SLang V0.1：一个用java和c++写的半“编译”半“解释”的编译器

（本作品是本人自学了编译原理少部分知识、并参考了少部分CPython虚拟机实现（主要参考了变量命名方法）的基础上，结合自己的瞎搞（其实主要是瞎搞）写的一个半“编译”半“解释”的编译器，算第一个编译方面的作品（？），仅供玩和学习参考（可能参考价值也不大）），同时会有大量的BUG等待挖掘。当然因为架构、方法不是很成熟，算是编译原理中的hello world，这个项目只会维护一小段时间。

欢迎指出批评、建议或者帮助完善/重构代码，或者写sl语言的示例代码（哈哈）。入坑没多久，水平较差，请见谅 :)。

1、慢速起步

（1）前提：JRE环境、C++编译器（如clang、g++等）

（2）下载slang，里面包含编译器slang.jar、解释器源代码svm.cpp，一些系统运行时组件runtime目录。

（3）用gcc或clang等编译svm.cpp。

（4）任选位置建立一个slang项目目录，如test。

（5）将三个文件(slang.jar, svm可执行文件, runtime)复制进入项目目录。

（6）在项目目录中新建helloworld.sl，写入如下代码：

```
`runtime/io.sl`
writeln("hello, world!");
```

（7）用slang.jar（slang编译器）将sl源代码编译成可读的“字节码”：

```
java –jar slang.jar –c helloworld.sl –i helloworld.sli
```

（8）进一步压缩成“字节码”：（假设可执行文件为svm，windows下为svm.exe），并用密码123456加密“字节码”

```
svm –a helloworld.sli –o helloworld.slb –p “123456”
```

（9）运行

```
svm –r helloworld.slb –p “123456”
```

2、运行方式：编译 -> 汇编 -> svm（slang virtual machine）解释执行

（1）编译

因为主要是为了学习，且为了便于开发调试，编译后直接产生的代码是人类可以看懂的文本形式（非二进制形式），这里称为“中间代码”（和编译原理中的中间代码不是一个意思）。比如下列代码：

```
var int a = 2;
var int b = 3;
var int c;
c = a + b;
```

会被编译成（#号及后面内容是人为添加的）：

```
0 VMALLOC 3 # 申请三个全局变量的内存空间
2 LOAD_CONSTANT 0 # 0号常量“2(int)”进入操作数栈（0号常量定义在下面）
4 STORE_NAME_GLOBAL 0 # 取操作数栈栈顶，存在0号全局变量里
6 LOAD_CONSTANT 1
8 STORE_NAME_GLOBAL 1
10 LOAD_INT 0
12 STORE_NAME_GLOBAL 2 # 未手动初始化的int变量默认初始化为0
14 LOAD_NAME_GLOBAL 0 # 0号全局变量（a）入操作数栈
```

16 LOAD_NAME_GLOBAL 1 # 1号（b）入操作数栈
18 BINARY_OP 0 # 栈顶两元素弹栈，将和压栈
20 STORE_NAME_GLOBAL 2 # 栈顶元素弹栈，存到2号变量里
22 HALT # 程序结束
0 CMALLOC 2 # 申请两个常量的内存空间
0 CONSTANT 0 2 1 # 申请一个int（0）型常量2，被引用了1次
1 CONSTANT 0 3 1 # 申请一个int（0）型常量3，被引用了1次
编译部分是java写的。通过运行：
java –jar slang.jar –c 源文件 –i 输出的中间代码文件
如
java –jar slang.jar –c hello.sl –i hello.sli
编译成功会提示Done，否则会有相关的错误提示。
其他的编译指令：
1、**java –jar slang.jar –a 源文件**（输出该文件进行词法分析、语法分析后构建的抽象语法树AST）
2、**java –jar slang.jar –t** 语法分析是自顶向下的预测表法，该指令可以输出预测表。

（2）汇编（**不同于x86汇编！！！**）和解释执行
但是这样直接存下来太大。成熟的编译器，如java，是直接从源代码生成到字节码的。然而如果我生成字节码还需要考虑不同字节大小的数的存储等因素，为了简单起见，**SLang的“汇编”过程就是将中间代码中的换行符号去除、把英文的指令名称替换成数字编号、并能够顺便对源代码加密**。（很是简陋，呵呵）
汇编和解释都是由C++程序svm.cpp提供的。首先需要对svm.cpp进行编译，如：
g++ svm.cpp –o svm –O2 –std=c++11

然后对编译后的中间代码进行“汇编”：
./svm –a “中间代码”文件 –o 输出文件
./svm –a hello.sli –o hello.slb
也可以进行带有密码的“汇编”：
./svm –a hello.sli –o hello.slb –p “password”

最终产生的slb文件被称为“字节码文件”（其实不是真正的二进制形式存储的字节码，呵呵），直接通过svm运行：
./svm –r hello.slb
即可。对于带有密码的文件，运行需要提供正确密码：
./svm –r hello.slb –p “password”

如果希望运行时可以逐个指令的”单步调试“，并且实时查看相关信息（目前不支持变量跟踪什么的，但是会有很多提示文字），可以进入verbose mode，如**:**
./svm –r hello.slb –v

如果想将“字节码文件”逆向转化为可读的中间代码文件，则：
./svm –d hello.slb > hello.sli

如果带有密码，需要提供正确密码：

```
./svm –d hello.slb –p “password” > hello.sli
```

3、基本语法

目前支持的语法特性很少。

（1）类型、变量声明、作用域

支持int（64位整数），float（双精度浮点数）、数组

var <TypeIdentifier> <Identifier>[<ArrayDeclaration>] [= <Initializer>]

如：

```
var int m; # 没有显式初始化，自动初始化成0
var int a = 4;
var int b[100] = [1, 2, 3]; # 前三个分别设成1、2、3，其他都是默认值0
var int c[] = [1,2,3]; # 根据右边的数组初始化表达式来确定c的大小，为3
var int c[][] = [[1,2], [3,4], [5,6]];
var int c[3][4];
```

字符串（字符数组）：

```
var char s[] = “abcdefg”;
```

等价于：

```
var char s[] = [‘a’,’b’,’c’,’d’,’e’,’f’,’g’,’\0’];
```

支持块级作用域，如：

```
if (a == 1) {
        var int b = 3;
}
write(b); # 会报错，b不在块级作用域内部
```

目前不支持动态数组，也不支持指针（呵呵）。

（2）if语句 if (布尔类型表达式) { … } else { … }

单行语句可以不加大括号。

（3）for语句、break、continue 类似于c，但是必须先声明循环变量。如：

```
var int i;
for (i = 0; i < 100; i = i + 1) …
```

（4）while语句、break、continue

（5）函数定义

```
func <返回值类型> f(形参列表) {
        …
}
```

返回值

```
ret + 表达式
```

如

```
# 求数组a中所有元素的和
func int f(int a[], int n) {
        var int i, s;
        for (i = 0; i < n; i = i + 1) s = s + a[i];
        ret s;
```

```
}
```
数组作为函数形式参数时，必须给定除了第一个维度以外的其他所有维度的维数。如int a[][3]

支持函数重载（overload）和重写（override）的特性。可以有不同形参类型列表的同名函数。可以重写定义过的函数。函数只有被调用过才会被写入进字节码。

（6）赋值
支持数组赋值，比如a=[1,2,3,4]，相当于a[0]=1, a[1]=2, a[2]=3, a[3]=4
字符串，a=“aaabbbccc”

（7）直接生成到字节码：__svm__ 指令 [参数（数字或者&+标识符）]
如write(char ch)函数源代码：
```
func void write(char ch) {
    __svm__ LOAD_NAME &ch;
    __svm__ PUTCH;
}
```
&ch可以取得变量在系统内部的编号。这个方法首先加载局部变量ch到操作数栈，然后通过PUTCH指令在屏幕上输出了一个字符。

（8）`sizeof(a)` 获取a的大小（数组就是申请空间时元素个数，其他都是1）
（9）`printk a;` dump表达式a的值，包含值和类型。
（10）还有几个，忘了，而且可能还会增加新的语法特性（比如指针、结构体、对象）以后再补充。

4、引入文件
用：
```
`file_path`
```
来引入其他文件。一个文件不会被包含多次。

5、关于Runtime库和其他
runtime库本质上也是用slang写的几个函数库，随便写了几个，还缺很多。可以不断往上面加函数hhh，或者改进上面已有的代码。具体使用可以直接看源码参考下。