SML - Outline

- Primitive datatypes
- Variables
- Let expressions
- Structured types
- Functions and control expresssions
- Parameter - argument association through pattern matching
  - How to use to define functions on structured types?
- Higher order functions and defining operators
- Exceptions
- Mutually recursive functions

SML

- Standard ML of NJ, Dave MacQueen's group at Bell Laboratories and Andrew Appel's group at Princeton
- Strongly typed, statically checked PL
- Garbage collected implementation
- Strict PL, arguments are evaluated before function call
- Higher order, nested functions

SML

- **Variable bindings are static**
- **Has side effects from imperative constructs**
- **Has formally defined semantics that is complete**
  - **Each legal program has a deterministic result**
  - **All illegal programs are recognizable as such by a compiler**

SML

- **Subset we will use is purely functional**
  - **Referential transparency: function application context does NOT affect returned value**
  - **Functions are first-class citizens**
- **SML interpreter uses typical Lisp *read-eval-print* loop which yields values and their types**

SML by Example

> 2 + 2;

val it = 4 : int

Name of expression — Computed value — Type of value

>fun divide(x,y) = x / y;

*val divide = fn : real *real → real*

Primitive Data Types

TYPE	VALUES	OPERATIONS
bool	true, false	not, andalso, oralso if--then--else
integer	1, 25, ~3	+ - * div, mod, real <>,=,<=,>=
real	3.4, 3E2	+ - * /, floor, <>, =, <=, >=
string	"barbara"	size, ^ (concatenate)
unit	( )	type used for things that have no type (e.g., procs w/o return values)

type converters (real, floor)

Note: + * - are all overloaded operators, but there is NO COERCION in ML. 1+ 3.5 is ILLEGAL!

Also: x+y must be written *x:int + y* or *x:real + y* to distinguish the selected + operator.

Bound Variables

- Using *val*
  - *Val* is not assignment
  - *Val* binds a new instance of a name to a value

    >val x = 17; *binds value 17 to x.*

    >val y = x; *binds value 17 to y.*

    >val x = true; *creates a new x, hiding the previous x, and binds true to the new x.*

    >val z = x; *binds true to z.*

Bound Variables

- Using *let*

  let <decls> in <expr> end;

  Sets up some declarations whose scope is the expression <expr>; body of let expression is evaluated with respect to the environment in which it is written, augmented by the declarations

  Value associated with the *let,* is value of the <expr>

Examples

```
EG1: > let  val x = 2 in
            let  val y = x + 1 in   nested lets
                    y* y
            end;
          end;
```

val it = 9:int

```
EG2: > let    val x = 5
              val y = x + 3
              val x = 3  in     here x is 3, y is 8
                 2 * x * y end;
```

val it = 48 : int

Examples

```
EG3:  let val x = 2 in                      x is 2
              let val y = x + 3 in          y is 5
                      let val x = 4 in      x is 4
                           2 * x * y        2 * 4 * 5
                      end
              end
        end;
```

val it = 40 : int

Structured Types

- *Tuples* - finite sequence of (possibly) differently typed elements

  (3, true, 5.2) : int * bool * real

  – Equality check is done *component-wise*

  (true, 7) = ("abc", ());

  type error: bool * int can't match string * unit

- *Lists* - sequence of same-type elements
  – Equality check is done *component-wise*
  – Cons is shown as ::

  :: is of type 'a * 'a-list → 'a -list

  e::r means e is type τ and r is type τ-list

Structured Types

– @ denotes list append operator

> [2] @ [3, 4] ;

val it = [2, 3,4] : int list

> 2 :: [3, 4, 5];

val it = [2, 3, 4, 5] : int list

– *nil* - a polymorphic object that can inhabit a number of structurally related types; used to show end of list

- *nil* is of type 'a-list
- *nil* also written [ ]

SML type variable

Control Structures as Expressions

- **conditionals *if..then..else, case***

  if x = 1 then y else 2*y;

  case <expr> of [ ] => … | [2::s]=> … | _ => …
- **lets - create static scopes**
- **function application**
- **exceptions - provide different type of function return**

  exception negArg;

  fun areacircle r = if r<0 then raise negArg else (3.1416*r*r);

Functions

- **Function application is the main control structure**
  - **(e1 e2) is function application**
    - **e1 evaluates to a function, usually curried**
    - **e2 is function argument**
  - **e1: $\sigma \rightarrow \tau$, e2: σ**
  - ***call by value* parameter passing (because SML is a strict PL)**

Examples

fun <func_name> <parameter> = <func_body>

>fun areacircle r = 3.14159 * r * r;

val areacircle= fn : real → real

>areacircle 1.0;

val it = 3.14159 : real

>fun areasquare r = r * r; %no good because SML can't type overloaded * operator

>fun areasquare r = r:int * r;

val areasquare = fn: int → int

Examples

>fun areatriangle(b,h) = b * h / 2 %no good because use real division with integer argument

>fun areatriangle(b,h) = b * h / 2.0

*val areatriangle = fn: real * real → real*

>fun curriedareatri b h = b * h / 2.0

val curriedareatri = fn : real → (real → real)

>curriedareatri (4.0);

val it = fn: real → real **%will be area fcn for triangles with base 4.0**

Nested Functions

```
fun reverse (y) =
   let fun rev  nil  z = z | rev (hd::tl)  z = rev  tl  hd::z
   in   rev  y  nil
   end;
```

What is the type of reverse?

(you should be able to show type is: ‘a list → ‘a list)

Anonymous Functions

- **Function values do not necessarily have names associated with them!**

  > val f = (fn n => n+1); — **this form is more like a lambda expression**

  val f = fn: int → int

  > fun g(n) = n + 1;

  val g = fn: int → int

  >val areacircle = fn r =>3.14159 * r * r

  val areacircle = fn: real→ real

Patterns

Pattern **- an expression built from variables and constants by value constructors;**

> val x = (false, 17);

val x = (false, 17) : bool * int

> val s = [“lo”, “high”, “mid”]

val s = [“lo”, “high”, “mid”] : string list

> val hd ::tl = s;

val hd = “lo” :string

val tl = [“high”, “mid”] : string list

Patterns

- **Patterns can have nested elements**

  > val x = ((“foo”, “bar”), true);

  val x = ((“foo” , “bar”), true) : (string * string) * bool

- **SML compiler may complain about your patterns**
  - ***Redundant*** **- means pattern will never be used because all previous patterns match all alternatives (in function definition)**
  - ***Not exhaustive*** **- means there is an uncovered kind of argument that matches none of your patterns.**

List Patterns

pattern	matches	binding	does not match
nil	nil	none	2::nil
x::nil	5::nil	x=5	nil; 2::1::nil
x::y	3::2::1::nil	x=3; y=2::1::nil	nil
[ ]	[ ]	none	[2]
[x]	[5]	x=5	[ ]; [2,1]
[ ]::x	[ ]::[[1]]	x = [[1]]	[1]
(x, _ , y)	(1, 2, 3)	x=1 and y=3	matches all 3 elem tuples
(x, y, z::nil)	(1,2,3::nil)	x=1,y=2, z=3	(1,2, [3,2])

Patterns in Function Abstractions

- Functions on constructed types are often defined using pattern matching to select the function body relevant for specific values of the arguments
- Function abstraction form is:

  fun <id> <pattern1> = <expr1> |
  <id> <pattern2> = <expr2> | …|
  <id> <patternK> = <exprk>

Examples

```
fun length nil = 0  |  length hd::tl = 1 + length (tl);

fun append (nil, r) = r  |  append (hd::tl, r) =
    hd::append(tl, r);

fun power2  0  f  x = x |  power2  n  f  x =
    power2 (n-1) f  (f x)
```

Note: order of alternatives in function body matters since SML picks first one that “matches”

Higher Order Functions

- Using functions as arguments to other functions

```
>fun  map  f  nil = nil  |
   map  f  (x::xs) =  f(x) :: (map  f  xs);
val  map  fn:(‘a → ‘b) → ‘a list → ‘b list
>fun add  x  y:int  = x + y ;
val add = fn :int → int → int
>val succ = (add 1);
val succ = fn : int → int
>map succ  [1];
val it = [2] : int list
>map (fn n => 2*n) [1, 2, 3];  (*use of an anonymous fcn*)
val it = [2,4,6] : int list
```

Higher Order Functions

- **Returning functions as values**

  e.g., succ is returned value from (add 1) which is a function.

  >val pred = add(~1);

  val pred = f: int → int

  >comp f g x = f (g (x));

  val comp = fn: (‘a → ‘b) → (‘c → ‘a) → ‘c → ‘b

  >comp succ pred 3;

  val it = 3: int

  >comp succ pred;

  *val it = fn: int → int (*is the identity function on integers*)*


```
(* this is an sml-based, use of higher order functions parser  which uses exceptions to
   signal syntax error and for control flow between alternative parses *)
exception Fail and Failtoken1 and Failtoken2 and Failvar and Failnum and
   Failpgm;
(* composes 2 functions A and B *)
infix 3 &; fun op&(A,B) = B o A;
(* simulates an OR in a BNF rule*)
infix 2 //; fun f//g = fn s=>(f(s) handle Fail  => g(s) );
(* expr ::= <digit> == <digit>  *)
val expr = let val f= (token "==") in num & f & num  end;
(* note special syntax for mutually recursive functions *)
(* stmts are <var>:=<digit> OR if <expr> then <var>:=<digit> *)
fun stmt0 s = (let val f = (token ":=") in (var & f & num) s end)
and stmt1 s = (let val g=(token "if") val h=(token "else")
   val w=(token "then")
   in      (g & expr & w & stmt & h & stmt) s end)
and stmt s = (stmt0 // stmt1) s;
fun pgm [] = raise Fail
 | pgm x = if ((stmt x)=[]) then (print "successful parse"; print " Hooray!!")
   else print "failed parse, extra input";
```

parserToshow.sml

Exceptions

- **Defined as a unique name, optional parameters**
- **Raise with the keyword *raise***
- **If raised within calculation of an expression, can define an associated handler**

<expr> *handle* <match>

if <expr> evals w/o exception occurring, then value is returned;

if <expr> raises an exception, then try to match raised exception to a listed handler; if not possible, exception escapes to enclosing handler or percolates up the stack of exprs under evaluation until a handler is found

(f x) handle OutOfRange(0,0)= … | OutOfRange(n,m)=… etc.

in parseToshow.sml name of exception was used for debugging; as program failed in different functions, uncaught exception told which function had failed

Defining Operators

compose.sml

Want to define composition as an infix operator with 2 operands, rather than a function, for ease of use

>infix 3 &; fun op&(A,B) = B o A;

Then what are the types of these functions?

>fun g x = (succ & pred) x; ***means (pred (succ x))***

>val h = succ & pred; ***why is val correct here?***

What's going on here?

```
(* composes 2 functions A and B *)
infix 3 &; fun op&(A,B) = B o A;
(* simulates an OR in a BNF rule*)
infix 2 //; fun f//g = fn s=>(f(s) handle Fail  => g(s) );
```

Using Higher Order Fcns: &

```
(* expr ::= <digit> == <digit>  *)
val expr = let val f= (token "==") in num & f & num  end;
```

Here, we are directly coding the BNF rule as a functional composition of parsers for each of the non-terminals and terminal symbols

num recognizes digits, f calls lexer to recognize an equality comparison operator

After looking at lexer, we can see that the type of expr is:

string-list --> string-list


```
(*_____________the lexer__________________*)
(* recognizes token t *)
fun token t [] = raise Fail (*Failtoken1*)
 | token t (s::rest) = if t=s then rest else raise Fail (*Failtoken2*);

fun varId s = "a"<=s andalso s<="z";
(* recognizes a variable name *)
fun var  (s::rest)= if (varId s)then rest else (print(s);raise Fail(*Failvar*))
 | var [] = (print "empty var"; raise Fail(*Failvar*));

(* build the comparisons *)
fun num (w::s) = if (w>="0" andalso w<="9") then s else raise Fail(*Failnum*)
 | num [] = raise Fail;(*Failnum*)

(*_______________end of simple lexer________*)
```

Mutually Recursive Functions

Need to define mutually recursive functions where need to use one function's name in defining the other and vice versa. How can we do this?

e.g., definition of statements needs *stmt* in the body of the *if-stmt(stmt1)* and needs *stmt1* in the body of *stmt*

```
(* stmts are <var>:=<digit> OR if <expr> then <var>:=<digit> *)
fun stmt0 s = (let val f = (token ":=") in (var & f & num) s end)
and stmt1 s = (let val g=(token "if") val h=(token "else")
    val w=(token "then")
    in     (g & expr & w & stmt & h & stmt) s end)
and stmt s = (stmt0 // stmt1) s;
```


Programs in Example

Wrapping this up, a program is simply a statement. if recognition succeeds and uses up all the input, then it is successful; otherwise, it fails.

Print <*string*> is a simple output statement in SML, but it sometimes 'messes up' the standard output you expect

Compound expressions can be formed from sequences of expressions separated by semicolons

```
fun pgm [] = raise Fail
 | pgm x =if ((stmt x)=[]) then (print "successful parse"; print " Hooray!!") else
    print "failed parse, extra input";
-pgm ["x",":=","1"];
successful parse, Hooray!!val it = () : unit;
```