Report on Lab 3

编译方法

在 `Code` 文件夹下输入命令 `make` 即可自动编译生成 `parser` 文件。

功能简介

文件结构如下所示：

```
Code
├── Makefile
├── intercode.c
├── intercode.h
├── lexical.l
├── main.c
├── semantic.c
├── semantic.h
├── symboltable.c
├── symboltable.h
├── syntax.y
├── syntaxtree.c
├── syntaxtree.h
├── translate.c
└── translate.h
```

Lab 1：`lexical.h` 存放与词法分析相关的 `flex` 代码；`syntax.y` 存放与语法分析相关的 `bison` 代码；`syntaxtree.h` 和 `syntaxtree.c` 用于存放与语法树相关的结构体定义和函数。
Lab 2：`symboltable.c` 和 `symboltable.h` 存放与类型相关的结构体、函数和已定义的 Symbol/Structure/Function 表；`semantic.c` 和 `semantic.h` 存放进行语义分析的函数。
Lab 3：`main.c` 存放主函数；`intercode.h` 和 `intercode.c` 存放与中间代码的构造与操作相关的定义和函数；`translate.h` 和 `translate.c` 存放与翻译的函数。

IR表示

我使用双向链表存储IR，如下所示：

```
struct InterCodes {
    InterCode* head, *tail;
};

struct InterCode {
    enum {
        IC_LABEL, IC_FUNCTION,
        IC_ASSIGN, IC_PLUS, IC_MINUS, IC_PROD, IC_DIV, IC_ADDROF, IC_ASSIGNFROMADDR,
IC_ASSIGNTOADDR,
        IC_GOTO, IC_IFGOTO,
        IC_RETURN, IC_DEC, IC_ARG, IC_ASSIGNCALL, IC_PARAM,
        IC_READ, IC_WRITE
    } kind;
```

```
    char* arg1, *arg2, *dest;
    // For IC_FUNCTION, dest <- f
    // For IC_IFGOTO, arg1 <- x, arg2 <- y, dest <- z
    // For IC_ASSIGNCALL, dest <- x, arg1 <- f
    // For other InterCodes, dest <- x, arg1 <- y and arg2 <- z
    int size; // For IC_DEC
    InterRelop* relop; // For IC_IFGOTO

    InterCode* next, *prev;
};

struct InterRelop {
    enum {
        REL_LT, REL_GT, REL_LEQ, REL_GEQ, REL_EQ, REL_NEQ, REL_NEG
    } kind;
};
```

成员及作用见注释。

除此之外，我提供了对于IR的封装函数。任何与IR相关的修改或构造操作都必须使用封装的函数完成：

```
// new
char* new_temp();
char* new_label();

// InterCode
InterCode* new_InterCode(int kind, ...);
InterCodes* get_empty_InterCodes();
InterCodes* get_InterCode_wrapped(InterCode* code);
void append_InterCode(InterCodes* target, InterCode* cur);
void append_InterCodes(InterCodes* target, InterCodes* cur);

// Output
void fprint_InterRelop(FILE* f, InterRelop* relop);
void fprint_InterCode(FILE* f, InterCode* code);
void fprint_InterCodes(FILE* f, InterCodes* codes);

// ParamPass
void registerParam(char* name);
bool isregisteredParam(char* name);
```

其中 ParamPass 用于查询一个非基础的变量是否是通过传参被传递的。如果是，则说明该变量需要以引用形式进行访存。append_InterCode 和 append_InterCodes 分别会将存储单条IR/IR链表的 cur 拼接到 target 后方。

翻译

使用文档中给出的SDT即可。这里对于未提及的结构体和数组的翻译做出说明：

我在 `semantic.h/.c` 中添加了 `int getTypeSize(Type* ty);` 函数，用于返回一个特定的类型的总字节数；同时修改了 `symboltable.h` 中对于 `Type` 的定义：对于结构体类型新增 `int stru_size;` 用于存储结构体总字节数，同时在 `FieldList` 中新增 `offset` 项存储每一个项相对于结构体本身的偏移量、对于数组类型新增 `int elem_size;` 用于存储数组中元素的字节数。以上新增的信息全部在语义分析的过程中完成处理，在翻译过程中直接使用即可。

在处理赋值语句时，需要对左值的类型进行推导，具体会有以下情况：

- 左值是一个 EXP DOT ID，此时是结构体
- 左值是一个 EXP LB EXP RB，此时是数组
- 左值是一个 ID，此时可以直接赋值
  我实现了 `Offset_Query translate_Exp_Offset(Node* node, char* place)` 函数，它会递归地将当前项相对于其父项的偏移累加到 place 中，并返回累加所需要的 IR 和当前分析到的 Type。递归地终止条件是遇到了第三种情况（左值是一个ID），此时递归终止，并层层向上返回。返回的途中会完成 IR 的生成和拼接。

需要注意的是，C--允许数组之间的直接赋值。因此在赋值时，需要额外留意左值是否是一个未展开的数组。这可以通过此前实现的 `translate_Exp_Offset` 的返回值中对类型的描述完成判断：若整体类型为 BASIC 则直接赋值；反之生成将整个数组全部赋值的代码（不断在指针间赋值，赋值一次后两个指针同时 +4）。

此外的内容按照手册和文法进行判别即可。

测试数据补强

这里贴出一个在对于选做一的强测试数据（错误程序可通过OJ，但不能通过该测试数据）：

```
struct STRUCT_T {
    struct {
        int ssa[5];
        int ssb[5];
    } sa[10], sb[10];
};

int cross(struct STRUCT_T A, struct STRUCT_T B) {
    int temp[5] = A.sa[3].ssb;
    B.sa[9].ssa = temp;
    return 0;
}

int main() {
    struct STRUCT_T s, t;
    s.sa[4].ssa[2] = 114514;
    s.sb[7].ssb = s.sa[3].ssb = s.sb[5].ssa = s.sa[4].ssa;
    cross(s, t);
    write(t.sa[9].ssa[2]);
    return 0;
}
```

期望的输出是 `114514`。该测试数据的测试范围包括：

- 包含数组的结构体、结构体互相嵌套
- 结构体的局部定义和传参
- 结构体中数组整体赋值、赋值语句的返回值
- 数组在定义时直接赋值