Section 3.2: Recursively Defined Functions and Procedures

Function: Has inputs ("arguments", "operands") and output ("result")
No "side effects".

Procedure: May have side effects, e.g., "print(...)"

A recursive function (or procedure) calls itself!

A function f is recursively defined if at least one value of f(x) is defined in terms of another value, f(y), where x ≠ y.

Similarly: a procedure P is recursively defined if the action of P(x) is defined in terms of another action, P(y), where x ≠ y.

When an argument to a function is inductively defined, here is a technique for creating a recursive function definition:

1. Specify a value of f(x) for each basis element x in S.
2. For each inductive rule that defines an element x in S in terms of some element y already in S, specify rules in the function that compute f(x) in terms of f(y).
**Example:** Find a recursive definition for function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(n) = 0 + 3 + 6 + \ldots + 3n.$$  

e.g.,  

- $f(0) = 0$
- $f(1) = 0 + 3$
- $f(2) = 0 + 3 + 6$

**Solution:** Notice that $\mathbb{N}$ is an inductively defined set:

- $0 \in \mathbb{N}$
- $n \in \mathbb{N}$ implies $n+1 \in \mathbb{N}$

So we need to give $f(0)$ a value and we need to define $f(n+1)$ in terms of $f(n)$.

The value for $f(0)$ should be 0. What about $f(n+1)$?

$$f(n+1) = 0 + 3 + 6 + \ldots 3n + 3(n+1)$$
$$= f(n) + 3(n+1)$$

So here is our (recursive) definition for $f$:

- $f(0) = 0$
- $f(n+1) = f(n) + 3(n+1)$

We could also write:

- $f(0) = 0$
- $f(n) = f(n-1) + 3n$ for $n > 0$

Here is a more programming-like definition:

$$f(n) = (\text{if } n=0 \text{ then } 0 \text{ else } f(n-1)+3n \text{ endif })$$
Example: Find a recursive definition for
\[
\text{cat} : A^* \times A^* \rightarrow A^*
\]
defined by \( \text{cat}(s,t) = st \)

Solution: Notice that \( A^* \) is inductively defined.
Basis: \( \Lambda \in A^* \); Induction: \( a \in A \) and \( x \in A^* \) imply \( ax \in A^* \)

We can define \( \text{cat} \) recursively using the first argument.

The definition of \( \text{cat} \) gives
\[
\text{cat}(\Lambda, t) = \Lambda t = t.
\]
For the recursive part we can write
\[
\text{cat}(ax, t) = axt = a(xt) = a\text{cat}(x, t)
\]

Here is a definition:
\[
\text{cat}(\Lambda, t) = t
\]
\[
\text{cat}(ax, t) = a\text{cat}(x, t)
\]

Here is the if-then-else form:
\[
\text{cat}(s, t) = \begin{cases} t & \text{if } s = \Lambda \\ \text{head}(s)\text{cat}(\text{tail}(s), t) & \text{else} \end{cases}
\]
Example: Find a definition of \( f: \text{lists}(Q) \rightarrow Q \) defined by
\[
f(<x_1, \ldots, x_n>) = x_1 + \ldots + x_n
\]

Solution: Notice that the set \( \text{lists}(Q) \) is defined recursively.

Basis: \(<> \in \text{lists}(Q)\)
Induction: \( h \in Q \) and \( t \in \text{lists}(Q) \) imply \( h::t \in \text{lists}(Q) \)

To discover a recursive definition, we can use the definition of \( f \) as follows:
\[
f(<x_1, \ldots, x_n>)
= x_1 + x_2 \ldots + x_n
= x_1 + (x_2 + \ldots + x_n)
= x_1 + f(<x_2, \ldots, x_n>)
= \text{head}(<x_1, \ldots, x_n>) + f(\text{tail}(<x_1, \ldots, x_n>))
\]

So, here is our recursive definition:
\[
f(<>) = 0
f(h::t) = h + f(t)
\]
Expressing this in the if-then-else form:
\[
f(L) = \text{if } L=<> \text{ then } 0 \text{ else } \text{head}(L)+f(\text{tail}(L))
\]
Example:  Given $f: \mathbb{N} \rightarrow \mathbb{N}$ as defined by
\[
\begin{align*}
  f(0) &= 0 \\
  f(1) &= 0 \\
  f(x+2) &= 1 + f(x)
\end{align*}
\]

Here is the if-then-else formulation:
\[
f(x) = \text{if } (x=0 \text{ or } x=1) \text{ then } 0 \text{ else } 1 + f(x-2)
\]

What exactly does this function do?

Let’s try to get an idea by enumerating a few values.
\[
\text{map}(f, <0,1,2,3,4,5,6,7,8,9>) = <0,0,1,1,2,2,3,3,4,4>
\]

So $f(x)$ returns the floor of $x/2$. That is, $f(x) = \lfloor x/2 \rfloor$. 

Example:  Find a recursive definition for the function \( f: \text{lists}(\mathbb{Q}) \rightarrow \mathbb{Q} \) as defined by:
\[
f(<x_1, ..., x_n>) = x_1x_2 + x_2x_3 + ... + x_{n-1}x_n
\]

Approach:
Let \( f(<>) = 0 \) and \( f(<x>) = 0 \). Then for \( n \geq 2 \) we can write:
\[
f(<x_1, ..., x_n>)
\]
\[
= x_1x_2 + x_2x_3 + ... + x_{n-1}x_n
\]
\[
= x_1x_2 + (x_2x_3 + ... + x_{n-1}x_n)
\]
\[
= x_1x_2 + f(<x_2, ..., x_n>)
\]
So here is our recursive definition:
\[
f(<>) = 0
\]
\[
f(<x>) = 0
\]
\[
f(h::t) = h \cdot \text{head}(t) + f(t).
\]

We can express this in if-then-else form as:
\[
f(L) = \begin{cases} 
0 & \text{if } (L=<> \text{ or } \text{tail}(L)=<>) \\
\text{if} & \text{else} \\
\text{head(L)} \cdot \text{head(tail(L))} + f(tail(L)) & \text{endIf}
\end{cases}
\]
Example: Find a recursive definition for the function

\[ \text{isin} : A \times \text{lists}(A) \rightarrow \{\text{true, false}\} \]

where \( \text{isin}(x,L) \) means that \( x \) occurs in the list \( L \).

Solution:

\[
\text{isin}(x,<>) = \text{false} \\
\text{isin}(x,x::t) = \text{true} \\
\text{isin}(x,y::t) = \text{isin}(x,t), \text{ where } x \neq y
\]

Here’s the if-then-else form:

\[
\text{isin}(x,L) = \text{if } L=<> \\
\text{then} \\
\text{false} \\
\text{else} \\
\text{if } x=\text{head}(L) \\
\text{then} \\
\text{true} \\
\text{else} \\
\text{isin}(x,\text{tail}(L)) \\
\text{endIf} \\
\text{endIf}
\]
**Example:** Find a recursive definition for the function

\[ \text{isin} : A \times \text{lists}(A) \rightarrow \{\text{true, false}\} \]

where \( \text{isin}(x, L) \) means that \( x \) occurs in the list \( L \).

**Solution:**

- \( \text{isin}(x, <> ) = \text{false} \)
- \( \text{isin}(x, x::t) = \text{true} \)
- \( \text{isin}(x, y::t) = \text{isin}(x, t), \text{where } x \neq y \)

Here’s the if-then-else form:

\[
\text{isin}(x, L) = \begin{cases} 
\text{false} & \text{if } L=<> \\
\text{false} & \text{then} \\
\text{else} & x=\text{head}(L) \text{ or } \text{isin}(x, \text{tail}(L)) \\
\text{endIf}
\end{cases}
\]
**Example:** Find a recursive definition for
\[
\text{sub}: \text{lists}(A) \times \text{lists}(A) \rightarrow \{\text{true}, \text{false}\}
\]
where \(\text{sub}(L,M)\) means the elements of \(L\) are elements of \(M\).

**Solution:**

Here is a pattern-matching solution:
\[
\text{sub}(<>,M) = \text{true}
\]
\[
\text{sub}(h::t,M) = \text{if isin}(h,M) \text{ then sub}(t,M) \text{ else false}
\]

Here is a programmatic (executable) version:
\[
\text{sub}(L,M) = \text{if L} = <>
\]
\[
\text{then true}
\]
\[
\text{else if isin(head(L),M) then sub(tail(L),M) else false}
\]
\[
\text{endIf}
\]
\[
\text{endIf}
\]
Example: Find a recursive definition for
\[
\text{intree: } Q \times \text{binSearchTrees}(Q) \to \{\text{true, false}\}
\]
where \( \text{intree}(x, T) \) means \( x \) is in the binary search tree \( T \).

Solution:
\[
\begin{align*}
\text{intree}(x, <> ) &= \text{false} \\
\text{intree}(x, <L, x, R>) &= \text{true} \\
\text{intree}(x, <L, y, R>) &= \text{if } x < y \text{ then } \text{intree}(x, L) \text{ else } \text{intree}(x, R)
\end{align*}
\]

Why is this a better definition?
\[
\begin{align*}
\text{intree}(x, <> ) &= \text{false} \\
\text{intree}(x, <L, y, R>) &= \begin{cases} 
\text{true, if } x = y \\
\text{intree}(x, L), \text{ if } x < y \\
\text{intree}(x, R), \text{ if } x > y 
\end{cases}
\end{align*}
\]

Here is the if-then-else form:
\[
\begin{align*}
\text{intree}(x, T) &= \begin{cases} 
\text{if } T = <> \text{ then false} \\
\text{elseif } x = \text{root}(T) \text{ then true} \\
\text{elseif } x < \text{root}(T) \text{ then } \text{intree}(x, \text{left}(T)) \\
\text{else } \text{intree}(x, \text{right}(T)) 
\end{cases}
\end{align*}
\]
Traversing Binary Trees

There are 3 ways to traverse a binary tree. Each is defined recursively.

**preorder(T):** if T≠<> then
  visit root; preorder(left(T)); preorder(right(T))

**inorder(T):** if T≠<> then
  inorder(left(T)); visit root; inorder(right(T))

**postorder(T):** if T≠<> then
  postorder(left(T)); postorder(right(T); visit root

**Example:** Traverse this tree in each of the orders:

```
  a
 /   \
/     /
b     c
|     |
|     |
d     e
```

**Solution:**
pre-order: 
in-order: 
post-order:
Traversing Binary Trees

There are 3 ways to traverse a binary tree. Each is defined recursively.

preorder(T): if T\neq\emptyset then
visit root; preorder(left(T)); preorder(right(T))

inorder(T): if T\neq\emptyset then
inorder(left(T)); visit root; inorder(right(T))

postorder(T): if T\neq\emptyset then
postorder(left(T)); postorder(right(T); visit root

Example: Traverse this tree in each of the orders:

```
   a
  / \  
 b   c
 /   /
 d   e
```

Solution: pre-order: \(a\ (b\ d\ e)\ (c)\)
in-order: \((d\ b\ e)\ a\ (c)\)
post-order: \((d\ e\ b)\ (c)\ a\)
Example: Find a recursive definition for
post: binaryTrees(A) → lists(A)
where post(T) is the list of nodes from a post-order traversal of T.

Solution:
post(<> ) = <>
post(<L,x,R>) = cat(post(L),cat(post(R),<x>))

The function cat will concatenate two lists, and can be defined as:
cat(<> ,L) = L
cat(h::t,L) = h::cat(t,L)

Example: Find a recursive definition for
sumnodes: binaryTrees(Q) → Q
where sumnodes(T) returns the sum of the nodes in T.

Solution:
sumnodes(<> ) = 0
sumnodes(<L,x,R>) = x + sumnodes(L) + sumnodes(R)
Infinite Sequences
We can construct recursive definitions for infinite sequences by defining a value f(x) in terms of x and f(y) for some value y in the sequence.

Example: Suppose we want to define a function f that returns an infinite sequence. The function f should return this sequence:

\[ f(x) = <x^1, x^2, x^4, x^8, x^{16}, \ldots > \]

Approach:
Look at the definition and try to find a solution:

\[
\begin{align*}
f(x) &= <x^1, x^2, x^4, x^8, x^{16}, \ldots > \\
&= x :: <x^2, x^4, x^8, x^{16}, \ldots > \\
&= x :: f(x^2)
\end{align*}
\]

So we can define:
\[ f(x) = x :: f(x^2) \]
This function returns an infinite sequence.

Q: Of what use is such a function in computing???
A: We can use “lazy evaluation”: When we need an element from f(x), we’ll need to evaluate f. Yes, this is an infinite computation, but we’ll do only as much work as necessary to get the element we need.
**Example:** What sequence is defined by $g(x,k) = x^k :: g(x,k+1)$?

**Solution:**

$$
g(x,k) = x^k :: g(x,k+1)
= x^k :: x^{k+1} :: g(x,k+2)
= <x^k, x^{k+1}, x^{k+2}, ...>
$$

**Example:** How do we obtain the sequence $<x, x^3, x^5, x^7, ...>$?

**Solution:** Define $f(x) = h(x,1)$
where $h(x,k) = x^k :: h(x,k+2)$

**Example:** How do we obtain the sequence $<1, x^2, x^4, x^6, x^8, ...>$?

**Solution:** Define $f(x) = h(x,0)$, where $h$ is from the previous example.