MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1996

Lecture Notes – Sept. 5, 1996

Introduction and Scheme Basics

In today's lecture, we will do three things:

- Administrivia
- Overview of the course
- LISP/Scheme

Administrivia

The General Information handout contains most of the information you need to know about how the course is run. Please note that the course secretary's office is NE43-711. This is where you should go if you need to switch section assignments, and where you should turn in your forms for assigning recitations, if you should choose to do so after class (though we would REALLY prefer them at the end of class today).

There is a 6.001 Web site, which can be accessed at:

http://www-eecs.mit.edu/class-materials.html

Note that the Instrument Desk is available for the distribution of course notes only between the hours of 1:00pm and 5:00pm, Mondays to Fridays, and between 6:45pm and 8:45pm, Mondays to Fridays.

Overview

Computer science is not really about computers, but rather is primarily focused on the concept of *processes*, and on ways to describe them.

Processes deal with *Imperative* or "How to" knowledge, as opposed to *Declarative*, or "What is" knowledge.

In talking about "how to" knowledge, we will distinguish between a process, which is *actual mechanism by which a computation is executed* and a procedure, which is *a means of describing a process.*

Our need to describe processes leads to a need for a language in which to execute that description, with its own vocabulary and rules of syntax and semantics. In this course we will be using a dialect of LISP, called Scheme, for that purpose.

Once we have developed this language for describing processes, we will want to use it to describe large and complex processes. To do this, we will need some solid engineering tools for controlling complexity. The three key themes we will explore for controlling complexity are

- black box abstraction
- conventional interfaces
- meta-linguistic abstraction

Basic Scheme

The remainder of today's lecture will focus on basic components of the LISP programming language, especially the syntax and semantics of simple expressions in LISP.

Every LISP program is constructed from *expressions*, each of which has a *syntax* (rules of legal formation) and a *semantics* (rules for deducing meaning).

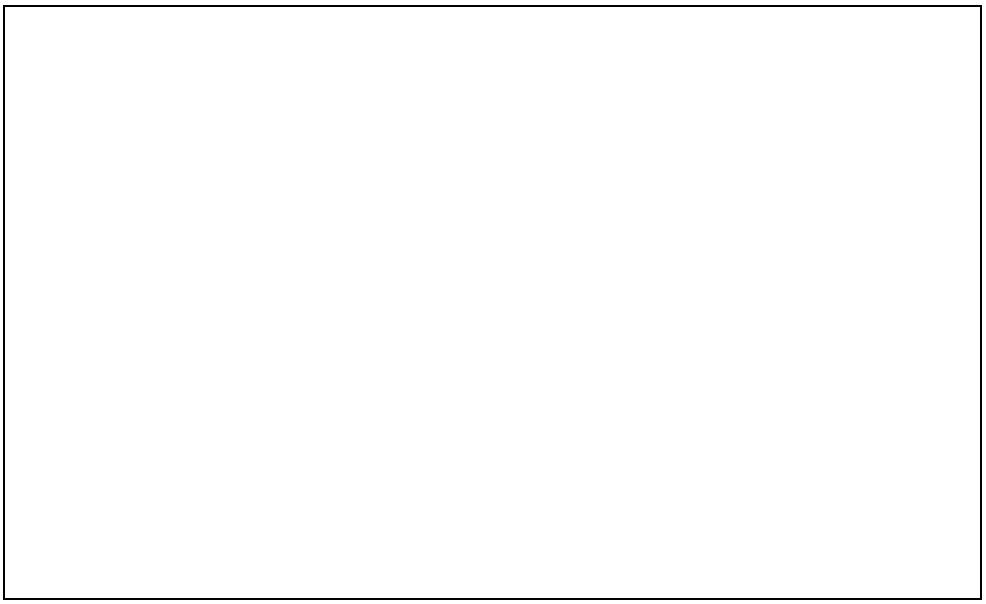
The three main components of any language are

- primitives
- means of combination
- and means of abstraction.

In the space below, fill in examples of the different types of Lisp expressions:

	Syntax	Semantics
Primitives		
Combinations		
Abstractions		

Below is some scratch space for including examples:

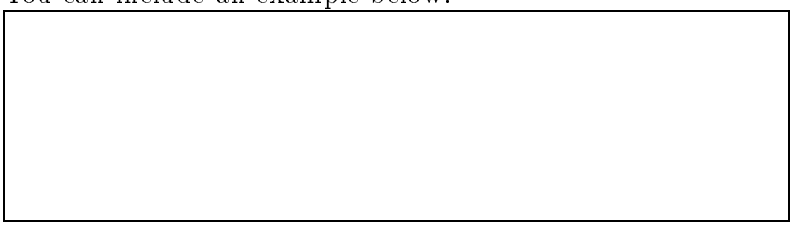


RULES FOR EVALUATION:

- the value of a numeral is *itself*
- the value of a primitive operation is *a pointer to the internal machine instructions to accomplish it*
- the value of a name is *the value associated with that name in an environment*
- the value of a combination is obtained by:
  - Evaluating *the subexpressions in any order*
  - Applying the value of the *operator* subexpression to the values of the other subexpressions, where applying a compound procedure means *evaluating the body of the procedure* with each formal parameter replaced by is corresponding value.

Scheme's fundamental procedure maker is a *lambda* expression, which consists of *a set of formal parameters and a body.*

You can include an example below:



To trace the evaluation of expressions that use such constructed procedures, we can use the set rules of evaluation above, amplified to note that substitution of formal parameters does not cross the barriers of a `lambda` expression.

An example:

```
(define square (lambda (x) (* x x)))

(define sum-of-squares
   (lambda (x y)
      (+ (square x) (square y))))

(define f (lambda (a)
            (diff-of-squares (+ a 2) (* a 3))))
```