Introduction to Functional Languages

1 Referential transparency, no side effects
   “substitution of equals for equals”

2 Function definitions can be used
   Suppose $f$ is defined to be the function $(\text{fn } x=>\text{exp})$, then $f(\text{arg})$ can be replaced by $\text{exp}[x := \text{arg}]$

3 Lists not arrays

4 Recursion not iteration

5 Universal parametric polymorphism, type reconstruction

6 Higher-order functions
   New idioms, total procedural abstraction
Barendreit, 2013, page xvii. The power of fun programming derives from:

- constant meaning (referential transparency)
- flexibility of high-order functions
- goal direction (no storage management)
• In a functional language an expression is the program plus its input.
• Expressions have parts which can be reduced $\triangle$

\[ \Delta \rightarrow \blacksquare \]

• Reduction continues until no more reducible parts exist
• The result corresponds to the output.
Schematic Representation of Reduction

△ △ △

▲ △ △

▲ △ △

▲ △ △
fun square x = x * x;
fun sos (x,y) = (square x) + (square y);

sos (3,4)
==> (square 3) + (square 4) [Def’n of sos]
==> 3*3 + (square 4) [Def’n of square]
==> 9 + (square 4) [Def’n of *]
==> 9 + 4*4 [Def’n of square]
==> 9 + 16 [Def’n of *]
==> 25 [Def’n of +]
Language of expressions only, no statements.

```haskell
fun test (x) = if x>20 then "big" else "small"

test (sos (3,4))
==> test(25)
==> if 25>20 then "big" else "small"
==> "big"
```
 Canonical value. A canonical value is one which cannot be rewritten further. For example, 2+3 is not canonical, it evaluates to 5; 5 is a canonical value. See canonical in the “The on-line hacker Jargon File,” version 4.4.7, 29 Dec 2003.
History of Functional Languages

1959  LISP: List processing, John McCarthy
1975  Scheme: MIT
1977  FP: John Backus
1980  Hope: Burstall, McQueen, Sannella
1984  COMMON LISP: Guy Steele
1985  ML: meta-language (of LCF), Robin Milner
1986  Miranda: Turner
1990  Haskell: Hudak & Wadler editors
## History of Lisp Dialects

<table>
<thead>
<tr>
<th>Year</th>
<th>Dialect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1955</td>
<td>LISP 1, 1.5, LISP 2 (abandoned)</td>
</tr>
<tr>
<td>1960</td>
<td>Maclisp</td>
</tr>
<tr>
<td>1965</td>
<td>Interlisp</td>
</tr>
<tr>
<td>1970</td>
<td>Lisp Machine Lisp</td>
</tr>
<tr>
<td>1975</td>
<td>Scheme</td>
</tr>
<tr>
<td>1980</td>
<td>R5RS</td>
</tr>
<tr>
<td>1985</td>
<td>R6RS</td>
</tr>
<tr>
<td>1990</td>
<td>R7RS small</td>
</tr>
<tr>
<td>1995</td>
<td>Common Lisp</td>
</tr>
<tr>
<td>2000</td>
<td>Le Lisp</td>
</tr>
<tr>
<td>2005</td>
<td>T</td>
</tr>
<tr>
<td>2010</td>
<td>Emacs Lisp</td>
</tr>
<tr>
<td>2015</td>
<td>AutoLISP</td>
</tr>
<tr>
<td>2019</td>
<td>OpenLISP</td>
</tr>
</tbody>
</table>

### Additional Dialects

- PicoLisp
- EuLisp
- ISLISP
- newLISP
- Racket
- GNU Guile
- Visual LISP
- Qi, Qii
- Shen
- Clojure
- Arc
- LFE
- Hy
xkcd—a webcomic of romance, sarcasm, math, and language
by Randall Munroe
Lazy Versus Eager

Lazy: don’t evaluate the function (constructor) arguments until needed (call-by-name), e.g., Haskell. Permits infinite data structures.
Eager: call-by-value, e.g., ML
Lazy versus Eager

A tutorial video
by the University of Glasgow

Call-by-need is a descriptive, but distinct variation of the call-by-name parameter passing mechanism.
Made key breakthroughs in disparate areas: proofs, languages, concurrency
John Robin Gorell Milner was educated at Eton College and Cambridge. He worked for a few years in Ferranti Ltd before joining the University of Edinburgh in 1973, becoming Professor of Computation Theory in 1984. In 1986, with colleagues, he founded the Laboratory for Foundations of Computer Science at Edinburgh. He was elected Fellow of the Royal Society in 1988, and in 1991 won the Turing Award. Robin Milner was appointed Professor of Theoretical Computer Science at Cambridge University in 1995, and was Head of the Computer Laboratory there from January
1991 Turing Award citation lists three achievements:

1. LCF, the mechanisation of Scott’s Logic of Computable Functions, probably the first theoretically based yet practical tool for machine-assisted proof construction;

2. ML, the first language to contain polymorphic type inference together with a type-safe exception handling mechanism;

3. CCS (Calculus of Communicating Systems), a general theory of concurrency.
Salient Features of SML

1. Strongly-typed, eager, functional language
2. Polymorphic types, type inference
3. Algebraic type definitions
4. Pattern matching function definitions
5. Exception handling
6. Module (signatures/structures) system
7. Interactive
Information about ML


ML and Haskell

- Similar to ML: functional, strongly-typed, algebraic data types, type inferencing
- Differences: no references, exception handling, or side effects of any kind; lazy evaluation, list comprehensions
Introduction to Haskell

1. Haskell (1.0) 1990
2. By 1997 four iterations of language design (1.4)
Salient Features of Haskell

1. Strongly-typed, lazy, functional language
2. Polymorphic types, type inference
3. Algebraic type definitions
4. Pattern matching function definitions
5. System of classes
6. Interactive
Loading package base-1.0 ... linking ... done.
Prelude> :load u:main
Compiling Main
Ok, modules loaded: Main.
*Main> f "abcdefggh"
"hgfedcba"
*Main> :reload
Ok, modules loaded: Main.
*Main> :quit

ghci> import System.IO as IO

-- main program to reverse each line of input
main =
do IO.getContents >>= IO.putStrLn . unlines . map f . lines

f line = reverse line

(--
   built-in functions:
   -- unlines :: [String] -> String ; breaks input into separate lines
   -- lines :: String -> [String] ; combines separate lines into one string
   -- reverse :: [a] -> [a] ; reverse list back to front
--)

 GHC Interactive, version 6.4.1, for Haskell 98.
http://www.haskell.org/ghc/
Type :? for help.
Information about Haskell


Hutton, Graham, *Programming in Haskell.*

O’Donnell et al., *Discrete Mathematics Using a Computer.*
Haskell

- Similar to ML: functional, strongly-typed, algebraic data types, type inferencing
- Differences: no references, exception handling, or side effects of any kind; lazy evaluation, list comprehensions

```haskell
fac n = if n==0 then 1 else n * fac (n-1)
data Tree = Leaf | Node (Tree, String, Tree)
size (Leaf) = 1
size (Node (l,_,r)) = size (l) + size (r)
squares = [ n*n | n <- [0..] ]
pascal = iterate (\row->zipWith (+) ([0]++row) (row++[0])) [1]
```
Patterns are a very natural way of expressing complex choices. Consider the code to re-balance red-black trees. This is usually quite complex to express in a programming language. But with patterns it can be more concise. Notice that constructors of user-defined types (line `RBTree`) as well as pre-defined types (like list) can be used in patterns.
data Color = R | B deriving (Show, Read)
data RBTree a = Empty | T Color (RBTree a) a (RBTree a) deriving (Show, Read)

balance :: RBTree a -> a -> RBTree a -> RBTree a
balance (T R a x b) y (T R c z d)=T R (T B a x b) y (T B c z d)
balance (T R (T R a x b) y c) z d=T R (T B a x b) y (T B c z d)
balance (T R a x (T R b y c)) z d=T R (T B a x b) y (T B c z d)
balance a x (T R b y (T R c z d))=T R (T B a x b) y (T B c z d)
balance a x (T R (T R b y c) z d)=T R (T B a x b) y (T B c z d)
balance a x b = T B a x b
Read the On-Line Tutorial
Learn You a Haskell For Great Good