Functional Core SML Quick Reference[1]

1 Basic properties of SML

1. Static typing
2. Strong typing
3. Deep binding
4. Static scoping
5. Pass-by-value parameter passing

2 Interacting with SML

The Standard ML[2] interactive environment is installed at `/tools/cs/smlnj/bin/sml` on CSEL machines. You can interact with it as follows:

`X;`	Evaluate expression `X` and infer its type (note the semicolon)
`use("`*`file.sml`*`")`	Evaluate the contents of *file.sml* as if they had been typed in, and list all newly defined entities by name and type
`it`	This expression contains the result of the previous computation
⟨*Ctrl*⟩ − ⟨*C*⟩	Abort computation/input completion ("`=`" prompt), return to regular ("`-`") prompt
⟨*Ctrl*⟩ − ⟨*D*⟩	Quit SML, only works from the regular ("`-`") prompt

3 Built-in Types

The following is only a selection of the most important types. A full listing is available from the `http://www.smlnj.org` website.

Type name	**Values of this type**	**Description**
`unit`	`()`	The type with only one element
`int`	`~1073741824`...`1073741823`	Integers (`~1` denotes -1)
`bool`	`true; false`	Booleans
`char`	`#"A"; #"1"; #"\001"`	Characters
`string`	`""; "a"; "foo"; "\t---\n"`	Character strings
`'a list`	`nil; []; [1]; [true,false,true]`	Polymorphic lists

4 The type system

SML uses *type judgements* to tell the types of things: If it says $x : T$, then x is of type T. These judgements can be specified by programmers; in that case, they are called *type annotations*.

Judgement	**Description**	**Requirements**
$c : T$	Literal value	iff c is a value of T.
$(x_1, \ldots, x_n) : T_1 * \ldots * T_n$	Tuple construction	iff, for all $i \in \{1 \ldots n\}$, $x_i : T_i$.
$(\texttt{fn}\ x\ \texttt{=>}\ y) : T \to U$	Function construction	iff $x : T$ and $y : U$.
$(fg) : U$	Function application	iff $f : T \to U$ and $g : T$.

e-mail: `reichenb@colorado.edu`

[2]See `http://www.smlnj.org` for the full distribution and manuals.

5 Int

name	type	semantics
~	int → int	Negation
abs	int → int	Absolute value
(div)	int * int → int	Integer division
(mod)	int * int → int	Modulo
(*)	int * int → int	Multiplication
(+)	int * int → int	Addition
(-)	int * int → int	Subtraction
(<)	int * int → bool	Less-than operator
(>)	int * int → bool	Greater-than operator
(<=)	int * int → bool	Less-than-or-equal operator
(>=)	int * int → bool	Greater-than-or-equal operator
(=)	int * int → bool	Equality test

6 Bool

name	type	semantics
not	bool → bool	Negates a boolean value
(=)	bool * bool → bool	Equality test

7 Char

name	type	semantics
ord	char → int	Maps a character to its ASCII value
chr	int → char	Interprets a number as an ASCII character
(=)	char * char → bool	Equality test

8 String

name	type	semantics
explode	string → char list	Turns a string into a list of characters
implode	char list → string	Concatenates characters in a list into a string
size	string → int	Determines the length of a string
(^)	string * string → string	Concatenates two strings
(=)	string * string → bool	Equality test (by value)

9 Lists

name	type	semantics
nil	'a list	Same as [] (the empty list)
hd	'a list → 'a	Returns the head (first element) of a list, raises an exception on the empty list
tl	'a list → 'a	Returns the tail of a list (all elements but the first one), raises exception on the empty list
length	'a list → int	Determines the length of a list
map	('a → 'b) → 'a list → 'b list	Applies a function to all elements in a list
rev	'a list → 'a list	Reverses a list
(@)	('a list * 'a list) → 'a list	Concatenates two lists
(=)	string * string → bool	Equality test (by value)

For 'a and 'b, you can substitute any type.

10 On recursion

For solving a problem by recursion, consider that your function will examine each possible sub-problem at some point. Find all sub-problems with immediate answers, mark these *induction anchor*s. Find all sub-problems whose solutions depend on solutions to their respective sub-problems; mark them *induction step*s. Then determine a way to *distinguish* between all cases.

10.1 Induction anchors

- Answer is immediately known
- No recursion is needed
- Usually very easy to determine

10.2 Induction steps

- Answer can be derived from answers to sub-problems
- Recursion is needed
- Answer requires understanding of the relation of a problem to its immediate sub-problems

10.3 Example: Multiplication for natural numbers

We want to define multiplication for positive integers as `mul: int * int -> int`. We observe:

- Easy case (#1): $\texttt{mul}(x, 0) = 0$
- Complex case (#2): $\texttt{mul}(x, y) = \texttt{mul}(x, y - 1) + x$, if $y > 0$
- Distinction: $\texttt{mul}(x, y) =$ if $y = 0$ then #1, else #2

We must also make sure that we cover all cases, and that any recursion will eventually terminate.

Solution (distinguishing through pattern matching):

```
fun mul (x, 0) = 0                  (* induction anchor *)
  | mul (x, y) = mul(x, y-1) + x;   (* induction step   *)
```

11 User-defined Types

All user-defined types implicitly have an implicit (=) comparison operator.

$$
\begin{array}{rcl}
\langle\text{TypeDecl}\rangle & \longrightarrow & \langle\text{TypeAlias}\rangle \mid \langle\text{Datatype}\rangle \\
\\
\langle\text{TypeAlias}\rangle & \longrightarrow & \texttt{type}\ \langle\text{TypeAliasSeq}\rangle \\
\langle\text{TypeAliasSeq}\rangle & \longrightarrow & \langle\text{TypeAliasDecl}\rangle \mid \langle\text{TypeAliasDecl}\rangle\ \texttt{and}\ \langle\text{TypeAliasSeq}\rangle \\
\langle\text{TypeAliasDecl}\rangle & \longrightarrow & \langle\text{Name}\rangle\ \texttt{=}\ \langle\text{Type}\rangle \\
\\
\langle\text{Datatype}\rangle & \longrightarrow & \texttt{datatype}\ \langle\text{DatatypeSeq}\rangle \\
\langle\text{DatatypeSeq}\rangle & \longrightarrow & \langle\text{DatatypeDecl}\rangle \mid \langle\text{DatatypeDecl}\rangle\ \texttt{and}\ \langle\text{DatatypeSeq}\rangle \\
\langle\text{DatatypeDecl}\rangle & \longrightarrow & \langle\text{Name}\rangle\ \texttt{=}\ \langle\text{DTOptionList}\rangle \\
\langle\text{DTOptionList}\rangle & \longrightarrow & \langle\text{DTOption}\rangle \mid \langle\text{DTOption}\rangle\ \texttt{|}\ \langle\text{DTOptionList}\rangle \\
\langle\text{DTOption}\rangle & \longrightarrow & \langle\text{Name}\rangle \mid \langle\text{Name}\rangle\ \texttt{of}\ \langle\text{Type}\rangle
\end{array}
$$

12 Expression Syntax

<table>
<tr><td>⟨Expr⟩</td><td>⟶</td><td>⟨Literal⟩</td><td>Denotes the literal value ⟨Literal⟩ of one of the built-in types.</td></tr>
<tr><td></td><td></td><td>| `(` ⟨Expr⟩$_1$ `,...,` ⟨Expr⟩$_n$`)`</td><td>Denotes a tuple of n expressions.</td></tr>
<tr><td></td><td></td><td>| ⟨Expr⟩ ⟨Op⟩ ⟨Expr⟩</td><td>Infix operator/constructor application of ⟨Op⟩.</td></tr>
<tr><td></td><td></td><td>| ⟨Expr⟩$_1$ ⟨Expr⟩$_2$</td><td>Function application of ⟨Expr⟩$_1$ to ⟨Expr⟩$_2$.</td></tr>
<tr><td></td><td></td><td>| `let` ⟨DLst⟩ `in` ⟨Expr⟩ `end`</td><td>Evaluates to whatever ⟨Expr⟩ evaluates if all definitions in ⟨DList⟩ (temporarily) hold.</td></tr>
<tr><td></td><td></td><td>| `(`⟨Expr⟩$_1$ `;...;` ⟨Expr⟩$_n$`)`</td><td>Computes all contained expressions in ascending sequence, but evaluates to ⟨Expr⟩$_n$.</td></tr>
<tr><td></td><td></td><td>| `case` ⟨Expr⟩ `of` ⟨Optns⟩</td><td>Matches the value of ⟨Expr⟩ to one of the patterns in ⟨Optns⟩ and selects the corresponding branch.</td></tr>
<tr><td></td><td></td><td>| `fn` ⟨Optns⟩</td><td>Denotes a function which evaluates to an expression matching some pattern within ⟨Optns⟩.</td></tr>
<tr><td>⟨DLst⟩</td><td>⟶</td><td>ε | ⟨Decl⟩ ⟨DLst⟩</td><td></td></tr>
<tr><td>⟨Decl⟩</td><td>⟶</td><td>`val` ⟨Pat⟩ `=` ⟨Expr⟩</td><td>Introduce global name(s), set to the result of the evaluation of ⟨`Expr`⟩.</td></tr>
<tr><td></td><td></td><td>| `fun` ⟨NmOpts⟩</td><td>Syntactic sugar for `val` ⟨Name⟩ `=` `fn` ⟨Opts⟩. Also allows recursion.</td></tr>
<tr><td>⟨NmOpts⟩</td><td>⟶</td><td>⟨NmOpt⟩ | ⟨NmOpt⟩ `|` ⟨NmOpts⟩</td><td>A sequence of options with function names. All function names must be the same.</td></tr>
<tr><td>⟨NmOpt⟩</td><td>⟶</td><td>⟨Name⟩ ⟨Pat⟩ ⟨TAnn⟩ `=` ⟨Expr⟩</td><td>If pattern ⟨Pat⟩ is matched, ⟨Expr⟩ is executed. The type annotation ⟨TAnn⟩ is optional.</td></tr>
<tr><td>⟨Pat⟩</td><td>⟶</td><td>⟨Name⟩ ⟨TAnn⟩</td><td>A pattern can be a simple name.</td></tr>
<tr><td></td><td></td><td>| (⟨Pat⟩, ..., ⟨Pat⟩) ⟨TAnn⟩</td><td>A tuple pattern construction.</td></tr>
<tr><td></td><td></td><td>| ⟨Name⟩ ⟨Pat⟩ ⟨TAnn⟩</td><td>Where ⟨Name⟩ is a constructor.</td></tr>
<tr><td></td><td></td><td>| ⟨Literal⟩ ⟨TAnn⟩</td><td>Literal values can form patterns.</td></tr>
<tr><td></td><td></td><td>| `_` ⟨TAnn⟩</td><td>The wildcard pattern</td></tr>
<tr><td>⟨TAnn⟩</td><td>⟶</td><td>ε | : ⟨Type⟩</td><td>Optional type annotation</td></tr>
<tr><td>⟨Type⟩</td><td>⟶</td><td>⟨Name⟩</td><td>Any of the built-in types (int, string etc.) or any user-defined type.</td></tr>
<tr><td></td><td></td><td>| ⟨Type⟩ `*` ⟨Type⟩</td><td>Tuple construction.</td></tr>
<tr><td></td><td></td><td>| ⟨Type⟩ `->` ⟨Type⟩</td><td>Function construction.</td></tr>
<tr><td>⟨Optns⟩</td><td>⟶</td><td>⟨Optn⟩ | ⟨Optn⟩ `|` ⟨Optns⟩</td><td>One or more options, separated by bars.</td></tr>
<tr><td>⟨Optn⟩</td><td>⟶</td><td>⟨Pat⟩ `=>` ⟨Expr⟩</td><td>Evaluates to ⟨Expr⟩ iff the input matches ⟨Pat⟩ and no previous pattern was matched.</td></tr>
<tr><td>⟨Name⟩</td><td>⟶</td><td>a | b | ...</td><td>Any name, except for the names of operators (such as o).</td></tr>
</table>