Reading Suggestions

- Chapter 3
  - Sections 3.1 – 3.4 [careful]
    - but skip or browse [3.4.4, 3.4.7, 3.4.8]
  - Sections 3.6, 3.7 [careful]

- And of course the handouts!
Organizational

- Midterm exam: next lecture (8)

Making Computations

Iterative

Accumulators
This Lecture

- Making computations iterative
- Higher-order programming
- Abstract data types

Using Accumulators

- Accumulator stores intermediate result

- Finding an accumulator amounts to finding a state invariant
  - recursion maintains state invariant
  - state invariant must hold initially
  - result must be obtainable from state invariant
Summary So Far

- Use accumulators
  - yields iterative computation
  - find state invariant
- Exploit structural properties
  - for example: \( x^{2n} = (xx)^n \)
- Exploit both kinds of knowledge
  - on how programs execute (abstract machine)
  - on application/problem domain

More on Accumulators

- Accumulators on numbers, fine…
- How about lists?
- List type is a form of recursive data type
- How about recursive data types?
Type Notation

- Type information simplifies program development
- Many functions are designed based on the type of the input arguments
- Let us have some type notation
- The list type is subtype of the record type
- Other useful types are trees, etc

Type of List

- Based on the type hierarchy
  - ⟨Value⟩, ⟨Record⟩, ...
  - ⟨Record⟩ ⊂ ⟨Value⟩
    - The Record type is a subtype of the Value type

- List is either nil or X | Xr where Xr is a list where X is an arbitrary value

- ⟨List⟩ ::= nil | ⟨Value⟩ | '⟨List⟩
On Types

- A list whose elements are of given type

\[
\langle\text{List } T\rangle ::= \text{nil} \mid \langle T \rangle \mid \langle\text{List } T\rangle
\]

- \( T \) is a type variable
- \( \langle\text{List } T\rangle \) is a type function
- \( \langle\text{List } \langle\text{Int}\rangle\rangle \) : a list whose elements are integers
- \( \langle\text{List } \langle\text{Value}\rangle\rangle \) is equal to \( \langle\text{List}\rangle \)

On Types (trees)

- Binary trees

\[
\langle\text{BTree } T\rangle ::= \langle\text{leaf}\rangle \\
\text{tree}\langle\text{key}:\langle\text{Literal}\rangle, \text{value}:T, \text{left}:\langle\text{BTree } T\rangle, \text{right}:\langle\text{BTree } T\rangle\rangle
\]

- Binary tree representing a dictionary mapping keys to values
- Binary tree is either a leaf (atom leaf), or
- An internal node with label tree, with left and right subtrees, a key and a value
- The key is of literal type and the value of type \( T \)
On Types: procedures and functions

- The type of a procedure is
  \[ \langle \text{proc} \{ T_1 \ldots T_n \} \rangle \]
  - \( T_1 \ldots T_n \) are the types of the arguments

- The type of a function is
  \[ \langle \text{fun} \{ T_1 \ldots T_n \} : T \rangle \]
  - \( T_1 \ldots T_n \) are the types of the arguments, and \( T \) is the type of the result
  - \[ \langle \text{fun} \{ \langle \text{List} \rangle \langle \text{List} \rangle \} : \langle \text{List} \rangle \rangle \] is a function that takes two lists and returns a list
On Types: procedures and functions

- The type of a function is

\[ \langle \text{fun} \{ T_1 \ldots T_n \} : T \rangle \]

- \( T_1 \ldots T_n \) are the types of the arguments, and \( T \) is the type of the result

- \( \langle \text{fun} \{ \text{Append} \langle \text{List} \rangle \langle \text{List} \rangle \} : \langle \text{List} \rangle \rangle \) is the type of \( \text{Append} \)

Back to Recursive and Iterative Computation

- We will use type definition to construct programs
- Reversing a list
- Type of \( \text{Reverse} \) is \( \langle \text{fun} \{ \langle \langle \text{List} \rangle \rangle \} : \langle \text{List} \rangle \rangle \)
- How to reverse the elements of a list

\[ \{ \text{Reverse} \ [a \ b \ c \ d] \} \]

returns

\[ [d \ c \ b \ a] \]
Reversing a List

- How to reverse the elements of a list
- Reverse of nil is nil
- Reverse of \( X|Xs \) is \( Z \), where reverse of \( Xs \) is \( Ys \), and append \( Ys \) and \([X]\) to get \( Z \)

Question

- What is correct

\[
\{\text{Append } \{\text{Reverse } Xr\} \ X}\}
\]

or

\[
\{\text{Append } \{\text{Reverse } Xr\} \ [X]\}\}
\]
Naïve Reverse Function

```lisp
fun {NRev Xs}
    case Xs
        of nil then nil
        [] X|Xr then
            {Append {NRev Xr} [X]}
        end
    end
end
```

Question

- What is the problem with naïve reverse?
- Possible answers
  - not tail recursive
  - and also Append is costly
Cost of Naïve Reverse

- Suppose a recursive call \{\text{NR ev } Xs\}
  - where \{\text{Length } Xs\}=n
  - assume cost of \{\text{NR ev } Xs\} is \(c(n)\)
    number of function calls
  - then \(c(0) = 0\)
  - \(c(n) = c(\{\text{Append } \{\text{NR ev } Xr\} \ldots\}) + c(n-1)\)
    \(= n + c(n-1)\)
    \(= n + (n-1) + c(n-2)\)

  - this yields: \(c(n) = \frac{n^2(n+1)}{2}\)
- For list of length \(n\), \text{NR ev} uses approx. \(n^2\) calls!

Doing Better for Reverse

- Use an accumulator, of course
  - technique: already reversed part of list

- How does \text{Reverse} compute?

- Some abbreviations
  - \{\text{IR Xs}\} for \{\text{IterRev Xs}\}
  - \text{Xs ++ Ys} for \{\text{Append Xs Ys}\}
Computing $\text{NRev}$

\[
\begin{align*}
\{\text{NRev} \ [a \ b]\} &=  \\
\{\text{NRev} \ [b]\} \ ++ \ [a] &=  \\
(\{\text{NRev} \ \text{nil}\} \ ++ \ [b]) \ ++ \ [a] &=  \\
(\text{nil} \ ++ \ [b]) \ ++ \ [a] &=  \\
[b] \ ++ \ [a] &=  \\
[b \ a] &=
\end{align*}
\]

IterRev (IR)

State Transformation

<table>
<thead>
<tr>
<th>$Xs$</th>
<th>$Rs$</th>
<th>$\Rightarrow$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[a b c nil]</td>
<td>[b c] a</td>
<td>nil</td>
</tr>
<tr>
<td>[b c] a</td>
<td>nil</td>
<td></td>
</tr>
<tr>
<td>[c] b</td>
<td>a</td>
<td>nil</td>
</tr>
<tr>
<td>nil c</td>
<td>b</td>
<td>a</td>
</tr>
</tbody>
</table>
IterRev (IR)
State Transformation

\[
\begin{array}{c|c}
Xs & Rs \\
\hline
[a \ b \ c] & \text{nil} \\
\hline
[b \ c] & a|\text{nil} \\
\hline
[c] & b|a|\text{nil} \\
\hline
\text{nil} & c|b|a|\text{nil} = [c \ b \ a]
\end{array}
\]

- The general pattern:
  \[
  \{\text{IR } X|Xr \ RS\} \Rightarrow \{\text{IR } Xr \ X|RS\}
  \]

Using an Accumulator for IterRev (IR)

\[
\begin{align*}
\{\text{IR } [a \ b]\} & ++ \text{nil} = \\
\{\text{IR } [b]\} & ++ ([a] ++ \text{nil}) = \\
\{\text{IR } [b]\} & ++ ([a]) = \\
\{\text{IR } \text{nil}\} & ++ ([b] ++ [a]) = \\
\{\text{IR } \text{nil}\} & ++ ([b \ a]) = \\
\text{nil} & ++ ([b \ a]) = \\
[b \ a]
\end{align*}
\]
Reverse Intermediate Step

fun \{\text{IterRev} \ Xs \ Ys\}
\begin{align*}
\text{case } \ Xs \\
\text{of } \text{nil} & \text{ then } \ Ys \\
[ ] \ X|\ Xr & \text{ then} \\
\{\text{IterRev} \ Xr \ X|\ Ys\} & \text{ end} \\
\text{end} \\
\end{align*}

\text{Is tail recursive now}

State Invariant for Reverse

\begin{itemize}
\item Now it is easy to see that
\begin{align*}
\{\text{Reverse } [X_1 \ldots X_n]\} \\
&= \\
\{\text{Reverse } [X_{i+1} \ldots X_n]\} & + [X_i \ldots X_1]
\end{align*}
\text{as we go along}
\end{itemize}
Reverse Proper

local
  fun {IterRev Xs Ys}
  case Xs
    of nil then Ys
      [] X|Xr then {IterRev Xr X|YS}
    end
  end
in
  fun {Rev Xs} {IterRev Xs nil} end
end

Summary

- Accumulators for lists work as they do for integers
- Think of the problem as state transformation
- Essential: finding state invariant
  - find it by some hand computation
Constructing Programs by Following the Type

- Observe, programs so far that takes lists, has a form that corresponds to the list type

- \( \langle \text{List T} \rangle ::= \text{nil} \mid \langle \text{T} \rangle | \langle \text{List T} \rangle \)

- case Xs
  of _nil_ then \langle expr \rangle % base case
  [ ] \times|\times r then \langle expr \rangle % recursive call
  end

This helps us when the type gets complicated

- Nested lists are lists whose elements can be lists

- Example Xs: [[[1 2] 4 nil [[5] 10]]

- Find the number of elements in a nested list

  \{\text{Length Xs}\} = 5
Constructing Programs by Following the Type

- Nested lists

\[
\langle \text{NList T} \rangle ::= \text{nil} \\
| \langle \text{NList T} \rangle | \langle \text{NList T} \rangle' \\
| \text{T } | \langle \text{NList T} \rangle \quad (T \text{ is not nil nor a cons)}
\]

- `case xs of nil then \langle \text{expr} \rangle \% base case \\
[\text{[]}] \text{X|XR and then } \{\text{IsList X}\} \text{ then } \\
\langle \text{expr} \rangle \% \text{recursive calls for x and xr} \\
[\text{[]}] \text{X|XR then } \\
\langle \text{expr} \rangle \% \text{recursive call for xr} \\
end`
Constructing Programs by Following the Type

- Nested lists: \( \langle \text{fun } \{ \text{Length } \langle \text{NList } T \rangle ; \langle \text{Int} \rangle \} \rangle \)

- \text{fun } \{ \text{Length } Xs \} \text{ case } Xs \text{ of } \text{nil } \text{ then } 0 \ % \text{ base case} \text{ [] } X|Xr \text{ andthen } \{ \text{IsList } X \} \text{ then} \text{ \{Length } X \} + \text{ \{Length } Xr \} \text{ [] } X|Xr \text{ then} \text{ 1+\{Length } Xr \} \text{ end} \text{ end}
Higher-Order Programming

- Higher-order programming = the set of programming techniques that are possible with procedure values (lexically-scoped closures)
- higher-order programming is the foundation of secure data abstraction component-based programming and object-oriented programming

Higher-Order Programming

- Basic operations
  - Procedural abstraction: the ability to convert any statement into a procedure value
  - Genericity: the ability to pass procedure values as arguments to a procedure call
  - Instantiation: the ability to return procedure values as results from a procedure call
  - Embedding: the ability to put procedure values in data structures
Higher-order programming

- Control abstractions
  - The ability to define control constructs
  - Integer and list loops, accumulator loops, folding a list (left and right)

Procedural Abstraction

- Procedural abstraction is the ability to convert any statement into a procedure value

\[
P = \text{proc}\.\langle\text{Statement}\rangle\end\text{proc}
\]

Normal Execution: time

Delayed Execution: time

\{P\}
Procedural Abstraction

- A procedure value is usually called a closure, or more precisely, a lexically-scoped closure
  - A procedure value is a pair: it combines the procedure code with the contextual environment

- Basic scheme:
  - Consider any statement <s>
  - Convert it into a procedure value:
    \[
    P = \text{proc } \{ \$ \} \ <s> \ \text{end}
    \]
  - Executing \{P\} has exactly the same effect as executing <s>

The same holds for expressions

- A procedure value is usually called a closure, or more precisely, a lexically-scoped closure
  - A procedure value is a pair: it combines the procedure code with the contextual environment

- Basic scheme:
  - Consider any expression <E>
  - Convert it into a function value:
    \[
    F = \text{fun } \{ \$ \} \ <E> \ \text{end}
    \]
  - Executing \(X = \{F\}\) has exactly the same effect as executing \(X = E\)
Example

Suppose we want to define the operator andthen (&& in Java) as a function:

```latex
fun {AndThen B1 B2}
    if B1 then B2 else false end
end

If {AndThen X>0 Y>0} then ... else ...
```

Example

Does not work because both X>0 and Y>0 are evaluated

```latex
fun {AndThen B1 B2}
    if B1 then B2 else false end
end

If {AndThen X>0 Y>0} then ... else ...
```
Example: use procedural abstractions

```plaintext
fun {AndThen B1 B2}
    if {B1} then {B2} else false end
end

If {AndThen
    fun{$} X>0 end
    fun{$} Y>0 end
} then ... else ... end
```

Genericity

- To make a function generic is to let any specific entity (i.e., any operation or value) in the function body become an argument of the function.
- The entity is abstracted out of the function body.
Genericity

- Replace specific entities (zero 0 and addition +) by function arguments

```
fun {SumList ls}
    case ls
    of nil then 0
       []x|lr then x+{SumList lr}
    end
end
```

```
fun {FoldR l f u}
    case l
    of nil then u
       []x|l2 then {f x {FoldR l2 f u}}
    end
end
```
Genericity SumList

fun \{SumList Ls\}
{FoldR L
  fun \{\$ X Y\} X+Y end 0\}
end

Genericity ProductList

fun \{ProductList Ls\}
{FoldR L
  fun \{\$ X Y\} X*Y end 1\}
end
Genericity Some

fun {Some Ls}
  {FoldRL fun {$ X Y} X orelse Y end false}
end

List Mapping

- Mapping
  - each element recursively
  - calling function for each element
  - Snoctruct list that takes output

- Separate function calling by passing function as argument
Other generic functions: Map

```latex
fun \{\text{Map } Xs \ F\}
    case Xs
    of \text{nil} \text{ then } \text{nil} \[
        \text{X} | \text{Xr} \text{ then }
            \{F \ X\} | \{\text{Map } \text{Xr} \ F\}
    \text{end}
end

\{\text{Browse } \{\text{Map } [1 \ 2 \ 3] \}
    \text{fun } \{$ \ X\} \ X^2 \text{ end}\}\}
```

Other generic functions: Filter

```latex
fun \{\text{Filter } Xs \ P\}
    case Xs
    of \text{nil} \text{ then } \text{nil} \[
        \text{X} | \text{Xr} \text{ andthen } \{P \ X\} \text{ then }
            \text{X} | \{\text{Filter } \text{Xr} \ P\}
        \text{X} | \text{Xr} \text{ then } \{\text{Filter } \text{Xr} \ P\}
    \text{end}
End

\{\text{Browse } \{\text{Filter } [1 \ 2 \ 3] \ IsOdd\}\}\}
```
Instantiation

- **Instantiation**: the ability to return procedure values as results from a procedure call
- A factory of specialized functions

```plaintext
fun {Add X}
  fun {Y} X+Y end
end

Inc = {Add 1}
{Browse {Inc 5}} % shows 6
```

Instantiation: An application (protected values)

- Given a value (e.g. [1 2 3]) we would like to seal it for any other client except for a person that has a key
- A procedure that has the key can access the list
- No other procedures can access the value
Instantiation: An application (protected values)

New basic data type is names
A name is unforgable atom:
• Cannot be displayed
• The only possible operation is comparison

\[ X = \{\text{NewName}\} \]
\[ Y = \{\text{NewName}\} \]
• \( X \) and \( Y \) are different
Wrappers

- proc {NewWrapper Wrap Unwrap}
  Key={NewName}
  in
  fun {Wrap X} ... end
  fun {Unwrap W} {W Key} end
end

? Indicate that Wrap and UnWrap is output arguments (just a comment)
Wrappers

- declare W UW
  {NewWrapper W UW}
  RL = {W [1 2 3]}
  {Browse RL} % cannot see
  {Browse {UW RL}} % shows [1 2 3]

- NewWrapper is a factory that creates two related procedures

Instantiation: An application (protected values)

{wrap [1 2 3]} → [1 2 3]

{Unwrap [1 2 3]} → [1 2 3]
Wrappers

- proc {NewWrapper ?Wrap ?Unwrap}
  Key={NewName}
  in
  fun {Wrap X}
    fun{$ K} if K==Key then X end
    end
  fun {Unwrap W} {W Key} end
  end

Instantiation: An application (protected values)

\[ [1 \ 2 \ 3] \text{ wrap} \rightarrow [1 \ 2 \ 3] \]

\[ [1 \ 2 \ 3] \text{ unwrap} \rightarrow [1 \ 2 \ 3] \]
Embedding

- Embedding is when procedure values are put in data structures.
- Embedding has many uses:
  - Modules: a module is a record that groups together a set of related operations.
  - Software components: a software component is a generic function that takes a set of modules as its arguments and returns a new module. It can be seen as specifying a module in terms of the modules it needs.

Control Abstractions

```plaintext
proc {For I J P}
   if I > J then skip
   else {P I} {For I+1 J P}
   end
end

{For 1 10 Browse}

for I in 1..10 do {Browse I} end
```
Control Abstractions

proc \{ForAll Xs P\}
    case Xs
    of nil then skip
    [] X|Xr then
        {P X} \{ForAll Xr P\}
    end
end
End

\{ForAll [a b c d]\}
proc\{I\}
    \{System.showInfo "the item is: " # I\}
end

for I in [a b c d] do
    \{System.showInfo "the item is: " # I\}
end
List Mapping

- Mapping
  - each element recursively
  - *calling function for each element*
  - Snooject list that takes output

- Separate function calling by passing function as argument

Other Examples

- \{Filter Xs F\}
  returns all elements of \(Xs\) for which \(F\) returns true

- \{Some Xs F\}
  tests whether \(Xs\) has an element for which \(F\) returns true

- \{All Xs F\}
  tests whether \(F\) returns true for all elements of \(Xs\)
Folding Lists

- Consider computing the sum of list elements
  - ...or the product
  - ...or all elements appended to a list
  - ...or the maximum
  - ...

- What do they have in common?

- Snocider example: `SumList`

```haskell
fun {SumList Xs}
  case Xs
  of nil then 0
  [] X | Xr then {SumList Xr} + X
  end end

- First step: make tail-recursive with accumulator
**SumList: Tail-Recursive**

fun {SumList Xs N}
    case Xs
    of nil  then N
        [] X|Xr then {SumList Xr N+X}
    end
end
{SumList Xs 0}

- Question:
  - what is about computing the sum?
  - what is generic?
How Does \texttt{SumList} Compute?

\begin{align*}
\{\texttt{SumList} \ [2 \ 5 \ 7] \ 0\} & = \\
\{\texttt{SumList} \ [5 \ 7] \ 0+2\} & = \\
\{\texttt{SumList} \ [7] \ (0+2)+5\} & = \\
\{\texttt{SumList} \ \texttt{nil} \ ((0+2)+5)+7\} & = \ldots
\end{align*}

\texttt{SumList} Slightly Rewritten...

\begin{align*}
\{\texttt{SumList} \ [2 \ 5 \ 7] \ 0\} & = \\
\{\texttt{SumList} \ [5 \ 7] \ \{F \ 0 \ 2\}\} & = \\
\{\texttt{SumList} \ [7] \ \{F \ \{F \ 0 \ 2\} \ 5\}\} & = \\
\{\texttt{SumList} \ \texttt{nil} \ \{F \ \{F \ \{F \ 0 \ 2\} \ 5\} \ 7\}\} & = \\
\ldots
\end{align*}

with

\begin{align*}
\textbf{fun} \ \{F \ X \ Y\} \ X+Y \ \textbf{end}
\end{align*}
Left-Folding

- Two values define “folding”
  - initial value 0 for SumList
  - binary function + for SumList

- Left-folding \{FoldL [x_1 \ldots x_n] F S\}

  \{F \ldots \{F \{F S x_1\} x_2\} \ldots x_n\}

  or

  \((\ldots ((S \otimes_F x_1) \otimes_F x_2) \ldots \otimes_F x_n)\)
**FoldL**

```ocaml
fun {FoldL Xs F S}
    case Xs
    of nil then S
        [] X|Xr then {FoldL Xr F {F S X}}
    end
end
```

**SumList With FoldL**

```ocaml
local
    fun {Plus X Y} X+Y end
in
    fun {SumList Xs}
        {FoldL Xs Plus 0}
    end
end
```
Properties of **FoldL**

- Tail recursive
- First element of list if folded first...
  - what does that mean?

---

**What Does This Do?**

```cpp
local
  fun {Snoc Xr X}
    X | Xr
  end
in
  fun {Foo Xs}
    {FoldL Xs Snoc nil}
  end
end
```
How Does \texttt{Foo} Compute?

\[
\{\texttt{Foo} [a \ b \ c]\} = \\
\{\texttt{FoldL} [a \ b \ c] \ \text{Snoc} \ \text{nil}\} = \\
\{\texttt{FoldL} [b \ c] \ \text{Snoc} \ \{\text{Snoc} \ \text{nil} \ a\}\} = \\
\{\texttt{FoldL} [b \ c] \ \text{Snoc} \ [a]\} = \\
\{\texttt{FoldL} [c] \ \text{Snoc} \ \{\text{Snoc} \ [a] \ b\}\} = \\
\{\texttt{FoldL} [c] \ \text{Snoc} \ [b \ a]\} = \\
\{\texttt{FoldL} \ \text{nil} \ \text{Snoc} \ \{\text{Snoc} \ [b \ a] \ c\}\} = \\
\{\texttt{FoldL} \ \text{nil} \ \text{Snoc} \ [c \ b \ a]\} = [c \ b \ a]
\]

Right-Folding

- Two values define “folding”
  - initial value
  - binary function

- Right-folding \(\{\text{FoldR} \ [x_1 \ldots x_n] \ F \ S\}\)
  \[
  \{F \ x_1 \ \{F \ x_2 \ \ldots \ \{F \ x_n \ S\} \ \ldots\}\}
  \]
  or
  \[
  x_1 \otimes_F (x_2 \otimes_F ( \ldots (x_n \otimes_F S) \ldots ))
  \]
Right-Folding

- Two values define “folding”
  - initial value
  - binary function

- Right-folding \( \text{FoldR} [x_1 \ldots x_n] F S \)

or

\[ x_1 \otimes_F (x_2 \otimes_F ( \ldots (x_n \otimes_F S) \ldots )) \]

---

FoldR

```verbatim
fun \{FoldR Xs F S\}
  case Xs
  of nil    then S
[] X | Xr then \{F X \{FoldR Xr F S\}\}
end
end
```
Properties of FoldR

- Not tail-recursive
- Elements folded in order

FoldL or FoldR?

- FoldL and FoldR compute same value, if function $F$ commutes:
  \[ \{F \, X \, Y\} == \{F \, Y \, X\} \]

- If function commutes: FoldL
  - FoldL tail-recursive

- Otherwise: FoldL or FoldR
  - depending on required order of result
Example: Appending Lists

- Given: list of lists
  
  \[
  [[a\ b]\ [1\ 2]\ [e]\ [g]]
  \]

- Task: compute all elements in one list in order

- Solution:
  
  \[
  \textit{fun}\ \{\text{AppAll}\ Xs\}
  
  \{\text{FoldR}\ Xs\ \text{Append}\ \text{nil}\}
  
  \textit{end}
  \]

- Question: What would happen with \texttt{FoldL}?

Tuples and Records...

- Techniques for lists explored here of course applicable...
  
  \[
  \text{...see tutorial}\n  \]
Summary

- Many operations can be partitioned into
  - pattern implementing
    - recursion
    - application of operations
  - operations to be applied

- Typical patterns
  - Map: mapping elements
  - FoldL/FoldR: folding elements
  - Filter: filtering elements
  - Sort: sorting elements
  - ...

Abstract Data Types
Data Types

- Data type
  - set of values
  - operations on these values

- Primitive data types
  - records
  - numbers
  - ...

- Abstract data types
  - completely defined by its operations (interface)
  - implementation can be changed without changing use

Example: Lab Assignment 2

- Abstract data type for Huffman-trees

- Different implementations

- Same interface
**Motivation**

- Sufficient to understand interface only

- Software components can be developed independently
  - as long as only interface is used

- Developers need not to know implementation details

**Outlook**

- How to define abstract data types

- How to organize abstract datatypes

- How to use abstract datatypes
Example: Dictionaries

- Designing the interface
  - \{\text{MakeDict}\}
    - returns new dictionary
  - \{\text{DictMember } D \ F}\}
    - tests whether feature \(F\) is member of dictionary \(D\)
  - \{\text{DictAccess } D \ F\}
    - return value of feature \(F\) in \(D\)
  - \{\text{DictAdjoin } D \ F \ X\}
    - return dictionary with value \(X\) at feature \(F\) adjoined

- Interface depends on purpose, could be richer (for example, \text{DictCondSelect})

Using the \text{Dict} ADT

- Now we can write programs using the ADT without even having an implementation for it
- Implementation can be provided later
- Eases software development in large teams
Implementing the Dict ADT

- Now we can decide on a possible implementation for the Dict ADT
  - based on pairlists
  - based on records

- Regardless on what we decide, programs using the ADT will work!
  - the interface is a contract between use and implementation

Dict: Pairlists

fun {MakeDict}
  nil
end
fun {DictMember D F}
  case D
    of nil then false
    [] G#X|Dr then
      if G==F then true
      else {DictMember Dr F}
    end
  end
end
...
Dict: Records

fun (MakeDict)
  mt
end

fun (DictMember D F)
  {HasFeature D F}
end

fun (DictAccess D F)
  D.F
end

fun (DictAdjoin D F X)
  {AdjoinAt D F X}
end

Example Frequency Counting

local

fun (Inc D X)
  if {DictMember D X} then
      {DictAdjoin D X (DictAccess D X)+1}
  else
      {DictAdjoin D X 1}
  end
end

in

fun (Cnt Xs)
  % returns dictionary
  {FoldL Xs Inc (MakeDict)}
end
end
Example Frequency Counting

```haskell
local
  fun {Inc D X}
    if {DictMember D X}
      {DictAdjoin D X {DictAccess D X}+1}
    else {DictAdjoin D X 1}
  end
end

fun {Cnt Xs}
  % returns dictionary
  {FoldL Xs Inc {MakeDict}}
end
end
```

homework: understand and try this example!

Evolution of ADTs

- Important aspect of developing ADTs
  - start with simple (possibly inefficient) implementation
  - refine to better (more efficient) implementation
  - refine to carefully chosen implementation
    - hash table
    - search tree

- All of evolution is local to ADT
  - no change of programs using ADT is needed!
Next Lecture

- Midterm exam
- More on abstract data types
- Software components, modules, functors

See You Next Week!
Have a Nice Weekend