compiler--lab4

1. 实验目标

1.1. 流图构建与活跃变量分析

为了使目标代码执行的效率更高，减少访存操作，我们首先为中间代码构造CFG并执行一次活跃变量分析以检查哪些代码可以不用从寄存器写回内存

1.2. 目标代码生成

1.2.1. 代码选择

为三地址码选择合适的中间代码

1.2.2. 寄存器指派

采取局部寄存器分配算法，并尝试使用活跃变量分析信息进行全局优化。

2. 完成的任务

基础任务与静态分析框架构建，活跃变量分析算法的实现，CFG可视化。

3. 实验构思

3.1. IR的扩充

为了方便流图构建，我们将整个程序按照函数划分成不同的块。每个函数由`IRFunDec`定义。另外中间代码也需要一张符号表，它的描述符为`VarInfo`，其中保存了该变量的类型（函数参数/临时值/变量），栈内偏移量，所属函数，内存描述符等信息。

3.2. 栈结构

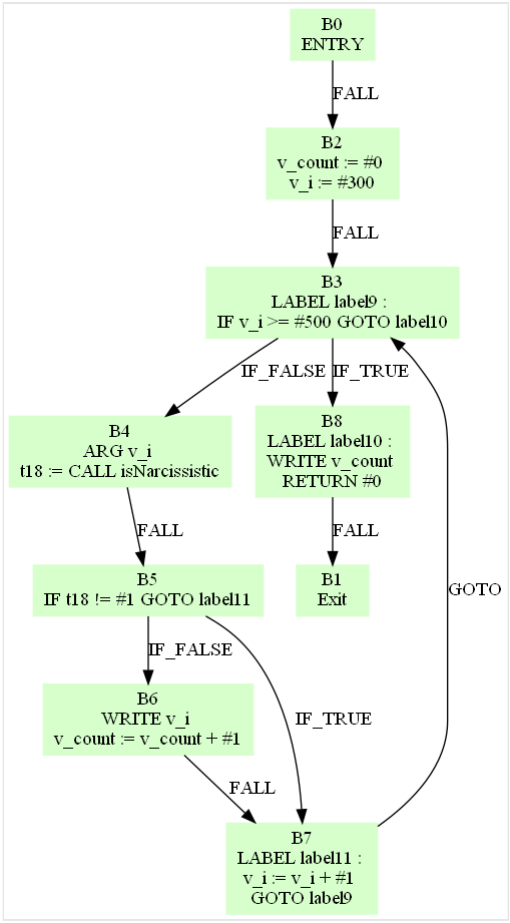
```
/**
 *      +-------------------+ <- fp
 *      |   return addr     |
 *      +-------------------+ -4
 *      |   caller fp       |
 *      +-------------------+ -8
 *      |       s0          |
 *      +-------------------+ -12
 *      |       s1          |
 *      +-------.....-------+ -16
 *      |       s7          |
 *      +-------------------+ -40
 *      |       t0          |
 *      +-------------------+ -44
 *      |       t1          |
 *      +-------.....-------+
 *      |       t9          |
 *      +-------------------+ -80
```

```
 *      |       a3          |
 *      +-------------------+ -84
 *      |       a2          |
 *      +-------------------+ -88
 *      |       a1          |
 *      +-------------------+ -92
 *      |       a0          |
 *      +-------------------+ -96
 *      |     vars tmps     |   // excludes parameters
 *      +-------------------+ <- sp
 */
```

我们为调用者和被调用者保存的寄存器在栈中固定的位置分配了存储位置，函数前四个参数紧随其后。接下来是临时值与变量。

3.3. CFG的构建

理论中的CFG构建算法和本次实验中构造CFG的策略稍有不同。本实验中将`CALL`和`RETURN`语句也作为基本快的最后一条语句，第一条`ARG`语句也作为基本块的第一条语句。另外也通过`graphviz`简单实现了一个CFG可视化模块（`./cfg/cfg_drawer`）方便调试寄存器的指派算法。测试用例中的一个main函数可视化后如下



3.4. 静态分析框架

这里的实现主要参考了Tai-e的框架结构（`./sa/`），拥有前后向`worklist`求解器，日后扩展只须完成数据流分析的几个重要函数即可。`optimizer`类将对每一张CFG的分析结果记录下来。上图中的活跃变量分析如下

```
Live variable:
==========Dataflow analysis result=========
Function: main
B0: { }
B1: { }
B2: { v_count v_i }
B3: { v_count v_i }
B4: { v_count v_i t18 }
B5: { v_count v_i }
B6: { v_count v_i }
B7: { v_count v_i }
B8: { }
=========================================
```

3.5. 寄存器分配算法

在目前的实现中我们采取了局部寄存器分配算法的思路并希望用到活跃变量这一全局信息进一步优化该算法。在一条语句翻译结束时，我们需要决定该条语句使用到的寄存器的去留，稍加改进`isNeeded`函数：如果某一变量在基本块的后续语句中出现，`isNeeded`返回`NEEDED`；如果变量在活跃变量分析结果中，返回`GLOBAL_NEEDED`；否则该变量既不会在基本块的后续语句中使用，也不会在全局中被使用。对于最后一种情况，我们不需要写回寄存器，另外两种情况需要在语句结束时写回内存。

然而，在实验中，上述思路只能勉强支撑我们通过大部分测试用例，仍然有两个测试用例无法通过。多次尝试无果后最终还是决定忍痛放弃全局活跃变量信息，即如果在基本块内部不再使用就将该寄存器内容写回，继续使用的寄存器保留。

尽管有些许遗憾，但是局部分配的效果仍然明显好于朴素分配算法。

4. 难点与总结

实验的难点在于，寄存器分配本质上就是在模拟运行时刻环境，而生成的代码是静态的。因此指令选择中会时不时陷入动态与静态纠缠不清的泥淖。比如我们使用基本块的信息对代码顺序遍历时，在退出基本块为了满足正确性必须将**此时**活跃的寄存器写回内存，这里的“此时”是指静态代码中运行到此的活跃寄存器。不同基本块的活跃寄存器不同，因此在退出基本块时必须将寄存器状态初始化，否则会导致不同基本块之间分配情况模糊不清。

另外，对于实验指导中关于“无空闲寄存器时分配最迟使用的变量”的描述我认为需要稍加修正，其实是指距离程序点最远的变量。即**不应当**从最后一条语句向前扫描，应当从当前语句向后寻找**第一次被使用的时间最晚的**变量。将该变量的寄存器作为目标寄存器。

总结：本次实验完成目标代码生成，即该实验的完成标志着我们的编译器能将C--源代码翻译成不出错的机器代码。下一次实验，将对中间代码优化以期生成效率更高的代码。