编译原理实验三：C--语言的中间代码生成

软件工程 范兆基 20331011 819402765@qq.com

始、报告简述

- 本次报告的内容依然是：(1) 编写实验代码时遇到的一些问题以及 (2) 个人认为该代码中比较重要的内容。毕竟面面俱到的话内容过多，而且没有意思，我懂的内容难道老师和助教会不懂吗？

一、底层数据结构

主要是基于实验指导给出的数据结构进行修改

1.1 操作数

```
typedef struct Operand_* Operand;
// 操作数
struct Operand_
{
    enum { VARIABLE, CONSTANT, TEMP} kind;// 三种类型：变量、常量、临时变量
    // 或许这个union直接用一个int代替即可，但是可改可不改，小偷一下懒
    union
    {
        int no;// VARIBLE/ADDRESS/TEMP的序号，分开统计
        int value;// CONSTANT的值
    } u;
    // 以下三个变量用于控制取值、解引用IR的生成
    int if_addr;// 用于判断是否为地址，若为地址，则不会对VARIABLE取址(&v)
    int if_deref;// 是否在操作数前加取值符&，表示取地址
    int if_take_addr;// 是否在操作数前加解引用符*。表示解引用
};
```

1.2 中间代码

```
// 中间代码
struct InterCode
{
    // IR类型
    enum { ASSIGN_IR, ADD_IR, SUB_IR, MUL_IR, DIV_IR,
            LABEL_IR, RETURN_IR, GOTO_IR, IF_IR,
            READ_IR, WRITE_IR,FUNC_IR, ARG_IR, PARAM_IR, CALL_IR,
            DEC_IR } ir_kind;
    // 对应的操作数
    union
    {
        struct { Operand right, left; } assign;
        struct { Operand result, op1, op2; } binary_op;
        struct { Operand result, op1; } unary_op;
        struct { int label_no; } label;
        struct { Operand return_operand; } return_op;
        struct { Operand left, right; int label_no; char* relop; } if_op;
        struct { Operand place; } read_write_arg_param_return;// READ_IR、WRITE_IR、PARAM_IR、RETURN_IR共用
        struct { Operand left; char* func; } func_call;// FUNC_IR、CALL_IR共用
        struct { Operand place; int size; } dec;
    } u;
};

// 再形成链表链表
typedef struct InterCodes_* InterCodes;
struct InterCodes_
{
    struct InterCode code;
    InterCodes prev, nxt;
};
```

1.3 实参链表与变量链表

```
// 用于记录函数调用的实参
typedef struct Arg_List_* Arg_List;
struct Arg_List_
{
    Operand arg;
    Arg_List nxt;
};

// 用于记录一个函数中有哪些变量(包括形参)
typedef struct Operand_List_* Operand_List;
struct Operand_List_
{
    Operand ope;
    struct Operand_List_* nxt;
};
```

二、结构体与数组

- 我们需要解决两个问题：
  (1)对应产生式($Exp \rightarrow Exp\ DOT\ ID \mid Exp\ LP\ Exp\ RP$)的Exp是否需要进行解引用
  (2)产生式($Exp \rightarrow ID$)是否需要取地址。
- 单纯讲比较抽象，我**举例子简单说明**：$a.b = c.d$ 。
  - 问题(1)：对于$a.b$，我们要先获取a对应的地址(在此之前会先为a开辟空间)，此时就需要对a进行取址&a，然后再根据b的偏移量得到真正对应的地址，即 $\&a + bia\_b$ 。$c.d$ 同理。
  - 问题(2)：对于左右两边的赋值，我们需要计算出左右两边代表的地址addr_1和addr_2，先对addr_2解引用得到所存储的值value，然后再对addr_1进行解引用将value存进去
- 对于问题(1)，还有一种情况需要考虑，就是多维数组以及结构体连续取域($a.b.c.d$、$a[1][2[3]]$)，内层的Exp不需要进行解引用，只需参与计算地址，只有最外层得到最终地址后才可能需要进行解引用
- 代码解决思路：
  - 在translate_Exp()函数中多加一个参数remain_addr，表示所翻译的Exp如果是代表一个地址，是否需要保留为地址形式，而不进行解引用
  - 对于$a.b = c.d$，在一开始处理$a.b$时令remain_addr=1，保留为地址形式addr_1，处理$c.d$时则无需保留，直接获取空间中所存放的值value，然后再将value存放到addr_1所代表的空间中($*addr\_1 = value$)
  - 对于多维数组以及结构体连续取域或两者综合($a.b[5].c$)，一般而言是内层不解引用，到最外层才解引用.代码中记录数组取值(array_layer)和结构体取域(filed_layer)的层数，遍历语法树时每进入一个对应结点，层数+1，退出时-1.只有当层数为1时(array_layer+field_layer==1)，才进行解引用。
    - 但需要注意一个**特殊情况**：存在一个高维数组$inta[3][3][3]$，同时存在一个函数为$int\ func(intb[3])$，在处理函数调用$func(a[0][0])$时，$a[0][0]$就要作为一个地址传入(C--中结构体和数组传递地址)，此时虽然已经是最外层，但是也要保留为地址，不可解引用取出值。**此时便还需要根据remain_addr这个参数判断是否进行解引用。**

三、IR简化

- 在初始的版本中，程序输出的IR执行会存在多余的临时变量和ASSIGN_IR。IR简化就是通过减少临时变量数目，从而减少ASSIGN_IR数目，最终减少需要执行的IR数目。
- 对于本代码的IR简化，我需要先讲一下其中的翻译Exp的函数 `translate_Exp(void translate_Exp(struct Node* nd, Operand place,int remain_addr)` 。remain_addr参数前面已经讲过，nd参数表示要翻译的子树，翻译该子树所得的值存放place这个操作数中。
- 实验指导的 `tranlate_Exp` 及其他翻译方案是先创建一个临时变量操作数place_tem，存放对子树翻译后所得的值，然后再将place_tem赋值给place。这样会产生多余的临时变量与IR。
- 而在我的代码实现中，我是继续将place传递下去，直至必须要赋值的时候，才对place进行赋值，省去了大量临时变量。
- 例子：$a = b.c$
  - 实验指导：

    ```
    t1 := &v2
    t2 := t1 + 8
    t3 := *t2
    v1 = t3
    ```

  - 我的实现：

    ```
    t1 := &v2 + 8
    v1 = *t1
    ```

四、测试案例test_o4.cmm

- 我个人认为这个测试案例并没有问题，我的代码也能生成正确的对应IR。
- 我私下和提出这个问题的同学进行了交流，他认为C--的文法不支持结构体连续取域。进行讨论之后，我认为C--的文法是支持结构体连续取域的。
- 例子：$a.b.c$可以对应这样的推导过程：$Exp \rightarrow\ Exp\ DOT\ ID \xrightarrow{l} Exp\ DOT\ ID\ DOT\ ID$

五、代码编译与运行

1. 代码编译：进入Code目录下执行make命令，会在该目录下产生相应可执行文件parser
2. 代码测试：编译完成得到parser文件后，进入Code目录下执行make test命令，会自动测试Test目录下的所有文件，并生成一个IR目录，里面存放与TEST目录中文件对应的.ir文件

六、实验总结

又是纯参考实验指导写编译原理实验的一星期。但是这一次的过程没有上一次那么艰难。这一次实验指导已经给出了大部分的翻译方案，我只需要再做一部分补充、改进工作就好。此外，实验二的代码中保留了所有结点的属性，虽然这样的写法很捞，但给这次实验带来了便利，也算是误打误撞了。原以为这次的代码量会远少于实验二，但不知不觉也写了1300多行。另外，在写实验三的过程中去解决实验二的bug真的好搞笑，哈哈哈，只能说提交给老师的实验二代码还不够完善。当然实验三的代码也未能尽善尽美，生成的IR中的临时变量的序号就不连续，希望不会影响后续的实验，这个等到后面有想法或者后来人解决了。好了，就这样了吧，可以去过我愉快的周末了。