PROLOG EXAMPLES

Script started on Thu 03 Mar 2005 12:23:23 AM CST

redstone\% cat min.pro
min(X, Y, X) :- X < Y.
min(X, Y, Y) :- X \geq Y.

redstone\% sbp
SB-Prolog Version 3.1
\% ?- consult('min.pro').
yes
\% ?- min(0, 1, 0).

yes
\% ?- min(17, 45, Minimum).
Minimum = 17
yes
\% ?- min(65, 3, Minimum).
Minimum = 3
yes
\% 'D
Halt. Program terminated normally

redstone\% cat factorial.pro
factorial(0, 1).
factorial(N, Factorial) :-
  M is N - 1, factorial(M, M_Factorial), Factorial is N * M_Factorial.

redstone\% sbp
SB-Prolog Version 3.1
\% ?- consult('factorial.pro').
yes
\% ?- factorial(5, Factorial).
Factorial = 120
yes
\% ?- factorial(10, Factorial).
Factorial = 3628800
yes
\% 'D
Halt. Program terminated normally
redstone% cat length.pro
length([], 0).
length([X | Xs], Length) :- length(Xs, Xs_Length), Length is Xs_Length + 1.

redstone% sbp
SB-Prolog Version 3.1
| ?- consult('length.pro').
yes
Length = 4
yes
Length = 2
yes
Length = 0
yes
Length = 4
yes
| ?- 'D
Halt. Program terminated normally
redstone% cat lisp_functions.pro
car([X | Xs], X).
cdr([X | Xs], Xs).
cons(X, Xs, [X | Xs]).
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).

redstone% sbp
SB-Prolog Version 3.1
| ?- consult('lisp_functions.pro').
yes
| ?- cons(x, [], A).
A = [x]
yes
| ?- cons(y, [x], B).
B = [y,x]
yes
| ?- append([y, x], [z], D).
D = [y,x,z]
yes
| ?- append([y, x], [z], D), cdr(D, Cdr_D), car(Cdr_D, Cadr_D).
D = [y,x,z]
Cdr_D = [x,z]
Cadr_D = x
yes
| ☹. ´D
Halt. Program terminated normally

redstone% cat ancestors.pro
female(shelley).
female(mary).
female(lisa).
female(joan).

male(bill).
male(jake).
male(bob).
male(frank).

mother(mary, jake).
mother(mary, shelley).
mother(lisa, mary).
mother(joan, bill).

father(bill, jake).
father(bill, shelley).
father(bob, mary).
father(frank, bill).

parent(Father, Child) :- father(Father, Child).
parent(Mother, Child) :- mother(Mother, Child).

parents(Father, Mother, Child) :- father(Father, Child), mother (Mother, Child).

sibling(Child1, Child2) :-
    father(Father, Child1), father(Father, Child2),
    mother(Mother, Child1), mother(Mother, Child2).

true_sibling(Child1, Child2) :- sibling(Child1, Child2), not(Child1 = Child2).

ancestor(Ancestor, Descendant) :- parent(Ancestor, Descendant).
ancestor(Ancestor, Descendant) :-
    parent(Descendants_Parent, Descendant),
    ancestor(Ancestor, Descendants_Parent).

redstone% sbp
SB-Prolog Version 3.1
| ?- consult('ancestors.pro').
yes
| ?- parents(Father, Mother, jake).
Father = bill
Mother = mary
yes
| ?- parents(bill, mary, Child).
Child = jake;
Child = shelley;
no
| ?- parents(Father, Mother, Child).
Father = bill
Mother = mary
Child = jake;
Father = bill
Mother = mary
Child = shelley;
Father = bob
Mother = lisa
Child = mary;
Father = frank
Mother = joan
Child = bill;
no
| ?- sibling(Child1, Child2).

Child1 = jake
Child2 = jake;
Child1 = jake
Child2 = shelley;
Child1 = shelley
Child2 = jake;
Child1 = shelley
Child2 = shelley;
Child1 = mary
Child2 = mary;
Child1 = bill
Child2 = bill;
no
| ?- true_sibling(Child1, Child2).

Child1 = jake
Child2 = shelley;
Child1 = shelley
Child2 = jake;
no
| ?- ancestor(Ancestor, jake).

Ancestor = bill;
Ancestor = mary;
Ancestor = frank;
Ancestor = joan;
Ancestor = joan;
Ancestor = bob;
Ancestor = lisa;
no
| ?- "D
redstone% cat postfix.pro
/
  This program converts a postfix expression, represented as a list of
  operands and operators, into a syntax tree.
 /*

postfix(Exp, Tree) :- postfix_stack(Exp, [], [Tree]).

postfix_stack([], Tree, Tree).

postfix_stack([Operator | Rest_exp], [Top, Next_top | Rest_stack], Tree) :-
  operator(Operator),
  postfix_stack(Rest_exp, [tree(Operator, Next_top, Top) | Rest_stack], Tree).

postfix_stack([Operand | Rest_exp], Stack, Tree) :-
  postfix_stack(Rest_exp, [Operand | Stack], Tree).

operator(+).
operator(-).
operator(*)..
operator(/).

redstone% sbp
SB-Prolog Version 3.1
| ?- consult('postfix.pro').
yes
| ?- postfix([a,b,c,*,+],Tree).
Tree = tree(+,a,tree(*,b,c))
yes
| ?- postfix([a,b,+c,*,d,/,e,-],Tree).
Tree = tree(-,tree(/,tree(*,tree(+,a,b),c),d),e)
yes
| ?- ^D
Halt. Program terminated normally
redstone% cat parser.pro
start_symbol(e).

production(e, [t, e1], 1).
production(e1, [+ t e1], 2).
production(e1, [t], 3).
production(t, [f t1], 4).
production(t1, [* f t1], 5).
production(t1, [t], 6).
production(f, [', e ', ']', 7).
production(f, [id], 8).
production(id, [a], 9).
production(id, [b], 10).
production(id, [c], 11).

equal(X, X).

append([], Y, Y).
append([X | Xs], Y, [X | Zs]) :- append(Xs, Y, Zs).

/* pda simulates the moves of a pushdown automaton, producing a left parse */
pda([], [], []). /* string consumed, stack empty, accept */
pda([Head2 | Tail2], LeftParse1) :-
  production(Head2, [], N), /* e-production */
  pda([], Tail2, LeftParse2), /* pop Head2 */
  append([N], LeftParse2, LeftParse1).
pda([Head1 | Tail1], [Head | Tail2], LeftParse) :-
  pda(Tail1, Tail2, LeftParse). /* pop matching input and top of stack */
pda([Head1 | Tail1], [Head2 | Tail2], LeftParse1) :-
  not(equal(Head1, Head2)),
  production(Head2, RHS, N),
  append(RHS, Tail2, Stack), /* push production right side for Head2 on stack */
  pda([Head1 | Tail1], Stack, LeftParse2), /* try again with new stack */
  append([N], LeftParse2, LeftParse1).

parse(String, LeftParse) :- start_symbol(S), pda(String, [S], LeftParse).

redstone% sbp
SB-Prolog Version 3.1
| ?- consult('parser.pro').
yes
| ?- parse([a], LeftParse).
LeftParse = [1,4,8,9,6,3]
yes
| ?- parse([a, +, b], LeftParse).
LeftParse = [1,4,8,9,6,2,4,8,10,6,3]
yes
| ?- parse([a, +, b, *, c], LeftParse).
LeftParse = [1,4,8,9,6,2,4,8,10,5,8,11,6,3]
yes
| ?- 'D
Halt. Program terminated normally
redstone% exit