1 Introduction

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 do it. Logic programming (what we are learning) is a type of declarative programming.

In this class, we will be using Pygic, designed to use syntax similar to the Python language.

NOTE: if you have not done Lab 14, do it as soon as possible! The best way to learn logic programming is to play around with it!

2 Facts

In Pygic, you can define facts and rules. Here’s an example of a fact:

P?> fact sells(supermarket, groceries)

This line of code says: “This is a fact: supermarkets sell groceries”. When we declare something as a fact, we are simply saying that it is a true statement.

“sells” is a quality that relates two things, “supermarket” and “groceries.” What are the values of “supermarket” and “groceries”? They have no values! They are symbols – symbols are Pygic’s primitives.

Now that we have defined facts, we can check if a statement is correct:
Having defined some facts, we can make queries – in other words, we can ask Pygic for information:

P?> sells(supermarket, ?stuff)
Yes.
?stuff = groceries

The query above is equivalent to asking “What do supermarkets sell?” Pygic replies that supermarkets sell groceries, based on the previously defined fact.

?stuff is a variable in Pygic, whereas supermarket is a symbol (a primitive). supermarket is always going to be supermarket, but ?stuff is unknown – it is only after the query that we know what the value of ?stuff is.

A similar query is

P?> sells(?place, groceries)
Yes.
?place = supermarket

which is equivalent to asking “Which places sell groceries?” Once again, Pygic replies based on the previously defined fact.

We can also query both parameters:

P?> sells(?place, ?stuff)
Yes.
?place = supermarkets
?stuff = groceries

This is equivalent to asking “What are places that sell stuff, and what stuff do they sell?” Pygic will tell you what each variable should be.

### 2.1 Questions

1. Write a fact that checks if two elements are equal.
2. Define a set of facts for a “mall,” which has the following qualities:

- malls sell shoes and clothes
- malls are larger than supermarkets
- malls are popular

3  Lists

Lists are Pygic’s built-in data structure. The syntax for a list is the following:

<1, 2, 3, 4>

We can use lists in facts and rules:

P?> fact append(<1, 2>, <3, 4>, <1, 2, 3, 4>)

This is equivalent to saying “appending <1, 2> and <3, 4> will result in <1, 2, 3, 4>”.

You can split lists by using the | operator. For example:

P?> fact combined(?x, ?y, <?x | ?y>)
Yes.
P?> combined(3, <2, 1>, ?result)
Yes.
?result = <3, 2, 1>

4  Rules

Pygic also has “rules,” which are just more complex facts. For example:

P?> rule sells_same(?store1, ?store2):
    sells(?store1, ?item)
    sells(?store2, ?item)

The idea of a rule is the following:

rule ```'conclusion'':
    ```'hypothesis1'`
This is equivalent to saying “the conclusion is true if all the hypotheses are true.” If even one of the hypotheses is false, the conclusion will also be false.

For example, the sells_same rule is equivalent to saying “store1 and store2 sell the same thing if store1 sells item and store2 also sells item.”

You can perform fact-checking with rules, just like with facts:

P?> fact sells(farmers_market, groceries)
Yes.
P?> fact sells(starbucks, coffee)
Yes.
P?> sells_same(supermarket, farmers_market)
Yes.
P?> sells_same(supermarket, starbucks)
No.

We can also do querying:

P?> sells_same(?store, supermarket)
Yes.
?store = farmers_market

This is equivalent to asking “what store sells the same thing as a supermarket?”

We can also ask “what stores sell the same thing?”

P?> sells_same(?store1, ?store2)
Yes.
?store1 = supermarket
?store2 = supermarket

That’s pretty obvious, but it is true nonetheless. Are there any other matches?

P?> more?
Yes.
?store1 = farmers_market
?store2 = supermarket

We use the more? command to ask Pygic if there are any more matches that satisfy the query. If there are, Pygic automatically returns the next match. If not, Pygic will return “No.”
4.1 Questions

1. Write facts and rules for every_other, a relation between two lists that is satisfied if and only if the second list is the same as the first list, but with every other element removed.

P?> every_other(<frodo, merry, sam, pippin>, ?x)
Yes.
?x = <frodo, sam>
P?> every-other(<gandalf>, ?x)
Yes.
?x = <gandalf>

2. Write rules for prefix, a relation between two lists that is satisfied if and only if elements of the first list are the first elements of the second list, in order.

P?> prefix(<being, for, the>, <being, for, the, benefit, of, mister, kite>)
Yes.
P?> prefix(<for, no, one>, <for, no, one>)
Yes.
P?> prefix(<>, <got, to, get, you, into, my, life>)
Yes.
P?> prefix(<want, i, to>, <i, want, to, hold, your, hand>)
No.
P?> prefix(<to, hold, your>, <i, want, to, hold, your, hand>)
No.
P?> prefix(<i, want, to, tell, you>, <i, want, to>)
No.
3. Write facts and rules for sublist, a relation between two lists that is satisfied if and only if the first is a consecutive sublist of the second. For example:

P?> sublist(<give>, <never, gonna, give, you, up>)
Yes.
P?> sublist(<you, up>, <never, gonna, give, you, up>)
Yes.
P?> sublist(<never, gonna, give>, <never, gonna, give, you, up>)
Yes.
P?> sublist(<>, <never, gonna, give, you, up>)
Yes.
P?> sublist(<never, give, up>, <never, gonna, give, you, up>)
No.
P?> sublist(<let, you, down>, <never, gonna, give, you, up>)
No.

Hint: You will want to use the prefix rule that you previously defined.
4. Write a set of rules to implement the `subs` relation with components `old`, `new`, `input`, and `output`. The first two are symbols; the last two can be symbols or lists. The output should be the same as the input except that every appearance of `old` is replaced by `new`.

```prolog
P?> subs(romeo, fred, <romeo, oh, romeo, why, art, thou, romeo>, ?x)
Yes.
?x = <fred, oh, fred, why, art, thou, fred>
```