Compiler Lab Report

刘嘉欣

2023 年 11 月 28 日

1 功能与实现

1.1 中间代码生成

中间代码的表示采用双向链表（线性 IR），生成最终 ir 文件时将链表翻译即可. 翻译模式按照讲义的思路既可实现大部分语句的翻译，这部分自己的思路不多，可以略去不讲. 我是先不考虑数组有关的任何内容，逐步实现了其他功能，确保基本无问题后，（此时 OJ 的成绩仅有 50 分）最后再补上数组的翻译的.

值得一提的是 CodePlus() 函数的实现，也就是将多个链表串联起来的函数，采用了可变参数 va_list，第一个形式参数为链表数目，后面紧跟着若干链表表示的 IR.

对于 place 为空的表达式，最后生成 IR 时生成的赋值语句可以不用翻译 (如"1+2")，但是要留意函数调用可能没有存放返回值的变量，这时可以创建一个临时变量来存放，以便生成对应的 IR.

由于要识别 write 和 read 函数，语义识别不能出错，故需要在进行语义分析之前事先往符号表中插入这两个函数.

由于没有全局变量,实验 2 中创建新的域相关的内容可以略去,让 newScope() 和 deleteScope() 函数直接返回即可.

类似实验 2 的符号表，本次实验实现了一个操作数表，应对表达式处理中 ID 的对应，查找函数是 look_up().

1.2 数组部分的实现

选做内容还要求实现高维数组.

1.2.1 变量定义 Def 和函数定义 FunDec

实现了一个数组处理的函数 Oprand deal_array(char *name, int *size)，它从符号表中查找数组的名字，并向符号表中插入符号（若不在符号表中），并返回分配的变量符号和分配的大小（单位为字节）.

在后续 Debug 过程中发现对于函数参数的定义，若传入数组，压栈的其实是地址，后续访问数组时不能再对地址取地址. 解决的办法就是在 FunDec 中设置数组符号的 para=1，在表达式翻译 ID 的时候，若 ID 是数组且为函数参数，不需要取地址而是单纯的赋值 (地址传递).

1.2.2 表达式中出现

需要进行改变的地方如下：

1. ID: 若 ID 为数组变量，则需要取地址（注意前面，是函数参数的时候不取地址），其他情况直接传递即可.

2. Exp LB Exp RB: 这里需要实现多维数组地址的计算（基址和偏移）. 多维数组基址的计算递归调用 translate_Exp()，如果最终得到的数组的值不是 INT 型，则赋值传递地址；否则需要解引用（需要修改，见下）.

3. Exp ASSIGNOP Exp: 需要实现左边表达式是数组的情况. 这部分最为复杂，在处理完数组的翻译之后才补全这部分代码. 这里需要对上面 Exp LB Exp RB 的情况进行修改：即使最终得到的数组是 INT 型，但传递回来的应当是地址，之后再进行 Store 操作（如 a[1] = 1，翻译的最后应当有 *v = t 类似的式子）. 具体实现只需要在处理 Exp1 时放置于 place 的操作符的新增的 addr 属性设为 1, 再在 Exp LB Exp RB 中进行识别即可. 实现到这一步，OJ 已经有 94 分了，但是 A3 没有通过. 最后发现有以下情况需特殊处理：”int b[2], a[5]; b = a; write(b[1]);” 这里虽然将 a 地址赋值给 b 对应的变量没有问题（有取地址符），但后续对 b 解引用时却对这个地址再次取地址而出错. 注意到这里有点类似函数传数组地址的情况，直接在 ASSIGNOP 左边表达式为 ID 且是数组类型时设置 para=1 即可.

至此完成了所有内容.

2 编译方式

采用 Makefile 进行编译，直接 make 即可. 执行的命令如下：(其中，syntaxTree.c 和 syntaxTree.h 保留了实验 1 语法树的定义以及相关操作;semantic.c 和 semantic.h 为语义分析的内容, 其中符号定义和符号表的定义在 symbol.c 和 symbol.h 中，中间代码生成相关的 intercode.c 和 intercode.h 中)

1. flex -o lex.yy.c lexical.l 使用 Flex 编译 lexical.l 文件，生成 lex.yy.c

2. bison -o syntax.tab.c -d -v syntax.y 使用 Flex 编译 syntax.y，生成 syntax.tab.c 和 syntax.tab.h 两个文件，其中 lexical.l 引用了 syntax.tab.h

3. gcc -c syntax.tab.c -o syntax.tab.o 编译 syntax.tab.c，生成 syntax.tab.o

4. gcc -std=c99 -c -o syntaxTree.o syntaxTree.c 编译 syntaxTree.c，gcc -std=c99 -c -o main.o main.c , gcc -std=c99 -c -o symbol.o symbol.c 和 gcc -std=c99 -c -o semantic.o semantic.c 将所需的 syntaxTree.c、main.c，symbol.c 和 semantic.c 文件编译生成 syntaxTree.o、main.o、symbol.o 和 semantic.o. **本次实验:** gcc -std=c99 -c -o intercode.o intercode.c

5. gcc -o ./parser intercode.o syntax.tab.o syntaxTree.o main.o symbol.o semantic.o -lfl -ly 把 main.o 和 syntax.tab.o 和 syntaxTree.o 放到一起进行编译成最终的可执行文件 parser