CS2PLD: Programming Language Design Paradigms
Maribel Fernández & Odinaldo Rodrigues

Revision (sketch of sample solutions)

Exercise 1:

1. Suppose a Prolog program contains the following clauses:

   ```
   sunny(mon).
   sunny(tue).
   sunny(wed).
   warm(mon).
   warm(wed).
   glorious(Day) :- sunny(Day), warm(Day).
   ```

   Write down all the responses which Prolog will give to the query

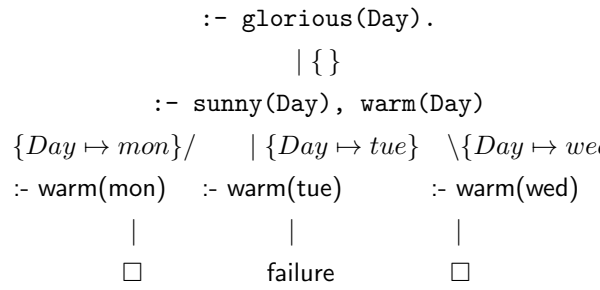
   `:- glorious(Day).`

   in the order they are given.

2. Draw the SLD-tree built by Prolog for the previous query.

Sample solution:

1. The answers are: Day = mon and Day = wed.
2. SLD-tree:



Exercise 2:

1. Define a function *until* that takes a function $f :: \alpha \to \alpha$, a predicate $p :: \alpha \to Bool$ and an element $x :: \alpha$ as arguments, and iterates f until an element that satisfies p is obtained.
2. Give the type of *until*.

Sample solution:

`until::`$(\alpha \to \alpha) \to (\alpha \to Bool) \to \alpha \to \alpha$

```
until f p x = if (p x) then x else until f p (f x)
```

Exercise 3:

Consider the SFUN program:

$$double(x) \quad = \quad 2 * x$$

- Show by induction that for any natural number n, $double(n)$ is equivalent to $n+n$ (i.e. show that $double(n)$ and $n+n$ produce the same value, for any natural number n). You can use either call-by-value or call-by-name semantics.
- Explain why for terms of the form $double(t)$ where t is an arithmetic expression, the call-by-value and call-by-name strategies have the same behaviour and are equally efficient.
- Assume empty lists are represented by nil and non-empty lists by $x\colon l$ where x is the first element and l the rest of the list. Explain in your own words the behaviour of the function map defined below, and describe a call-by-name evaluation for

  $$map\ double\ list$$

  where $list$ is a function that returns a list of numbers.

$$\begin{array}{lcl} map\ f\ nil & = & nil \\ map\ f\ (x\colon l) & = & (f\ x)\colon (map\ f\ l) \end{array}$$

Sample solution:

- Induction on n:
Base Case: If $n = 0$ then $dobule(0) = 2 * 0 = 0 + 0$.
Inductive Step:
Assume $double(n) = n + n$. Then:
$double(n + 1) = 2 * (n + 1) = 2 * n + 2 = double(n) + 2 = n + n + 2 = (n + 1) + (n + 1)$
as required.

- Both strategies have the same behaviour because all arithmetic expressions are terminating. Moreover they are equally efficient because the function uses x, and it uses it only once.

- The function map has a function and a list as arguments, and applies the function to all the elements in the list, producing a new list.
To evaluate $map\ double\ list$ using a call-by-name strategy, we need to evaluate $list$ just enough to know whether it is empty or non-empty. If $list$ reduces to nil, then the result is nil. If $list$ reduces to $(x\colon l)$ then $map\ double\ list \rightarrow (double\ x)\colon (map\ double\ l) \rightarrow (2 * x)\colon (map\ double\ l)$ and the process continues in the same way.

Exercise 4:

Consider the following logic program:

```
even(0).
even(N) :- N > 1, N1 is N-2, even(N1).
```

and goals:

```
:- even(4).
:- even(3).
```

- Assuming `N > 1` succeeds when `N` is instantiated by a number greater than 1 and fails otherwise, describe the SLD-resolution tree for each goal, using the computation rule that always selects the leftmost literal as the one resolved upon.
- What are the answers that Prolog gives for these goals?

- We now replace the second clause by:

  ```
  even(N) :- N1 is N-2, even(N1), N > 1.
  ```

  Explain how Prolog will behave now with the same goals as above. What will be the answers for these goals?

Sample solution:

- Drawing the trees we observe that the first has one success branch, the second only a fail branch.

- The answers are Yes and No, since Prolog's strategy will find the success in the first tree but the second has only a fail branch.

- Now both trees have an infinite branch. Prolog will answer yes for the first query and then loop if asked for alternative solutions. Prolog will loop in the second query and provide no answer.