# Funkcionální a logické programování (Functional and Logic Programming)

Jan Janoušek
*Katedra teoretické informatiky*
*Fakulta informačních technologií ČVUT v Praze*

February 14, 2011

# Pravidla predmetu

Ucitelé:

- prednášky: doc. Ing. Jan Janoušek, Ph.D.
- cvicení: Ing. Martin Poliak

Predmet se zabývá funkcionálním a logickým programováním. Vysvetluje základní principy techto paradigmat neimperativního programování a podrobne se venuje programování v jazycích Lisp a Prolog. Dále jsou studenti seznámeni se základními principy implementace jazyku Lisp a Prolog.
Semestrální práce se zabývá naprogramováním chování agenta hrajícího hru vybíjená.

Další informace – www stránka predmetu:
`https://edux.fit.cvut.cz/courses/MI-FLP/start`

# Imperative Programming Languages

The main idea: the program describes the sequence of statements **how** to compute.

Typical language constructions: variables, assign statement, iterative cycles (for, while, ...).

Strongly connected with von Neumann architecture of computers (memory, processor, I/O): the state of the run of a program is represented by variables.

- Procedural: Fortran, Cobol, Basic, Algol, C, Ada, Pascal, ...
- Object–oriented: C++, Perl, Python, Java, Smalltalk, PHP, C#, ...

# Nonimperative Programming Languages

The main idea: the program expresses **what** to compute.

Typical language constructions: recursion, rules, function definitions, constraints, ...

They are often use in domain–specific descriptions of problems (databases), artificial intelligence, ...

- Functional (based on evaluation of functions): LISP, Haskell, Mathematica, ML, XSLT, XQuery, F# ...
- Domain–specific (description of domain–specific problems): SQL, Flex, ...
- Logic (based on evaluation of mathematical logic): Prolog, Godel, ...
- ...

# Pure functional programming

is based on the following principles:

- The value of an expression depends only on the values of its subexpressions, if any,
- programming as evaluation of mathematical functions **without any side effects performed by the functions (eg. assignment of a global variable in a function is not possible).**
- Implicit storage management. Storage is allocated as necessary by built-in operations on data. Storage that becomes inaccessible is automatically deallocated. (LISP was the first programming language with a garbage collection).

# Pure functional programming, contd.

- Assignments = are not used (neither standard variables are used).
- Iteration cycles are not used. Repeating is done by means of recursion.
- **Functions** are **first-class values**, ie. functions have the same status as any other values.
- A function can be the value of an expressions, can be passed as an argument, can be produced as a result of a function, and can be put in a data structure. Such functions are so-called **high-order functions.**

# Lambda calculus – the smallest universal programming language and a theoretical foundation for functional programming

# HISTORY

Alonzo Church (1903 – 1995)

- Professor at Princeton (1929 – 1967) and UCLA (1967 – 1990).
- Had a few successful graduate students, including
  - Stephen Kleene (Regular expressions)
  - Michael O. Rabin (Nondeterministic automata)
  - Dana Scott (Formal programming language semantics)
  - Alan Turing (Turing machines)

# Turing Machines vs. Lambda Calculus

In 1936:

- Alan Turing invented the Turing machine
- Alonzo Church invented the lambda calculus

In 1937:

- Turing proved that the two models were equivalent, i.e., that they define the same class of computable functions (ie. recursively–enumerable languages).

Functional languages are the lambda calculus with a more syntax.

# THE SYNTAX OF THE LAMBDA CALCULUS

$$
\begin{array}{ccc}
<\mathit{expression}> & ::= & <\mathit{name}> \\
 & | & <\mathit{function}> \\
 & | & <\mathit{application}> \\
<\mathit{expression}> & ::= & ( \; <\mathit{expression}> \; ) \\
<\mathit{name}> & ::= & \mathit{constant} \\
 & | & \mathit{variable} \\
<\mathit{function}> & ::= & \lambda \; \mathit{variable}. <\mathit{expression}> \\
<\mathit{application}> & ::= & <\mathit{expression}> \; <\mathit{expression}>
\end{array}
$$

Constants are numbers and built-in functions; variables are identifiers.

# Simple Examples

$(+(*\ 5\ 6)(*\ 8\ 3))$
prefix notation

Evaluation: select a redex and evaluate it:

$$\begin{array}{rcl} (+(*\ 5\ 6)(*\ 8\ 3)) & \rightarrow & (+\ 30\ (*\ 8\ 3)) \\ & \rightarrow & (+\ 30\ 24) \\ & \rightarrow & 54 \end{array}$$

# A Simple Example of Function

The only other thing in the lambda calculus is lambda abstraction: a notation for defining unnamed functions.

$(\lambda\, x\ .\ +\ x\ 1)$

That function of $x$ that adds $x$ to 1.

Function application associates left-to-right:

$$\begin{array}{ccc} (+\,3\,4) & \rightarrow & ((+\,3)\,4) \\ & \rightarrow & 7 \end{array}$$

# Beta-Reduction

Evaluation of a lambda abstraction – *beta-reduction* – is just substitution:

$$\begin{array}{lcl} (\lambda\, x\,.\; +\; x\; 1)4 & \rightarrow & (+\; 4\; 1) \\ & \rightarrow & 5 \end{array}$$

The argument may appear more than once

$$\begin{array}{lcl} (\lambda\, x\,.\; +\; x\; x)4 & \rightarrow & (+\; 4\; 4) \\ & \rightarrow & 8 \end{array}$$

or not at all

$$(\lambda\, x\,.\; +\; 1\; 2)4 \quad \rightarrow \quad 3$$

# Beta-Reduction

Functions may be arguments:

$$
\begin{array}{rcl}
(\lambda f\,.f\;3)(\lambda x\,.\;+\;x\,1) & \rightarrow & (\lambda x\,.\;+\;x\,1)3 \\
 & \rightarrow & 4
\end{array}
$$

# Free and Bound Variables

$(\lambda\, x\, .\ +\ x\, y)4$
Here, $x$ is like a function argument but $y$ is like a global variable.

Technically, $x$ occurs **bound** and $y$ occurs **free** in $(\lambda\, x\, .\ +\ x\, y)$

However, both $x$ and $y$ occur free in $(+xy)$

# Beta-Reduction More Formally

$$(\lambda x \,.\, E)\; F \to_\beta E'$$

where $E'$ is obtained from $E$ by replacing every instance of $x$ that appears free in $E$ with $F$.

The definition of free and bound mean variables have scopes. Only the rightmost $x$ appears free in

$$(\lambda x \,.\, + \; (- \; x \; 1)) \; x \; 3$$

so

$$\begin{aligned}(\lambda x \,.\, (\lambda x \,.\, + \; (- \; x \; 1)) \; x \; 3) \; 9 &\to (\lambda \; x \,.\, + \; (- \; x \; 1)) \; 9 \; 3 \\ &\to + \; (- \; 9 \; 1) \; 3 \\ &\to + \; 8 \; 3 \\ &\to 11\end{aligned}$$

# Another Example

$$
\begin{aligned}
\big(\lambda x\,.\,\lambda y\,.\,+\;x\,\big((\lambda x\,.\,-\;x\,3)\;y\big)\big)\,5\,6 &\rightarrow \big(\lambda y\,.\,+\;5\,\big((\lambda x\,.\,-\;x\,3)\;y\big)\big)\,6 \\
&\rightarrow +\,5\,\big((\lambda x\,.\,-\;x\,3)\,6\big) \\
&\rightarrow +\,5\,(-\,6\,3) \\
&\rightarrow +\,5\,3 \\
&\rightarrow 8
\end{aligned}
$$

# Alpha-Conversion

One way to confuse yourself less is to do $\alpha$-conversion: renaming a $\lambda$ argument and its bound variables.

Formal parameters are only names: they are correct if they are consistent.

$$
\begin{aligned}
(\lambda x\,.\,(\lambda x\,.\,+\,(-\,x\,1))\;x\,3)\;9 &\leftrightarrow (\lambda x\,.\,(\lambda y\,.\,+\,(-\,y\,1))\;x\,3)\;9\\
&\rightarrow ((\lambda y\,.\,+\,(-\,y\,1))\;9\;3)\\
&\rightarrow (+\,(-\,9\;1)\;3)\\
&\rightarrow (+\,8\;3)\\
&\rightarrow 11
\end{aligned}
$$

# Beta-Abstraction and Eta-Conversion

Running $\beta$-reduction in reverse, leaving the "meaning" of a lambda expression unchanged, is called *beta abstraction*:

$$+\,4\,1 \leftarrow (\lambda x\,.\,+\;x\,1)\,4$$

Eta-conversion is another type of conversion that leaves "meaning" unchanged:

$$(\lambda x\,.\,+\;1\;x) \leftrightarrow_{\eta} (+\,1)$$

Formally, if $F$ is a function in which $x$ does not occur free,

$$(\lambda x\,.\,F\,x) \leftrightarrow_{\eta} F$$

# Reduction Order

The order in which you reduce things can matter.

$$(\lambda x\,.\,\lambda y\,.\,y)\,\big((\lambda z\,.\,z\,z)\,(\lambda z\,.\,z\,z)\big)$$

Two things can be reduced:

$$(\lambda z\,.\,z\,z)\,(\lambda z\,.\,z\,z)$$

$$(\lambda x\,.\,\lambda y\,.\,y)\,(\cdots)$$

However,

$$(\lambda z\,.\,z\,z)\,(\lambda z\,.\,z\,z) \rightarrow (\lambda z\,.\,z\,z)\,(\lambda z\,.\,z\,z)$$

$$(\lambda x\,.\,\lambda y\,.\,y)\,(\cdots) \rightarrow (\lambda y\,.\,y)$$

# Normal Form

A lambda expression that cannot be $\beta$-reduced is in *normal form*. Thus,

$$\lambda y \,.\, y$$

is the normal form of

$$(\lambda x \,.\, \lambda y \,.\, y)\ \big((\lambda z \,.\, z\, z)\ (\lambda z \,.\, z\, z)\big)$$

Not everything has a normal form. E.g.,

$$(\lambda z \,.\, z\, z)\ (\lambda z \,.\, z\, z)$$

can only be reduced to itself, so it never produces an non-reducible expression.

# Church-Rosser Theorem

If a lambda calculus expression can be evaluated in two different ways and both ways terminate, then both ways will yield the same result.

Furthermore, if there is a way for an expression evaluation to terminate, using normal order will cause termination.

# Recursion

Where is recursion in the lambda calculus?

$$FAC = \left(\lambda n \,.\, IF\ (=\ n\ 0)\ 1\ \left(*\ n\ \left(FAC\ (-\ n\ 1)\right)\right)\right)$$

This does not work: functions are unnamed in the lambda calculus. But it is possible to express recursion *as a function*.

$$\begin{aligned} FAC &= (\lambda n \,.\, \ldots\ FAC\ \ldots) \\ &\leftarrow_\beta (\lambda f \,.\, (\lambda n \,.\, \ldots\ f\ \ldots))\ FAC \\ &= H\ FAC \end{aligned}$$

That is, the factorial function, *FAC*, is a *fixed point* of the (non-recursive) function *H*:

$$H = \lambda f \,.\, \lambda n \,.\, IF\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1)))$$

# Recursion

Let's invent a function $Y$ that computes $FAC$ from $H$, i.e., $FAC = Y\ H$:

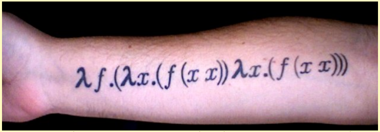
$$\begin{array}{rcl} FAC & = & H\ FAC \\ Y\ H & = & H\ (Y\ H) \end{array}$$

$$\begin{array}{rcl}
FAC\ 1 & = & Y\ H\ 1 \\
 & = & H\ (Y\ H)\ 1 \\
 & = & (\lambda f\,.\,\lambda n\,.\,IF\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ 1 \\
 & \to & (\lambda n\,.\,IF\ (=\ n\ 0)\ 1\ (*\ n\ ((Y\ H)\ (-\ n\ 1))))\ 1 \\
 & \to & IF\ (=\ 1\ 0)\ 1\ (*\ 1\ ((Y\ H)\ (-\ 1\ 1))) \\
 & \to & *\ 1\ (Y\ H\ 0) \\
 & = & *\ 1\ (H\ (Y\ H)\ 0) \\
 & = & *\ 1\ ((\lambda f\,.\,\lambda n\,.\,IF\ (=\ n\ 0)\ 1\ (*\ n\ (f\ (-\ n\ 1))))\ (Y\ H)\ 0) \\
 & \to & *\ 1\ ((\lambda n\,.\,IF\ (=\ n\ 0)\ 1\ (*\ n\ (Y\ H\ (-\ n\ 1))))\ 0) \\
 & \to & *\ 1\ (IF\ (=\ 0\ 0)\ 1\ (*\ 0\ (Y\ H\ (-\ 0\ 1)))) \\
 & \to & *\ 1\ 1 \\
 & \to & 1
\end{array}$$

# The $Y$ Combinator

Here's the eye-popping part: $Y$ can be a simple lambda expression.

$$Y =$$




$$= \lambda f\,.\,\big(\lambda x\,.\,f\,(x\,x)\big)\,\big(\lambda x\,.\,f\,(x\,x)\big)$$

$$\begin{aligned}
Y\,H &= \Big(\lambda f\,.\,\big(\lambda x\,.\,f\,(x\,x)\big)\,\big(\lambda x\,.\,f\,(x\,x)\big)\Big)\;H \\
&\to \big(\lambda x\,.\,H\,(x\,x)\big)\,\big(\lambda x\,.\,H\,(x\,x)\big) \\
&\to H\,\Big(\big(\lambda x\,.\,H\,(x\,x)\big)\,\big(\lambda x\,.\,H\,(x\,x)\big)\Big) \\
&\leftrightarrow H\,\bigg(\Big(\lambda f\,.\,\big(\lambda x\,.\,f\,(x\,x)\big)\,\big(\lambda x\,.\,f\,(x\,x)\big)\Big)\;H\bigg) \\
&= H\,(Y\,H)
\end{aligned}$$

"Y: The function that takes a function $f$ and returns $f(f(f(f(\cdots))))$"

# Combinators

Generally, a lambda calculus expression with no free variables is called a combinator.

Example of other combinators:

- I: $\lambda x.x$ (Identity)
- App: $\lambda f.\lambda x.(f\ x)$ (Application)
- C: $\lambda f.\lambda g.\lambda x.(f\ (g\ x))$ (Composition)
- L: $(\lambda x.(x\ x)\ \lambda x.(x\ x))$ (Loop)
- Cur: $\lambda f.\lambda x.\lambda y.((f\ x)\ y)$ (Currying) ie. function of more parameters
- Seq: $\lambda x.\lambda y.(\lambda z.y\ x)$ (Sequencing--normal order)
- ASeq: $\lambda x.\lambda y.(y\ x)$ (Sequencing--applicative order)
- Y: $\lambda f\cdot(\lambda x\cdot f\ (x\ x))\ (\lambda x\cdot f\ (x\ x))$ (recursion, see previous slides)

where y denotes a thunk, i.e., a lambda abstraction wrapping the second expression to evaluate.