
AI

Artificial Intelligence

8.1.2
Programming Languages for AI

Programming Languages

- We will be using Python in this class.
- However, the first examples will involve Lisp.
- We will **not** be using other AI programming languages such as Haskell or Prolog though you should check them out.
- Why not Java/C++?

Introduction to Lisp

- LISP = LISt Processing
- In Lisp, nearly everything is a function. Even the mathematical operators.

```
(+ (* 2 3) 1)
```

- `setq` is used to set variables.

```
(setq var 32)
(setq str "Connecticut")
(setq lst '(1 2 3))
```

Introduction to Lisp

- The three types of data here are numbers, strings, and lists.
- Lisp, unlike Java, is dynamically typed.
- The ' in list statement is called the quote operator, and tells Lisp that the input is a list, and not to interpret it as a function.

Introduction to Lisp

- `setq` sets the variable globally. To create a local variable (e.g. inside a function), use the let function.

```
(let
   ((a 1)
    (b 2))
   ... )
```

The variables `a` and `b` will only be defined within `let` parentheses.

Introduction to Lisp

- To comment in Lisp, prefix the line with ;

```
; This is a comment!
```

Introduction to Lisp

- The *if* statement is a bit different from other programming languages.

```
(if (< 2 3)
   (... true ...)
   (... false ...))
```

- Lisp executes the first block of code if the conditional statement is true. The second statement (which is optional) serves as the *else* statement.

Introduction to Lisp

- If you want to execute multiple functions in the *if* statement (which is common) use the `progn` function, which serves to group multiple functions together.

```
(if (> 3 4)
  (progn
    (setq x (+ x 1))
    (setq y x))
  (setq x 0))
```

Introduction to Lisp

- If you want an if-elsif-else statement, then you'd want to use the `cond` function

```
(cond
  ((> x 1) (setq y 1))
  ((< x 1) (setq y 2))
  (t (setq y 0)))
```

Introduction to Lisp

- For Boolean values, `t` represents true, and `nil` represents false. Lisp treats an empty list `'()` (or `nil`) as false and all other inputs as true.
- This is a convenient feature of the language to know. For instance, to do something only if a list is not empty, the following two chunks of code are identical.

```
(if (> (length lst) 0)
  (...))

(if lst
  (...))
```

Introduction to Lisp

- While loops are accomplished in the following manner:

```
(loop while (> n 0)
      (setq n (- n 1)))
```

- Although for the most part, recursion is the more popular way to accomplish loops.

Introduction to Lisp

- Lists are Lisp's most fundamental data structure.

```
(setq lst '(1 2 3))
```

- To get the first item from the list, use the `first` function. To get the rest of the items, use `rest`. These are historically known as `car` and `cdr`.

```
(first lst) => 1
(rest lst) => (2 3)
```

Introduction to Lisp

- Lisp provides some helpful shortcuts to access other items in the list as well.

```
(second lst) => 2
(third lst) => 3
(fourth lst) => nil
```

- You could access these elements without these functions through repeatedly using `first` and `rest`.

Introduction to Lisp

- To add an item to the beginning of the list, use the `cons` function. `cons` returns a new list with the element prefixed to the beginning of the list.

```
(cons 0 lst) => (0 1 2 3)
```

Introduction to Lisp

- `defun` is used to define functions.

```
(defun square (x)
  (* x x))
(defun add (x y)
  (+ x y))
```

- Lisp implicitly returns the value of the last statement in a function.

```
(square 9) => 81
(add 2 4) => 6
```

Introduction to Lisp

- Recursion is very prevalent in Lisp. Below is an example of a recursive sum function which uses both `first` and `rest` in a recursive context.

```
(defun sum (lst)
  (if (not lst)
    0
    (+ (first lst) (sum (rest lst)))))
```

Computing factorials

```
(defun factorial (n)
  "Compute the factorial of N."
  (if (= n 1) 1
      (* n (factorial (- n 1)))))
```

Introduction to Lisp

- One of Lisp's most powerful features is the ability to pass functions to other functions. Most of these functions that take advantage of this feature take two arguments, a function and a list.

- `mapcar (map)`. Returns the list of the results from applying the function to each of the items in the list. The following example returns a new list with all the elements squared.

```
(mapcar 'square '(1 2 3 4 5)) => '(1 4 9 16 25)
```

Introduction to Lisp

- `remove-if`. Removes items from the list if the item, when plugged into the function, returns true. The following example returns a new list with all the odd numbers removed.

```
(remove-if 'oddp '(1 2 3 4 5)) => '(2 4)
```

- Note: Built-in functions in Lisp that end in a 'p' are predicates and return a Boolean value.
- `reduce`. Reduces a list to a single value by applying the function to each of the items. The following example is equivalent to the sum function.

```
(reduce '+ '(1 2 3 4 5)) => 15
```

Introduction to Lisp

- Writing your own function to take in a function is not hard. Below is an example of how you could implement your own `mapcar`.

```
(defun my-mapcar (fn lst)
  (if lst
    (cons (funcall fn (first lst))
          (my-mapcar fn (rest lst)))))
```

- `funcall` is used to run functions that are stored in variables.

Introduction to Lisp

- Anonymous functions: it's occasionally useful (particularly with the higher-order functions mentioned earlier) to create a function without a name, typically because it is only getting used once.
- For instance, say you wanted to double all the elements in a list. A function to double a number would rarely get used outside this call, so this is a good opportunity to create an anonymous function. The following two chunks of code are equivalent.

```
(defun double (x) (* x 2))
(mapcar 'double '(1 2 3 4 5)) => '(1 4 6 8 10)
(mapcar (lambda (x) (* x 2)) '(1 2 3 4 5)) => '(1 4 6 8 10)
```

- It'll be a judgment call whether to go with the brevity of an anonymous function or the readability afforded by naming the function.

Introduction to Lisp

- The print function can be used for basic output.

```
(print 512)
```

- For more complicated printing Lisp has the `format` function, which is analogous to the `printf` function in C. The basic structure is

```
(format t "~a ~a of beer on the wall.~%" 99 "bottles")
```

- The `t` argument means to print to the standard output, `~a` says to replace with the variable, and `~%` means newline.

Lisp tutorials

- External:
  - http://cs.gmu.edu/~sean/lisp/LispTutorial.html
  - https://www.cs.utexas.edu/~mtimkvch/lisp_tutorial.html
- Many more listed on the course page

Introduction to Python

- On your own
- Some external sites:
  - https://www.coursera.org/learn/python
  - https://www.coursera.org/specializations/python
  - http://www.tutorialspoint.com/python/
  - http://www.learnpython.org/

Comparing Lisp and Python

- Peter Norvig:
  - *Python*/**Lisp** is an interpreted **and compiled**, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together.
  - *Python*/**Lisp**'s simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.
  - *Python*/**Lisp** supports *modules and* packages, which encourages program modularity and code reuse.
  - The *Python*/**Lisp** interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.
  - Often, programmers fall in love with *Python*/**Lisp** because of the increased productivity it provides. Since there is no **separate** compilation step, the edit-test-debug cycle is incredibly fast.
  - Debugging *Python*/**Lisp** programs is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the program doesn't catch the exception, the interpreter prints a stack trace. A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on. The debugger is written in *Python*/**Lisp** itself, testifying to *Python*/**Lisp**'s introspective power. On the other hand, often the quickest way to debug a program is to add a few print statements to the source: the fast edit-test-debug cycle makes this simple approach very effective. “
- http://norvig.com/python-lisp.html

Sample program in Lisp (1/2)

```
(defparameter *grammar*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Noun))
    (verb-phrase -> (Verb noun-phrase))
    (Article -> the a)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked))
  "A grammar for a trivial subset of English.")

(defun random-elt (list)
  (elt list
       (random (length list))))

(defun generate (phrase)
  "Generate a random sentence or phrase"
  (cond ((listp phrase)
         (mappend #'generate phrase))
        ((rewrites phrase)
         (generate (random-elt (rewrites phrase))))
        (t (list phrase))))
```

grammar.lisp

Sample program in Lisp (2/2)

```
(defun generate-tree (phrase)
  "Generate a random sentence or phrase, with a complete parse tree."
  (cond ((listp phrase)
         (mapcar #'generate-tree phrase))
        ((rewrites phrase)
         (cons phrase
               (generate-tree (random-elt (rewrites phrase)))))
        (t (list phrase))))

(defun mappend (fn list)
  "Append the results of calling fn on each element of list.
  Like mapcon, but uses append instead of nconc."
  (apply #'append (mapcar fn list)))

(defun rule-rhs (rule)
  "The right hand side of a rule."
  (rest (rest rule)))

(defun rewrites (category)
  "Return a list of the possible rewrites for this category."
  (rule-rhs (assoc category *grammar*)))

(generate 'sentence)
```

Sample program in Python (1/2)

```
"""Generate random sentences from a grammar.  The grammar consists of entries that can be
written as S = 'NP VP | S and S', which gets translated to {'S': [['NP', 'VP'], ['S', 'and',
'S']]}, and means that one of the top-level lists will be chosen at random, and then each
element of the second-level list will be rewritten; if a symbol is not in the grammar it
rewrites as itself.   The functions generate and generate_tree generate a string and tree
representation, respectively, of a random sentence."""

import random

def Grammar(**grammar):
  "Create a dictionary mapping symbols to alternatives."
  for (cat, rhs) in grammar.items():
    grammar[cat] = [alt.split() for alt in rhs.split('|')]
  return grammar

grammar = Grammar(
  S  = 'NP VP',
  NP = 'Art N',
  VP = 'V NP',
  Art= 'the | a',
  N  = 'man | ball | woman | table',
  V  = 'hit | took | saw | liked'
  )
```

grammar.py

Sample program in Python (2/2)

```
def generate(symbol='S'):
  "Replace symbol with a random entry in grammar (recursively); join into a string."
  if symbol not in grammar:
    return symbol
  else:
    return ' '.join(map(generate, random.choice(grammar[symbol])))

def generate_tree(symbol='S'):
  "Replace symbol with a random entry in grammar (recursively); return a tree."
  if symbol not in grammar:
    return symbol
  else:
    return {symbol: map(generate_tree, random.choice(grammar[symbol]))}

if __name__ == "__main__":
    import sys

    sent = generate()
    print sent
```

Lisp Links

- https://www.youtube.com/watch?v=M-BFgErib4k
- https://www.cs.sfu.ca/CourseCentral/310/pwfong/Lisp/1/tutorial1.html
- http://www.gigamonkeys.com/book/
- https://www.tutorialspoint.com/lisp/


AI