Repetition and Structures

1 Repetition

Most of the examples so far have been pretty simple, and we have carefully avoided programs that employ any sort of repetition. Here we properly introduce repetition.

While many languages use loops for repetition, these are intentionally absent in Prolog. Since state cannot be mutated in Prolog, it is difficult to write a loop that does any actual real work. Instead, Prolog uses recursion to perform repetition. This is made possible with recursively-defined rules.

For example, let's consider the factorial function (`!`) from mathematics, shown below as a piecewise function:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

The above definition is recursive, and can quite naturally be expressed in Prolog using recursively-defined rules. Prolog code implementing the above function is shown below:

```
1 factorial(0, 1).
2 factorial(N, Result) :-
3     N > 0,
4     MinOne is N - 1,
5     factorial(MinOne, RestResult),
6     Result is N * RestResult.
```

A line-by-line explanation of the above code follows:

1. Fact implementing the behavior that the factorial of 0 is 1, directly from the original math
2. Rule implementing the recursive case of factorial.
3. Check that the input n is greater than 0
4. Calculation of $n - 1$ in the original math
5. Calculation of $(n - 1)!$ in the original math
6. Calculation of $n \times (n - 1)!$ in the original math

Example queries to the above code are shown below. Note that the engine had additional choices to explore and appeared to "hang", as occurs whenever the engine has further nondeterministic choices it can explore. In this case, extra exploration always leads to failure. As such, instead of pressing semicolon (`;`) for more solutions, period (`.`) was instead pressed to stop the search.

```
?- factorial(0, N).
N = 1.
?- factorial(1, N).
N = 1.
?- factorial(2, N).
N = 2.
?- factorial(3, N).
N = 6.
?- factorial(4, N).
N = 24.
```

As shown with the example queries above, the first parameter to `factorial` is the number to get the factorial of, and the second parameter holds the result.

2 Structures

In addition to integers and atoms, we can also form composite data structures. Composite data structures (or just structures) allow us to hold multiple values at once, much like a tuple. The only real difference from a tuple is that they also have a given name associated with them. Compared to SimpleScala, Prolog structures behave like a combination of a constructor and a tuple. To illustrate this, the code snippet below shows a structure named `foo` which contains the values `1` and `2`:

```
foo(1, 2)
```

Structures can be used to build up larger data structures, and often recursive data structures like lists and trees. To demonstrate, we will implement some list operations below which operate on a custom list definition, where `cons` is the name of a structure representing a non-empty list, and `nil` is an atom representing an empty list. With this in mind, we will defined a `myAppend` routine which appends two lists together, resulting in a third list.

```
myAppend(nil, List, List).
myAppend(cons(Head, Tail), List, cons(Head, Rest)) :-
  myAppend(Tail, List, Rest).
```

A line-by-line explanation of the above code follows:

1. If the first list is empty, the result (held in the third parameter) should be the same as the second list. In other words, if we append an empty list onto some other list `List`, then the result should be `List`.

2. If the first list is non-empty, name the components of the list `Head` (for the first element of the list) and `Tail` (for the rest of the list). The result list should start with this same `Head`, followed by the other list `Rest`, which has not yet been defined.

3. Recursively call `myAppend`, using `Tail` (the rest of the elements in the first parameter), `List` (the second parameter), and `Rest` (the recursive result).

An example query using the above `myAppend` definition follows. This query appends the list `1, 2, 3` onto the list `4, 5, 6`, yielding the result list `1, 2, 3, 4, 5, 6`. The query follows:

```
?- myAppend(cons(1, cons(2, cons(3, nil))),
            cons(4, cons(5, cons(6, nil))),
            Result).
Result = cons(1, cons(2, cons(3, cons(4, cons(5, cons(6, nil)))))).
```

Lists are very commonly used in Prolog. As such, there is built-in notation for lists, which helps improve readability and gets rid of all the extra parentheses above. This notation is translated down into normal structures which behave very similarly to `cons` and `nil` above; in fact, the only differences are in the structure names used. Some notes on the built-in list notation follow:

- `[]`: The empty list
- `[1]`: A list containing one element, namely `1`
- `[1, 2]`: A list containing two elements, namely `1` and `2`. This same pattern can be used for a list of length 3 (e.g., `[1, 2, 3]`), and so on.
- `[Head|Tail]`: A non-empty list that starts with the element `Head`, where the rest of the list is held in `Tail`.
- `[First, Second|Rest]`: A list with a minimum length of 2, where the first element is `First`, the second element is `Second`, and the rest of the elements (as in, all elements after `Second`) are in the list `Rest`.

We can rewrite `myAppend` with this updated list notation. This rewrite (yielding `myAppend2`) is shown below, along with the updated query:

```
myAppend2([], List, List).
myAppend2([Head|Tail], List, [Head|Rest]) :-
  myAppend2(Tail, List, Rest).

?- myAppend2([1,2,3],
             [4,5,6],
             Result).
Result = [1, 2, 3, 4, 5, 6].
```

While the code above looks very different from the code before, this behaves in exactly the same way.

2.1 Executing "Backwards"

Before concluding this section, there is an important point to make about how `myAppend2` works. In the previous example, we simply appended the lists `[1, 2, 3]` and `[4, 5, 6]` together, yielding the unsurprising result `[1, 2, 3, 4, 5, 6]`. By this point the idea of appending lists together in this way is old-hat, so this does not really show off anything particularly interesting about Prolog.

What is interesting about `myAppend2` is that we can execute it effectively "in reverse". For example, instead of giving it the known inputs of `[1, 2, 3]` and `[4, 5, 6]` and asking for the output, we can instead give a known output of `[1, 2, 3, 4, 5, 6]` and ask for inputs. Such a query is shown below:

```
?- myAppend2(Input1, Input2, [1, 2, 3, 4, 5, 6]).
```

If we keep hitting semicolon (`;`) and ask for different query results, we end up with 7 in all, listed below:

1. `Input1 = [], Input2 = [1, 2, 3, 4, 5, 6]`
2. `Input1 = [1], Input2 = [2, 3, 4, 5, 6]`
3. `Input1 = [1, 2], Input2 = [3, 4, 5, 6]`
4. `Input1 = [1, 2, 3], Input2 = [4, 5, 6]`
5. `Input1 = [1, 2, 3, 4], Input2 = [5, 6]`
6. `Input1 = [1, 2, 3, 4, 5], Input2 = [6]`
7. `Input1 = [1, 2, 3, 4, 5, 6], Input2 = []`

As shown with these results, such a query effectively asks "which two unknown lists, when appended to each other, yield the list `[1, 2, 3, 4, 5, 6]`?" Each of the above answers reflects some possible combination of the input lists `Input1` and `Input2` to yield this expected result list (e.g., the first result appends the empty list onto `[1, 2, 3, 4, 5, 6]`, the second result appends the list `[1]` onto the list `[2, 3, 4, 5, 6]`, and so on). In this way, depending on the query we issue to `myAppend2`, we can effectively execute "backwards", going from known outputs to unknown inputs.