Lecture 9: Declarative Programming. Recursion and Iteration over Lists and Trees.

September 27nd, 2011

Chapter 3 — Overview

§3.1 What is declarativeness?
§3.2 Iterative computation.
§3.3 Recursive computation.
§3.4 Programming with recursion.
§3.5 *Time and space efficiency.*[1]
§3.6 Higher-order programming.
§3.7 Abstract data types.
§3.8 *Nondeclarative needs.*

[1] Items in gray are non-mandatory.

Lecture Outline

Declarative Programming
Declarative Programming with XML-Related Technologies
Recursion and Iteration with Lists
Recursion and Iteration with Trees
Difference Lists
Recursion and Iteration with Trees *contd*

Summary

Declarative Programming

What is declarative programming?

The term 'declarative programming' may be understood in differrent ways. According to CTMCP, declarative programming is programming based on using declarative operations, i.e., operations that

- given a specific input always produce the same output;
- the result cannot be influenced by changes in the state of some objects external to the operation;
- the operation does not have any internal state that can be changed.

In principle, a declarative operation is deterministic:

- its behaviour is consistent;
- its behaviour is preditable.

This use of 'declarative' in CTMCP is close to what is otherwise called 'functional'.

Declarative Programming *contd*

Usually, a programming language is considered declarative if programs written in the language

- state what properties the desired results of a computation must have, but
- do not state how to achieve the results.

Note: This is not exactly what CTMCP means by 'declarative', as most of the declarative$_{\text{CTMCP}}$ programming discussed in Chapters 3 and 4 is in fact based on defining and executing procedures, rather than on asserting constraints.

In a prototypical declarative programming language, there is

- no stress on the order of operations;
- no explicit control over the flow of a computation;
- no destructive assignment.

Algorithms for processing queries are part of the language, rather than of programs in the language.[2]

[2] In fact, most declarative (in the above sense) languages allow the programmer to control the execution by, e.g., ordering the assertions in different ways.

Declarative Programming *contd*

- Descriptive: languages for describing data structures, formatting, etc. (e.g., HTML, XML, CSS).
- Programmable with observational declarativeness: declarative programming in a language that does not guarantee declarativity (e.g., Java).
- Programmable with definitional declarativeness: programming in a language that guarantees declarativeness.
  - the declarative language models of Oz (Chapters 2, 3 and 4);
  - pure functional programming (e.g., Haskell);
  - pure logic programming (Mercury?).

Declarative Programming with XML

Many of the technologies related to XML can be seen as based on declarative (programming) languages—documents written in, e.g., CSS, DTD, XSD, XSL, or even pure XML,[3] can all be seen as examples of declarative programs:

- an XML document provides a structured representation of some content;
- a DTD or an XSD document specifies how XML documents representing some particular kind of content should look like;
- an XSL document specifies a transformation of XML documents into other (possibly XML) form;
- a CSS document specifies how an XML document should appear in a browser (or in a printout, etc.).

[3] CSS: Cascading Style Sheets; DTD: Document Type Definition; XSD: XML Schema Definition; XSL: Extensible Stylesheet Language; XML: Extensible Markup Language.

Declarative Programming with XML *contd*

Example (Representing Oz records with XML)

`code/record.xml`

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/css" href="record.css"?>
<record label="tree">
   <field feature="right" value="Right" />
   <field feature="left" value="Left" />
</record>
```

This XML document represents an Oz record value expression:

- `tree(left:Left right:Right)`

Example (Specifying XML documents with DTD)

`code/record.dtd`

```
<!ELEMENT record (field*)>
<!ELEMENT field EMPTY>
<!ATTLIST record
   label CDATA #REQUIRED>
<!ATTLIST field
   feature CDATA #IMPLIED
   value CDATA #REQUIRED>
```

This DTD document specifies the form of XML documents describing records.

- The XML document `record.xml` is valid wrt. the DTD above.

Example (Specifying XML documents with XSD)

`code/record.xsd`

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="record" type="Record" />
   <xs:complexType name="Record">
      <xs:sequence>
         <xs:element name="field" type="Field"
                     minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>
      <xs:attribute name="label" type="Atom" />
   </xs:complexType>
   ...
   <xs:simpleType name="Atom">
      <xs:restriction base="xs:string">
         <xs:pattern value="[a-z][a-zA-Z0-9]*" />
      ...
```

This XSD document specifies the form of XML documents describing records.[4]

- The XML document `record.xml` is valid wrt. the XSD above.

[4] XSD documents are themselves XML documents, and can be processed as any other XML files.

Example (Specifying XML transformations with XSL)

`code/record.xsl`

```
...
<xsl:template match="record">
   <xsl:value-of select="@label" />
   <xsl:if test="field">
      <xsl:text>(&#10;</xsl:text>
         <xsl:for-each select="field">
            <xsl:sort select="@feature" />
            <xsl:text>  </xsl:text>
            <xsl:value-of select="@feature" />
            <xsl:text>:</xsl:text>
            <xsl:value-of select="@value" />
            <xsl:text>&#10;</xsl:text>
         </xsl:for-each>
         ...
```

This XSL document specifies a transformation of XML documents describing records into plain text.[5]

[5] XSL documents are themselves XML documents, and can be processed as any other XML files. The XSL language is Turing complete; see, e.g., http://www.unidex.com/turing/ .

Example (Specifying XML presentation with CSS)

`code/record.css`

```
record {
   font-family: monospace;
}
record:before {
   content: attr(label) "(";
}
record:after {
   content: ")";
}
field {
   display: list-item;
   content: attr(feature) ":" attr(value);
   padding-left: 2em;
}
```

This CSS document specifies how a record XML document should be presented in a browser.

Declarative Programming with XML *contd*

Try it!

- Use an XML validator to validate both XML documents against DTD and XSD:

```
$ xmllint --dtdvalid record.dtd --noout record.xml
$ xmllint --schema record.xsd --noout record.xml
```

- Use an XSLT processor to transform an XML document according to an XSL template:

```
$ xsltproc record.xsl record.xml
```

- Use a browser to format and display an XML document according to a CSS specification:

```
$ firefox record.xml &
```


Recursion and Iteration with Lists

We have already discussed recursive procedures and the difference between recursive and iterative computations.

- A recursive procedure is one that calls itself, directly or indirectly (e.g., through mutual recursion).

The execution of a recursive procedure may lead to

- a recursive process (stack size grows linearly or superlinearly);
- an iterative process (stack size is constant).

Recursion and Iteration with Lists *contd*

Example (Computing the length of a list, linear stack size)

```
fun {Length List}
   case List of nil then 0
   [] _|Tail then 1 + {Length Tail}
   end
end
```

This approach implements a divide and conquer strategy: to compute the length of a list,

- divide the list into smaller parts (the head and the tail),
- solve the problem for the smaller input (the tail),
- combine the partial results (add 1 for the head to the length of the tail).

Recursion and Iteration with Lists *contd*

Example (Computing the length of a list, constant stack size)

```
fun {Length List}
   fun {Iterate Computed Remaining}
      case Remaining of nil then Computed
      [] _|Tail then {Iterate Computed + 1 Tail}
      end
   end
in
   {Iterate 0 List}
end
```

This approach implements what could be called a 'contribute and forward' strategy: to compute the length of a list,

- start with an already established result for the base case (the empty list) and the whole input list,
- if the list is empty, return the already computed result,
- otherwise, update the partial result (add 1 for a non-empty list containing at least one lement) and the input (remove the head of the list), and repeat the procedure.

Recursion and Iteration with Lists *contd*

A computation is process of transition from a state to a state.

- In a recursive process, the state of the computation is collectively described by the input to a nested call and the sequence of postponed operations.
- In an iterative process, the state of the computation is completely described by the input to the nested call.

In an iterative process, a complete description of the state of the computation is threaded through the sequence of recursive calls.[6]

[6] In the declarative model of computation. In a non-declarative setting, the same can be achieved by modifying a finite number of mutable state variables.

Recursion and Iteration with Lists *contd*

Following the observation above, we can summarize iterative processes with a generic scheme.

Control abstraction for iterative processes

iterate(S_i):
  if *final*(S_i) then return S_i
  else do *iterate*(*update*(S_i))

To apply the scheme, we need to

- construct a complete description of the initial state S_0;
- define the Boolean function *final* that tests whether a state is final;
- define the function *update* that maps a state onto the next state;
- apply *iterate* to S_0, using *final* and *update*.

Recursion and Iteration with Lists *contd*

We can easily implement the generic scheme in Oz.

Control abstraction for iterative processes *contd*

```
fun {Iterate CurrentState IsFinal Update}
   if {IsFinal CurrentState} then
      CurrentState
   else
      {Iterate {Update CurrentState} IsFinal Update}
   end
end
```

- `CurrentState` is an encoding of the current state.
- `IsFinal` is a one-argument state-testing Boolean function.
- `Update` is a state-to-state mapping function.

Recursion and Iteration with Lists *contd*

Example (Iterative `Factorial`, direct implementation[7])

```
fun {Factorial N}
   fun {Iterate Iterator Result}
      if Iterator > N then Result
      else {Iterate Iterator+1 Iterator*Result} end in
   {Iterate 2 1}
end
```

At each call, the state of the computation is completely specified by two values:

- the value of the iterator variable `Iterator`;
- the value of the accumulator variable `Result`.

To use the control abstraction defined previously, we need to wrap these two values into a complete state description, and also implement `IsFinal` and `Update`.

[7] Yes, you have seen this before.

Recursion and Iteration with Lists *contd*

States can be encoded as records.

- In the case of `Factorial`, we need two fields:

```
state(iterator:Iterator result:Result)
```

- `IsFinal` needs to test whether the iterator is larger than the original value for which the factorial is computed:[8]

```
fun {IsFinal State N}
   State.iterator > N
end
```

- `Update` needs to increase the iterator and multiply the result:

```
fun {Update State}
   state(iterator:State.iterator+1
         result:State.iterator*State.result)
end
```

[8] The `IsFinal` function in the control abstraction takes one argument; this can be easily fixed by defining `IsFinal` within `Factorial`—using lexical scoping.

Recursion and Iteration with Lists *contd*

Example (Iterative `Factorial` using control abstraction)

```
fun {Factorial N}
   {Iterate
    state(iterator:2 result:1)             % CurrentState
    fun {$ State}                          % IsFinal
        State.iterator > N
    end
    fun {$ State}                          % Update
        state(iterator:State.iterator+1
              result:State.iterator*State.result)
    end
   }.result
end
```

Note: `Iterate` returns a complete encoding of the final state; the factorial is in the `result` field.

Recursion and Iteration with Lists *contd*

Example (Iterative `Factorial` using control abstraction, alternative)

```
fun {Factorial N}
   {Iterate
    state(iterator:N result:1)             % CurrentState
    fun {$ State}                          % IsFinal
        State.iterator < 2 end
    fun {$ State}                          % Update
        state(iterator:State.iterator-1
              result:State.iterator*State.result)
   }.result
end
```

This version performs a top-down computation, while the previous performed a bottom-up ccomputation; both keep the stack constant.

Recursion and Iteration with Lists *contd*

In some cases replacing recursive processes with iterative processes may be a little tricky.

Example (Reversing a list)

```
fun {Reverse List}
   case List of nil then nil
   [] Head|Tail then {Append {Reverse Tail} [Head]}
   end
end
```

The procedure is not tail-recursive, and the process is not iterative.[9] How can we fix this?

[9] The use of `Append`, which has $O(n)$ runtime, causes `Reverse` to have $O(n^2)$ runtime (where n is the length of the input list).

Recursion and Iteration with Lists *contd*

We can completely encode the states of the computation by including (e.g., in a record) the part of the input list not yet processed and the already constructed partial result.

Example (Reversing a list, iterative)

```
fun {Reverse List}
   {Iterate
    List#nil                    % CurrentState
    fun {$ List#_}              % IsFinal
       List == nil end
    fun {$ (Head|Tail)#Result}  % Update
       Tail#(Head|Result) end
   }.2
end
```

At each step, the state is completely specified as a pair of lists. The computation is $O(n)$ in time (and space), since at each step we perform only constant-time operations, and there are `N` steps.

Recursion and Iteration with Lists *contd*

Note the syntactic extension:

- The practical language code

```
fun {$ (Head|Tail)#Result}
   ...
end
```

- is shorthand for the quasi-kernel language code[10]

```
proc {$ Input Output}
   case Input of '#'(List Result) then
      case List of '|'(Head Tail) then
         ...
         Output = ...
      end end
end
```

- What if the input does not match `(Head|Tail)#Result`?

[10] This is not legal in the kernel language—why?

Recursion and Iteration with Lists *contd*

According to CTMCP, a computation that keeps the stack constant in size is iterative. However, it is not only stack that matters.

Example (Iteration?)

```
fun {Factorial N}
  fun {Factorial N Compute}
    if N==0 then {Compute}
    else {Factorial N-1 fun {$} N*{Compute} end}
  end
in
  {Factorial N fun {$} 1 end}
end
```

The (internal) `Factorial` is tail-recursive, and the computation does keep the stack constant in size. However, there is a problem with the heap:

- each nested call to `Factorial` creates a new closure that contains a mapping for `N` from the calling `Factorial`.
- The number of variables allocated on the heap and reachable throughout the whole computation is linear, not constant, in `N`.

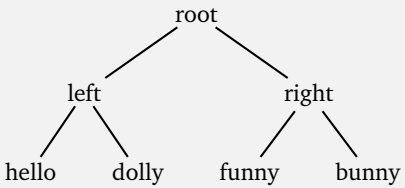
Recursion and Iteration with Trees

In what follows, we have a look at a few algorithms that operate on (binary) trees. We will represent trees as records:

- A tree is either a leaf (the ground case), or a binary tree where both branches are trees (the recursive case).
- Each node in a tree has some content.
- Leaf nodes are represented as records with the label `leaf` and one field (conveniently named `1`) holding the node's value.
- Non-leaf nodes are represented as records with the label `tree` and three fields (conveniently named `1`, `2`, and `3`) holding the node's value and the tree's branches.[11]

[11] For simplicity, we will operate on binary trees, but it is straightforward to generalize the discussion.

Example (Representing trees with records)



code/tree.oz

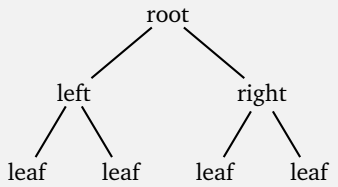
```
Tree =
tree(root
     tree(left leaf(hello) leaf(dolly))
     tree(right leaf(funny) leaf(bunny)))
```


Let us consider the following problem:[12]

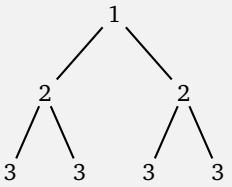
- Given a tree, construct a tree of the same shape but with every node labelled with its depth in the tree.

Example (Depthizing a tree)

an input tree:



the result tree:



[12] In the lack of a better name, call it 'depthizing a tree'.

Example (Depthizing a tree)

code/depthize.oz

```
fun {Depthize Tree}
   fun {Recur Tree Depth}
      case Tree
      of leaf(_) then
         leaf(Depth)
      [] tree(_ Left Right) then
         tree(Depth
              {Recur Left Depth+1}
              {Recur Right Depth+1}) end end in
   {Recur Tree 1}
end
```

- `Depthize` performs (indirectly) a series of recursive calls.
- Each recursive call receives as arguments a subtree and an increased depth value, and constructs an analogous tree with new labels.

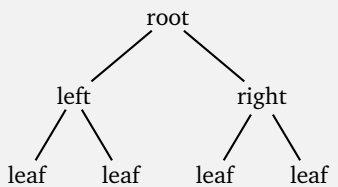
Easy.

Let us now consider the following problem:

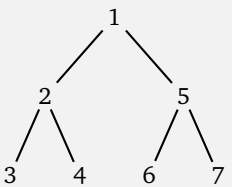
- Given a tree, construct a tree with the same shape, but label its nodes with successive integers, starting from 1, in the depth-first order.

Example (Depth-first tree numbering)

an input tree:



the result tree:



Recursion and Iteration with Trees *contd*

Example (Depth-first tree numbering)

`code/dfn.oz`

```
fun {DFN Tree}
   fun {Recur Tree First ?Last}
      case Tree
      of leaf(_) then
         Last = First
         leaf(First)
      [] tree(_ Left Right) then
         Intermediate in
         tree(First
              {Recur Left First+1 Intermediate}
              {Recur Right Intermediate+1 Last}) end end in
   {Recur Tree 1 _}
end
```

- `Recur` takes as input a tree (`Tree`) and an integer (`First`) to be used as the label for the first node found,
- binds its explicit return argument (`Last`) to the integer it used to label the last node found, and
- returns a re-labelled tree.

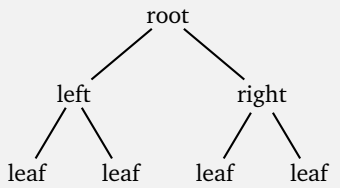
Recursion and Iteration with Trees *contd*

Let us now consider the following problem:

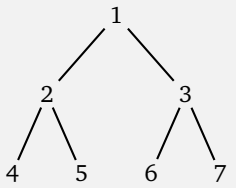
- Given a tree, construct a tree with the same shape, but label its nodes with successive integers, starting from 1, in the breadth-first order.

Example (Breadth-first tree numbering)

an input tree:



the result tree:



Recursion and Iteration with Trees *contd*

How do we proceed?

- In the depth-first case we used (implicitly) a LIFO structure (a stack) to store the nodes yet to be processed.[13]
- In the breadth-first case, we need to (explicitly) use a FIFO structure (a queue) to store the nodes yet to be processed.

In what follows, we consider a few alternatives for implementing a queue.

[13] `Recur` was a non-tail recursive procedure, and we exploited the fact that the recursive calls cause semantic statements to be pushed onto the semantic stack—we used the semantic stack as a store for unprocessed nodes.

Difference Lists

We can implement queues using plain lists.

Example (Naive implementation of queues)

`code/queue-naive.oz`

```
fun {Queue}
   nil end
fun {Enqueue Queue Item}
   Item|Queue end
fun {Dequeue Queue ?Item}
   case Queue of Head|nil then
      Item = Head nil
   [] Head|Tail then
      Head|{Dequeue Tail Item} end end
```

- `Queue` returns a new, empty queue (the empty list).
- `Enqueue` returns an extended queue, with a new element added at the front of the list.
- `Dequeue` returns a reduced queue, with the last element from the list removed (and returned in the explicit return argument).[14]

[14] You may want to add code for handling the case when the queue is empty.

Difference Lists *contd*

The implementation above is inefficient: `Dequeue` needs to traverse the whole list to return the last element (`Dequeue`'s performance is $O(n)$).

Example (Naive implementation of queues *contd*)

`code/queue-naive-alternative.oz`

```
fun {Queue}
   nil end
fun {Enqueue Queue Item}
   {Append Queue [Item]} end
fun {Dequeue Queue ?Item}
   case Queue of Head|Tail then
      Item = Head
      Tail end end
```

- `Queue` returns a new, empty queue (the empty list).
- `Enqueue` returns an extended queue, with a new element added at the end of the list.
- `Dequeue` returns a reduced queue, with the first element from the list removed (and returned in the explicit return argument).

Difference Lists *contd*

The implementation above is no better: it is now `Enqueue` that needs to traverse the whole list to add an element—this complexity is hidden in the single call to `Append`.[15]

- We need another solution; for example, we can implement queues using difference lists.

Difference lists

A difference list is a data structure composed of two plain lists l_1 and l_2 such that l_2 is a suffix of l_1.

A difference list can be interpreted as representing a list equivalent to the prefix of l_1 obtained by removing the l_2 suffix.

[15] In principle, the underlying implementation of lists and `Append` might make such operations $O(1)$.

Difference Lists *contd*

Example (Difference lists)

Difference list	Interpretation
`Suffix#Suffix`	`nil`
`nil#nil`	`nil`
`[a]#[a]`	`nil`
`(a|b|c|Suffix)#Suffix`	`[a b c]`
`(a|b|c|d|Suffix)#(d|Suffix)`	`[a b c]`
`[a b c d]#[d]`	`[a b c]`

The pair `(a|b|c|Suffix)#Suffix` can be interpreted as representing the list `[a b c]` irrespectively of the value of `Suffix`. In particular, `Suffix` may be

- `nil`,
- any non-empty list,
- any non-list object,
- an unbound variable.[16]

[16] Strictly speaking, the last two cases do not follow the definition above, since none of the components of such a difference list is a plain list (in the second case, possibly not yet a list).

Difference Lists *contd*

With an unbound variable as the suffix, difference lists are an efficient alternative to plain lists.

Example (`Append` for plain and difference lists)

plain lists:

```
fun {Append List1 List2}
   case List1#List2
   of nil#_ then List2
   [] _#nil then List1
   [] (Head|Tail)#_ then Head1|{Append Tail List2} end end
```

difference lists:

```
fun {Append List1 List2}
   (Start1#End1)#(Start2#End2) = List1#List2 in
   End1 = Start2
   Start1#End2 end
```

Note: The difference list version exploits the direct access to the unbound suffix of the first list; the operation is $O(1)$.

Difference Lists *contd*

Two more versions of `Append`:

Example (`Append` for difference lists)

binding `End1` with `Start2` by capturing them with a single identifier:

```
fun {Append List1 List2}
   (Start1#Unify)#(Unify#End2) = List1#List2 in
   Start1#End2 end
```

with implicit pattern matching in the function's parameter list:

```
fun {Append Start1#End1 Start2#End2}
   End1 = Start2
   Start1#End2 end
```

Note: Once you have appended a difference list to another difference list, the latter can no longer be appended to using `Append` (though it still represents the same list).

Difference Lists *contd*

What about the length of a list?

Example (`Length` for plain and difference lists)

plain lists:

```
fun {Length List}
   case List of nil then 0
   [] _|Tail then 1 + {Length Tail} end end
```

[17]

difference lists:

```
fun {Length Start#End}
   case Start of !End then 0
   [] _|Tail then 1 + {Length Tail#End} end end
```

Oops! If the list is not empty (`Start` $\neq$ `End`), the `case` statement freezes over the first clause (`End` is unbound, `case` waits for a pattern to match against).

[17] This version is not tail-recursive, but we have already seen a tail-recursive variant.

Difference Lists *contd*

What if we reverse the order of clauses in the `case` statement?

Example (`Length` for difference lists)

```
fun {Length Start#End}
   case Start of _|Tail then 1 + {Length Tail#End}
   [] !End then 0 end end
```

Oops! It won't help: if the list is empty (`Start` = `End`), the `case` statement freezes over the first clause (`Start` is unbound, `case` waits for a value to match against the pattern).

We need to encode the length of a difference list explicitly.

Example (Difference lists with explicit length variable)

```
(a|b|c|Suffix)#Suffix#3   % represents [a b c]
```


Difference Lists *contd*

Can we implement a procedure that converts plain lists into difference lists with unbound variables as the suffix? For example, `{AsDifferenceList [1]}` should return `(1|Unbound)#Unbound`, where `Unbound` is an unbound variable.

Example (Wrapping plain lists into difference lists)

`code/asdl.oz`

```
fun {AsDifferenceList List}
   Unbound
   fun {Iterate List}
      case List of nil then Unbound
      [] Head|Tail then Head|{Iterate Tail} end end in
   {Iterate List}#Unbound
end
```

- Produce a copy of the original list with the final `nil` replaced with a fresh unbound variable.
- Return a tuple consisting of the modified list and the unbound variable.

Difference Lists *contd*

We can use difference lists to implement efficient queues, with both `Enqueue` and `Dequeue` $O(1)$ (constant time) in performance.

Example (Difference list-based, efficient queues)

`code/queue-dl.oz`

```
fun {Queue}
   Unbound in
   Unbound#Unbound#0 end
fun {Enqueue Start#End#Length Item}
   NewEnd in
   End = Item|NewEnd
   Start#NewEnd#(Length+1) end
fun {Dequeue Start#End#Length ?Item}
   case Start of Head|Tail then
      Item = Head
      Tail#End#(Length-1) end end
```


Recursion and Iteration with Trees *contd*

Back to trees: we can now provide an efficient solution to the breadth-first tree numbering problem.

Example (Breadth-first tree numbering)

```
fun {BFN Tree}
   proc {Iterate In Out Index}
      case [In Out] ... end[18]
   end
   InEnd OutEnd Result in
   {Iterate (Tree|InEnd)#InEnd#1 (Result|OutEnd)#OutEnd 1}
   Result
end
```

To breadth-first re-label a tree, operate simultaneously on two queues:

- `In`, the original (sub)trees to be relabelled;
- `Out`, fresh variables to be bound to relabelled (sub)trees.

[18] It would be more efficient to combine `In` and `Out` into a tuple, `In#Out`, but for the convenience of syntactic clarity we will use the equally valid list notation, `[In Out]`.

Recursion and Iteration with Trees *contd*

How does `BFN` work?

- Initially, the input queue, `In`, contains just the whole original tree (we need to explicitly encode the queue's length):

```
InEnd ... in
{Iterate (Tree|InEnd)#InEnd#1 ...}
```

- Initially, the output queue, `Out`, contains just one unbound variable, for the whole re-labelled tree (we don't care about the queue's length, it's always the same length as `In`):

```
... OutEnd Result in
{Iterate ... (Result|OutEnd)#OutEnd ...}
```

- Initially, the counter variable, `Index`, used to label tree nodes equals 1:

```
{Iterate ... 1}
```


Recursion and Iteration with Trees *contd*

`Iterate` performs pattern matching over both queues simultaneously.

- If `In` is empty, there is nothing to do, the result is ready:

```
case [In Out] of [_#_#0 _] then skip
```

- If the first element in `In` is a leaf node, bind the first element of `Out` to a leaf node labelled with the current index, and proceed with the rest of both queues:

```
[] [(leaf(_)|InTail)#InEnd#Length
    (OutHead|OutTail)#OutEnd] then
   OutHead = leaf(Index)
   {Iterate InTail#InEnd#(Length-1) OutTail#OutEnd Index+1}
```

Recursion and Iteration with Trees *contd*

`Iterate` performs pattern matching over both queues simultaneously *contd*.

- If the first element in `In` is a tree node,
  - bind the first element of `Out` to a tree node labelled with the current index and with two ubound variables as branches,
  - enqueue the branches of the original tree onto `In`,
  - enqueue the two ubound variables onto `Out`, and proceed:

```
[] [(tree(_ Left Right)|InTail)#InEnd#Length
    (OutHead|OutTail)#OutEnd] then
   InEndNew LeftNew RightNew OutEndNew in
   InEnd = Left|Right|InEndNew
   OutHead = tree(Index LeftNew RightNew)
   OutEnd = LeftNew|RightNew|OutEndNew
   {Iterate
    InTail#InEndNew#(Length-1+2)
    OutTail#OutEndNew
    Index+1}
```

Note: `Length` is increased by 1, since we dequeue one item (-1) and then enqueue two items (+2).

Recursion and Iteration with Trees *contd*

Given the input tree:

```
tree(root tree(left leaf(hello) leaf(dolly))
          tree(right leaf(funny) leaf(bunny)))
```

`BFN` proceeds as follows:

1. `In` `tree(root tree(left ...) tree(right ...))`
   `Out` ν_1
   `Result` ν_1
2. `In` `tree(left ...)`, `tree(right ...)`
   `Out` ν_2, ν_3
   `Result` `tree(1` ν_2 ν_3`)`
3. `In` `tree(right ...)`, `leaf(hello)`, `leaf(dolly)`
   `Out` ν_3, ν_4, ν_5
   `Result` `tree(1 tree(2` ν_4 ν_5`)` ν_3`)`
4. `In` `leaf(hello)`, `leaf(dolly)`, `leaf(funny)`, `leaf(bunny)`
   `Out` $\nu_4, \nu_5, \nu_6, \nu_7$
   `Result` `tree(1 tree(2` ν_4 ν_5`) tree(3` ν_6 ν_7`))`

Recursion and Iteration with Trees *contd*

Given the input tree:

```
tree(root tree(left leaf(hello) leaf(dolly))
          tree(right leaf(funny) leaf(bunny)))
```

`BFN` proceeds as follows *contd*:

5. `In` `leaf(dolly)`, `leaf(funny)`, `leaf(bunny)`
   `Out` ν_5, ν_6, ν_7
   `Result` `tree(1 tree(2 leaf(4)` ν_5`) tree(3` ν_6 ν_7`))`
6. `In` `leaf(funny)`, `leaf(bunny)`
   `Out` ν_6, ν_7
   `Result` `tree(1 tree(2 leaf(4) leaf(5)) tree(3` ν_6 ν_7`))`
7. `In` `leaf(bunny)`
   `Out` ν_7
   `Result` `tree(1 tree(2 leaf(4) leaf(5)) tree(3 leaf(6)` ν_7`))`
8. `In`
   `Out`
   `Result` `tree(1 tree(2 leaf(4) leaf(5)) tree(3 leaf(6) leaf(7)))`

Recursion and Iteration with Trees *contd*

Try it!

- Open the Mozart IDE and load the file `code/bfn-test.oz`.
- Start the debugger (C-. C-. d).
- Execute the program (C-. C-b).
- In the debugger, step through the program (type `s`) and observe the stack and the content of global and local variables.

Read the Mozart documentation if necessary.[19]

[19] See http://www.mozart-oz.org/documentation/ozcar/ .

Lecture Outline

Declarative Programming
Declarative Programming with XML-Related Technologies
Recursion and Iteration with Lists
Recursion and Iteration with Trees
Difference Lists
Recursion and Iteration with Trees *contd*

Summary

Summary

This time
- Declarative programming.
- Recursion and iteration with lists.
- Difference lists, efficient queues.
- Traversing trees.

Next time
- Higher-order programming.

Summary *contd*

Homework
- Examine and try out today's code, read Mozart/Oz documentation if necessary.
- Read Ch. 3 in CTMCP (Secc. 3.1-3.4, optionally 3.5).

Pensum
- All of today's slides, except for non-Oz code (XML etc.) and details of the tree-numbering algorithms (but you need to be able to explain how the corresponding code works).

Further reading
- Read the article *Breadth-First Numbering:. Lessons from a Small Exercise in Algorithm Design* by Chris Okasaki.[20]

[20] See http://www.eecs.usma.edu/webs/people/okasaki/icfp00.ps .