Variables

Identifiers in Common Lisp are symbols. These can be either functions or variables.

Naming

Since Lisp is an old language, it is internally implemented with (mostly) all capital letters. Therefore, it is case insensitive, so that the following symbols all refer to the same entity.

```
foo
Foo
FOO
```

Other naming rules are very loose in comparison to other languages. There are very few symbols that can not be in a name. You may not use: **whitespace " ' ` , : \ |**, unless you want to escape these characters each time (and Lisp even provides the option of enclosing the entire sequence of these symbols between vertical bars). In addition, names may not be of a valid numeric form or consist solely of periods.

```
this\ is\|a\ valid\:name
this| is also a valid |name
4list
45.8x3
45.8\d3
8*9

;;invalid names
.
..
...
45.8d3
```

A convention among Lisp programmers is to take advantage of the availability of the - character to separate words in a name. For example, whereas a Java programmer might name a function helloWorld or hello_world, in Lisp it is common to name it hello-world.

There are some keywords (including NIL and some built in functions) that you are not able to redefine.

You cannot give two variables the same name. Doing so creates an error.

Variable Declarations

Lisp has options to declare variables in order to optimize performance, but generally it relies on implicit declarations. In this way, it is similar to Python. In Python the programmer never specifies a type for the variable, though the first use must be on the left hand side of an assignment statement or in a parameter listing. In Lisp, the first use will be as a parameter, or else in a special operator which acts as an assignment (for example: defun, let, defparameter(MACRO)). In contrast to python, variables generally are only on the left hand side once – their values do not change. This is not always true, though, since there are imperative constructs in lisp that enable this behavior, such as setf.

The following code prints out the sum of x and y, which is 6.

```
(defun foo (x)              ;define x
    (let ((y 2))            ;define y
       (format t (+ x y)))) ;use x and y

(foo 4)
```

Identifier Scoping

Lisp is statically scoped.

The basic rule is that there is a global scope, to which you can add values by defun, defvar, and defparameter. Functions have a local scope containing their parameters, which extends to the end of the function. Within a function, a **let** statement can define a nested scope.

```
(defvar x 5)

(defun foo (x)       ;function scope x = 4
   (let ((x 2))      ;inner scope's x = 2
      (format t x))  ;print 2
   (format t x))     ;print 4

(foo 4)
(format t x)         ;print 5
```

In the following complicated example, the variable **count** is initialized to zero, and the lambda function (described in Functions) is within the variable's scope. Since the **let** statement returns the reference to this function, the value for count cannot die, acting like a static variable, or else subsequent calls to the function (via the alias ***fn***) would be undefined.

defparameter is a macro that creates variables with global scope. This allows **funcall** on the next line to reference ***fn***.

```
(defparameter *fn* (let ((count 0)) #'(lambda() (setf count (1+count)))))

(funcall *fn*) => 1 ;accesses the value of count since the function is in scope
(funcall *fn*) => 2 ;accesses the value of count since the function is in scope

;(format t count)     ;not within the scope of count
```

In particular, this is not valid

```
(defun foo (y)
       (+ x y))

(let ((x 6)) (foo 4))
```

This behavior is like static variables in a function in C.

```c
int counter()
{
   static int count = 0;
   count ++;
   return count;
}

int main()
{
   printf("%d\n",counter());  //1
   printf("%d\n",counter());  //2
   //printf("%d\n",count);    //out of scope
}
```