实验三　中间代码生成

一、程序功能及其实现

1. 功能概述

本程序基于实验一的词法分析、语法分析程序和实验二的语义分析程序，实现了 C-- 语言的中间代码生成。程序的输入是一个 C-- 源程序，输出是对应的中间代码。程序实现了全部 19 种中间代码生成规则，没有涉及优化和选做部分。

2. 实现细节

(1) **中间代码的存储方式：**本程序使用一个大指针数组来存放中间代码。每条中间代码是一个字符串，在遍历语法树的过程中逐条生成，并插入到数组中。当语法树遍历完成后，程序会将中间代码写入到输出文件中。

(2) **中间代码的生成方式：**程序从语法树的根节点开始，按照先序遍历的顺序访问每个结点。不同的结点有不同的处理函数 `translateXXX`，其中 `XXX` 是结点的名称。在 `translateXXX` 内部，通过递归地调用函数来处理嵌套的语法结构。在生成中间代码时，通常会调用一些 `genYYY` 函数，以简化中间代码的生成，减少冗余代码。

(3) **数组处理：**与数组相关的操作有三种：声明一个数组、访问数组元素和给数组元素赋值。声明数组时，数组的大小可以由语义分析的结果确定。访问和赋值时，为了获取变量所在地址，需要按照如下公式计算：

```
addr = base + offset*4
```

其中，`base` 通过对数组名取地址 (`&`) 得到，`offset` 通过计算下标得到。计算出 `addr` 后，便可以通过 `*addr` 来进行访问和赋值操作。

二、程序编译方式

1. 实验环境

- 操作系统：Ubuntu 20.04.6 LTS
- 编译器：gcc version 9.4.0
- Flex：2.6.4
- Bison：3.5.1

2. 文件结构

```
lab3-code/（根目录）
├── build.sh（编译脚本）
├── lexical.l（词法分析器源代码）
├── main.c（主程序源代码）
├── nodeType.h（语法树结点类型定义）
├── semantic.c（语义分析器源代码）
├── semantic.h（语义分析器头文件）
├── syntax.y（语法分析器源代码）
├── translate.c（中间代码生成源代码）
├── translate.h（中间代码生成头文件）
├── tree.c（语法树操作函数）
├── tree.h（语法树头文件）
├── utils.c（工具函数）
└── utils.h（工具函数头文件）
```

3. 编译方式

在根目录下执行 `./build.sh` 即可编译程序。编译完成后，会生成一个名为 `main` 的可执行文件。可以通过 `./main path/to/test.cmm path/to/out.ir` 运行程序，其中 `test.cmm` 是待分析的 C-- 源代码文件，`out.ir` 是输出的中间代码文件。

三、程序创新与亮点

1. 函数相关

为了实现函数调用的中间代码生成，程序首先扫描参数列表，并将其变量名存入一个临时表中，这里的变量名可能是 vi、ti 或常数。然后，反向遍历参数列表，生成 param 语句，最后再生成 call 语句。

为实现在函数内部对参数的使用，程序在处理 funDec 时，会为每个参数分配一个 vi 变量名，并将其存入符号表中，与源代码中的形参名对应。我在语义分析实验中仅存储了函数参数的类型，而没有存储形参名，（形参名作为独立的变量存储在符号表中，无法通过函数获取），因此我对 semantic.c、semantic.h 中的相关代码和数据结构进行了一些修改，参照结构体的记录方法，使用 FieldList 来存储形参列表。

2. 工具函数的使用

为了简化中间代码的生成，程序实现了一系列工具函数，如 newTemp、genAssign 等。通过使用这些函数，程序的代码量大大减少，可读性和可维护性得到了提高。仿照实验指导书的做法，本程序使用 v + 数字的方式替换原来的变量名，并使用 t + 数字的方式生成临时变量名。为了存储 vi 与源代码中变量名的对应关系，程序对符号表的结构进行了修改，在 FieldList 中添加了一个域 varName，存放中间代码生成过程中为其分配的变量名。变量和临时变量名生成的相关代码实现在函数 newVar 和 newTemp 中。为了避免重复，函数内部将 varCnt 和 tempCnt 设置为 static，并在每次调用后加 1。

3. 模块化设计

本程序的语义分析过程与中间代码生成分离，即先完成语义分析，再进行中间代码生成，提高了程序的模块化程度。在中间代码生成过程中，程序的每个函数都有明确的功能，例如 translateExp 用于处理表达式，translateStmt 用于处理语句，translateCond 用于处理条件表达式等。这种模块化设计使得代码易于理解和维护。

4. 内存管理

当生成新的临时变量或标签时，程序会动态分配内存；在不再需要这些临时变量或标签时，则会释放内存。这种内存管理策略可以有效防止内存泄漏。