F28PL1 Programming Languages

Lecture 15: Standard ML 5

String operations

- `string` is not list of `char`
- `explode: string -> char list`

```
- explode “abc”;
> [#”a”,#”b”,#”c”] : char list
```

- `implode: char list -> string`

```
- implode [#“a”,#”b”,#”c”];
> “abc” : string
```

Digit string to integer

- keep value so far in v
- `[] ==>` return v
- `(h:t) ==>` get value from `t` with `10*v`+value for h

```
- fun toInt1 v [] = v |
      toInt1 v (h::t) =
        toInt1 (10*v+ord h-ord #"0") t;
> val toInt1 = fn : int -> string -> int
- fun toInt s = toInt1 0 (explode s);
> val toInt = fn : string -> int
- toInt “4321”;
> 4321 : int
```

Digit string to integer

```
toInt “4321” ==>
toInt1 0 [#”4”,#”3”,#”2”,#”1”] ==>
toInt1 (10*0+ord #”4”-ord #”0”)
       [#”3”,#”2”,#”1”] ==>
toInt1 4 [#”3”,#”2”,#”1”] ==>
toInt1 (10*4+ord #”3”-ord #”0”)
        [#”2”,#”1”] ==>
toInt1 43 [#”2”,#”1”] ==>
toInt (10*43+ord #”2”-ord #”0”) [#”1”] ==>
toInt 432 [#”1”] ==> ... ==> 4321
```

Integer to digit string

1. produce list of individual digits
2. convert each digit to `char`
3. `implode` list of `chars`

- produce digit list by repeatedly finding the remainder with 10
- stop at 0

```
- fun getDigits1 0 = []  |
      getDigits1 i =
       getDigits1 (i div 10)@[i mod 10];
> val getDigits1 = fn : int -> int list
```

Integer to digit string

- treat `0` as a special case

```
- fun getDigits 0 = [0] |
      getDigits i = getDigits1 i;
> val getDigits = fn : int -> int list
```

- `- getDigits 4321;`
- `> [4,3,2,1] : int list`

Integer to digit string

- getDigits 4321 ==>

```
getDigits1 4321 ==>
getDigits1 432@[1] ==>
getDigits1 43@[2]@[1] ==>
getDigits1 4@[3]@[2]@[1] ==>
getDigits1 0@[4]@[3]@[2]@[1] ==>
[]@[4]@[3]@[2]@[1] ==>
[4,3,2,1]
```

Integer to digit string

```
- fun toChar v = chr (v+ord #"0");
> val toChar = fn : int -> char

- fun fromInt i =
        implode (map toChar (getDigits i));
> val fromInt = fn : int -> string
- fromInt 4321;
> “4321” : string
```

Integer to digit string

```
fromInt 4321 ==>
implode (map toChar (getDigits 4321)) ==>
implode (map toChar [4,3,2,1]) ==>
implode [#”4”,#”3”,#”2”,#”1”] ==>
“4321”
```

Input/Output

- libraries provided via module system
- accessed via *interfaces* specified as *signatures* instantiated by *structures*
  - like polymorphic object classes
- `TextIO` module
- I/O is via `TextIO.vector` type
  - we see these as `string`
- filenames are strings

Input/Output

- input stream type: `TextIO.instream`
- to open a file for input:

```
TextIO.openIn : string -> TextIO.instream
```

- to input all of a stream at once:

```
TextIO.inputAll : TextIO.inputstream ->
string
```

- to input a line from a stream:

```
TextIO.inputLine : TextIO.inputstream ->
string
```

Input/Output

- to input N characters from a stream:

```
TextIO.inputN : TextIO.inputstream * int ->
string
```

- to close an input stream:

```
TextIO.closeIn: TextIO.instream -> unit
```

- standard input stream: `TextIO.stdIn`

Input/Output

- output stream type: `TextIO.outstream`
- to open a file for output:

```
TextIO.openOut : string -> TextIO.outstream
```

- to output to a stream:

```
output : outstream * string -> unit
```

- to close an output stream:

```
TextIO.closeOut: textIO.outstream -> unit
```

- standard output stream: `TextIO.stdOut`

Input/Output

- imperative I/O
- need to sequence actions in time
- use local definition or function composition

Example: echo keyboard to display

```
- fun echo () =
  let val inVal = TextIO.input TextIO.stdIn
  in
    let val outVal =
          TextIO.output(TextIO.stdOut,inVal)
    in echo ()
    end
  end;
> val echo = fn : unit -> unit
```

Example: echo keyboard to display

```
- echo();
hello
```
`hello`
```
there
```
`there`
```
...
```

Example: character frequency count

- e.g. count frequency of characters in file
- hold counts as list of tuple of character & count
- for a new character c:
- `[] ==>` new tuple for c with count 1
- `((c1,i1)::t)` – c=c1 ==>increment `i1`
- `((c1,i1)::t)` – c<>c1 `==>` increment for c in `t` and put `(c1,i1)` back on front

Example: character frequency count

```
- fun freqUpdate c [] = [(c,1)] |
    freqUpdate c ((c1,i1)::t) =
     if c=c1
     then (c1,i1+1)::t
     else (c1,i1)::freqUpdate c t;
> val freqUpdate =
   fn : ‘’a -> (‘’a * int) list ->
        (‘’a * int) list
```

Example: character frequency count

```
- freqUpdate “b” [(“a”,2),(“e”,4),(“b”,3)]
> [(“a”,2),(“e”,4)::(“b”,4)] : (string * int) list


freqUpdate “b” [(“a”,2),(“e”,4),(“b”,3)] ==>
(“a”,2)::freqUpdate “b” [(“e”,4),(“b”,3)] ==>
(“a”,2)::(”e”,4)::freqUpdate “b” [(“b”,3)] ==>
(“a”,2)::(“e”,4)::(“b”,4)::[] ==>
[(“a”,2),(“e”,4)::(“b”,4)]
```

Example: character frequency count

- count frequencies for list where `f` is counts so far
- `[] ==> f`
- `(h::t) ==>` count frequencies for `t` with `f`, and then update for `h`

```
- fun countFreq [] f = f |
      countFreq (h::t) f =
              countFreq t (freqUpdate h f);
> val countFreq =
   fn : ‘’a list -> (‘’a * int) list ->
        (‘’a * int) list
```

Example: character frequency count

```
- countFreq [“a”,”c”,”a”,”c”,”b”] [];
> [(“a”,2),(“c”,2),(“b”,1)] :
   (string * int) list
```

- `countFreq [“a”,”c”,”a”,”c”,”b”] [] ==>`

```
countFreq [“c”,”a”,”c”,”b”] [(“a”,1)] ==>
countFreq
   [“a”,”c”,”b”] [(“a”,1),(“c”,1)] ==>
countFreq [“c”,”b”] [(“a”,2),(“c”,1)] ==>
countFreq [“b”] [(“a”,2),(“c”,2)] ==>
countFreq [] [(“a”,2),(“c”,2),(“b”,1)] ==>
[(“a”,2),(“c”,2),(“b”,1)]
```

Example: character frequency count

```
-  countFile f =
    countFreq
    (explode
     (TextIO.inputAll
      (TextIO.openIn f))) [];
> val countFile =
     fn : string -> (string * int) list
- countFile “l15.sml”;
> val it =
   [(#"e",121),(#"x",11),(#"p",23),(#"l",46),
    (#"o",66),(#"d",27),(#" ",560),(#"\"",98),
    (#"a",58),(#"b",5),(#"c",29),(#";",48),...] :
    (char * int) list
```

User defined types

- *discriminated union*
- `datatype` *`id`* `=` *`id1`* `|` *`id2`* `| ...` *`idN`*`;`
- *`id`* is a new type
- *`id1`* `...` *`idN`* are *constructors*
  - values of type *`id`*

```
- datatype STATE = ON | OFF;
> datatype STATE = ON | OFF;
- ON;
> ON : STATE
```

User defined types

- can pattern match on constructors
- behave like constants

```
- fun switch ON = OFF |
      switch OFF = ON;
> val switch = fn : STATE -> STATE
- switch OFF;
> ON : STATE
```

- for functions over user defined types, give a separate case for each constructor

User defined types

- can associate values with constructors
- `datatype` *`id`* `=` *`idi`* `of` *`typei`*`| ...`
- *`idi`* used like a tag for values *`typei`*
- can pattern match on constructor and values
- nice way to organise programs
  - structure of processing follows structure of data

User defined types

- e.g. mixed mode arithmetic

```
- datatype NUMB = INT of int | REAL of real;
> datatype NUMB = INT of int | REAL of real;
- INT 33;
> INT 33 : NUMB
- REAL 4.56;
> REAL 4.56 : NUMB
```

- `ADD INT INT → INT`
- `ADD INT REAL → REAL`
- `ADD REAL INT → REAL`
- `ADD REAL REAL → REAL`

User defined types

```
- fun ADD (INT i1) (INT i2) =
       INT (i1+i2) |
      ADD (INT i) (REAL r) =
       REAL (real i+r) |
      ADD (REAL r) (INT i) =
       REAL (r+real i) |
      ADD (REAL r1) (REAL r2) =
       REAL (r1+r2);
> ADD : NUMB -> NUMB -> NUMB
- ADD (REAL 3.4) (INT 5);
> REAL 8.4 : NUMB
```

Lists as user defined types

- an integer list is empty or is an integer followed by an integer list

```
- datatype ILIST = INULL |
                   ICONS of int * ILIST;
> ...
- ICONS (1,ICONS (2,ICONS (3,INULL)));
> ICONS (1,ICONS (2,ICONS (3,INULL))) : ILIST
```

Lists as user defined types

- to sum the elements of an integer list
- base case: `INULL` ==> 0
- recursion case: `ICONS (value,next)` ==>

  `value` + sum of `next`

```
- fun iSum INULL = 0 |
      iSum (ICONS (value,next)) =
              value+iSum next;
val iSum = fn : ILIST -> int
- iSum (ICONS (1,ICONS (2,ICONS (3,INULL))));
> 6 : int
```

Trees as user defined types

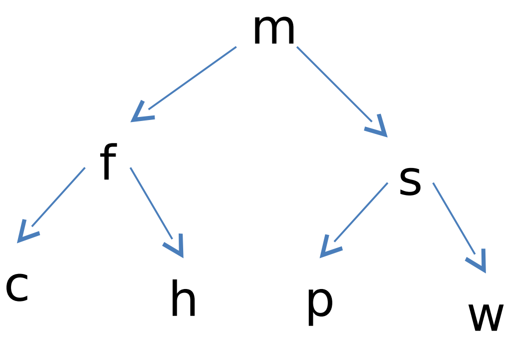
- a string tree is empty or has a value and left and right branches to string trees

```
- datatype STREE =
    SNULL | SNODE of string * STREE * STREE;
- SNODE ("m",
         SNODE ("f",SNODE("c",SNULL,SNULL),
                    SNODE("h",SNULL,SNULL)),
         SNODE ("s",SNODE("p",SNULL,SNULL),
                    SNODE("w",SNULL,SNULL)));
> ... : STREE
```

Trees as user defined types

```
SNODE ("m",
       SNODE ("f",
              SNODE("c",SNULL,SNULL),

              SNODE("h",SNULL,SNULL)),
       SNODE ("s",
              SNODE("p",SNULL,SNULL),
              SNODE("w",SNULL,SNULL)));
```



Trees as user defined types

- to traverse a string tree in ascending order
- base case: `SNULL ==> []`
- recursion case: `SNODE(s,left,right) ==>`

  append traverse of `left` to `s` to traverse of `right`

```
- fun toList SNULL = [] |
      toList (SNODE(s,left,right)) =
        toList left@(s::toList right);
> toList = fn : STREE -> string list
```

Trees as user defined types

```
- toList (SNODE ("m",...));
> [“c”,”f”,”h”,”m”,”p”,”s”,”w”] : string list
```

- `tolist (SNODE (“m”,SNODE(“f”,...),(SNODE(“s”,...))) ==>`

```
toList (SNODE (“f”,...))@[“m”]@tolist (SNODE (“s”,...)) ==>
toList (SNODE (“c”,...))@[“f”]@toList (SNODE (“h”,...))@
  [“m”]@
   toList(SNODE(“p”,...))@[“s”]@toList (SNODE(“w”...)) ==>
... ==>
[“c”]@[“f”]@[“h”]@[“m”]@[“p”]@[“s”]@[“w”] ==>
[“c”,”f”,”h”,”m”,”p”,”s”,”w”]
```

Arithmetic trees

```
- datatype EXP = INT of int |
                 ADD of EXP * EXP |
                 SUB of EXP * EXP |
                 MULT of EXP * EXP |
                 DIV of EXP * EXP ;
> ...
- val t = MULT (ADD (INT 1,INT 2),
                SUB (INT 3,INT 4));
> val t = ... : EXP
```

Arithmetic trees

- to convert a tree to a string
- `INT i ==> i as string`
- *OP*`(left,right) ==>` join converted `left` to string for *OP* to converted `right`
- `(...)` round sub-expressions

```
fun show (INT i) = fromInt i |
    show (ADD(e1,e2)) = "("^show e1^"+"^show e2^")" |
    show (SUB(e1,e2)) = "("^show e1^"-"^show e2^")" |
    show (MULT(e1,e2)) = "("^show e1^"*"^show e2^")" |
    show (DIV(e1,e2)) =  "("^show e1^"/"^show e2^")" ;
> val show = fn : EXP -> string
```

Arithmetic trees

```
- show t;
> “((1+2)*(3-4))” : string
```

- `show (MULT (ADD (...),(SUB(...)) ==>`

```
“(“^show (ADD (INT 1,INT2)^”*”^
 show (SUB (INT 3,INT 4))^”)” ==>
“(“^”(“^show (INT 1)^”+”^show (INT2)^”)”^”*”^
 “(“^show (INT 3)^”-”^show (INT 4)^”)”^”)” ==>
“(“^”(“^”1”^”+”^”2”^”)”^”*
 ”^”(“^”3”^”-”^”4”^”)”^”)” ==>
 “((1+2)*(3-4))”
```

Arithmetic trees

- to evaluate a tree
- `INT i ==> i`
- *OP*`(e1,e2) ==>` value of `e1` *OP*ed with value of `e2`

```
fun eval (INT i) = i |
    eval (ADD(e1,e2)) = eval e1+eval e2 |
    eval (SUB(e1,e2)) = eval e1-eval e2 |
    eval (MULT(e1,e2)) = eval e1*eval e2 |
    eval (DIV(e1,e2)) = eval e1 div eval e2;
> val eval = fn ; EXP -> int
```

Arithmetic trees

- `- eval t;`
- `> -3: int`
- `eval (MULT (ADD(...),SUB(...))) ==>`

```
eval (ADD (INT 1,INT 2))*
 eval (SUB (INT 3,INT 4)) ==>
(eval (INT 1)+eval (INT 2))*
 (eval (INT3)-eval (INT 4)) ==>
(1+2)*(3-4) ==>
-3
```

Polymorphic user defined types

- can parameterise user defined types
- e.g. polymorphic lists

```
- datatype ‘a LIST = NULL |
                     CONS of ‘a * ‘a LIST;
> ...
- CONS (1,CONS (2,CONS (3,NULL)));
> CONS (1,CONS (2,CONS (3,NULL))) : int LIST
-  CONS ((1,”one”),
         CONS((2,”two”),CONS((3,”three”),NULL)));
> CONS ((1,”one”),
         CONS((2,”two”),CONS((3,”three”),NULL))):
  (int * string) LIST
```

Polymorphic user defined types

- length of list

```
- fun Length NULL = 0 |
      Length (CONS (_,t)) = 1+Length t;
> val LENGTH = fn : ‘a LIST -> int
- Length (CONS (1,CONS (2,CONS (3,NULL))));
> 3 : int
```

SML: other features

- records
  - tuples with named fields & selection
- case expression
  - similar to switch but based on pattern matching
- anonymous functions
  - nameless lambda functions
- abstract types
  - user defined type + functions
  - like an OO class

SML: other features

- mutual definitions

```
let ...
and ...
in ...
end
```

- abstract types
  - user defined type + functions
  - like an OO class

SML: modules & libraries

- module system
  - basis of libraries & interfaces
- rich basis library
  - includes I/O, system functions etc
- E. Ganser & J. Reppy, The Standard ML Basis Library, Cambridge, 2004

SML summary: types

- strong, static types
- base types
  - int, real, char, bool
- structured types
  - tuple, list, user defined
- ad-hoc polymorphism
  - operator overloading
- parametric polymorphism
  - functions, lists, user defined types

SML summary: data abstraction

- variable
  - name/value association
  - cannot be changed
- address/memory not visible

SML summary: data abstraction

- variable introduction
  - global definition
  - local definition
  - formal parameter
- scope
  - lexical
- extent
  - local definition, function body,

SML summary: control abstraction

- expressions
  - abstract from arithmetic/logic/flow of control sequences
- conditional expression
- pattern matching
  - abstracts from constant matching
- functions
  - call by value parameter passing
- recursion

SML summary: pragmatics

- higher level than imperative programs
- many to one mapping from expression to machine code
- must be compiled to machine code (or interpreted)
- very succinct
- strong typing ㅍ reduces run-time errors
- good correspondence between program structure & data structure
- automatic memory management
  - garbage collection

SML summary: pragmatics

- not as fast as some imperative languages
  - garbage collection overhead
- CPU independent
  - highly portable
- used for:
  - rapid prototyping
  - reasoning about programs
  - designing parallel frameworks e.g. Google map-reduce