编译原理实验报告

实验三：中间代码生成

高亦远 151160014

一、 实验目的

是在词法分析、语法分析和语义分析程序的基础上，将 C--源代码翻译为中间代码。将中间代码输出成线性结构，从而可以使用虚拟机小程序测试中间代码的运行结果。

二、 实验要求

在实验一、二的基础上编写一个程序，将符合一定规则的 C--源代码翻译为中间代码，中间代码的形式及操作规范如表 1 所示。

表 1. 中间代码的形式及操作规范

LABEL x :	定义标号 x
FUNCTION f :	定义函数 f
x := y	赋值操作
x := y + z	加法操作
x := y - z	减法操作
x := y * z	乘法操作
x := y / z	除法操作
x := &y	取 y 的地址赋给
x := *y	取以 y 值为地址的内存单元的内容赋给 x
*x := y	取 y 值赋给以 x 值为地址的内存单元
GOTO x	无条件跳转至标号 x
IF x [relop] y GOTO z	如果 x 与 y 满足[relop]关系则跳转至标号 z
RETURN x	退出当前函数并返回 x 值
DEC x [size]	内存空间申请，大小为 4 的倍数
ARG x	传实参 x
x := CALL f	调用函数，并将其返回值赋给 x。
PARAM x	函数参数声明
READ x	从控制台读取 x 的值
WRITE x	向控制台打印 x 的值

3.1 程序结构

实验三的程序的源文件主要由以下文件组成：

词法分析文件 lexical.l；

语法分析文件 syntax.y；

语法树相关代码 tree.c 及其头文件；

语义分析相关代码 semantic.c 及其头文件；
中间代码生成相关代码 intercode.c 及其头文件；
程序入口 main.c。

实验三在实验二的基础上，新增了中间代码生成文件 intercode.c 及其对应头文件 intercode.h。在 intercode.c 中定义和实现了中间代码的数据结构和相关操作。同时在 semantic.c 中增加了生成中间代码相关的函数。在 main 函数中也有相关变动。

使用 makefile 可以快速进行编译。经过编译后生成中间文件 lex.yy.c、syntax.tab.c、syntax.tab.h 及执行程序 complier。

测试时在根目录下使用 make 命令进行编译，使用./complier test1.cmm 来对测试文件进行测试；使用./complier test1.cmm out1.ir 将结果输出到文件中

3.2 程序功能

本程序实现了将 C—源代码翻译为中间代码的功能。当源码通过了语法和语义检查后，会输出对应的中间址码。例如对于测试代码

```
int main()
{
    int a = 3, b = 0;
    b = read();
    if (a > b)
        write(a);
    else
        write(b);
    return 0;
}
```

会输出以下中间代码

```
FUNCTION main :
a := #3
b := #0
READ t1
b := t1
IF a > b GOTO label1
GOTO label2
LABEL label1 :
WRITE a
GOTO label3
LABEL label2 :
WRITE b
LABEL label3 :
RETURN #0
```

3.3 功能实现

3.3.1 单条中间代码的数据结构

为了能够对代码进行调整和优化，不能一边对语法树进行处理一边把要输出，需要将所生成的中间代码先保存到内存中。为此需要设计合适的数据结构保存每条语句。本程序采用了一个动态数组保存 IR 表，这个数组保存了指向单一语句的指针。单一 IR 语句的结构定义如下：

```
typedef struct InterCode_t {
    //对应 19 种中间代码形式
    enum {
            LABEL_IR, FUNCTION_IR, ASSIGN_IR, PLUS_IR, MINUS_IR, STAR_IR, DIV_IR,
GET_ADDR_IR, GET_VALUE_IR,
            TO_MEMORY_IR, GOTO_IR, IF_GOTO_IR, RETURN_IR, DEC_IR, ARG_IR, CALL_IR,
PARAM_IR, READ_IR, WRITE_IR, DEBUG_IR, RIGHTAT_IR
    }kind;

    union{
        //1 操作数: LABEL_IR FUNCTION_IR GOTO_IR RETURN_IR ARG_IR PARAM_IR
READ_IR WRITE_IR DEBUG_IR
        struct{
            Operand op;
        }singleOP;

        //2 操作数: ASSIGN_IR GET_VALUE_IR TO_MEMORY_IR CALL_IR
        struct{
            Operand left;
            Operand right;
        }doubleOP;

        //3 操作数: PLUS_IR MINUS_IR STAR_IR DIV_IR GET_ADDR_IR
        struct{
            Operand result;
            Operand op1;
            Operand op2;
        }tripleOP;

        // IF_GOTO_IR
        struct{
            Operand op1;
            Operand op2;
            Operand label;
            char relop[32];
        }ifgotoOP;

        // DEC_IR
        struct{
            Operand op;
```

```
            int size;
        }decOP;
    }u;
}InterCode_t;
```

其中：枚举类型 kind 对应了可能出现的中间代码类型；联合 u 中定义了多种结构体，分别对应不同操作数数量的 IR 语句；

Operand 是结构 Operand_t 的指针，其表示 IR 中的某个操作数，定义如下

```
typedef struct Operand_t {
    enum {VARIABLE_OP, TEMP_VAR_OP, CONSTANT_OP, ADDRESS_OP, TEMP_ADDR_OP,
LABEL_OP, FUNCTION_OP, DEBUG_OP } kind;
    union {
        int tvar_no;        //TEMP_VAR_OP
        int label_no;       //LABEL
        char value[32];     //VARIABLE_OP/CONSTANT_OP/FUNCTION_OP/DEBUG_OP
        Operand name;           //ADDRESS_OP /TEMP_ADDR_OP
    }u;
    struct Operand_t *nextArgs;
    struct Operand_t *prevArgs;
}Operand_t;
```

3.3.2 中间代码生成

中间代码的生成与实验二生成语法树的过程类似。也需要遍历语法树的每个节点。因此本实验三大幅增改了 semantic.c，使之不仅能实现语义分析功能，同时能够生成中间代码语句。事实上，符号表的生成和 IR 表的生成是同步进行的。例如，在遇到“FunDec”节点时，不仅在符号表中插入相关项，同时也会执行以下语句

```
Operand funcOp = (Operand)malloc(sizeof(struct Operand_t));
memset(funcOp, 0, sizeof(Operand_t));
funcOp->kind = FUNCTION_OP;             //设置操作数类型为 FUNCTION_OP
strcpy(funcOp->u.value, field->name);
InterCode funcIR = (InterCode)malloc(sizeof(InterCode_t));
memset(funcIR, 0, sizeof(InterCode_t));
funcIR->kind = FUNCTION_IR;             //设置语句类型为 FUNCTION_IR
funcIR->u.singleOP.op = funcOp;
insertCode(funcIR);
```

生成对应的 IR 项 funcIR，为其分配存储空间并通过 insertCode(funcIR)函数插入到 IR 表中。

3.3.3 函数的调用

函数的调用由语法单元 Exp 推导而来，例如当出现 ID LP RP 语句时，就是出现了函数调用，需要进行相关的处理，生成 IR 语句。由于实验要求中规定了两个需要特殊对待的函数 read 和 write，故当我们从符号表中找到 ID 对应的函数名时不能直接生成函数调用代码，而是应该先判断函数名是否为 read 或 write。

3.3.4 数组

由于实验要求中假设并不存在结构体和高维数组，也规定了不会出现浮点数，因而数组的寻址则变得比较简单。只要用首地址加上 4×下标即可。