CS61A Lecture 26
Logic Programming

Jom Magrotker
UC Berkeley EECS
August 1, 2012

COMPUTER SCIENCE IN THE NEWS

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)))))

The stream we have to determine

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

**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*.

**IMPERATIVE PROGRAMMING**

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

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

To find the absolute value of \( x \):
- Check if \( x \) is greater than, or equal to, zero.
- If so, return the number itself.
- Otherwise, return the negative of the number.

**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!

---

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.

---

DECLARATIVE PROGRAMMING

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.

---

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.

---

ANNOUNCEMENTS: FINAL

- Final is Thursday, August 9.
  - Where? 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.
LOGIC PROGRAMMING

One of the most commonly used logic programming languages is 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.

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.

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

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.

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.

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.

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)

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.

PYGIC: LISTS

We can also represent lists in PyGic.

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

Here, we assert a fact about a list of three symbols that represent natural numbers.
We can perform simple queries on lists.

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

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.

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

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

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

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

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

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

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.

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.
PYGIC: RULES

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

PYGIC: RULES

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

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

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?

PYGIC: RULES FOR LISTS

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

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

True is a PyGIC keyword that is, well, true.
**PyGIC: Rules for Lists**

**Fact 2:**
Two lists are equal if their first elements are equal, and if the rest of their elements are equal.

P?> rule equal_lists(?rest1, ?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**

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**

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**

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**

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

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

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

P?> rule member(?x, <? | ?rest>):
  ... member(?x, ?rest)
We can run queries as before:

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

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

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

We want to append one list to another:

```prolog
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>
```

What 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?

```prolog
P?> fact append(<> , ?z , ?z)
  ...  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?