CSC 530 Lecture Notes Week 1

Introduction to the Course

Introduction to Lisp

I. **Meaning.**

A. We will focus upon the *meaning* of programming languages

B. E.g., what does it mean to be

   1. *functional*?

   2. *strongly typed*?

   3. *object oriented*?

   4. *more powerful?*

   5. *evil and dangerous*?

What does it mean, cont’d

C. We must investigate how meaning can be expressed.

1. Formal semantics
2. Define semantics like BNF defines syntax.
3. Concise and formal, without ambiguity.
4. There are a number of approaches

II. **How is meaning defined?**

A. I.e., in general, how’s it done?

B. For English:

1. In a dictionary
2. Anthropologically
3. Structurally

Meaning defined, cont’d

C. For programming languages, use similar techniques.

  1. A compiler or interpreter.

  2. Historically.

  3. With formal definitions.

Meaning defined, cont’d

D. Forms of semantic definition:

1. Operational semantics
2. Attribute grammars
3. Denotational semantics
4. Axiomatics semantics
5. Algebraic semantics

III. **Programming lang’s as religion.**

A. Computer scientists are fond of heated discussion.

B. Debate is relly moot.

C. Despite what they *know*, they debate what we *believe*.

D. We will join the debate in this class.

IV. **This class’ belief system**

A. applicative (aka, functional)

B. "opposing" viewpoints given fair treatment.

C. Other aspects include: ...

V. **Some initial definitions.**

A. *applicative language* = side-effect free

B. *imperative language* = instructions modify state memory.

VI. **Foundations**

A. Distinction between applicative and imperative is fundamental

B. To examine fully, we'll go back to pre-history of computing.

VII. **Turing machines and the imperative model**

A. Founders: Alan Turing, John von Neumann, and others.

B. A TM is a model of *effective computability*

C. Formally, TM is a state machine:

Turing machines, cont’d

1. Infinite memory tape

2. Movable head, that performs

    a. Read a symbol

    b. Write a symbol

    c. Move one slot

Turing machines, cont’d

D. A set of quintuples

```
(current state,
 symbol read,
 new state,
 symbol written,
 move direction)
```

Turing machines, cont’d

A very simple example (compute the unary constant 4):

```
(0, ,1,1,R)
(1, ,2,1,R)
(2, ,3,1,R)
(3, ,4,1,R)
```

Turing machines, cont’d

E. Another example

```
(0,1,1,X,R)
(0,,,0,,,R)
(0,:,3,:,R)

(1,1,1,1,R)
(1,,, 1,,,R)
(1,:, 1,:,R)
(1, , 2,1,L)

(2,1,2,1,L)
(2,:,2,:,L)
(2,,,2,:,L)
(2,X,0,X,R)
```

Turing machines, cont’d

1. A sample input tape (to add 2+3):

```
11,111:
^
0
```

2. The resulting output tape:

```
XX,XXX:11111
       ^
       3
```

Turing machines, cont’d

3. What each state does

State	**Description**
0	check for 1, ’,’, or ’:’ ...
1	carry a 1 over ...
2	go back ...
3	halt

VIII. **Recursive function theory and the applicative model**

A. Founders: Stephen Kleene, Alonso Church, and others.

B. Alternative (and equivalent) model of effective computability

C. Formally, defined as:

RFT, cont’d

1. *Zero* function: $Z(x) = 0$

2. *Successor* func’n: $S(x) = x + 1$

3. *Composition* of functions:

$$f(x_0,\ldots,x_n) = h(g_0(x_0,\ldots,x_n),\ldots, g_k(x_0,\ldots,x_n))$$

RFT, cont'd

4. inductive recursion scheme:

$$f(0,x_1,\ldots,x_n) = g(x_1,\ldots,x_n)$$

$$f(S(n),x_1,\ldots,x_n) = h(f(x_1,\ldots,x_n),n, x_1,\ldots,x_n)$$

where g and h are defined recursively

RFT, cont’d

D. RF def constant 4:

```
Four(x) = S(S(S(S(Z(x)))))
```

E. Definition of addition (second TM example):

```
Add(0,y) = y

Add(S(n),y) = S(Add(n,y))
```

IX. **Equivalence of TMs and RFT**

A. Can be proved formally

B. An important (and comforting) result

C. Equivalent, but also equivalently unsuited for practical programming.

D. What is important is what the models represent

Equivalence, cont’d

E. The *Churh Hypothesis*

1. TM’s and RFT each capture *essense* of effective computability.

2. No devisable system is fundamentally more powerful.

3. Hypothesis is unprovable, but generally believed by all.

X. **Practical comparison**

A. In the TM model:

1. Computation defined by a sequence of instructions
2. Data stored in equential memory, which changes state
3. Computation carried out executing instructions sequentially

Practical comparison, cont’d

B. In the RFT model:

1. Computation defined by a set of functions
2. Data passed as parameters and returned as values
3. Computation carried out by invoking functions

Practical comparison, cont’d

C. Summary:

1. The TM model is the fundamental basis for imperative languages.
2. The RFT model is the fundamental basis for applicative languages.

XI. **Compelling motivations for applicative programming**

A. Concurrency

B. Verifiability

C. Referential transparency

D. We’ll discuss in upcoming lectures.

XII. **A question for the applicative zealot**

A. *If the advantages of applicative languages are so compelling, why is their use not more widespread?*

A question, cont’d

B. Answer 1: Programmers are inherently lazy and weak-willed.

C. Answer 2: Present-day hardware isn’t any good.

D. Answer 3: We are at an unhappy point in the natural evolution of programming languages.

We now proceed to examine applicative languages in detail, beginning with Pure Lisp.

XIII. **Assignmentless programming**

A. Take your favorite imperative programming language and throw out assignment statements.

B. Such represents the essence of applicative programming.

C. Fundamental tenet of applicative programming is that *data do not change*

Assignmentless programming, cont'd

D. An applicative language cannot be constructed simply by removing assignment statements from some imperative language.

XIV. **The necessary evil of imperative constructs**

A. Few real languages are completely applicative.

B. Languages are *primarily* one category or the other.

C. We begin our study from a pure standpoint.

D. Subsequently, we will see how imperative features can fit into an applicative framework.

XV. **Motivation for Pure Lisp**

A. Defines most fundamental aspects in simple and elegant way.

B. Useful to introduce purely applicative programming.

C. Also useful to describe operational semantics.

D. Good tool for rapid prototyping of translators.

XVI. **General features of Pure Lisp**

A. Syntax difference; profoundly unimportant.

B. Lisp is *untyped*.

C. Lisp is an *expression language*.

D. Overall style is recursive, not iterative.

E. Lisp is built on simple and orthogonal primitives.

XVII. **The function definition**

A. A simple example

```
(defun APlusB (a b)
    (+ a b)
)
```

B. The equivalent in C:

```
int APlusB(int a,b) {
    return a + b;
}
```

Function definition, cont’d

C. Observations

1. Basic concept same in Lisp as in C.

2. Note again that Lisp is *untyped*.

3. *All* expressions in prefix notation.

4. Lack of a return statement in Lisp.

XVIII. **cond**

A. Comparable to if-then-elsif-else

B. General form:

(cond
  ((*test-expression*$_1$)
    (*test-expression*$_1$))
    . . .
  ((*test-expression*$_n$)
    (*test-expression*$_n$))

C. Takes some getting used to

XIX. **The heterogeneous list**

A. A collection or zero more elements.

B. Precise definition ...

C. Fundamental ops: car, cdr, cons.

D. Fundamental relationships:

   - (car (cons X Y)) = X
   - (cdr (cons X Y)) = Y

XX. **`quote`**

A. There is an interesting potential problem

B. No syntactic distinction between function invocation and a list datum.

1. E.g, consider

```
(defun f (x) ... )
(defun a (x) ...)
```

`quote` cont'd

2. What does the following represent?

    `(f (a b))`

3. Is it

    a. A call to `f` with the list argment `(a b)`?

    b. A call to `f`, with argument that is call to `a`?

`quote` cont'd

4. The answer is (b).

5. Default meaning for a list is a function call.

6. To obtain the alternate meaning (a), we must use *quote*.

7. I.e.,

```
(f ’(a b))
```

XXI. **Iteration through recursion**

A. In applicative languages, iterative control replaced by recursion.

B. E.g.,

Recursion, cont’d

1. **Lisp:**

```
(defun avg (l)
    (/ (sum l) (length l))
)
(defun sum (l)
    (cond ((null l) 0)
          (t (+ (car l)
                (sum (cdr l))))
    )
)
(defun main ()
    (avg ’(1 2 3 4 5))
)
```

Recursion, cont’d

2. **C:**

```
int avg(int l[], int length) {
    int i, sum;
    for (i=0, sum=0; i<length; i++
        sum += l[i];
    return sum/length;
}

main() {
    int l[] = {1,2,3,4,5};
    printf("%d0, avg(l, 5));
}
```

Recursion, cont’d

C. Observations ...

1. Lisp uses *tail recursion*.

2. Transliteration into C

```
int sum(list l) {
    if (null(l))
        return 0;
    else
        return car(l) +
            sum(cdr(l));
}
```

XXII. **Another list-processing example**

A. Many functions in real Lisp environments.

B. *Any* can be built using the three primitives.

C. E.g.,

```
(defun my-nth (n l)
    (cond ( (< n 0) nil )
          ( (eq n 0) (car l) )
          ( t (my-nth (- n 1)
                   (cdr l)) )
    )
)
```