# 编译原理实验报告

## 实验三：中间代码生成

**191220008 陈南疃**

924690736@qq.com

## 一、运行方式

```
$ make
$ ../parser inputfile outputfile
$ make clean
```

## 二、实现功能

任务编号：11

选做部分：3.2

### （一）中间代码生成

#### 1、基本数据结构

本实验中，中间代码的生成是基于各个语法单元进行翻译的，在此前的实验中，我们已经通过词法分析和语法分析构建起了一棵语法单元的树，因此我们只需要重新遍历这棵树，对于不同的语法单元编写对应的函数进行处理，就能够在遍历一遍树后得到最终的中间代码。

#### 2、将嵌套作用域修改为全局作用域

由于在实验二部分，我的选做功能是实现嵌套作用域，按照实验手册的方法实现后会出现语义分析结束后符号表会被清空（或只剩下最外层作用域的符号）；此外，为了实现嵌套作用域，我设置了一些额外的变量，更改了一些查找、增添符号的函数的调用方式和实现方法，因而导致在本实验中存在不兼容的情况。因此，我在本实验开始初期，先将实验二重新时限为全局作用域的版本，然后再继续进行本实验。不过由于在上次实验中代码的模块化和接口完成的比较好，与其他模块的耦合性基本解耦，所以在实验三初期写的代码基本不需要因此而更改。

#### 3、对已定义变量的操作数的保存

在中间代码生成的过程中，对于已定义的变量或者函数形参（如 v3），当下一次调用的时候需要重新得到该操作数，因此，我设计了一个操作数链表，在变量定义或函数形参定义的过程中将每个变量（数组）的操作数加入到该链表中，并存储下该变量的名字。当下一次调用变量时，通过变量名进行查找即可。

```
typedef struct Operands_* Operands;
struct Operands_ {
    Operand operand;
    Operands next;
};
...
void addVariableOperands(Operand operand);
Operand searchVariableOperands(char* name);
```

但需要注意的是，一个变量可能会定义两次（一次是正常定义，一次是函数参数定义），所以我将插入操作数的时候用前插法，查找的时候从链表头部开始查找，这样就能查找到最近的定义，即所需的变量操作数。

**4、函数实参倒序传入**

函数实参可利用上述的数据结构进行存储，再将 translate_Args 函数的参数和返回值改为：

```
Operands translate_Args(Node* node, Operands argOperandsHead);
```

同理，在插入时用前插法，生成中间代码时从链表头部开始生成，即可得到倒序的函数实参中间代码。

**5、place为空的情况**

按照实验手册的方法，translate_Exp的参数里有一个名为place的参数：

```
void translate_Exp(Node* node, Operand place);
```

然而，该参数并非总是存在，当 Stmt -> Exp SEMI 的时候，传入的 place 即为 NULL。因此在 translate_Exp 中需要对 place 是否为 NULL 进行判断。需要注意的是，Stmt -> Exp SEMI 时 Exp 的翻译方式本该只有几种，但在某些超强测试样例中，出现了不常规的翻译方式，因此需要对所有的 place 进行判断。

**6、&和*的打印方式**

在实验手册的样例中间代码中，可以发现一些变量和临时变量前面出现了 & 或 * ，为了能够适时输出这些符号，我仔细梳理了所有可能出现 & 和 * 的情况，最后决定将取地址或取值的判断放在了中间代码的输出过程中，通过判断操作数的类型来决定是否输出 & 和 *，如：

```
case IR_ASSIGN: {
    if(interCodes->code->u.binop.op1->kind == OP_ADDRESS && interCodes->code->u.binop.op2->kind != OP_ARRAY) {
        fprintf(file, "*");
    }
    printOperand(file, interCodes->code->u.binop.op1);
    fprintf(file, " := ");
    if(interCodes->code->u.binop.op2->kind == OP_ADDRESS) {
        fprintf(file, "*");
    } else if(interCodes->code->u.binop.op2->kind == OP_ARRAY) {
        fprintf(file, "&");
    }
    printOperand(file, interCodes->code->u.binop.op2);
    break;
}
```

**7、Union 类型的信息存储方式**

在操作数的结构体中，用了一个联合类型来保存对应操作数类型的相关信息，由于变量类型的操作数需要保存数目和名字两个信息，故不能都放在 union 中（否则信息会被覆盖），因此我将每种类型的信息都改为结构体类型，在结构体中保存相关信息，这样就能保证信息保存和取用的一致性和便捷性。

```
struct Operand_ {
    ...
    union {
        struct { int varNum; char* varName; } var;
        struct { int tempNum; } temp;
        struct { int labelNum; } label;
        struct { int constValue; } constant;
        struct { char* funcName; } func;
    } u;
};
```

#### 8、多维数组的实现

多维数组的实现原理其实并不复杂，只需要将取地址、取值和存值的方式的实现好，就能够很容易的实现多维数组。

举个例子，现在有一个多维数组 `a[3][4]`，如果在其他地方出现了 `a[2]`，我们可以将其看成一个子数组，得到该子数组的元素类型为长度为 4 的一维数组，并得到元素的大小为 16，那么我们就能计算出 `a[2]` 的地址，进而能够进行赋值操作。所以问题的关键就是如何得到子数组的元素类型和大小。在本实验中我利用了实验二中的对语法单元 Exp 进行语义分析的函数的返回值得到元素类型，再将该类型作为实参传入递归函数 getSize 得到元素大小，从而实现多维数组元素的地址计算：

```
Type Exp(Node* node);
int getSize(Type type);
```

### （二）中间代码优化

#### 1、减少临时变量的赋值

在中间代码的生成过程中可以发现，有许多不必要的临时变量，将其删去可以让多次赋值缩减为一次，主要是在语法单元 Exp 的处理过程中，如：

- Exp 直接生成 ID 时，可以不额外生成临时变量，而是直接将 place 修改为变量类型的操作数。
- Exp 直接生成 INT 时，可以不额外生成临时变量，而是直接将 place 修改为常量类型的操作数。
- Exp 生成赋值操作时，判断左右操作数的类型，若为 ID 或 INT，则不额外生成临时变量。
- Exp 生成加减乘除的表达式时，若左右操作数都为常量，则不额外生成临时变量，而是直接计算结果，将 place 修改为常量类型的操作数。
- ……

## 三、心得和建议

实验三总体来说思路比较清晰，再加上实验手册上有部分语法单元的中间代码翻译方法提示，因此实验过程中步骤比较明确，但实验的难点在于中间代码和操作数的数据结构的设计，即有哪些类型的操作数以及如何表示它们。如果结构体设计不当，则可能会出现信息冗余、信息冲突和信息缺失的情况，容易打乱编码思路，增加实验编写的复杂性。因此，我采用了和实验二中相同的方法，在草稿纸上画出数据结构图和数据流向图，很基本确定该数据结构的可行性，再继续写代码，这样即使实验过程中发现数据结构的问题，也不会有太大的改动。

本实验的另一个大模块——中间代码优化，由于最近事情太多，没有来得及专门去写，只写了一点细枝末节的优化，有些遗憾，不过听说以后会单独作为一个实验，也觉得比较合理。此外，我想感谢我的几位舍友，我中途遇到了一个很难以理解的 bug，在他们的帮助下，一直找 bug 找到了凌晨一点半，才解决了问题。