Manual for the Koioslisp-13 Programming Language

Yash Tulsyan

February 8, 2013

Contents

1 Foreword **5**

2 A Tutorial **7**

2.1 Getting Started 7

Chapter 1

Foreword

Chapter 2

A Tutorial

2.1 Getting Started

The canonical first task is to write a program that prints out the words "`hello, world`" and a newline. The simplest way to do this is to go to a REPL and type in `(princ "hello, world \n")` and then hit return. As promised, it will print "`hello, world`" and a newline; `princ` prints the text given to it. This could be replicated by `(begin (princ "hello,") (princ "world \n"))`. However, some would argue that this doesn't count in that it is not a procedure that can be called. To remedy this:

```
KL-USER> (defun hello-world () (princ "hello, world \n"))
hello-world
KL-USER> (hello-world)
hello, world
```

Now it must be explained how this procedure was built. `defun` defines a procedure that can be called by its first argument, its name. The second arguments represents the arguments it takes in the form of a list. The remaining arguments are the function body. Note that the type of the function need not be specified–it can be declared for efficiency purposes, as we will cover later
It is often said that the canonical example focuses too much on input-output, and that several other examples should be given. One common example is a program to calculate the nth factorial number. It follows:

```
KL-USER> (defun my-factorial (n)
             "Returns the factorial of its argument"
             (cond ((< n 1) 0)
                   ((= n 1) 1)
                   ((> n 1) (* n (my-factorial (- n 1))))))
```

Two questions arise: What does the string do? and What does `cond` do? The string is a *docstring*, put in between the argument list and the body of `defun` if specified. Docstrings are not a mere convention: they are used by the built-in functions `documentation` and `apropos` to provide help for users who cannot remember the name of a function or perhaps what a function with a given name does. `cond` is a *conditional*, as implied by its name; it consists of several *clauses*, each consisting of a *test* and a *body*. The tests are evaluated sequentially; if any one clause evaluates to true, it evaluates the body of the clause and returns the value of the *body* as the value of the `cond` form. The "default" test for a `cond` form is `t`: because `t` is the canonical truth value, it *always* evaluates as true and can thus focus as a catch-all test.