Lists and Recursion

List processing – handling sequences of elements – is a powerful technique in Prolog. In this tutorial, we explain what lists are and how to declare them, and then give several examples that show how you might use list processing in your own applications. We also define two well known Prolog predicates – member and append – while looking at list processing from both a recursive and a procedural standpoint.

After that, we introduce findall, a Visual Prolog standard predicate that enables you to find and collect all solutions to a single goal. We round out this tutorial with a discussion of compound lists – combinations of different types of elements – and an example of parsing by difference lists.

What Is a List?

In Prolog, *a list* is an object that contains an arbitrary number of other objects within it. Lists correspond roughly to arrays in other languages, but, unlike an array, a list does not require you to declare how big it will be before you use it.

A list that contains the numbers **1**, **2**, and **3** is written as

```
[1, 2, 3]
```

The order of the elements in this list matters:

- Number "1" is the first element,
- "2" - the second,
- "3" - the third.

The list **[1, 2, 3]** is different from the list **[1, 3, 2]**.

Each item contained in the list is known as an element. To form a list data structure, you separate the elements of a list with commas and then enclose them in square brackets. Here are some examples:

```
["dog", "cat", "canary"]
["valerie ann", "jennifer caitlin", "benjamin thomas"]
```

The same element can be present in the list several times, for example:

```
[1, 2, 1, 3, 1]
```

Lists Domains

It **T** is a type/domain then **T*** is the domain of lists containing **T** elements. I.e. an asterisk after a domain means "List of that domain".

The elements in a list can be anything, including other lists. However, all elements in a list must belong to the ***same*** domain.

Heads and Tails

A list is really a recursive compound object. It consists of two parts: the head, of a list, which is the first element, and the tail, which is a list comprising all the subsequent elements.

The tail of a list is always a list; the head of a list is an element.

For example,

```
the head of [a, b, c] is a
the tail of [a, b, c] is [b, c]
```

What happens when you get down to a one-element list? The answer is that:

```
the head of [c] is c
the tail of [c] is []
```

If you take the first element from the tail of a list enough times, you will eventually get down to an empty list (**[]**).

The empty list cannot be broken into head and tail.

This means that, conceptually, lists have a tree structure just like other compound objects. The tree structure of [a, b, c, d] is:

```
      list
     /    \
    a    list
        /    \
       b    list
           /    \
          c    list
              /    \
             d     []
```

Further, a one-element list such as [a] is not the same as the element that it contains, because [a] is really the compound data structure shown here:

```
      list
     /    \
    a     []
```

List Processing

Prolog provides a way to make a head and a tail of a list explicit. Instead of separating elements with commas, you can separate the head and tail with a vertical bar (|). For instance,

```
[a, b, c]
```

is equivalent to

```
[a | [b, c]]
```

which continuing the process is equivalent to

```
[a | [b | [c]]]
```

which is equivalent to

```
[a | [b | [c | []]]]
```

You can even use both kinds of separators in the same list, provided the vertical bar is the last separator. So, if you really want to, you can write [a, b, c, d] as [a, b|[c, d]].

Table 1 gives more examples.

Table 1: Heads and Tails of Lists

List	Head	Tail
['a', 'b', 'c']	'a'	['b', 'c']
['a']	'a'	[]
[]	does not exist	does not exist
[[1, 2, 3], [2, 3, 4], []]	[1, 2, 3]	[[2, 3, 4], []]

Table 2 gives several examples of list unification.

Table 2: Unification of Lists

List 1	List 2	Variable Binding
[X, Y, Z]	[egbert, eats, icecream]	X=egbert, Y=eats, Z=icecream
[7]	[X | Y]	X=7, Y=[]
[1, 2, 3, 4]	[X, Y | Z]	X=1, Y=2, Z=[3,4]
[1, 2]	[3 | X]	fail

Using Lists

Because a list is really a recursive compound data structure, you need recursive algorithms to process it. The most basic way to process a list is to work through it, doing something to each element until you reach the end.

An algorithm of this kind usually needs two clauses. One of them says what to do with an ordinary list (one that can be divided into a head and a tail). The other says what to do with an empty list.

Writing Lists

For example, if you just want to print out the elements of the list, here is what you do:

```
implement main

clauses
   run() :-
```

```
        write_a_list([1, 2, 3]).

    class predicates
        write_a_list : (integer* List).
    clauses
        write_a_list([]).
            /* If the list is empty, do nothing more. */
        write_a_list([H | T]) :-
            /* Match the head to H and the tail to T, then... */
            stdio::write(H),
            stdio::nl,
            write_a_list(T).

    end implement main

    goal
        console::runUtf8(main::run).
```

Here are the two write_a_list clauses described in natural language:

- To write an empty list, do nothing.
- Otherwise, to write a list, write its head (which is a single element), then write its tail (a list).

The **write_a_list** predicate is called from **run** clause:

```
write_a_list([1, 2, 3])
```

This matches the second clause, with H=1 and T=[2, 3]; this writes 1 and then calls write_a_list recursively with the tail of the list:

```
write_a_list([2, 3]).
    /* This is write_a_list(T). */
```

This recursive call matches the second clause, this time with H=2 and T=[3], so it writes 2 and again calls write_a_list recursively:

```
write_a_list([3]).
```

Now, which clause will this goal match? Recall that, even though the list [3] has only one element; it does have a head and tail; the head is 3 and the tail is []. So, again the goal matches the second clause, with H=3 and T=[]. Hence, 3 is written and write_a_list is called recursively like this:

```
write_a_list([]).
```

Now you see why this program needs the first clause. The second clause will not match this goal because [] cannot be divided into head and tail. So, if the first clause were not there, the goal would fail. As it is, the first clause matches and the goal succeeds without doing anything further.

Counting List Elements

Now consider how you might find out how many elements are in a list. What is the length of a list, anyway? Here is a simple logical definition:

- The length of an empty list, **[]**, is **0**.
- The length of any other list is the length of its tail plus **1**.

Can you implement this? In Prolog it is very easy. It takes just two clauses:

```
class main

predicates
    length_of : (A* List) -> integer Length.

end class

implement main

clauses
    length_of([]) = 0.
    length_of([_ | T]) = Length :-
        TailLength = length_of(T),
        Length = TailLength + 1.

end implement main

goal
    console::init(),
    Length = main::length_of([1, 2, 3]),
    stdio::write(Length).
```

Take a look at the second clause first. Crucially, **[_ | T]** will match any nonempty list, binding **T** to the tail of the list. The value of the head is unimportant; as long as it exists, it can be counted it as one element.

So the goal:

```
Length = main::length_of([1, 2, 3])
```

will match the second clause, with **T** = **[2, 3]**. The next step is to compute the length of **T**. When this is done (never mind how), **TailLength** will get the value **2**, and the computer can then add **1** to it and bind **Length** to **3**.

So how is the middle step executed? That step was to find the length of **[2, 3]** by satisfying the goal

```
TailLength = main::length_of([2, 3])
```

In other words, **length_of** calls itself recursively. This goal matches the second clause, binding

- **[3]** in the goal to **T** in the clause and

- **TailLength** in the goal to **Length** in the clause.

Recall that **TailLength** in the goal will not interfere with **TailLength** in the clause, because ***each recursive invocation of a clause has its own set of variables***.

So now the problem is to find the length of **[3]**, which will be **1**, and then add **1** to that to get the length of **[2, 3]**, which will be **2**. So far, so good.

Likewise, **length_of** will call itself recursively again to get the length of **[3]**. The tail of **[3]** is **[]**, so **T** is bound to **[]**, and the problem is to get the length of **[]**, then add 1 to it, giving the length of **[3]**.

This time it's easy. The goal:

```
TailLength = main::length_of([])
```

matches the *first* clause, binding **TailLength** to **0**. So now the computer can add **1** to that, giving the length of **[3]**, and return to the calling clause. This, in turn, will add **1** again, giving the length of **[2, 3]**, and return to the clause that called it; this original clause will add **1** again, giving the length of **[1, 2, 3]**.

Confused yet? We hope not. In the following brief illustration we'll summarize the calls. We've used suffixes to indicate that similarly named variables in different clauses – or different invocations of the same clause – are distinct. Please notice that you do not need to implement such predicate in your own code, you can use **list::length** predicate from PFC.

```
Length_1 = main::length_of([1, 2, 3]).
   TailLength_1 = main::length_of([2, 3]).
      TailLength_2 = main::length_of([3]).
         TailLength_3 = main::length_of([]).
         TailLength_3 = 0.
      Length_3 =  TailLength_3 + 1 = 1.
   Length_2 = Length_3 + 1 = 2.
Length_1 = Length_2 + 1 = 3.
```

Notice that the code above can be written more compact without the use of intermediate variables like this:

```
clauses
   length_of([]) = 0.
   length_of([_ | T]) = length_of(T) + 1.
```

These clauses more or less directly states the original specification:

- The length of an empty list, **[]**, is **0**.
- The length of any other list is the length of its tail plus **1**.

Tail Recursion

You probably noticed that **length_of** is not, and can't be, tail-recursive, because the recursive call is not the last step in its clause. Can you create a tail-recursive list-length predicate? Yes, but it will take some effort.

The problem with **length_of** is that you can't compute the length of a list until you've already computed the length of the tail. It turns out there's a way around this. You'll need a list-length predicate with two arguments.

- The first is the list, which the computer will whittle away on each call until it eventually becomes empty, just as before.
- The second is an counter that starts out as **0** and increments on each call.

When the list is finally empty, you'll return the counter as the list length.

```
class main

predicates
   length_of : (A* List, integer Counter) -> integer Result.

end class

implement main

clauses
   length_of([], Counter) = Counter.
   length_of([_ | T], Counter) = Result :-
      NewCounter = Counter + 1,
      Result = length_of(T, NewCounter).

end implement main

goal
   console::init(),
   main::length_of([1, 2, 3], L, 0), /* start with Counter = 0 */
   stdio::write(" L = ", L).
```

This version of the **length_of** predicate is more complicated, and in many ways less logical, than the previous one. We've presented it merely to show you that, *by devious means, you can often find a tail-recursive algorithm for a problem that seems to demand a different type of recursion*.

Another Example – Modifying the List

Sometimes you want to take a list and create another list from it. You do this by working through the list element by element, replacing each element with a computed value. For example, here is a program that takes a list of numbers and adds 1 to each of them:

```
class main

predicates
   add1 : (integer* NumberList) -> integer* IncrList.

end class

implement main
```

```
clauses
    add1([]) = []. % boundary condition
    add1([Head | Tail]) = Result :-
        % separate the head  from the rest of the list
        Head1 = Head + 1, % add 1 to the first element
        Tail1 = add1(Tail), % call element with the rest of the list
        Result = [Head1 | Tail1]. % Put the new head infront of the new tail

end implement main

goal
    console::init(),
    NewList = main::add1([1, 2, 3, 4]),
    stdio::write(NewList).
```

To paraphrase this in natural language:

To add 1 to all the elements of the empty list,
just produce another empty list.

To add 1 to all the elements of any other list,
add 1 to the head and make it the head of the result, and then
add 1 to each element of the tail and make that the tail of the result.

Runnig the the predicate with **[1, 2, 3, 4]**:

```
NewList = add1([1,2,3,4]).
```

Will result in

```
NewList = [2, 3, 4, 5]
```

Tail Recursion Again

Is **add1** tail-recursive? If you're accustomed to using Lisp or Pascal, you might think it isn't, because you think of it as performing the following operations:

- Split the list into **Head** and **Tail**.
- Add **1** to **Head**, giving **Head1**.
- Recursively add 1 to all the elements of **Tail**, giving **Tail1**.
- Combine **Head1** and **Tail1**, giving the resulting list.

This isn't tail-recursive, because the recursive call is not the last step.

But – and this is important – *that is not how Prolog does it*. In Visual Prolog, **add1** is tail-recursive, because its steps are really the following:

- Bind the head and tail of the original list to **Head** and **Tail**.
- Bind the head and tail of **Result** to **Head1** and **Tail1**. (**Head1** and **Tail1** do not have values yet.)
- Add **1** to **Head**, giving **Head1**.
- Recursively add **1** to all the elements of **Tail**, giving **Tail1**.

When this is done, **Head1** and **Tail1** are *already* the head and tail of **Result**; there is no separate operation of combining them. So the recursive call really is the last step.

More on Modifying Lists

Of course, you don't actually need to put in a replacement for every element. Here's a program that scans a list of numbers and copies it, leaving out the negative numbers:

```
class main

predicates
    discard_negatives : (integer*, integer* [out]).

end class

implement main

clauses
    discard_negatives([], []).
    discard_negatives([H | T], ProcessedTail) :-
        H < 0,
        !, /* If H is negative, just skip it */
        discard_negatives(T, ProcessedTail).
    discard_negatives([H | T], [H | ProcessedTail]) :-
        discard_negatives(T, ProcessedTail).

end implement main

goal
    console::init(),
    main::discard_negatives([2, -45, 3, 468], X),
    stdio::write(X).
```

For example, the goal

```
main::discard_negatives([2, -45, 3, 468], X)
```

gives

```
X=[2, 3, 468].
```

And here's a predicate that copies the elements of a list, making each element occur twice:

```
clauses
    doubletalk([], []).
    doubletalk([H | T], [H, H | DoubledTail]) :-
        doubletalk(T, DoubledTail).
```

List Membership

Suppose you have a list with the names *John*, *Leonard*, *Eric*, and *Frank* and would like to use Visual Prolog to investigate if a given name is in this list. In other words, you must express the relation "membership" between two arguments: a name and a list of names. This corresponds to the predicate

```
isMember : (name, name*).
    /* "name" is a member of "name*" */
```

In the program below, the first clause investigates the head of the list. If the head of the list is equal to the name you're searching for, then you can conclude that Name is a member of the list. Since the tail of the list is of no interest, it is indicated by the anonymous variable. Thanks to this first clause, the goal

```
main::isMember("john", ["john", "leonard", "eric", "frank"])
```

is satisfied.

```
class main

predicates
    isMember : (A, A*) determ.

end class

implement main

clauses
    isMember(Name, [Name | _]) :-
        !.
    isMember(Name, [_ | Tail]) :-
        isMember(Name, Tail).

end implement main

goal
    console::init(),
    main::isMember("john", ["john", "leonard", "eric", "frank"]),
    !,
    stdio::write("Success")
    or
    stdio::write("No solution").
```

If the head of the list is not equal to Name, you need to investigate whether Name can be found in the tail of the list.

In English:

Name is a member of the list if Name is the first element of the list, or
Name is a member of the list if Name is a member of the tail.

The second clause of isMember relates to this relationship. In Visual Prolog:

```
    isMember(Name, [_ | Tail]) :-
        isMember(Name, Tail).
```

Appending One List to Another: Declarative and Procedural Programming

As given, the member predicate of the program works in two ways. Consider its clauses once again:

```
clauses
    isMember(Name, [Name | _]) :-
        !.
    isMember(Name, [_ | Tail]) :-
        isMember(Name, Tail).
```

You can look at these clauses from two different points of view: declarative and procedural.

- From a declarative viewpoint, the clauses say:Name is a member of a list if the head is equal to Name;

if not, Name is a member of the list if it is a member of the tail.

- From a procedural viewpoint, the two clauses could be interpreted as saying:To find a member of a list, find its head;

otherwise, find a member of its tail.

These two points of view correspond to the goals

```
member(2, [1, 2, 3, 4]).
```

and

```
member(X, [1, 2, 3, 4]).
```

In effect, the first goal asks Visual Prolog to check whether something is true; the second asks Visual Prolog to find all members of the list [1,2,3,4]. Don't be confused by this. The member predicate is the same in both cases, but its behavior may be viewed from different angles.

Recursion from a Procedural Viewpoint

The beauty of Prolog is that, often, when you construct the clauses for a predicate from one point of view, they'll work from the other. To see this duality, in this next example you'll construct a predicate to append one list to another. You'll define the predicate append with three arguments:

```
append(List1, List2, List3).
```

This combines List1 and List2 to form List3. Once again you are using recursion (this time from a procedural point of view).

If List1 is empty, the result of appending *List1* and List2 will be the same as List2. In Prolog:

```
append([], List2, List2).
```

If List1 is not empty, you can combine List1 and List2 to form *List3* by making the head of List1 the head of List3. (In the following code, the variable H is used as the head of both List1 and List3.) The tail of List3 is L3, which is composed of the rest of List1 (namely, L1) and all of List2. In Prolog:

```
append([H | L1], List2, [H | L3]) :-
   append(L1, List2, L3).
```

The append predicate operates as follows: While List1 is not empty, the recursive rule transfers one element at a time to List3. When List1 is empty, the first clause ensures that List2 hooks onto the back of List3.

One Predicate Can Have Different Uses

Looking at append from a declarative point of view, you have defined a relation between three lists. This relation also holds if List1 and List3 are known but List2 isn't. However, it also holds true if only List3 is known. For example, to find which two lists could be appended to form a known list, you could use a goal of the form

```
append(L1, L2, [1, 2, 4]).
```

With this goal, Visual Prolog will find these solutions:

```
1L1=[], L2=[1,2,4]
L1=[1], L2=[2,4]L1=[1,2], L2=[4]L1=[1,2,4], L2=[]4 Solutions
```

You can also use append to find which list you could append to [3,4] to form the list [1,2,3,4]. Try giving the goal

```
append(L1, [3,4], [1,2,3,4]).
```

Visual Prolog finds the solution

```
L1=[1,2].
```

This append predicate has defined a relation between an *input set* and an *output set* in such a way that the relation applies both ways. Given that relation, you can ask

Which output corresponds to this given input?

or

Which input corresponds to this given output?

The status of the arguments to a given predicate when you call that predicate is referred to as a *flow pattern*. An argument that is bound or instantiated at the time of the call is an input argument, signified by (i); a free argument is an output argument, signified by (o).

The append predicate has the ability to handle any flow pattern you provide. However, not all predicates have the capability of being called with different flow patterns. When a Prolog clause is able to handle multiple flow patterns, it is known as an invertible clause. Many of list predicate can be found in the **list** class.

Finding All the Solutions at Once

Backtracking and recursion are two ways to perform repetitive processes. Recursion won out because, unlike backtracking, it can pass information (through arguments) from one recursive call to the next. Because of this, a recursive procedure can keep track of partial results or counters as it goes along.

But there's one thing backtracking can do that recursion can't do – namely, find all the alternative solutions to a goal. So you may find yourself in a quandary: You need all the solutions to a goal, but you need them all at once, as part of a single compound data structure. What do you do?

Fortunately, Visual Prolog provides a way out of this impasse. The built-in construction list comprehension takes a goal as one of its arguments and collects all of the solutions to that goal into an output single list. list comprehension takes two arguments:

- The first argument, VarName, specifies which argument in the specified predicate is to be collected into a list.
- The second, mypredicate, indicates the predicate from which the values will be collected.
- The output ListParam, is a variable that holds the list of values collected through backtracking.

The this program uses list comprehension to print the average age of a group of people.

```
class main

domains
   name = string.
   address = string.
   age = integer.

predicates
   person : (name Name [out], address Address [out], age Age [out]) multi.
   sumlist : (age* AgeList, age Age [out], integer Count [out]).
```

```
end class

implement main

clauses
    sumlist([], 0, 0).
    sumlist([H | T], Sum, N) :-
        sumlist(T, S1, N1),
        Sum = H + S1,
        N = 1 + N1.

clauses
    person("Sherlock Holmes", "22B Baker Street", 42).
    person("Pete Spiers", "Apt. 22, 21st Street", 36).
    person("Mary Darrow", "Suite 2, Omega Home", 51).

end implement main

goal
    console::init(),
    L = [Age || main::person(_, _, Age)],
    main::sumlist(L, Sum, N),
    Average = Sum / N,
    stdio::writef("Average = %.", Average).
```

The list comprehension clause in this program creates a list L, which is a collection of all the ages obtained from the predicate person. If you wanted to collect a list of all the people who are 42 years old, you could give the following subgoal:

```
List = [Who || main::person(Who, _, 42)]
```

The following code will create the list of positive items:

```
List = [X || X in [2,-8,-3,6], X > 0]
```

Compound Lists

A list of integers can be simply declared as

```
integer*
```

The same is true for a list of real numbers, a list of symbols, or a list of strings.

However, it is often valuable to store a combination of different types of elements within a list, such as:

```
[2, 3, 5.12, ["food", "goo"], "new"] % Not correct Visual Prolog
```

Compound lists are lists that contain more than one type of element. You need special declarations to handle lists of multiple-type elements, because Visual Prolog requires that all elements in a list belong to the ***same*** domain. The way to create a list in Prolog that stores these different types of elements is to use functors, because a domain can contain ***more than one*** data type as arguments to functors.

The following is an example of a domain declaration for a list that can contain an integer, a character, a string, or a list of any of these:

```
domains
    llist = l(list); i(integer); c(char); s(string).
        % the functors are l, i, c, and s
```

The list

```
[2, 9, ["food", "goo"], "new"] % Not correct Visual Prolog
```

would be written in Visual Prolog as:

```
[i(2), i(9), l([s("food"), s("goo")]), s("new")] % Correct Visual Prolog
```

The following example of append shows how to use this domain declaration in a typical list-manipulation program.

```
class main

domains
    llist =
        l(list);
        i(integer);
        c(char);
        s(string).

predicates
    test : ().

end class

implement main

class predicates
    append : (A*, A*, A* [out]).
clauses
    append([], L, L).
    append([X | L1], L2, [X | L3]) :-
        append(L1, L2, L3).

clauses
    test() :-
        append([s("likes"), l([s("bill"), s("mary")])], [s("bill"), s("sue")], Ans),
        stdio::write("FIRST LIST: ", Ans, "\n\n"),
        append([l([s("This"), s("is"), s("a"), s("list")]), s("bee")], [c('c')], Ans2),
        stdio::write("SECOND LIST: ", Ans2, "\n\n").

end implement main
```

Parsing by Difference Lists

The ch07e10.pro program demonstrates parsing by *difference lists*. The process of parsing by difference lists works by reducing the problem; in this example we transform a string of input into a Prolog structure that can be used or evaluated later.

The parser in this example is for a very primitive computer language. Although this example is very advanced for this point in the tutorial, we decided to put it here because parsing is one of the areas where Visual Prolog is very powerful. If you do not feel ready for this topic, you can skip this example and continue reading the tutorial without any loss of continuity.

```
/*******************************************
* Lexer
*******************************************/

class lexer

predicates
   tokl : (string Input) -> string* TokenTL.

end class lexer

implement lexer

clauses
   tokl(Input) = [Tok | tokl(Rest)] :-
      string::fronttoken(Input, Tok, Rest),
      !.
   tokl(_) = [].

end implement lexer

/*******************************************
* Parser
*******************************************/

class parser

domains
   program = statement*.

domains
   statement =
      if_Then_Else(expression Condition, statement ThenPart, statement ElsePart);
      if_Then(expression Condition, statement ThenPart);
      while(expression Condition, statement Body);
      assign(id Variable, expression Expression).

domains
   expression =
      plus(expression, expression);
      minus(expression, expression);
      var(id);
      int(integer).

domains
   id = string.

predicates
   program : (string* TL) -> program ProgramTree determ.

end class

implement parser

clauses
   program(TL) = Prog :-
      Prog = statement_list(TL, TL0),
      [] = TL0.

class predicates
   statement_list : (string* TL1, string* TL0 [out]) -> program Program determ.
clauses
   statement_list([], []) = [] :-
      !.
   statement_list(TL1, TL0) = [Statement | Program] :-
      Statement = statement(TL1, TL2),
      TL2 = [";" | TL3],
      Program = statement_list(TL3, TL0).

class predicates
   statement : (string* TL1, string* TL0 [out]) -> statement Statement determ.
clauses
   statement(["if" | TL1], TL0) = if_then_else(Condition, ThenPart, ElsePart) :-
      Condition = expression(TL1, TL2),
      TL2 = ["then" | TL3],
      ThenPart = statement(TL3, TL4),
      TL4 = ["else" | TL5],
      !,
      ElsePart = statement(TL5, TL6),
      TL6 = ["fi" | TL0].

   statement(["if" | TL1], TL0) = if_then(Condition, Statement) :-
      !,
      Condition = expression(TL1, TL2),
      TL2 = ["then" | TL3],
      Statement = statement(TL3, TL4),
      TL4 = ["fi" | TL0].

   statement(["do" | TL1], TL0) = while(Condition, Statement) :-
      !,
      Statement = statement(TL1, TL2),
      TL2 = ["while" | TL3],
      Condition = expression(TL3, TL0).

   statement([ID | TL1], TL0) = assign(Id, Exp) :-
      string::isname(ID),
```

```
        TL1 = ["=" | TL2],
        Exp = expression(TL2, TL0).

class predicates
    expression : (string* TL1, string* TL0 [out]) -> expression Expression determ.
clauses
    expression(TL1, TL0) = Exp :-
        PreTerm = term(TL1, TL2),
        Exp = termTail(TL2, TL0, PreTerm).

class predicates
    termTail : (string* TL1, string* TL0 [out], expression PreExpr) -> expression Expr determ.
clauses
    termTail(["+" | TL1], TL0, Exp1) = Exp :-
        !,
        Exp2 = term(TL1, TL2),
        Exp = termTail(TL2, TL0, plus(Exp1, Exp2)).

    termTail(["-" | TL1], TL0, Exp1) = Exp :-
        !,
        Exp2 = term(TL1, TL2),
        Exp = termTail(TL2, TL0, minus(Exp1, Exp2)).

    termTail(TL, TL, Exp) = Exp.

class predicates
    term : (string* TL1, string* TL0 [out]) -> expression Expression determ.
clauses
    term([Int | Rest], Rest) = int(I) :-
        try
            I = toTerm(Int)
        catch _ do
            fail
        end try,
        !.

    term([Id | Rest], Rest) = var(Id) :-
        string::isName(Id).

end implement parser

goal
    console::init(),
    TokenTL = lexer::tokl("b=2; if b then a=1 else a=2 fi; do a=a-1 while a;"),
    if ProgramTree = parser::program(TokenTL) then
        foreach Stmt in ProgramTree do
            stdio::write(Stmt, "\n")
        end foreach
    else
        stdio::write("Parse error\n")
    end if.
```

The program will write:

```
assign("b",int(2))
if_then_else(var("b"),assign("a",int(1)),assign("a",int(2)))
while(var("a"),assign("a",minus(var("a"),int(1))))
```

The transformation in this example is divided into two stages: scanning and parsing. The tokl predicate is the scanner; it accepts a string and converts it into a list of tokens. In this example the input text is a Pascal-like program made up of Pascal-like statements. This programming language only understands certain statements: IF THEN ELSE, IF THEN, DO WHILE, and ASSIGNMENT. Statements are made up of expressions and other statements. Expressions are addition, subtraction, variables, and integers.

Here's how this example works:

- The **program** predicate takes a list of tokens call the **statement_list** predicate on that list. And then is test that all tokens was used in the parse process.
- The predicate **statement_list** takes a list of tokens and tests if the tokens can be divided up into individual statements, each ending with a semicolon.
- The predicate **statement** tests if the first tokens of the token list make up a legal statement. If so, the statement is returned in a structure and the remaining tokens are returned back to **statement_list**.
- The four clauses of the **statement** correspond to the four types of statements the parser understands.
  - If the first **statement** clause is unable to transform the list of tokens into an *IF THEN ELSE* statement, the clause fails and backtracks to the next **statement** clause.
  - The second clause tries to transform the list of tokens into an *IF THEN* statement.
  - If that clause fails, the next clause tries to transform the list of tokens into a *DO WHILE* statement.
  - And if all the first three **statement** clauses fail, the last clause for that predicate tests if the statement does assignment. This clause tests for assignment by testing if the first term is a symbol, the second term is "=", and the next terms make up a simple math expression.
- The **expression**, **term**, and **termTail** predicates work in simailar ways, by testing if the first terms are expressions and – if so – returning the remainder of the terms and an expression structure back to **statement**.

Summary

These are the important points covered in this tutorial:

- *Lists* can contain an arbitrary number of elements; you declare them by adding an asterisk at the end of a previously defined domain.
- A list is a recursive compound object that consists of a head and a tail. The head is the first element and the tail is the rest of the list (without the first element). The tail of a list is always a list; the head of a list is an element. A list can contain zero or more elements; the empty list is written [].
- The elements in a list can be anything, including other lists; all elements in a list must belong to the same domain. The domain declaration for the elements must be of this form:

```
domains
   element_list = elements*.
   elements = ....
```

where elements = one of the standard domains (integer, real, etc.) or a set of alternatives marked with different functors (int(integer); rl(real); smb(symbol); etc.). You can only mix types in a list in Visual Prolog by enclosing them in compound objects/functors.

- You can use separators (commas, [, and |) to make the head and tail of a list explicit; for example, the list

```
[a, b, c, d]
```

can be written as:

```
[a | [b, c, d]] or
[a, b | [c, d]] or
[a, b, c | [d]] or
[a | [b | [c, d]]] or
[a | [b | [c | [d]]]] or
[a | [b | [c | [d | []]]]]
```

- List processing consists of recursively removing the head of the list (and usually doing something with it) until the list is an empty list.
- The list predicates can be found in the **list** class.
- Visual Prolog provides a built-in construction, list comprehension, which takes a goal as one of its arguments and collects all of the solutions to that goal into a single list. It has the syntax

```
Result = [Argument || myPredicate(Argument)]
```

- Because Visual Prolog requires that all elements in a list belong to the same domain, you use functors to create a list that stores different types of elements.
- The process of *parsing by difference lists* works by reducing the problem; the example in this tutorial transforms a string of input into a Prolog structure that can be used or evaluated later.