CSCE 476/876 Spring 2015
Lisp Tutorial #1*

Constraint Systems Laboratory
University of Nebraska-Lincoln

January 26, 2015

1 Entering Lisp Expressions in the Lisp Listener

This tutorial will take you through some of the basics of the Lisp language. Everything covered herein can be found within the first 3 chapters of the Winston/Horn LISP book. You should run these commands interactively through emacs.

1.1 Basic Numeric Functions.

Try to predict the results of the Lisp expressions below and then run them to see if you are correct:

1. (+ 2 3)
2. (+ 1 2 3 4)
3. (* 4 (+ 2 2))
4. (- 10 5)
5. (- 10 5 5 5)
6. (/ 4 3)
7. (/ 4.0 3.0)

Note what was done with the third expression. We have embedded all call to the + function in the call to the * function. The value from + is fed as data straight into the * function. Expect to use this feature a lot. With expression 5, see how every argument after the first is subtracted from the first. We could have just as easily written

*Prepared by previous GTAs of this course: Jamie Schirf, Yaling Zheng, Nick Zielinski.

```
(- 10 (+ 5 5 5))
```

to get the same effect. Finally, pay particular attention to the expressions 6 and 7, since they can be tricky. The type of result returned by the / function depends on its arguments. If integers are passed, and they do not divide 'cleanly,' a fraction is returned. If the result of the division is an integer, an integer is returned. However, if either argument is a floating point number such as 2.0 or 3.14, the result will always be a floating point number.

Type the expressions and watch the returnd values:

1. (float (/ 4 3))
2. (round (/ 4 3))

The Lisp function `round`, like some other Lisp functions, returns two values. Type the following and watch how you can 'access' these values.

```
(setf x (round (/ 4 3))
(pprint x)

(multiple-value-bind (x y)
    (round (/ 4 3))
  (pprint y))
```

Within the 'environment' of the `multiple-value-bind` the variable `y` stores the second the value returned by `round`. If you want to write a function that returns multiple values, you should use, at the end of your function, the function `values` and give it as arguments all the variables whose values you want your function to return.

1.2 Basic List Handling

One of Lisp's greatest 'advantages' is its natural handling of lists. Try running the following expressions to see what they return:

1. (first '(1 2 3 4))
2. (rest '(1 2 3 4))
3. (first ())
4. (rest ())
5. (rest (first '(1 2 3))

As you can see, first and rest are basically complementary to each other. Given a list of elements, first returns the first element. Last returns a list of all elements except for the first.

In cases where there is nothing to return, such as when used on an empty list, an empty list, or NIL, is returned.

One thing you might ask is why that single quote (') precedes some of the lists. To answer this, try entering the first expression without it: `(first '(1 2 3 4))`.

You should get back something like this:

```
[2] CL-USER(17): (first (1 2 3 4))
Error: Funcall of 1 which is a non-function.
  [condition type: TYPE-ERROR]

Restart actions (select using :continue):
 0: Return to Debug Level 2 (an "abort" restart).
 1: Return to Debug Level 1 (an "abort" restart).
 2: Return to Top Level (an "abort" restart).
 3: Abort entirely from this (lisp) process.
[3] CL-USER(18):
```

The first line shows the offending command. The second line is the actual error. It says that it tried to call a function 1 but failed because 1 is not a function.

Lisp evaluates the first element of a list as a function. We use the quotation mark to suppress this behavior. When we use the quotation mark here, we make it clear that the 1 is data, and not to be interpreted as a function.

To escape the above error, type :pop or :res, which respectively pop you out of the break stack or bring you out of it, returning you to the regular LISP listener.

1.3 Variables

Of course, like any full featured language, Lisp allows us to bind values to names, creating variables. We do this with the setf special form. Type the following series of statements:

```
(setf brightred '(255 0 0))
brightred
(setf redvalue (first brightred))
redvalue
(setf greenvalue (first (rest brightred)))
greenvalue
```

Here we start by creating a list of numbers, which we assign to the variable `brightred`. We then start extracting values from this list and assigning them to other variables.

Note that typing the variable names by themselves was not necessary, But I want you to see that if you do so, it shows you the value of that variable.

2 Storing your code in files

If you put your code in a file named `week4.lisp`, then you can first load your code into the lisp environment by the following command:

`(load "week4.lisp")` or `:ld week4.lisp`

Then you compile the file using the following command:

`(compile-file "week4.lisp")` or `:cl week4.lisp`

Some usefull functions to learn:

`mapcar`, `reduce`, `remove-if`, `apply`, `funcall`, `some`, `every`, `count-if`, `eval`

Open a file with a `.lisp` extension and use one or more of the above functions to write the functions below. Write your definitions in the buffer containing your file, save your code, go to the lisp listener (`*common-lisp*` buffer), and load your file to evaluate and test your code.

1. Define a function that takes a list and return the first three items and the last three items. For example, for the list `'(a b c this is a list 1 2 3)`, this function returns `'(a b c 1 2 3)`.

2. Given a list of lists, return the union of these lists. For example, for the list `'((1 2)(1 3)(1 5 6))`, this function returns `'(1 2 3 5 6)`. Do not use the CL primitive `union`.

3. Compute the summation of 1 through a specified positive integer.