What is SML?

- Mostly-pure safe strongly-typed functional programming language
- Suitable both for programming in the small and programming in the large
Programming in the small
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 \textit{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 (\textit{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>integers</td>
</tr>
<tr>
<td>int</td>
<td>0, 1, 42, ~1</td>
<td>won’t be used much in this course</td>
</tr>
<tr>
<td>real</td>
<td>0.0, 1.2, ~2.45</td>
<td>characters</td>
</tr>
<tr>
<td>char</td>
<td>#&quot;a&quot;, #newline</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td>&quot;foo&quot;, &quot;&quot;, &quot;bar baz&quot;</td>
<td><em>not</em> the same as a list of characters</td>
</tr>
</tbody>
</table>
## Expressions: basic types

<table>
<thead>
<tr>
<th>Expression</th>
<th>Types</th>
<th>Result type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( b ) then ( e_1 ) else ( e_2 )</td>
<td>( b ) of type bool, ( e_1, e_2 ) of the same type</td>
<td>same as ( e_i )</td>
<td></td>
</tr>
<tr>
<td>( e_1 ) andalso ( e_2 )</td>
<td>( e_1, e_2 ) : bool</td>
<td>bool</td>
<td>“&amp;&amp;”, “</td>
</tr>
<tr>
<td>( e_1 ) orelse ( e_2 )</td>
<td>( e_1, e_2 ) : bool</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>( e_1 + e_2, e_1 - e_2, e_1 * e_2 )</td>
<td>( e_1, e_2 ) either both int, or both real</td>
<td>same as ( e_i )</td>
<td>overloaded operators</td>
</tr>
<tr>
<td>( e_1 &lt; e_2 )</td>
<td>( e_1, e_2 ) either both int, or both real</td>
<td>bool</td>
<td>also overloaded</td>
</tr>
<tr>
<td>( e_1 = e_2 )</td>
<td>( e_1, e_2 ) : same equality type</td>
<td>bool</td>
<td></td>
</tr>
<tr>
<td>( e_1/e_2 )</td>
<td>( e_1, e_2 ) : real</td>
<td>real</td>
<td>real division</td>
</tr>
<tr>
<td>( e_1 ) div ( e_2 )</td>
<td>( e_1, e_2 ) : int</td>
<td>int</td>
<td>integer division</td>
</tr>
<tr>
<td>( e_1 ^ e_2 )</td>
<td>( e_1, e_2 ) : string</td>
<td>string</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
More types: defining your own

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>_</td>
<td>anything</td>
<td>wildcard</td>
</tr>
<tr>
<td>x</td>
<td>anything</td>
<td>binds x to the value</td>
</tr>
<tr>
<td>42</td>
<td>the integer 42</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td>the boolean <em>false</em></td>
<td></td>
</tr>
<tr>
<td>(pat₁, pat₂)</td>
<td>a pair ((v₁, v₂)) if (patᵢ) matches (vᵢ)</td>
<td></td>
</tr>
<tr>
<td>((x, _)</td>
<td>matches ((false, 42)), binds (x) to (false)</td>
<td></td>
</tr>
<tr>
<td>((pat₁, pat₂, pat₃))</td>
<td>a triple ((v₁, v₂, v₃)) ...</td>
<td></td>
</tr>
<tr>
<td>NONE</td>
<td>matches NONE of any option type</td>
<td></td>
</tr>
<tr>
<td>SOME (pat)</td>
<td>matches SOME (v) if (pat) matches (v)</td>
<td></td>
</tr>
<tr>
<td>(pat₁ ::∶ pat₂)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
val Declarations

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.

```plaintext
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.
fun Declarations

In addition to val declarations, can also define functions:

```
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`
Recursion, polymorphism

Functions may be recursive, and indeed, polymorphic:

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

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

We can have some local declarations within an expression.

fun countSubordinates (Grunt _) = 0
  | countSubordinates (Manager m) = let
    val (_,grunts) = m
    in
    length grunts
  end

- Each declaration in scope from its point of definition until the end.
- Useful for naming intermediate results, and helper functions
Mutual Recursion - Datatypes, functions

Sometimes, useful to have mutually recursive datatypes:

datatype 'a evenlist = Empty | EvenCons of 'a * 'a oddlist
and 'a oddlist = One of 'a | OddCons of 'a * 'a evenlist

Similarly, can have mutually recursive functions:

fun evenlength Empty = 0
| evenlength (EvenCons (_,ol)) = 1 + oddlength ol

and oddlength (One _) = 1
| oddlength (OddCons (_,el)) = 1 + evenlength el
Anonymous and first class functions

In SML, as in other functional languages, functions may return functions, or take functions as arguments:

- `fun addN n = fn x => n + x`

Take an `int`, return anonymous function that adds `n` to `x`

- Has type `int -> int -> int`

- `fun modifyAge f (name, age) = (name, f age)`

Two patterns: match a thing and call it `f`, match a person

- Can be given the type: `(int -> int) -> person -> person`
Example: \texttt{modifyAge\ (addN \ 1)} has type \texttt{person \to \ person}

\begin{verbatim}
fun map f nil = nil
| map f (x::xs) = (f x) :: (map f xs)
\end{verbatim}

This function has type
\texttt{('a \to \ 'b) \to \ 'a \ list \to \ 'b \ list}

\texttt{map\ (modifyAge\ (addN\ 1))\ somePeople}
Exceptions

- Functions must deal with unexpected input. Sometimes there is no sensible result type.

- Sometimes one can modify the function to return an option type, and return \texttt{NONE} on bad input.

- However sometimes need truly exceptional behavior: no sensible way to deal with bad data locally.
Exceptions: declaration

Exceptions are declared by an exception declaration:

```ml
exception NegativeAge of person
```
fun canRentCar (p as (_, age)) = 
if age <= 0 then raise (NegativeAge p)
else age >= 25

Note I snuck in another pattern in here: \( y \text{ as } pat \) matches if \( pat \) matches the entire value, and also binds \( y \) to that value
Handling exceptions is done as follows:

```sml
canRentCar aPerson
  handle (NegativeAge p) => false
  | Div => false (* raised by integer divide by zero *)
```

Handle has some patterns that matches some exceptions. No need to handle all exceptions: unhandled ones propagate up to top level. May reraise an exception:

```sml
foo handle e => raise e
```
Programming in the large
Programming in the large

Full SML: a program is a collection of structures (i.e., modules) and an expression to be evaluated

- Collaboration of multiple programmers
- Factoring of independent components
- Code reuse
Structures

A structure declaration is a namespace for type, variable and function definitions.

structure BinaryTree = struct
  datatype 'a tree = ...
  val empty = ...
  fun addElem (t,x) = ...
  fun map f t = ...
  fun isEmpty t = ...
  fun toList t = ...
end

Outside the structure, refer to BinaryTree.empty, etc.
Signatures as the type of structures

- Just as values have types, structures have a signature.
- Compiler will infer a principal signature for a structure that includes:
  - the definition of every type abbreviation and datatype
  - the declaration of every exception
  - the type of every value and function

```plaintext
sig
  datatype 'a tree = ...
  val empty : 'a tree
  val addElem : 'a * 'a tree -> 'a tree
  val mapTree : ('a -> 'b) -> 'a tree -> 'b tree
  ...
end
```
Abstraction: non-principal signatures

You can also write down less specific signatures:

- Hide the definition of types or datatypes
- Abstract data types
- Hide helper functions

signature BINARY_TREE = sig
  type 'a tree
  val empty : 'a tree
  val addElem : 'a * 'a tree -> 'a tree
  val map : ('a -> 'b) -> 'a tree -> 'b tree
end
signature Ascription: hiding the implementation

structure BinaryTree :> BINARY_TREE = struct
    datatype 'a tree = ...
    val empty = ...
    fun addElem (t,x) = ...
    fun map f t = ...
    fun isEmpty t = ...
    fun toList t = ...
end

Outside the implementation, the representation of trees and several functions are inaccessible.
Functors: parametrized implementation

To facilitate code reuse, possible to parametrize the implementation of a structure by zero or more other structures:

```sml
functor BalancedBinaryTreeFn (structure B : BINARY_TREE) =
struct
  fun balance t = ... (* mentions B.addElem, etc *)
end
```

Then use as:

```sml
structure MyBBT = BalancedBinaryTreeFn (structure B = BinaryTree)
structure BetterBBT = BalancedBinaryTreeFn (structure B = BetterBinaryTree)
```
Conclusion
Conclusion: what I haven’t told you

- Anything about side-effects: I/O, mutable store, concurrency, etc.
  - Turns out you can accomplish a lot (nearly everything for this course) without them.

- The truth about equality types.
  - But you can mostly pretend they don’t exist.

- Record types: like tuples with named components

- Projection functions for tuples and records
What I haven’t told you, cont’d

- Advanced modular programming: substructures, sharing specs, where-type

- Useful library functions.
  - Read the SML Basis documentation on the SML/NJ webpage [www.smlnj.org](http://www.smlnj.org)

- Compilation management
  - aka CM, documented on the SML/NJ webpage. Will provide instructions in the programming assignments.
Further reading

- Ullman, “Elements of ML Programming”
- Paulson, “ML for the Working Programmer”
- Harper, “Programming in Standard ML”
- SML/NJ webpage
- Our course website