# Assignment

- 初始化
- 轉譯常數與變數宣告
- 生成 expr與stmts code
- 生成 if-else & loop code
- 生成 func-invoke code

# Language Restrictions

- 沒有 READ 敘述
- 沒有 array宣告&使用
- 沒有 float type
- 沒有 string type

# Submit

- 編譯器
- 文件描述
- Makefile
- 測試程式

# Java as

Java Bytecode Assembler（或簡稱 Assembler）是一個程序，它可以轉換用“Java”編寫的代碼
彙編語言”轉換為有效的 Java .class 文件。 它可用於非商業目的。 該程序可以在班級主頁上下載。 可以在以下位置找到 Java 指令的在線參考

http://en.wikipedia.org/wiki/Java_bytecode_instruction_listings。

# Generate

本節描述了將 Kotlin* 程序翻譯成 Java 的四個主要部分 (參見第 1 節) 彙編代碼。
本文件介紹了每個部分的代碼生成方法，並提供了 Kotlin* 範例 以及 Java 彙編代碼。
請注意，示例中的標籤編號是任意分配的。
此外，將添加一個額外的部分來生成用於初始化的代碼。

# initialize

```
class example {
  fun main() {

  }
}
```

將會生成

```
class example
{
  method public static void main(java.lang.String[])
  max_stack 15
  max_locals 15
  {
    return
  }
}
```

因此，一旦定義了程序的名稱，就必須聲明相應的class name被生成。
此外，必須生成聲明為 public 和 static 的 main 方法。

# Declarations for Variables and Constants

在為 Kotlin 語句生成 Java 彙編命令之前，您必須為聲明的變量分配存儲空間並存儲常量的值。

## Allocating Storage for Variables

變量可以分為兩種類型：全局變量和局部變量。 所有在函數外部聲明的變量都是全局變量，而其他變量是局部變量。
所有全局變量必須在函數聲明之前聲明。

# Global variables

全局變量將被建模為 Java 彙編語言中的類字段。
字段將在類名聲明之後立即聲明。
每個全局變量 var 將通過表單聲明為靜態字段

```
field static type var <=const_value>
```

原始輸入

```
var a: int
var b = 10
var c: int
```

翻譯輸出

```
field static int a
field static int b = 10
field static int c
```

# Local variables

Kotlin* 中的實例變量將被轉換為 Java 彙編中方法的局部變量。
與字段（即 Kotlin*-* 的全局變量）不同，局部變量不會在 Java 彙編程序中顯式聲明。
相反，局部變量將被編號，並且引用局部變量的指令採用整數操作數指示要使用的變量。
為了給局部變量編號，符號表應該維護一個計數器來指定 "下一個可用的數字"。
例如，考慮以下程序片段：

```
var i: int
var j: int
var k: int
```

```
entering block, next number 0
  i = 0, next number 1
  j = 1, next number 2
  k = 2, next number 3
leaving block, symbol table entries:
  <"i", variable, integer, 0>
  <"j", variable, integer, 1>
  <"k", variable, integer, 2>
```

此外，如果給定了一個初始值，則必須生成語句將該值放入操作數堆疊，然後將其存儲到局部變量中。
例如，如果上面示例的最後一條語句更改為：

```
var k: int = 100
```

將會生成

```
sipush 100
istore 2
```

# Storing Constants in Symbol Table

Kotlin 中的常量變量不會被轉換為 Java 彙編中的字段或局部變量。
這常量變量的值將存儲在符號表中

# Expression

表達式可以是變量、常量變量、算術表達式或boolean表達式。

## vars

由於不考慮對字符串變量的操作以及Java虛擬機沒有boolean指令
一個變量將通過 iload 指令加載到操作數堆棧，如果
如果它是全局變量，則它是局部變量或 getstatic。

考慮以下程序片段

```
class example {
  var a : int
  ...
  fun foo ( ) : int {
    var b: int
    ...
      = a ...
      = b ...
    ...
  }
}
```

生成

```
getstatic int example.a

iload 1
```

## Consts

在 Java 虛擬機中加載常量的指令是 iconst value 或 sipush value，
如果常量是一個boolean值或一個整數，或者 ldc string是常量，是一個字符串

考慮以下程序片段

```
val a = 10
val b :bool = true
val s :string = "string"
  ...
    = a ...
    = b ...
print s
   = 5 ...
```

生成

```
sipush 10

iconst_1

ldc "string"

sipush 5
```

## Arithmetic and Boolean Expressions

一旦編譯器對算術表達式或Boolean表達式執行歸約，操作的操作數將已經在操作數堆棧上。因此，只會生成運算子。

下表列出了 Kotlin* 中的運算符與 Java 彙編語言中的相應指令之間的映射。

| operator | command |
| --- | --- |
| + | iadd |
| - | isub |

| operator | command |
| --- | --- |
| * | imul |
| / | idiv |
| % | irem |
| -a | ineg |
| & | iand |
| \| | ior |
| ! | ixor |

具有關係運算符的表達式將由減法指令建模，然後是條件跳轉。
例如，考慮 `a < b`

```
    isub
    iflt L1
    iconst_0
    goto L2
L1: iconst_1
L2:
```

The following table summarizes the conditional jumps for each relational operator:

| operator | command |
| --- | --- |
| < | iflt |
| <= | ifle |
| > | ifgt |
| >= | ifge |
| == | ifeq |
| != | ifne |

## Statements

`id = expr`
要生成的代碼是將堆棧頂部的值存儲在 id 中。

如果 id 是一個局部變量，那麼存儲結果的指令是 `istore 2`
若是全域變數，則 `putstatic [type] example.id`

`print expr`
Kotlin* 中的 PRINT 語句是通過使用以下格式調用 java.io 包中的 print 方法來建模的

```
getstatic java.io.PrintStream java.lang.System.out
invokevirtual void java.io.PrintStream.print(java.lang.String)
```

如果表達式的類型是字符串。

如果表達式的類型是整數或邏輯，則 int 或 boolean 類型將替換 java.lang.String。
同樣，字符串類型表達式的 PRINTLN 語句將被編譯為以下 java 彙編代碼：

```
getstatic java.io.PrintStream java.lang.System.out
invokevirtual void java.io.PrintStream.println(java.lang.String)
```

# if & loops

為 IF 和 WHILE 語句生成代碼相當簡單。考慮以下 if-then-else 語句：

```
if (false)
  i = 5
else
  i = 10
```

生成

```
  iconst_0
  ifeq Lfalse
  sipush 5
  istore 2
  goto Lexit
Lfalse:
  sipush 10
  istore 2
Lexit:
```

## while

對於每個 while 循環，在boolean表達式之前插入一個標籤，以及一個測試和一個條件跳轉將在boolean表達式之後執行。考慮以下 WHILE 循環

```
n = 1
while( n <= 10)
  n = n+1
```

生成

```
sipush 1 /* constant 1 */
istore 1 /* local variable number of n is 1 */
Lbegin:
  iload 1
  sipush 10
  isub
  ifle Ltrue
  iconst_0
  goto Lfalse
Ltrue:
  iconst_1
Lfalse:
  ifeq Lexit
  iload 1 /* local variable number of n is 1 */
  sipush 1
  iadd
  istore 1
  goto Lbegin
Lexit:
```

## for

```
for(n in 1 .. 10)
```

生成

```
sipush 1 /* constant 1 */
istore 1 /* local variable number of n is 1 */
Lbegin:
  iload 1
  sipush 10
  isub
  ifle Ltrue
  iconst_0
  goto Lfalse
Ltrue:
  iconst_1
Lfalse:
  ifeq Lexit
  getstatic java.io.PrintStream java.lang.System.out
  iload 1
  invokevirtual void java.io.PrintStream.println(int)
  iload 1 /* local variable number of n is 1 */
  sipush 1
  iadd
  istore 1
  goto Lbegin
Lexit:
```

# Procedure

如果將 n 個參數傳遞給靜態 Java 方法，
它們將被編號為 0 到 n − 1. 參數按傳遞順序接收

```
fun add (a:int, b:int): int {
  return a+b
}
```

生成

```
method public static int add(int, int)
max_stack 15
max_locals 15
{
  iload 0
  iload 1
  iadd
  ireturn
}
```

當一個Procudure被聲明為沒有返回值時，其對應方法的類型為void，返回指令為return。

### invoke

```
= add(a , 10)
```

生成

```
iload 1
sipush 10
invokestatic int example.int(int, int)
```

```
invokestatic <type> example.<fn_name>(...<arg type>)
```

# Note

## Local

如果一個方法有 n 個形參，那麼參數的編號從 0 到 n-1，而第一個局部變量的編號為 n。

# Example

```
class example {
  val a = 5
  var c : int

  fun add (a: int, b: int) : int {
    return a+b
  }

  fun main() {
    c = add(a, 10)
    if (c > 10)
      print -c
    else
      print c
    println ("Hello World")
  }
}
```

output:

```
class example
{
  field static int c

  method public static int add(int, int)
  max_stack 15
  max_locals 15
  {
    iload 0
    iload 1
    iadd
    ireturn
  }

  method public static void main(java.lang.String[])
  max_stack 15
  max_locals 15
  {
    sipush 5
    sipush 10
    invokestatic int example.add(int, int)
    putstatic int example.c
    getstatic int example.c
    sipush 10
    isub
    ifgt L0
    iconst_0
    goto L1
  L0:
    iconst_1
  L1:
    ifeq L2
    getstatic java.io.PrintStream java.lang.System.out
    getstatic int example.c
    ineg
    invokevirtual void java.io.PrintStream.print(int)
    goto L3
  L2:
    getstatic java.io.PrintStream java.lang.System.out
    getstatic int example.c
    invokevirtual void java.io.PrintStream.print(int)
  L3:
    getstatic java.io.PrintStream java.lang.System.out
    ldc "Hello World"
    invokevirtual void java.io.PrintStream.println(java.lang.String)
    return
  }
}
```