CS61A Notes – Week 07b: Nondeterministic and Logic programming

Nondeterministic and Indecisive

The nondeterministic evaluator extends the metacircular evaluator with nondeterministic searching. It's good, clean American fun. First we take a look at the new special form, `amb`*, which takes in any number of arguments. If there are no arguments, it* fails*. If there are arguments, it chooses the first one; if the first one causes a failure, it chooses the second one; if that one causes a failure, it chooses the third, etc., until it runs out of arguments, at which point it itself signals a failure.*

A companion to `amb` is `require`, which takes in a predicate value, and causes a failure if the value is `#f`:

```
    (define (require pred?)
        (if (not pred?) (amb)))
```

Since you are trained on the imperative and functional ways of programming, this makes the structure of your program very unintuitive. But just imagine that the nondeterministic evaluator works magically; when an error is signaled by `(amb)`, you are allowed to fly to the nearest non-empty call to `amb`, try the next option and start over.

The book and the lectures already present several examples of programs written for the nondeterministic evaluator. But I will provide another motivating example…

Consider the problem of finding a path in a binary tree from the root to an element. Consider this tree `t`:

```
                     3
                 /       \
               5           6
             /   \       /   \
           1       30  2       5
          /           / \     / \
        20           0   12  1   32
```

Note that this is not a binary search tree! Here's what we'd like:

```
> (path-to t 12)
(r l r) ;; to get from root to 12, go right, left, right
> (path-to t 32)
(r r r) ;; to get to 32, go right, right, right
> (path-to t 19)
#f ;; 19 is not in the tree
```

Here's how we'd need to write `path-to` in good old Scheme:

```
    (define (path-to tree x)
        (cond ((empty-tree? tree) #f)
              ((eq? (datum tree) x) '())
              (else
                  (let ((result (path-to (left-branch tree) x)))
                      (if result
                          (cons 'L result)
                          (let ((result (path-to (right-branch tree) x)))
                              (if result
                                  (cons 'R result)
                                  #f)))))))
```

An ugly little thing! But suppose you take advantage of the nondeterministic evaluator:

```
    (define (path-to tree x)
        (cond ((empty-tree? tree) (amb))
              ((eq? (datum tree) x) '())
              (else (amb (cons 'L (path-to (left-branch tree) x))
                         (cons 'R (path-to (right-branch tree) x))))))
```

In the recursive case, we use `amb` to choose *either* walking down the left branch or the right branch of the tree. Suppose we chose left; then, if, eventually, we reach an empty tree, we're going to signal an error, causing us to switch to right. If that throws another error, we ourselves will signal an error (since our `amb` has run out of options), and our parent will explore other options (either trying the right branch or signaling failure to its parent).

Even better – if there are two elements in the tree, we can get the path to one and, after entering `try-again`, can get the path to the other. Think of the nightmare of doing that with regular Scheme!

QUESTIONS

1. Suppose we type the following into the amb evaluator:

```
> (* 2 (if (amb #t #f #t)
           (amb 3 4)
          5))
```

What are all possible answers we can get?

2. Write a function `an-atom-of` that dispenses the atomic elements of a deep list (not including empty lists). For example,

```
> (an-atom-of '((a) ((b (c))))) => a
> try-again => b
```

3. Use `an-atom-of` to write `deep-member?`.

4. Fill in the blanks:

```
> (define (choose-member L R)
      (cond ((null? R) (amb))
            ((= (car L) (car R)) (car L))
            (else (amb (choose-member L (cdr R))
                       (choose-member (cdr L) R)))))
> (choose-member '(1 2 3) '(4 2 3))
```

____

```
> try-again
```

____

```
> try-again
```

____

Paradigm Shift Again (Why Not?)

With this many paradigm shifts in a single semester, we expect you to at least be able to pronounce the word "paradigm" correctly after this class.

To reiterate, we are now in the realm of logic or declarative programming. Here in the magical world of the non-imperative, we can say exactly what we want – and have the computer figure out how to get it for us. Instead of saying how to get the solution, we describe – declare – what the solution looks like.

The mind-boggling part of all of this is that it all just works through pattern matching. That is, there are no "procedures" in the way you're used to; when you write out a parenthesized statement, it's not really a procedure call, and you don't really get a return value. Instead, either you get entries from some database, or nothing at all.

Be Assertive And Ask Questions (Fitter, Happier, More Productive)

There are two things you can type into our query system: an **assertion** and a **query**.

A **query** asks whether a given expression matches some fact that is already in the database. If the query matches, then the system prints out all matches in the database. If the query doesn't match to anything, you get no results.

An **assertion** states something true; it adds an entry to the database. You can either assert a simple fact, or a class of facts (specified by a "rule").

So here is an assertion that you have seen: `(assert! (jon likes omnom))`

You can also assert rules. In general, a rule looks like: `(rule <conclusion> <subconditions>)`

And it's read: "conclusion is true if and only if all the subconditions are true". Note that you don't have to have subconditions! Here's a very simple rule:

```
        (rule (same ?x ?x))
```

The above says two things satisfy the "same" relation if they can be bound to the same variable. It's deceptively simple – no subconditions are provided to check the "sameness" of the two arguments. Instead, either the query system can bind the two arguments to the same variable `?x` – and it can only do so if the two are equivalent – or, the query system can't.

And, of course, the rule of love:

```
(assert! (rule (?person1 loves ?person2)
               (and (?person1 likes ?something)
                    (?person2 likes ?something)
                    (not (same ?person1 ?person2))))))
```

The above rule means that `?person1` loves `?person2` if the three conditions following can all be satisfied – that is, `?person1` likes `?something` that `?person2` also likes, and that `?person1` is not the same person as `?person2`.

Note the "`and`" special form enclosing the three conditions – an entry in the database must satisfy *ALL* three conditions to be a match for the query. If you would like a rule to be satisfied by this *or* that condition, you can either use the `or` special form in the obvious way, or you can make two separate rules. For example, if one loves another if they like the same things *or* if `?person1` is a parent of `?person2`, we would add the following to the database:

```
        (assert! (rule (?person1 loves ?person2) (parent ?person1 ?person2)))
```

Note the new rule does NOT "overwrite" the previous rule; this is not the same thing as redefining a procedure in Scheme. Instead, the new rule complements the previous rule.

To add to confusion, you can also use the `and` special form for queries. For example,

```
        (and (jon loves ?someone) (?someone likes om nom))
```

is a query that finds a person Jon loves because that person likes to om nom.

There's another special form called `lisp-value`: `(lisp-value <pred?> <arg1, arg2, ...>)`

The `lisp-value` condition is satisfied if the given `pred?` applied to the list of `args` return `#t`. For example, `(lisp-value even? 4)` is satisfied, but `(lisp-value < 3 4 1)` is not. `lisp-value` is useful mostly for numerical comparisons (things that the logic system isn't so great at).

A note on writing rules: it's often tempting to think in terms of procedures – "this rule takes in so and so, and returns such and such". **This is not the right way to approach these problems** – remember, *nothing* is returned; an expression or query either has a match, or it doesn't. So often, you need to have both the "arguments" and the "return value" in the expression of the rule, and the rule is satisfied if "return value" is what would be returned if the rule were a normal Scheme procedure given the "arguments". Always keep in mind that everything is a Yes or No question, and your rule can only say if something is a correct answer or not. So when you write a rule, instead of trying to "build" to a solution like you've been doing in Scheme, think of it as trying to check if a given solution is correct or not.

In fact, **this is so important I'll say it again:** when you define rules, don't think of it as defining procedures in the traditional sense. Instead, think of it as, *given arguments and a proposed answer, check if the answer is correct.* The proposed answer can either be derived from the arguments, or it can't.

A different approach for writing declarative rules is to try to convert a Scheme program to a rule. For example, let's take a crack at the popular `append`:

```
        (define (append ls1 ls2)
            (cond ((null? ls1) ls2)
                  (else (cons (car ls1) (append (cdr ls1) ls2))))
```

The `cond` specifies an "either-or" relationship; either `ls1` is null or it is not. This implies that, for logic programming, we'd need two separate rules, each corresponding to each `cond` clause. The first one is straightforward:

```
        (rule (append () ?ls2 ?ls2))
```

The second `cond` clause breaks `ls1` into two parts – its `car` and its `cdr` – and basically says that appending `ls1` and `ls2` is the same as `cons`ing the first element of `ls1` to the list obtained by appending the `cdr` of `ls1` to `ls2`. Translated to logic programming, it means the `cdr` of the result is equivalent to `append`ing the `cdr` of `ls1` to `ls2`, and that the `car` of the result is just the `car` of `ls1`. This implies:

```
        (rule (append (?car . ?cdr) ?ls2 (?car . ?r-cdr))
              (append ?cdr ?ls2 ?r-cdr))
```

We will try the above techniques on some of the harder problems below.

The Search For Truth and Honor (defined as such in our database)

Don't think declarative programming is useless – you might see it more than you think. The most common use of it is in database queries, and if you're worked with databases before, you know that SQL is a declarative language:

```
    SELECT * FROM people WHERE age=13 ORDER BY first_name;
```

The above is a valid query into the table (or "relation") called "`people`" to select all columns of entries with column "`age`" equal to 13, and we want the result to be ordered by the column "`first_name`". Note how we did *not* instruct to the database on how to give us the result; we simply trust the database management system to figure out the most efficient way to satisfy our desires. This is one place where declarative programming is absolutely natural (and think of the nightmare you'd have if you have to specify your queries imperatively!)

Of course, database queries is about the easiest use of logic programming, and you've had plenty of practice with `microshaft` (a rather obscene name if you ask me) in lab and homework. So let's move on to the interesting stuff.

Lists Again (and again, and again, and again, and again...)

Since lists are just patterns of symbols, logic programming is especially good at dealing with them.

```
(my-list (1 2 3 4))
```

Let's look at the following queries. Note the explanations – it's easy to see intuitively, sometimes, why something is bound to something, but you should get used to the way the query processor thinks!

`(my-list ?x) =>` `?x` is bound to `(1 2 3 4)` because there's an entry in the database starting with "`my-list`" and followed by one more thing.

`(my-list (1 ?x 3 4)) =>` `?x` is bound to `2` because there's an entry in the database starting with "`my-list`" and followed by a list of four elements, the first, third and fourth of which are `1`, `3` and `4`.

`(my-list (1 ?x)) =>` nothing. There is no entry in the database that starts with "`my-list`" followed by a two-element list – and, note carefully, `(1 ?x)` is a list of two elements!

`(my-list (1 . ?x)) =>` `?x` is bound to `(2 3 4)` because there's an entry in the database starting with "`my-list`" followed by a list whose first element is `1`. `?x` is simply bound to the rest. Note that this is because `(1 . (2 3 4))` is equivalent to `(1 2 3 4)`.

`(?x (1 2 3 4)) =>` `?x` is bound to `my-list` because there's a database entry whose second element is `(1 2 3 4)`.

Now, let's play with a few.

QUESTIONS

1. **Write a rule for `car` of list. For example, `(car (1 2 3 4) ?x)` would have `?x` bound to `1`.**

2. **Write a rule for `cdr` of list. For example, `(cdr (1 2 3) ?y)` would have `?y` bound to `(2 3)`.**

3. **Define our old friend, `member`, so that `(member 4 (1 2 3 4 5))` would be satisfied, and `(member 3 (4 5 6))` would not, and `(member 3 (1 2 (3 4) 5))` would not.**

4. **Define its cousin, `deep-member`, so that `(deep-member 3 (1 2 (3 4) 5))` would be satisfied as well.**

5. Define another old friend, `reverse`, so that `(reverse (1 2 3) (3 2 1))` would be satisfied.

6. (HARD!) Define its cousin, `deep-reverse`, so that `(deep-reverse (1 2 (3 4) 5) (5 (4 3) 2 1))` would be satisfied.

7. Write the rule `remove` so that `(remove 3 (1 2 3 4 3 2) ?what)` binds `?what` to `(1 2 4 2)` – the list with 3 removed.

8. Write the rule `interleave` so that `(interleave (1 2 3) (a b c d) ?what)` would bind `?what` to `(1 a 2 b 3 c d)`.

9. Consider this not very interesting rule: `(rule (listify ?x (?x)))`. So if we do `(listify 3 ?what)`, `?what` would be bound to `(3)`.

Define a rule `map` with syntax `(map procedure list result)`, so that `(map listify (1 2 3) ((1) (2) (3)))` would be satisfied, as would `(map reverse ((1 2) (3 4 5)) ((2 1) (5 4 3)))`. In fact, we should be able to do something cool like `(map ?what (1 2 3) ((1) (2) (3)))` and have `?what` bound to the word "`listify`". Assume the "procedures" we pass into `map` are of the form `(procedure-name argument result)`.

10. We can let predicates have the form `(predicate-name argument)`. Define a rule `even` so that `(even 3)` is not satisfied, and `(even 4)` is satisfied.

11. The above is a way to make predicates. And once we have predicates, we can – and will , of course – write a `filter` rule with the syntax `(filter predicate list result)` so that `(filter even (1 2 3 4 5 6) (2 4 6))` returns `Yes`, and querying `(filter ?what (10 11 12 13) (10 12))` would bind `?what` to the word "even".

Number Theory (The Bizarre Way)

Logic programming is bad at working with numbers, since arithmetic isn't really pattern matching. Or isn't it? Suppose we use a new numbering system:

```
()          => zero
(a)         => one
(a a)       => two
```

And so on. The number of `a`'s in the list represents the number. This way, we don't have to use `lisp-value`, and we can do a ton of cool stuff, including reinventing arithmetic. For example, let's try the rule `sum`:

```
(rule (sum () ?x ?x))              ;; The sum of 0 and a number is itself
(rule (sum (a . ?x) ?y (a . ?z))
      (sum ?x ?y ?z))
```

The first rule is obvious, but the second is a bit puzzling. Note that the first rule is the "base case", and as in everywhere else in this course, we're always trying to reduce a problem down to a base case. We have seen the second rule before, but where? Wait, that's the `append` rule! Notice that adding two numbers in this new system is the same as appending two lists together, because the length of the final list is equal to the sum of the lengths of the two lists that we are appending. We use this fact, and the fact that all of the "lists" in this new system are made up of `a`'s, to come up with the `sum` rule. Sum rule, eh?

QUESTIONS

1. Write the rule `subtract` using the same syntax as `sum`. Assume that the first argument will always be greater than the second (since we don't support negative numbers with our system!)

2. **Write the rule `product`. You may use rules that you have defined before.**

3. **Write the rule `divide` that divides the first argument with the second, and returns the quotient and the remainder. For example, `(divide (a a a a a a a) (a a a) ?quo ?rem)` would bind `?quo` to `(a a)` and `?rem` to `(a)`. You may (and should!) use rules that you have defined before.**

4. **Define the rule `exp` (for exponent, of course), with the first argument the base and second the power, so that `(exp (a a) (a a) ?what)` would bind `?what` to `(a a a a)`.**

5. **Write the rule `factorial`, so that `(factorial (a a a) ?what)` would bind `?what` to `(a a a a a a)`.**

6. **Write the rule `appearances` that counts how many times something appears in a list. For example, `(appearances 3 (1 2 3 3 2 3 3) ?what)` would bind `?what` to `(a a a a)`.**