编译原理实验 3 实验报告

鄢振宇（基础班）　　171240518

2020 年 5 月 15 日

I　代码框架

```
/Lab
    /Code #存放代码的文件夹
        Makefile     #用于编译的文件
        common.h     #常用的宏/函数的声明/定义
        error.c      #用于报错的相关函数的定义
        error.h      #用于报错的相关函数的声明
        handlers.c   #包含若干接受node*的函数的定义
        handlers.h   #包含若干接受node*的函数的声明
        ir.c         #包含中间表示、label、operand相关函数的定义
        ir.h         #包含中间表示、label、operand相关函数的声明
        lexical.l    #词法处理工具flex的源代码
        list.h       #类似于<sys/queue.h>，提供可以用于定义链表的宏
        main.c       #包含main函数
        optimizaiton.h #定义各种优化的等级。高于parser等级的优化在运行时会被忽略
        option.c     #用于处理命令行参数
        syntax.y     #语法处理工具bison的源代码
        table.c      #符号表相关函数的定义
        table.h      #符号表相关函数的声明
        test.c       #单元测试相关函数的定义
        type.c       #类型相关函数的定义
        type.h       #类型相关函数的声明
    README #框架包含的一个自述文件
    /Test #存放测试文件的文件夹，.cmm后缀表示C—源代码
        ...
```

```
    lab3-*.cmm #实验三相关测试
parser      #预先编译好的二进制文件
report.pdf #报告
```

II 我的程序应该如何被编译

使用给定的 Makefile

III 我的程序实现了哪些功能

按照讲义要求实现了中间代码的翻译。并且自己做了一些优化。

IV 实现简述

在语法分析时，为每个节点配备一个函数指针，将函数指针指向对应的函数。这样，每个节点通过这个函数指针就能知道自己所对应的产生式要采取的动作。由此一来，每个函数内部的 if-else 都可以大大减少。

每个节点还维护一个 enum 类型，用于指示当前节点对应语句/算术表达式/逻辑表达式。以讲义的 translate_exp 为例，如果当前节点的类型是 ARITH，则直接调用其函数指针即可。如果当前节点的类型是 COND，则调用其函数指针，在 true 和 false 的地方给临时变量赋上 0 或 1 的值

每个函数内部，负责将自己对应部分翻译成 ir 对应指令，然后通过调用不同的函数（如 add_assign_ir，add_arith_ir）向 IR 的链表插入不同的 ir。

这样的实现对 ir.c 之外的文件屏蔽 ir 的具体结构，方便 ir.c 中对 IR 的结构进行调整。

V 自认为的亮点

值传参和引用传参

由于在 cmm 中，结构体和数组是使用引用传值的方式。如果我们声明 struct A a 的时候，采用 dec a [num]。而传参数的时候传的却是 &a，就会导致代码上的不美观（通过参数得到的结构体不需要先取地址，而函数体内部声明的需要先取地址）

所以，为了统一，对于所有非 BASIC 变量，我都将其当做” 指针” 进行处理。如果声明 struct A a, 我的中间代码会翻译成

```
DEC raw_a [size]
a := &raw_a
```

那么，传参的时候也只需要传 a 即可。

数组/结构体赋值

在 assign_handler 中，获得了 lhs 和 rhs 的信息之后，我会进行一个判断，如果 lhs 的 Type 为 BASIC，则它是基本类型，增加一条赋值 ir 指令即可。否则，说明其是数组或者结构体。可以手动写出与 while 循环等价的 ir 指令完成赋值。

实现了函数内联

函数内联并不复杂，只需要维护被内联的函数的所有变量/label 到内联进的地方的新变量名/label 标号，然后将所有 ARG 指令改为赋值指令，将 RETURN 指令改为一个赋值指令加上一个 GOTO 指令即可

单论函数内联，对指令运行条数影响并不大，主要还是在内联后通过常量传播、死代码/无用代码消除减少指令

其他优化一览

- 常算术表达式优化
- 常逻辑表达式优化 (并用于判定 if 的走向)
- 连续两条 LABEL（可以将其合并）
- 目的是另一条跳转的跳转语句（龙书 8.7.3）
- 反转 if 的条件以减少 goto 数量（龙书 6.6.5）
- 删除跳转到自己下一行的 GOTO
- 死代码消除（在 RETURN, GOTO 后的代码都是死代码），消除死代码后可能会触发上面的其它优化