# Chapter 9 - High-level language

## 9.0 介绍 Introduction

#### Jack语言

Jack语言是一种简单的，基于对象的语言。有过编程经验的读者必然会发现，Jack语言在语法风格上和Java语言非常相似，而本身的语法却比一般通用的许多高级语言，例如C++、Java等，要简单很多。然而尽管Jack非常简单，它却依然是一个图灵完备的语言。Jack语言和其它高级语言具有相同的计算能力，即一个通用图灵机的计算能力，换言之，其他高级语言可以实现的功能，Jack语言都可以在有限的空间和时间开销内实现。

#### 学习目标

在接下来的10-11章，我们将学习编写一个将Jack程序翻译成VM代码的编译器，而在12章开发基于Jack/Hack的操作系统，因此本章的学习目的在于让读者Jack语言的使用，以为后面的学习奠定坚实的基础。

在Nand2Tetris课程中我们的任务是实现一个以Jack语言开发的应用程序，而在这里我们将选择当下最热门的竞技游戏——围棋，作为我们的讲解内容。

#### 围棋

围棋是一种策略性的两人棋类游戏，黑白双方在一个横纵各19条线交叉形成的361个格点上交替行棋，以围地多的一方获胜。一般情况下由于黑棋先行处于有利一方，因此人为规定下黑方在终局时要给白方让出一定的地以达到平衡。在9.4节我们将系统地介绍围棋规则以及如何使用Jack语言实现。

## 9.1 背景 Background

#### 逻辑模型

背景：

1. 了解计算机设计的整体结构。

2. 熟悉虚拟机语言能够实现的功能。

3. 能够书写虚拟机语言代码。

目标：

1. 理解高级编程语言在计算机设计上的定位和作用。

2. 理解如何使用虚拟机语言来表述Jack语言。

3. 具备书写和调试Jack语言的能力。

效果：

1. 能使用Jack语言撰写一个围棋棋盘程序。

2. 能够明白Jack程序的意思并进行debug。

输出：

一个使用Jack语言实现的Go棋盘。

过程：

1. 学习基本编程理论。如：变量、顺序/循环结构、函数与过程等基本概念。

2. 学习Jack语言的语法规范和API。如：Jack语言中可用的变量、与屏幕交互的函数等。

3. 学习相应需要的数据结构和算法。如：栈与队列、宽度优先搜索等。

4. 学习围棋规则和抽象实现方法。如：棋盘规格、落子规矩、终局计算等。

5. 进行编程与debug。

输入：

1. 高级编程语言基本理论。

2. Jack操作系统及其用于应用的编程接口。

3. 数据结构与算法。

4. 围棋规则。

#### 为什么要有高级语言？

在先前的学习中，我们针对计算机整体从电子电路底层向上一直到人机交互的操作系统层的体系结构，一直采用的是自底向上的分层策略。不难发现我们先前学习的底层HDL、机器语言、汇编语言以及虚拟机语言，都是从电子电路设计层面设计的语言。如果对计算机有了解的读者必然会想起，在ENIAC诞生的那个年代，计算机的运行都需要专门的操作员进入机房根据程序员书写的程序语言进行电路接线。然而，这样的设计具有严重的底层依赖性，我们必须了解的一个事实是，一旦底层电子电路的设计发生了改变，先前我们搭建的计算机大厦会在顷刻间倒塌。因此，我们需要一个独立于底层的，即在无论什么样的CPU上都能够运行的编程语言，而高级语言就在这里应运而生。

## 9.2 高级语言 High-level language

### 9.2.1 什么是高级语言

#### 定义

现在我们需要跳出之前的思维，自顶向下来设计一种利于人类与计算机对话的语言，那不如就从人类本身的语言开始。因此，高级语言是一种基于人类语言的编程语言，代码易于人类编写，且具有较高的可读性。纵使各种各样的高级编程语言各有各的不同，它们都已经基本做到与硬件结构和指令系统无关，因而具有较强的表达能力。而现代我们所接触到的各式各样的软件，都是直接通过高级语言编写而成的，甚至在之后的学习中我们也会知道，操作系统本身也是高级语言的产物。

#### 分类

接下来我们会介绍四种常见的高级语言分类，更确切的说，是编程语言的分类

函数式编程语言：函数式编程语言是基于函数模型产生的语言，即从输入对应输出，它将程序看作一个又一个的函数嵌套起来。例如：Scheme。

面向对象式编程语言：面向对象式编程语言定义了对象这一概念概念，它认为程序的所有操作都应针对对象而言。但在历史发展的进程中，这种编程思想的价值远远超过了其语言本身的价值，因而在现代，很少有到完全的面向对象式编程语言。支持面向对象式编程的语言例子有：C++、Visual Basic、Java、C#

命令式编程语言：命令式编程语言是基于图灵机的数据访问/操作模型设计的编程语言，它将程序的运行视为具有时序性的、无歧义的步骤。例如：Ada、C、Pascal。值得注意的是，我们先前学习的低级语言，也可以视为命令式编程语言。

宣告式编程语言：宣告式编程语言是从结果出发而设计的语言，它具有固定语句格式，程序员只需要根据格式输入它支持的操作，它就会利用已经封装好的处理方法给出程序员需要的结构，常见的例子有：Prolog、SQL语言。

在接下来的学习中我们将明白，Jack语言是一种面向对象式的语言。

### 9.2.2 高级语言：Jack

#### 语法要素

Jack语言是由许多分隔开的字元组成的，其中字元分为符号、保留字、常数和标识符四种。

#### 程序结构

Jack语言的基本编程单元，也就是对象，叫做类。类的格式如下：

class 名称 {

成员字段和静态变量的声明

子程序与语句

}

其中子程序具有以下格式：

子程序类型 子程序名称(参数列表) {

成员字段和静态变量的声明

子程序与语句

}

#### 变量

Jack语言的基本变量类型为三种数据类型：

boolean：真或假

int：16位2进制补码表示的整数

char：Unicode字符

以及程序中用户自己创建的对象类。

变量的定义使用变量类型开头，如var int x; field char y; 等。

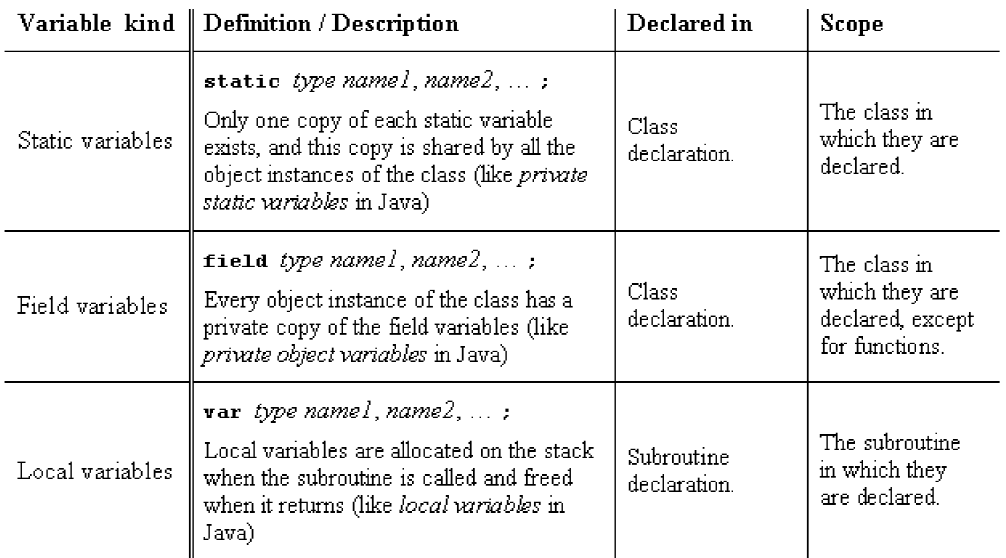
在这些类型上有扩展类型：

数组：Jack语言中的数组只针对一维情况，使用 var Array x; 来定义，let 变量名 = Array.new(长度) 来申请长度，用 [下标] 来进行访问。如果要使用多维数组的话，由于编译器不支持多[]嵌套，可以对多维下表进行多进制数编码，如将 [i][j] 化成 [i n + j] 。

字符串：Jack语言中的双引号内的字符串将被默认为String类，使用 var String x; 来定义，利用 String 类的方法进行操作。

类型转换：Jack语言规范中并没有定义从某种类型转为另一种类型的结果，这样会方便后面编译器的书写。

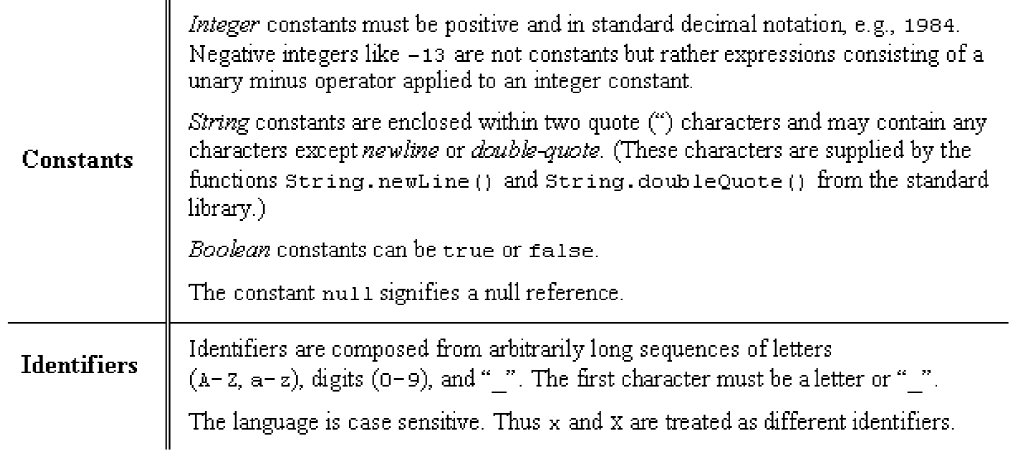
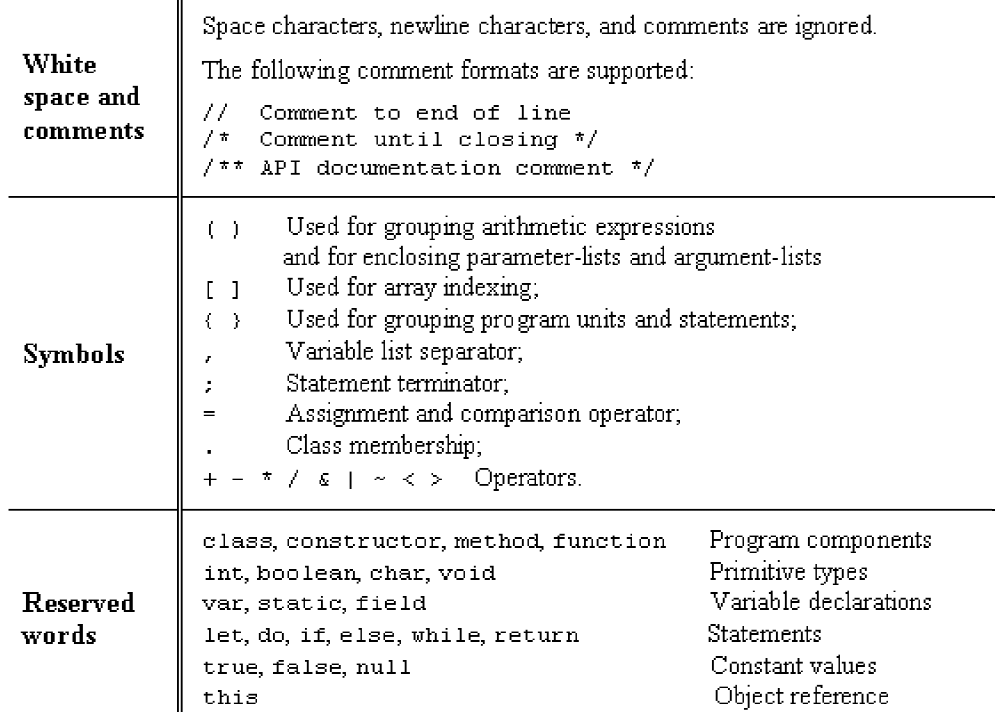
作用域：Jack语言中的变量都有作用域，即声明的变量只有在一定范围内才能使用，常见的变量有三种类型。



注意子程序的参数列表属于var类型。

#### 语句

Jack语言支持五种常见的语句形式，定义和描述如下：



#### 表达式

符合以下规则的式子为适用于Jack的表达式：

常数

在作用域内的变量

关键字this，引用当前对象

返回值不为void的子程序调用

一元运算符 表达式

表达式 二元运算符 表达式

(表达式)

值得注意的是，Jack语言并没有运算符优先级的规定，即是否支持常见的乘除优先于加减取决于你自己设计的编译器。

#### 子程序

子程序分为方法、函数和构造对象三类，语法形式如下。若没有参数，则保留括号：

method 子程序名 (参数类型 参数名, 参数类型 参数名, ...)

function 返回类型 子程序名 (参数类型 参数名, 参数类型 参数名, ...)

constructor 返回类型 子程序名 (参数类型 参数名, 参数类型 参数名, ...)

其中，方法类子程序没有返回值，函数的返回类型为不是类的变量，构造对象的返回类型为一个类。

当然，我们可以通过 类名.子程序名(参数列表) 来调用其它类中间的子程序

Jack操作系统用于应用编程的接口

Jack语言中内置了一个标准库，这个标准库由八个类组成，其内部子程序列表请参考Jack OS API.pdf

### **9.2.3 其它高级语言**

#### C/C++

C语言于1972年诞生于美国贝尔实验室，事实上，它是根据B语言改良后得到的一种编程语言。C语言的代码往往具有良好的移植性，并且拥有较为广泛的变量类型和运算操作，这与我们先前提出的合约基本一致。然而，C语言的代码的数据封装性较差，而在编译器的设计上也对语法没有特别严格的要求，这使得C语言的代码安全性较低，要熟练运用C语言需要很高的编程素养。C++最初的设定是C with class，即在C语言的基础上支持数据封装和对象概念。不可否认的是，C和C++具有典型的高级编程语言的特点，不仅是目前应用最广泛的编程语言，也是众多编程课程喜爱的教学语言。

#### Java

Java语言的最初版本名为Oak语言，是针对C++语言的高复杂性与硬件资源匮乏的单片嵌入式系统之间的矛盾建立的一种较为简单的面向对象式编程语言。然而，Oak语言在当时并未普及，因为当时的人们对Oak还一无所知。直到Mosaic浏览器的诞生之时，人们迫切地需要一种规模较小的、便于传输的编程语言，这给了Oak重生的机会。Java语言属于一种完全的面向对象式语言，具有良好的封装性、继承性和稳健性。值得一提的是，广义的Java不仅仅包括Java语言，还有搭载它运行的Java环境，本质上它也是一种虚拟机，因而Java语言的可移植性体现在安装了Java环境的不同系统上。

#### HTML超文本标记语言

HTML语言并不属于计算机系统结构中的任何一层，因为它的设计初衷是标记存储数据的显示方式，以方便人们通过互联网阅读其它设备上的信息。HTML语言属于一种标记语言，标记语言是将存储数据与存储数据的相关信息(如显示位置、图像大小等)进行对应的语言，另一个常见的标记语言的例子是TeX。事实上，HTML诞生于网页浏览器之前，因此当时网页浏览器的设计都是针对解析HTML语言来设计的，在第十章对token的学习中我们会发现，经过分词器得到的结果和HTML语言的代码非常相似。

#### JavaScript

JavaScript语言是一种直译式的脚本语言，它和Java语言非常相似，常内嵌于HTML语言中，基于但不属于面向对象式编程语言。脚本语言定义的是一种规则，即计算机只需要根据脚本语言定义的规则进行工作即可，与命令式编程语言不同的是，脚本语言产生的初衷是为了缩短传统的高级语言编写-编译-链接-运行过程所开发出来的编程语言，因而一般情况下脚本语言无需编译成下一级的虚拟机语言或机器语言，而是直接解释并输出运行结果。可以看到，脚本语言在体系中实际是一种跨越分层结构的系统，这虽然让程序的编写和开发便捷了很多，且从理论上而言它应该比编译执行的高级语言运行速度块，但不可否认的是它没有遵循我们设计的计算机体系结构，而且事实上，程序在投入使用时并不需要反复编译，由于低级的机器语言运行速度必然快于解释执行的脚本语言，因此如果一个程序被反复调用的话，解释执行的脚本语言的处理速度反而会慢于编译执行的命令式语言。

## **9.3 预备知识 Prepare knowledge**

接下来我们讨论程序的概念，从实际层面上而言，程序即使用我们之前设计的语言编写出来的代码。但从设计的层面上而言，我们必须要了解的一个公式是：程序 = 数据结构 + 算法，接下来我们会介绍数据结构与算法的概念，并引入一些为后面围棋实现做铺垫的例子。

### **9.3.1 数据结构**

#### 定义

数据结构指的是程序存储、读取数据的方法，是数据项结构化的集合，其结构性表现为数据项之间的相互关系及作用，也可以理解为定义于数据项之间的某种逻辑次序。一般依据这种逻辑次序的复杂程度，可以将数据结构划分为线性结构、半线性结构和非线性结构三种。在这里我们选取两个典型的线性结构：栈与队列，来进行阐述。

#### 栈

在之前的学习中我们已经接触过栈的概念，栈是一类特殊的线性容器，我们可以定义唯一确定的首、末元素，并且数据的入栈和出栈只能在一端进行，可以操作的这一端称为栈顶，不可操作的另一端叫做栈底。从这份合约中不难看出，栈是一个满足后入先出的结构。

图

#### 队列

与栈类似，队列也是一类特殊的线性容器，我们也可以定义唯一确定的首、末元素，与栈不同的是，数据的入队和出队只能不同端进行，只能出队的这一端称为队首，只能入队的这一端称为队尾。从这份合约中不难看出，队列是一个满足先入先出的结构。

图

### **9.3.2 算法**

#### 定义

算法是一系列有序的、明确的、可操作的指令集，它描述了一个特定的计算过程，告诉程序如何满足我们的设计合约，将输入与输出对应起来。

#### 广度优先搜索

广度优先搜索算法是定义在图上的一种遍历方法，图的结构由两个集合表示，点集合G和边集合E，其中点集合内的元素为一个单独的点x，边集合中的元素为一对点构成的关系<x,y>，这表示由x可以扩展到y。下面的算法给出广度优先搜索的流程：

1. 每次从点集G中取出一个没有访问过的点x，并将x插入一个空队列，标记x为访问过的点。

2. 每次取出队头front，并找出边集合E中的所有关系<x,y>满足x = front，若点y没有被访问过，则将点y插入这个队列，并标记y为访问过的点。否则跳过此关系。

3. 重复2操作直至队列为空。

4. 重复1、2、3操作直至点集G中的所有点都被访问过。

伪代码如下：

伪代码

由这个算法流程不难看出，点集G中的每个点只会进入队列一次，而如果我们能够找到一个数据结构帮助我们快速找到步骤2中的所有关系<x,y>，则每个关系也只会被计算一次。这样的实现方式显然会比枚举两个点x和y并判断关系要具有更小的复杂度。

#### KMP匹配算法

KMP匹配算法全称为Knuth-Morris-Pratt匹配算法，是在计算字符串匹配问题上的一个加速算法，字符串匹配问题指的是在某个固定的模板字符串S中一个给定的字符串T出现的次数问题，其基本思想基于合理利用之前暴力判别留下来的有效信息。KMP匹配算法的核心在于一个辅助数组fail，其中，fail[i]表示最大的j满足j < i，且对于字符串S[0..i]，字符串S[0..j]既是S[0..i]的前缀(显然)，也是它的后缀。若这样的j不存在，则定义fail[i] = -1

辅助数组fail的计算方法是迭代法，它基于以下这个很显然的结论：

若 j = fail[i], 则S[j] = S[i]，去掉两个字符串的最后一位可以得到：S[0..j - 1]必然也是S[0..i - 1]的前缀和后缀。

将fail[i]为最大的j这一条件带入，可以得到以下定理：

对于一个给定的i，能找到一个唯一确定的序列{A\_0, A\_1, A\_2, ..., A\_k}满足A0 = -1，Ak = i，且对于任意 0 <= j < k 满足 fail[A\_{j+1}] = A\_j，且满足：

fail[i + 1] = fail[A\_p] + 1

其中 p 为最大的 j 满足 0 <= j < k 且 S[fail[A\_j] + 1] = S[i + 1]

有了以上定理，我们就可以设计出KMP算法的流程了：

1. 定义fail[0] = -1，字符串下标从0开始。

2. 将i从1循环到n - 1，依次计算fail[i]：

3. 令p = fail[i - 1]

4. 当p != -1 且 S[p + 1] != S[i]时，令p = fail[p]

5. 重复4操作直至某一条件满足，若S[p + 1] = S[i]则令fail[i] = p + 1，否则令fail[i] = -1

伪代码如下：

伪代码

由这个算法不难看出，每次循环开始时p会不定次数的向前移动若干位，不过由于每次循环结束时p每次只会向后移动一位，因此p向前移动的次数不会多于向后移动的次数的常数倍。这样的实现方式显然会比从小到大枚举fail[i]来计算要具有更小的复杂度。

#### 哈希算法

哈希算法也属于一种匹配算法，不过它的作用范围更广，它用于判断两个较大规模的状态是否完全相同。哈希算法的思想是通过某种函数将规模较大的、不方便存储的状态转化成规模较小的、方便存储的状态。哈希算法的实现方式有很多种，这里介绍比较经典的一种实现方式：

1. 我们将所有可能出现的规模较大的状态书写成向量的形式(x0, x1, x2, ..., xn)，且对于 0 <= i <= n，满足 xi < p 且 xi 和 p 都是非负整数。

2. 将这个状态看作一个较大的p进制数S = x0x1x2...xn，并取一个较小的质数q，用S mod q来代替这个向量。

3. 选出若干对(pi,qi)分别计算Si mod qi并将其存储为一个向量(y0, y1, y2, ..., ym)，其中满足 m << n (m远小于n)

这样我们就可以用一个m维的向量来表示一个n维向量了，计算的S mod q的复杂度与n成线性关系。

由哈希算法的过程我们可以得到以下两个结论：

1. 对于两个相同的n维向量S和T，经过哈希函数转化后的m维向量一定相同。

2. 对于两个n维向量S和T，若它们经过哈希函数转化后的m维向量相同，则S和T有很大概率是相同的。

接下来定义冲突的概念：即对于两个不同的n维向量S和T，它们经过哈希函数转化后的m维向量相同，则称n维向量对<S,T>为一个冲突。

由数论知识可得，当模数q为一个质数的时候，向量对<S,T>为冲突的概率最小。

理论上，如果转化后的向量能够表示的状态数大于围棋棋盘本身可能出现的状态数，则这个哈希算法具有区别所有棋盘的能力。

## **9.4 搭建围棋棋盘 Build a GoBoard**

有了数据结构和算法的支持，接下来我们就能开始搭建我们自己的围棋棋盘了。在接下来的过程中我们将介绍如何实现一个满足设计合约的围棋棋盘，不过在此之前，让我们先梳理一下我们的设计合约：

后继条件：

每次操作后能够在屏幕上输出正确的棋盘状态。

先决条件：

每次操作通过键盘输入且必须合法，如棋子落下的位置不能超出棋盘。

不变量：

每次操作后棋盘的状态必须满足围棋的规则。

注：以下内容中出现的规格和坐标，甚至整个实现方法本身，只是一种参考，读者可以根据自己的想法进行适当的修改。

### **9.4.1 搭建一个没有棋子的围棋棋盘**

#### 设计合约

后继条件：在屏幕上输出一个由横竖各19条线构成的361个格点组成的棋盘，其中将9个星点突出表示，并在屏幕的左侧显示一个提示棋子，表示下一个棋子由哪一方下。

先决条件：无

不变量：无

#### 规格

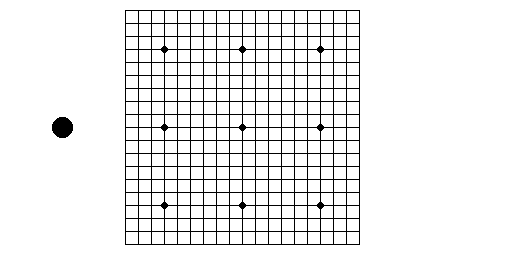
左上角的点坐标：(125, 9)

线与线之间的距离：12

9星突出点的半径：3

提示符半径：10

#### 效果图



### **9.4.2 在棋盘上放子**

#### 设计合约

后继条件：利用一个指示器代表下棋位置，回车键表示在此放子，并在每次操作后将棋盘的状态实时进行存储。

先决条件：屏幕上有一个棋盘，通过键盘控制每次下棋的位置，且位置需要合法。

不变量：屏幕上的状态必须与内部的存储状态正确对应。

#### 棋盘状态存储

一个19x19的二维数组(实际上是一维数组)来对应棋盘的状态，其中-1表示没有棋子，0表示有黑棋，1表示有白棋。

一个int来对应提示符的状态，0表示下一颗棋子为黑棋，1表示下一颗棋子为白棋。

#### 规格

黑棋半径：6

白棋：黑棋上输出半径为5的圆形

#### 指示器

两个变量ClickX和ClickY来表示指示器位于第几行第几列。

指示器在屏幕上显示为一个黑色的小方格，边长为11。

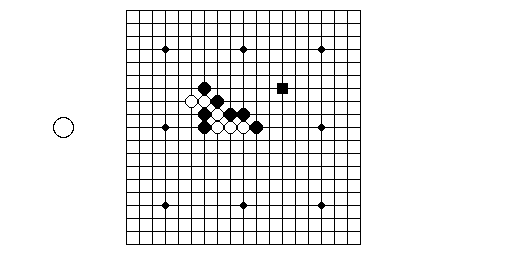
每次读取到键盘输入的回车时，根据二维数组判断这个位置是否已经有棋子了，如果没有就修改这里的状态，并修改提示符的状态，然后将屏幕上的状态进行对应的修改。

每次读取到键盘输入的方向键时，修改指示器的方位(注意判断边界条件)，并将屏幕上的状态进行对应的修改。

如何让指示器闪烁呢？我们用一个变量在一定区间内循环，如果这个变量在这个区间的某个子区间内我们就将指示器的状态，否则不输出指示器的状态。请注意，指示器的输出应该放在棋子输出之后，即指示器应该位于棋子上方。

注：这样实现指示器的方法在白棋上的效果很好，但在黑棋上还是有些不清晰，请记住：屏幕大小会直接限制程序的显示效果。

#### 效果图



### **9.4.3 规则1：气尽提子**

#### 气的概念

对于一个棋子，与它直接正交相邻的点中，没有放置棋子的点的个数称为气。对于直接正交相邻的相同颜色的棋子，我们需将其看作一个整体考虑。

#### 气尽提子

若一个整体没有气了，则它应该被自动从棋盘上移走。我们称一个整体为联通块。

#### 设计合约

后继条件：每次操作后，棋盘上不能存在没有气的棋子干扰棋盘。

先决条件：一个能够支持正确放棋子的棋盘。

不变量：每次操作后，对方的所有棋子所形成的联通块必须与空点相连。

#### 实现

为了找到所有联通块并检测其边界，我们将棋盘上的每个点抽象成图的点集G，并将已经放置同色棋子的相邻两点x和y建立关系<x,y>和<y,x>，接下来我们只需在这个图上进行广度优先搜索，就可以找到每一个同色联通块。我们在搜索算法中通过先判断两个点是否相邻，再判断两者颜色是否相同，就可以针对每一个联通块判断它的边界情况。

#### 优化

注意到由于操作完后下子权交给对方，而对方下子时我方的没有气的棋子不会对棋盘造成不利于对方的影响，条件可以简化为棋盘上不能存在对方的没有气的棋子。这样我们在每次操作之后只需要对对方颜色的棋子进行搜索即可。

### **9.4.4 悔棋操作**

#### 设计合约

后继条件：一个可以满足先前条件且可以悔棋的棋盘。

先决条件：一个支持规则1的可以下棋的棋盘。

不变量：每次悔棋操作之后，棋盘的状态和对应操作的前一个状态完全相同。

#### 实现

如果我们将棋盘状态抽象成一个数据，那么会发现下棋操作和悔棋操作正好对应栈的入栈操作和出栈操作。但由于每次下棋后棋盘上状态改变的数量是不确定的，因此我们需要两个栈结构：

一个栈存储棋盘上单个状态的改变的位置。

一个栈存储一次下棋后状态的改变次数。

在实际存储时，我们先将下棋改变加入栈中，然后将提子改变加入栈中，这样做的好处是每次需要悔棋时，我们指导弹出来的改变中，最后一个改变是下棋改变，其它都是提子改变。这样我们就只需记录每次改变发生的位置，而不用记录具体发生了什么类型的改变。

### **9.4.5 清空棋盘和退出程序**

#### 设计合约

后继条件：棋局能够重新开始和中途终止。

先决条件：一个支持规则1且可以悔棋、可以下棋的棋盘。

不变量：重新开始之后，棋盘状态和程序运行的初期相同。

#### 实现

对于退出操作，只需要专门设置一个按键，在按下时退出从键盘读入的循环即可。

对于重开操作，由于我们在定义上棋盘的初始值与默认的初始值不一样，因而在最开始我们需要将保存棋盘状态的数组重新初始化。将所有的初始化过程放入一个方法子程序中，并固定一个按键即可。

值得注意的一点是：在进行编程时，我们每定义一个变量，都应当具有关注初始值的好习惯。

### **9.4.6 规则2：禁止循环**

#### 定义

由于提子后留下的空点可以被重新下子，因而棋盘状态会出现循环的状态，这在围棋棋局中是不允许的。

#### 设计合约

后继条件：当棋盘进入状态循环时，系统应当识别出循环并终止程序。

先决条件：一个支持所有基本操作和规则1的棋盘。

不变量：棋盘状态不能重复出现，且每次下棋位置的序列也不能连续重复出现。

#### 实现

根据不变量，我们可以提出两种实现方法

1. 从棋盘状态出发：判断状态是否重复出现

我们可以将每次下棋后棋盘的状态存储下来，并判断这个状态是否出现过。而棋盘的规模相当大，因此我们可以使用哈希算法将规模缩小。

不要忘记我们之前实现的悔棋操作，当我们发现某个先前出现过的状态和当前状态哈希后的结果相同时，可以利用悔棋操作还原先前的那个棋盘判断它们是否是一个冲突。

2. 从下棋序列出发：判断下棋序列是否重复

当循环发生时，我们将先前栈中存储的下棋的位置看作字符，则栈中存储的序列具有一个形如SS的后缀，S为一个字符串。我们可以通过枚举S的长度进行暴力扫描判定这样的后缀是否存在。

实际上我们可以将栈中存储的序列倒过来，则这个序列必有一个形如SS的前缀，即存在一个非负整数k使得fail[2 k + 1] = k，这样判定会比暴力扫描法更加快速。

事实上，如果我们使用一种叫做后缀自动机的数据结构存储这个下棋序列，可以将判断的复杂度降到更低。但这种实现数据结构具有一定困难，因此在这里我们不进行叙述。

### **9.4.7 计算终局**

#### 设计合约

后继条件：一个满足能够判断出结束时哪一方获胜的棋盘。

先决条件：一个满足先前所有条件的棋盘。

不变量：双方的占点数之和为361

#### 实现

计算终局需要按顺序进行三个步骤

1. 人工剔除死棋：死棋是指双方协商后认为棋局接着进行下去不会为己方产生占点数产生贡献的棋子。这样的棋子应在计算时从棋盘上提走。

我们可以通过类似于下棋操作的方法实现删除棋子的操作。

思考问题：计算机如何判别死棋？现在人工智能发展非常火，但如何在没有文件输入的Hack系统中训练自己的围棋人工智能呢？。

2. 占点计算规则

在没有死棋的情况下，如果一个点被放置了棋子，那么对应颜色方的占点数+1

对于每个由空点构成的联通块，判断它的边界，如果边界完全由一种颜色的棋子构成，则对应颜色方站点数+这个联通块的大小，否则双方各+联通快大小/2。

实现：广度优先搜索

3. 先手让子规则

由于黑方先手具有一定优势，因此在计算终局的时候，一般规定黑方需要贴出3.75子让给白方以达到游戏平衡。

## 9.5 一些想法 Perspective

### **9.5.1 编程时的建议**

#### 屏幕大小问题和系统错误提示

屏幕大小：256 x 512

如果画线的时候超出了屏幕边界，系统会提示错误8，读者可以通过12章了解到各个系统错误提示的含义。

#### 编辑器选取问题与Jack语法高亮

Jack语言的高亮规则是编码为xml格式的文件，因此最好选择一个代码高亮支持xml格式的文本编辑器。

#### 表达式优先级问题

如前文所述，Jack语言并没有具体规定表达式的优先级，因此 A + B C 的结果很可能是 (A + B) C

#### 模拟器使用问题

在使用VMEmulator时，请记得将Animate设置为No animation，否则程序的运行速度会特别慢。

### **9.5.2 高级语言是如何工作的**

高级语言的工作方式是将其转化为低级的虚拟机语言，而这个转化过程称之为编译。程序在编写完毕后具有的初始状态称为源程序，编译完毕后的程序称为目标程序。最简单的编译由词法分析器，语法分析器和代码生成器三个单元构成，其中，词法分析器用于识别程序中哪些字符串构成最小单元，语法分析器利用最小单元生成语法树，代码生成器根据语法树生成目标代码。然而现代我们使用的编译器大多都是经过复杂的改良后的版本，它们往往还具有预编译、代码优化等复杂的步骤，这些步骤可以起到简化编程、加快编译过程和优化编译结果等效果。不过无论怎样，编译器都是将高级语言转化为低级语言的工具，它们在整个计算机系统中起到了桥梁般的作用，使得我们先前搭建的基于电子电路的硬件层和基于人类交互方便的软件层连接了起来。在接下来的第10、11章我们将进行详细的介绍编译器的实现方式。

### **9.5.3 与JavaScript/HTML语言实现版本的比较**

我们同样也可以使用JavaScript和HTML语言来实现围棋棋盘，下面对两种实现版本进行一个比较：

文件大小：由于Jack语言是比较简单的语言，因此它的变成复杂度较高，代码量也较大，而JavaScript和HTML语言已经是广泛应用的开发语言，因此它的封装库已经相当完善，代码也较为简便。

运行速度：Jack语言的运行平台是我们一层层搭建上来的16位单线程计算机，因而在运行速度上不及个人常用的32或64位多线程计算机上运行的JavaScript和HTML语言。这告诉我们，纵使高级语言在设计上是期望脱离计算机底层的束缚的，但它的运行依旧是依赖于计算机底层的，因此提升整个计算机运行的最有效的方法，还是从底层电子电路的实现上下手。

实现原理：Jack语言的运行与一般的高级语言基本一致，而JavaScript和HTML语言的实现是完全不同的，某种意义上，HTML语言本身就属于网页结构中的底层结构，而网页结构本身是依赖于整个计算机结构之上的。

## 9.6 关键词 Glossary

高级语言 Jack 围棋 数据结构 算法

## 9.7 参考文献 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 深入阅读资料 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.