Logic Programming: Totally Not Scheme!

1. What is Logic Programming?

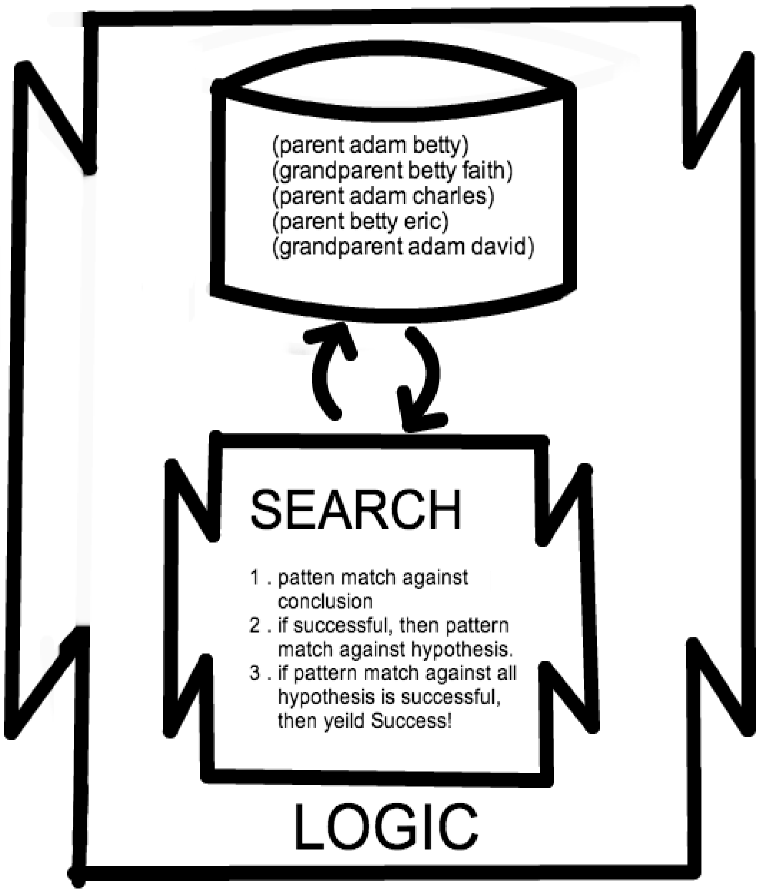
Logic is a type of **declarative programming**, where we ask the Logic Interpreter a question in the hopes that the program can answer it for us. How does that work?
Logic is a program already written to store information that we have inputed and uses only that information to figure out other things that we ask for it.

Think of Logic as your brain. As you learn information, you need to store that information as a series of **facts** into your brain. Then later on, when you need to recall the parts of the information that you can't remember or when you want to infer other connections from that information that is not explicitly stated (you'll learn more about this in the Recursion section), you can **query** your brain for the information.

Here's how to use Logic Programming:

1. We give Logic **facts** in the form of `(fact (relationship symbol_1 symbol_2 ...)),` where different **symbols** are tied together by a relationship.

2. If we are unsure of any part of a fact, then we **query** and replace that part with a word prepended by a question mark (this becomes a **variable**, which Logic tries to pattern match). The form becomes `(query (... ?variable symbol...))`. If Logic cannot pattern match the query to any of the facts, then it will print out a "Failed."


(parent adam betty)
(grandparent betty faith)
(parent adam charles)
(parent betty eric)
(grandparent adam david)
SEARCH
1 . patten match against conclusion
2 . if successful, then pattern match against hypothesis.
3 . if pattern match against all hypothesis is successful, then yeild Success!
LOGIC

2. Hierarchical Data

In order to depict more complicated ideas into our facts, we can include relations inside of relations!

For example, a person has more than one type of attribute. They have a name, an age, a height (in feet) and a weight (in lb). We can include the relations of these attributes inside of the relation of a person like this:

```
(fact (person (name name_symbol) (age age_symbol)
              (height height_symbol) (weight weight_symbol)))
```

We then can create the person who has a name J, age 20, height 4, and weight 129.

```
(fact (person (name J) (age 20) (height 4) (weight 129)))
```

Why is this so important?

What if we wanted to create a super complicated relation of people in different departments (where we listed all the people in a specific department, for example, the department of security)? We'd get something like this:

```
(fact (department-of-security
         (person (name J) (age 20) (height 4) (weight 129))
         (person (name K) (age 30) (height 5) (weight 100))
          ...))
```

Just by having this hierarchy (relations in relations), we are able to model more complicated relations.

3. Compound Facts

Facts may also contain variables as well as multiple sub-expressions. A multi-expression fact begins with a conclusion, followed by hypotheses. For the conclusion to be true, all of the hypotheses must be satisfied.

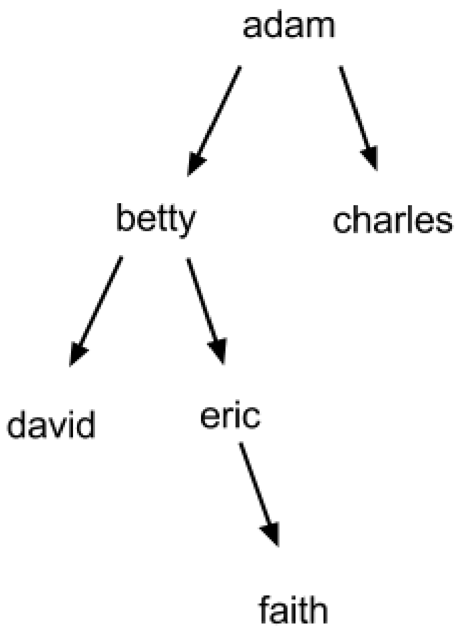
Here is the basic syntax of a compound fact:

```
logic> (fact (<conclusion>)
              (<hypothesis 1>)
              (<hypothesis 2>)
              ...
              (<hypothesis n>))
```

What can we do with this?
In the first example, we defined a series of family relationships

```
logic> (fact (parent adam betty))
logic> (fact (grandparent betty faith))
logic> (fact (parent adam charles))
logic> (fact (parent betty eric))
logic> (fact (grandparent adam david))
```

to create a genealogy like this:



Notice that we never stated that `adam` was the grandparent of `eric`. Can we create a series of facts so that we can determine the grandparents if nothing explicit is stated about it?

Well, when we trace the genealogy from adam to eric, we discover a middle person: `betty`. `Betty` is the parent of `eric` and the child of `adam`. Therefore, in order to find the grandparent, we must find a person Y who is the child of grandparent X and the parent of child Z.

This decomposes to 2 hypothesis:

```
logic>(fact (grandparent
?grandparent ?child)
            (parent ?parent ?child)
            (parent ?grandparent ?parent))
```

Now we can query Logic for this new grandparent relationship:

```
logic> (query (grandparent adam ?who))
```

```
Success!
?who: david
?who: eric
```

This is one of the powers of logic! It can infer from facts to confirm or create new information.

Questions:

1. In the previous examples, to trace genealogy, we asked for the grandparent or great-grandparent. Let's create a fact that finds any ancestor of a person.

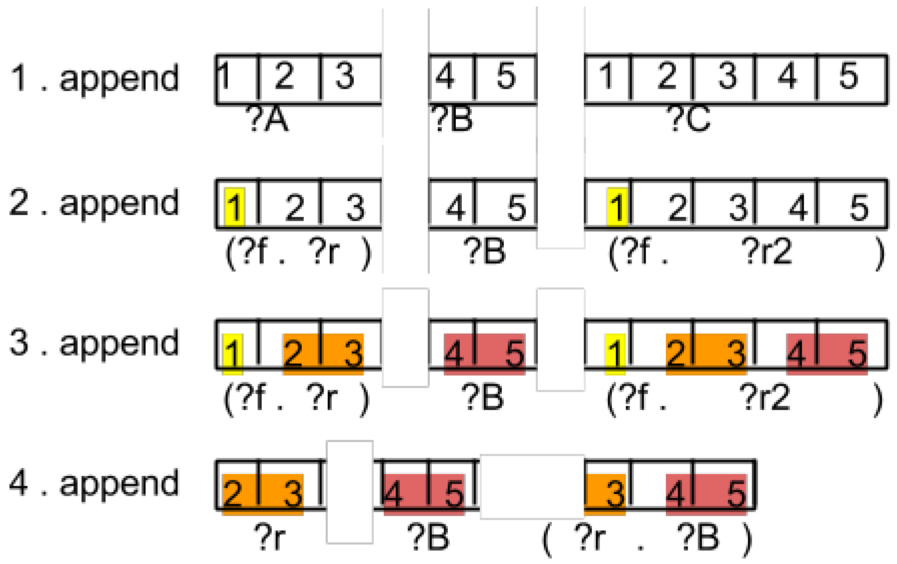
5. Recursive Facts

When we incorporate recursion into logic, we can solve harder problems and make logic infer more. One of the best ways to solve recursion is to decompose your problem, or break down the problem in a more tangible way. Drawing is one way of decomposing our problem.

For example, let's try to figure out append, which takes in two lists and concatenates them.
We have List `?A` and `?B` and we want to append them to make List `?C` (the "List"s we are referring to are Linked Lists but we'll be calling them "List" for short).

Appending is supposed to get the last item of A and link it to B. But we have no access to the last item of A. What should we do to solve this problem? **(Look to the below diagram to follow the steps below.)**

1. One way we can do this is to get the first of one of the lists and find someway to get more information about the problem. Let's start with list `?A`. To get the first of a list `?A`, we replace the list symbol with `(?f . ?r)`. ?f then becomes the first item of list `?A` and `?r` is the rest of the list that composes list `?A`.
2. If we think more about it, the first of `?A` should also be the first of `?C`! We can denote this by splitting list `?C` into `(?f . ?r2)`.
3. Now that we have a bit of information on `?C`, let's take a better look at it. What is the `?r2` made out of? We realize that the rest of `?C` is a concatenation of `?r` and `?B`. Thus, we can just append them together. Now we have our recursion!



In summary, we get

```
(fact (append (?f . ?r) ?B (?f . ?r2))
      (append ?r ?B ?r2))
```

But we can’t forget that base case!

```
(fact (append () ?B ?B))
```

Questions:

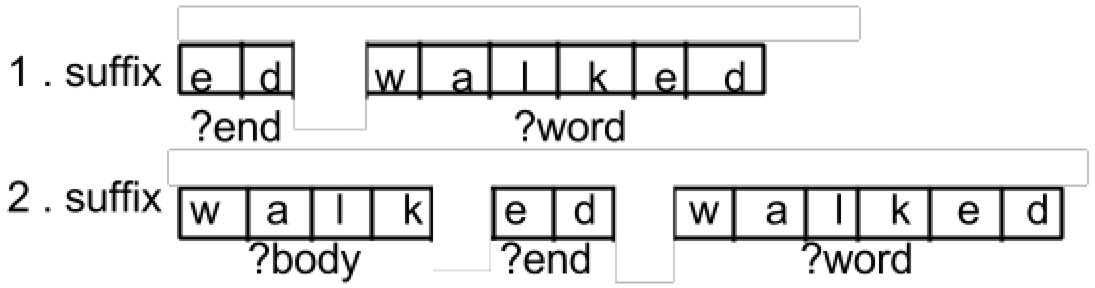
1. Can we write the base case also as `(fact (append ?lst () ?lst))`? Or does it not matter?

2. Write `suffix` that determines if `?end` is the suffix of `?body`.

6. The short cut

How did you write out `suffix`? Did you do it the tricky way or write it all out?
This section is all about how to solve list-based logic problems in a “shortcut way”.
Let’s try to understand what `suffix` is asking for.

`Suffix` is all about trying to see if a `?word` is composed of a `?body` and an `?end`. Isn't that what appending was all about (seeing if `?C` could be composed of `?A` and `?B`)? So let's rewrite it as `append` then.



```
(fact (suffix ?end ?body)
      (append ?body ?end ?word))
```

We just wrote `suffix` as another version of `append`!! Furthermore, if you wrote `suffix` the long way, you'll see that it's the same as `append`. A recurring theme of programming is to not do the same work twice. If you can use something you've already made, why not use it?

Questions:

1. Why didn't we need to write a base case for suffix using the shortcut method above.

2. Write delete that deletes an `?item` from `?list`.

3. Write `insert` that inserts an `?item` into a `?list`.

4. Write `anagram` that determines if `?word1` is an anagram of `?word2`. (HINT: YOU CAN USE `DELETE/INSERT`)

5. Write `middle` which determines if an `?item` is the middle element of odd_length `?list`.