Flat Lists, Induction, and Recursion

1 Flat Lists

This section introduces flat lists – that is, lists that may contain any number of elements of any number of data types, but no subsidiary lists. Such lists in the syntax of Scheme would therefore include lists like (1 a #t "hi!"), but not lists like (1 a (uh oh) 4), since the latter includes the subsidiary list, (uh oh), as one of its elements. Lists that contain other lists will be addressed later on.

The presentation of flat lists has many parallels with the presentation of the natural numbers seen in the previous handout.

1.1 The Domain, \(D\), of List Contents

Often, it is useful to restrict the kinds of data that can appear as elements in a list. For example, we might want to restrict attention to lists of numbers or lists of symbols. In other cases, we might want to consider lists of numbers, symbols, or strings. In this handout, we use \(D\) to denote the set of entities that can appear in our lists. For example, if we want to consider lists whose elements are restricted to the integers, then \(D\) will be the set of integers. Similarly, if we want to consider lists whose elements are restricted to symbols, then \(D\) will be the set of symbols. Given the restriction that our lists are not allowed to contain subsidiary lists, we require that \(D\) not contain any lists. For convenience, we shall frequently refer to \(D\) as the domain of list contents.

1.2 The Definition of a Set of Lists

Given a domain of list contents, \(D\), as described above, we define \(L_D\) to be the smallest set that satisfies Axioms \(L_1\)–\(L_5\), below. We call \(L_D\) the set of lists with domain \(D\).

\(L_1\) The empty list, denoted by \(\llbracket\), is an an element of \(L_D\).

\(L_2\) For each item \(d \in D\), and each list \(\ell \in L_D\), there is a constructed list in \(L_D\), denoted by \((d : \ell)\). We call \(d\) the first element of the constructed list, \((d : \ell)\); and we call \(\ell\) the rest of the constructed list, \((d : \ell)\).

\(L_3\) No constructed list is equal to the empty list.

\(L_4\) For any items \(d_1, d_2 \in D\), and any lists \(\ell_1, \ell_2 \in L_D\), if either \(d_1 \neq d_2\) or \(\ell_1 \neq \ell_2\), then \((d_1 : \ell_1) \neq (d_2 : \ell_2)\).

\(L_5\) Let \(P\) be any property for which the following conditions hold.

- \(P(\llbracket)\) holds.

---

1 Handout written by Luke Hunsberger, 2009
2 Notice that \(D\) is an ordinary, mathematical set, not a list.
For any \( d \in \mathcal{D} \) and any \( \ell \in \mathcal{L}_D \), if \( P(\ell) \) holds, then \( P(d : \ell) \) also holds.

Then \( P(\ell) \) holds for every \( \ell \in \mathcal{L}_D \).

Axioms \( L_1-L_5 \) parallel Axioms \( P_1-P_3 \) in the handout on the natural numbers. Of course, there are some differences. Below we consider each axiom in turn.

First, recall that Axiom \( P_1 \) stipulated that the natural numbers includes an element called 0. Similarly, Axiom \( L_1 \) stipulates that the set of lists contains the null element, \( \text{\textbullet} \), called the empty list.

Second, recall that Axiom \( P_2 \) stipulated that every natural number has a successor. Thus, Axiom \( P_2 \) can be construed as a constructor axiom: it stipulates how additional natural numbers can be constructed from existing numbers. Similarly, Axiom \( L_2 \) stipulates the construction of additional lists from existing lists. In particular, given some list, \( \ell \), you can construct a new list by prepending an item \( d \in \mathcal{D} \) onto the front of \( \ell \). The resulting list is denoted by \( (d : \ell) \).

The axiom also defines names for the constituents of this new list. In particular, \( d \) is called the first element of the new list, and \( \ell \) is called the rest of the list. Notice that \( d \in \mathcal{D} \), whereas \( \ell \in \mathcal{L}_D \). As we’ll see shortly, the names first and rest can be thought of as accessor functions: they give access to the parts of a constructed list. Incidentally, given the successor, \( \text{Succ}(n) \), of some natural number, \( n \), we can think of the predecessor function, \( \text{Pred} \), as an accessor function. In particular, \( \text{Pred}(\text{Succ}(n)) = n \).

Third, recall that Axiom \( P_3 \) stipulated that zero is not the successor of any natural number. Similarly, Axiom \( L_3 \) stipulates that the empty list cannot be obtained by constructing a list of the form, \( (d : \ell) \). In other words, the empty list is different from all of the constructed lists.

Fourth, recall that Axiom \( P_4 \) stipulated that different natural numbers have different successors. Similarly, Axiom \( L_4 \) stipulates that if the building blocks of two constructed lists differ in some way, then the resulting constructed lists are different. The building blocks can differ in two ways: \( d_1 \) might be different from \( d_2 \) or \( \ell_1 \) might be different from \( \ell_2 \). Of course, both pairs of building blocks might be different.

Fifth, recall that Axiom \( P_5 \) formed the basis for the proof procedure known as induction (over the natural numbers). Similarly, Axiom \( L_5 \) forms the basis for a proof procedure known as induction (over flat lists). The bulleted conditions in Axiom \( L_5 \) have the same general form as those in Axiom \( P_5 \). The main difference is that the second bulleted condition in Axiom \( L_5 \) includes the phrase, “for any \( d \in \mathcal{D} \)”.

2 PROOFS BY INDUCTION OVER FLAT LISTS

This section illustrates the process of proving theorems by induction over flat lists. The proof procedure is entirely analogous to that of induction over the natural numbers, except that we have to allow for the fact that the individual elements of a list can be arbitrary items in the domain \( \mathcal{D} \).

THEOREM 1. Every list in \( \mathcal{L}_D \) has one of two forms: either \( \text{\textbullet} \) or \( (d : \ell) \) for some \( d \in \mathcal{D} \) and some \( \ell \in \mathcal{L}_D \).

PROOF. Let \( P(\ell') \) be the property that \( \ell' \) has the form \( \text{\textbullet} \) or \( (d : \ell) \) for some \( d \in \mathcal{D} \) and some \( \ell \in \mathcal{L}_D \). We want to show that \( P(\ell') \) holds for all \( \ell' \in \mathcal{L}_D \).

- Consider \( P(\text{\textbullet}) \). It holds since \( \text{\textbullet} \) has the form \( \text{\textbullet} \).
Suppose $P(\ell_i)$ holds for some list $\ell_i$. Let $d_i$ be some arbitrary element of $D$. We must show that $P(d_i : \ell_i)$ also holds. But $(d_i : \ell_i)$ certainly has the form $(d : \ell)$ for some $d \in D$ and some $\ell \in L_D$.

Thus, by Axiom $L_5$, $P(\ell')$ holds for all $\ell' \in L_D$.

2.1 Recursive Functions over Flat Lists

Theorem 1 states that a list in $L_D$ has one of the following forms: either $\bullet$ or $(d : \ell)$. Thus, we can define a function, $f$, on all lists, as follows:

- Define $f(\bullet)$.
- Define $f(d : \ell)$ in terms of $d$ and $f(\ell)$.

Defining $f(\bullet)$ is the base case of the definition. Defining $f(d : \ell)$ in terms of $d$ and $f(\ell)$ is the recursive case of the definition. It is recursive because the definition of $f(d : \ell)$ depends on $f(\ell)$ being defined.

**Theorem 2.** If a function, $f$, is defined as described above, then it is defined for all lists in $L_D$.

**Proof.** Let $P(\ell)$ be the proposition that $f(\ell)$ is defined. The proof by induction is left as an exercise. (Use Theorem 2 in the previous handout as a guide.)

3 Some Recursively Defined Functions on Flat Lists

These section shows how a variety of recursive functions can be defined on flat lists. In each case, it gives a math-like definition and a Scheme-based definition. The Scheme definitions make use of the following built-in functions:

- The `cons` list-constructor function. This function takes an item $d \in D$ and some list $\ell \in L_D$ as its inputs. It returns as its output the list $(d : \ell)$. Of course, the syntax of Scheme makes things look a little different, as demonstrated by the following drScheme session:

  ```scheme
  > (cons 3 '(a b c))
  (3 a b c)
  > (cons '+ '(1 2 3))
  (+ 1 2 3)
  > (cons 1 '())
  (1)
  ```

- The `first` and `rest` accessor functions. These functions provide access to the parts of a constructed list, as demonstrated below.

  ```scheme
  > (first '(1 2 3))
  1
  > (first '(+ #t a))
  ```
+ (rest '(1 2 3)) (2 3) (rest '('(1))) () (rest ' (+ #t a #f)) (#t a #f)

It should be kept in mind that these functions will generate errors if they are applied
to the empty list! They should only be applied to constructed lists!
The following drScheme session illustrates that cons is sort of the dual of first and rest.

> (define listy '(a b c d e))
> (cons (first listy) (rest listy))
(a b c d e)
> (first (cons 'z listy))
z
> (rest (cons 'z listy))
(a b c d e)

• The null? type-checker predicate. This predicate returns boolean true if its input is the
empty list; otherwise, it returns boolean false, as illustrated below.

> (null? '()) #t
> (null? '(a b c d)) #f
> (null? (cons 1 '(a b c))) #f

Notice that an expression of the form, (null? (cons ... ...)), invariably evaluates to
boolean false.

The length of a list. The length of a list (i.e., the number of elements contained in
the list) can be recursively defined, as follows:

• \( f(\epsilon) = 0 \)

• \( f(d:\ell) = 1 + f(\ell) \)

In Scheme, the definition looks like this:

(define lengthy
 (lambda (listy)
   (if (null? listy)
     0
     (+ 1 (lengthy (rest listy))))))

3 Scheme includes a built-in function associated with the length symbol that computes the length of a list.
It behaves as follows:

```scheme
> (lengthy '())
0
> (lengthy '(a b c d e))
5
```

**Summing the Elements in a List.** The sum of the elements in a list can be recursively defined, as follows.

- \( g(\varepsilon) = 0 \)
- \( g(d : \ell) = d + g(\ell) \)

In Scheme, it looks like this:

```scheme
> (define summy
  (lambda (listy)
    (if (null? listy)
      0
      (+ (first listy)
          (summy (rest listy))))))
> (summy '(1 2 3))
6
> (summy '())
0
> (summy '(1 10 100 1000))
1111
```

Notice the use of indentation to make the Scheme definition easier to look at. For example, the expressions, \((\text{first listy})\) and \((\text{summy (rest listy)})\), which are both inputs to the + operator, are indented the same amount.

**Squaring the Elements in a List.** In this case, we want a function that takes a list of numbers as its input. It should return a list of the same length as its output, where each entry in the output list should be the square of the corresponding entry from the input list, as demonstrated below:

```scheme
> (square-all '(1 2 3 4 5))
(1 4 9 16 25)
> (square-all '(10 20 30 100))
(100 400 900 10000)
```

Such a function can be recursively defined as follows.

- \( h(\varepsilon) = \varepsilon \)
- \( h(d : \ell) = (d * d : h(\ell)) \)

Notice that if this function is given the empty list as its input, it returns the empty list as its output. However, if it is given a constructed list as its input, it returns a constructed list as its output. In either case, the output is a list.

Here's the Scheme definition:
(define square-all
  (lambda (listy)
    (if (null? listy)
        '()
      (cons (* (first listy) (first listy))
           (square-all (rest listy))))))

Determining whether an item is a member of a list. Given an item, \(x\), and a list, \(\ell\), we desire a function, \(F\), to return \(true\) if \(x\) is a member of \(\ell\); \(false\) otherwise. Notice that this function takes two inputs: an item and a list. However, we can still define it recursively:

- \(F(x, \bullet) = false\)
- \(F(x, (d : \ell)) = true\) if \(x = d\) or \(F(x, \ell) = true\)

Here’s a Scheme definition:

> (define memb
  (lambda (item listy)
    (if (null? listy)
        #f
      (or (= item (first listy))
        (memb item (rest listy))))))

> (memb 3 '(1 2 3 4 5))
#t
> (memb 3 '(1 2 4 5 6))
#f

Notice the use of the \(or\) special form instead of using a second \(if\) expression. This makes the code easier to read. We can take this one step further, as follows:

> (define memby
  (lambda (item listy)
    (and (not (null? listy))
      (or (= item (first listy))
        (memby item (rest listy))))))

> (memby 3 '(1 2 3 4 5))
#t
> (memby 3 '(1 2 4 5 6))
#f

You should walk through the code to convince yourself that it effectively implements the same function.

---

4 There is a built-in Scheme function associated with the \textit{member} symbol that does essentially the same thing as the \textit{memb} function defined here.
Below, the definitions of the length function, \( f \), and the square-all function, \( h \), are repeated.

\[
\begin{align*}
(1) \quad f(\texttt{\texttt{0}}) &= 0 \\
(2) \quad f(d : \ell) &= 1 + f(\ell) \\
(3) \quad h(\texttt{\texttt{0}}) &= \texttt{\texttt{0}} \\
(4) \quad h(d : \ell) &= (d * d : h(\ell))
\end{align*}
\]

Notice that the clauses in these definitions have been numbered for easy reference.

**Theorem 3.** Let \( \mathcal{D} \) be the set of natural numbers. For each \( \ell \in \mathcal{L}_{\mathcal{D}} \), the length of \( \ell \) is the same as the length of \( h(\ell) \). Stated more concisely: \( f(\ell) = f(h(\ell)) \).

**Proof.** Let \( P(\ell) \) be the proposition that \( f(\ell) = f(h(\ell)) \). We want to show that \( P(\ell) \) holds for all \( \ell \in \mathcal{L}_{\mathcal{D}} \).

- \( P(\texttt{\texttt{0}}) \) is: \( f(\texttt{\texttt{0}}) = f(h(\texttt{\texttt{0}})) \). Starting with the right-hand side, notice that \( f(h(\texttt{\texttt{0}})) = f(\texttt{\texttt{0}}) \) since \( h(\texttt{\texttt{0}}) = \texttt{\texttt{0}} \), by clause (2). Thus, \( P(\texttt{\texttt{0}}) \) holds.

- Suppose \( d \) is some element of \( \mathcal{D} \) and \( \ell \) is some element of \( \mathcal{L}_{\mathcal{D}} \) for which \( P(\ell) \) holds (i.e., \( f(\ell) = f(h(\ell)) \)). We need to show that \( P(d : \ell) \) holds (i.e., that \( f(d : \ell) = f(h(d : \ell)) \) holds). Starting with the right-hand side:

\[
\begin{align*}
\text{Right-hand side: } & f(h(d : \ell)) \\
= & f(d * d : h(\ell)), \text{ by (4)} \\
= & 1 + f(h(\ell)), \text{ by (2)} \\
= & 1 + f(\ell), \text{ since } P(\ell) \text{ holds} \\
= & f(d : \ell), \text{ by (2)} \\
= & \text{Left-hand side!}
\end{align*}
\]

Thus, by Axiom \( L_5 \), \( P(\ell) \) holds for all \( \ell \in \mathcal{L}_{\mathcal{D}} \).