中间代码生成

程序功能(必做+部分选做)

程序接口说明

```
./parser in out
```

其中 `in` 为输入文件，转换生成的中间代码将存储在 `out` 文件中。输出目标文件缺省值为 `out.ir` ，另外还会生成 `original.ir` 文件，写入内容为未经过优化的代码。

中间代码生成

将中间代码输出成**线性结构**，存储在**双向链表**的结构中。

中间代码优化

实现了**常量折叠**和**代数运算简化**的优化功能。具体说来，先定位将常量赋值给临时变量的赋值语句，并在其后使用这个临时变量时将其替换为对应的常量；再对可能会由此产生一些可以直接进行计算的四则运算表达式进行求值。之后**迭代**进行常量赋值表达式消除和代数运算简化，直到产生的中间代码条数不发生变化，具体实现在 `optimize.c` 文件中的 `interCodeOptimize()` 函数。

实现思路

相关源码文件

```
.
├── ir.c
├── translate.c
└── optimize.c
```

数据结构设计

对于每一条中间代码，使用 `InterCode` 结构体类型，设计 `Operand_` 结构体类型表示每一个对应操作的相关信息。定义在 `ir.h` 文件中。

```
typedef struct Operand_ {
    enum { OP_VARIABLE, OP_TEMPORARY, OP_CONSTANT, OP_ADDRESS, OP_LABEL, OP_FUNCTI
ON, OP_VALUE, OP_SIZE } kind;
    union {
        int no;
        int value;
    } u;
} Operand_;

typedef struct InterCode {
    enum {
      IR_ADD, IR_SUB, IR_MUL, IR_DIV, IR_IFGOTO,
      IR_LABEL, IR_FUNC, IR_GOTO, IR_RET, IR_DEC, IR_ARG, IR_PARAM, IR_READ, IR_WR
ITE,
      IR_ASSIGN, IR_GETADD, IR_GETVAL, IR_SETVAL, IR_CALL
    } kind;

    union {
        struct { Operand x; } op;
        struct { Operand x, y; } biop;
        struct { Operand x, y, z; } triop;
        struct { Operand x, y, z; Relop relop; } ifop;
    };
} InterCode;
```

使用双向链表的结构存储中间代码，具体实现参考了Linux内核的 `list_head` 结构。

```
typedef struct InterCodes
{
    InterCode intercode;
    struct list_head list;
    bool isLeader;
} InterCodes;
```

翻译流程的架构设计

类似实验二中的自顶向下分析方法，从 `Program` 开始进行逐层函数调用，在底层进行中间代码生成，必要的时候返回调用者 `place` 参数，具体功能实现参照翻译模式。临时变量和标号的计数使用了全局变量。对于变量和函数的符号表，原则上可以沿用语义分析中的设计，我对之前的版本进行了完善。

数组的翻译模式设计

为提高编译器运行效率，使用循环而非递归进行数组的翻译模式设计，具体实现在 `translate.c` 中的 `translate_Array` 。

以四维数组 `int array[7][8][9][10]` 为例说明设计的翻译模式，设 `&v1` 表示 `array` 的地

址，`v2`，`v3`，`v4`，`v5`分别对应`i`，`j`，`k`，`s`的值。计算数组元素`array[i][j][k][s]`的地址，有

```
&v1 + v2 * 8 * 9 * 10 * 4 + v3 * 9 * 10 * 4 + v4 * 10 * 4 + v5 * 4
= &v1 + v2 * 8 * 9 * 10 * 4 + v3 * 9 * 10 * 4 + v4 * 10 * 4 + v5 * 4
= &v1 + (((v2 * 8 + v3) * 9 + v4) * 10 + v5) * 4
```

对应的中间代码翻译为

```
t2 := #8
t3 := #9
t4 := #10

t5 := v2 * t2
t6 := t5 * t3
t7 := t6 * t4
t8 := t7 * #4
t15 := &v1 + t8

t9 := v3 * t3
t10 := t9 * t4
t11 := t10 * #4
t16 := t15 + t11

t12 := v4 * t4
t13 := t12 * #4
t17 := t16 + t13

t14 := v5 * #4
t18 := t17 + t14
```

写在最后

个人觉得中间代码生成部分的必做实验难度要略低于语义分析部分。按照讲义上的指导一步步完成相对比较顺利。其中，耗时最长的数组的翻译模式设计，我使用了从特殊到一般的归纳法。先写出低维数组的翻译函数，然后类比推理出任意维的数组的翻译函数。另外，对于中间代码的优化，虽然我基本理解了课上老师介绍的使用DAG进行基本块内部优化的理论，但是在实践过程中遇到了重重的困难。经过两天的尝试，为了尽量保证中间生成代码的正确性，我最后选择了相对保守的、实现复杂度较低的优化策略，在实现复杂度、优化效果和正确性之前进行了权衡。

王芷芙 151220117@smail.nju.edu.cn