Report on Lab 2

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
├── lexical.l
├── main.c
├── semantic.c
├── semantic.h
├── symboltable.c
├── symboltable.h
├── syntax.y
├── syntaxtree.c
└── syntaxtree.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：`main.c` 存放主函数；`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。

类型表示

我使用手册中推荐的方式存储类型，定义了如下的结构体：

```
struct Type {
    enum { BASIC, ARRAY, STRUCTURE } kind;
    union {
        int basic; // for BASIC; 0 for int, 1 for float
        struct { Type* elem; int size; }; // for ARRAY
        struct { FieldList* structure; FieldList* tail; }; // for STRUCTURE
    };
};
struct FieldList {
    Type* type;
    char* name;
    FieldList* next;
};
```

```
struct FunctionArg {
    Type* type;
    FunctionArg* next;
};
struct Function {
    char* name;
    Type* returnType;
    int argc;
    FunctionArg* argv, *tail;
};

Type* structTable[16384];
Type* symbolTable[16384];
Function* funcTable[16384];
```

成员及作用见注释。

这里想强调：类型是纯粹的类型，不含有任何与变量本身相关的信息（名称、是否为左值等）。与变量相关的信息需要额外存储（例如：结构体中域名存储在 `FieldList` 中）。

结构体在存储时使用链表存储，链表项为 `FieldList`，其中存储了该项的名称和类型；**函数**在存储时，存储名称、返回类型、参数数量和参数列表；参数列表的每一项 `FunctionArg` 存储该项的形参类型。

为了避免重复的代码，我为涉及到以上结构体和数组的操作尽最大可能**进行封装**，在 `semantic.c` 中尽量只使用以下的函数进行操作：

```
// CONSTRUCT
// basic type
Type* getTypeInt();
Type* getTypeFloat();

// array
Type* getTypeArray(Type* typeRight, int size);

// struct
Type* getTypeStruct();
bool haveFieldList();
FieldList* getFieldList(char* name, Type* type);
void addFieldListToTypeStruct(Type* tyStruct, FieldList* fieldList);

// function
// Function* getFunc(char* name, Type* returnType, int argc, ...);
Function* getFunc(char* name, Type* returnType);
FunctionArg* getFunctionArg(Type* type);
void addFunctionArgToFunc(Function* func, FunctionArg* arg);

// INSERT
void insertStruct(char* name, Type* ty);
void insertSymbol(char* name, Type* ty);
void insertFunc(Function* func);
```

```
// QUERY
Type* querySymbol(char* name);
Type* queryStruct(char* name);
Function* queryFunc(char* name);
bool isSameType(Type* x, Type* y);
bool isMatchFuncArg(Function* funcx, Function* funcy);

// DEBUG
void OutputType(Type* ty, int depth);
```

这样做的好处还在于可以进行单元测试。我单独为 `symboltable.c` 中的函数设计了测试，但由于后续接口发生变动，测试代码没有反馈在最终版本的提交中。

AST遍历

对于每一个非终结符，我编写了一个对应的函数进行解析。为了方便起见，我加入了 `Node* getSon(Node* node, char* name)` 函数用于获取一个终结符的特定名称的儿子。

使用 `getSon` 进行AST的遍历是十分有必要的，因为**遍历时不能提前确定某个儿子是否存在**。例如：$CompSt \rightarrow LC\ DefList\ StmtList\ RC$，虽然 $CompSt$ 和 $DefList$ 都没有直接推导出 ε 的能力，但是 $DefList$ 可能会经过多步推导规约为 ε，从而不存在于AST中。于是，可能会出现 $CompSt$ 的子节点只有 $LC\ StmtList\ RC$ 的情况。所以为了能够精确判别和定位某个子节点，应该使用 `getSon` 而不是直接以 `node->son->next->next` 的指针形式访问子节点。

为了简化程序（不显式地引入综合属性和继承属性），我**利用递归的传参完成了属性的传递**。对于inh属性向下传递，我将其放置在下层函数的参数中；对于inh属性的向上传递，我将其放置在下层函数的返回值中。使用这样的方法，我让程序更加简洁易懂。例如：`Type* Specifier(Node* node)` 返回解析好的 `Specifier` 的类型指针；`void Stmt(Node* node, Function* func)` 在正常的结点指针之外额外接受一个当前所在的函数，用于在遇到 `RETURN` 时判定返回值是否合法。

此外的内容按照手册和文法进行判别即可。