Today, we will be discussing:

- The importance of abstraction
- Using modules for abstraction, and for dividing up large programs
- The SML module system: signatures and structures

1 Abstraction

So far, we’ve seen how to use SML to write small programs consisting of a few functions. Of course, programming languages that are to be used at scale need to support development by a team of programmers working on different parts of the code. SML provides a module system for this purpose. The basic idea is:

- Divide the program up into modules or structures which perform logically separate tasks.
- Give each module an interface or signature that describes what it does and how the rest of the program should interact with it.

The key idea underlying this system and how we will use it is abstraction. Abstraction is a very broad and important idea in computer science, but a large part of the essence of it is that you shouldn’t have to know how something is implemented in order to use it.

Abstraction has several concrete benefits:

1. Separate development: different programmers can implement different modules, and easily put them together if they all satisfy the interface.

2. Separate compilation: you can recompile a module without needing to recompile the whole program.

3. The implementation of a module can be changed (e.g. made faster) without the users (clients) of the module even needing to know.
2 Signatures

We define an interface for a module to meet using a new type of declaration:

signature QUEUE =
  sig
    type 'a q
    exception Empty
    val empty : 'a q
    val isEmpty : 'a q -> bool
    val enq : 'a q -> 'a -> 'a q
    val deq : 'a q -> 'a * 'a q
  end

This signature defines the interface for a queue datatype. This signature says that any module implementing the signature QUEUE must supply:

• A type 'a q
• An exception Empty
• A value empty : 'a q
• Functions isEmpty, enq and deq with the specified types.

Defining the signature doesn’t actually give us any of these things. It just specifies what an appropriate module must define. Note that non-function and function values are treated identically in signatures. It is not even valid syntax to include fun declarations in signatures. A signature is just a specification, it does not implement any functions.

3 Structures

3.1 A first queue implementation

A signature is implemented by a structure. For example, we can implement queues as lists:

structure ListQueue : QUEUE =
  struct
    type 'a q = 'a list
    exception Empty
  end
val empty : 'a q = []

fun isEmpty ([] : 'a q) = true
    | isEmpty _ = false

fun enq (l : 'a q) (x : 'a) = l @ [x]
fun deq ([] : 'a q) = raise Empty
    | deq (x::xs) = (x, xs)
end

3.2 Transparent vs. opaque ascription

The top line, structure ListQueue : QUEUE, says we are implementing a structure called ListQueue that implements the signature QUEUE. We may read the : as “ascribing” the signature to the structure. This is a particular kind of ascription called transparent ascription. What makes it transparent is what happens when we try to use the operations provided by ListQueue in the REPL:

- val onetwo = ListQueue.enq (ListQueue.enq ListQueue.empty 1) 2;
  val onetwo = [1,2] : int ListQueue.q

(Note that, to use the operations provided by a structure S, we use the syntax S.f.) When we make a queue, SML will happily tell us that it is a list (though at least it annotates the type as int ListQueue.q. It will also happily let us make our own queues:

- val one = ListQueue.deq [1,2,3,4,5];
  val one = (1,[2,3,4,5]) : int * int ListQueue.q

This seems to violate the ideas of abstraction, since we shouldn’t need or be able to know how the queue is implemented. On the other hand, it’s occasionally useful for debugging.

SML also provides opaque ascription, which prohibits the above. Opaque ascription is indicated with the signature :>.

structure ListQueue :> QUEUE =
struct

  type 'a q = 'a list

  exception Empty

  val empty : 'a q = []

  fun isEmpty ([] : 'a q) = true
      | isEmpty _ = false

  fun enq (l : 'a q) (x : 'a) = l @ [x]
  fun deq ([] : 'a q) = raise Empty

end
\begin{verbatim}
| deq (x::xs) = (x, xs)

end

- val onetwo = ListQueue.enq (ListQueue.enq ListQueue.empty 1) 2;
val onetwo = - : int ListQueue.q

val one = ListQueue.deq [1,2,3,4,5];

stdIn:7.5-7.36 Error: operator and operand don't agree [tycon mismatch]
  operator domain: 'Z ListQueue.q
  operand: [int ty] list
in expression:
  ListQueue.deq (1 :: 2 :: 3 :: <exp> :: <exp>)

Now, even though the type int ListQueue.q is still implemented as int list, SML gives us no way of finding that out.

3.3 A more efficient queue implementation

We can make enqueue more efficient by implementing queues as a pair of lists \((\text{front}, \text{back})\). In this representation, the queue is considered to be \text{front} @ (\text{rev back}). We enqueue values by consing onto the back, and still dequeue by simply pulling off the front. If the front is empty, we may need to take some items from the back (which we reverse all at once).

structure TwoListQueue :> QUEUE =
  struct

  type 'a q = 'a list * 'a list

  exception Empty

  val empty : 'a q = ([], [])

  fun isEmpty (([], []): 'a q) = true
    | isEmpty _ = false

  fun enq ((front, back): 'a q) (x: 'a) = (front, x::back)
  fun deq (([], []): 'a q) = raise Empty
    | deq ([], back) =
      let val x::front = List.rev back
      in
        (x, (front, []))
      end
    | deq (x::xs, back) = (x, (xs, back))

end

4
4 Abstract datatypes

Why would we want opaque ascription? Isn’t it always better to know how things are being represented? No! Because then people can mess with your internal representation and break your invariants.

Remember our representation of polynomials as int lists in normal form:

signature NF =
sig
  type nf
  val xnf : nf
  val constnf : int -> nf
  val plusnf : nf * nf -> nf
  val eqnf : nf * nf -> bool
end

structure IntListNF : NF =
struct

(* represent \(c_0 x^0 + c_1 x + c_2 x^2 + \ldots\)
by the list \([c_0, c_1, c_2, \ldots]\), with no trailing zeroes.
empty list is [] *)
type nf = int list
val xnf : nf = [0,1]
fun constnf (c : int) : nf = if c = 0 then [] else [c]

(* strip : nf -> nf
* REQUIRES: true
* ENSURES strip (l @ [0, \ldots, 0]) ==> l
*)
fun strip ([]: nf) : nf = []
| strip (c::cs) : nf =
  case (c, strip cs) of
    (0, []) => []
    | (_, cs') => c::cs'

(* plusnf : nf * nf -> nf
* REQUIRES: n1, n2 have no trailing zeroes
* ENSURES: plusnf (n1, n2) is the nf representation of n1 + n2, with no trailing zeroes
*)
fun plusnf (n1 : nf , n2 : nf) : nf =
  let fun plus_int (n1, n2) =
    case (n1, n2) of
in
  strip (plus_int (n1, n2))
end

(* eqnf : nf * nf -> bool
  * REQUIRES: n1, n2 have no trailing zeroes
  * ENSURES: eqnf (n1, n2) ==> true if n1 and n2 represent the same polynomial,
    false otherwise
*)

fun eqnf ([]: nf, []: nf) : bool = true
  | eqnf ([], _) = false
  | eqnf (_, []) = false
  | eqnf (c1::cs1, c2::cs2) = (c1 = c2) andalso (eqnf (cs1, cs2))

end

Our specs for plusnf and eqnf require that the normal forms have no trailing zeroes, thus
making them unique. The implementation of plusnf strips off the trailing zeroes before returning,
and xnf and constnf return lists with no trailing zeroes, so nfs returned by our functions will
always meet the spec. For example,
  - structure N = IntListNF;
  structure N : NF
    - N.plusnf (N.constnf ~5, N.constnf 5);
    val it = [] : IntListNF.nf
    - N.eqnf (N.plusnf (N.constnf ~5, N.constnf 5), N.constnf 0);
    val it = true : bool

  (The first line rebinds the structure IntListNF to the name N, so we can use the shorter name; this
  is helpful if you’ll be typing the same structure name a lot.) But clients can still see that nf is just
  a list and make their own:
  - N.eqnf (N.plusnf (N.constnf ~5, N.constnf 5), [0]);
    val it = false : bool

  But we can change the code to use opaque ascription:

structure IntListNF :> NF =

and then the code above fails to type check.
  - N.eqnf (N.plusnf (N.constnf ~5, N.constnf 5), [0]);
  stdIn:24.1-24.51 Error: operator and operand don’t agree [tycon mismatch]
    operator domain: IntListNF.nf * IntListNF.nf
    operand: IntListNF.nf * [int ty] list
    in expression:
      N.eqnf (N.plusnf (N.constnf <exp>, N.constnf <exp>), 0 :: nil)
With opaque ascription, the only way someone can get something of type \( nf \) is by using our functions, and so if all of our functions enforce the spec, then our functions can also assume the spec is met.

## 5 Dictionaries

Let’s see another example: dictionaries whose keys are strings and whose values are \( 'a \).

```plaintext
signature DICT =
  sig
    type key = string (* concrete *)
    type 'a entry = key * 'a (* concrete *)
    type 'a dict (* abstract *)

    val empty : 'a dict
    val lookup : 'a dict -> key -> 'a option
    val insert : 'a dict * 'a entry -> 'a dict
  end
```

Note that, unlike with functions, we can provide full type definitions inside signatures. These types are called concrete, which is the opposite of abstract.

We’ve already seen one implementation of dictionaries as functions.

```plaintext
structure FunDict : DICT =
  struct

    type key = string
    type 'a entry = key * 'a
    type 'a dict = string -> 'a option

    val empty : 'a dict = fn _ => NONE
    fun lookup (f : 'a dict) (k : string) = f k
    fun insert (d : 'a dict, (k, v) : 'a entry) : 'a dict =
      fn k' => if k = k' then SOME v
              else lookup d k
  end
```

Now here’s another one in terms of binary search trees:

```plaintext
structure BST : DICT =
  struct
```

---

\(^1\)For three years of college, I lived in a dorm built in the Brutalist architectural style, which uses lots of exposed unfinished concrete; think Wean Hall. The dorm used to put out a literary magazine called *The Concrete Abstract.*
type key = string

type 'a entry = key * 'a

datatype 'a tree =
  | Empty
  | Node of 'a tree * 'a entry * 'a tree

type 'a dict = 'a tree

val empty = Empty

fun lookup (Empty : 'a tree) (k : string) = NONE
  | lookup (Node (l, (k', v), r)) k =
    (case String.compare (k, k') of
      EQUAL => SOME v
    | LESS => lookup l k
    | GREATER => lookup r k)

fun insert (Empty, e) = Node(Empty, e, Empty)
  | insert (Node(l, (k', v'), r), (k, v)) =
    (case String.compare (k, k') of
      EQUAL => Node (l, (k, v), r)
    | LESS => Node (insert (l, (k, v)), (k', v'), r)
    | GREATER => Node (l, (k', v'), insert (r, (k, v))))

end

These dictionaries aren’t as polymorphic as we would like. We would like to be able to make
the key type arbitrary too, but then in order to implement dictionaries as BSTs, we’d need a
comparison function on keys. Next lecture, we’ll see a way to write a general module that can be
specialized to work with a particular comparison function, so we can have dictionaries with string
keys, int keys, list keys, and so on.