LISP

LISt Processing

Or

Lots of Infuriating Silly Parentheses

Introduction

- AI languages
  - LISP
  - Prolog
  - POP11
  - C++, Java, . . .
- Venerable language with a long history
  - Based on λ–calculus (functional)
  - Gave the world:
    - Conditionals (if ---- then -----)
    - Garbage collection
  - Continuous development
    - Still very much in use.

Syntactic Basis

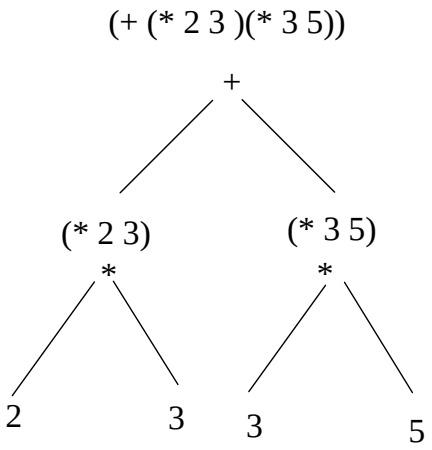
- Symbolic Expressions
  - s-expressions (everything in lisp)
  - Atoms
    - letters, numbers and
      * - + / @ $ % ^ & _ < >
  - Lists
    - Sequence of atoms or other lists
    - Arbitrary depth of nesting
    - Empty list: `()` or `nil`
      - Atom or list
    - Extremely flexible
  - Programs and data are s-expressions

Read-eval-print loop

- Interactive dialogue with the LISP interpreter
  - *reads* user input, tries to *evaluate* it and, if successful *prints* the result
    - First element of list interpreted as a function
    - Remaining elements are the *arguments* of the function
      - (f x y) in LISP is equivalent to f(x,y) in maths
  - Value printed is the result of *applying* the function to its arguments
  - *Forms:* expressions that can be meaningfully evaluated.

Binding and Evaluation

- LISP evaluates everything!
  - Numbers evaluate to themselves
  - Symbols may have a value *bound*
  - Nested functions evaluate recursively (because arguments are evaluated first



Control of LISP Evaluation

- **`quote`**
  - Prevents evaluation of s-expressions
    - ie arguments to be treated as data
  - Abbreviated to `‘`
- **`eval`**
  - Complement to `quote`
    - Reverses its effect
  - Forces evaluation *again*
  - Used in ordinary evaluation by LISP
    - Availability to user extends flexibility

Creating new functions

- Many built-in functions
  - (see “Common Lisp the Reference”)
- New functions defined by: `defun`
  - Once defined: same status as built-ins
  - `defun` does not evaluate arguments
    - Specifies new function
    - Returns value: name of new function

```
(defun <function name> (<formalparameters>)
  <function body>)
```

Conditionals and Predicates

- Branching based on evaluation

```
(cond (<condition1> <action1>)
      (<condition1> <action1>)
     . . . .
      (<condition_n> <action_n>)
```

  - Conditions evaluated first
    - If true then action evaluated, else `nil` returned
- Predicates used for conditions
  - Tests for presence or absence of a property
  - Can test for relations between s-expressions
- Wide variety of control constructs
  - `If, while, repeat,`

Logical Connectives

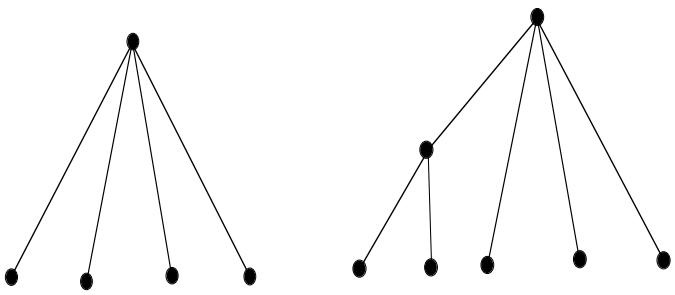
- NOT
  - Returns `t` if argument is `nil`, and `nil` otherwise]
- AND
  - Evaluates args from L to R till one is nil or last arg has been evaluated
  - Returns value of last arg evaluated
- OR
  - Evaluates args from L to R only till one is non-nil
  - This value is returned as the result

Functions and Lists

- Lists build complex data structures
- Lists facilitate recursion
  - `car`: returns first element of list
    - "contents of address part of register"
    - `first` is the modern term
  - `cdr`: returns the list minus the first element
    - "contents of decrement part of register"
    - `rest` is the modern term
  - Facilitates operating on lists of unknown length
    - To perform operation on each element:
      - If list empty: quit
      - Perform operation on first element and recur on remainder of list
  - `Cons` constructs lists
    - As does `append`

Nested Lists

```
> (cons '(1 2) '(3 4))
((1 2) 3 4)
> (append '(1 2) '(3 4))
(1 2 3 4)
```



```
> (length '((1 2) 3 (1 (4 (5)))))
3
```

Binding and Defining

- Functional languages should avoid side effects
- `set` binds in the global environment
  - Or in most global context
  - `setq` and `setf` as well
- Local variables defined by `let`
  - Bounds function body

```
(let (<local-variables>) <expressions>)

<local-variables> ::
  (<symbol><expression>)
```