CMPSCI 630 Systems **Spring 2023**

Lecture 4

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

In this lecture, we are discuss the features and mistakes of LISP, and 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; A lambda function is an anonymous (unnamed) function. It is typically an expression based on its parameters, but it may also use variables from its environment. A lambda function together with that environment is called a *closure*. Developed with that that lambda calculus in mind, 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 languages. The term "functional language" itself has a broad meaning. Commonly it refers to function-oriented programming where functions return values. A function excludes or limits global state or mutation; commonly referred as "pure" function since it prevents any side effects. However, pure functions commonly require a lot of value copying.

Interpreted Language. The `eval` statement takes a string and executes it, making LISP an interpreted language. LISP treats program as data, for example: `'(CAR (CDR '(1 2 3)))'`. This is a powerful concept for bootstrapping a programming language, where we implement a programming language with itself. We only need to implement a small core, which we then use to implement the rest of the language. 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 typing requires the language to pair values with its type ("boxing" variables, i.e., treating even primitive values as objects). Another language with dynamic type is JavaScript, as shown in the example below.

```
let x;    // a "boxed" variable, with internal fields indicating type and 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 = await x + 12;       // if x is not ready yet, then wait (blocking)
```

4.1.2 Mistakes in LISP

Inefficient Value Copying. Pure functions usually require copying values. That copying is inefficient compared to updating them in place. Consider a function that has takes a list as a parameter and returns a list with a new element in the end, as shown in the 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 return it. Also, consider another function that removes the first element as shown in the example below. Similarly, the function need to allocate a new list and copy almost all of the elements from the input list.

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

A more efficient mechanism would do the operation in constant time by moving 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 makes it hard to compile the code in advance, as it could produce runtime errors, i.e., errors only 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 see those in any iPhone's log. Because of this, some people prefer compiled languages.

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

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

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

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

Accidentally Dynamically Scoped. Besides being dynamically typed, LISP is also accidentally dynamically scoped. Dynamic scope makes the code hard to read and reason about. The creator of LISP did not intend that and tried to correct it with a lexically scoped version named 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 refers to the second a
```

A dynamically scoped language executes 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
```

In a lexically scoped language, every variable needs to be known in the scope where the variable is used. This makes 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 would not compile since `x` is undefined in the function's scope.

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

Another example of dynamic scope is `var` in JavaScript. Just like COMMON LISP was introduced to as a lexically scoped version of othe original LISP, JavaScript supports lexically scoped variables when these are declared with the `let` keyword (rather than `var`).

Inefficient Garbage Collection. The reference-counting method used in LISP has poor performance. For example, `emacs`, an editor built on LISP, regularly will pause for seconds to do garbage collection. This is known as "stopping the world". Because of this, some people see GC as a bad thing that 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 advanced mechanisms for efficient GC were developed, for example as in Java and JavaScript.

4.1.3 Why do People Repeat the Same Mistakes?

Despite the lessons taught by previous mistakes, people still repeat these in later programming languages. Some examples:

- Facebook used 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 used Objective-C, an interpreted language which has virtual methods that can surface "method not found" error. Then, Apple replaced that with Swift. However, both Objective-C and Swift use reference-counting for GC.

- Java was initially interpreted, but over time become 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.

Why people repeat the same mistakes when developing new programming languages? 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: to get something usable out quickly, then see whether people will use it or not ("Worse is better").

Reference counting is used in Swift for an additional reason. Though less efficient, reference counting is entirely local and can be used to automatically release other resources than memory, such as sockets or file descriptors.

4.2 Garbage Collection (GC)

As mentioned in the previous lecture, there are two main types of GC: reference-counting and mark-and-sweep. All those GC mechanism have the same goal, that is to manage resources 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 releases the previously allocated 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 ran for 1 second and released 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, in between these extremes, 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 kinds of GC: conservative and precise. Precise GC requires us to precisely differentiate pointer and value. If we cannot precisely identify pointers, 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.