Prolog continuation

Simple data types

•Atoms

Atoms are essentially user-defined names, but have no meaning on their own. They can optionally be enclosed in single quotes, e.g. `foo`, `'blah'`, `x` etc.

•Numeric values

Prolog supports traditional integer and floating point numeric values, e.g. 32, 123.456, etc.

•Character values

Characters can be represented by preceding them with `0'`, e.g. `0'z` for the lowercase z character.

•String values

Strings in prolog are represented within double quotes, e.g. "This is a string", which internally are stored as lists of the characters' ascii values.

Atoms and characters

Atom values

Any non-reserved lowercase word is an atom, representing a unique entity, e.g. fred, blue, castle, etc.
If we want an atom name to include spaces, begin with uppercase, include special characters, etc, then we can define the atom in single quotes, e.g. 'Hi there!'.

Character values

Internally they are stored as their ascii values, e.g. `S is 0'x.` instantiates variable S with value 120.
If you store characters as single-character atoms, e.g. 'x', then you can use the

lower_upper(AtomCharLower, AtomCharUpper) to convert between lower and uppercase.
atom_length(Atom, Length) - number of characters in an atom
atom_concat(A1,A2,Result) - concatenate two atoms together to get a third

String values and functions

- Strings in (swi)prolog can be represented either as a double-quoted string, e.g. "foo", or
- as a list of ascii/unicode character values, e.g. [102,111,111]. (It's pretty much a matter of checking documentation to determine which format is used/expected.)

Various conversion functions

Converting to/from double-quoted strings:

atom_string(foo, "foo")
number_string(123, "123")
term_string(foo(x),"foo(x)")

Converting to/from lists of ascii codes:

name(foo,[102,111,111])
name('foo',[102,111,111])
name("foo",[102,111,111])
name(123.4,[49,50,51,46,52])

Converting to/from atoms:

atom_chars(foo,[f,o,o]) (atom vs a list of single-char atoms)
number_atom(123, '123') (number vs atom)
number_chars(123, ['1','2','3'])
string_chars("foo", [f,o,o])

String built-in list functions(predicates)

The built-in list functions work if you have your string as a list of codes
(e.g. append(str1,str2,result) while other functions can be applied if you have a double-quoted string. Examples include:

string_length("hello",N) gives N=5,
string_concat("foo","blah",S) gives S="fooblah")

Split string lets you break a large string into a list of smaller ones, where you can specify what character(s) to split at, and any padding characters to be removed from the result. E.g. below we split on commas and remove periods:

split_string("this , is some, text.", ",", ".", L)

gives L = ["this","is some","text"]

Conditionals

Prolog has built-in support for conditionals through the use of predicates. There are two types of conditionals in Prolog: the if-then-else construct and the pattern matching construct. Here's an example of each:

If-then-else construct:

```
max(X, Y, Max) :- X > Y, Max is X.
max(X, Y, Max) :- Y >= X, Max is Y.
```

Using -> operator

```
max(A, B, Max) :-
 A > B    ->  Max = A  ;   Max = B.
```

Pattern matching construct:

```
max(X, Y, Z, X) :- X >= Y, X>Z.
max(X, Y, Z, Y) :- Y>X, Y>Z.
```

Loops

Prolog is a declarative programming language that is based on the concept of relations and logical inference, and it does not have traditional loops like in imperative programming languages. However, you can achieve ***looping behavior in Prolog using recursive predicates.***

countdown(0) :- write('Go!'), nl.

countdown(N) :- N > 0, write(N), nl, N1 is N - 1, countdown(N1).

Using keyword repeat (infinite loop with fail)

get_valid_number(N) :-

repeat,

write('Enter a number: ‘),

read(N), integer(N), !.

```
get_valid_number(N) :- repeat,
write('Enter a number: ‘),
read(N),
( integer(N) -> ! ;  write('Invalid input. Please enter a number.\n'), fail ).
```

Input/output

In Prolog, the built-in predicates **read/1** and **write/1** are used to read input from the user and write output to the screen, respectively.

```
write('Hello, World!').
```

The **read/1** predicate is used to read input from the user. It takes a single argument, which is a variable to be bound to the input value. For example:

```
read(X),
write('You entered: ‘),
write(X).
```

NOTE: read/1 expects the user to enter a valid Prolog term, so if the user enters something that is not a valid term, Prolog will throw a syntax error.

Read_line vs Read_line_to_string

Use the built-in **read_line/1** predicate or **read_line_to_string/2** predicate to read a line of input from the user as a string.

Main difference between **read_line/1** and **read_line_to_string/2** is that **read_line/1** returns the input as a list of character codes, while **read_line_to_string/2** returns the input as a string, including the end-of-line character. Which one to use depends on the needs of your specific use case. If you need to work with the input as a list of character codes, then **read_line/1** may be more appropriate. If you need to work with the input as a string, including the end-of-line character, then **read_line_to_string/2** may be more appropriate.

See Examples

List Syntax

Samples:

[mia, vincent, jules, yolanda]

[mia, robber(honey_bunny), X, 2, mia]

[]

[mia, [vincent, jules], [butch, girlfriend(butch)]]

[[], dead(z), [2, [b, c]], [], Z, [2, [b, c]]]

List Syntax

- Enclose lists in brackets
- Any type elements
- Empty list is []
- Lists can contain lists
- All lists have an implied empty list at the end

Head and Tail

- Use [H | T] to extract the first element:

  [Head|Tail] = [mia, vincent, jules, yolanda].
  Head = mia
  Tail = [vincent,jules,yolanda]

- Multiple elements:

  [H1, H2 |Tail] = [mia, vincent, jules, yolanda].
  H1 = mia
  H2 = vincent
  Tail = [jules,yolanda]

- No more elements:

  [H1 |Tail] = [mia].
  H1 = mia
  Tail = [ ]

- Empy list has no Head and Tail

Are you a member of a list?

- Either X is the head
- Or x is a member of the tail
- Remember the empty list has no Head and Tail so the empty tail wont fit member(X,[X|T]).
- member(X,[X|T]).
  member(X,[H|T]) :- member(X,T).
- Because you don't care about H in second or T in first, replace with _
- member(X,[X|_]).
  member(X,[_|T]) :- member(X,T).

Recurse a2b

- The empty list is a non-recursive true a2b fact
  - a2b([],[]).
- The head of one list being a and the other being b plus the rest of each list fitting a2b
  - a2b([a|Ta],[b|Tb]) :- a2b(Ta,Tb).
- a2b([a,a,a,a],X).

  X = [b,b,b,b].
- a2b(X,[b,b,b,b]).

  X = [a,a,a,a]

Translate a list

```
tran(eins,one).
tran(zwei,two).
tran(drei,three).
tran(vier,four).
tran(fuenf,five).
tran(sechs,six).
tran(sieben,seven).
tran(acht,eight).
tran(neun,nine).
```

Create listtran([eins,neun,zwei],X). That will return the translation: X = [one,nine,two].

Translation Answer

- Good translation: A list of no items translates to a list of no items
  - listtran([],[]).
- Another good translation: the head of the german list being part of a tran fact with the head of the english list plus the rest of the lists being a translation as well.
- listtran([Hg|Tg], [He|Te]):- tran(Hg,He), listtran(Tg,Te).

Append 2 lists

- Append is true when nothing added to a List yields that List
  - append([],L,L).
- Append is true when the first element of the list = the first element of another list, and when the tail of the first list is appended to the tail of the result.

  append([H|T],L2,[H|L3]) :- append(T,L2,L3).

Trace append

Call: (10) append([a, b, c, d, e], [1, 2, 3], _G518) ? creep

Call: (11) append([b, c, d, e], [1, 2, 3], _24700) ? creep

Call: (12) append([c, d, e], [1, 2, 3], _24700) ? creep

Call: (13) append([d, e], [1, 2, 3], _24700) ? creep

Call: (14) append([e], [1, 2, 3], _24700) ? creep

Exit: (14) append([e], [1, 2, 3], [e, 1, 2, 3]) ? creep

Exit: (13) append([d, e], [1, 2, 3], [d, e, 1, 2, 3]) ? creep

Exit: (12) append([c, d, e], [1, 2, 3], [c, d, e, 1, 2, 3]) ? creep

Exit: (11) append([b, c, d, e], [1, 2, 3], [b, c, d, e, 1, 2, 3]) ? creep

Exit: (10) append([a, b, c, d, e], [1, 2, 3], [a, b, c, d, e, 1, 2, 3]) ? creep

_G518 = [a, b, c, d, e, 1, 2, 3].

prefix

- How to get all the elements that are prefixes in the list?
- prefix(X,[a,b,c,d]).

X = [] ;

X = [a] ;

X = [a,b] ;

X = [a,b,c] ;

X = [a,b,c,d] ;

prefix

- prefix(P,L):- append(P,_,L).

Exercise: Query for Suffix

- ?- suffix(X,[a,b,c,d]).

  X = [a,b,c,d] ;

  X = [b,c,d] ;

  X = [c,d] ;

  X = [d] ;

  X = [] ;

Reversing a list

- Naïve:

```
nrev([],[]).
nrev([H|T],R):-
        nrev(     T,RevT), append(RevT,[H], R)        .
```