Pengfei Wang

Professor Christopher A. Healy

Computer Science 475

11 April 2016

Compare LISP with Object-Oriented Programming Languages

Introduction

LISP, which stands for List Processor, is a family of functional programming languages, invented by John McCarthy in MIT based on Lambda Calculus in 1958. It is the second-oldest high-level programming language and it was originally designed for Artificial Intelligence, but it is a good general-purpose language as well.

This paper is going to discuss the differences between LISP and Object-Oriented Languages, focusing mainly on LISP. This paper is going to firstly introduce functional programming language and LISP. Then, advantages and disadvantages of LISP comparing to object-oriented languages will be discussed, and finally a specific case study will be done.

Functional Programming Languages and List Processor

Typically, programming languages are divided into two general types, imperative languages and declarative languages.

The computer program written in imperative languages consists of statements, instructions, which indicate how (and in what order) a computation should be done. Imperative languages include all of the object-oriented languages. On the other hand, declarative languages define functions, formulas or relationships. Declarative languages do not specify how the functions are performed step-by-step in program. Functional

programming language is one type of the declarative languages.

```
;; Example: Written in Common LISP

(format t "Result: ~A.~%" (* 2 (+ 2 3))) ; functions

/* Example: Written in C*/

int a = 2 + 3; /*statement*/

int b = a * 2; /*statement*/

printf("Result: %d.\n", b); /*statement*/
```

Function languages employ a computational model based on the recursive definition of functions. In essence, a program is considered a function from inputs to outputs, defined in term of simpler functions through a process of refinement (Scott 9).

Among all of the functional programming languages, LISP is the special one and is claimed to be the most powerful language in the world.

Richard Stallman, who founded the GNU Project, once said "The most powerful programming language is Lisp. If you don't know Lisp (or its variant, Scheme), you don't know what it means for a programming language to be powerful and elegant (Stallman, How I do my computing)."

The common structure of this language LISP is list. Even functions (the basic component) are expressed by lists. For example, (+ 2 3) is a function which returns the addition of 2 and 3, and a list with three elements, "+", "2", and "3". Everything in LISP can be expressed as a list. "Meanwhile, ideas borrowed from LISP increasingly turn up in the mainstream: interactive programming environments, garbage collection, and run-time typing, to name a few" (Graham 6).

Classical LISP compilers are interactive. After one line of code is finished, the interpreter (and/or compiler) will evaluate that line of code, and print the return value to the console. Then the programmer can start the second line. This process is called REPL (read-evaluation-print loop). This process forces programmers to debug their code piece-by-piece, line-by-line. LISP is also the first language with garbage collection and run-time typing.

Advantages

1. Variables being Immutable, Higher-order Functions and Closures

Rather than regarding variables to be immutable, a better approach is to see what a function is able to do in LISP. The only result of an evaluation is the function's return value.

Most importantly, this feature and the R-E-P loop make the most time-costing process, debugging, really simple. The only possible error, if the algorithm is implemented correctly, is resulted from a wrong return value from a function. The programmer can immediately realize when mistakes occur and he only has to trace the return value back to a lower-order function repeatedly, by evaluating the functions manually one by one.

In contrast, it is complicated to test every statement independently in object-oriented language because of their imperative nature. The execution of every statement is affected by its context, which means the same error probably will not appear if it is tested independently. Secondly, bugs are harder to trace because the value of variables might be changed. In this case, programmers have to first locate what changes have been made and then check on each change to see if anything goes wrong.

Then, how can LISP achieve the same goal as object-oriented language do, when values

are “expected” to be modified in object-oriented language? The idea is not to simulate the same functionality and change variables as object-oriented programmers do. Instead, in functional programming, programmers don’t have to change the value of variables. Generally, in object-oriented programming, programmers want to change the value of a variable in order to manipulate and pass the value between objects. In functional programming, programmers only need to recursively take a function as a higher-order function’s input and a new value will be returned and passed to the higher-order function, instead of changing anything.

```
;; Input function example, Common LISP
(/1 (+ 2 3)) ; addition is the input function
;; Output function example, Common LISP (closure)
(defun test (a) #’ (lambda (b) (+ b a)))
;; Function call, Common LISP
(funcall (test 1) 2) ; “(test 1)” returns lambda function
```

The “test” function returns an anonymous function without specifying its name, adding its input value with another parameter. “#” makes sure that this function (lambda) will not be evaluated until it is actively called by “funcall” function. Actually, when a function (i.e. lambda (x)) refers to (and be defined with) a variable outside of it (i.e. a), this function is called a closure.

In some object-oriented languages, for example JavaScript, functions can also be returned. However, these cases are very rare, and, indeed, JavaScript is heavily influenced by Scheme, which is one of the two main dialects of LISP. JavaScript has noticeable common features with LISP. JavaScript can return a non-anonymous function, which is shown below.

```
// Output function example, JavaScript (closure)
function f1(int a) { return function f2(int b) { return a+b } }
```

In C++ or C, programmers have two ways to accomplish this. Programmers can either use a “typedef” to return a pointer to a function, or use an anonymous function (lambda object), which was recently implemented in C++11. However, creating a pointer to a function is not returning a function fundamentally, and C++11 version of anonymous function, which is originated from functional programming thought, is not as concise as LISP, as shown in the bottom (Sean, Returning functions).

```
// C++ Anonymous Function Returning
std::function<int (int)> retFun(){return [](int x){return x;};}
```

Actually, you can assign a value to a variable and change the value of the variable in LISP. However, these functions are highly discouraged, which cause unexpected errors. These functions are called destructive functions. In LISP, most destructive functions have safe versions. These safe versions return new data structures instead of changing them. For example, “delete” function deletes a variable in the list and returns null, but “remove” function returns a manipulated list without changing the original list.

Interestingly, because of LISP’s interactive nature, you can even change the code while they are running.

```
;; Print 1 to 1000000
(do ((x 1 (+ x 1))) ((< x 1000000)) (format t "~A~%" x))
```

You can evaluate this function and change ((x 1 (+ x 1))) to ((x 5 (+ x 1))) while the program is running, and there will be unexpected errors in the result. This is analogous to the

outcome of evaluating destructive functions.

2. Less deadlocks

Actually, LISP doesn't need a "lock" on variables because typically it will never change their values. Therefore, on multi-thread programs, LISP is more efficient than object-oriented languages, because it can be run in parallel more efficiently. Even if the program is single-thread, the compiler is able to make the whole program (or probably some part of it) run in parallel in CPU.

```
;; Example
(someFunction1 x)
(someFunction2 y)
(someFunction3 x y)
```

These functions can be run in parallel because the first two functions cannot change the value of x and y typically. An object-oriented programming compiler, however, will worry about whether x and y's values are changed in the previous functions and therefore "someFunction3" will run at last. This is also an example of why destructive functions are not recommended. Destructive functions might change the values and cause unexpected errors.

3. Extendable Language: Macro

LISP can be extended by itself, using macro, which changes the behavior of a compiler during compiling. Programmers can use "defmacro" to define a macro and use "defun" to define and return a function in this macro. Thus, the content of the returned function can be changed depending on what parameter is given to the macro. Literally, programmers can write a program, which can write programs.

```
;; Macro Example
(defmacro someMacro (input1 input2)
   (defun someFunction (…) #’ (lambda (…) (…)))
```

Also, by macro, programmers can simulate any grammar, for example the whole JAVA. Theoretically, with macro, LISP is equivalent to Lambda Calculus.

It is possible to change the behavior of the compiler when it reads every possible character, even the pre-defined ones. For example, programmers may change the functionality of “+”.

```
;; Reader Macro Example: set the functionality of ”+” to minus
(defun addn (x) (- (nth 0 x) (nth 1 x)))
(set-macro-character #\+ #'(lambda (stream char)
      (list 'addn (read stream t nil t))))
```

These two functionalities of macro are the important reasons why LISP was designed for Artificial Intelligence. It is flexible and powerful enough to build an A.I.

Beneath the surface, LISP’s macro is able to manipulate the abstract syntax tree of the language itself. Programmers are thus able to change the behavior of the compiler, and are also able to decide whether the interpreter should evaluate the function during macro expansion or after macro expansion. Theoretically, this is almost the upper bound of what a programming language can do.

In contrast, macro in C++ is only able to create a simple substitution table between macro and lines of code, without changing the compilation behavior.

This is why LISP is claimed to be the most powerful language in the world. Actually,

there are only less than thirty built-in symbols in LISP, and the rest are all macros. The flexibility and power of macro in C++ is hence worse than LISP.

4. Fast and Concise.

In the previous examples, you probably have witnessed one of the most important advantages of LISP, conciseness. It is one of the most concise languages being widely used up till now. According to the research done by Donnie Berkholz from RedMonk, LISP family occupies 9 positions among the top-30 most expressive languages. The article explains the meaning of expressiveness in the rank. It is "how many lines of code change in each commit. This would provide a view into how expressive each language enables you to be in the same amount of space" (Berkholz <Programming Language Ranked by Expressiveness>).

To illustrate LISP's conciseness against other object-oriented languages, a Hello World program is enough. The rank below is from the longest to the shortest.

Rank 4	JAVA (#char=97, #line=5)
Rank 3	C++ (#char=79, #line=5)
Rank 2	SmallTalk (#char=33, #line=1)
Rank 1	LISP (#char=15, #line=1)

```
;; written in Common LISP (rank#1, #char=15, #line=1)
;; HelloWorld source code in other languages are in the folder
"Hello, world!"
```

The REPL process will automatically print the return value of the function (which only contains a simple string in this case) to the console. If programmers are not using IDE, there is no difference between the actual printed message and the automatic generated message.

LISP has other ways of active printing. Both of them are more concise than SmallTalk.

```
;; written in Common LISP (alternative 1, #char=23, #line=1)
(format t "Hello, world!")
;; written in Common LISP (alternative 2, #char=21, #line=1)
(print "Hello, world!")
```

However, as said in *ANSI Common LISP*: "Lisp is really two languages: a language for writing fast programs and a language for writing programs fast." Programmers usually write programs fast in early development phase, and then try to optimize it. To do optimization, LISP programmers have many options, like optimizing the code by themselves or the compiler (or interpreter). The CPU-time of a program can decrease significantly after appropriate optimization. It can be slower than JAVA or faster than C. However, it is very complicated to do optimization in LISP. Detailed optimization process will be discussed later in this paper after analyzing my own experiment result.

Disadvantages

1. Hard to learn

LISP was originally created as a mathematical notation for computer programs. LISP programmers have to learn Lambda Calculus in order to fully understand LISP's mechanism. Lambda Calculus is not covered in undergraduate courses in most universities or colleges. Therefore, it is nearly impossible for newbies to master LISP, but it is relatively much easier to master an object-oriented language.

Furthermore, due to differences between functional programming and object-oriented programming, it is highly difficult for programmers to switch between object-oriented

languages (e.g. JAVA) and functional languages (e.g. LISP).

2. Small Libraries

Because of the difficulty, the programmer community is very small, and hence the libraries are small.

In Google Trend, accessed in April 18, 2016, within all of the searches about programming languages, Java occupies 54% of the searches and LISP occupies less than 1%.

In Amazon Books, accessed in April 18, 2016, if you search JAVA, 27846 results show up and only 1286 results show up for LISP.

This is essential, because productivity matters in companies. Small programmer community means small main or third party libraries. With one update followed by another, JAVA API is growing increasingly bigger and contains 4024 items right now. However, Common LISP only has 978 symbols.

Actually, LISP programmers are expected to create their own algorithms by themselves. For example, if a programmer forgets how to write a sorting algorithm, a normal JAVA developer can just use “collection.sort()”. However, a LISP programmer should do this on their owns, or look for help on Google.

Case Study

In order to show a relatively more complex example and compare LISP to object-oriented languages, bubble sort is used to do a case study.

Data Set: 1000 Numbers List; Run 100 times

OS & CPU: Windows 10; CPU @ 2.90GHz (8 CPUs)

Compiler: JAVA 1.8.0; Steel Bank Common LISP

```
;; Bubble sort example written in Common LISP, by WikiBooks
;; Bubble sort source code in JAVA is in the folder
(defun bubble-sort (lst)
   (loop repeat (1- (length lst)) do
      (loop for ls on lst while (rest ls) do
         (when (> (first ls) (second ls))
            (rotatef (first ls) (second ls)))))
   lst)
```

At first, another study done by Dr. Erann Gat is examined before reaching my test result. Dr. Erann Gat asked 14 programmers to submit 16 programs. Twelve programs were written in Common LISP and four were written in Scheme.

"We used the same problem statement (slightly edited but essentially unchanged), the same program input files, and the same kind of machine for the benchmark tests: a SPARC Ultra 1" (Gat 21). Generally, two results were given.

"First, development time for the Lisp programs was significantly lower than the development time for the C, C+, and Java programs …

Second, although execution times of the fastest C and C++ programs were faster than the fastest Lisp programs, the runtime performance of the Lisp programs in the aggregate was substantially better than C and C++ (and vastly better than Java) (Gat 22)."

However, my testing result is totally different:

	Common LISP	**Java**

Lines	6	15
Characters	142	260
Run-Time	343ms	16ms

We can see that Common LISP is so concise but its speed is not ideal, comparing to the study done by Dr. Erann Gat. This is a good example of why optimization is necessary, but not only recommended.

In this case, firstly, accessing an array in JAVA is O(1), but accessing a list in LISP is O(n). Furthermore, this Common LISP program uses a destructive function called “rotatef”. Using imperative style in LISP (e.g. rotatef), instead of functional programming style, will make it super slow. Ideally, no destructive function should be ever used. Theoretically, even assignment function, “setf” is a destructive function, because typically there is no variable in functional programming in which the values can be reassigned. Though, “setf” is commonly allowed in functional programming style in order to provide names for data, if it is not used to change those values. On the other hand, Common LISP is a multi-paradigm language. It allows many styles to be performed. You can even do procedural programming, imperative programming, or object-oriented programming, if compiler is correctly modified using macro to change its behavior and grammar. It allows Common LISP to be written in direct and easy understanding way in early developing phase, but this tolerance becomes disadvantage if no optimization is performed.

More importantly, bubble sort is not a functional algorithm. General algorithms, which

are available in computer science courses and online sources (e.g. Wikibooks or Rosetta code), are not designed functionally. In order to make bubble sort functional, the whole algorithm needs to be modified. When a switch between two elements in the list are expected, the function should return a new list which have done the element switching and pass the new list to the high-order function, but not change the value of the old list. Changing values are not functional way of thinking. In this case, in a 1000 numbers list, if the function returns a new list every time, there will be much more amount of data accessed and created. Obviously, this is inefficient.

Therefore, instead of trying to make bubble sort functional, a better solution is to choose a more functional algorithm, for example, merge sort. The list operations in merge sort are well supported by LISP.

```
;; written in Common LISP by Rosetta Code
(defun merge-sort (result-type sequence predicate)
  (let ((split (floor (length sequence) 2)))
    (if (zerop split)
      (copy-seq sequence) (merge result-type
          (merge-sort result-type
          (subseq sequence 0 split) predicate)
          (merge-sort result-type
          (subseq sequence split) predicate)
   predicate))))
```

Different from bubble sort, this algorithm is very “functional” and LISP-styled. In the

code above, none of the destructive functions is used. As we could expect, the improvement in percentage difference comparing to JAVA is huge.

	Bubble sort		**Merge sort**	
	Common LISP	**Java**	**Common LISP**	**Java**
Run-Time	343ms	16ms	37ms	51ms

It was originally ~2000% slower than JAVA, but it is now ~25% faster.

Other Optimizations

In the case above, the necessity of optimization is shown. Optimization is much more significant and necessary than it is in object-oriented languages like Java. Without optimization, LISP could be more than twenty times slower than JAVA, but it could be faster than C if programmers optimize it carefully, as discussed by Dr. Didier Verna in *How to make LISP faster than C*. LISP programmers should "use appropriate data structures, type declarations and optimization settings" in order to make it fast (Verna).

Besides the strategies suggested by Dr. Didier Verna, further optimizations are available to make LISP faster, which can hardly be performed in object-oriented languages.

1. Choose a better dialect of LISP with better optimization.

LISP is not a single language, but a family of languages. Due to its flexibility, it has many variants, like Clojure, Scheme, Common LISP, and newLISP. It is very easy to switch between these family members, even easier than switching from JAVA to C++. For example, in JAVA, programmers do not use pointers, but, in C++, pointers are recommended to use rather than global variables. Furthermore, different languages in these family members have different advantages. For example, Cojure, according to RedMonk's project research, is one

of the most expressive languages in the world. In the bubble sort case, newLISP is recommended because its performance of accessing a list is O(1). This feature may significantly improve the run-time of bubble sort in LISP (although it still uses a destructive function, “rotatef”). Users have many choices, which are all easy to learn, depending on the tasks.

2. Compiler/Interpreter Optimization Example: Tail Recursion to Loop

As it was mentioned previously in this paper, LISP programmers can extend the language by itself. However, extendibility is only one of the advantages of using macro. Macros can also be used to optimize the compiler (or interpreter), and using macro is only one of the ways to optimize the compiler (or interpreter). Programmers can optimize the compiler by defining built-in or new macros, or writing an optimization algorithm in other language.

Optimizing tail recursion to loop is a simple example. Tail recursion happens when a function call is performed as the last action within a function. As you can imagine, this can be converted to loop.

```
;; written in Common LISP, Tail recursion
(defun addn (a cont)
  (if (equal cont 0) nil
   (if (< a 10) (+ a 1) cont 1))
   (addn a cont)))
;; written in Common LISP, Loop
(do ((x 1 (+ x 1))) ((> x 10)) ())
```

As mentioned above, LISP is originally created as a mathematical notation. Therefore,

theoretically, if two functions can be proven to be mathematically equivalent, the compiler is able to convert one to the other by appropriate optimization. Here, an example of interpreter optimization written in C is provided in the folder (interpreter.c). It can also be rewritten in LISP.

In contrast, object-oriented language programmers can hardly optimize a given compiler itself. Programmers have to find different compilers (if available) for different purposes or for different requirements. At least, these work are far more complicated and time-consuming than just writing a LISP macro and evaluating it.

Conclusion

LISP has many convenient and flexible features, making it capable of many tasks, which cannot be done by object-oriented languages. However, to mater these features, a programmer has to devote much more effort to fully understand how LISP works. Furthermore, though it can be both concise and fast, but it need to be optimized according to different tasks, and because of the small programmer community, finding such optimizations is not as easy as finding JAVA's. Therefore, LISP is not a good language, which can be massively used by companies, but it is a good language for research purposes and education. Though, some software, like AutoCAD, is written in LISP. Optimized appropriately, it is fast, concise and powerful, and thus superior to object-oriented languages, for professional users.

The good news is that modern versions of LISP interpreter (with compiler) are implementing increasing number of optimizations with it. For example, Steel Bank Common LISP (interpreter + compiler) can do automatic tail recursion optimization to some extent.

Works cited

Berkholz, Donnie. "Programming Languages Ranked by Expressiveness." RedMonk. 25 Mar. 2013. Web. 16 Apr. 2016. <http://redmonk.com/ dberkholz/ 2013/03/25/ programming-languages-ranked-by-expressiveness/>

Gat, Erann. "Lisp as an Alternative to Java." Intelligence 11.4 (2000): 21-24. Print.

Graham, Paul. ANSI Common Lisp. Englewood Cliffs, NJ: Prentice Hall, 1996. Print.

Rosetta Code. "Sorting Algorithms/Merge Sort." Rosetta Code. 5 Feb. 2008. Web. 18 Apr. 2016.

Scott, Michael Lee. Programming Language Pragmatics. Elsevier Science, 2006. Print.

Sean. "Returning Functions." Stackoverflow. 18 Feb. 11. Web. 16 Apr. 16. < http://stackoverflow.com/questions/4726768/returning-functions>

Stallman, Richard. "How I Do My Computing." Richard Stallman's Personal Site. Richard

Stallman. Web. 11 Apr. 2016. <https://stallman.org/stallman-computing.html>.

Verna, Didier. "How to Make LISP Faster than C." IAENG International Journal of Computer Science 19th ser. 32.4 (2016). Print.