# Lecture 14

## Assignment 2—Lisp: language and implementation

- Assignment 2 is about a programming language called Lisp.
- We will understand a small subset of the language, and build a Lisp interpreter that understand it.
- We will do it in such a way that extensions towards some particular directions are easy.
- In this tutorial we discuss the Lisp language, the focus and requirements of the assignment, and also some implementation tips.

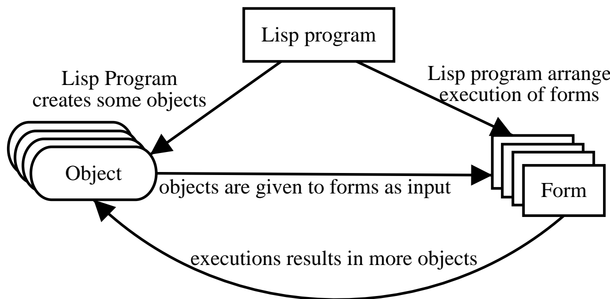


### What's Lisp?

- Lisp is a programming language that depends heavily on recursive lists constructions.
- It is special in that a Lisp interpreter is extremely easy to write. For example, for somebody who knows Lisp and C++ beforehand, this assignment should require no more than 6 hours.
- Being a programming language means that there is some data in the memory, and we are going to manipulate these data in certain ways.
- We call the data "objects", and the things that manipulate them is called forms.
- A Lisp program specifies what objects to build and how these forms are performed when the program is executed.
- Emacs is a good reference: type Lisp expression and then type ^U ^X ^E to evaluate it.



### The abstract picture



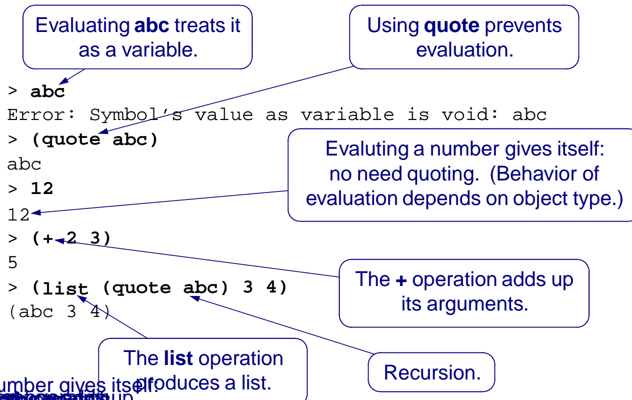



### Lisp basics

- In the memory, there are a pool of Lisp objects. Some of these objects can be referenced using a name.
- There is a type of Lisp object similar to C++ struct, called cons-cell. It is very simple: it always contains two pointers, or references, to other objects.
- Cons-cells are used to build larger structures of Lisp, like lists.
- To make things even simpler, a Lisp program is simply a list of Lisp objects. Specifying the forms to execute and the order to execute them.
- Each object can be executed as a program statement, or **evaluated**.
- Lisp is highly recursive. For example, when a Lisp list is evaluated, most of the time all the list elements (except the first one) are evaluated first, and then the form specified by the first list element is executed.

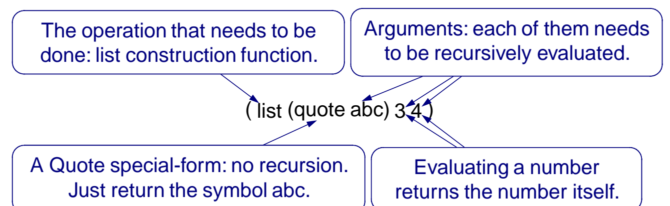


### Learn By Example: Sample run






### The anatomy of a Lisp statement in form of list



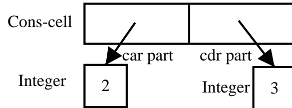

The evaluation process of (list (quote abc) 3 4):

- Evaluate (quote abc): return the symbol abc.
- Evaluate 3: return 3.
- Evaluate 4: return 4.
- Perform the operation: return (abc 3 4).

## Data types

There are just a few data types in our Lisp interpreter:

- Integer: represents a 32-bit integer. Anything in the program starting with a digit is treated as an integer. E.g., 12.
- Symbol: a string of characters other than spaces and parentheses, not starting with a digit. E.g., abc.
- Cons-cell: an object that contains references to two other objects. Written in a dotted notation, e.g. "(2 . 3)":

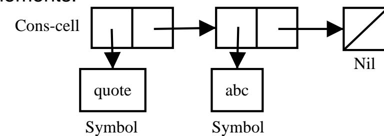



- Nil: represents "nothing", like the NULL pointer of C++. Written as an empty list "()".

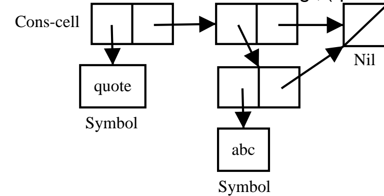


## Cons-cells refering to cons-cells: lists

We can build a list using cons-cell. For example, "(quote abc)" is a list containing 2 elements:



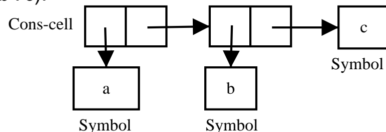

The car-part of a cons-cell can also be a list. E.g., (quote (abc)):






## Improper lists and notations

The cdr-part of a cons-cell need not refer to a cons-cell or nil object. We write it like, (a b . c):




Note that we can write a list in improper-list notation. The following all represents the same thing: a list containing the symbols "quote" and "abc".

(quote abc)<br>
(quote . (abc))<br>
(quote . (abc . ()))<br>
(quote abc . ())



## Forms and functions

Forms are the "computational core" of Lisp programs. Each form is capable of performing a specific operation. For example:

- Addition, Subtraction, Multiplication, Division: perform arithematic operations.
- Find-Car, Find-Cdr, Make-Cons, Make-List: perform operations related to cons cells.
- Set-Variable, Set-Function: setting variables.
- Quote: leave objects unevaluated.

All of them, except Quote, have a standard recursive behaviour to evaluate all the arguments before evaluating itself. We call such forms **functions**.

Quote is not a function, it is said to be a **special-form**, meaning "a form that is not a function".



## Form associations

Initially, a symbol is associated with each form. For example, the symbol '+' is associated with the addition function.

The association can be modified using the form setting function. Here is a rather obscured example:

```
> (fset (quote f) (quote fset))
fset
> (f (quote fset) (quote list))
+
> (fset 2 3)
(2 3)
```

Note that association of anything can be changed, including pre-defined associations. So we purposely don't call the Set-Form function as the "fset function": fset can refer to anything by changing the form association.



## Variable associations

- As well as associating with a form, a symbol can be associated with a value. We call this the variable value of the variable.
- We can use the Set-Variable function to change the association. However, the nil symbol always refer to the nil object and cannot be changed.
- Different association is used when a symbol appear at a different position: when a symbol is evaluated, its variable value is used. When a symbol is the first element of a list to be evaluated, its form value is used:

```
> (fset (quote a) (quote +))
+
> (set (quote a) 5)
5
> (a a a a)
15
```

Note: Emacs uses one more level of dereference in fset, so behaviour of fset is a bit different from our interpreter.

## A more complicated example

Here is one more example on the interpreter. Try to understand how the whole thing work.

```
> (set (quote a) 5)
5
> (set (quote a) (cons a a))
(5 . 5)
> (set (quote b) (list a (quote b) a))
((5 . 5) b (5 . 5))
> (car (cdr b))
b
> (+ (car (car b)) (cdr (car b)))
10
> (+ (car (car b)) (cdr (car c)))
Error: Symbol's value as variable is void: c
```



## Focus: extensibility

The main focus of the assignment is extensibility. That means:

- Your program should have infra-structure that makes it easy for new functions to be defined, without affecting the natural meaning of the rest of the system.
- This should be possible without touching any of the code already written, except that we allow a one or two line changes to the main program.
- Similarly, it should be easy to add support for a new data type.
- When assessing your progress, such infra-structures are the main concern, **not the set of supported features**.
- That is, hard-coding some simple case into the program for just one function will earn you very little credit.

Note that all these are possible only if we know the direction of extension.



## Achieving extensibility: class structure

The key to achieve extensibility is to think ahead.

- We know the main operations exactly in a high level, and that's not going to be changed. It simply reads an object, evaluate it, print the result, and repeat.
- But we don't know the possible set of data type. To allow extensibility, we must be able to use a variable to store a "Lisp object" without knowing the type—i.e., need abstract class for Lisp object.
- To implement a Lisp object type means to be able to read, print and evaluate such an object: pure virtual functions of the Lisp object class.
- We also don't know the set of possible forms: we expect that they will be extended. Again we must use an abstract class of forms here.
- To implement a new form means to be able to execute it given a set of arguments: a evaluation function.



## Variable and form table

There are two types of associations in the program: variable and form associations.

- We keep a data structure for storing each type of associations.
- The first table maps a string (e.g., a) to a Lisp object (e.g., 1). It is also necessary to keep a set of strings for which the association is constant.
- The second table maps a string (e.g., +) to a form.
- For the second table to work, there must be an object for each form. This is easy, since we already must have a class for each form anyway.
- These table must support modification and query.
- For the program to be usable, the data structure used must be reasonably efficient. Vectors and linked-lists are both bad choices, since both require linear searching for the association.



## Making tables: C++ STL maps

Luckily, C++ STL provide a way for you to very easily make a good table: map.

- A map is a template class that have two class parameters: the input type and the output type.
- The input type must be comparable, i.e., `operator<` must work. Examples are string, date, etc.
- The output type can be any type. E.g., integer.
- So the type looks like map<string, int>
- The function of a map is simple: store a value of the output type for each value of the input type. By default the value stored is the default value of the type, i.e., null pointer for pointers, 0 for integers, etc.
- The map is efficiently implemented using a red-black tree.



## The parser

The first step of the main loop should be to read in an object. That is, read enough characters from cin to get a Lisp object.

**Parse the input into a Lisp object when you get it. Do not delay until you need to evaluate the object.** Doing otherwise will make your code extremely error-prone, un-extensible, messy, … (put your bad words here).

We usually call a function that perform such task a "parser". It's purpose is to get exactly one object from the input stream.

However, to allow extensibility, we have to write our program so that part of the responsibility of the parser is given to the implementation of the Lisp object classes.

## Getting the input you need

We will assume that the type of an object in the program can be determined by the first character—even after extension. Currently, we have:

- If it is a '(', it is either a nil object or a cons object.
- If it is a digit, it is an integer.
- Otherwise, if it is not '.' or ')', it is a symbol.

But the code to get an object should still be in the implementation file of the corresponding object type.

The parser code: "if the first character is in the range from '0' to '9', call get_integer. Otherwise, …"

You can even make one more map, in the lines that "if the first character is in the range from '0' to '9', use the object integer_maker. Otherwise, …". See the **factory pattern** to see how to do this.



## Getting a dotted form list

It might seem very difficult to get such a recursive list. It turned out that this is not difficult at all: the following mutually recursive function do the job nicely.

```
Read cell:
    get the '('
    read−half−list cell

Read−half−list cell:
    [read cell->car]
    skip spaces
    if the next character is ')'
        cell->cdr = nil
    else if the next character is '.'
        [read cell->cdr]
        get ')'
    else if the next character is ')'
        get−half−list cell->cdr
    else
        error
```



## Dependencies in parser code

Note that to read a list (implementation of cons-cell), we require to read an object in the middle (interface of parser).

The reverse is also true: to read an object (implementation of parser), we may require to read a list (interface of cons-cell).

So the dependencies look like this:



## Packing everything together: Program structure

- Organize your program so that it is contained in multiple files and can be separately compiled.
- Implement each data type in its own source file (except perhaps nil and cons-cell). The implementation of reading, printing and evaluation of such data type should all live in these source file.
- Make an class for each data type, so that Lisp objects of that type are instances of that class.
- Create a map for variables and forms. The main program initialize it with the default mapping, and start the main loop.
- Implement forms separated from the rest of the program. Make a class for each form, and make an object in the implementation file. Export only this object in the header file.



## Other matters: dynamic_cast

- To implement the forms, we usually need to check whether a Lisp object is of a particular type. E.g., the arithematic operators need to know whether the input is an integer. This can easily be done using dynamic_cast:

```
if (LispInteger *li=dynamic_cast<LispInteger*>(lisp_object)) {
    // lisp_object is an integer, do the required operation with li
} else
    // An error!
```

- For efficiency, we should not do this for the nil class. Instead, makes sure that there is only one nil object, and use pointer comparison directly:

```
extern Nil_Object *nil_object;  // Defined in the implementation of Nil.
...
if (lisp_object == nil_object)
    // Yes, it is nil.
else
    // no, it's not nil.
```



## Error handling and exceptions

- Finally, there should be full error reporting in your program.
- It is in fact very easy: just throw when you see something wrong:

```
if (LispInteger *li=dynamic_cast<LispInteger*>(lisp_object)) {
    // lisp_object is an integer, do the required operation with li
} else
    throw EvalError("Wrong argument type - not an integer",
lisp_object);
```

- The only exception handlers are in the main loop. There are two types of errors: read errors and evaluation errors.
- The handler needs to print the object causing the error into cerr. The easiest way to do this is to give the printing function a ostream reference:

```
class LispObject {
    print(ostream &ost);
}
```