编译原理第四次实验报告

161240005 陈勇虎

1. 完成的功能:
   (1) 实验一,二,三已经完成的功能
   (2) 在词法分析，语法分析和语义分析程序和中间代码生成程序的基础上，将 C-源代码翻译成 MIPS32 指令序列(可以包括伪指令),并在 SPIM Simulator 上运行。
   同样，为了方便测试，lab4 允许输出到终端和文件两种操作，其中输出到终端的内容(输出的终端的话)颜色统一设定为红字黑底,部分打印情况如下:(输出到文件的话，将不会有颜色设定)。具体颜色设定情况可见 Code/grammertree.h 文件

```
main:
  addi $sp, $sp, -4
  sw $ra, 0($sp)
  jal read
  lw $ra, 0($sp)
  addi $sp, $sp, 4
  move $t0, $v0
  move $t1, $t0
  li $t2, 1
  bgt $t1, $t2, label4
  j label5
```

   (3) 目前已完成了必做的样例，没有做功能的拓展
2. 实现方法:
   (1) 实验中已经使用 flex 和 bison 实现了词法分析和词法分析，语义分析，中间代码生成等功能
   (2) 因为中间代码的生成过程中，采用了简单的线性 IR 模式，所以指令选择的过程可以采取简单的将 IR 中间代码逐条对应到目标代码上，方式类似实验讲义中的内容.
3. 数据结构
   1. 实验中新增的数据结构表示如下，见 Code/MIPS32.h 文件

```
typedef struct var_t{
  int reg_no;
  Operand op;
  struct var_t *next;
}VarDef;

typedef struct reg_t{
  char name[10];
  struct var_t *var;
  int old;
}RegDef;

typedef struct stack_t{
  int length;
  int old[1024];
  int from;
  VarDef *varStack[1024];
}StkDef;
```

2. Vardef 结构体用于存储当前变量，使用的寄存器编号

RegDef 结构体用于存储寄存器的名字，存储的变量，old 用于后续的寄存器的选择.

StkDef 结构体用于后续函数调用中存储寄存器中内容和上下文恢复。

因为样例较为简单，所以栈的大小并不需要很大.:wq

4. 编译运行方法

为了方便测试,添加了两个伪目标用于执行，具体为；

```
# 定义的一些伪目标
.PHONY: clean test run
test:
        ulimit -c 1024
        ./parser ../Test/test1.cmm ../Test/out1.s

run:
        ulimit -c 1024
        ./parser ../Test/test2.cmm
clean:
```

1）make && make test 输出到*.s 文件

2）make && make run 输出到终端(会有颜色)

运行示例；

假定,待分析文本均位于 Test 文件夹下面，输出的文件也在 Test 文件夹下.

如果我们想要对文件名为 test2.cmm 文件进行分析，并生成对应的 MIPS32 指令。

1) 如果希望 MIPS32 指令输出结果输出到文件 out2.s，只需要：

将 test 伪目标内容中的../Test/test1.cmm 更改为../Test/test2.cmm

将 test 伪目标内容中的../Tes/out1.s 更改为为../Test/out2.s

随后 make && make test 即可

后续进行 spim –file out2.s 即可验证功能

2）如果希望 MIPS32 指令输出到终端，只需要:

将 run 伪目标内容中的../Test/test1.cmm 更改为../Test/test2.cmm

随后 make && make run 即可。(这样无法进行 spim 验证，只是方便查看生成的 SPIM 指令)

(ulimit –c 1024 是 debug 中用到的，与实验无关，无需考虑)

实验总结

借助 flex 和 bison 可以很快完成对一段代码的分析，并通过自定义的数据结构完成对代码分析树的建立。得到我们需要的语法分析树后，我们根据匹配表达式的翻译规则，生成我们需要的中间代码。在中间代码的基础上，我们可以生成相应的 MIPS32 指令，并通过 spim simulator 进行功能的验证。