Chapter 6
Abstract Datatypes

(Version of 17 November 2005)

1. Application: correctly parenthesised texts ........... 6.2
2. An abstract datatype for stacks ......................... 6.6
3. Realisation of the stack abstract datatype .............. 6.8
4. An abstract datatype for FIFO queues ................. 6.13
5. Realisation of the queue abstract datatype ............. 6.15
6. An abstract datatype for binary trees ................. 6.19
7. Realisation of the bTree abstract datatype ........... 6.21
8. An abstract datatype for binary search trees .... 6.27
9. Realisation of the bsTree abstract datatype ....... 6.29
6.1. Application: correctly parenthesised texts

Determine whether a text is correctly parenthesised

**Specification**

function parenthesised text
TYPE: char list → bool
PRE: (none)
POST: true if text is correctly parenthesised  
false otherwise

**Examples and counter-examples**

- 
\[ ((a+b) \times (c−d)) \]
  true
- 
\[ ((a+b) \times (c−d)) \]
  false
- 
\[ (a+b) \times ((c−d) \]
  false
- 
\[ (a[(b+c) \times d] + e) \times f \]
  true
- 
\[ (a[(b+c) \times d] + e] \times f \]
  false
- 
\[ (ab [+b(c \times)]) \]
  true
Analysis

- Ignore everything except the parentheses
- The number of left parentheses of each kind must be equal to the number of right parentheses of the same kind
- Each right parenthesis must correspond to a left parenthesis of the same kind, which precedes it
- Each left parenthesis must correspond to a right parenthesis of the same kind, which follows it
- Between two corresponding parentheses, the text must be correctly parenthesised

Strategy: Generalise the problem

```plaintext
function parGen S L
TYPE: ?? → char list → bool
PRE: S is a ‘list’ of left-parenthesis characters
POST: true if ‘S@L’ is correctly parenthesised
      false otherwise
```

Example

```
parGen ["(" , "]" ] ["[" , "]", "+", "+", "]", "+", "c", "])"]
shall be true because ([a]+b)−c is correctly parenthesised
```
Construction

Variant: the length of L

**Base case:** L is []

Then S@L is correctly parenthesised iff S is empty

**General case:** L is of the form (x::xs)

If x is not a parenthesis, then S@L is correctly parenthesised iff S@xs is correctly parenthesised

Hence the recursive call parGen S xs

If x is a left parenthesis, then S@L is correctly parenthesised iff S.x@xs is correctly parenthesised, where S.x denotes the addition of x to the end (top) of S (so S seems to be a stack)

Hence the recursive call parGen (S.x) xs

If x is a right parenthesis, then S@L is correctly parenthesised iff

- S has the corresponding left parenthesis at its top, and
- S'@xs is correctly parenthesised, where S' is S without its top element

Hence the recursive call parGen S' xs
Hence:

- Necessity of an *abstract datatype* for stacks
- Necessity of the auxiliary functions
  leftParenthesis, rightParenthesis, and corresponds

**function** parGen $S \ L$

*TYPE*: char stack $\rightarrow$ char list $\rightarrow$ bool

*PRE*: $S$ only has left-parenthesis characters

*POST*: `true` if rev(showStack $S$)$@L$ is correctly parenthesised
  `false` otherwise

**function** leftParenthesis $x$

*TYPE*: char $\rightarrow$ bool

*PRE*: (none)

*POST*: `true` if $x$ is a left parenthesis
  `false` otherwise

**function** rightParenthesis $x$

*TYPE*: char $\rightarrow$ bool

*PRE*: (none)

*POST*: `true` if $x$ is a right parenthesis
  `false` otherwise

**function** corresponds $x \ y$

*TYPE*: char $\rightarrow$ char $\rightarrow$ bool

*PRE*: (none)

*POST*: `true` if $x$ is a left parenthesis
  and $y$ is a right parenthesis corresponding to $x$
  `false` otherwise
6.2. An abstract datatype for stacks

Stacks of objects of type \( \alpha \): \( \alpha \) stack

Operations

**value** emptyStack
TYPE: \( \alpha \) stack
VALUE: the empty stack

**function** isEmptyStack \( S \)
TYPE: \( \alpha \) stack \( \rightarrow \) bool
PRE: (none)
POST: \( \text{true} \) if \( S \) is empty
\( \text{false} \) 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: \( \alpha \) stack \( \rightarrow \) \( \alpha \)
PRE: \( S \) is non-empty
POST: the top element of \( S \)

**function** pop \( S \)
TYPE: \( \alpha \) stack \( \rightarrow \) \( \alpha \) stack
PRE: \( S \) is non-empty
POST: the stack \( S \) without its top element
‘Formal’ semantics

isEmptyStack emptyStack = true
\(\forall v,S : \text{isEmptyStack} (\text{push} \ v \ S) = \text{false}\)
top emptyStack = ... error ...
\(\forall v,S : \text{top} (\text{push} \ v \ S) = v\)
pop emptyStack = ... error ...
\(\forall v,S : \text{pop} (\text{push} \ v \ S) = S\)

Example: correctly parenthesised text (parentheses.sml)

local
  fun leftParenthesis v = ...
  fun rightParenthesis w = ...
  fun corresponds v w = ...
  fun parGen S [ ] = isEmptyStack S
  | parGen S (x::xs) =
    if leftParenthesis x
    then parGen (push x S) xs
    else if rightParenthesis x
    then not (isEmptyStack S) andalso
        corresponds (top S) x andalso
        parGen (pop S) xs
    else parGen S xs
  in
  fun parenthesised text = parGen emptyStack text
end
6.3. Realisation of the stack abstract datatype

Version 1

Representation of a stack by a list:

```haskell
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

Realisation of the operations (stack1.sml)

```haskell
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
```

- This realisation does not force the usage of the `stack` type
- 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!
Version 2

Definition of a new constructed type using the list type:

```
datatype \( \alpha \) stack = Stack of \( \alpha \) 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

### Realisation of the 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 (stack2.sml)

Objective: encapsulate the definition of the stack type and its operations in a parameterised abstract datatype

\textbf{abstype } \texttt{'a stack = Stack of } \texttt{'a list} \\
\textbf{with} \\
\hspace{1em} \textbf{val} \hspace{0.5em} \text{emptyStack = Stack [ ]} \\
\hspace{1em} \textbf{fun} \hspace{0.5em} \text{isEmptyStack (Stack S) = (S = [ ])} \\
\hspace{1em} \textbf{fun} \hspace{0.5em} \text{push v (Stack S) = Stack (v::S)} \\
\hspace{1em} \textbf{fun} \hspace{0.5em} \text{top (Stack [ ]) = error "top: empty stack"} \\
\hspace{2em} \text{top (Stack (x::xs)) = x} \\
\hspace{1em} \textbf{fun} \hspace{0.5em} \text{pop (Stack [ ]) = error "pop: empty stack"} \\
\hspace{2em} \text{pop (Stack (x::xs)) = Stack xs} \\
\textbf{end}

- The stack type is an abstract datatype (ADT)
- The concrete representation of a stack is hidden
- An object of the stack type can only be manipulated via the functions defined in its ADT declaration
- The Stack 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 parameterisation allows the usage of stacks of integers, reals, strings, integer functions, etc, from a single definition!
6.3. Realisation of the stack abstract datatype

- \textbf{abstype} \ 'a stack = Stack of \ 'a list with \ldots \ ;

\textbf{type} \ 'a stack

\textbf{val} \ 'a emptyStack = \ - \ : \ 'a stack

\textbf{val} \ "'a isEmptyStack = fn : "'a stack \to \text{bool}

\ldots

- \textbf{push} 1 (Stack \ [ \ ] ) ;

\textit{Error: unbound variable or constructor: Stack}

- \textbf{push} 1 emptyStack ;

\textbf{val} \ it = \ - \ : \ int \ stack

It is \textit{impossible} to compare two stacks:

- \textbf{emptyStack} = \textbf{emptyStack} ;

\textit{Error: operator and operand don’t agree}

\textit{[equality type required]}

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

\textbf{function} \ showStack \ S

\textbf{TYPE:} \ \alpha \ \text{stack} \to \ \alpha \ \text{list}

\textbf{PRE:} \ (\text{none})

\textbf{POST:} \ the \ representation \ of \ S \ in \ list \ form, \ with \ the \ top \ of \ S \ as \ head, \ etc

\textbf{abstype} \ \ 'a stack = \text{Stack of} \ \ 'a \ list

\textbf{with}

\ldots

\textbf{fun} \ showStack \ (Stack \ S) = \ S

\textbf{end}

\- \ The \ result \ of \ \textbf{showStack} \ is \ \textit{not} \ of \ the \ \text{stack} \ type

\- \ One \ can \ thus \ \textit{not} \ apply \ the \ stack \ operations \ to \ it
Version 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 (`stack3.sml`)

```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,
but with access to the elements *from the right*!
6.4. An abstract datatype for FIFO queues

First-in first-out (FIFO) queues of objects of type $\alpha$: $\alpha$ queue

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

Operations

value emptyQueue
TYPE: $\alpha$ queue
VALUE: the empty queue

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

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

‘Formal’ semantics

isEmptyQueue emptyQueue = true
\( \forall v,Q : \) isEmptyQueue (enqueue v Q) = false
head emptyQueue = ... error ...
\( \forall v,Q : \) head (enqueue v Q) = if isEmptyQueue Q then v else head Q
dequeue emptyQueue = ... error ...
\( \forall v,Q : \) dequeue (enqueue v Q) = if isEmptyQueue Q then emptyQueue else enqueue v (dequeue Q)
6.5. Realisation of the queue abstract datatype

Version 1

Representation of a FIFO queue by a list:

```haskell
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

```
head  tail
3 8 7 5 0 2
```

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?
Version 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 ([x₁, x₂, ..., xₙ], [y₁, y₂, ..., yₘ])
```

represents the queue

```
\begin{array}{cccccccc}
\text{read} & \text{head} & \text{tail} \\
\hline
x₁ & x₂ & \ldots & xₙ & yₘ & \ldots & y₂ & y₁
\end{array}
```

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 $\text{Queue} ([], [y_1, \ldots, y_m])$ with $m > 0$

into $\text{Queue} ([y_m, \ldots, y_1], [])$, which indeed represents the same queue.

We thus have:

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

function normalise Q

TYPE: $\alpha$ queue $\rightarrow \alpha$ queue

PRE: (none)

POST: if Q is of the form $\text{Queue} ([], [y_1, \ldots, y_m])$

then $\text{Queue} ([y_m, \ldots, y_1], [])$

else Q

Realisation of the operations (queue2.sm1)

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.
Ch. 6: Abstract Datatypes

6.5. Realisation of the queue abstract datatype

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!
6.6. An abstract datatype for binary trees

Concepts and terminology

Binary trees of objects of type $\alpha$: $\alpha\ bTree$

Operations

value emptyBtree
TYPE: $\alpha\ bTree$
VALUE: the empty binary tree

function isEmptyBtree T
TYPE: $\alpha\ bTree \rightarrow bool$
PRE: (none)
POST: true if T is empty
false otherwise
function consBtree v L R
TYPE: $\alpha \rightarrow \alpha$ bTree $\rightarrow \alpha$ bTree $\rightarrow \alpha$ bTree
PRE: (none)
POST: the binary tree with root v, left sub-tree L, and right sub-tree R

function left T
TYPE: $\alpha$ bTree $\rightarrow \alpha$ bTree
PRE: T is non-empty
POST: the left sub-tree of T

function right T
TYPE: $\alpha$ bTree $\rightarrow \alpha$ bTree
PRE: T is non-empty
POST: the right sub-tree of T

function root T
TYPE: $\alpha$ bTree $\rightarrow \alpha$
PRE: T is non-empty
POST: the root of T

‘Formal’ semantics

isEmptyBtree emptyBtree = true
$\forall v, L, R : $ isEmptyBtree (consBtree v L R) = false
root emptyBtree = ... error ...
$\forall v, L, R : $ root (consBtree v L R) = v
left emptyBtree = ... error ...
$\forall v, L, R : $ left (consBtree v L R) = L
right emptyBtree = ... error ...
$\forall v, L, R : $ right (consBtree v L R) = R
6.7. Realisation of the bTree abstract datatype

Representation

datatype bTree = Void
  | Bt of int * bTree * bTree

REPRESENTATION CONVENTION: a binary tree with root x, left subtree L, and right subtree R is represented by Bt(x,L,R)
EXAMPLE: Bt(4, Bt(2, Bt(1,Void,Void), Bt(3,Void,Void)), Bt(8, Bt(6, Bt(5,Void,Void), Bt(7,Void,Void)), Bt(9,Void,Void)))

Realisation of the operations (bTree.sml)

abstype 'a bTree = Void
  | Bt of 'a * 'a bTree * 'a bTree

with
  val emptyBtree = Void
  fun isEmptyBtree Void = true
  | isEmptyBtre (Bt(v,L,R)) = false
  fun consBtree v L R = Bt(v,L,R)
  fun left Void = error "left: empty bTree"
  | left (Bt(v,L,R)) = L
  fun right Void = error "right: empty bTree"
  | right (Bt(v,L,R)) = R
  fun root Void = error "root: empty bTree"
  | root (Bt(v,L,R)) = v

end
Walk operations (inorder.sml)

function inorder T
TYPE: \(\alpha\) bTree \(\rightarrow\) \(\alpha\) list
PRE: (none)
POST: the nodes of T upon an inorder walk
fun inorder Void = [ ]
  | inorder (Bt(v,L,R)) = (inorder L) @ (v :: inorder R)

No tail recursion!
It takes \(\Theta(n \log n)\) time for a binary tree of \(n\) nodes.

function inorderGen T acc
TYPE: \(\alpha\) bTree \(\rightarrow\) \(\alpha\) list \(\rightarrow\) \(\alpha\) list
PRE: (none)
POST: (the nodes of T upon an inorder walk) @ acc
fun inorderGen Void acc = acc
  | inorderGen (Bt(v,L,R)) acc =
      let val rAcc = inorderGen R acc
      in inorderGen L (v::rAcc) end

fun inorder t = inorderGen t [ ]

One tail recursion! No call to @ (concatenation)!
It takes \(\Theta(n)\) time for a binary tree of \(n\) nodes

Exercises

• Efficiently realise the preorder and postorder walks
  of a binary tree, and analyse the underlying algorithms
• How to test the equality of two binary trees?
Other operations

function exists k T
TYPE: \( \alpha^= \rightarrow \alpha^= \text{bTree} \rightarrow \text{bool} \)
PRE: (none)
POST: true if T contains node k
false otherwise

function insert k T
TYPE: \( \alpha^= \rightarrow \alpha^= \text{bTree} \rightarrow \alpha^= \text{bTree} \)
PRE: (none)
POST: T with node k

function delete k T
TYPE: \( \alpha^= \rightarrow \alpha^= \text{bTree} \rightarrow \alpha^= \text{bTree} \)
PRE: (none)
POST: if exists k T, then T without one occurrence of node k, otherwise T

function nbNodes T
TYPE: \( \alpha \text{ bTree} \rightarrow \text{int} \)
PRE: (none)
POST: the number of nodes of T

function nbLeaves T
TYPE: \( \alpha \text{ bTree} \rightarrow \text{int} \)
PRE: (none)
POST: the number of leaves of T

Exercises

• Efficiently realise these five functions

• Show that their algorithms at worst take \( \Theta(n) \) time,
  if not \( \Theta(1) \) time, on a binary tree with initially \( n \) nodes
Height of a binary tree \((\text{height.sml})\)

- The **height of a node** is the length of the longest path (measured in its number of nodes) from that node to a leaf.
- The **height of a tree** is the height of its root.

**function** \(\text{height} \ T\)

**TYPE:** \(\alpha \ b\Tree \rightarrow \text{int}\)

**PRE:** (none) ; **POST:** the height of \(T\)

**fun** \(\text{height Void} = 0\)

\(\quad | \ \text{height (Bt(v,L,R))} = 1 + \text{Int.max (height L, height R)}\)

No tail recursion!

It takes \(\Theta(n)\) time for a binary tree of \(n\) nodes.

Note that \(\text{heightGen T acc} = \text{acc} + \text{height T}\) does *not* suffice to get a tail recursion: why?!

**function** \(\text{heightGen T acc hMax}\)

**TYPE:** \(\alpha \ b\Tree \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int}\)

**PRE:** (none) ; **POST:** \(\text{max (acc + height T, hMax)}\)

**fun** \(\text{heightGen Void acc hMax} = \text{Int.max (acc, hMax)}\)

\(\quad | \ \text{heightGen (Bt(v,L,R)) acc hMax} =\)

\(\quad \quad | \ \text{let val h1 = heightGen R (acc+1) hMax}\)

\(\quad \quad \quad | \ \text{in heightGen L (acc+1) h1 end}\)

**fun** \(\text{height2 bt} = \text{heightGen bt 0 0}\)

One tail recursion!

It also takes \(\Theta(n)\) *time* for a binary tree of \(n\) nodes, but it takes less *space*!
6.8. An ADT for Binary Search Trees (BSTs)

Concepts and terminology (see the tree on slide 6.19)

Binary search trees of nodes of type \((\alpha, \beta)\): \((\alpha, \beta)\) bsTree where:

- \(\alpha\) is the type of the keys (need for an equality test)
- \(\beta\) is the type of the satellite data for each key

are binary trees with:

(REPRESENTATION) INVARIANT: for a binary search tree with \((k, s)\) in the root, left subtree L, and right subtree R:
  - every element of L has a key smaller than \(k\)
  - every element of R has a key larger than \(k\)

Note that we (arbitrarily) ruled out duplicate keys

Benefits

- The inorder walk of a binary search tree lists its nodes by increasing order of their keys!
- The basic operations at worst take \(\Theta(n)\) time on a binary search tree with (initially) \(n\) nodes, but they take \(O(lg \ n)\) time on randomly built binary search trees

Let us restrict our realisation to integer keys: \(\beta\) bsTree
Some operations

value emptyBsTree
TYPE: β bsTree
VALUE: the empty binary search tree

function isEmptyBsTree T
TYPE: β bsTree → bool
PRE: (none)
POST: true if T is empty
false otherwise

function exists k T
TYPE: int → β bsTree → bool
PRE: (none)
POST: true if T contains a node with key k
false otherwise

function insert (k,s) T
TYPE: (int *β) → β bsTree → β bsTree
PRE: (none)
POST: if exists k T, then T with s as satellite data for key k
otherwise T with node (k,s)

function retrieve k T
TYPE: int → β bsTree → β
PRE: exists k T
POST: the satellite data associated to key k in T

function delete k T
TYPE: int → β bsTree → β bsTree
PRE: (none)
POST: if exists k T, then T without the node with key k, otherwise T
6.9. Realisation of the bsTree ADT

Representation

datatype 'b bsTree = Void
    | Bst of (int * 'b) * 'b bsTree * 'b bsTree

REPRESENTATION CONVENTION: a BST with (k,s) in the root, left subtree L, and right subtree R is represented by Bst((k,s),L,R)
REPRESENTATION INVARIANT: (see slide 6.25)

Realisation of the operations (bsTree.sml)

val emptyBsTree = Void

fun isEmptyBsTree Void = true
    | isEmptyBsTree (Bst((key,sat),L,R)) = false

fun exists k Void = false
    | exists k (Bst((key,sat),L,R)) =
        if k = key then true
        else if k < key then exists k L
        else (k > key) exists k R

fun insert (k,s) Void = Bst((k,s),Void,Void)
    | insert (k,s) (Bst((key,sat),L,R)) =
        if k = key then Bst((k,s),L,R)
        else if k < key then Bst((key,sat), (insert (k,s) L), R)
        else (k > key) Bst((key,sat), L, (insert (k,s) R))

fun retrieve k Void = error "retrieve: non-existing node"
    | retrieve k (Bst((key,sat),L,R)) =
        if k = key then sat
        else if k < key then retrieve k L
        else (k > key) retrieve k R
When deleting a node \((\text{key}, \text{sat})\) whose subtrees \(L\) and \(R\) are \textit{both} non-empty, we must not violate the repr. invariant!

1. Replace \((\text{key}, \text{sat})\) by the node with the \textit{maximal} key of \(L\), whose key is smaller than the key of \textit{any} node of \(R\) (one could also replace by the node with the \textit{minimal} key of \(R\))

2. Remove this node with the maximal key from \(L\)

So we need a \texttt{deleteMax} function:

\[
\text{function deleteMax } T \\
\text{TYPE: } \beta \text{ bsTree } \rightarrow (\text{int } \ast \beta) \ast \beta \text{ bsTree} \\
\text{PRE: } T \text{ is non-empty} \\
\text{POST: (max, NT), where max is the node of T with the maximal key, and NT is T without max}
\]

\[
\text{fun deleteMax Void } = \text{error } "\text{deleteMax: empty bsTree}"
\]
\!
\[
\text{fun deleteMax (Bst(r,L,Void): 'b bsTree) } = (r, L)
\]
\]
\[
\text{fun deleteMax (Bst(r,L,R)) } =
\]
\[
\text{let val } (\text{max, newR) } = \text{deleteMax R}
\]
\[
\text{in } (\text{max, Bst(r,L,newR)) end}
\]
\]
\[
\text{fun delete k Void } = \text{Void}
\]
\[
\text{fun delete k (Bst((key,sat),L,R)) } =
\]
\[
\text{if } k < \text{key then } \text{Bst((key,sat), (delete k L), R)}
\]
\[
\text{else if } k > \text{key then } \text{Bst((key,sat), L, (delete k R))}
\]
\[
\text{else } (\ast k = \text{key } \ast)
\]
\[
\text{case (L,R) of}
\]
\[
(\text{Void, _ } ) \Rightarrow R
\]
\]
\[
(\_ ,\text{Void}) \Rightarrow L
\]
\]
\[
(\_ ,\_ ) \Rightarrow \text{let val } (\text{max, newL) } = \text{deleteMax L}
\]
\[
\text{in Bst(max,newL,R) end}
\]