# Linked Lists and Prolog

## 1 Background on Lists

Linked lists are very common in both functional programming and logic programming, much more so than in typical imperative or object-oriented languages (e.g., Java). This is rooted in the fact that functional and logical languages tend to intentionally lack arrays, as arrays are fundamentally a mutable data structure. For example, a basic array operation is to update an element at a particular index, which implies mutable state.

Arrays often force mutable state to be used even if we don't really need it. For example, in Java, there is no way to simultaneously create and initialize an array of a length unknown at compile time. The assumption with Java is that the programmer will instead write something like this:

```
int length = readIntFromUser();
int[] array = new int[length];
for (int x = 0; x < length; x++) {
  array[x] = someComputation(x);
}
```

Such use of mutable state is undesirable or even impossible in functional and logical languages, so linked lists become the next best thing for representing collections of elements. While linked lists have disadvantages when it comes to performance, these tend to be insignificant in practice. For example, while arrays offer constant time access to arbitrary elements, it's very rare that this is required. Linear traversals, in contrast, are quite common for both data structures, and a linear traversal can be done in $\mathcal{O}(n)$ for either an array or a linked list. For the specific cases where constant-time arbitrary access is needed, functional and logic programming languages usually offer some way of doing this, though the solutions may not be clean.

## 2 List Encoding

A common list encoding used in both functional languages and logical languages is based on the following recursive definition:

**Definition 2.1** *List A list is one of two things:*

1. *An empty list, represented with a constant symbol*
2. *A pair (i.e., a 2-tuple) of items, where the first item is the first element of the list, and the second item is a list*

Lisp, Scheme, Racket, and Prolog use the above definition to encode lists. For example, in Racket, an empty list is represented with the symbol `'()`: that is, a symbol with the name of a left-parenthesis followed by a right-parenthesis. A pair in Racket is constructed with the `cons` function, like so:

```
(cons 1 '())
```

The above code snippet creates a pair of the integer `1`, followed by the symbol `'()`. From the definition above, this pair just so happens to encode a list containing the element `1`, and nothing else.

Prolog uses the same sort of representation, but instead of using `'()` and `cons`, Prolog instead uses the atom `[]` for the empty list and a structure named `.` to encode a pair. With this in mind, we can represent the same list as above in Prolog:

```
X = .(1, []).
```

In the above snippet, the variable `X` is unified with a list containing `1`, and nothing else.

As another example, consider a list containing the elements `1`, `2`, and `3`, in that order. Such a list can be represented in Racket like so:

```
(cons 1
  (cons 2
    (cons 3 '())))
```

...or in Prolog, like so:

```
X = .(1, .(2, .(3, []))).
```

## 3 List Shorthand Representation in Prolog

Because lists are so commonly used in Prolog, there is special syntax used for creating and manipulating lists. For example, a more common shorthand for creating the three-element list before is the following:

```
X = [1, 2, 3].
```

That is, square brackets are used to denote lists. The above truly is shorthand; this will end up being turned into the more verbose representation with `.` internally.

There is another common shorthand used to denote lists that start with a particular element. Consider the following example:

```
?-  X = [1, 2, 3],
    X = [H|T].
X = [1, 2, 3],
H = 1,
T = [2, 3].
```

In the above example, `X = [H|T]` means to unify `X` with a list starting with `H` as the first element, followed by the rest of the list `T`. Given the underlying representation of lists, the above query behaves identically to the following:

```
?-  X = [1, 2, 3],
    X = .(H, T).
X = [1, 2, 3],
H = 1,
T = [2, 3].
```

That is, `[H|T]` is synonymous with `.(H, T)`.

To be clear, the above unification does not add any new unification rules to what you already know. By saying `.(H, T)`, or equivalently `[H|T]`, we are talking about structures which contain variables. It just so happens that when we work with lists, we tend to perform lots of unifications with structures.

It is also possible to specify a list that begins with multiple elements, and is followed by some subsequent elements in a list. Consider the following example:

```
?-  X = [1, 2, 3, 4],
    X = [A, B|C].
X = [1, 2, 3, 4],
A = 1,
B = 2,
C = [3, 4].
```

The above query is synonymous with the following:

```
?-  X = [1, 2, 3, 4],
    X = .(A, .(B, C)).
X = [1, 2, 3, 4],
A = 1,
B = 2,
C = [3, 4].
```

All the above shorthands can be combined with each other, for example:

```
?-  X = [[1, 2], [3, 4, 5]],
    X = [[A, B], [C, D|E]|F].
X = [[1, 2], [3, 4, 5]],
A = 1,
B = 2,
C = 3,
D = 4,
E = [5],
F = [].
```

Keeping in mind that this is just normal unification in play, information can be exchanged in either direction, like so:

```
?-  [A, 2|B] = [1, C, 3, 4].
A = 1,
B = [3, 4],
C = 2.
```