Logic and Functional Programming

Basic Elements of LISP

Dr. Cristian-Paul Bara

Computer Science Department
Faculty of Mathematics and Computer Science,
Babeş Bolyai University,
Cluj-Napoca, Romania

Table of Contents

Functional Programming

Introduction to The LISP Programming Language

Functional Programming

- New Programming Language. Focusing on value propagation through expressions.
- Forgoes the assignment operation. I.e., Functional programming works at a higher level of abstraction similar to how the `goto` statement is abstracted by `while, repeat, for`.
- Caters to parallel programming due to the lack of assignment and independance of final values or order of execution.
- AI was an important promoter of the paradigm. The focus of the field being modeling of behaviors considered intelligent: cognitive processes, machine translation, information retrieval; which require symbolic manipulation rather than calculation.

Functional Programming

- Advantages include the manipulation of arbitrarely complex structures, dynamically created structures, notably: lists, strings, or binary trees.
- A purely functional language implies the lack of any procedural behavior: memory allocation, assignment, non-recursive structures like a for loop.
- Focuses on What not How.
- Examples include: LISP (1958), Hope, ML, Scheme, Miranda, Haskel, Erlang (1995)
- LISP is not a purely functional language
- Functional programming can also be achieved in languages like: Python, Scala, F#

Functional Programming

- Lazy Evaluation: an expression's evaluations is delayed until its value is required.
- Eager/Strict Evaluation: an expression is evaluated when its value could be needed
- Example: $f(x, y) = 2x; k = f(d, e)$
  - *lazy evaluation*: only d is evaluated
  - *eager evaluation*: both d and e are evaluated even though e is never used

Very Briefly: Higher Order Logic

- Prolog is a representation of *First order logic*
- Functional Programming is a representation of *Lambda Calculus*

$$< expression >:=< name > \| < function > \| < application >$$
$$< function >:= \lambda < name > . < expression >$$
$$< application >:=< expression >< expression >$$

$$(\lambda x.x^2)7 \rightarrow 49$$

The LISP Programming Language

- LISP (LISt Processing) was first created by John McCarthy in 1958 for list processing from the idea of transcription of algebraic expressions
  - Works with symbolic expressions over numbers
  - Informations is represented by lists
  - Function compositions for creting more complex Functions
  - Recursive function calls
  - A LISP program is a LISP data structure
  - Uses garbage collection as a means of memory deallocation

The LISP Programming Language

- The funtions are a fundamental object, passed as a parameter, returned as a rezult, and part of a data structure
- Used for symbolic processing which allows the manipulations if hierachical structures
- Used in: AI, Expert systems, data mining, natural language processing, agent systems, theorem proving, machine learning, speech recognition, image processing, planning for robotics

"Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot." (Paul Graham, 2001)

- GNU Emacs is written in LISP

The LISP Programming Language

- A LISP program processes symbolic expressions (S-Expressions), The whole program is an S-Expressions
- Standards: CommonLisp (functionsl and imperative, object oriented, can be declarative), CLOS (Common Lisp Object System)
- CommonLisp is implicitly strictly evaluated, though the extension CLAZY allows for lazy evaluation
- Types are dynamically checked
- Basic objects are: atoms (equivalent to a variable and denotes and S-Expression) and lists (most S-Expressions, can be linear or hierachical)
- Prefix notation is used, e.g., $(+\ 1\ 2) = 3$.
- The evaluation of an S-Expression implies determining its value
- No Non-Determinism

Dynamic Data Structures

- Probably the best known: singly linked lists
- An element of a list is formed by a value-link paradigm
- Variations of this structure can generate a large number of structures: lists, trees, graphs, etc.
- Any list can be represented as a set of link-value pairs, though not all sets of link-value pairs are lists
- In LISP lists are recursively defined:
    1. is A is an atom the list (A) is equivalent to the pair (A.NIL)
    2. Let $p = (l_1 l_2 ... l_n)$ be a sublist, the list $(l\ l_1 l_2 ... l_n)$ can be represented as $(l.p)$

Dynamic Data Structures

- (A.B) has no list equivalent
- ((A.NIL).(A.NIL)) has no list equivalent
- (A)$\leftrightarrow$(A.NIL)
- (A B) $\leftrightarrow$ (A.(B))
- (A B C) $\leftrightarrow$ (A.(B.(C.NIL)))
- ((A B) C) $\leftrightarrow$ ((A.(B.NIL)).(C.NIL))
- (A B (C)) $\leftrightarrow$ (A.(B.((C.NIL).NIL)))
- ((A)) $\leftrightarrow$ ((A.NIL).NIL)
- ((A) (B)) $\leftrightarrow$ ((A.NIL).((B.NIL).NIL))

Syntax

- numeric atom: (1), (+12.NIL), (-3)
- string atom: ("asdf")
- symbol: (A)
- atom: a number atom, string atom, symbol, or empty list ()$\leftrightarrow$NIL
- list: $(e_1 e_2 ... e_n)$
- point pair: $(e_1 . e_2)$
- S-Expression: an atom, a list, or a point pair
- form: an evaluable S-Expression
- a LISP program: a set of forms.

Evaluation Rules

- a numeric atom evaluates to its value
- a string atom evaluates to that string including double quotes
- a list is evaluable (is a form) is the first element of a list is the name of a function or an operator

>'A A	>(quote A) A	>(A) the function A is undefined	>(NIL) the function NIL is undefined
>'(A) (A)	>'(NIL) (NIL)	>() NIL	>(()) the function NIL is undefined
>NIL NIL	>'(()) (NIL)	>(A.B) the function A\.B is undefined	>'(A.B) (A\.B)

LISP Functions

- (CONS e_1 e_2): list or point pair
- The constructor function
- Forms the point pair containing its parameters

  (CONS 'A 'B) = (A . B)

  (CONS 'A '(B)) = (A B)

  (CONS '(A B) '(C)) = ((A B) C)

  (CONS '(A B) '(C D)) = ((A B) C D)

  (CONS 'A '(B C)) = (A B C)

  (CONS 'A (CONS 'B '(C))) = (A B C)

LISP Functions

- (CAR <list or point pair>): expression, returns the left side of a point pair
- (CDR <list or point pair>): expression, returns the right side of a point pair

  (CAR '(A B C)) = A

  (CAR '(A . B)) = A

  (CAR '((A B) C D)) = (A B)

  (CAR (CONS '(B C) '(D E))) = (B C)

  (CDR '(A B C)) = (B C)

  (CDR '(A . B)) = B

  (CDR '((A B) C D)) = (C D)

  (CDR (CONS '(B C) '(D E))) = (D E)

LISP Functions

- CONS will remake the the pair that CAR and CDR break apart
  (CDR (CONS '(B C) '(D E))) = (D E)
  (CONS (CAR '(A B C)) (CDR '(A B C))) = (A B C)
  (CAR (CONS 'A '(B C))) = A
  (CDR (CONS 'A '(B C))) = (B C)
- repeated uses of CAR and CDR can be contracted
  (CAADDR '((A B) C (D E))) = D
  (CDAAAR '((((A) B) C) (D E))) = NIL
  (CAR '(CAR (A B C))) = CAR

What about Loops? *Don's worry, won't be on the test*

- Y combinator

$$Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

- (defun (Y f) ((funcall (x) (f (x x))) (funcall (x) (f (x x)))))

  ```
  Y = lambda f:  (lambda c:  f(lambda x:  c(c)(x)))(lambda c:
  f(lambda x:  c(c)(x)))
  ```

What about Loops? *Don's worry, won't be on the test*

- Recursive factorial

  ```
  fact = lambda n:  1 if n==0 else n*fact(n-1)
  ```

- using Y combinator

  ```
  fact_nr = lambda f:  lambda n:  1 if n==0 else n*f(n-1)

  Y = lambda f:  (lambda c:  f(lambda x:  c(c)(x)))(lambda c:
  f(lambda x:  c(c)(x)))

  fact = Y(fact_nr)
  ```