编译原理实验三报告

学号	姓名	邮箱	提交日期
181860109	吴润泽	181860109@smail.nju.edu.cn	2021/5/26

实现功能

生成三地址代码

操作数 Operand

实验中将操作数分为若干类，种类较多不一一列举：

```
OP_VARIABLE,        int var_no;      // 变量编号
OP_FUNCTION,        char* func_name; // 函数名字
OP_LABEL,           int label_no;    // 跳转编号
OP_TEMP,            int temp_no;     // 临时变量编号
```

由于源代码中可能出现以下划线开头的变量名，同时模拟器并不支持这种命名方式，故将所有的操作数（函数名除外）均采用编号的方式进行保存。

同时这种方式也为后续数据流分析提供了便利，便于在转移方程和控制流函数计算时变量名与向量索引的映射。

本次实验分配给我的是要求3.2，一方面无需考虑结构体变量的定义和使用，另一方面要考虑数组作为函数参数、高维数组类型变量的访问、数组深拷贝等问题。因此，在操作数中额外记录数组变量和（数组的引用）地址变量的元素类型和元素个数。

指令 InterCode

实验中根据指令使用到的操作数个数，分为若干类，种类较多不一一列举：

```
IR_ASSIGN,  // x := y
IR_ADDR,    // x := &y
IR_LOAD,    // x := *y
IR_STORE,   // *x := y
IR_CALL,    // x := CALL f
```

指令存储表示 InterCodeList

出于实现上的简单，以及指令列表任意位置增删操作复杂度的考虑，我采用了循环双向链表的方式来存储指令。

指令翻译

当词法分析、语法分析、语义分析全部完后，进行指令的翻译，与语义分析相同，根据已有的语法树自定向下进行翻译的工作。具体的翻译方案，则参考实验手册和龙书的SDT方案来实现，在此指出我关于数组互相赋值和函数实际参数关于数组的处理方式。

- 数组赋值：两数组元结构等价可相互赋值，将右值从低地址到高地址依次赋给左值，复制个数为两数组中元素数目较少的一方，返回值为右值地址。
- 函数实际参数：设有二维数组int a[2][2]，根据实际参数的类型分别处理：
  - 当实际参数为数组时，参数传递的是数组的地址；
  - 当实际参数为地址时，判断该地址中的数组元素类型，若为空(a[x][y])，即访问的数组中存放的元素；否则不为空(a[x])，即实际参数是低维数组。

中间代码优化（亮点）

关于中间代码的优化，我在翻译时的单条指令和翻译后的整体均进行了优化。

翻译时优化

- $\text{Exp} \to \text{ID} \mid \text{Exp} \to \text{INT}$，将变量对应的编号或者数值直接赋给返回值，而不再去生成诸如 place:=id.no, place:=int.val 指令语句；
- $\text{Exp} \to \text{Exp} = \text{Exp}$，将右值直接赋给返回值，而不再去生成 place:= left 指令语句，同时使用右值，利于之后的活跃变量分析；
- $\text{Exp} \to \text{Exp} + \text{Exp}$，进行算数运算时，若左值与右值均为立即数，则直接将计算后的结果赋给返回值；

经过测试，这种简单的单条指令的优化已经减少了很多冗余代码。

数据流分析

根据龙书的提示，我将翻译后的指令列表，分为若干函数块，每个函数块中划分基本块并建立控制流图，假设共 n 条指令，m 个基本块，t 个变量。

- 常量传播：与龙书所给框架不同，我没有想到较好的方式去计算每个基本块的 gen 和 kill，因此采取逐条语句分析其出口变量的信息，每一轮迭代的复杂度为$O(nt)$。

- 活跃变量分析：按照龙书所给框架进行迭代，，每一轮迭代的复杂度为$O(mt)$，迭代终止则得到了安全且正确的出口和入口活跃变量情况，并根据出口活跃变量情况，将无效的定值语句删除。

首先进行常量传播，之后进行活跃变量分析，经过测试，这种优化带来了巨大的提升。

编译&测试方式

进入 Code 文件夹所在路径，执行 make 命令，即可获得 parser 可执行文件。

执行 ./parser filenamae ir_path 命令，即可对一个待测试的源文件进行词法、语法、语义的分析，并将生成的中间代码输出到 ir_path 中。

实验感悟

本次实验是实现编译器的中间代码的翻译，使翻译出的中间代码可以在模拟器中正确地执行。

在我看来，生成中间代码本身的难度并不大，只需要利用好之前实验中的语法树和符号表所给的信息，确定合适的时机生成正确类型的指令即可。

而实现一个良好的中间代码优化则比较困难，一个原因是全局优化涉及数据流分析的理论，老师讲解的时间较晚，需要自己看龙书理解消化，很多地方存在理解误区，为实现带来更多困难，当然实现的工作量也远大于生成中间代码；灵一个原因是数据流分析的时间开销很大，要保证分析能在限定时间内完成。但是成功实现出一个优化安全，效果还不错，效率也没有难以让人忍受的优化方法是一件很有成就感的事情。