CMPSCI 630 Systems **Spring 2023**

Lecture 4

Lecturer: Emery Berger *Scribe: Fadhil I. Kurnia*

In this lecture, we are discussing the features and mistakes of LISP, also Garbage Collection (GC).

4.1 LISP Programming Language

One of the main goals of LISP development is to easily program lambda calculus, which is a totally different goal than what FORTRAN and COBOL have. Lambda calculus is a powerful way to express functions where we have both function and scope; Lambda is function + environment. Commonly people also referring lambda as closures. Developed with that that lambda calculus under consideration, LISP was specifically designed to represent Symbolic Artificial Intelligence (AI) code such as logic, lists, structures, and function over them. Some of the LISP's features and their bad consequences are described below.

4.1.1 Key Features of LISP

Recursion. Recursion is a unique feature LISP offers compared to other programming languages at the same era. This enables execution with divide and conquer approach.

Functional Language. LISP is among the first functional language. The term "functional language" itself has a broad meaning. Commonly it refers to function-oriented language where function returns value. A function excludes or limits global state or mutation; commonly referred as "pure" function since it prevents any side effects. However, pure function commonly require a lot of value copying.

Interpreted Language. The `eval` in LISP which takes string and runs it makes LISP as an interpreted language. LISP treats program as data, for example: `'(CAR (CDR '(1 2 3)))'`. This is a powerful concept for bootstrapping where we write a programming language with itself. We only need to implement a small core and implement the rest of the language with that implementation. However, this also makes the code hard to compile in advance; making LISP as an inefficient language, both in space and time aspects.

Garbage Collected. LISP has an automatic garbage collection process using reference counting method. However, reference-counting is considered inefficient since there are many cascading operations involved. This was explained in the previous lecture.

Dynamic Types. LISP has dynamic types, for example a list in LISP can have multiple elements with different types. Dynamic type requires the language to pair value with its type ("boxed" variable). Another language with dynamic type is JavaScript, as shown in the example below.

```
let x;    // a "boxed" variable, it needs to have both the type and value [TYPE | VALUE]
x = 31;   // now the variable contains an integer {<int>: 34}
```

Starting Point for Async. A variant of LISP developed in MIT also introduces the concept of `future` for concurrent execution. That is similar as async/await. See the code below for an example.

```
Let x = future fib(1000);   // run fib in another thread
...
Let q = awat x + 12;        // if x is not present yet, then wait (blocking)
```

4.1.2 Mistakes in LISP

Inefficient Value Copying. A pure function, as in LISP, which handle everything in its scope and need to return new value usually require copying value. That value copying is inefficient compared to updating the value in-place. Consider a function that has list as the parameter and return a list with new element in the end, as shown by the input and output example below.

```
Input: ( 1 2 ... 1000000 )
Output: ( 1 2 ... 1000000 1000001 )
```

In a pure function, we need to copy the input list into a new variable with larger size, add a new element in the end, then returning it. Also, consider another function that need to remove the first element as shown in the input-output example below. Similarly, the function need to allocate a new list and copy almost all of the element from the input list.

```
Input: ( 1 2 ... 1000000 )
Output: ( 2 3 ... 1000000 )
```

A more efficient mechanism would do the operation constantly by arranging the pointer, either at the head or the tail of the list. This inefficiency in LISP makes the O(1) operation into O(n), therefore people perceive LISP as a slow language.

Evil Eval. The introduction of `eval` in LISP make teh code hard to be compiled in advance, it could produce runtime-error which can only be known during execution. An example of this is a typo in a method invocation as shown below. This happens a lot in Objective-C which is extensively used in iOS; you can try to see your phone log to check this. Because of this, some people prefer compiled language than interpreted one.

```
void doSomething() {
   ....
}

x->doSomethang(); // results in runtime error: method not found
```

Another example is when there is an undefined variable encountered as shown below.

```
void foo() {
    let q;
    ...
    q = x + 123; // results in undefined var error
}
```

Accidentally Dynamically Scoped. Other than having dynamic typing, LISP is accidentally dynamically scoped. Dynamic scope makes the code hard to read and reason about. The creator of LISP did not intend that and try to create a lexically scoped version of LISP which they called as COMMON LISP.

Scope is about name resolution in the code. Example of scoping is shown below.

```
( ... int a;     // first variable called a
   ( ... int a;  // another variable called a
       a         // the variable a here refer to the second a
```

A dynamically scoped language execute the code based on the scope of the caller. As an example, see the code below.

```
int Foo() {
    return x + 2;   // x is not defined in the function
}

x = 12;             // the definition of x, outside of Foo()
q = Foo();          // this Foo() invocation uses x=12
```

While in a lexically scoped language, every variable need to be defined in the same scope where the variable is used. This make a small part of the code easy to read and reason about without having to check another part of the code. As an example, the same `Foo()` function above will not be compiled since `x` is undefined in the function's scope.

```
int Foo() {
    return x + 2;   // x is not defined, error during compilation
}
```

Other than LISP, another dynamically scoped language is `var` in JavaScript. Similar as LISP that finally make a lexically scoped language with COMMON LISP, JavaScript offers lexically scoped variable when the variable is written with `let`, instead of `var`.

Inefficient Garbage Collection. The reference-counting method used in LISP has poor performance. For example, in `emacs`, a workplace written in LISP regularly will pause for seconds to do garbage collection, this is known as "stop the world". Because of this, some people see GC as a bad thing and should be avoided. After LISP, the next popular language with GC is Java, it was introduced in the 80s, 20-30 years after LISP. Many advance mechanisms for efficient GC were developed, for example as in Java and JavaScript.

4.1.3 Why People Repeating the Same Mistakes?

There are many lessons learned from the previous mistakes, such as preferring compiled language over interpreted one, avoiding reference-counting for GC. Despite that, people still repat the same mistakes for other programming languages. Some examples:

- Facebook uses PHP (an interpreted language) before developing Hack (a language with static type).
- Twitter uses Ruby (an interpreted languge with dynamic type) with Ruby on Rails framework.
- Apple initially uses Objective-C, an interpreted language which has virtual methods that can surface "method not found" error. Then, Apple replace that with Swift. However, both Objective-C and Swift use reference-counting method for GC.

- Java was initially interpreted, then it becomes a compiled language with efficient GC.
- Julia was initially developed to replace FORTRAN. However, there is already a solution for that called `numpy` in Python. `numpy` uses efficient FORTRAN under the hood.

The big question is: why people repeat the same mistakes when developing a new programming language? Most people create a new programming language with a specific purpose and hope people will use it, which is an expensive bet. The main reason is the ease of development. Getting something out there soon, then see whether people will use it or not ("Worse is better").

Beside that reasoning, the usage of reference-counting in Swift has a specific reasoning. Even though inefficient, reference-counting is entirely local and enable other resources, such as socket or file descriptor, to work well.

4.2 Garbage Collection (GC)

As mentioned in the previous lecture, there are two main types of GC: reference-counting and mark-sweep. All those GC mechanism have the same goal, that is to manage the resource efficiently. Consider a simple allocator and collector in the code below:

```
int buffer[INFINITY];
int offset = 0;

void collector() {
    return;
}

int *allocator(int size) {
    int* crt = buffer + offset;
    offset += size;
    return crt;
}
```

The simple GC above is valid, but inefficient since it never release the previously used memory. Consider the code below that will use O(1000) memory instead of O(1).

```
for (int i=0; i < 1000; i++)
   var O = new Object();
```

However, the notion "efficient" here is relative. What if the code above only run for 1 second and release all the memory after termination? Therefore, GC is about space-time tradeoffs. We can do GC in every iteration and only use O(1) memory, never do it and use O(1000) memory, or do it for every certain amount of memory allocated.

```
memory usage:     1  ..............x..............  1,000
GC frequency:   every         every certain         never
              iteration         iteration
```

In terms of identifying all unused memory, there are two kind of GC: conservative and precise. A precise GC requires us to precisely differentiate pointer and value. If we can not precisely identify pointer, we can not do compaction and relocation. Generally there are two method to make compaction faster: a mutator which run the compaction and relocation in background, and parallel compaction with multiple threads/processes.

In the next lecture, we are going to learn about copying and generational GC.