Stacks

(Version of March 15, 2010)
(Based on original slides by Yves Deville)

Stacks of objects of variable type α (denoted 'a): α stack

Operations

value emptyStack
TYPE: α stack
VALUE: the empty stack

function isEmptyStack S
TYPE: α stack $\rightarrow$ bool
PRE: (none)
POST: **true** if S is empty
false otherwise

function push v S
TYPE: $\alpha \rightarrow \alpha$ stack $\rightarrow \alpha$ stack
PRE: (none)
POST: the stack S with v added as new top element

function top S
TYPE: α stack $\rightarrow \alpha$
PRE: S is non-empty
POST: the top element of S

function pop S
TYPE: α stack $\rightarrow \alpha$ stack
PRE: S is non-empty
POST: the stack S without its top element

Realisation 1

Representation of a stack by a *list*:

type α stack = α **list**
REPRESENTATION CONVENTION: the head of the list is the top of the stack, the 2nd element of the list is the element below the top, etc

where **stack** is a *type constructor*

Operations

```
val emptyStack = [ ]

fun isEmptyStack S = (S = [ ])

fun push v S = v::S

fun top [ ] = error "top: empty stack"
  | top (x::xs) = x

fun pop [ ] = error "pop: empty stack"
  | pop (x::xs) = xs
```

Note that the **error** function is *not* a built-in of SML.

- This realisation does *not* force the usage of the type α **stack**
- The operations can *also* be used with objects of type α **list**, even if they do not represent stacks!
- It is possible to access the elements of the stack *without* using the operations specified above: no encapsulation!

Realisation 2

Definition of a *new* constructed type using the α **list** type:

datatype α stack = Stack **of** α **list**
REPRESENTATION CONVENTION: the head of the list is the top of the stack, the 2nd element of the list is the element below the top, etc

where Stack is a *value constructor*

Operations

```
val emptyStack = Stack [ ]

fun isEmptyStack (Stack S) = (S = [ ])

fun push v (Stack S) = Stack (v::S)

fun top (Stack [ ]) = error "top: empty stack"
  | top (Stack (x::xs)) = x

fun pop (Stack [ ]) = error "pop: empty stack"
  | pop (Stack (x::xs)) = Stack xs
```

- The operations are now *only* defined for stacks
- It is *still* possible to access the elements of the stack *without* using the operations specified above, namely by pattern matching

An Abstract Datatype

Objective: encapsulate the definition of the α **stack** type and its operations in a parametrised *abstract datatype*

```
abstype 'a stack = Stack of 'a list
with
    val emptyStack = Stack [ ]
    fun isEmptyStack (Stack S) = (S = [ ])
    fun push v (Stack S) = Stack (v::S)
    fun top (Stack [ ]) = error "top: empty stack"
      | top (Stack (x::xs)) = x
    fun pop (Stack [ ]) = error "pop: empty stack"
      | pop (Stack (x::xs)) = Stack xs
end
```

- The α **stack** type is an *abstract datatype* (ADT)
- The concrete representation of a stack is *hidden*
- An object of the α **stack** type can *only* be manipulated via the functions defined in its ADT declaration
- The **Stack** value constructor is *invisible* outside the ADT
- It is now *impossible* to access the representation of a stack outside the declarations of the functions of the ADT
- The parametrisation (by α) allows the usage of stacks of integers, reals, strings, integer functions, etc, from a *single* definition!

```
- abstype 'a stack = Stack of 'a list with ... ;
  type 'a stack
  val 'a emptyStack = - : 'a stack
  val ''a isEmptyStack = fn : ''a stack -> bool
  ...
- push 1 (Stack [ ]) ;
  Error: unbound variable or constructor: Stack
- push 1 emptyStack ;
  val it = - : int stack
```

It is *impossible* to compare two stacks:

```
- emptyStack = emptyStack ;
  Error: operator and operand don't agree
  [equality type required]
```

It is *impossible* to see the contents of a stack without popping its elements, so let us add a visualisation function:

function showStack S
TYPE: α stack $\rightarrow$ α list
PRE: (none)
POST: the representation of S in list form, with the top of S as head, etc

```
abstype 'a stack = Stack of 'a list
with
   ...
   fun showStack (Stack S) = S
end
```

- The result of showStack is *not* of the α stack type
- One can thus *not* apply the stack operations to it

Realisation 3

Definition of a *recursive* new constructed type:

datatype α stack = EmptyStack
| >> **of** α stack $*$ α

infix >>

EXAMPLE: EmptyStack >> 3 >> 5 >> 2 represents the stack with top 2

REPRESENTATION CONVENTION: the right-most value is the top of the stack, its left neighbour is the element below the top, etc

An Abstract Datatype

```
abstype 'a stack = EmptyStack | >> of 'a stack * 'a
with
    infix >>
    val emptyStack = EmptyStack
    fun isEmptyStack EmptyStack = true
      | isEmptyStack (S>>v) = false
    fun push v S = S>>v
    fun top EmptyStack = error "top: empty stack"
      | top (S>>v) = v
    fun pop EmptyStack = error "pop: empty stack"
      | pop (S>>v) = S
    fun showStack EmptyStack = [ ]
      | showStack (S>>v) = v :: (showStack S)
end
```

We have thus defined a new list constructor, with $\Theta(1)$ time (direct) access to the elements *from the right*!

FIFO Queues

(Version of March 15, 2010, based on original by Yves Deville)

First-in first-out (FIFO) queues of objects of type α: α queue

- Addition of elements to the rear (*tail*)
- Deletion of elements from the front (*head*)

Operations

value emptyQueue
TYPE: α queue
VALUE: the empty queue

function isEmptyQueue Q
TYPE: α queue $\rightarrow$ bool; PRE: (none)
POST: **true** , if Q is empty; **false** , otherwise

function enqueue v Q
TYPE: $\alpha \rightarrow \alpha$ queue $\rightarrow \alpha$ queue; PRE: (none)
POST: the queue Q with v added as new tail element

function head Q
TYPE: α queue $\rightarrow \alpha$
PRE: Q is non-empty
POST: the head element of Q

function dequeue Q
TYPE: α queue $\rightarrow \alpha$ queue
PRE: Q is non-empty
POST: the queue Q without its head element

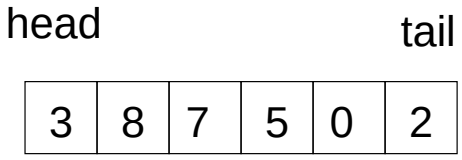
function showQueue Q
TYPE: α queue $\rightarrow \alpha$ list; PRE: (none)
POST: the representation of Q in list form, with the head of Q as head, etc

Realisation 1

Representation of a FIFO queue by a *list*:

type α queue = α **list**
REPRESENTATION CONVENTION: the head of the list is the head of the queue, the 2nd element of the list is behind the head of the queue, and so on, and the last element of the list is the tail of the queue

Example: the queue



is represented by the list [3,8,7,5,0,2]

Exercises

- Realise the **queue** ADT using this representation
- What is the time complexity of enqueuing an element?
- What is the time complexity of dequeuing an element?

Realisation 2

Representation of a FIFO queue by a *pair of lists*:

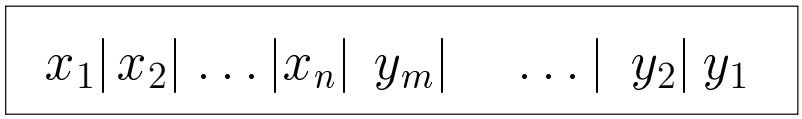
datatype α queue = Queue **of** α **list** $*$ α **list**

REPRESENTATION CONVENTION: the term

Queue $([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_m])$

represents the queue

head tail



REPRESENTATION INVARIANT: (see next slide)

- It is *now* possible to enqueue in $\Theta(1)$ time
- It is still possible to dequeue in $\Theta(1)$ time, but *only* if $n \geq 1$
- What if $n = 0$ while $m > 0$?!
- The same queue can thus be represented in *different* ways
- How to test the equality of two queues?

Normalisation

Objective: avoid the case where $n = 0$ while $m > 0$

When this case appears, transform (or: normalise)
the representation of the queue:
transform Queue $([\,], [y_1, \ldots, y_m])$ with $m > 0$
into Queue $([y_m, \ldots, y_1], [\,])$,
which indeed represents the same queue

We thus have:

REPRESENTATION INVARIANT: a non-empty queue is never represented by Queue $([\,], [y_1, \ldots, y_m])$

function normalise Q
TYPE: α queue $\rightarrow$ α queue
PRE: (none)
POST: if Q is of the form Queue $([\,], [y_1, \ldots, y_m])$
then Queue $([y_m, \ldots, y_1], [\,])$
else Q

Operations

Construction of an abstract datatype:
the normalise function may be *local* to the ADT,
as it is only used for realising some operations on queues

```
abstype 'a queue = Queue of 'a list * 'a list
with
  val emptyQueue = Queue ([ ],[ ])
  fun isEmptyQueue (Queue ([ ],[ ])) = true
    | isEmptyQueue (Queue (xs,ys)) = false
  fun head (Queue (x::xs,ys)) = x
    | head (Queue ([ ],[ ])) = error "head: empty queue"
    | head (Queue ([ ],y::ys)) = error "head: non-normalised queue"
  local
    fun normalise (Queue ([ ],ys)) = Queue (rev ys,[ ])
      | normalise Q = Q
  in
    fun enqueue v (Queue (xs,ys)) = normalise (Queue (xs,v::ys))
    fun dequeue (Queue (x::xs,ys)) = normalise (Queue (xs,ys))
      | dequeue (Queue ([ ],[ ])) = error "dequeue: empty queue"
      | dequeue (Queue ([ ],y::ys)) = error "dequeue: non-norm. queue"
  end
  fun showQueue (Queue (xs,ys)) = xs @ (rev ys)
  fun equalQueues Q1 Q2 = (showQueue Q1 = showQueue Q2)
end
```

- Why do the **head** and **dequeue** functions *not* normalise the queue instead of stopping the execution with an error?
- The normalisation and representation invariant are *hidden* in the realisation of the abstract datatype
- On average, the time of enqueuing and dequeuing is $\Theta(1)$
- This representation is thus *very* efficient!