Introduction

Formal Methods

- Application of theoretical computer science fundamentals
- Logic calculi
- Formal languages
- Automata theory
- Program semantics
- Type systems
- Algebraic data types

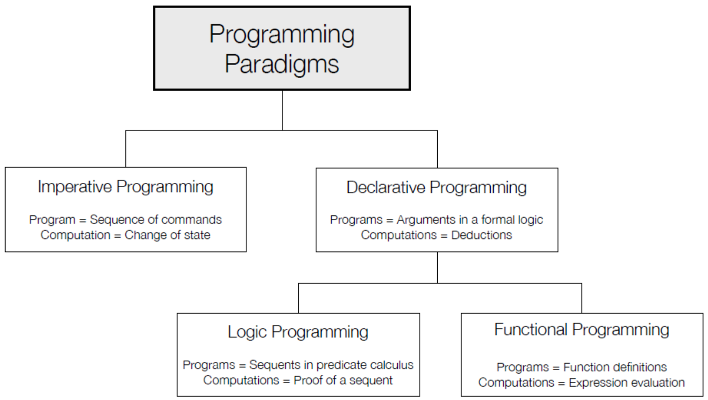
Formal Language

- Set of strings of symbols
- Constrained by specific rules
- Programming languages
- Usage: Specify, invent, transform, analyse, verify, reason about programming languages
- Informal: living natural languages

Execution-based vs. Rule-based thinking

	Execution-based thinking	Rule-based thinking
Basis	Implementation	Rule-based calculus
Focus	Execution runs	Properties
Example 0	Counting with one's fingers	Algebra
Example 1	Truth tables	Propositional calculus
Example 2	Debugging / Test cases	Reasoning about program correctness
First steps	Initially easy to visualize	Needs mental training
Abstraction	Low degree of abstraction possible	High degree of abstraction possible
Scalability	Limits the complexity of problems that can be solved (e.g. truth table with 20 variables)	Allows larger and more complex problems to be solved

Programming Paradigms



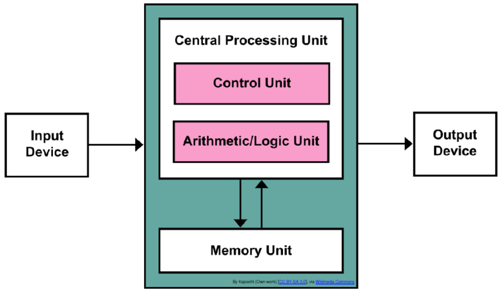
Imperative

- Befehlend
- Focuses on **how** a program operates
- Commands change a program's state

Common building blocks:

- Assignment: $x := x + 1$
- Sequential composition: $(\ldots ; \ldots)$
- Conditional execution: $(if \ldots then \ldots else)$
- Repetition: $(while \ldots do \ldots)$ / $(goto \ldots)$

Von Neumann architecture



Declarative

- Expresses the logic of a computation without describing its control flow
- Describes **what** the program should accomplish
- **how** - left to the language's implementation
- eliminates / minimises side effects

Examples:

- Spreadsheets
- Regular expressions
- Query languages
- Functional programming languages
- Logic programming

Functional Programming Introduction

Basic Features

- Referential transparency
- Functions as first-class citizens
- Higher-order functions
- Algebraic data types
- Pattern matching
- Recursion
- Types & type inference
- Haskell specific: Type classes, Functors, Applicatives, Monads

What is FP?

- Declarative programming paradigm
- Foundation: Chruch's Lambda calculus
- Pure: No (or controlled) mutable state
- Pure: Expressions are by default side effect free

Why functions?

- Simple concept and properties
- High level of abstraction possible
- Powerful reasoning more easily possible

Why use FP?

- Easier to reason about
- Easier to write
- Easier to get right

No Mutable State

- Referential Transparency: WYSIWYG for programmers
- $f(x)$ only depends on the def. of f and the value of x
- **No** mutable variables
- **No** assignments
- **No** imperative control structures
- All data structures are immutable

Problem with mutable state

- Every statement can potentially change the underlying state of the program
- Executions of a statement can depend on previously executed statements

Functions are first-class Citizens

- just like any other values: $1, true$
- Can be anonymous: $(\lambda x.x + 1)$
- Can be input/output to other functions
- Can be composed in powerful ways $f o g$

More FP in the future

- Increased expectations on reliability of software
- Increased demands on scalability, complexity, performance
- FP can surpass the limitations of the mainstream
- FP is an active area of applied research
- Increased adoption of FP features in mainstream languages (generics)

Haskell Introduction

Standard Prelude

Select the first element of a list

```
head [1,2,3,4,5]
1
```

Remove the first element of a list

```
tail [1,2,3,4,5]
[2,3,4,5]
```

Select the nth element of a list

```
[1,2,3,4,5] !! 2
3
```

Select the first n elements of a list

```
take 3 [1,2,3,4,5]
[1,2,3]
```

Remove the first n elements from a list

```
drop 3 [1,2,3,4,5]
[4,5]
```

Calculate the length of a list

```
length [1,2,3,4,5]
5
```

Calculate the sum of a list of numbers

```
sum [1,2,3,4,5]
15
```

Calculate the product of a list of numbers

```
product [1,2,3,4,5]
120
```

Append two lists

```
[1,2,3] ++ [4,5]
[1,2,3,4,5]
```

Reverse a list

```
reverse [1,2,3,4,5]
[5,4,3,2,1]
```

Function Application Syntax

```
f a b + c * d
```

- Function application is denoted using space
- Multiplication is denoted using $*$
- Function application has higher prio than other operators

Mathematics	Haskell
$f(x)$	`f x`
$f(x,y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x,g(y))$	`f x (g y)`
$f(x)\ g(y)$	`f x * g y`

Useful GHCi Commands

Command	Meaning
`:load` *name*	load script name
`:reload`	reload current script
`:set editor` *name*	set editor to *name*
`:edit` *name*	edit script *name*
`:edit`	edit current script
`:type` *expr*	show type of *expr*
`:?`	show all commands
`:quit`	quit GHCi

Naming Requirements

- Function and argument names: begin with lowercase letter
- List arguments: s-suffix, by convention: xs, ns, nss

The Layout Rule

- In a sequence of definitions, definitions must begin in the same column
- Avoids the need for braces and semicolons

Types and Classes

Type

- Name for a collection of related values
- e.g. $Bool = False|True$
- Every well-formed expression has a type
- Type can be automatically calculated at compile time: *type inference*
- Removing the need for type checks at run time =¿ safer /faster

Type Error

- Applying a function to one or more arguments of the wrong type

Basic Types

- **Bool**: logical values
- **Char**: single values
- **String**: strings of chars
- **Int**: fixed-precision int
- **Integer**: arbitrary-precision integers
- **Float**: floating-point numbers

List Types

- Sequence of values of the same type
- Type of a list says nothing about the length
- Lists of lists possible

```
[['a'], ['b','c']] :: [[Char]]
```

Tuple Types

- Sequence of values of different types
- Type of a tuple encodes its size
- Type of components is unrestricted

```
(False, 'a', True) :: (Bool, Char, Bool)
```

Function Types

- Mapping from values of one type to values of another
- Argument and result types are unrestricted
- Functions with multiple arguments / results possible using lists / tuples

```
add :: (Int, Int) -> Int
add (x,y) = x + y
```

Curried Functions

- Multiple input arguments typically implemented by returning functions as results
- Functions that take their arguments one at a time
- Functions with more that two arguments can be curried by returning nested functions

```
add' :: Int -> (Int -> Int)
add' x y = x + y
```

Currying Conventions

- $->$ operator is right associative
- Rightmost type is the result
- Precending types are the input
- Consequence: Function application is left associative
- All functions in Haskell are normally defined in curried form

Polymorphic Functions

- Function type contains one or more type variables
- Type variables must begin with lowercase letters

```
length :: [a] -> Int
```

Overloaded Functions

- Function type contains one or more class constraints
- Class constraints are expressed as type classes
- Can be instantiated to any types that satisfy the constraints

```
(+) :: Num a => a -> a -> a
```

Type classes in Haskell:

- Num: Numeric types
- Eq: Equality types
- Ord: Ordered types

```
(+) :: Num a => a -> a -> a
(==) :: Eq a => a -> a -> Bool
(<) :: Ord a => a -> a -> Bool
```

Defining Functions

Conditional Expressions

- Can be nested
- Must always have an else branch

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

Guarded Equations

- Make definitions using multiple conditions
- Easier to read

```
abs n
  | n >= 0 = n
  | otherwise = -n
```

Pattern Matching

- Can be defined many different ways
- Some ways may be more efficient
- _: wildcard pattern, matches any argument
- Patterns are matched in order
- Patterns may not repeat variables

```
not :: Bool -> Bool
not False = True
not True = False
```

List Patterns

- Non-empty lists are constructed by repeated use of *cons* operator: (:)
- Functions on lists can be defined using $x : xs$ patterns

```
[1,2,3,4] = 1 : (2 : (3 : (4 : [])))
```

Lambda Expressions

- Used to define anonymous functions
- e.g. $\backslash x- > x + x$
- Can give a formal meaning to functions defined using currying

```
add x y = x + y
add = \x -> (\y -> x + y)
```

Avoid naming functions that are only referenced once:

```
odds n = map f [0 .. n - 1]
         where
            f x = x * 2 + 1
odss n = map (\x -> x * 2 + 1) [0 .. n - 1]
```

Operators

- **Prefix** notation is used for function application
- Is **Infix** notation desired, one may use operators (e.g. +)
- Functions can be converted into operators using backticks: '*div*'
- Operators can be converted to functions using brackets: (+)

List Comprehension

- Define functions in very compact manner
- Elegant way to perform iteration in declarative style

```
factors n = [x | x <- [1 .. n], n `mod` x == 0]
prime n = factors n == [1, n]
primes n = [x | x <- [2 .. n], prime x]
```

Generators

- States how to generate values for a variable

```
x <- [1 .. 5]
```

Multiple Generators

- Order of generators changes order of elements
- Multiple generators act like nested loops

```
[(x,y) | x <- [1,2,3], y <- [4,5]]
  [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

Dependant Generators

- Later generators can depend on variables, introduced by earlier generators

```
[(x,y) | x <- [1 .. 3], y <- [x .. 3]]
```

Guards

- Restrict values produced by earlier generators

```
[x | x <- [1 .. 10], even x]
```

zip - Function

- Maps two lists to a list of pairs

```
zip :: [a] -> [b] -> [(a, b)]
```

String Comprehensions

- String: sequence of chars enclosed in double quotes
- Internally strings are represented as lists of chars
- Polymorphic list-functions can be applied to strings

```
"abc" :: String
['a', 'b', 'c'] :: [Char]
```

Recursive Functions

Advantages

- Some functions are simpler to define using recursion
- Properties can be proven using induction

Declaring Types and Classes

Type Declarations

- New name for an existing type
- Can make other types easier to read
- Can have type parameters
- Can be nested
- Cannot be recursive

```
type String = [Char]
------------------------
type Pos = (Int, Int)
origin :: Pos
origin = (0, 0)

left :: Pos -> Pos
left (x,y) = (x-1, y)

-- Type parameter --
type Pair a = (a,a)
mult :: Pair Int -> Int
mult (m,n) ) m*n

-- Nested --
type Trans = Pos -> Pos
```

Data Declarations

- Completely new type by specifying its values
- Values of new types can be used the same ways as built in types
- Constructors may also have parameters
- May also have type parameters
- Can be recursive

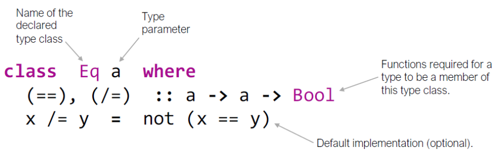
```
data Bool = False | True
data Answer = Yes | No | Unknown

-- Parameter --
data Shape = Circle Float | Rect Float Float
square :: Float -> Shape
square n = Rect n n

-- Type parameters --
data Maybe a = Nothing | Just a

-- Recursive --
data Nat = Zero | Succ Nat
Succ (Succ (Succ Zero)) = 3
```

Polymorphism

1. Ad-hoc Polymorphism: function with the same name denotes different implementations (function overloading / interfaces)
2. Parametric Polymorphism: Code written to work with many possible types
3. Subtype Polymorphism: one type can be substituted for another (subtype / supertype)

Type Classes

- Declared using class declarations



Higher-Order Functions

A function is called higher-order if it takes a function as an argument or returns a function as a result.

```
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

Why are they useful?

- **Common programming idioms** can be encoded as functions within the language itself
- **Domain specific languages** can be defined as collections of higher-order functions
- **Algebraic properties** of higher-order functions can be used to reason about programs

Examples

map

Applies a function to every element of a list.

```
map :: (a -> b) -> [a] -> [b]
-- defined using list comprehension --
map f xs = [f x | x <- xs]
-- defined using recursion --
map f [] = []
map f (x : xs) = f x : map f xs
-- for example: --
map (+1) [1,3,5,7]
 [2,4,6,8]
```

filter

Selects every element from a list that satisfies a predicate.

```
 filter :: (a -> Bool) -> [a] -> [a]
-- defined using list comprehension --
 filter p xs = [x | x <- xs, p x]
-- defined using recursion --
 filter p [] = []
 filter p (x : xs)
     | p x = x : filter p xs
     | otherwise = filter p xs
-- example --
 filter even [1 .. 10]
 [2,4,6,8,10]
```

foldr

A number of functions on lists can be defined using the following simple pattern of recursion:

```
f [] = v
f (x : xs) = x ⊕ f xs
```

Some function $\bigoplus$ is applied to the head of non-empty lists, and f to its tail. The value **v** is typically the identity element of $\bigoplus$.

For example:

```
sum [] = 0
sum (x : xs) = x + sum xs

product [] = 1
product (x : xs) = x * product xs

and [] = True
and (x : xs) = x && and xs
```

The higher-order library function **foldr** (fold right) uses this pattern of recursion with the function $\bigoplus$ and the value **v** as arguments:

```
-- defined using recursion --
 foldr :: (a -> b -> b) -> b -> [a] -> b
 foldr f v [] = v
 foldr f v (x : xs) = f x (foldr f v xs)
-- example --
sum = foldr (+) 0
product = foldr (*) 1
and = foldr (&&) True
```