Recursion, Mathematical Datatypes and Functional Programming

R. Sekar
(Pure) Functional Programming

- Programs consist of a set of functions
- These functions behave like as functions in Mathematics.
- Variables are like variables in algebra or logic
  - They are placeholders for (input) values
  - Unlike most programming languages (e.g., Java), there is no concept of “assigning” to them or “modifying” their values.
  - The term “absence of side-effects” is used to described this property
- Data structures are based on mathematical types
  - Sets
  - Cartesian products and other type constructors to construct new types from existing ones.
ML was developed initially as a “meta language” for a theorem proving system (Logic of Computable Functions).

The two main dialects, SML and CAML, have many features in common:
- data type definition, type inference, interactive top-level, ... 

SML syntax is closer to mathematics, so we will use it in this course.

CAML has more features for programming (e.g., object-orientation). It is a more modern project, and hence prefer it over SML in real projects.
Installing Standard ML of New Jersey (SML/NJ)

**Ubuntu/Debian Linux:** Run `sudo apt install smlnj` at the command-line

**Windows:** Your best bet is likely to be an install within a Windows subsystem for Linux (WSL): [https://docs.microsoft.com/en-us/windows/wsl/install-win10](https://docs.microsoft.com/en-us/windows/wsl/install-win10)
- Within the Linux subsystem, follow the above instructions.

**On a browser:** You can also run SML directly on your web browser at [https://www.tutorialspoint.com/execute_smlnj_online.php](https://www.tutorialspoint.com/execute_smlnj_online.php). No installation is required, but you have to cut/paste everything from local files on your machine (or else your programs may not be saved)

**Other:** Please see [https://www.smlnj.org/](https://www.smlnj.org/) and especially [http://www.smlnj.org/install/](http://www.smlnj.org/install/)
Using SML

- On Linux, run `sml` on the command-line. If you are using it in a browser, you will already be at the top-level SML prompt.
- SML prompts with `-`
- Enter new code definitions, evaluate expressions, or issue directives at the prompt.
- Control-D to exit SML
- We will use SML interactive toplevel throughout for examples.

Run `sml` inside readline wrapper as follows:
```
rlwrap sml
```
in order to be able to go up/down and correct your mistakes.
Instead of typing our program at the prompt, we usually enter it into a file and execute it.

- Type the command `use "abc"` in order to load the file `abc` in the current directory.
- Contents of the file are processed as if they were directly typed in
- Not available on the browser (no access to files).
  - You can instead cut/paste from a file stored on your computer.
Expressions

Examples:

<table>
<thead>
<tr>
<th>User Input</th>
<th>SML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 * 3;</td>
<td>val it = 6 : int</td>
</tr>
</tbody>
</table>

- **val**: Indicates that the result (aka output) is a value
- **it**: variable representing the output of the last expression
- **6**: the number 6
- **:**: separator between the output and its type
- **int**: the type of output

A few key points:

- Use semicolons at the end of definitions. Spurious semicolons in the middle confuse SML.
- When you type a multi-line definition, you get a secondary prompt = from SML.
  - When you finish the definition and type a semicolon, SML goes back to its top-level prompt -
  - If SML is confused and prints = when you think you are done with a definition, press Ctrl-C to get back to the primary prompt.
## Expression Evaluation (Contd.)

More examples:

<table>
<thead>
<tr>
<th>User Input</th>
<th>SML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 3 * 4;</td>
<td>val it = 14 : int</td>
</tr>
<tr>
<td>~2 + 3 * 4;</td>
<td>val it = 10 : int</td>
</tr>
<tr>
<td>(2 + 3) * 4;</td>
<td>val it = 20 : int</td>
</tr>
<tr>
<td>4.4 * 2.0;</td>
<td>val it = 9.68 : real</td>
</tr>
</tbody>
</table>
| 2 + 2.2;         | stdIn:15.1-15.8 Error: operator and operand don’t agree [overload conflict]  
|                  | operator domain: [+ ty] * [+ ty]  
|                  | operand: [+ ty] * real  
|                  | in expression:  
|                  | 2 + 2.2                                              |
## Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>+, -</td>
<td>Integer or real arithmetic</td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>true, false</td>
<td>Boolean constants</td>
</tr>
<tr>
<td>div</td>
<td>Integer division</td>
</tr>
<tr>
<td>/</td>
<td>Real number division</td>
</tr>
<tr>
<td>andalso, orelse, not</td>
<td>Boolean operations</td>
</tr>
</tbody>
</table>
Value definitions

- Syntax: `val ⟨name⟩ = ⟨expression⟩ ;`

- Examples:

<table>
<thead>
<tr>
<th>User Input</th>
<th>SML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>val x = 1;</code></td>
<td><code>val x = 1: int</code></td>
</tr>
<tr>
<td><code>val y = x + 1;</code></td>
<td><code>val y = 2 : int</code></td>
</tr>
<tr>
<td><code>val x = x + 1;</code></td>
<td><code>val x = 3 : int</code></td>
</tr>
<tr>
<td><code>val z = &quot;SML rocks!&quot;;</code></td>
<td><code>val z = &quot;SML rocks!&quot;: string</code></td>
</tr>
</tbody>
</table>
## Functions

**Syntax:** \( \text{fun} \langle \text{name} \rangle (\langle \text{arguments} \rangle) = \langle \text{expression} \rangle ; \)

**Examples:**

<table>
<thead>
<tr>
<th>User Input</th>
<th>SML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{fun} \textit{f}(x) = 1;</td>
<td>\textbf{val} f = fn : `'a -&gt; int</td>
</tr>
<tr>
<td>\textbf{fun} \textit{g}(x) = x;</td>
<td>\textbf{val} g = fn : <code>'a -&gt; </code>'a</td>
</tr>
<tr>
<td>\textbf{fun} \textit{inc}(x) = x + 1;</td>
<td>\textbf{val} inc = fn : int -&gt; int</td>
</tr>
<tr>
<td>\textbf{fun} \textit{sum}(x,y) = x+y;</td>
<td>\textbf{val} sum = fn : int * int -&gt; int</td>
</tr>
</tbody>
</table>
| \textbf{fun} \textit{max}(x, y) = \begin{cases} 
  x & \text{if } x < y \\
  y & \text{else} \end{cases} | \textbf{val} max = fn : int * int -> int |
| max(2,3);                | \textbf{val} it = 3 : int       |
Let Statements

- We often want to break down a complex expression into a series of steps to make it easier to understand.

  \[
  \text{let } \langle \text{definition} \rangle \ \text{in} \ \langle \text{expression} \rangle;
  \]

- A let can define one or more values (left), functions (right), or a mix of them.

  - fun f(x, y) = let
    \[
    \begin{align*}
    \text{val } x2 &= x \ast x \\
    \text{val } y2 &= y \ast y
    \end{align*}
    \]
  in
  \[
  x2+y2
  \]
  end;

  - fun f(x, y) = let
    \[
    \begin{align*}
    \text{fun } sq(z) &= z \ast z \\
    \text{val } x2 &= x \ast x \\
    \text{val } y2 &= y \ast y
    \end{align*}
    \]
  in
  \[
  \text{sq}(x) \ast \text{sq}(y)
  \]
  end;

  - fun f(x, y) = x \ast x + y \ast y;
The built-in function `print` prints strings (and only strings)

```
print;
val it = fn : string -> unit
```

Use predefined functions `Int.toString` and `Real.toString` to covert int’s and real’s to strings.

Use string concatenation operator `^` to construct complex strings.

```
print("I know 10 * 20 is " ^ Int.toString(10*20) ^ "\n");
val it = () : unit
```
Recursion

fun \( g(0) = 1 \) (* Base Case 1 *)
| \( g(1) = 1 \) (* Base Case 2 *)
| \( g(n) = g(n-1) + g(n-2); \) (* Recursive Case *)

fun \( f(0) = 1 \) (* Base Case *)
| \( f(n) = 2*f(n-1); \) (* Recursive Case *)

fun \( h(1) = 1 \) (* Base Case *)
| \( h(n) = 2*h(n \text{ div } 2); \) (* Recursive Case *)
fun fac(0) = 1
|   fac(n) = n * fac(n-1);

fun euclid(0, b) = b
|  euclid(a, b) = euclid(b mod a, a);
fun gcd(a,b) = if a < b then euclid(a, b) else euclid(b,a);
Example: Euclid’s Algorithm for GCD

Base: \( \text{euclid}(0, x) = x \)

Induction step: \( \text{euclid}(a, b) = \begin{cases} \text{euclid}(b \mod a, a) \end{cases} \)

\( \text{gcd}(a, b) = \text{gcd}(b \mod a, a) \)
Using Recurrences to Analyze Runtime of Recursive Programs

Example: Factorial

\[ \text{fac}(0) = 1 \]
\[ \text{fac}(n) = n \times \text{fac}(n-1) \]
\[ T(0) = 1 \]
\[ T(n) = T(n-1) + 1 \]

\[ T(n) = n + 1 \]
\[ T(n) = T(n-1) + 1 = T(n-2) + 1 + 1 = T(n-3) + 1 + 1 + 1 \ldots \]
\[ T(n) = O(n) \]
Using Recurrences to Analyze Runtime of Recursive Programs

Example: Fibonacci Sequence

\[
\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2)
\end{align*}
\]

\[
T(n) > \text{fib}(n)
\]

\[
\text{fib}(n) = \frac{1}{\sqrt{5}} \left( \frac{\sqrt{5} + 1}{2} \right)^n
\]

\[
T(0) = 1 \\
T(1) = 1 \\
T(n) = T(n-1) + T(n-2) + 1.6^n
\]
Why is Fibonacci So Slow?

\[ \text{fib}(n) \sim \text{fib}(2) \text{ roughly } 2^n \]

\[ \text{fib}(0) \rightarrow \text{fib}(1) \rightarrow \text{fib}(2) \rightarrow \text{fib}(3) \rightarrow \ldots \]
More Efficient Version of Fibonacci

- Recursion is top-down: may sometimes perform redundant computations.
- If we compute inductively, i.e., bottom-up, then we can avoid this inefficiency.
  - Compute the sequence $fib(0), fib(1), fib(2), \ldots$
  - Use the two preceding values in the sequence to compute the next value
  - Can compute in linear time.
- Write a helper function that uses the last two Fibonacci #s to compute the next one.

```plaintext
fun f(0, x, y) = x + y (* x is the last Fibonacci #, and y the previous one*)
| f(n, x, y) = f(n-1, x+y, x);

(* The following function handles base cases, calls f for other cases*)
fun fib(n) = if (n=0) then 0 else if (n=1) then 1 else f(n-2, 1, 0);

fib(4) = f(2, 1, 0) = f(1, 1, 1) = f(0, 2, 1) = 3
```
More Efficient Version of Fibonacci

- Recursion is top-down: may sometimes perform redundant computations.
- If we compute inductively, i.e., bottom-up, then we can avoid this inefficiency.
  - Compute the sequence \( \text{fib}(0), \text{fib}(1), \text{fib}(2), \ldots \)
  - Use the two preceding values in the sequence to compute the next value
  - Can compute in linear time.

- Write a helper function that uses the last two Fibonacci #s to compute the next one.

```ml
fun f(0, x, y) = x + y (* x is the last Fibonacci #, and y the previous one*)
| f(n, x, y) = f(n-1, x+y, x);

(* The following function handles base cases, calls f for other cases*)
fun fib(n) = if (n=0) then 0 else if (n=1) then 1 else f(n-2, 1, 0);
```
(Mathematical) Data Types in SML

- **Pre-defined Sets**
  - Built-in Sets: $\text{int} \subseteq \mathbb{N}$, $\text{real} \subseteq \mathbb{R}$, bool, string
  - Programmer-defined sets ("enumerated types")
    ```sml
datatype Suit = Spades | Clubs | Diamonds | Hearts;
    (* the type corresponding to the set \{Spades, Clubs, Diamonds, Hearts\} *)

datatype Rank = One | Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten | Jack | Queen | King | Ace;
```

- **Tuples:** Cartesian Products of Sets
  ```sml
type Card = Suit * Rank; (* Type corresponding to the set \text{Suit} \times \text{Rank} *)
type PokerHand = Card * Card * Card * Card * Card;
    (* A hand with five cards. Note that this is not a set: the order matters.*)
```

- **Sequences:** A list in SML represents the type $\bigcup_{i=0}^{\infty} S^i$ for any set $S$.
  ```sml
type Hand = Card list;
```
## List Examples in SML

<table>
<thead>
<tr>
<th>User Input</th>
<th>SML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[1];</code></td>
<td><code>val it = [1] : int list</code></td>
</tr>
<tr>
<td><code>[4.1, 2.7, 3.1];</code></td>
<td><code>val it = [4.1, 2.7, 3.1] : real list</code></td>
</tr>
<tr>
<td><code>[4.1, 2];</code></td>
<td><code>Error: operator and operand don’t agree [overload conflict]</code></td>
</tr>
<tr>
<td></td>
<td><code>operator domain: real * real list</code></td>
</tr>
<tr>
<td></td>
<td><code>operand: real * [int ty] list</code></td>
</tr>
<tr>
<td></td>
<td><code>in expression:</code></td>
</tr>
<tr>
<td></td>
<td><code>4.1 :: 2 :: nil</code></td>
</tr>
<tr>
<td><code>[[1,2],[4,8,16]];</code></td>
<td><code>val it = [[1,2], [4,8,16]] : int list list</code></td>
</tr>
<tr>
<td><code>1::2::[]</code></td>
<td><code>val it = [1, 2] : int list</code></td>
</tr>
</tbody>
</table>

```
Tuple Examples in SML

(2,"Andrew") : int * string
(true,3.5,"x") : bool * real * string
((4,2),(7,3)) : (int * int) * (int * int)
[(2,3),(2,2),(9,1)] : (int * int) list

- Lists are homogeneous: All elements must have the same type.
- But tuples are heterogeneous: components can be of different types.
Functions on Lists

- Like functions on natural numbers, functions on lists also use base and recursive cases.
  - For integers, the base case corresponds to \( n = 0 \)
  - For lists, the base case typically corresponds to the empty list
- We use pattern-matching to distinguish between cases
  - On the surface, this looks very similar to what we have seen before
- Note that \texttt{nil} and \texttt{[]} both denote empty lists
- Non-empty lists are of the form \texttt{h::t} where \texttt{h} denotes the first element of the list (“head”) and \texttt{t} denotes the rest of the list (“tail”)
  - For a list of type \texttt{’a list}, the head has the type \texttt{’a} while the tail has the type \texttt{’a list}
- List constants are typically specified without using ::, e.g., \texttt{[1, 2, 3, 4]}
  - But you can use :: if you wanted: \texttt{1::2::3::4::nil}
  - You can also mix and match the :: and [] notations, e.g., \texttt{1::2::[3,4]} or \texttt{1::2::[]}
Example List Functions

fun \text{length}(\text{nil}) = 0.
| \text{length}(x::xs) = 1 + \text{length}(xs);

fun \text{append}(\text{nil}, y) = y
| \text{append}(x::xs, y) = x::\text{append}(xs, y);

fun \text{reverse}(\text{nil}) = \text{nil}
| \text{reverse}(x::xs) = \text{append}(\text{reverse}(xs), [x]);

There is an infix operator \( @ \) for append: e.g., \( [1, 2] @ [3, 4] \) yields \( [1, 2, 3, 4] \).
Sieve of Eratosthenes

- Start with a list of natural numbers > 1
- Take the first number that has not been crossed out, and output it
  - It is a prime number
- Cross out all multiples of this number
- Repeat until the list is exhausted.
Sieve of Eratosthenes: Approach

1. Define a function `gen` to generate a list of integers

2. Define a helper function `elim` that takes a list and a number as arguments, and deletes all multiples of the number from the list

3. Define the `sieve` function that
   - takes the current list as input,
   - moves the first element to a list of prime numbers
   - uses `elim` to eliminate multiples of this number from the list,
   - and finally, repeats this until done

4. Define a top-level function that generates a list and then invokes `sieve` on it.
fun gen(0, m) = [] (* 1st arg is number of elems, m is the next elem *)
  | gen(n, m) = m::gen(n-1, m+1);

  gen(5, 3) = 3:: gen(4, 4) = 3:: 4:: gen(3, 5) = 3:: 4:: 5:: gen(2, 6) = 3:: 4:: 5:: 6:: gen(1, 7) = 3:: 4:: 5:: 6:: 7:: []

fun elim([], n) = []
  | elim(x::xs, n) = if (x mod n = 0) then elim(xs, n) else x::elim(xs, n);

fun sieve([], primes) = primes
  | sieve(n::ns, primes) = sieve(elim(ns, n), n::primes);

fun dosieve(n) = sieve(gen(n-1, 2), []);
Lists are *not* sets: they represent sequences.

But we can represent sets using lists: keep list elements in a sorted order, and eliminate duplicates.

If we start with lists that satisfy these properties, then we can implement set operations easily using list operations.
Set Operations in SML

- Start with the definition of Cartesian product
  
  \[ X \times Y = \{(x, y) | x \in X \land y \in Y\} \]

- To express it in SML, break into two steps:
  - First iterate through the set \( Y \):
    
    \[
    \text{fun } \text{cart1}(x, \text{[]}) = \text{[]} \\
    | \text{cart1}(x, y::ys) = (x, y)::\text{cart1}(x, ys);
    \]
  - then iterate through the set \( X \)
    
    \[
    \text{fun } \text{cart}([\text{}], ys) = \text{[]} \\
    | \text{cart}(x::xs, ys) = \text{cart1}(x, ys) @ \text{cart}(xs, ys);
    \]
Functions

Set Operations in SML

\[ X \cup Y = \{ a | a \in X \lor a \in B \} \]

In the SML code below, we iterate simultaneously through \( X \) and \( Y \). We have some extra work over the above definition so that we maintain elements in sorted order.

```sml
fun union([], ys) = ys
| union(xs, []) = xs
| union(x::xs, y::ys) = if (x = y)
  then x::union(xs, ys)
  else if (x < y)
  then x::union(xs, y::ys)
  else y::union(x::xs, ys);
```
Set Operations in SML: Exercises

- **Membership**

- **Intersection**
  \[ X \cap Y = \{ a | a \in X \land a \in Y \} \]

- **Subset**
  \[ X \subseteq Y \text{ iff } X \cup Y = Y \]

- **Difference**
  \[ X - Y = \{ a | a \in X \land a \notin Y \} \]

- **Complement**
  Test your implementation by checking De Morgan’s laws
Recall that a binary relation $R : X \rightarrow Y$ is a subset of $X \times Y$. For instance, the following list represents a binary relation from $\mathbb{N}$ to $\mathbb{N}$:

$$[(1,3), (2,4), (1,4), (3,5)]$$

```ml
fun getDomain([]) = []
| getDomain((x,y)::xys) = union([x], getDomain(xys));

fun getRange([]) = []
| getRange((x,y)::xys) = union([y], getRange(xys));

fun isTotal(A, R) = (A = getDomain(R));

fun isOnto(B, R) = (B = getRange(R));
```

4 surjective
Functions

Relations

fun getTails([]) = []
| getTails((x,y)::xys) = x::getTails(xys);

fun getHeads([]) = []
| getHeads((x,y)::xys) = y::getHeads(xys);

fun isFun(R) = (getDomain(R) = getTails(R));

fun isOneToOne(R) = (getRange(R) = getHeads(R));

fun isBijection(X, Y, R) = isFun(R) andalso isTotal(X, R)
andalso isOneToOne(R) andalso isOnto(Y, R);
Composition of relations $R : X \rightarrow Y$ and $S : Y \rightarrow Z$

$$R \circ S = \{(x, z) \mid \exists y (x, y) \in R \land (y, z) \in S\}$$

Compute $R^n$, where $R : X \rightarrow X$ is a binary relation on $X$.

Check for reachability in a graph

Check if a given walk (or path or circuit) is valid

Compute paths in a graph
Generating Combinations (Subsets of Given Size)

- Generate subsets of size $n$ that include the first element
  - Combine first element with any $n - 1$-member subset of the remaining elements
- Generate subsets of size $n$ that don’t include the first element
  - In this case, we are generating $n$-member subsets of the remaining elements

```ml
fun prefixAll([], x) = [] (* put x at the beginning of all lists in the first arg *)
| prefixAll(xs::xss, x) = (x::xs)::prefixAll(xss, x);

fun subsets(xs, 0) = [[]] (* All sets have ONE subset of size 0 *)
| subsets([], n) = [] (* An empty set has NO subsets of size > 0 *)
| subsets(x::xs, n) = prefixAll(subsets(xs, n-1), x) @ subsets(xs, n);

fun genlist(0, s) = []
| genlist(r, s) = s::genlist(r-1, s+1);

fun choose(m, n) = List.length(subsets(genlist(m, 1), n));
```
Generating Permutations

- Generate all permutations of all but the first element
- Insert the first element at every possible position in each of these permutations.

```plaintext
fun insertAt(0, xs, y) = y::xs (* Insert y into xs at position given by 1st argument *)
| insertAt(n, x::xs, y) = x::insertAt(n-1, xs, y);

fun insAtAll(xs, y) = (* Returns |xs|+1 lists: ith list is xs with y inserted at ith pos *)
let fun doInsAtAll(0, xs, y) = [insertAt(0, xs, y)]
| doInsAtAll(n, xs, y) = insertAt(n, xs, y)::doInsAtAll(n-1, xs, y)
in doInsAtAll(List.length(xs), xs, y) end;

fun insAtAllIntoAll([], x) = [] (* insert x into all possible pos in all lists in the 1st arg *)
| insAtAllIntoAll(zs::zss, x) = insAtAll(zs, x) @ insAtAllIntoAll(zss,x);

fun permute([], ) = [[]] (* Not [] but [[]]: there is one permutation of empty list *)
| permute(x::xs) = insAtAllIntoAll(permute(xs), x);
```
Counting: \( n \) of \( m \) Books, Omit \( k \) after a selected book

- An analytical solution to this problem required a new way of thinking, but its code is not far from the \( \binom{m}{n} \) problem
- Just discard \( k \) elements in one of the two recursive cases
- The program is simple because it is brute-force:
  - Recursive calls implicitly construct the tree diagram, and then we simply count the leaves
- Sometimes, it is easier to apply conditions in a post-processing phase.

```haskell
fun dropN(0, xs) = xs 
| dropN(n, []) = []
| dropN(n, x::xs) = dropN(n-1, xs)

fun books(xs, 0, k) = [[]] (* All sets have ONE subset of size 0 *)
| books([], n, k) = [] (* An empty set has NO books of size > 0 *)
| books(x::xs, n, k) = 
  prefixAll(books(dropN(k, xs), n-1, k), x) @ books(xs, n, k);

fun countBooks(m, n, k) = List.length(books(genlist(m, 1), n, k));
```
Generate the Powerset of a given set

Generate all possible words from a given string, also count their number.

How many numbers between 1 to 1M contain the digit 3?

In how many ways can we pick $n$ fruits subject to the following constraints?
- The number of apples must be even.
- The number of bananas must be a multiple of 5.
- There can be at most four oranges.
- There can be at most one pear.

How many 5-card hands have a jack, queen, or king, but not all of them?
Constructors are operators that construct new data from existing data. They:

- combine multiple data elements into one
- attach a “tag” to the combination

Tags are used to distinguish base and inductive cases

```
datatype iTree = Leaf of int (* Base case *)
  | BNode of int * iTree * iTree (* Inductive case 1 *)
  | SNode of int * iTree (* Inductive case 2 *)
```

Tree Denoted by

```
BNode(1,
  BNode(2,
    Leaf(4),
    Leaf(5))
  SNode(3, Leaf(6)))
```
Lists are Recursive Types!

- They could have been defined as:
  
  ```
  datatype 'a list = nil          (* Base case *)
  | :: of 'a * 'a list          (* Recursive case *)
  ```

- Starting from an arbitrary type A, we are constructing a recursive type for representing a sequence of (0 or more) A’s.

- In SML, names prefixed with single quotes are used to refer to type variables that can stand for an arbitrary type such as A.

- Because SML predefines this particular recursive type, we are able to use infix notation for :: rather than the usual prefix notation for type constructors.

```
- nil;
val it = [] : 'a list
- 1::2::nil;
val it = [1,2] : int list
```
More on Binary Trees

- Complete or Full Trees: every internal node has two children.
- Perfectly balanced: all root-to-leaf paths have the same length.
  - What is the number of nodes in a perfectly balanced binary tree of height $h$?
  - What fraction of nodes are leaved?
- Balanced: At every node, the height of left and right children differ by at most one.
Functions on Binary Trees: height, leaves, nodes

fun height (Leaf(x)) = 0
  | height (SNode(x, t)) = 1 + height (t)
  | height (BNode(x, t1, t2)) = 1 + max (height (t1), height (t2))

fun nleaves (Leaf(x)) = 1
  | nleaves (SNode(x, t)) = nleaves (t)
  | nleaves (BNode(x, t1, t2)) = nleaves (t1) + nleaves (t2)
Binary Trees and Searching

- All nodes in the left subtree are less than the value at the node
- All nodes in the right subtree are greater than the value at the node
- Search is efficient: takes $O(\log n)$ time, where $n$ is the number of nodes in the tree.

![Diagram of a binary tree with annotations for log n and total number of nodes.](image-url)

Total # of nodes = \[1 + 2 + 4 + \ldots + 2^h = \frac{2^{h+1} - 1}{2 - 1} = \frac{2^{h+1}}{2 - 1} - 1\]
Tree Traversals

- Prefix traversal: visit node before either children
  
  \[
  \text{Preorder}(t): \quad 7 \ 4 \ 2 \ 1 \ 0 \ 3 \ 8
  \]

- Postfix traversal: visit both children before the node

- Infix traversal: visit left child, then node, then right child
Propositions

datatype PropFormula = T
| F
| VAR of int
| NOT of PropFormula
| AND of PropFormula*PropFormula
| OR of PropFormula*PropFormula
| IMPL of PropFormula*PropFormula;

Example: $x_1 \land x_2 \rightarrow \neg x_3$ becomes IMPL(AND(VAR(1),VAR(2)), NOT(VAR(3)))

fun find([], y) = false
| find(x::xs, y) = (x=y) orelse find(xs, y);

fun eval(T, asg) = true
| eval(F, asg) = false
| eval(VAR(x), asg) = find(asg, x)
| eval(NOT(f), asg) = not(eval(f, asg))
| eval(AND(f1, f2), asg) = eval(f1, asg) andalso eval(f2, asg)
| eval(OR(f1, f2), asg) = eval(f1, asg) orelse eval(f2, asg)
| eval(IMPL(f1, f2), asg) = not(eval(f1, asg)) orelse eval(f2, asg);
Predicates

datatype PredFormula = PRED of int * int list 
  | NOT of PredFormula 
  | AND of PredFormula*PredFormula 
  | OR of PredFormula*PredFormula 
  | IMPL of PredFormula*PredFormula 
  | EXISTS of int*PredFormula 
  | FORALL of int*PredFormula

Example:  \( \forall x_1 \ (p_1(x_1) \rightarrow \exists x_2 \ p_1(x_2) \land p_2(x_1, x_2)) \) becomes  
FORALL(1, 
  IMPL(PRED(1, [1]), 
  EXISTS(2, 
    AND(PRED(1, [2]), 
      PRED(2, [1,2]))) 
  )
)
N queens?

\[ 10 \cdot \left( \frac{7}{8} \right)^n \quad \text{dist. in } n+1 \text{ hour} = 10 \cdot \left( \frac{7}{8} \right)^n \]

\[ 10 \left( 1 + \frac{7}{8} + \left( \frac{7}{8} \right)^2 + \left( \frac{7}{8} \right)^3 + \cdots \right) \]

\[ 10 \left( 1 - \frac{1}{\frac{1}{\frac{7}{8}}} \right) = 10 \cdot \frac{1}{\left( 1 - \frac{7}{8} \right)} = 10 \cdot \frac{1}{\frac{1}{8}} = 80 \]
Boolean formula simplification

\[ T(n) = n + \frac{1}{n-1} \sum_{i=1}^{n-1} T(i) \]

Prove: \( T(n) \leq 2n \)

\[ T(n+1) = (n+1) + \frac{1}{n} \sum_{i=1}^{n} T(i) \]

\[ = (n+1) + \frac{1}{n} \left( T(1) + T(2) + \cdots + T(n) \right) \]

\[ \leq (n+1) + \frac{1}{n} \left( \frac{2 \cdot 1 + 2 \cdot 2 + \cdots + 2 \cdot n}{2} \right) \]

\[ = (n+1) + \frac{2}{n} \left( \frac{n \cdot (n+1)}{2} \right) \]

\[ = 2n + 2 \]