# 2024大作业

## 任务一、实现一个更加原始的虚拟计算机

之前的Command模式中，我们实现了包含复合指令（if，while等）的虚拟计算机。但是在实际的计算机中，我们是通过跳转指令来实现各种控制流语句的。比如一个选择语句

if (cond)

then-branch;

else

else-branch;

它的代码实际上类似于：

计算cond表达式，并将结果存放于内存m的代码;

如果m中的值为0，则跳转到else-branch分支对应代码的开始处的指令；

then-branch的代码;

跳转到if语句结束处的指令

else-branch的指令

上面红色部分的指令分别是条件跳转指令和无条件跳转指令。通过这些指令，我们就能够实现程序设计语言中常见的控制流语句。

本任务就是实现一个虚拟机。它的基本指令集合就是Command模式练习中的基本指令；但是不包含复合指令，而是替换成为了一组跳转指令。我们假设有一个带有10000个整数类型内存的虚拟计算机。它接受如下形式的指令：

op x y z

各条指令的具体描述见下表。其中红色的指令是新增加的指令。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OP | x | y | z | 注释 |
| 0 | i |  | addr | 常量存储，语义：MEM[addr] = i |
| 1 | x | B | idx | 数组存储运算符，语义：MEM[B+MEM[idx]] = MEM[x] |
| 2 | B | idx | x | 数组元素读取，语义：MEM[x] = MEM[B+MEM[idx]] |
| 3 | x |  | y | 拷贝指令，语义：MEM[y] = MEM[x] |
| 4,5,…,13 | x | y | z | 二目运算符指令，语义：MEM[z] = MEM[x] op MEM[y]  运算和op值的对应关系：4 +，5 -， 6 \*，7 /，8 %，  9 ==，10 >，11 <，12 &&，13 ||。  其中，等于、大于、小于判断时，如果关系成立，结果为1；否则为0；与、或、非运算的结果也是0（FALSE）或者1（TRUE） |
| 14 | x |  | z | MEM[z] = !MEM[x] |
| 20 | x |  | B | 如果MEM[x]的值非零，跳转到第B条指令继续执行。 |
| 30 |  |  | B | 无条件直接跳转到第B条指令继续执行。 |
| 40 |  |  | x | 无条件跳转到第MEM[x]条指令继续执行。 |
| 50 | x |  |  | 输出MEM[x]，并且后面加上一个空格 |
| 60 | x |  |  | 读取一个整数值，并存放到MEM[x] |

这个虚拟机的 “程序”是一个指令的序列，从0位置开始编号。运行时从第0号开始运行，一直运行到最后一条语句结束。例如，下面的程序可以计算输入的整数x的绝对值：

0： 60 0 //读入一个值，存放到MEM[0]

1: 0 0 1 //将常量0存放到MEM[1]

2: 10 0 1 2 //MEM[2] = (MEM[0] > MEM[1])

3: 20 2 5 //当MEM[2]非零时，跳转到第5条指令

4: 5 1 0 0 //MEM[0] = MEM[1] – MEM[0]，注意，此时MEM[1]==0

5: 50 0 //输出MEM[0]

If(MEM[2] == 0)

MEM[0] = MEM[1] – MEM[0];

else

；

output MEM[0];

## 任务二、表达式的处理和代码生成

使用递归函数来扫描表达式，得到表达式的抽象语法树的内部表示（这个可以考虑使用composite模式来表示表达式的内部结构），并且根据表达式的内部表示，为表达式生成相应的指令序列，该指令序列可以完成表达式的求值，并将值存放在一个约定的位置。

使用递归程序对输入的表达式进行分析，部分同学已经完成了。后面我会找时间讲一下。

表达式 -》 值（根据子表达式的值，求出表达式的值）

求值的代码（根据子表达式的代码，组合出表达式的代码）

Epxr1 op Expr2

Expr1的代码；结果存放到M1处

Expr2的代码；结果存放到M2处（保证Expr2的代码不会改变M1的值）

op M1 M2 M //整个表达式的值存放到M处。

生成代码的难点在于：

你需要为每个子表达式申请一个内存，来存放中间结果。

为了简化，我们假设MEM[0~99]的区间专门用于存放表达式的值和子表达式的中间结果。并且假设整个表达式的值总是存放于MEM[0]。

例如，对于表达式 a \* b - c / (d + 1)

* 首先为整个表达式申请一个内存地址0，也就是说，结果需要存放在内存地址0中；
* 然后，递归处理子表达式a \* b的时候，申请一个内存地址1，因为它的子表达式已经都是全局变量，可以直接生成相应的指令（假设a、b分别存放在地址101和102中）

6 101 102 1

* 再递归处理子表达式c / d，类似地可以得到指令：7 103 104 2
* 最后，为整个表达式生成指令 4 1 2 0

最后得到整个的指令序列：

6 101 102 1

7 103 104 2

4 1 2 0

执行这个指令序列的效果就是：

计算a\*b - c/d的值，并存放到内存0处。

如果一个表达式的结果存放在M位置，那么这个表达式的子表达式的中间结果的存放地址都大于M。因此，这个表达式的求值代码不会影响到M之前的位置。

所以基本的递归处理过程大致是这样的：

generateCode(Expression expr， int addressOfResult)

//expr是表达式的内部表示，而addressOfResult是存放表达式结果的地址

{ if expr是全局变量或者常量，生成一个将相应的值存放到addressOfResult的指令，

else if expr是局部变量，按照局部变量的寻址方法生成相应的指令序列（后面讲）

else

遍历expr的子表达式，对于第i个子表达式expr\_i，递归调用

generateCode(expr\_i, addressOfResult + 1 + i);

即为第i个子表达式生成一个讲结果存放于addressOfResult+1+i的指令序列

根据expr的运算符，生成相应的运算指令，例如对于二目运算符，生成的指令是： op addressOfResult + 1 addressOfResult + 2 addressOfResult

}

在生成代码时，可以加上你碰到的新变量的地址时从101开始按序存放的。比如，对于表达式a\*b + b \* c，你可以认为a,b,c的存放位置分别时101，102，103。（先假设都是全局变量，也就是说每个变量都有确定的地址）

在调试的时候，直接阅读生成的代码会比较困难。因此在调试的时候，在生成的每一条指令后面加一个注释。在正式版本的时候，把注释去掉就行。否则很难记住每个指令码对应的操作。

## 任务三、为程序生成指令代码：

我们这个“高级程序设计语言”比较简单，不过还是包括：

全局变量声明、函数定义，函数调用。并且函数定义内部还包括局部变量的定义（注意，全局变量和局部变量的访问是有点不一样的，后面会说）。具体的语法见后面的附录。

首先说一下建议的内存布局：

1. 内存中MEM[0..100]是分配给表达式计算的，用于存放表达式计算中的结果和中间结果；
2. MEM[100]用于存放函数调用的返回值，也就是说每个函数返回的时候，将结果存放到MEM[100]并返回。而调用者需要把MEM[0]的返回值拷贝到相应的位置。
3. MEM[101..199]用于存放全局变量。我们可以根据全局变量的声明，为它们顺序分配内存位置。
4. 局部变量会被存放到栈区，而MEM[200]用于存放栈顶的位置。MEM[201]开始存放函数调用时的局部变量，这些局部变量存放的位置就是我们通常说的栈区。
   1. 每次调用函数F时，需要根据F中声明的局部变量数量，增加MEM[200]的值；在退出之前需要相应减少MEM[200]的值；
   2. 局部变量的存放位置是相对于栈顶而言的。假设函数F有3个局部变量x,y,z，那么在执行F中的代码时，x,y,z的存放位置分别是MEM[200]-0, MEM[200]-1, MEM[200]-2。
   3. 栈里面有一个特殊的变量：返回地址。我们假设F有3个局部变量，那么会把将返回地址存放在位置MEM[200]-4。它是离开栈顶最远的位置，并且对于调用者来说，这个位置就是栈顶+1。
5. 根据上面的分配方法，全局变量的访问比较简单，但是对于局部变量的访问稍微复杂一点。如果局部变量x是函数F的第4个局部变量，那么将x的值拷贝到MEM[3]的代码可以是：

1 4 3 //将常量4加入到MEM[3]

5 200 3 2 //MEM[2] = MEM[200]–MEM[3]，

//MEM[2] = 栈顶减4，就是b的实际地址

2 0 2 3 //MEM[3] = MEM[0 + MEM[2]]; 即b的值被拷贝到MEM[3]

因此，整个代码生成过程会包含下面几个部分：

1. 为全局变量分配内存。这时我们要记录各个全局变量对应的位置（用于后面为表达式、赋值语句生成代码）；
2. 为各个函数定义生成代码
   1. 首先为局部变量分配内存；记住各个局部变量的内存位置（用于后面的代码生成），以及局部变量总数。因为局部变量只有在函数正被调用时才分配内存，所以对局部变量的访问和对全局变量的访问不同。
   2. 为函数体生成代码，包括两个部分：
      1. 函数开始时，增加栈顶MEM[200]的值的代码（是为函数的局部变量申请空间）。如果函数声明了n个局部变量，那么MEM[200]需要增加n+1（一个额外的空间用于存放返回地址）。
      2. 对于函数中的返回语句return x; 代码是这样的：

将x存放到MEM[0]的指令代码;

减少MEM[200]的值的代码（即释放局部变量的空间）

间接跳转到MEM[200]+1所指位置的间接跳转指令。

1. 为不同语句生成代码
   * 1. 对于表达式的代码生成，我们已经讲了。但是要注意全局变量和局部变量的访问方法不同。例如对于表达式a+b，如果a是全局变量，而b是局部变量。我们通过查表，发现a的地址是102，而b的地址是3，那么代码如下：

3 102 1 //将a的值拷贝到MEM[1]

1 3 4 //将常量3加入到MEM[4]

5 200 4 3 //MEM[3] = MEM[200]–MEM[4]，即b的实际地址

2 0 3 2 //MEM[2] = MEM[0 + MEM[3]]; 即b的值

4 1 2 0 //MEM[0] = MEM[1] + MEM[2]; 即a+b

* + 1. 对于赋值语句x = expr，我们只需要在expr的代码后面加入拷贝指令，将MEM[0]中的值拷贝到x的位置。（注意，如果x是局部变量，拷贝过程会稍微赋值一点）
    2. 函数调用语句x = funcName();的代码主要包括：

将当前指令序号加2的值存放到MEM[200]+1所指向的位置的指令;

//这个实际上是调用返回后，继续执行的指令的序号。MEM[200]是

//栈顶位置，调用时存放到当前栈顶加上1。对于被调用者来说，就

//就是它的栈的最下面的位置。

无条件跳转到funcName的第一条指令位置的指令；

将MEM[0]的值拷贝到x的指令；

//注意这一条指令的序号就是前面存放到MEM[200]+1位置的

//指令序号。也就是说，当被调用函数执行完毕后，就会跳转到

//这里。

* + 1. 对于控制流语句，我们需要为语句的各个部分生成代码，然后通过不同的跳转指令讲它们连接起来。这里要注意的是如何确定跳转指令的目标。例如，一个while语句

while(cond) statement;

对应的代码是：

cond的代码； //这个代码将cond的值存放到了MEM[0]

条件跳转指令：当cond非零时跳转到statement的开始处

无条件跳转到整个while语句的代码之后的跳转指令

statement对应的代码段 //

… //while语句代码之后的位置。

难点在于：各个跳转指令的目标是整个代码序列的目标，但是我们在生成各个语句代码的时候，是不知道相应的代码到底排在什么地方。建议使用下面的方法来解决：

1. 生成跳转指令的时候，记住它相对于当前生成的代码序列的序号。
2. 在代码拼接的时候，再寻找各个跳转指令，跳转它们的序号。

## 附录一、“高级语言的语法”

一、”高级程序设计语言“的语法：

1、常量、变量、和表达式的语法

a、常量：即整数常量

b、变量：即标识符，需要先声明再使用

c、表达式：

包含加减乘除运算、取模运算、关系运算（==，>，<）、逻辑运算（与&&、或||、非 !）

2、函数定义的语法：

'def' funcName()

{

变量声明 ‘;'

语句序列

}

3、变量声明的语法：

'VAR' 标识符序列

4、语句： 语句可以分为：空语句、赋值语句、函数调用语句，if语句、while语句、复合语句、输入语句、输出语句、返回语句

空语句： ';'

函数调用语句： funcName() ';'

赋值语句： varName = 表达式 ';'

if语句：

'if' (表达式)

语句

'else'

语句

while语句

'while' '(' 表达式 ')'

语句

复合语句：

'{' 语句序列 ‘}'

输入语句： 'input' var;

输出语句： 'output' var;

返回语句： 'return' var;

程序：

变量声明 ';' //这里声明的是全局变量

函数定义序列 //定义了多个函数。一定需要一个main函数