Lecture 03: Programming Languages [CMPU-235]
September 7, 2017

Sameer S Pradhan
Vassar College
Functional Programming
Pure Functional Programming

- The result of a function application is independent of the context in which it occurs (referential transparency).
- Variables are bound to values only through the linking of actual parameters to formal parameters in function calls (no side effects).
- Control flow is governed by function calls and conditional expressions (and recursion).
- Storage management is implicit (and requires garbage collection).
- Functions are first class citizens!
“Look, Ma, No Hands!”
- No assignment statements!
- No iteration!

How is it possible to write programs in a language like this?

Some Simple Examples:
- append
- length
- flatten
- atomcount
What is SML?

- Mostly-pure safe strongly-typed functional programming language
- Suitable both for programming in the small and programming in the large
Core-SML: a program is a series of declarations and an expression to be evaluated

- The declarations provide definitions of types, and functions — “the environment”
- The expression is evaluated to get the answer
- Comments are written (* like this *)
Expressions

- Every expression in SML has a type.
  - But never have to write it down. SML will infer the type for you.
  - If you’re get type-errors, sometimes helps to write out types of expressions.
- The most basic expressions are values
- All other expressions are evaluated down to values
  - Some expressions have side-effects (e.g., print), but I won’t talk about those today.
## Values: Basic Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>()</td>
<td>A trivial type with a single value</td>
</tr>
<tr>
<td>bool</td>
<td>true, false</td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>0, 1, 42, ~1</td>
<td>integers</td>
</tr>
<tr>
<td>real</td>
<td>0.0, 1.2, ~2.45</td>
<td>won’t be used much in this course</td>
</tr>
<tr>
<td>char</td>
<td>&quot;a&quot;, #newline</td>
<td>characters</td>
</tr>
<tr>
<td>string</td>
<td>&quot;foo&quot;, &quot;&quot;, &quot;bar baz&quot;</td>
<td>not the same as a list of characters</td>
</tr>
</tbody>
</table>
## Expression: Basic Types

<table>
<thead>
<tr>
<th>Expression</th>
<th>Types</th>
<th>Result type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>if b then e1</code></td>
<td><code>b</code> of type <code>bool</code></td>
<td>same as <code>e1</code></td>
<td></td>
</tr>
<tr>
<td><code>else e2</code></td>
<td><code>e1, e2</code> of the same type</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>e1 andalso e2</code></td>
<td><code>e1, e2 : bool</code></td>
<td><code>bool</code></td>
<td>“&amp;&amp;”, “</td>
</tr>
<tr>
<td><code>e1 orelse e2</code></td>
<td><code>e1, e2 : bool</code></td>
<td><code>bool</code></td>
<td></td>
</tr>
<tr>
<td><code>e1 + e2, e1 - e2, e1 * e2</code></td>
<td><code>e1, e2</code> either both <code>int</code> or both <code>real</code></td>
<td>same as <code>e1</code></td>
<td>overloaded operators</td>
</tr>
<tr>
<td><code>e1 &lt; e2</code></td>
<td><code>e1, e2</code> either both <code>int</code> or both <code>real</code></td>
<td><code>bool</code></td>
<td>also overloaded</td>
</tr>
<tr>
<td><code>e1 = e2</code></td>
<td><code>e1, e2</code> : same equality type</td>
<td><code>bool</code></td>
<td></td>
</tr>
<tr>
<td><code>e1/e2</code></td>
<td><code>e1, e2</code> : <code>real</code></td>
<td><code>real</code></td>
<td>real division</td>
</tr>
<tr>
<td><code>e1 div e2</code></td>
<td><code>e1, e2</code> : <code>int</code></td>
<td><code>int</code></td>
<td>integer division</td>
</tr>
<tr>
<td><code>e1 ^ e2</code></td>
<td><code>e1, e2</code> : <code>string</code></td>
<td><code>string</code></td>
<td>concatenation</td>
</tr>
</tbody>
</table>

For now, “equality types” are just `unit, int, char, string, but not real`.

⚠️ A wart in SML. As we’ll see, you can avoid = most of the time
### More Types: Tuples, Lists, Option

<table>
<thead>
<tr>
<th>Type</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>int * bool</td>
<td>(3,true)</td>
</tr>
<tr>
<td>int * string * string</td>
<td>(42, &quot;foo&quot;, &quot;J.Random Hacker&quot;)</td>
</tr>
<tr>
<td>int list</td>
<td>nil, 2::3::nil</td>
</tr>
<tr>
<td>(int * bool) list</td>
<td>[], [(3,true), (~ 2,false)]</td>
</tr>
<tr>
<td>(string * bool) option</td>
<td>NONE, SOME (&quot;foo&quot;, 23)</td>
</tr>
<tr>
<td>int option list</td>
<td>[NONE, NONE, SOME 1, SOME 3, NONE]</td>
</tr>
</tbody>
</table>

- Tuples are of fixed but arbitrary arity
- Lists are homogeneous
- List and option are polymorphic: more on that later
Defining your own Type

Two mechanisms:

- **Type abbreviations:**
  - type age = int
  - type person = string * age
    - ("J. Random Hacker", 12)

- **New datatypes:**
  - datatype employee =
    Grunt of person
    | Manager of person * employee list
    - Grunt ("J. Random Hacker", 12)
    - Manager ("PHB", 51),
      [Grunt ("J.Random Hacker", 12)]

- The datatype declares several constructor names that must be unique to the datatype
- May be recursive (e.g., employee above)
Datatypes: Two built in ones

Turns out that **list** and option are standard datatypes

- **datatype 'a option = NONE | SOME of 'a**
- **'a is a type variable**: stands for any type
  - SOME 42 : int option
  - SOME ("J. Random Hacker, 12") : person option
  - NONE : 'a option

- **datatype 'a list = nil | :: of 'a * 'a list**

  plus a fixity declaration to make :: be right-associative
Analyzing values: Patterns

In SML, analysis of values is done using **pattern matching**:

Informally speaking, a pattern describes the structure of a value, and binds some variables to the subcomponents of the value.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>_</code></td>
<td>anything</td>
<td>wildcard</td>
</tr>
<tr>
<td><code>x</code></td>
<td>anything</td>
<td>binds x to the value</td>
</tr>
<tr>
<td><code>42</code></td>
<td>the integer 42</td>
<td></td>
</tr>
<tr>
<td><code>false</code></td>
<td>the boolean <code>false</code></td>
<td></td>
</tr>
<tr>
<td><code>(pat_1, pat_2)</code></td>
<td>a pair (v_1, v_2) if (pat_i) matches (v_i)</td>
<td></td>
</tr>
<tr>
<td><code>(x, _)</code></td>
<td>matches ((\text{false}, 42), \text{binds} x \text{ to} \text{false})</td>
<td></td>
</tr>
<tr>
<td><code>(pat_1, pat_2, pat_3)</code></td>
<td>a triple (v_1, v_2, v_3) ...</td>
<td></td>
</tr>
<tr>
<td><code>NONE</code></td>
<td>matches <code>NONE</code> of any option type</td>
<td></td>
</tr>
<tr>
<td><code>SOME pat</code></td>
<td>matches <code>SOME v</code> if <code>pat</code> matches <code>v</code></td>
<td></td>
</tr>
<tr>
<td><code>pat_1 :: pat_2</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Patterns may be used (e.g., at the SML prompt) to define some variables:

- `val x = 42`
- `val (x,y) = (42, false)`
- `val Manager (phb,lackeys) = 
  Manager ("PHB", 51), 
  [Grunt ("J.Random Hacker", 12))]
- `val piApprox = 3.14159`
- `val SOME x = NONE`

Compiler comes back with a warning “Non-exhaustive match”, then runs the code anyway and comes back with a runtime error “binding failure” or “match error”
case Expressions

Analyze a value by cases: like a generalized if-expression.

case (42, false) of
  (x, true) => x
| (23, false) => 17
| _ => 0

- Tests each pattern in order, executes the branch that matches.
- Exhaustiveness-checks at compile-time: generates a warning.
In addition to `val` declarations, can also define functions:

```ml
fun employeeName (Manager ((name, _), _)) = name
  | employeeName (Grunt (name, _)) = name
```

The compiler:

- infers the type of the function `employee -> string`
- checks that we covered all the cases for `employees`
Functions may be recursive, and indeed, polymorphic:

```ml
fun length nil = 0
| length (_::l') = 1 + (length l')
```

Since we don’t case about the elements of the list, this function has type `'a list -> int`