

The University of Hong Kong
Department of Computer Science and Information Systems
Programming Methodology and Object Oriented Programming
for Software Engineering and Double Degree students (CSIS0396A)

Assignment 2

Assigned: Wednesday Mar 7, 2001

Deadline: Wednesday Apr 4, 2001, 5:00pm.

This is a programming assignment. For assignments that are submitted at or before deadline, submit your assignment by using the handin command. For assignments that are late for at most 48 hours, send a mail to the tutor directly. In the latter case, the marks will be multiplied by 0.8. Assignments that are more than 48 hours late are not accepted.

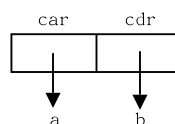
1. Functional requirement: Lisp engine prototype

Lisp (LISt Processing) is a very common programming language. **Lisp** and its derivatives are used in a lot of tools, including Emacs, xcin (the X Chinese input server), sawfish (the window manager for Gnome), and many other tools. In this assignment, you write a prototype **Lisp** interpreter that implements a small subset of **Lisp**, in a way that allows easy extensions. This allows you to practice writing abstract classes and deriving concrete classes from them. It also makes you familiar with the Unix programming development tools, including **make** and **CVS**.

1.1. Objects: Nouns in LISP

The interpreter handles 4 types of objects.

- **INTEGERS**. They represent a 32-bit signed integer value. Examples are 0, -5 and 1024. For ease of implementation, negative numbers cannot appear in a **Lisp** program.
- **SYMBOLS**. These are objects that consist of a string of characters other than whitespaces, periods and parentheses, with the restriction that it cannot begin with a digit. Symbols are case-sensitive. Examples include `List_Of_Silly_Parentheses`, `+1` and `cu2molo`.
- **NIL**. This means “nothing”. It is usually used to terminate lists (see Section 1.2).
- **CONS-CELLS**. These are objects that consist of two references. Traditionally the references are called the **car** and **cdr** part of the cons-cell. Each part refers to one **Lisp** object. We write it like `(a . b)`, which is the cons-cell referring to two objects a and b:

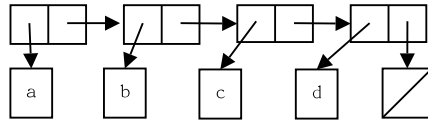


1.2. Lists: Noun phrases in LISP

Each part of a cons-cell can refer to another cons-cell. So cons-cells can be used to build lists. In **Lisp**, the **car** component of a cons-cell is the first element of the list, and the **cdr** component of a cons-cell is the remainder of the list. For example, the following is a list:

(a . (b . (c . (d . ())))

The list has the following structure:



It can be written like (a b c d). The Nil value is considered the empty list (). Since the car part of a cons-cell can refer to a list, we can have a list of lists. For example,

((a . (b . ())) . (c . (d . ())))

is a list ((a b) c d), containing 3 elements (a b), c and d.

The cdr part of a cons-cell need not be a cons-cell or a Nil. In this case it does not represent a list. We write it with the help of a "." part. For example, the cons-cell

(a . (b . (c . (d . e))))

can be written as (a b c d . e) (or (a . (b c d . e)), or (a b .(c d .e)), etc).

1.3. Forms: Verbs in LISP

The operations supported by **Lisp** are called forms. There are two types of forms, one called functions, the other called special-forms. Each form take some arguments and returns an object. Here are the supported **functions**:

- **Addition:** add up two or more arguments. All arguments should be integers. For example, if the arguments are 1, 2, 3 and 4, the function returns 10.
- **Subtraction:** subtract the second argument and all arguments following it from the first arguments. Again, all arguments should be integers. For example, if the arguments are 1, 2, 3 and 4, the function returns -8 (the result of 1 - 2 - 3 - 4).
- **Multiplication:** similar to addition, except that the operation is multiply. For example, if the arguments are 1, 2, 3 and 4, the function returns 24.
- **Division:** similar to subtraction, except that the operation is integer division. For example, if the arguments are 50, 2 and 3, the function returns 8.
- **Make-Cons:** accepts two arguments, and return a cons-cell referring to them. For example, if the arguments are a and b, it returns (a . b).
- **Find-Car:** accepts a cons-cell or Nil as argument, and returns the car part of the argument if it is a cons-cell. For example, if the argument is (a b c), then the function returns a. In case the argument is Nil, it returns Nil.
- **Find-Cdr:** similar to the Find-Car function, except that it returns the cdr part. For example, if the argument is (a b c), then the function returns (b c).
- **Make-List:** accepts any number of arguments, and returns a list of them. For example, if the arguments are (a . b), c and 5, then the function return the list ((a . b) c 5).
- **Set-Variable:** accepts a symbol and another object as arguments. It sets the object as the

value associated with the symbol (see the next section). For example, if the arguments are a and (b c), then the value associated with symbol a is set to (b c). The function returns the value set.

- **Set-Form:** accepts two symbols as arguments. It sets the function associated with the first symbol to be that associated with the second symbol (see the next section). For example, if the arguments are a and +, and + is originally associated with addition, then a is associated with addition as well. The function returns the second argument.

Special-forms differ from functions only in the way they interpret arguments (see Section 1.5). One special-form is supported:

- **Quote:** accepts at least one argument, and return the first of them.

1.4. Associations: Pronouns in LISP

Each symbol can be associated with a value and a form, which are set by the set-variable and set-form functions. There is one symbol which has a **constant value association**. It cannot be modified by set-variable.

- `nil`: the Nil object. There should be only one Nil object in the program.

Some symbols have **default form associations**, which can be changed using set-form.

- `quote`: associated with the quote special-form.
- `+`, `-`, `*`, `/`: associated with the addition, subtraction, multiplication and division functions respectively.
- `cons`, `list`: associated with the make-cons and make-list function.
- `car`, `cdr`: associated with the find-car and find-cdr functions respectively.
- `set`, `fset`: associated with the set-variable and set-form functions respectively.

1.5. Programs in LISP

A **Lisp** program is a sequence of objects, called expressions. The interpreter reads each of these expressions and construct the Lisp object(s) corresponding to the expression in the memory. After each expression is read, it is *evaluated* (see below), and the result is printed to **cout**.

When an object is **evaluated**, either a result is returned, or an error is triggered:

- If a symbol is evaluated, its associated value is returned.
- If a list is evaluated, the first element of the list must be a symbol with an associated form. If the associated form is a special-form, the remainder of the list is passed to the special-form directly as arguments. If the associated form is a function, each list remaining element is evaluated, and the results are passed as the arguments of the function. In either case, the value returned by the special-form or function is returned as the result of evaluation.
- If a cons-cell that does not represent a list is evaluated, an error is triggered.
- If anything else is evaluated, the object itself is returned.

Some objects can be written in more than one way. The interpreter always use the “standard” way: If the Nil object is printed, it is printed like (). If a cons cell is printed, it is printed in a way that minimizes the number of parenthesis required, and one space is inserted only between two list elements, or between an element and the “.” symbol. For example, the cons-cell “((a.(b)).(c.d))” is printed like “((a b) c . d)”.

If any of the operation fails, an error should be written to the standard error stream, and the evaluation of the whole expression stops.

The interpreter may generate objects during its evaluation. No attempts is needed to destroy these objects (i.e., it is okay for the interpreter to leak memory).

2. Example input and output

A sample run of the interpreter looks like this (input lines are in **bold**):

```
a
Error: Symbol's value as variable is void: a
(quote a b c)
a
nil
()
12
12
(set (quote a) 12)
12
a
12
(a 2)
Error: Symbol's function definition is void: a
(fset (quote a) (quote quote))
quote
(a 2)
2
(cons a (a ()))
(12)
(list a b c)
Error: Symbol's value as variable is void: b
(list a (quote b) (a c))
(12 b c)
(+ (* (- 5 12) a a) 12 5)
-991
(set (a c) (a (a (b . c) d . e)))
(a (b . c) d . e)
(car (cdr c))
(b . c)
(cdr (car c))
```

```
Error: Wrong argument type - not a list: a
(cdr nil)
()
(+ 2 c)
Error: Wrong argument type - not a number: (a (b . c) d . e)
(fset (a +) (a *))
*
(+ 3 4)
12
```

3. Extensibility requirements and Program structures

We expect that the program will be extended to handle new types like strings, vectors or maps; and new forms like setting the car part of a cons-cell, finding the square root of a number, performing a if-else construct, etc. The set of default associations will also be changed frequently.

The program should be **readily extensible**. To implement a new function, it should not require more than writing a new C++ source file and a new header file, adding one line of **#include** directive in the main source file, modifying one line of the main source file to make the default association, and modifying one line in the Makefile. To add support for a new object type, we should only require similar changes, plus to modify the *input parser*, the function or object that parse the **Lisp** program.

Separate source files should be used for independent functions and object types, and each source file should be separately compiled using Makefile. The whole project thus consists of some C++ source files, some header files and a Makefile. It should also has a file **test.lisp** that contains a **Lisp** program and illustrate the capability of the interpreter, and another file **README** (see below).

You should hand-in a gzipped tar archive of the CVS repository containing only the above listed files (plus perhaps a .cvsignore file). The README file may specify the name of a branch that contains the most “stable” version of the program (i.e., contains the least bugs), while the main trunk should contains the version with the most functionality. This is to give you a chance if a bug in an advanced features make some of the simpler features to fail. The file should also contain a description of the supported functions of both versions, and any known problems.

4. Design and implementation hints

Big hints

The program repeatedly does three things: read an object, evaluate it, and print the result. We do not expect the interpreter to change by requiring one more step in it. So the main program is never extended, and it is okay to write the whole process into the main function.

On the other hand, we do expect that the program will be extended by adding new data types, and also adding new forms. So it is no good for the main program to specify how to evaluate or print a Lisp object.

Instead, we create an abstract class that represents any Lisp object, and from this class we derive concrete classes that represent each Lisp object type. A concrete class should contain a method to print an object of that type. We can thus ask an object to evaluate itself in the main program, no matter what is the real type of object. Printing is just similar.

Reading is more complicated, since we need to create an object without the knowledge about what is the type of object. Note that if we know the first character, we know the type of the object, with the only ambiguity between the Nil type and the cons type. So we can split the object reading code into three functions, one reading an integer, one reading a symbol, and one reading a list. Each of them is contained within the implementation C++ source file of the corresponding object type. It is also good to group the Nil type and the cons type to the same file, since they are highly related. An input parser can then call these functions after looking at the first character using `cin.peek()`. This input parser must be visible by the function that reads a list, since it need to call the parser recursively.

One small hint for reading: completely parse a list and turn it into cons cells **when it is read**. The hint of Milestone 5 below should tell you exactly how this can be done. Don't delay this until the object need to be evaluated. Such delay will make your evaluation function very complicated and error-prone.

Since we expect that the available forms are extensible, we cannot specify all the possible forms in the evaluation function of the cons-cell class. Instead, we must have an abstract class that represents a form (**Read the command pattern!**). Each specific special-form and function then derive from this abstract class. One global object is created for each of these classes, and a form table maps strings into such objects. When a cons-cell is evaluated, it should check that the object is really a list (i.e., repeatedly finding the cdr part of the cell until it is not a cons-cell, and make sure what remains is the Nil object). Then we check that the car part is a symbol, find the corresponding form object, and ask it to evaluate the cdr part of the cons-cell.

One final hint: we usually need to check whether a Lisp object is of a particular type. This can be done using `dynamic_cast`. However, this is not needed for the Nil type, since there should only be one Nil object. We can do simple pointer checking to check whether an object is the Nil object.

Small hints

- Emacs is a good reference. Just type in a **Lisp** expression and evaluate it with `^U ^X ^E`.
- Exceptions are very useful in this assignment.

5. Milestones

When writing a extensible program like this, it is usually the case that the progress seems to be slow until nearly the end of the project, when all at a sudden everything seems to pop out from nowhere. To give you an idea about how good you are doing, we define the following milestones. Each milestone is marked with a pair of score, in the form (x%, y%) below. It means that if you complete this step without failing any previous step, x% will be rewarded; and if you fail this step, at most a further y% will be rewarded. Use CVS to manage the code, and check-in frequently. After you complete each step, make sure that the program works. Create a tag in the CVS, so

that you can easily identify a good “stable” version at the end.

- Milestone 1 (20%, 10%). Define the Lisp objects class in a header file, make a main source file to include this. Implement the evaluation function by just returning itself.
- Milestone 2 (15%, 10%). Implement the integer type. Implement a simple parser that only reads integers. Implement a simple Lisp engine in the main source file to repeatedly parse a Lisp expression (currently, integer), evaluate it and print it.
- Milestone 3 (10%, 10%). Implement the Nil type. Create a global object of this type, which is used whenever we need a Nil value.
- Milestone 4 (15%, 30%). Implement the symbol type. Modify the parser as needed. Create a table to hold the associations from symbols to their values. **Hint:** use an STL map to map a string to a structure that holds a Lisp object and a constant flag.
- Milestone 5 (15%, 10%). Implement the cons-cell type. Modify the parser as needed. Leave the evaluate function unimplemented (i.e., it returns itself). **Hint:** the following recursive procedures read and print a cell in a list form.

Print cell:

```
print '('  
  print-half-list cell
```

Print-half-list cell:

```
print cell->car  
if (cell->cdr is Nil)  
  print ')'  
else if (cell->cdr is a
```

cons-cell)

```
  print ''  
  print-half-list
```

cell->cdr

```
  else  
    print " . "  
    print cell->cdr  
    print ')'
```

Read:

```
get the '('  
  return read-half-list
```

Read-half-list:

```
skip spaces  
if the next character is ')''  
  get the ')'  
  return Nil
```

create cell

```
cell->car = read
```

skip spaces

```
if the next character is ')''
```

```
  get the ')'
```

```
  cell->cdr = Nil
```

```
else if the next character is '.'
```

```
  get the '.'
```

```
  cell->cdr = read
```

```
  skip spaces
```

```
  get ')'
```

```
else
```

```
  cell->cdr =
```

```
  get-half-list
```

```
  return cell
```

- Milestone 6 (5%, 5%). Define the abstract base class of Lisp forms, and derive the quote special-form class from it.
- Milestone 7 (10%, 0%). Implement the symbol table to hold a form table to hold the associations from symbols to their values. Fill in the evaluation function of a cons-cell.
- Milestone 8 (10%, 0%). Derive an abstract function class from the form class, and derive concrete function classes from it. Implement the remaining functions and files.

Enjoy the new way to code!