# Chapter 9 - High-level language

## 9.0 Introduction

#### Jack language

Jack language is a simple, object-based language. If you used to have programming experience, you will find that Jack language has similar syntax style of Java language, but its syntax structure is much easier than many general high-level languages, such as C++, Java and so forth. But even if Jack is a simple language, it is still a Turing complete language. Jack language has the same computation ability as the other high-level languages, the computation ability of a universal Turing machine. In other words, any functions that can be implemented by other high-level languages, can be implemented in finite time and space cost by Jack language.

#### Learning target

In the next chapter 10 and 11, we will learn how to build a compiler that translate Jack program into VM code, and develop an operate system based on Jack/Hack in chapter 12, so in this chapter our study objective is becoming familiar with Jack language, which lays a solid foundation for further study.

In the Nand2Tetris courses our task is to invent an application using Jack language, but here we choose currently the most famous competitive game, the game of Go, as the content of this course.

#### The game of Go

The game of Go is a two-men strategy chess game, black and white players place the Stones on 361 grids intersected by 19 rows and 19 columns by turns, the one who surrounds more grids win. Usually in artificial regulation, black player should give compensation to white one for balance because black one can play in sente(first) with advantages. In chapter 9.4 we will formulate the rules of Go and how to implement with Jack language.

## 9.1 Background

#### Logic model

**Background**

1. Understand the whole structure of computer design.

2. Familiar with the functions that virtual machine language can implement.

3. Be able to code in virtual machine language.

**Target**

1. Understand the role and effect that high-level language plays a part in computer design.

2. Know how to convey the meaning of Jack language with virtual machine language.

3. Be able to code and debug Jack language program.

**Effect**

1. Be able to code a Go board program with Jack language.

2. Know the meaning one Jack program express and can debug.

**Output**

A Go board implemented by Jack language.

**Process**

1. Learn the basic theory of programming, such as: variable, sequence/loop/select structure、function and procedure.

2. Learn the syntax and API(application program interface), such as: available variable, interaction functions of Jack language.

3. Learn the data structures and algorithms needed in implementation, such as: stack and queue, breadth first search.

4. Learn the rules of Go game and corresponding abstract methods to realize, such as: the size of board, rules of placing Stones, way in count the end.

5. Programming and debugging.

**Input**

1. Introduction to high-level language.

2. Jack Operation System and its Application Program Interface.

3. Data structure and algorithm

4. Rules of Go game.

#### Why high-level language?

In our previous study, we keep using the bottom-up strategy in staging the whole computer architecture from the bottom electronic stage to the top operation system stage used for man-machine interaction. It clear that the former languages we learned such as HDL, Machine language, Assembly language and Virtual machine language, are all designed to be adapt to the electronic stage. That reminds the readers known well about computer of the era when ENIAC born, in which computer running needed specific operators going into the computer room and switching according to the program. However, this kind of design has considerable dependency on the bottom stage, what we have to know is that, once the design of bottom electronic stage changed, the computer mansion we built previously would be knocked down. So, we need an independent program, which can run on whatever lower structure, and that is when the high-level language born.

## 9.2 High-level language

### 9.2.1 What is high-level language

#### Definition

From now on we have to break our old thought, invent a language that benefits the conversation between human and machine with a top-down strategy. We’d better start from our human language. Thus, high-level language is a programming language based on human language, its code is easy for human to code, and is highly readable. Even varieties of high-level languages have difference with each other, they hardly rely on the hardware structure and instruction system, in that they present well. Nowadays almost all applications we see were directly programmed by high-level language, and in the following study we will know that the operate system is also the product of high-level language.

#### Category

Next, we will introduce 4 usual high-level language categories, more precisely, is the categories of programming language.

Functional programming language: Functional programming language is based on the model of math function, that from input to output. It considers program as the nest of functions. For example, Scheme.

Object-oriented programming language: Object-oriented programming language defines the concept of object, it considers that all operations from program should designed on an object. In the development in history, this kind of program thought is much more valuable than program itself, so now we hardly see the complete object-oriented programming language. The examples of supporting the object-oriented programming contain C++, Visual Basic, Java, C#.

Imperative programming language: Imperative programming language is a programming language based of the data access/manipulation model of the Turing machine, it considers program as an ordered set with unambiguous steps. For example, Ada, C, Pascal. It is worth noting that the low-level language we studied before, can be classified as the imperative programming language.

Declarative programming language: Declarative programming language is a language designed from the results, it has fixed code format, the programmer only need to input the available operations of the language according to format, and it will compute the output the programmer need with packaged methods, such as Prolog and SQL language.

And from the following study we will know that Jack is a object-oriented programming language.

### 9.2.2 High-level language: Jack

#### Syntax Elements

Jack language consists of many separated tokens, these tokens can be divided into 4 kinds: symbol, reserved words, constants and identifiers.

#### Program Structure

The basic program unit of Jack language, known as the object, named class. Class has the format below:

‘class’ Name {

Member fields and static variables clarification

Subroutine and sentences

}

The subroutine has the format below:

Subroutine type Name(Parameter list) {

Member fields and static variables clarification

Subroutine and sentences

}

#### Variables

Jack language has 3 data types as the basic variable types:

boolean: true or false.

int: 16 bits two’s complement notation integer

char: an Unicode character

And the classes that users defined in program.

Defining a variable start with the type of variable, such as ‘var int x;’, ‘filed char y;’

There are some expand on these types:

Array: Array in Jack language can only be one dimension, use sentence like ‘var Array x;’ to clarify, ‘let name = Array.new(length)’ to apply for length, ‘[subscript]’ to access. If we need multiple dimension Array, because the compiler does not support multiple ‘[]’ nesting, we can code multiple dimension subscripts as integer in multiple bases, such as ‘[i][j]’ to ‘[i \* n + j]’

String: Character string between double quotation marks are considered as a string in default, use sentence like ‘var String x;’ to clarify, use the methods in String class to operate.

Type switch: There are no clarifications of the result switching one variable type to another in Jack language rules, that will make the complier easier to implement. So we better not switch types directly.

Scope: Variables in Jack language have their scope, that the variables clarified can only be used in a specific range, there are 3 types in Jack language:

Picture 1: variable types

Note that the parameters are local variables.

#### Sentence

Jack language supports 5 usual sentences, definition and description are following:

Picture 2: senteces

#### Expression

We called a sentence ‘expression’ if it follow one of the following rules:

Constant

Variables in scope

Key word ‘this’, means the current object

Subroutine with returned value not void

‘Unary operator Expression’

‘Expression Binary operator Expression’

‘(Expression)’

Noting that Jack language has no definition with the order of the operator, so if you want your expression count multiplication operator before the addition operator, you should implement it in your own compiler.

#### Subroutine

Subroutine has 3 types of method, function and constructor, their syntactic format are below (If there are no parameters, reserve brackets):

‘method’ Name (Parameter type Parameter name, Parameter type Parameter name, …)

‘function’ Return type Name (Parameter type Parameter name, Parameter type Parameter name, …)

‘constructor’ Return type Name (Parameter type Parameter name, Parameter type Parameter name, …)

Among these, method has no returned value, function has a returned value that is not a class (can be void), constructor has a returned value that is a class.

Of course, we can use ‘Class name.Subroutine name(Parameters list)’ to call the subroutine in other class.

#### Jack Operation System and its Application Program Interface

There is a standard library inside Jack language, this standard library consists of 8 classes, the subroutines inside are recorded in Jack OS API.pdf

### **9.2.3 Other high-level language**

#### C/C++

C language was born in 1972 in Bell Labs in the United States, in fact, it was a programming language improved from the B language. Codes with C language usually have good portability, along with a wide range of variables and operations, which is basically consistent with the contract we proposed above. However, the data encapsulation of C language is poor, and in the compiler, syntax regulation is not stringent, which makes C codes have bad security, in order to be familiar with C language, programmer should have high level literacy. C++ was initially set as C with class, which means supporting data encapsulation and concept of object. It is undeniable that C and C++ are typical advanced programming language, and that is why they are not only the most widely used programming language, but also many programming lectures preferred language.

#### Java

The initial version of Java language, named Oak language, was a simple object-oriented programming language because of the contradiction between the highly complexity of C++ language and the lack resource of single chip embedded system. However, Oak language was not popularized at the time, because people nearly knew nothing about Oak language. Until Mosaic browser was born, people have an urgent need for a smaller, more portable programming language, which give Oak a chance to reuse. Java language is a completely object-oriented programming language, along with good encapsulation, inheritance and robustness. It is worth mentioning that the broad Java not only indicates Java language, but also the Java environment that load Java language, which is essentially a virtual machine. The portability of Java language is embodied in the different systems with the installation of the Java environment.

#### HTML Hyper-Text Markup Language

HTML belongs to no stage in the computer architecture, because it was designed for markup the display methods of data, so that people can read information on other devices easily. HTML is a markup language, which confirms the relation between data and its relevant information(display position, image size, etc.), another example for markup language is TeX. In fact, HTML was born before the Web browser, so the Web browser was designed to adapt for parsing HTML. In the learning of tokenization in chapter 10 we will find that the result of the lexical analyzer is similar to HTML code.

#### JavaScript

JavaScript language is a script language based but not belonging to object-oriented language that can translated directly, it is very similar to Java language, and often embedded in HTML. Script language defines rules, meaning that the computer only have to run according to the rules defined by script language. What is different from imperative programming language is that, script language was designed to shorten the traditional programming procedure of code, compilation, link and run, so generally script language need not to be compiled into lower language such as VM language or machine language, but to be translated directly and run the output. We can see that the script language skips the middle stages in our architecture, and script language indeed makes it easier to write and develop program. In theoretically, it should run faster than the high-level language with the process of compile execution. But we can not deny that it breaks the rules of our architecture, and in fact, when put into use, program do not have to be repeatedly complied, and the low level machine language must run fast than high level script language, so if one program is invoked repeatedly, the script language with the process of interpretive execution run slower than the imperative programming language with the process of compile execution.

## **9.3 Prepare knowledge**

Next we will discuss the concept of program, concretely speaking, program is the code we write in the language we learned. But from the design stage, we have to know that program is data structures and algorithms. We will introduce the concept of data structures and algorithms, and some simple examples that can used in implementing Go game.

### **9.3.1 Data Structure**

#### Definition

Data structure refers to the methods of storing, accessing data, or the set of structured data. Its structure means the relation and interaction between data, or some logic order between data items. Usually we can divide data structure into linear structure, semilinear structure or nonlinear structure by the complexity of logic order. Here we will clarify 2 typical linear structure: stack and queue.

#### Stack

We have already learned the concept of stack. It is a special linear container. We can define the only first element and last element, data push in and pop out can only be operating on a certain end. The end can operate is called stack top. The end can not operate is called stack bottom. According to this contract we know that stack is a structure satisfies last in first out.

Picture 3: Stack

#### Queue

Similar to stack, queue is also a special linear container. We can also define the only first element and last element. What is different from stack is that data push in and pop out should be on different end. The end can only push in is called queue tail, and the end can only pop out is called queue front. According to this contract we know that queue is a structure satisfies first in first out.

Picture 4: Queue

### **9.3.2 Algorithm**

#### Definition

Algorithm is a series of ordered, unambiguous, executable steps, it describes a specific computing process, tell the program how to meet our design contract, make the input and output corresponded.

#### Breadth First Search

Breadth first search algorithm is a traversal method defined on the graph. The structure of the graph is represented by two sets, the points set G and the edges set E. The element inside G is a single point x. The element inside E is a relation consists of 2 points, represented <x, y>, that means x and y are connected. The following algorithm gives the process of breadth first search:

1. Select a point x from G that is not visited yet. Push x into an empty queue and tag it visited.
2. Pop the queue front, and find out all relation <x, y> in E that x = front, if y is not visited, push y into queue and tag it visited.
3. Repeat step 2 until the queue is empty.
4. Repeat step 1, 2, 3 until all elements in G are visited.

Pseudocode:

for x belongs to G

if (vis[x] = false)

vis[x] <- true

queue.push(x)

while (~queue.empty())

for y that <queue.front(), y> belongs to E

if (vis[y] = false)

vis[y] <- true

queue.push(y)

queue.pop()

According to this process we know that every point in G will only be pushed in queue once, so if we can find a data structure that can help us find the relation <x, y> according to x, every relation in E will also be compute once. That makes breadth first search an algorithm with perfect performance.

#### KMP matching algorithm

KMP matching algorithm has a full name of Knuth-Morris-Pratt matching algorithm, which is an accelerating algorithm on computing string matching problem. This string matching problem is count how many times dose string T completely appear in string S. The basic idea of this algorithm is fully reuse the useful information from former brute force judging. KMP matching algorithm has a core of assist array fail, fail[i] is the maximum j that j < i and string S[0..j] is a suffix of string S[0..i](obviously S[0..j] is a prefix of S[0..i]). If j does not exist, we set fail[i] = -1.

There is an obvious conclusion that can help us compute fail:

If string S[0..j] is a suffix of string S[0..i], then S[j] = S[i] and S[0..j – 1] is also a suffix of S[0..i-1].

Fit fail[i] is the maximum j we know that:

For every fixed i, there exist only one certain sequence {A0, A1, A2, ..., Ak} that satisfy A0 = -1, Ak = i and for every 0 <= j < k, fail[Aj+1] = Aj, and also:

fail[i + 1] = fail[Aq] + 1

q is the maximum of j that 0 <= j < k and S[fail[Aj] + 1] = S[i + 1]

Based on conclusions above, we can design the process of KMP algorithm:

1. let fail[0] = -1. String S start from 0.
2. Let i from 1 to n – 1, n is the length of S.
3. Let p = fail[i – 1]
4. If ~(p = -1) and ~(S[p + 1] = S[i]), let p = fail[p]
5. Repeat step 4 until condition failed, if that time S[p + 1] = S[i] then let fail[i] = p + 1, else let p = -1;

Pseudocode:

fail[0] <- -1

for i from 1 to n – 1

p <- fail[i - 1]

while (~(p = -1) AND ~(S[p + 1] = S[i]))

p <- fail[p]

if (S[p + 1] = S[i])

fail[i] <- p + 1

else

fail[i] <- -1

It is clear that at the end of every loop p will increase 1 at most, so even p will decrease several times in a certain loop, the total times will less than n. So we can compute fail in a very good complexity.

#### Hash Algorithm

Hash algorithm is also a kind of matching algorithms, but it suits more conditions. Usually hash algorithm is used for judging whether 2 states are completely the same or not. The idea of Hash algorithm is to convert a larger, inconvenient storing state into a smaller, convenient storing state through a function. There are many ways to implement hash algorithm. Here we introduce a classic implementation:

1. We can write all the larger states that may appear in the form of a vector (x0, x1, x2, …, xn), and for each 0 <= i <= n, there exist a positive integer p that xi < p and xi is nonnegative integer.
2. Consider this vector as a bigger integer S = x0x1x2…xn in base p, then select a smaller prime number q, use S mod q to represent this vector.
3. Select several pairs of (pi, qi) and calculate Si mod qi separately, and store the results as a vector (y0, y1, y2, …, ym) that m << n(m is much smaller than n).

Now we can use a vector with m dimension to represent a vector with n dimension, the computation complexity is n \* m.

We have 2 conclusions from the process of hash algorithm:

1. For 2 vectors with n dimension S and T, if S and T are the same, then their results calculated by this hash function are the same.
2. For 2 vectors with n dimension S and T, if S and T are different, then their results calculated by this hash function may be the same in a small probability.

Now we have the concept of conflicts: For 2 different vectors with n dimension S and T, if their results calculated by this hash function are the same, then we call the pair of vectors <S, T> is a conflict.

From the number theory we know, when q is a prime number, pair of vectors <S, T> has a smaller probability to be a conflict.

Theoretically speaking, if the states number of the result vectors is larger than the number of larger states that may appear, we consider this hash algorithm has an ability to distinguish the larger states set.

## **9.4 Build a Go Board**

With the knowledge of data structures and algorithms, know we can start to build our own Go board. In the following chapter we will introduce how to realize a Go board that suits our design contract. But before the start, let us review our design contract:

Postconditions: Program can output the right state of the board on screen after every operation.

Preconditions: Operation from keyboard must be available, such as the Stone can not be placed out of the board.

Invariants: State of board must suits the rules of Go game after each operation.

Attention: sizes and coordinate in the following context, even this method of implementation, are reference only. You can modify this implementation and have your own personalization of Go board.

### **9.4.1 Build a Board without Stones**

#### Design Contract

Postconditions: Output a Go board with 361 grids intersected from 19 vertical lines and 19 horizontal lines, and bold 9 star grids along with a hint Stone on the left side, indicating which player next.

Preconditions: None.

Invariants: None.

#### Size

Coordinate of the top-left grid: (125, 9)

Interval between lines: 12

Radius of 9 star grids: 3

Radius of the hint Stone: 10

#### Picture

Picture 5

### **9.4.2 Place Stones on board**

#### Design Contract

Postconditions: Use an indicator to represents where we place the Stone, ‘Enter’ key represents placing a Stone here, and store the current state after every operation.

Preconditions: There is a Go board on the screen, we use keyboard to control the indicator, which makes the position legal.

Invariants: State displayed on screen and stored in the memory must be the same.

#### Store the state of board

One 19x19 array Grid with 2 dimensions(in fact stored as an array with 1 dimension) to record the state of board. We use -1 represents empty, 0 represents black Stone, 1 represents white Stone.

One int variable to record the state of hint Stone, 0 represents black Stone next, 1 represents white Stone next.

#### Size

Radius of black Stone: 6

Print a white Stone: a white circle with radius of 5 on a black Stone

#### Indicator

2 variables ClickX and ClickY to represents which row and column the indicator is.

Indicator is a black square on screen, length: 11

Each time we accept an ‘Enter’ key from keyboard, judge from the array Grid is there is a Stone already. If not then modify the state here and the state of the hint Stone, then modify the state on screen.

Each time we accept a direction key from keyboard, modify the location of the indicator(remember to judge boundary conditions), and modify the state on screen.

How to make the indicator blink? We use a variable cycle in a certain interval, if the variable in a subinterval of the certain interval we display the indicator, otherwise we do not display it. Attention: indicator should be output after the Stone, that makes the indicator on the screen.

Code:

method void Display() {

let cnt = cnt + 1;

if (cnt > 599) {

if (cnt = 600) { do DisplayPiece(ClickX \* Size + Click Y); }

if (cnt > 19999) { let cnt = 0; }

}

else { do DisplayClick(); }

return ;

}

Remember: This method to display indicator on the white Stone or empty grid has a good result, but still not very conspicuous on black one. Effect depends on the size of screen.

#### Picture

Picture 6

### **9.4.3 Rule 1: rule of liberty**

#### Liberty

The number of empty points that are directly orthogonally adjacent to the Stone. For Stones with the same color and connected with each other in the way above, we consider these Stones as a whole.

#### Rules of Liberty

If a whole has No liberty, it should be removed from board. We call a whole ‘connected Stones block’.

#### Design Contract

Postconditions: After every operation, there are no Stones that has no liberty and disturb the board.

Preconditions: A board support right placing Stone operation.

Invariants: After every operation, the connected Stones block should be adjacent to at least 1 empty point.

#### Implementation

In order to fine all connected Stones block, we should abstract every grid on board as the point in set G, and for pair of grids that are directly orthogonally adjacent with Stones in the same color, set relation <x, y> and <y, x> in edge set E. Then we run breadth first search on this graph, we can find every connected Stones block. If we search the grids that are directly orthogonally adjacent to the current state, and then judge if they have the same color, we can judge the surrounding situation of this connected Stones block.

#### Optimization

Noting that after every operation, next Stone will be placed by the opposite player. When the opposite player is placing, Stones in our color without liberty will not have unfair affection. So we can simplify our condition that there are no Stones in the opposite color without liberty. Now we only have to start our search for Stones in the opposite color.

### **9.4.4 Regret Operation**

#### Design Contract

Postconditions: A board that can regret.

Preconditions: A board supports placing Stones and rule 1.

Invariants: After each regret operation, state of board should be the same as the state right before the corresponding operation.

#### Implementation

If we abstract each operation to a data item, we will find that placing Stone along with removing operation and regret operation corresponds to push and pop operation of Stack. But since that the number of changes happened on board after each placing operation is uncertain, we need 2 Stacks:

One stores each change happened on board.

One stores how many changes happened after each placing.

In the actual storage, we push placing operation before removing operations. This method has an advantage that for our pop sequence, placing operation must be the last one pop out. Thus, we only have to store where each change happened. What kind of change happened is not necessary to store.

### **9.4.5 Clear the board and Exit**

#### Design Contract

Postconditions: The board can be cleared and stop.

Preconditions: A board supports regret, placing Stones and rule 1

Invariants: After clear operation, the state of board is same as the initialization.

#### Implementation

For exit, we only have to set a key from keyboard, when accepting that we break the loop of reading from keyboard.

For clear, since the default value we set is not equal to the default value of variable, so we have to initialize the state of Array(all memory set -1). We can put these initialization operations in a method function and bond it with a key from keyboard.

Remember: When programming, each time we clarify a variable, we should focus on its default value.

### **9.4.6 Rule 2: no Ko circle**

#### Definition

Since the empty point caused by rule 1 allows replace Stone again, so the state of board might be in circle. This is forbidden in the rules.

#### Design Contract

Postconditions: When the board go in circle, program should recognize this circle and stop this game.

Preconditions: A board supports all basic operations and rule 1.

Invariants: The state of board along with the sequence of operations could not appear twice.

#### Implementation

According to invariants, we have 2 methods:

1. From the state of boards: judging whether the states appears again.

We can record the state of board after each placing operation, and judge if this state has appeared yet. Since the scale of state is very large, we can shrink the scale of state by using hash algorithm.

How to avoid conflicts? Do not forget our regret operation! If we find some state has the same hashed state as the current state. We can use regret operation to see if it is a conflict or not.

1. From the sequence of placing operation: judging whether the sequence of placing operation appears again.

When going in a circle, we consider the place of placing operation as a character, then there must be a suffix like ‘SS’ stored in the Stack. ‘S’ is a string. We can enumerate the length of ‘S’ and use brute force to judge if this suffix appears.

In fact we can invert the sequence in Stack, then this sequence must have a prefix like ‘SS’, meaning that there exist an nonnegative integer k that fail[2 \* k + 1] = k, so we can use KMP algorithm to accelerate.

There is a data structure named suffix automaton we can use to store this sequence, that will accelerate our judging complexity twice. However, it is not easy to achieve this data structure, so we will not introduce this data structure.

### **9.4.7 Count the end**

#### Design Contract

Postconditions: A board can tell which player win in the end.

Preconditions: A board satisfy all conditions before.

Invariants: The sum of grids belongs to black and white equals 361.

#### Implementation

We have to follow 3 steps to count the end.

1. Pick out the dead Stones by human

Dead Stones means the stones that considered will not impact the game after negotiation. This kind of Stones should be removed from the board.

We can achieve this removing operation refers to placing operation.

Question: How to judge the dead Stone by computer? Currently artificial intelligence is a hot topic, but can we train our own artificial intelligence in the Jack/Hack system that without file read?

1. Rules to count occupied grids

When no dead Stones exist, if a grid was placed, the number of occupied grids in corresponding color + 1.

For the connected empty block, if it is surrounded by only one color Stones(ignore the side of board) then the number of occupied grids in corresponding color + the size of the block, else both numbers + the half size of block.

Implementation: Breadth first search.

1. Sente compensate rules

In the end, black player should give 3.75 grids for compensation to white one for balance because black one plays in sente(first) with advantages.

## 9.5 Perspective

### **9.5.1 Hints at Programming**

#### The size of screen and system error

Size of screen: 256 x 512

If output is beyond the screen, system will appear error 8, readers can learn all meaning of system errors from chapter 12.

#### Editor selection with Jack syntax highlight

Jack syntax highlight rules coded with the format of xml (\*.xml), so you better select an text ­­­­­­­­­­­editor that supports xml format.

#### Expression order

As mentioned above, Jack language have not an regulation on expression order, so A + B \* C might be (A + B) \* C.

#### VMEmulator

When using VMEmulator, remember to set Animate into No animation, or the program will run slowly.

### **9.5.2 How does high-level language work**

The working method of high-level language is to translate into low-level virtual machine language, this process is called compile. The initial state of program that just finished coding is called source program, while the program after compile called object program. The simplest complier consists 3 units: lexical analyzer, parser and code generator. Lexical analyzer can recognize which string from the source program represent a smallest unit ‘token’. Parser build a parse tree using these tokens. Code generator generates object program according to the parse tree. Currently the compilers we use are usually are complex, modified version, often with complex steps such as precompilation, code optimization and so forth, which can simplify programming, speed up the compilation process, and optimize the results of compilation. Whatever, compiler is a tool that translate high-level language to low-level language. They play a key role on the computer architecture, connecting the hardware stage based on electronic loop and the software stage based on man-machine interaction convenience. In the next chapter 10 and 11 we will detailly introduce how to build a compiler.

### **9.5.3 Compared to JavaScript/HTML version**

We can also use JavaScript and HTML to achieve a Go board, in the following context we will make a comparation between 2 methods:

Length of code: Jack language is a simple language, so it has high complexity on programming, which means long length of code. However JavaScript and HTML are widely-used programming language yet, so it has thorough standard class inside, and code is usually lighter.

Running speed: Jack language is running on 16 bits single thread computer we build, so it must be much slower than usual 32 or 64 bits multiple threads computer. That tells us that high-level language is independent from the basic stage of computer only on syntax, instead of running, so the best we to accelerate the efficient is to improve the lower stage.

Location: Jack language runs like most high-level language, but JavaScript and HTML are implemented completely different. In a sense, the HTML belongs to the underlying structure of the Web page, and the Web page relies on the entire computer architecture.

## 9.6 Glossary

High-level language, Jack, The game of Go, Data structure, Algorithm

## 9.7 Reference

1. Noam Nisan, S. S. (2005). The Elements of Computing Systems: Building a Modern Computer from First Principles. The MIT print.

2. Deng Junhui. (2013). Data structure: C++ language edition. 3rd edition. Tsinghua University press.

3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. (2009). Introduction to Algorithms. 3rd edition. The MIT print.

4. J. Glenn Brookshear. (2011) Computer Science: An overview. 11th edition. Addison Wesley.

5. Go(game). (28 December, 2017). Retrieved from Wikipedia, The Free Encyclopedia.: https://en.wikipedia.org/wiki/Go\_(game)

6. Project 9: High-Level Programming. (n.d.). Retrieved from nand2tetris. : http://www.nand2tetris.org/09.php

7. The Jack OS API.pdf. (n.d.). Retrieved from nand2tetris: http://www.nand2tetris.org/projects/09/Jack%20OS%20API.pdf

8. Lecture 9: High-Level Programming.pdf. (n.d.). Retrieved from nand2tetris: http://www.nand2tetris.org/lectures/PDF/lecture%2009%20high%20level%20language.pdf

## 9.6 Additional Reading Material

For other high-level language: Bruce A·Tate. (2010). Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages. Pragmatic Bookshelf.

For compilation theory: Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. (2006). Compilers. Addison Wesley.

For Suffix Automaton: Chen Lijie. (2012). Suffix Automaton.

For deeper Hash algorithm: Lv Kaifeng. (2013). Hash killer. Retrieved from blog.163.com: http://vfleaking.blog.163.com/blog/static/1748076342013229102430789/