Imperative vs Declarative

Algorithm = Logic + **Control**

Imperative
- How?
- *Explicit* Control
- Sequences of commands for the computer to execute

Declarative
- What?
- *Implicit* Control
- Steps to execute specified by language implementation
Declarative Programming
Subparadigms

● Functional programming
  ○ Building blocks: (higher order) functions
  ○ Computation model: unidirectional, deterministic reduction (λ-calculus, combinatory logic)

● Logic (relational) programming
  ○ Building blocks: predicates/relations, logical variables
  ○ Computation model: multidirectional, non-deterministic search (resolution for horn clauses, 1st order logic)

● Constraint programming

● Other examples: SQL, DSLs (Make), Regexperts etc
Functional Programming

- Program - collection of function definitions
- Computation by term rewriting.
- First-class and higher-order functions
- Recursion.
- Avoid mutable data
- Prominent (research) language: Haskell
- Deterministic and unidirectional (in contrast to logic prog.)
Functional Programming

- Common reduction strategies: leftmost-outermost with call-by-value or call-by-need
- \(\lambda\)-calculus reduction example (rightmost-first reduction)

\[
(\lambda x.x+1)((\lambda y.y+2)3) \rightarrow\rightarrow 6
\]

\[
\begin{align*}
\text{apply} & \rightarrow \text{apply} & \rightarrow \text{apply} & \rightarrow + & \rightarrow 6 \\
/ & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / \quad \rightarrow / \\
\lambda & \quad \rightarrow \quad \lambda \quad \rightarrow \quad \lambda & \quad \rightarrow \quad + & \rightarrow \quad \lambda & \quad \rightarrow \quad 5 & \quad \rightarrow \quad 5 & \quad \rightarrow \quad 1 \\
/ & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / \\
x & \quad \rightarrow + & \quad \lambda & \quad \rightarrow 3 & \quad x & \quad \rightarrow + & \quad 3 & \quad 2 & \quad x & \quad \rightarrow + \\
/ & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / \\
x & \quad \rightarrow 1 & \quad y & \quad \rightarrow + & \quad x & \quad \rightarrow 1 & \quad x & \quad \rightarrow 1 \\
/ & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / & \quad \rightarrow / \\
y & \quad \rightarrow 2
\end{align*}
\]
Logic Programming
Resolution and Unification

- Prominent language: Prolog
- Commonly based on resolution of **Horn clauses** (at most one positive literal), subset of 1st order logic
- **Resolution** - proof by refutation, negation of goal in conjunction with knowledge base leads to contradiction
- **Unification** - unify(f(x,B), f(A,y)) --> f(A,B)
- **Search** (backtracking, _nondeterminism_)
likes(John, Jane).
likes(y, Jim).
likes(y, friend(y)).

- Substitution
UNIFY(likes(John, x), likes(John, Jane)) = \{x/Jane\}
UNIFY(likes(John, x), likes(y, Jim)) = \{x/Jim, y/John\}

- Substitution makes following two sentences identical:
UNIFY(likes(John, x), likes(y, friend(y))) = \{x/John, x/friend(John)\}

- Can't substitute ground term with another
UNIFY(likes(John, x), likes(x, Elizabeth)) = fail

- Variable may not occur in the term it is being unified with.
UNIFY(x, F(x)) = fail
Logic Programming

Resolution example

\{C, \neg A, \neg B\}, \{A\}, \{B\}, \{\neg C\} \rightarrow \{C, \neg B\}, \{B\}, \{\neg C\} \rightarrow \{C\}, \{\neg C\} \rightarrow \{\}

Prolog-style:

C :- A, B.
A.
B.
Goal: C

- Proof by refutation: add "not C" and resolve to empty set i.e. show contradiction
- Use unification where necessary
"Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it." - E.Freuder
Constraint Programming

- Model and solve a problem by specifying constraints that fully characterize the problem. Featuring:
  - **Variables** that range over **domains**
  - Relations between variables stated as **constraints**.
  - **Constraint satisfaction**
    - (finite domains, combinatorial techniques)
  - **Constraint solving**
    - (infinite/complex domains, mathematical techniques)
- **Constraint Satisfaction Problem (CSP)**
  - Examples: Sudoku, map coloring, etc
- Usually in form of constraint logic programming
  - also: functional+constraints, imperative+constraints
Constraint Programming

- Constraints over specific domain:
  - boolean, true/false constraints (SAT problem)
  - integer, rational
  - linear, for linear functions only
  - finite, constraints are defined over finite sets
  - mixed

- Solvers: systematic search
  - Generate and test
  - Backtracking

- Improvements to systematic search, heuristics, simplex, various other domain-specific solvers etc
Logic Programming (LP)
and Constraint Logic Programming (CLP)

LP languages (like Prolog) can be viewed as a subset of CLP with:

- **ground terms** (variables, constants, function symbols)
- **single constraint** "=" (syntactic equality)
- **resolution algorithm** for solving
  - backtracking
  - generate and test
% Prolog + CLPFD constraint solver library.
:- use_module(library(clpfd)).

sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,Y],  % Create variables
    Digits ins 0..9,                        % Associate domains to variables
    S #\= 0,                                    % Constraint: S must be different from 0
    M #\= 0,                                      % Constraint: M must be different from 0
    all_different(Digits),         % all elements must take different values
    1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E #= 10000*M + 1000*O + 100*N + 10*E + Y,
    label(Digits).                  % Start the search
Hybrid
Functional + Logic + Constraint

- **Functional:**
  - Efficiency
    - (Deterministic) reduction of function expressions
    - More *control* compared to logic programming

- **Logic:**
  - Expressive power:
    - Logical variables
    - Built-in search

- **Constraint:**
  - Expressiveness, efficient solving strategies
  - **Narrowing** and/or **Residuation** techniques
Logic + Functional

- Extend logic language with functional concepts
- Example: Mercury
  - Based on Prolog
  - Strong, static, polymorphic types
  - Explicit determinism system
  - Closures, Currying, and Lambda expressions.
Functional + Logic

- Extend functional language with logic concepts
- Example: Curry
Curry
Language overview

- General purpose functional logic programming language.
- **Functional**: (deterministic) reduction of nested expressions, higher-order functions, lazy evaluation
- **Logic**: logical variables, partial data structures, built-in search
- **Constraint**: constraint structure, solvers
- **Concurrent**: Concurrent evaluation of constraints with synchronization on logical variables
- Syntax is mostly Haskell
  - Missing type classes :( (experimental support exists)
  - Added mainly "where x free" for logical variables
Curry
Search for solutions

- **Logic variable** - variable in the condition and/or right-hand side of a rewrite rule which does not occur in the left-hand side:
  - \( x == 2 + 2 \) *where* \( x \) *free*
  - \( \text{path } a \ z = \text{edge } a \ b \ && \text{path } b \ z \) *where* \( b \) *free*

- **Search for solutions** - compute values for the arguments of functions so that the functions can be evaluated
  - Instantiates logic variables
  - Compute all possible solutions, one at a time
Curry

Search for solutions : Example

1. Prelude> x &&(y || (not x)) where x,y free
   Free variables in goal: x, y
   Result: True
   Bindings:
   x=True
   y=True

2. Result: False
   Bindings: x=True
   y=False

3. Result: False
   Bindings: x=False
   y=y

4. No more solutions.
Curry
Non-deterministic functions

- Non-deterministic insert
  \[ \text{insert} :: \text{a} \to \text{[a]} \rightarrow \text{[a]} \]
  \[ \text{insert} \ x \ [] = [x] \]
  \[ \text{insert} \ x \ (y:ys) = x : y : ys \]
  \[ \text{insert} \ x \ (y:ys) = y : \text{insert} \ x \ ys \]

- Multiple result values:
  \[ \text{coin} :: \text{Int} \]
  \[ \text{coin} = 0 \]
  \[ \text{coin} = 1 \]

- Calls to non-deterministic functions?
  \[ \text{coin} + \text{coin} \]

- \[ (?) :: \text{a} \to \text{a} \to \text{a} \] -- choice operator from Prelude
Curry
Constraints

● Types:
  ○ success :: Success
    ■ no visible literal values
    ■ denotes result of successfully solved constraints

● Operators:
  ○ Constrained equality
    ■ (=:=) :: a -> a -> Success
  ○ Parallel conjunction
    ■ (&) :: Success -> Success -> Success
  ○ Constrained expression
    ■ (&>) :: Success -> a -> a
Curry
Operational semantics

- Lazy evaluation of expressions with possible instantiation of free variables in expression
  - ground expressions -> as lazy functional language
  - instantiations -> as in logic programming

- Answer expression:
  - substitution $\sigma$ + expression: $e$  (Example: $\{x=0,y=2\}2$)
  - solved if $e$ is data term

- Disjunctive expression:
  - Multiset of answer expressions
  - $\{ \sigma_1 e_1 \mid \sigma_2 e_2 \mid \ldots \mid \sigma_n e_n \}$

- Computation step: reduction in exactly one unsolved answer expression: $\{ \sigma_1 e_1 \mid \sigma_2 e_2 \mid \ldots \mid \sigma_n e_n \}$
Curry
Operational semantics

- **Examples:**
  - \( f \ 0 = 2 \)
  - \( f \ 1 = 3 \)
  - \( f \ 1 \) evaluates to 3
  - \( f \ x \) evaluates to disjunctive expression \( \{ \{ x=0 \} 2 \ | \ {x=1}3 \} \).

- **Value is *demanded***
  - in argument of a function call if the left-hand side of some rule has a constructor at this position.
  - case expressions
  - arguments of external functions

- **Free variables can occur where value is demanded.**

**Solutions:**
- Residuation
- Narrowing
Residuation - suspend function calls until they are ready for deterministic evaluation (free logic variable is bound)
- incomplete - unable to compute solutions if arguments of functions are not sufficiently instantiated during computation

Example:
- Primitive arithmetic operators
- Boolean equality: (==) :: a -> a -> Boolean
- Prelude> x == 2+2 where x free
  Free variables in goal: x
  *** Goal suspended!
Curry Narrowing

- **Narrowing** - combination of unification for parameter passing and reduction as evaluation mechanism
  - variable is bound to a value selected from among alternatives imposed by constraints.
  - complete in functional sense (normal forms computed if exist)
  - complete in logic sense (solutions computed if exist)

- **Example:**
  - Equational constraint: $(=:=) :: a \rightarrow a \rightarrow \text{Success}$
  - Prelude> x =:= 2+2 where x free
    Free variables in goal: x
    Result: success
    Bindings: x=4 ?
Curry

Rigid and flexible operators

- **Rigid** - operators that residue
  - most primitive operators (arithmetic etc)
- **Flexible** - operators that narrow
  - all defined operators
- For ground expressions (without logic variables) - no difference whether flexible/rigid
- `ensureNotFree` - primitive op to evaluate argument and suspend if logic variable
Curry
Example

- Prelude> x ++ [3,4] =:= [1,2,3,4] where x free
  Free variables in goal: x
  Result: success
  Bindings:
  x=[1,2]
  More solutions? [Y(es)/n(o)/a(ll)] y
  No more solutions.

- Prelude> x + 2 =:= 4 where x free
  Free variables in goal: x
  *** Goal suspended!
Curry
Evaluation example

2+x =:= y & f x =:= y
--> 
{x=1} 2+1 =:= y & 3 =:= y
--> 
{x=1,y=3} 2+1 =:= 3
--> 
{x=1,y=3} 3 =:= 3
--> 
{x=1,y=3}

• another solution: \{x=0, y=2\}
Curry
More examples

http://www.informatik.uni-kiel.de/~curry/examples/
Used materials

- wikipedia.org :)