A Few Comments on AI

- Knowledge Representation and Search
- If a computer only has the right knowledge, representation of that knowledge, and way of indexing that knowledge then it can be intelligent

Does this idea work?

Physical Symbol Hypothesis

- Intelligent activity in either human or machine requires [Newell & Simon]:
  - Symbol patterns to represent significant aspects of a problem domain
  - Operations on these patterns to generate potential solutions to problems
  - Search to select a solution from among these possibilities

Representation Schemes

- Schemes should be [page 36 in Luger]:
  - Expressive: The scheme must be adequate to express all necessary information
  - Efficient: Support efficient execution of the resulting code
  - Natural: Provide a natural scheme for expressing the required knowledge

Where a good representation is available, the solution to the problem may be easy.

Sometimes it’s hard to have a good representation – just imagine a representation for commonsense knowledge in a natural language system!

Representation may make a big difference in a programming language.

An example

- Task Description
  - To write a program that finds, for a given phone number, all possible encodings by words, and prints them. A phone number is an arbitrary(!) string of dashes -, slashes / and digits. The dashes and slashes will not be encoded. The words are taken from a dictionary which is given as an alphabetically sorted ASCII file (one word per line).
- Participants: 14 programmers (ave. experience: ~7 yr)
- Biggest experimental flaw: subjects self selected

Mapping

- The following mapping from letters to digits is given:
  
<table>
<thead>
<tr>
<th>E</th>
<th>J</th>
<th>N</th>
<th>Q</th>
<th>R</th>
<th>W</th>
<th>X</th>
<th>D</th>
<th>S</th>
<th>Y</th>
<th>F</th>
<th>T</th>
<th>A</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>j</td>
<td>n</td>
<td>q</td>
<td>r</td>
<td>w</td>
<td>x</td>
<td>d</td>
<td>s</td>
<td>y</td>
<td>f</td>
<td>t</td>
<td>a</td>
<td>m</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>I</td>
<td>V</td>
<td>B</td>
<td>K</td>
<td>U</td>
<td>L</td>
<td>O</td>
<td>P</td>
<td>G</td>
<td>H</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>i</td>
<td>v</td>
<td>b</td>
<td>k</td>
<td>u</td>
<td>l</td>
<td>o</td>
<td>p</td>
<td>g</td>
<td>h</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td></td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
The Results

- Using Lisp
  - Time (hr): 2 to 8.5, ave: 5
  - Lines of Code: 51 to 182
  - Run Time (median): 30 seconds
- Using C/C++
  - Time (hr): 3 to 25, ave: 11
  - Lines of Code: 107 to 614, ave: 277
  - Run Time (median): 54 seconds

  [Quickest C/C++ program ran faster than the quickest Lisp program]

Lisp

- Stands for List Processing
- Developed by John McCarthy in late 1950’s
- Strengths include
  - List processing
  - Symbol manipulation
  - Data and programs have the same syntactic form, so functions can be created and run while the program is executing

A show of hands

- How many have used Lisp?

Learning Lisp

- See my web site for on-line tutorials and information
- [handout given in class]

The Basics

- “No doubt about it, Common Lisp is a big language.” - Guy Steele
  - 622 built-in functions (in one pre-ANSI CL)
  - 86 macros
  - 27 special forms
  - 54 variables
  - 62 constants
- C++ has around 48 reserved words
- Lisp includes the kitchen sink!

But…

- We won’t always be using Lisp…
  - We’ll be using emacs Lisp (eLisp)
  - If you have emacs on your computer, then you have eLisp
- We will also use CMUCL
  - CMU Common Lisp
If you’re writing an interpreter…

- Why use Lisp?

Abuse Uses Lisp as its Scripting Language

Richard Gabriel

- The story behind “worse is better”

To Do

- Look @ article
- Lisp and side effects (example of side effect – printing as it also returns a value)
- Higher order functions
- Dynamic and lexical scoping
- Currying
- closure

Example Code

(defun rest (a-list)
  "rest gives you all of the list, but the first element."
  (cdr a-list))

More Examples

- Check out Lisp in the Great Language Shootout:
  - http://www.bagley.org/~doug/shootout/

- What’s wrong with these examples?
The von Neumann Architecture

- Otherwise known as the load/store architecture
  - Programs achieve their effect by changing the contents of a store location

- Are other architectures possible?

---

Functional Programming

- Programs achieve their effects by evaluating expressions
  - Why?
    - There are no side effects in purely functional programming
    - It's much easier to prove correctness
    - It's much easier to parallelize – different expressions are done on different processors

---

Lisp

- Is it purely functional?
- Why or why not?

---

Lisp?

- Lisp stands for "Lots of Infuriating Superfluous Parentheses"
- Reliability with functional language use?

---

Why Lisp?

- See the article on:
  - [http://www.paulgraham.com/icad.html](http://www.paulgraham.com/icad.html)

---

Lisp

- Lisp is a dynamically typed language
  - This leads to greater flexibility at the cost of less error catching during compile time
  - Since we know less type checking is done, we can check it ourselves with predicates
    - eLisp has around 30 predicates for type checking
    - Some common predicates: symbolp, atom, listp, floatp, function,
Lisp - variables

- All variables declared dynamically with a (setq var-name var-value) syntax will be **global** variables
- Make sure you declare your variables to be local using:
  - `let`
  - `let*`

Functions

- In Lisp
  - Functions may be used as arguments
    - `mapcar` is a good example of this
      - Example: `(mapcar `+ `(1 2 3 4) `(1 2 3 4))`

S-expressions

- S-expression: a "symbolic expression" and it can be anything from an atom to a list to a function
- Examples:
  - `(+ my-num 1)`
  - `(@(student 0)`
  - `my-var`

Lambda Expressions

- An s-expression whose first element is "lambda" is a lambda expression. Lambda expressions are like anonymous functions
  - Examples
    - `(lambda (x) (* x 10))`
    - `((lambda (x y) (+ x y)) 2 3)`

Scope Differences

- What does this code do?
  ```lisp
  (defun base (f a)
    ((lambda (f a)
       (+ (funcall f 10) a)
     ) f a))

  (defun two-digit (a b) ((lambda (a b)
    (base `(lambda (c) (* a c)) b)
     a b)
  )
  (two-digit 2 3)
  ```

Scope

- There are two kinds of scope here:
  - Lexical scope – what CMUCL uses and what we’re used to
  - Dynamic scope – what eLisp uses
Other Languages

- FP: purely functional and not meant to be used as anything but a research language
- Scheme: a dialect of Lisp
- ML: uses static type checking, strongly typed, but if the compiler can figure out the type, then you don't have to put it

ML Pattern Matching

- Matches on the right function for the right answer

```ml
fun isMember( x : int, nil : int list ) : bool = false
| isMember( x : int, first :: rest : int list ) : bool =
  if x = first then true
  else isMember( x, rest );
```

Lazy Evaluation

- In both ML and Lisp, parameters are called by value. It is sometimes useful for parameters to remain unevaluated until they're required.
- Infinite data structures become possible with lazy evaluation