代码解读

整体流程：从语法树到中间代码

1. **输入：语法树（AST）**
    - 语法树是编译器前端生成的（经过了词法分析、语法分析、语义分析），表示程序的结构。每个节点是程序的一个部分，例如语句、表达式、函数定义等。
2. **输出：中间代码链表**
    - 中间代码通常以三地址代码形式表示，线性化后以链表形式存储。
    - 每条中间代码代表一个基本操作，如加法、赋值、函数调用等。
3. **调用过程和逻辑划分：**
    - 中间代码生成以语法树的 **节点类型** 和 **语义上下文** 为依据，按需递归调用相应的翻译函数。

代码解读

整个中间代码生成部分文件分为 intercode.c 和 intercode.h，其中核心代码在 intercode.c 中

intercode.c 中代码我将其分为三大类：翻译函数、优化函数、辅助函数

以下是详细的代码解读（在 intercode.c 代码中也给出了注释）

翻译函数代码解读

翻译的起点（main.c 中调用）`translateProgram`

1. 处理外部定义 `translateExtDef`

翻译开始是 `translateExtDef`，它处理外部定义（全局作用域的结构体、函数）。

- 注意假设 4：没有全局变量的使用，并且所有变量均不重名。——因此这里不用处理全局变量

> **外部定义**（External Definition）是编程语言中定义程序顶层实体的语法结构，通常出现在程序的最外层，**具有全局可见性**。
>
> 它们是构成整个程序的基本单元，在语法分析时位于语法树的根节点的直接子节点中。
>
> **包含内容：**
>
> - 全局变量声明或定义。
> - 函数声明或定义。
> - 用户定义的类型（如 `struct`、`enum`）。

流程：

1. 检查第一个子节点的类型：
    - 如果是结构体（`ENUM_STRUCT`），将其结构体定义存入符号表
    - 如果是函数定义，生成 `FUNCTION` 和 `PARAM` 中间代码，更新符号表
2. 对函数体（`CompSt`）递归调用 `translateCompSt`。

> 需要维护作用域：
>
> - 在进入 `CompSt` 前压栈作用域（`pushLayer`）。
> - 退出后弹栈作用域（`popLayer`）。

3. 将函数体生成的代码附加到 `FUNCTION` 的后面。
4. 返回生成的中间代码链表。

请完成 TODO1，完成符号表中插入符号。

- 可参考处理函数定义的过程

请完成 TODO2，生成“函数参数声明的中间代码。

- 可参考生成 FUCTION 函数定义的中间代码部分
- 注意：生成后别忘了插入到中间代码链表中

PARAM x	函数参数声明。

2. 函数体翻译 `translateCompSt`

`translateCompSt` 负责翻译复合语句（代码块）。

```
C
InterCode translateCompSt(Node* root, char* funcName);
```

调用逻辑:

- **输入：** 函数体的 AST 节点（`CompSt`）。
- **输出：** 表示整个代码块的中间代码链表。

流程：

1. 处理代码块内部的声明和语句:
    - **变量声明：** 调用 `translateDefList` 翻译局部变量声明。
    - **语句列表：** 遍历语句列表（`StmtList`），递归调用 `translateStmtList` 翻译每条语句。

---

3. 声明翻译 `translateDef`

流程：

1. **提取类型信息：**
调用 `Specifier` 获取变量类型。
2. **变量声明处理：**
调用 `translateDecList` 遍历声明的变量，并生成赋值或内存分配代码。
3. **特殊类型处理：**
对数组或结构体，生成 `DEC_IR` 分配内存的中间代码。

---

4. 变量声明翻译 `translateDec`

流程：

1. 如果变量有初始化值：生成赋值中间代码（`ASSIGN_IR`）。
    a. 创建一个临时变量 tmp1，用来保存初始化表达式的值
    b. 调用 translateExp 翻译初始化表达式（如 5 或更复杂的 b + c），生成表达式的中间代码，插入到中间代码链表中。
    c. 生成赋值语句的中间代码，将表达式结果 tmp1 赋值给变量（如 a），插

入到中间代码链表中

2. 否则 return

请完成 TODO3：生成赋值语句的中间代码，将表达式结果 tmp1 赋值给变量（如 a），插入到中间代码链表中

---

5. 语句翻译 `translateStmt`

`translateStmt` 处理单个语句节点（`Stmt`）。

流程：

1. 区分语句类型：
   - 对嵌套的语句块递归调用 `translateCompSt` 或 `translateStmt`。
   - **返回语句**：生成 `RETURN` 语句，其中 return x; 的 x（也可能是 a + b 这种复杂表达式）调用 `translateExp` 进行翻译。
   - **条件语句（if/else）**：调用 `translateCond` 翻译条件语句。
   - **循环语句（while）**：生成循环的跳转和条件代码。

---

6. 表达式翻译 `translateExp`

`translateExp` 是中间代码生成的核心，翻译单个表达式节点（`Exp`）。

调用逻辑：

- **输入：** 表达式节点和结果的目标操作数（`place`）。
- **输出：** 返回生成的操作数和表达式中间代码链表。

区分不同表达式分别进行处理，大致流程如下

1. **赋值表达式**:
   - **单变量赋值**: `ID = exp`
     - 获取变量地址 `var`，翻译右侧表达式到临时变量 `tmp1`。
     - 生成赋值指令，将 `tmp1` 的值存入 `var`。

- 将结果存储到 `place`。
- **数组元素赋值**: `array[index] = exp`
  - 计算数组地址及偏移量，得出具体的内存地址。
  - 翻译右侧表达式，并生成 `TO_MEM` 指令将值存入指定位置。
- **结构体域赋值**: `struct.field = exp`
  - 计算域的偏移量，得到内存地址。
  - 翻译右侧表达式，将值写入结构体的域。
  - 结果存储到 `place`。

2. **加减乘除表达式**: `exp1 op exp2`
   - 翻译两侧表达式到 `tmp1` 和 `tmp2`。
   - 调用优化函数（如：`optimizePLUSIR` 等），根据操作符生成相应的优化指令（`PLUS`, `SUB`, `MUL`, `DIV`）。
   - 结果存入 `place`。
3. **取负表达式**: `-exp`
   - 翻译右侧表达式。
   - 生成一个减法操作 `0 - exp`。
4. **括号表达式**: `(exp)`
   - 翻译括号内的表达式。
   - 优化：直接将结果存入 `place`，避免冗余操作。
5. **条件表达式**: `NOT exp`, `exp1 RELOP exp2`, `exp1 AND exp2`, `exp1 OR exp2`
   - 调用条件表达式翻译函数 `translateCond`，生成布尔值存入 `place`。
6. **函数调用表达式**: `func(args)`
   - **无参函数**:
     - 如果是 `read` 函数，生成读取指令存入 `place`。
     - 否则生成调用指令。
   - **带参函数**:
     - 翻译参数列表，并生成 `ARG` 指令。
     - 如果是 `write` 函数，直接输出参数值。
     - 生成函数调用指令，将返回值存入 `place`。

7. **单变量表达式**: `ID`
    - 根据变量类型（基本类型、数组、结构体），决定返回值或地址。
    - 优化：直接将变量地址或值存入 `place`。
8. **常量表达式**: `INT`
    - 直接将常量值赋给 `place`，避免多余指令。
9. **数组元素**: `array[index]`
    - 计算数组元素地址。
    - 根据元素类型（基本类型或复合类型），返回值或地址。
10. **结构体域访问**: `struct.field`
    - 计算域的偏移量。
    - 根据域类型，决定返回值或地址。

优化函数代码解读

这些优化函数的核心是对中间代码生成过程中常见的四则运算、分支跳转等指令进行优化，旨在通过以下优化提高生成代码的效率：

1. 减少冗余指令，如对常量运算直接求值、忽略不必要的运算。
2. 改善控制流，移除多余的跳转语句，提高代码执行效率。

1. `optimizePLUSIR` 函数

优化加法运算的中间代码生成，处理以下几种特殊情况：

- **两个操作数都是常量**：直接计算结果，生成一个返回值而不创建加法指令。
- **加法中一项为 0**：跳过加法，直接将另一操作数赋值为结果（前提是操作数不涉及地址计算）。
- **普通情况**：生成标准的加法中间代码。

2. `optimizeSUBIR` 函数

优化减法运算的中间代码生成，逻辑类似于加法优化

请完成 TODO4：优化减法运算的中间代码生成，逻辑类似于加法优化

- 不是完全和加法的一样，有个小坑要注意

3. `optimizeMULIR` 函数

优化减法运算的中间代码生成，逻辑类似于除法优化

请完成 TODO5：优化乘法运算的中间代码生成，逻辑类似于除法优化

4. `optimizeDIVIR` 函数

优化除法运算的中间代码生成：

- **两个操作数都是常量**：直接计算商。
- **分子为 0**：结果直接为 0。
- **除数为 1**：结果为分子。
- **普通情况**：生成标准的除法指令。

5. `optimizeLABELBeforeGOTO` 函数

优化冗余的 `LABEL` 和紧随其后的 `GOTO` 语句：

- 如果当前代码链中最后一条指令是 `LABEL`，它紧接着一个 `GOTO` 语句，则优化掉这个冗余的 `LABEL` 指令。
- 同时，更新与该 `LABEL` 相关的跳转目标，确保控制流正确指向下一个有效的目标。

剩下的就是一些辅助函数，这里就不赘述了

代码生成过程示例

假设输入代码：

```c
int add(int a, int b) {
    return a + b;
}
```

调用过程

1. **`translateExtDef`** 处理函数定义：
    - 生成 `FUNCTION add` 和参数声明 `PARAM a`, `PARAM b`。
    - 调用 `translateCompSt` 翻译函数体。
2. **`translateCompSt`** 翻译复合语句：
    - 调用 `translateStmt` 翻译 `return a + b`。
3. **`translateStmt`** 翻译 `return` 语句：
    - 调用 `translateExp` 翻译 `a + b`。
4. **`translateExp`** 翻译加法：
    - 检查是否可优化（如常量折叠）。
5. 调用`printInterCodes`函数输出