Constructive Logic (15-317), Fall 2009
Assignment 7: Programming in Prolog

William Lovas (`wlovas@cs`)

Out: Thursday, October 22, 2009
Due: Thursday, October 29, 2009 (before class)

In this assignment, you'll practice programming in Prolog in order to learn the idiosyncracies and idioms particular to the discipline of logic programming. By the end of the assignment, you will be an expert Prolog hacker, and the basic ideas you learn in Prolog will carry over to other logic programming languages we encounter as the semester progresses.

The written portion of your work (Section 1) may be submitted at the beginning of class, typeset or handwritten, but alternatively, you may submit the written portion in comments as part of your programming portion.

The programming portion of your work (Sections 2 and 3) should be submitted via AFS by putting your code in a file `hw07.pl` and copying it to the directory

```
/afs/andrew/course/15/317/submit/<userid>/hw07
```

where `<userid>` is replaced with your Andrew ID.

For the programming problems, try to make your code as clean and general as possible, leveraging the intrinsic expressive power of logic programming to obtain clear and elegant programs. You may find it useful to refer to the GNU Prolog Manual, in particular the chapter on Prolog built-in predicates.

1 One Cut, Two Cut, Red Cut, Green Cut (8 points)

Recall the Prolog control mechanism `!`, pronounced "cut": as a subgoal, it always succeeds, but when it does so it has the side effect of forcing Prolog's proof search to commit to all choices made thus far; proof search never backtracks past a cut.

Remember also that cuts can be divided into *red cuts* and *green cuts*. Green cuts do not affect the meaning of a program: any predicate that holds when the cut is deleted still holds when the cut is present. Red cuts, on the other hand, prune search that might lead to new solutions, and in this way they change the meaning of the logic program. Sometimes the question of whether a cut is red or green depends upon our mode interpretation of the predicate in whose definition the cut appears.

Task 1 (8 pts). For each the following Prolog code fragments, identify the cut as *green* or *red*. Justify your choice with a short explanation, and be sure to state any assumptions you make.

Fragment 1

```
doFor(B, what) :- klondike_bar(B).
doFor(love, that) :- !, fail.
doFor(love, _).
```

Fragment 2

```
partition([], _, [], []).

partition([X|Xs], X0, [X|Ls], Gs) :-
    X =< X0, !, partition(Xs, X0, Ls, Gs).

partition([X|Xs], X0, Ls, [X|Gs]) :-
    X > X0, partition(Xs, X0, Ls, Gs).
```

Fragment 3

```
search(X, leaf(X)).
search(X, node(Left, Right)) :- search(X, Left), !.
search(X, node(Left, Right)) :- search(X, Right).
```

Fragment 4

```
classify(Msg, spam) :- is_spam(Msg), !.
classify(Msg, ham).
```

2 Sorting Lists (16 points)

In Lecture Notes 14, we saw how to code the quicksort algorithm in Prolog. Here, we'll consider two other sorting algorithms.

2.1 Bogo sort

Prolog can compactly and elegantly represent a sorting algorithm that is even worse than insertion sort. The canonical bogo sort works by randomly permuting a list until it becomes sorted. Here, we'll consider a deterministic variant that searches for a sorted permutation systematically.

Task 2 (6 pts). Write a predicate `sorted([integer])` which holds of any ≤-ordered list of Prolog integers. Then write a predicate `perm([integer], [integer])` which holds whenever its first argument is a permutation of its second. Use these together to implement a concise version of deterministic bogo sort as a predicate `bogosort([integer], [integer])` that searches for a sorted permutation of its first argument. Your predicate should be usable as a sorting function, at least on very small inputs.

2.2 Merge sort

Prolog can also compactly and elegantly represent efficient sorting algorithms like *merge sort*. To merge sort a list, first split it into two roughly equal halves; then, recursively merge sort the two halves; finally merge the two sorted halves into a sorted list.

Task 3 (10 pts). Implement a predicate `mergesort(+[integer], -[integer])` which merge sorts a given list, returning the sorted list. Your predicate should be usable as a sorting function on *much* larger inputs than `bogosort`.

3 Arithmetic Expressions (16 points)

Prolog can be used to do much more than list processing. Many symbolic processing tasks can elegantly be written as Prolog programs, in much the same way as symbolic tasks can elegantly be written as ML programs. Of course, the logic programming paradigm changes the game quite a bit: an elegant ML program might not transliterate directly to an elegant Prolog program, nor vice versa.

In this section we'll explore symbolic processing tasks involving a small language of arithmetic expressions with numerical constants, addition, and multiplication.

$$e ::= n \mid e_1 + e_2 \mid e_1 \times e_2$$

The following Prolog type predicate recognizes expressions built using the constructors `eInt`, `ePlus`, and `eTimes`:

```
exp(ePlus(E1, E2)) :- exp(E1), exp(E2).
exp(eTimes(E1, E2)) :- exp(E1), exp(E2).
exp(eInt(N)) :- integer(N).
```

Task 4 (6 pts). Write an evaluator predicate `eval(+exp, -integer)` that evaluates an arithmetic expression to its value.

3.1 Parsing

If we isolate certain terms as tokens, we can talk about writing a parser for arithmetic expressions using Dijkstra's classic shunting yard algorithm.

We'll use the following constructors for tokens:

```
token(tLparen).  token(tRparen).       % parentheses
token(tPlus).  token(tTimes).          % operators
token(tInt(N)) :- integer(N).          % integer constants
```

A parser for arithmetic expressions is one that transforms a list of tokens into an expression.

The shunting yard algorithm works using two stacks, a stack of operators and a stack of operands. When we say

> reduce an expression onto the operand stack,

we mean

> pop the top operator from the operator stack and the top two operands from the operand stack, build an expression out of them and push that expression onto the operand stack.

With this in mind, we can describe (a simplified variant of) the algorithm as follows. Repeatedly consider each token in the input list in turn:

- If the token is a number (`tInt(N)` in our syntax), then remove it from the input list and push it onto the operand stack.
- If the token is an operator (`tPlus` or `tTimes`), then
  - if the operator has stronger binding precedence than the one on top of the operator stack, or if the operator stack is empty, remove the operator from the input list and push it onto the operator stack,
  - if the operator has weaker binding precedence than the one on top of the operator stack, then leave it in the input list and reduce an expression onto the operand stack.
- If the token is a left paren (`tLparen`), then remove the operator from the input list and push it onto the operator stack.
- If the token is a right paren (`tRparen`), then
  - if the operator on top of the operator stack is a left paren (`tLparen`), the remove the right paren from the input list and pop the left paren from the operator stack,
  - otherwise reduce an expression onto the operand stack.
- If the input list is empty, then
  - if there is one expression on the operand stack, then return that expression as the final result,
  - otherwise, reduce an expression onto the operand stack.

The main idea behind this algorithm is to put off processing an operator until we know we have parsed both of its operands. This is accomplished by keeping the operator stack in precedence order, weakest at the bottom to strongest at the top. Figures 1 and 2 show two examples of its execution, where stacks are written with their tops to the left. Tokens are represented by characters, and expression trees have the form `op`(x, y).

Task 5 (10 pts). Write a predicate `parse(+[token], -exp)` that implements the shunting yard algorithm. Remember to try to make full use of the expressive power of logic programming to obtain an elegant program.

You may find GNU Prolog's trace facility useful to debug your program: it prints each predicate upon entry, exit, and failure. To turn it on, press `^C` at the prompt and then press `t`.

Extra Credit Task 1 (3 extra credit pts). The algorithm as specified above will succeed on inputs like [`+`, `3`, `5`], [`2`, `+`, `*`, `3`, `5`], and [`(`, `)`, `5`], which do not represent valid infix expressions. Extend your code to rule out these cases, and describe your solution in comments.

Input	Operands	Operators	Notes
2, +, 3, *, 5			Initial state
+, 3, *, 5	2		Numbers always get pushed
3, *, 5	2	$+$	
*, 5	$3, 2$	$+$	
5	$3, 2$	$\times, +$	$\times$ binds stronger than +, so $\times$ pushed
	$5, 3, 2$	$\times, +$	
	$\times(3, 5), 2$	$+$	Input is empty, so reduce an expression
	$+(2, \times(3, 5))$		Final result is on top of operand stack

Figure 1: Parse for the expression $2 + 3 * 5$.

Input	Operands	Operators	Notes
2, *, 3, +, 5			Initial state
*, 3, +, 5	2		Numbers always get pushed
3, +, 5	2	$\times$	
+, 5	$3, 2$	$\times$	
+, 5	$\times(2, 3)$		$+$ binds weaker than $\times$, so reduce
5	$\times(2, 3)$	$+$	Now $+$ can be pushed
	$5, \times(2, 3)$	$+$	
	$+(\times(2, 3), 5)$		Input is empty, so reduce an expression

Figure 2: Parse for the expression $2 * 3 + 5$.