The interpreter matches the pattern of the query to the patterns of the facts in its database.

This process of pattern matching is called ***unification***.

29

PyGic: Lists

We can also represent *lists* in PyGic.

```
P?> less_than_4(<1, 2, 3>)
Yes.
```

This is not the number 3! This is the string of one character "3". Numeric strings are *not* evaluated to the corresponding numbers.

Here, we assert a fact about a list of three ***symbols*** that represent natural numbers.

30

PyGic: Lists

We can perform simple queries on lists.

```
P?> less_than_4(<1, 2, ?x>)
Yes.
?x = 3
P?> more?
No.
```


PyGic: Lists

We can perform simple queries on lists.

```
P?> less_than_4(<1, 2, 3>)
Yes.
P?> less_than_4(<1, ?what>)
No.
P?> less_than_4(?what)
Yes.
?what = <1, 2, 3>
```


PyGic: Lists

Why are lists denoted by angle brackets (<>)?

PyGic lists are (internally) represented as *recursive lists* (RLists).

Many of our queries (and later, rules) will deal with the *first* and the *rest* of our lists separately.

PyGic: Lists

We use the "split" (|) character to separate the first of the list from the rest of the list.

```
P?> less_than_4(<?first | ?rest>)
Yes.
?first = 1
?rest = <2, 3>
```

Remember that these lists are recursively defined, so the list `<1, 2, 3>` is actually a pair whose first element is `1`, and whose second element (the "rest") is the list `<2, 3>`.

PyGic: Lists

We assert the following fact:

```
P?> fact favorite_things(<raindrops, roses, whiskers>)
```

What will the interpreter respond to the following queries:

```
P?> favorite_things(<?first | ?rest>)

P?> favorite_things(<?first, ?second | ?rest>)

P?> favorite_things(<?first, ?second, ?third>)

P?> favorite_things(<?f, ?s, ?t | ?fourth>)

P?> favorite_things(<?first>)
```


PyGic: Lists

We assert the following fact:

```
P?> fact favorite_things(<raindrops, roses, whiskers>)
```

What will the interpreter respond to the following queries:

```
P?> favorite_things(<?first | ?rest>)
?first = raindrops, ?rest = <roses, whiskers>
P?> favorite_things(<?first, ?second | ?rest>)
?first = raindrops, ?second = roses, ?rest = <whiskers>
P?> favorite_things(<?first, ?second, ?third>)
?first = raindrops, ?second = roses, ?third = whiskers
P?> favorite_things(<?f, ?s, ?t | ?fourth>)
?f = raindrops, ?s = roses, ?t = whiskers, ?fourth = <>
P?> favorite_things(<?first>)
No.
```

PyGic: Rules

Great! We can now assert facts and run queries on those facts.

PyGic will then attempt to match the *pattern* of the query against all of the facts, and will show those that match the pattern.

But, PyGic is more than a simple pattern matcher!

37

PyGic: Rules

PyGic can also *infer* a fact from other facts through user-defined ***rules***.

?person is the grandfather of ?grandson if...

```
P?> rule grandfather(?person, ?grandson):
```

... ?person is the father of ?son and...

```
...     father(?person, ?son)
```

... ?son is the father of ?grandson.

```
...     father(?son, ?grandson)
```

38

PyGic: Rules

These rules allow the interpreter to infer other facts that we did not initially specify:

```
P?> grandfather(james, ?who)
Yes.
?who = james_sirius
```

39

PyGic: Rules

These rules allow the interpreter to infer other facts that we did not initially specify:

```
P?> grandfather(?who, ?grandson)
Yes.
?who = james
?grandson = james_sirius
```

40

PyGic: Rules

Notice that all we did was define ***what*** a grandfather-grandson relationship was in terms of two father-son relationships.

We did not tell the interpreter ***how*** to determine a grandfather-grandson relationship.

PyGic uses *logical inference* to establish new facts.

41

PyGic: Rules

Conclusion

```
P?> rule grandfather(?person, ?grandson):
...     father(?person, ?son)
...     father(?son, ?grandson)
```

Hypotheses

The conclusion is true *only if* the hypotheses are true.

=

Can variables be replaced with values such that the hypotheses are true? If so, the conclusion is true too.

42

PyGic: Rules

CONCLUSION

```
P?> rule grandfather(?person, ?grandson):
...     father(?person, ?son)
...     father(?son, ?grandson)
```

HYPOTHESES

A hypothesis is *true* if there is a fact that matches it, *or* there is a true conclusion for another rule that matches it.

43

PyGic: Rules

If the interpreter can find values for variables `?person`, `?son`, and `?grandson`, such that the two hypotheses
`father(?person, ?son)`
`father(?son, ?grandson)`
match facts or conclusions of other rules, *then* the conclusion
`grandfather(?person, ?grandson)`
is true.

If the hypotheses match the conclusions of other rules, the truth value for the rules is checked recursively.

44

PyGic: Rules

Remember that these ***are not*** functions!
They ***cannot*** be composed.

```
P?> rule grandfather(?person, ?grandson):
...     father(?person, father(?grandson))
```

The above rule will ***not*** work.
There is no fact or rule that matches the hypothesis.

45

Break

http://www.bonkersworld.net/the-solution-to-oversleeping/
http://invisiblebread.com/2012/05/the-best-alarm/

46

PyGic: Rules for Lists

We can also define rules for lists.

For example, say we want to check if two (flat) lists are equal.

What are some facts we know about equal lists?

47

PyGic: Rules for Lists

Fact 1:
The empty list is only equal to itself.

```
P?> fact equal_lists(<>, <>)
```

A fact, by the way, is equivalent to a rule with `True` in the body:

```
P?> rule equal_lists(<>, <>):
...     True
```

True is a PyGic keyword that is, well, true.

48

PyGic: Rules for Lists

Fact 2:

Conclusion

Two lists are equal if

their first elements are equal, and if

the rest of their elements are equal.

Hypotheses

```
P?> rule equal_lists(<?x | ?rest1>, <?x | ?rest2>):
...      equal_lists(?rest1, ?rest2)
```

The same variable is used in two places. A list can therefore only match if the first elements have the same value

PyGic: Rules for Lists

We can now run queries as before:

```
P?> equal_lists(<1, 2, 3>, <1, 2, 3>)
Yes.
P?> equal_lists(<1, 2, 3>, <1, 2>)
No.
```


PyGic: Rules for Lists

But, we can now also find variables that *satisfy* our queries!

```
P?> equal_lists(<1, 2, 3>, <1, 2, ?what>)
Yes.
?what = 3
P?> equal_lists(<1, 2, 3>, ?what)
?what = <1, 2, 3>
```


PyGic: Rules for Lists

In the previous example, all we had to do was specify facts and rules about the equality of lists.

The interpreter then used these rules to not only check if two lists were equal, but also infer what variables could *make* two lists equal.

We will see how this works in more detail tomorrow: today, we will focus more on solving problems with logic programming.

PyGic: Rules for Lists

Another example: how do we check if an element is a member of a list?

What are some facts that we know about an element that is a member of a list?

If an element is a member of a list,

it must either be the first element, or

it must a member of the rest of the list.

PyGic: Rules for Lists

If an element is a member of a list, it must either be the first element:

```
P?> fact member(?x, <?x | _>)
```

We can ignore the rest of the list.

or it must be a member of the rest of the list:

```
P?> rule member(?x, <_ | ?rest>):
...      member(?x, ?rest)
```

We can ignore the first element of the list.

PyGic: Rules for Lists

We can run queries as before:

```
P?> member(2, <2, 3, 4>)
Yes.
P?> member(2, <3, 4>)
No.
P?> member(3, <>)
No.
```


PyGic: Rules for Lists

We can also find values for variables that satisfy our queries!

```
P?> member(?what, <2, 3>)
Yes.
?what = 2
P?> more?
Yes.
?what = 3
P?> more?
No.
```


PyGic: Rules for Lists (Practice)

We want to append one list to another:

```
P?> append(<1, 2, 3>, <4, 5>, <1, 2, 3, 4, 5>)
Yes.
P?> append(<1, 2, 3>, <4, 5>, ?what)
Yes.
?what = <1, 2, 3, 4, 5>
```


PyGic: Rules for Lists (Practice)

What are two facts that we can state about the problem?

Fact 1: Appending the empty list to any other list gives us the ________.

Fact 2: Appending one list to another is equivalent to adding the first element of the first list to the result of ____.

What facts or rules should we then define?

PyGic: Rules for Lists (Practice)

What are two facts that we can state about the problem?

Fact 1: Appending the empty list to any other list gives us the **other list**.

Fact 2: Appending one list to another is equivalent to adding the first element of the first list to the result of **appending the rest of the first list to the second list**.

What facts or rules should we then define?

PyGic: Rules for Lists (Practice)

```
P?> fact append(<>, ?z, ?z)
P?> rule append(<?x | ?u>, ?v, <?x | ?w>):
...     append(?u, ?v, ?w)
```

PyGic: Rules for Lists (Practice)

We can now run append “backwards”.

```
P?> append(<1, 2, 3>, ?what, <1, 2, 3, 4, 5>)
Yes.
?what = <4, 5>
```


PyGic: Other Useful Statements

- `listing(rule_name=None)`

Prints the rules with the same name as that provided. If no rule name is provided, all rules in the database are printed.

- `clear(rule_name=None)`

Removes the rules with the given name. If no rule name is provided, all rules in the database are removed.

Conclusion

- *Declarative programming* is a programming paradigm where the computer is presented with certain facts and rules about a problem, from which the computer must then deduce the solution.
- *Logic programming* is a type of declarative programming that uses logical inference.
- This is in contrast to *imperative programming*, where the computer is told *how* to do a problem.
- ***Preview***: What is happening under the hood?