

AI Languages

Background

- Interest in artificial intelligence first surfaced in the mid 1950.
- Linguistics, psychology, and mathematics were only some areas of application for AI. Linguists were concerned with natural language processing, while psychologists were interested in modeling human information and retrieval.

Background

- In 1956 John McCarthy first developed the basics behind Lisp. His motivation was to develop a list processing language for Artificial Intelligence.
- By 1965 the primary dialect of Lisp was created (version 1.5).
- By 1970 special-purpose computers known as Lisp Machines, were designed to run Lisp programs.

Significant Language Features

- Atoms & Lists - Lisp uses two different types of data structures, atoms and lists.
 - Atoms are similar to identifiers, but can also be numeric constants
 - Lists can be lists of atoms, lists, or any combination of the two
- Functional Programming Style - all computation is performed by applying functions to arguments.

Significant Language Features

- Variable declarations are rarely used.
- Uniform Representation of Data and Code - example: the list (A B C D)
 - a list of four elements (interpreted as data)
 - is the application of the function named A to the three parameters B, C, and D (interpreted as code)

Significant Language Features

- Reliance on Recursion - a strong reliance on recursion has allowed Lisp to be successful in many areas, including Artificial Intelligence.
- Garbage Collection - Lisp has built-in garbage collection, so programmers do not need to explicitly free dynamically allocated memory.

Areas of Application

- Lisp totally dominated Artificial Intelligence applications for a quarter of a century, and is still the most widely used language for AI.
- Many programming language researchers believe that functional programming is a much better approach to software development, than the use of Imperative Languages (Pascal, C++, etc).

Areas of Application

- Artificial Intelligence
 - AI Robots
 - Computer Games (Craps, BlackJack, Reversi)
 - Pattern Recognition
- List Handling and Processing
- Tree Traversal (Breath/Depth First Search)
- Educational Purposes (Functional Style Programming)

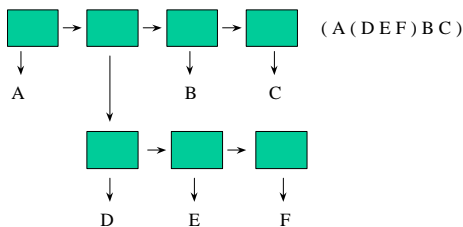
Functional PLs

- Early in AI research, there was a need for symbolic computing
- A functional approach was taken
 - Rather than a series of imperative statements executed in order, all instructions are functions which return a value which is then used in another function - this method was chosen due to the strong tie between the work being performed and mathematical functions
- Lisp began as a purely functional language although it picked up elements from other PLs

LISP

- A typeless language
- Data Structures: atoms and lists
 - Atom - any literal (char, string, number)
 - List - nil or (x) where x is an atom, a series of atoms, or a list, or any combination of these
 - Example of a list (A (B C D) E F)
 - List structures make use of pointers where each item has an info field and a next field where info might point to an atom or another list
- All program code composed of functions

List Structures



Early Lisp

- The original idea was to make Lisp a universal Turing Machine where both data and instructions would be treated in the same way -- therefore everything is a list including programs
- Lisp's primary function is EVAL which evaluates a list or atom at run-time
- Since Lisp code is represented as lists, Lisp is interpreted

Lisp Functions

- Lambda functions are nameless functions
- Lisp programs can be composed of lambda functions which are executed at the command line
- Lisp also has a function for defining and binding functions (that is, there are facilities for naming functions so that you can use those by calling them from other functions)

example math functions in Lisp

- 42 returns 42
- (* 3 7) returns 21
- (+ 5 7 8) returns 20
- (- 5 6) returns -1
- (- 15 7 2) returns 6
- (- 24 (* 4 3)) returns 12
- (* (+ 5 4) (- 8 (/ 6 2))) returns 45

Variables vs. Symbols and Lists

- Variables point to data items which are either symbols (similar in ways to strings) or lists
- A variable is denoted by its name (e.g., A)
- An atom or list is denoted using a quote mark as in 'A or '(A B C)
- To create a symbol, use the function QUOTE as in (QUOTE A) which returns 'A
- (QUOTE ...) can be also denoted by '

Common Lisp - 1984

- combines many features of earlier Lisps
- static scoping default but allows for dynamic scoping
- data types include arrays, records, complex #'s, strings
- more powerful I/O
- imperative features
- variety of control structures

Common Lisp functions

- PROG - a local block where global variables are unaffected by any statements within the block
- DOTIMES, DOLIST - for iteration (DOTIMES is equivalent to a for loop, DOLIST executes loop body once for each item in the list)
- SETQ for assignment
- DEFUN - defining functions
- Variable initialization to NIL

Two Member Functions in CL

- (DEFUN imember (atm lst)
(PROG ()
 loop_1
 (COND ((NULL lst) (RETURN NIL))
 ((EQUAL atm (CAR lst)) (RETURN T)))
 (SETQ lst (CDR lst))
 (GO loop_1)))
- (DEFUN rmember (atm lst)
(COND ((NULL lst) NIL)
 ((EQUAL atm (CAR lst)) T)
 (T (rmember atm (CDR lst)))))

Scheme

- While Lisp was a pioneering PL in AI and symbolic computing, it had many problems
- Scheme was one of many versions of Lisp that added functionality to the language
- Scheme emerged from MIT in the mid 70's
- Characterized by a small subset of Lisp features.

Some Functions in Scheme

- CAR - content of address register (returns first item of a list)
- CDR - content of decrement register (returns the remainder of the list)
- EQ? - equal (for atoms or lists)
- MEMBER - membership
- ATOM? - is the datum an atom?
- LIST? - is the datum a list?
- NULL? - is the datum a nil pointer or something?

More Scheme Functions

- EVEN?, ODD?, ZERO? - self explanatory
- =, <, >, <=, >= - self explanatory
- CONS - a list constructor (we'll go into more detail later)
- LIST - an alternative list constructor
- IF statement
 - (IF (condition) (then clause) (else clause))
- COND - conditional statement like a CASE statement, with ELSE clause

What does the function return?

- All functions return a value
- Predicate functions EQ?, ATOM?, NULL?, =, <, >, <=, >=, <>, EVEN?, ODD?, ZERO? all return T or Nil
- LIST and CONS list constructor functions return lists
- COND and IF statements return the value of the function executed based on the result of the condition

COND examples

- Form is (COND (test1 result1) (test2 result2) ...)
where tests and results are probably functions
- (COND ((EQ? x 0) (SETQ y 0))
((> x 0) (SETQ y 1))
(T (SETQ y -1)))
- (COND ((<= hours 40) (* hours wages))
(T (+ (* 40 wages)
(* (- hours 40) wages 1.5)))) or using IF
- (IF (<= hours 40) (* hours wages)
(+ (* 40 wages) (* (- hours 40) wages 1.5)))

CONS examples

- Takes two items (the second of which must be a list) and creates a new list whose CAR is the first item and whose CDR is the second
- (CONS 'A '(B C)) returns (A B C)
- (CONS '(A) '(B)) returns ((A) B)
- (CONS '(A B) '(C D)) returns ((A B) C D)
- (CONS 'A '((B C))) returns (A (B C))

Defining Functions in Scheme

- (DEFINE (name params) body)
- (DEFINE (square num) (* num num))
- (DEFINE (second alist) (cdr alist))
- (DEFINE (factorial n)
 (cond ((eq n 0) 1)
 (T (* n (factorial (- n 1))))))
- (DEFINE (foo list)
 (cond ((eq list nil) 0)
 (T (+ (foo (cdr list)) 1))))

More Scheme Functions

- (DEFINE (factorial n)
 (IF (= n 0) 1 (* n (factorial (SUB1 n)))))
- (DEFINE (member atm lis)
 (COND
 ((NULL? lis) NIL)
 ((EQ? Atm (CAR lis)) #T)
 (ELSE (member atm (CDR lis)))))

Another Scheme Function

- (DEFINE (equalsimp lis1 lis2)
 (COND
 ((NULL? lis1) (NULL? lis2))
 ((NULL? lis2) NIL)
 ((EQ? (CAR lis1) (Car lis2))
 (equalsimp (CDR lis1) (CDR lis2)))
 (ELSE NIL)))

A Similar Function

- (DEFINE (equal lis1 lis2)
 (COND
 ((ATOM? lis1) (EQ? lis1 lis2))
 ((ATOM? lis2) Nil)
 ((equal (CAR lis1) (Car lis2))
 (equal (CDR lis1) (CDR lis2)))
 (ELSE NIL)))

The Append Function

- (append '(A B) '(C D E)) returns (A B C D E)
- (append '((A B) C) '(D (E F))) returns
 ((A B) C D (E F))
- (DEFINE (append lis1 lis2)
 (COND
 ((NULL? lis1) lis2)
 (ELSE (CONS (CAR lis)
 (append (CDR lis1) lis2)))))

Two Adder functions

- (DEFINE (adder lis)
 (COND ((NULL? lis) 0)
 (ELSE (+ (CAR lis) (adder (CDR lis)))))
- Or, if we want to first create a list of the items to be added and then perform the add:
- (DEFINE (create-adder lis)
 (COND ((NULL? Lis) 0)
 (ELSE (EVAL (CONS '+ lis)))))
- (create-adder '(3 7 6 1)) creates the list (+ 3 7 6 1) which is then evaluated and the function returns 17

ML

- Statically scoped functional PL
- Syntax similar to Pascal
- Type declarations and type inferencing (meaning that variables do not need to be declared and are strongly typed).
- Exception handling
- Facilities for ADTs
- Functional definitions similar to Lisp but without ()'s or prefix notation

Haskell

- Similar to ML in terms of syntax, static scoping and strongly typed, using the same type of inferencing
- However, Haskell is purely functional
 - no assignment statements
 - no variables
 - no side effects
 - no imperative features (e.g., loops)
- Uses lazy evaluation -- does not evaluate a function until all of the parameter values are known

Haskell Examples

- Factorial:
 - $\text{Fact } 0 = 1$
 - $\text{Fact } n = n * \text{Fact } (n-1)$
- Fibonacci:
 - $\text{Fib } 0 = 1$
 - $\text{Fib } 1 = 1$
 - $\text{Fib } (n+2) = \text{Fib } (n+1) + \text{Fib } n$

More Haskell Examples

- Another version of Fact
 - $\text{Fact } n \mid n == 0 = 1 \mid n > 0 = n * \text{Fact } (n-1)$
- Can use otherwise in conditionals:
 - $\text{Fun } n \mid n < 10 = 0 \mid n > 100 = 2 \mid \text{otherwise} = 1$
- Lists are available using [], concatenation by ++, CONS by :, and arithmetic series can be applied using ..
 - $5:[2, 7, 9]$ results in $[5, 2, 7, 9]$
 - $[1, 5..7] ++ [2..4]$ results in $[1, 5, 6, 7, 2, 3, 4]$

Functional vs. Imperative PLs

- Functional Languages:
 - variables and memory usage is less visible making programming easier (at least in some situations)
 - simpler syntactic structures to deal with (since everything is a list)
 - concurrency easier to design and implement
 - interpreted nature makes large systems easier to build
 - exploits recursion as much as possible, more so than imperative languages

Applications for Functional PL

- Mostly used in AI research
 - Natural Language Understanding (easy parsing partially due to recursive nature)
 - Expert Systems (easy rule format)
 - Knowledge Representation (symbolic capabilities)
 - Machine Learning (dynamic storage)
 - etc...
- Also used to teach functional programming and mathematically oriented programming
- Used to implement EMACS and even some operating systems