Real World Programming requires I/O

- Slide title says it all, really . . .
- I/O has been problematical for (pure) functional languages
- In order to understand why I/O in Haskell is the way it is . . .
  - we need to know what is meant by “pure”
  - we need to know how Haskell is implemented (a little)
  - we need to understand the key problem with I/O
- Summing up, we first need to understand how functional languages work.

Functional Languages as Rewrite Systems

- We can view function/value definitions as rules describing how to transform (rewrite) an expression.
  - If we have a definition like:
    
    ```
    myfun this_pattern = result_expression
    ```
  - We then invoke the function in a call matching the above pattern
    ```
    myfun some_argument
    ```
  - We expect to see the call replaced by the result, with appropriate substitutions:
    ```
    result_expression [ some_argument / this_pattern ]
    ```
  - This is formalised in the so-called “Lambda-calculus”.

Definitions in Haskell

- One way\(^1\) to define a function called \texttt{myfun} (say) is as a series of \textit{declarations} of the form:
  ```
  myfun \texttt{pat\textsubscript{11} \texttt{pat\textsubscript{12}} ... \texttt{pat\textsubscript{1k}}} = \texttt{exp\textsubscript{1}}
  myfun \texttt{pat\textsubscript{21} \texttt{pat\textsubscript{22}} ... \texttt{pat\textsubscript{2k}}} = \texttt{exp\textsubscript{2}}
  ...
  myfun \texttt{pat\textsubscript{n1} \texttt{pat\textsubscript{n2}} ... \texttt{pat\textsubscript{nk}}} = \texttt{exp\textsubscript{n}}
  ```
  where each line has the same number of patterns (pat)
- Each pattern can be:
  - a constant value (number, character, string)
  - a variable (no variable can occur more than once in a declaration)
  - an expression built from type-constructors and patterns

\(^1\)There are many others — to be covered later!
Pattern Examples

- Expect three arbitrary arguments
  \[ \text{myfun \, x \, y \, z} \]
- Illegal — if we want first two arguments to be the same then we need to use a conditional (somehow).
  \[ \text{myfun \, x \, x \, z} \]
- First argument must be zero, second is arbitrary, and third is a non-empty list.
  \[ \text{myfun \, 0 \, y \, (z:zs)} \]
- First argument must be zero, second is arbitrary, and third is a non-empty list, whose first element is character 'c'
  \[ \text{myfun \, 0 \, y \, ('c':zs)} \]
- First argument must be zero, second is arbitrary, and third is a non-empty list, whose tail is a singleton.
  \[ \text{myfun \, 0 \, y \, (z:[z'])} \]

Pattern Matching

We describe how pattern matching works and what it does by example:

- A constant pattern matches the specified value only
  - pattern 3 only matches the value 3.
- A variable matches anything, and we get a binding of that variable to the value matched.
  - Pattern \( x \) matches any value \( v \), and the result is that variable \( x \) is bound to value \( v \).
- A constructor pattern matches something of the same “shape” as well as matching the corresponding sub-components.
  - Pattern \( x:xs \) matches a non-empty list, and binds \( x \) to the head value and \( xs \) to the tail value of the list.

Pattern Matching (summary)

- Pattern-matching can succeed or fail.
- If successful, a pattern match returns a (possibly empty) binding.
- A binding is a mapping from (pattern) variables to values.
- Examples:

<table>
<thead>
<tr>
<th>Patterns</th>
<th>Values</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x ) (y:ys) ( 3 )</td>
<td>99 ( [,] ) ( 3 )</td>
<td>Fail</td>
</tr>
<tr>
<td>( x ) (y:ys) ( 3 )</td>
<td>99 ( [1,2,3] ) ( 3 )</td>
<td>Ok, ( x \mapsto 99, , y \mapsto 1, , ys \mapsto [2,3] )</td>
</tr>
<tr>
<td>( x ) (1:ys) ( 3 )</td>
<td>99 ( [1,2,3] ) ( 3 )</td>
<td>Ok, ( x \mapsto 99, , ys \mapsto [2,3] )</td>
</tr>
</tbody>
</table>

| Binding \( x \mapsto 99, \, y \mapsto 1, \, ys \mapsto [2,3] \) can also be written as \( [99,1, [2,3]]/x, y, ys \) |

Definition Rewriting (a.k.a. Reduction)

- Given application
  \[ \text{myfun \, arg1 \, arg2 \, \ldots \, argk} \]
  we attempt to match against the patterns seen earlier, in order.
- If all matches fail we get a runtime error
- Otherwise, assume we succeeded in matching the following
  \[ \text{myfun \, pat_{1/1} \, pat_{1/2} \, \ldots \, pat_{1/k} = \, exp_i} \]
  with binding \( \rho \).
- The outcome is the rhs expression, with substitutions done according to the bindings.
  \[ (\text{exp_i})_{\rho} \]

This is used to overwrite the original application.
Haskell Execution (Rewriting version)

- Haskell execution proceeds by reducing function applications until this is no longer possible.
- An expression with no reducible applications is said to be in normal form.
- Generally the normal form (a value) is taken as the result/meaning of the program.
- Some (hard) theorems show that normal forms (if they exist) are unique. (Church-Rosser Theorems I & II)

Haskell Execution Example

- Assume definitions
  
  \[ \text{sum \ } \emptyset = 0 \]
  
  \[ \text{sum \ } (x:xs) = x + \text{sum} \ \text{xs} \]

- We want to execute the following program:
  
  \[ \text{sum \ } (3:2:1:[]) \]

- Reduction 1
  
  - Matching \( \text{sum \ } (3:2:1:[]) \) against pattern \( \text{sum} \ \emptyset \) fails.
  - Matching \( \text{sum} \ (3:2:1:[]) \) against pattern \( \text{sum} \ (x:xs) \) succeeds
    with binding \( x \mapsto 3, xs \mapsto (2:1:[]) \)
  - We replace \( \text{sum} \ (3:2:1:[]) \) by the rhs \( x + \text{sum} \ \text{xs} \), using the binding
  - Result: \( 3 + \text{sum} \ (2:1:[]) \)

Haskell Execution Example (cont.,)

- Reduction 2
  
  - Matching \( \text{sum} \ (2:1:[]) \) against pattern \( \text{sum} \ (x:xs) \) succeeds with binding \( x \mapsto 2, xs \mapsto (1:[]) \)
  - We replace \( \text{sum} \ (2:1:[]) \) by the rhs \( x + \text{sum} \ \text{xs} \),
  - Result: \( 3 + 2 + \text{sum} \ (1:[]) \)

- Reduction 3
  
  - Matching \( \text{sum} \ (1:[]) \) against pattern \( \text{sum} \ (x:xs) \) succeeds with binding \( x \mapsto 1, xs \mapsto (\emptyset) \)
  - We replace \( \text{sum} \ (1:[]) \) by the rhs \( x + \text{sum} \ \text{xs} \),
  - Result: \( 3 + 2 + 1 + \text{sum} \ \emptyset \)

- Reduction 4
  
  - Matching \( \text{sum} \ \emptyset \) against pattern \( \text{sum} \ \emptyset \) succeeds.
  - We replace \( \text{sum} \ \emptyset \) by the rhs \( 0 \),
  - Result: \( 3 + 2 + 1 + 0 \)

- The + operator can be defined recursively on numbers, to be reduced in a similar manner. However, in Haskell it is built-in, so would be executed directly.

Haskell Execution Example (a last look)

- In practise we wouldn’t show a reduction in such excruciating detail.

  Instead

  \[
  \text{sum} \ (3:2:1:[]) \\
  = \quad \text{"matches 2nd sum pattern "} \\
  3 + \text{sum} \ (2:1:[]) \\
  = \quad \text{"matches 2nd sum pattern "} \\
  3 + 2 + \text{sum} \ (1:[]) \\
  = \quad \text{"matches 2nd sum pattern "} \\
  3 + 2 + 1 + \text{sum} \ \emptyset \\
  = \quad \text{"matches 1st sum pattern "} \\
  3 + 2 + 1 + 0
  \]

Here we underline the application being matched at each step.
A Key Principle

- Haskell execution replaces sub-expressions, by ones defined to be equal (but hopefully simpler).
- This is an example of a general principle that it very desirable in functional languages — **Referential Transparency**.
- A language is **Referentially Transparent** if
  - replacing an expression by another equal expression does not change the meaning/value of the program as a whole.
  - e.g Given program $2 \times \text{sum}(3:2:1:[]) + x$, then the following are all equivalent programs:
    2 \times (3 + \text{sum}(2:1:[])) + x \\
    2 \times (3 + 2 + 1 + 0) + x \\
    2 \times 6 + x \\
    12 + x

Referential Transparency (Examples)

- **Referentially Transparent:**
  - A function whose output depends only on its inputs.
  - Expressions built from standard arithmetic operators.
  - None of the above have any “side-effects”.
- **Referentially Opaque:**
  - A function whose value depends on some global variable elsewhere.
  - A procedure/function that modifies global state.
  - The assignment statement.
  - A function that performs I/O, it depends on the global state of “real world”, and modifies it.
  - Most of the above are examples of “side-effects”

Why Referential Transparency matters

- Reasoning about program behaviour is easier “substituting equals for equals”
- Code optimization is much simpler
- Scope for code optimization is much greater
- A programming language where every construct is referentially transparent is called “pure”
  - Haskell (and Clean) are pure functional languages
  - ML, Scheme, LISP are impure functional languages (they have assignment and I/O side-effects).