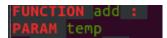
编译原理第三次实验报告

161240005 陈勇虎

1. 完成的功能:

- (1) 实验一,二已经完成的功能
- (2) 在词法分析,语法分析和语义分析程序的基础上,可以将 C-源代码翻译为中间代码,我是用的代码内部表示是线性结构,可在头文件找到自定义的 InterCode 数组。
- (3) 同样,为了方便测试, lab3 中输出的内容(输出的终端的话)颜色统一设定为 Green 或者红字黑底,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

(4) 目前已完成了所有的样例,选做部分只做了关于数组的第二个样例,结构体目前没有全部 完成,所以输出不全,目前只根据样例实现到了一维,二维数组的中间代码生成.

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
- (2) 思路:依次递归遍历整个语法树,遍历过程和语义分析相近,在遍历过程中可以生成每个代码块的中间代码,采用线性存储的方式,最后将中间代码依次输出
- (3) 基本表示式的翻译,基本仿照讲义中的内容,实验中,符号表沿用 lab2 中的符号表,作为全局变量,故实现函数中不需要作为函数的输入参数
- (4) 在遍历的过程中, 匹配到现在能相应的基本表示后, 根据遍历的次序, 依次生成中间代码, 并依次存入内存中
- (5) 目前存储为静态,分配空间大小会随着插入的单条中间代码增大。

3. 数据结构

1. 实验中单条间代码的数据结构表示如下,见 Code/grammertree.h 文件

```
typedef struct Operand_ {
   OP_KIND kind;
   union{
    int tvar_no;
    int label_no;
    char value[32];
   Operand name;
   }u;
   struct Operand_ *prevArgs;
   struct Operand_ *nextArgs;
}Operand_;
```

```
InterCode_ {
  CODE_KIND kind;
      Operand op;
    }single;
      Operand right, left;
    }assignOp;
      Operand result,op1,op2;
    }tripleOp;
      Operand op1;
      Operand op2;
      Operand label;
       har relop[3
     ifgotoOp;
      Operand op;
         size;
    }decOp;
}InterCode_;
```

- 为了简化程序,最后采用的数组存储,双向链表并没有使用操作数结构中将会记录操作数的类型,名称,标号等,具体见上图单条中间代码将会记录中间代码的类型,此外,目前分别了几大类;
 - 1. 有一个操作数
 - 2. 有两个操作数
 - 3. 有三元操作数
 - 4. 用于 ifgoto 语句的情况
 - 5. 用于 dec op size 的情况
- 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;

- 1) make && make test 输出到文件
- 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 test1.cmm 文件进行分析,并生成中间代码.

- 1) 如果希望中间代码的输出结果输出到文件 out1.ir 中,只需要:将 test 伪目标内容中的../Test/test4.cmm 更改为../Test/test1.cmm将 test 伪目标内容中的../Tes/out4.ir 更改为为../Test/out1.ir 随后 make && make test 即可
- 2) 如果希望中间代码输出结果输出到终端,只需要: 将 run 伪目标内容中的../Test/test4.cmm 更改为../Test/test1.cmm 随后 make && make run 即可。

(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。