Lab2：语法分析

include/tree.h:语法树节点父类

include/SysY_tree.h:SysY文法所有语法树节点类的定义

utils/ast_out.cc:语法树输出函数实现

include/type.h:语法树节点属性类的定义

需要阅读并编写的代码:

parser/SysY_parser.y:编写你想要实现的文法定义以及对应的处理函数, 本次实验中只需要构建出语法树即可，不需要其他的额外处理

Codes Reading

include/tree.h

```
#ifndef TREE_H
#define TREE_H

// tree definition (used for AST)
#include "type.h"
#include <iostream>

class tree_node {
protected:
    int line_number;

public:
    NodeAttribute attribute;    // 在类型检查阶段需要阅读该类的代码，语法分析阶段可以先忽略该变量
    int GetLineNumber() { return line_number; }
    void SetLineNumber(int t) { line_number = t; }

    virtual void codeIR() = 0;                                  // 中间代码生成
    virtual void printAST(std::ostream &s, int pad) = 0;    // 打印语法树
    virtual void TypeCheck() = 0;                               // 类型检查
};

#endif
```

该部分定义了**抽象语法树AST节点**的类**tree_node**，定义：

- 保护变量line_number，也就是行数
- 公共变量
  - attribute：用于类型检查
  - GetLineNumber函数，返回程序的行数line_number
  - SetLineNumber函数，传入参数t，赋值行数
  - codeIR函数，用于生成中间代码

- printAST函数，打印抽象语法树
- TypeCheck函数，用于类型检查

include/SysY_tree.h

定义__Expression**表达式类**，拥有tree_node的所有成员变量和成员函数，然后创建一个新的类型别名`Expression`，这个别名代表指向`__Expression`类对象的指针类型。

```
// exp basic_class
class __Expression : public tree_node {
public:
};
typedef __Expression *Expression;
```

实现AddExp功能，定义一个Exp类，Exp继承了__Expression类的所有成员变量和成员函数，然后定义了很多的公共变量：

- **addexp**：存储与加法表达式相关的子表达式
- **Exp(Expression add) : addexp(add) {}**：构造函数，接受一个表达式指针作为参数，用于初始化`addexp`成员变量

```
// AddExp
class Exp : public __Expression {
public:
    Expression addexp;
    Exp(Expression add) : addexp(add) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

实现复合运算功能，定义了AddExp_plus类，有两个表达式addexp和mulexp，分别为前面的和后面的表达式

AddExp_plus表示，两个表达式做加法

```
// AddExp + MulExp
class AddExp_plus : public __Expression {
public:
    Expression addexp;
    Expression mulexp;
    AddExp_plus(Expression add, Expression mul) : addexp(add), mulexp(mul) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

实现了减法的复合运算，定义了AddExp_sub，表示两个表达式做减法

```
// AddExp - MulExp
class AddExp_sub : public __Expression {
public:
    Expression addexp;
    Expression mulexp;
    AddExp_sub(Expression add, Expression mul) : addexp(add), mulexp(mul) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

实现了乘法的复合运算，MulExp_mul两个表达式相乘

```
// MulExp * UnaryExp
class MulExp_mul : public __Expression {
public:
    Expression mulexp;
    Expression unary_exp;
    MulExp_mul(Expression mul, Expression unary) : mulexp(mul), unary_exp(unary)
{}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

实现了除法的复合运算，定义了MulExp_div，两个表达式相除

```
// MulExp / UnaryExp
class MulExp_div : public __Expression {
public:
    Expression mulexp;
    Expression unary_exp;
    MulExp_div(Expression mul, Expression unary) : mulexp(mul), unary_exp(unary)
{}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

实现了取余的运算，定义了MulExp_mod，两个表达式相取余

```
// MulExp % UnaryExp
class MulExp_mod : public __Expression {
public:
    Expression mulexp;
    Expression unary_exp;
    MulExp_mod(Expression mul, Expression unary) : mulexp(mul), unary_exp(unary)
{}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

实现了：

- RelExp_leq：实现小于等于的运算 <=
- RelExp_lt：实现了小于的运算 <
- RelExp_geq：实现了大于等于的运算 >=
- RelExp_gt：实现了大于的运算 >
- EqExp_eq：实现了等于的运算 ==
- EqExp_neq：实现了不等于的运算！ =

```
// RelExp <= AddExp
class RelExp_leq : public __Expression {
public:
    Expression relexp;
    Expression addexp;
    // add constructor
    RelExp_leq(Expression relexp, Expression addexp) : relexp(relexp),
addexp(addexp) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// RelExp < AddExp
class RelExp_lt : public __Expression {
public:
    Expression relexp;
    Expression addexp;
    RelExp_lt(Expression relexp, Expression addexp) : relexp(relexp),
addexp(addexp) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// RelExp >= AddExp
class RelExp_geq : public __Expression {
public:
    Expression relexp;
    Expression addexp;
    RelExp_geq(Expression relexp, Expression addexp) : relexp(relexp),
addexp(addexp) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// RelExp > AddExp
class RelExp_gt : public __Expression {
public:
    Expression relexp;
    Expression addexp;
    RelExp_gt(Expression relexp, Expression addexp) : relexp(relexp),
addexp(addexp) {}
```

```
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// EqExp == RelExp
class EqExp_eq : public __Expression {
public:
    Expression eqexp;
    Expression relexp;
    EqExp_eq(Expression eqexp, Expression relexp) : eqexp(eqexp), relexp(relexp)
{}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// EqExp != RelExp
class EqExp_neq : public __Expression {
public:
    Expression eqexp;
    Expression relexp;
    EqExp_neq(Expression eqexp, Expression relexp) : eqexp(eqexp), relexp(relexp)
{}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

LAndExp_and实现了&&的运算

LOrExp_or实现了||的运算

```
// LAndExp && EqExp
class LAndExp_and : public __Expression {
public:
    Expression landexp;
    Expression eqexp;
    LAndExp_and(Expression landexp, Expression eqexp) : landexp(landexp),
eqexp(eqexp) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// LOrExp || LAndExp
class LOrExp_or : public __Expression {
public:
    Expression lorexp;
    Expression landexp;
    ;
    LOrExp_or(Expression lorexp, Expression landexp) : lorexp(lorexp),
landexp(landexp) {}
    void codeIR();
    void TypeCheck();
```

```
    void printAST(std::ostream &s, int pad);
};
```

ConstExp类，构造了一个常量表达式

```
// AddExp
class ConstExp : public __Expression {
public:
    Expression addexp;
    ConstExp(Expression addexp) : addexp(addexp) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

用于存储数组的[]内容，dims为数组下标，lval类实现了数组下标的表示

```
// lval   name{[dim]}   eg. a[4][5+4][3+4*9]
class Lval : public __Expression {
public:
    Symbol name;
    std::vector<Expression> *dims;
    // 如果dims为nullptr，表示该变量不含数组下标，你也可以通过其他方式判断，但需要修改
SysY_parser.y已有的代码

    int scope = -1;    // 在语义分析阶段填入正确的作用域
    Lval(Symbol n, std::vector<Expression> *d) : name(n), dims(d) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

存储参数表达式的列表，FuncRParams用于初始化params的成员变量

```
//{Exp,Exp,Exp,Exp}
class FuncRParams : public __Expression {
public:
    std::vector<Expression> *params{};
    FuncRParams(std::vector<Expression> *p) : params(p) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

调用函数，Func_call实现了调用函数的效果，name为函数的参数，f为函数

```
// name(FuncRParams)
class Func_call : public __Expression {
public:
    Symbol name;
    Expression funcr_params;
    Func_call(Symbol n, Expression f) : name(n), funcr_params(f) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

存储一元表达式

- UnaryExp_plus：表示 + 一元表达式
- UnaryExp_neg：表示 - 一元表达式
- UnaryExp_not：表示! 一元表达式

```
// + UnaryExp
class UnaryExp_plus : public __Expression {
public:
    Expression unary_exp;
    UnaryExp_plus(Expression unary_exp) : unary_exp(unary_exp) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// - UnaryExp
class UnaryExp_neg : public __Expression {
public:
    Expression unary_exp;
    UnaryExp_neg(Expression unary_exp) : unary_exp(unary_exp) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// ! UnaryExp
class UnaryExp_not : public __Expression {
public:
    Expression unary_exp;
    UnaryExp_not(Expression unary_exp) : unary_exp(unary_exp) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

常量类：

- IntConst，表示一个int型的常量
- FloatConst：表示一个浮点型的常量
- StringConst：表示一个字符串型的常量

```
class IntConst : public __Expression {
public:
    int val;
    IntConst(int v) : val(v) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class FloatConst : public __Expression {
public:
    float val;
    FloatConst(float v) : val(v) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class StringConst : public __Expression {
public:
    Symbol str;
    StringConst(Symbol s) : str(s) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

PrimaryExp_branch表示，对表达式进行括号操作

```
//( Exp )
class PrimaryExp_branch : public __Expression {
public:
    Expression exp;
    PrimaryExp_branch(Expression exp) : exp(exp) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

定义了__Block类和__Stmt类，__Stmt类继承所有的tree_node的成员变量和成员函数

```
class __Block;
typedef __Block *Block;
// stmt basic_class
class __Stmt : public tree_node {
public:
};
typedef __Stmt *Stmt;
```

定义了一系列的语句子类，都需要打印抽象语法树

- null_stmt：表示空语句，打印抽象语法树
- assign_stmt：表示左边等于右边
- expr_stmt：表示表达式

- block_stmt：表示一个块
- ifelse_stmt：ifelse语句
- if_stmt：if语句
- while_stmt：while语句
- continue_stmt：continue语句
- break_stmt：break语句
- return_stmt：return语句
- return_stmt_void：return空值的语句

```
class null_stmt : public __Stmt {
public:
    void codeIR() {}
    void TypeCheck() {}
    void printAST(std::ostream &s, int pad);
};

// lval = exp;
class assign_stmt : public __Stmt {
public:
    Expression lval;
    Expression exp;
    // construction
    assign_stmt(Expression l, Expression e) : lval(l), exp(e) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// exp;
class expr_stmt : public __Stmt {
public:
    Expression exp;
    expr_stmt(Expression e) : exp(e) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// block
class block_stmt : public __Stmt {
public:
    Block b;
    block_stmt(Block b) : b(b) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class ifelse_stmt : public __Stmt {
public:
    Expression Cond;
    Stmt ifstmt;
```

```
    Stmt elsestmt;    // else
    // construction
    ifelse_stmt(Expression c, Stmt i, Stmt t) : Cond(c), ifstmt(i), elsestmt(t)
{}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// only if
class if_stmt : public __Stmt {
public:
    Expression Cond;
    Stmt ifstmt;
    // construction
    if_stmt(Expression c, Stmt i) : Cond(c), ifstmt(i) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class while_stmt : public __Stmt {
public:
    Expression Cond;
    Stmt body;
    // construction
    while_stmt(Expression c, Stmt b) : Cond(c), body(b) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// continue;
class continue_stmt : public __Stmt {
public:
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// break;
class break_stmt : public __Stmt {
public:
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// return return_exp;
class return_stmt : public __Stmt {
public:
    Expression return_exp;
    return_stmt(Expression r) : return_exp(r) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
```

```
};

class return_stmt_void : public __Stmt {
public:
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

声明类的基类

- __InitVal：初始化基类
- ConstInitVal：由多个`InitVal`组成的列表形式的初始化值，例如`{2,3,{4,5,6},{3,{4,5,6,{3,5}}}}`
- ConstInitVal_exp：用于表示以单个表达式作为常量初始化值的情况
- VarInitVal：和上面差不多，为变量的初始化
- VarInitVal_exp：用于表示以单个表达式作为变量初始化的情况

```
class __Decl;
typedef __Decl *Decl;

class __InitVal : public tree_node {
public:
};
typedef __InitVal *InitVal;

// InitVal -> {InitVal,InitVal,InitVal,...}
// eg. {2,3,{4,5,6},{3,{4,5,6,{3,5}}}}
class ConstInitVal : public __InitVal {
public:
    std::vector<InitVal> *initval;
    ConstInitVal(std::vector<InitVal> *i) : initval(i) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class ConstInitVal_exp : public __InitVal {
public:
    Expression exp;
    ConstInitVal_exp(Expression e) : exp(e) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// InitVal -> {InitVal,InitVal,InitVal,...}
class VarInitVal : public __InitVal {
public:
    std::vector<InitVal> *initval;
    VarInitVal(std::vector<InitVal> *i) : initval(i) {}
    void codeIR();
    void TypeCheck();
```

```
    void printAST(std::ostream &s, int pad);
};

class VarInitVal_exp : public __InitVal {
public:
    Expression exp;
    VarInitVal_exp(Expression e) : exp(e) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

__Def类：

- VarDef_no_init：表示没有初始化值的变量定义
- VarDef：具有初始化值的变量定义
- ConstDef：常量定义

```
class __Def : public tree_node {
public:
    int scope = -1;    // 在语义分析阶段填入正确的作用域
};
typedef __Def *Def;

class VarDef_no_init : public __Def {
public:
    Symbol name;
    std::vector<Expression> *dims;
    // 如果dims为nullptr，表示该变量不含数组下标，你也可以通过其他方式判断，但需要修改
SysY_parser.y已有的代码
    VarDef_no_init(Symbol n, std::vector<Expression> *d) : name(n), dims(d) {}

    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class VarDef : public __Def {
public:
    Symbol name;
    std::vector<Expression> *dims;
    // 如果dims为nullptr，表示该变量不含数组下标，你也可以通过其他方式判断，但需要修改
SysY_parser.y已有的代码
    InitVal init;
    VarDef(Symbol n, std::vector<Expression> *d, InitVal i) : name(n), dims(d),
init(i) {}

    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class ConstDef : public __Def {
public:
```

```
    Symbol name;
    std::vector<Expression> *dims;
    // 如果dims为nullptr，表示该变量不含数组下标，你也可以通过其他方式判断，但需要修改
SysY_parser.y已有的代码
    InitVal init;
    ConstDef(Symbol n, std::vector<Expression> *d, InitVal i) : name(n), dims(d),
init(i) {}

    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

__Decl类：声明

- VarDecl：变量声明
- ConstDecl：常量声明

```
// decl basic_class
class __Decl : public tree_node {
public:
};

// var definition
class VarDecl : public __Decl {
public:
    Type::ty type_decl;
    std::vector<Def> *var_def_list{};
    // construction
    VarDecl(Type::ty t, std::vector<Def> *v) : type_decl(t), var_def_list(v) {}

    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

// const var definition
class ConstDecl : public __Decl {
public:
    Type::ty type_decl;
    std::vector<Def> *var_def_list{};
    // construction
    ConstDecl(Type::ty t, std::vector<Def> *v) : type_decl(t), var_def_list(v) {}

    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

__BlockItem：

- BlockItem_Decl：定义
- BlockItem_Stmt：语句

```
```

```
class __BlockItem : public tree_node {
public:
};
typedef __BlockItem *BlockItem;

class BlockItem_Decl : public __BlockItem {
public:
    Decl decl;
    BlockItem_Decl(Decl d) : decl(d) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class BlockItem_Stmt : public __BlockItem {
public:
    Stmt stmt;
    BlockItem_Stmt(Stmt s) : stmt(s) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

__Block类：代码块

```
// block
class __Block : public tree_node {
public:
    std::vector<BlockItem> *item_list{};
    // construction
    __Block() {}
    __Block(std::vector<BlockItem> *i) : item_list(i) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
```

- __FuncFParam:函数的参数
- __FuncDef:函数的定义

```
// FuncParam -> Type IDENT
// FuncParam -> Type IDENT [] {[Exp]}
class __FuncFParam : public tree_node {
public:
    Type::ty type_decl;
    std::vector<Expression> *dims;
    // 如果dims为nullptr，表示该变量不含数组下标，你也可以通过其他方式判断，但需要修改
SysY_parser.y已有的代码
    Symbol name;
    int scope = -1;    // 在语义分析阶段填入正确的作用域

    __FuncFParam(Type::ty t, Symbol n, std::vector<Expression> *d) {
        type_decl = t;
        name = n;
```

```
        dims = d;
    }
    __FuncFParam(Type::ty t, std::vector<Expression> *d) {
        type_decl = t;
        dims = d;
    }
    __FuncFParam(Type::ty t) : type_decl(t) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
typedef __FuncFParam *FuncFParam;

// return_type name '(' [formals] ')' block
class __FuncDef : public tree_node {
public:
    Type::ty return_type;
    Symbol name;
    std::vector<FuncFParam> *formals;
    Block block;
    __FuncDef(Type::ty t, Symbol functionName, std::vector<FuncFParam> *f, Block
b) {
        formals = f;
        name = functionName;
        return_type = t;
        block = b;
    }
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
typedef __FuncDef *FuncDef;
```

__CompUnit：独立可编译的单元

- CompUnit_Decl：编译单元的声明部分
- CompUnit_FuncDef：编译单元的函数定义部分

__Program：表示整个程序

```
class __CompUnit : public tree_node {
public:
};
typedef __CompUnit *CompUnit;

class CompUnit_Decl : public __CompUnit {
public:
    Decl decl;
    CompUnit_Decl(Decl d) : decl(d) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class CompUnit_FuncDef : public __CompUnit {
```

```
public:
    FuncDef func_def;
    CompUnit_FuncDef(FuncDef f) : func_def(f) {}
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};

class __Program : public tree_node {
public:
    std::vector<CompUnit> *comp_list;

    __Program(std::vector<CompUnit> *c) { comp_list = c; }
    void codeIR();
    void TypeCheck();
    void printAST(std::ostream &s, int pad);
};
typedef __Program *Program;
#endif
```

utils/ast_out.cc

```
std::string type_status[5] = {"Void", "Int", "Float", "Bool", "Ptr"};

std::string Type::GetTypeInfo() { return "Type: " + type_status[type]; }
```

声明了五个类型，无值、int、浮点型、bool型、指针型

GetTypeInfo函数表示返回对应的类型

```
std::string ConstValue::GetConstValueInfo(Type ty) {
    if (!ConstTag) {
        return "";
    }

    if (ty.type == Type::INT) {
        return "ConstValue: " + std::to_string(val.IntVal);
    } else if (ty.type == Type::FLOAT) {
        return "ConstValue: " + std::to_string(val.FloatVal);
    } else if (ty.type == Type::BOOL) {
        return "ConstValue: " + std::to_string(val.BoolVal);
    } else {
        return "";
    }
}
```

GetConstValueInfo函数表示返回对应的常量值

```
std::string NodeAttribute::GetAttributeInfo() { return T.GetTypeInfo() + "    " +
V.GetConstValueInfo(T); }
```

GetAttributeInfo函数表示返回节点的属性

```
void __Program::printAST(std::ostream &s, int pad) {
    s << std::string(pad, ' ') << "Program\n";
    if (comp_list != nullptr) {
        for (auto comp : (*comp_list)) {
            comp->printAST(s, pad + 2);
        }
    }
}
```

开始打印抽象语法树了

```
void CompUnit_Decl::printAST(std::ostream &s, int pad) { decl->printAST(s, pad);
}
```

```
void VarDecl::printAST(std::ostream &s, int pad) {
    s << std::string(pad, ' ') << "VarDecls   "
      << "Type: " << type_status[type_decl] << "\n";
    if (var_def_list != nullptr) {
        for (auto var : (*var_def_list)) {
            var->printAST(s, pad + 2);
        }
    }
}
```

后面那些printAST都是打印语法树的，一样的就不介绍了

```
void __FuncFParam::printAST(std::ostream &s, int pad) {
    s << std::string(pad, ' ') << "FuncFParam   name:" << name->get_string() << "
  Type:" << type_status[type_decl]
      << "   "
      << "scope:" << scope << "\n";
    if (dims != nullptr) {
        s << std::string(pad + 2, ' ') << "Dimensions:\n";
        for (auto dim : (*dims)) {
            if (dim == nullptr) {
                s << std::string(pad + 4, ' ') << "Null dim\n";
            } else {
                dim->printAST(s, pad + 4);
            }
        }
    }
    s << "\n";
}
```

打印函数的这个稍微看一下。

首先输出参数的名称和类型，还有作用域

然后打印维度信息

```
void assign_stmt::printAST(std::ostream &s, int pad) {
    s << std::string(pad, ' ') << "AssignStmt:\n";

    lval->printAST(s, pad + 2);
    exp->printAST(s, pad + 2);
}
```

有关stmt的打印，我们在前面需要加上AssignStmt：

其他就都是打印的情况了，我们再完善一下，添加一点其他的输出情况即可

include/type.h

```
class Type {
public:
    // 我们认为数组的类型为PTR
    enum ty { VOID = 0, INT = 1, FLOAT = 2, BOOL = 3, PTR = 4, DOUBLE = 5 } type;
    std::string GetTypeInfo();
    Type() { type = VOID; }
};
```

定义了数据类型，一共有五种数据类型VOID = 0, INT = 1, FLOAT = 2, BOOL = 3, PTR = 4, DOUBLE = 5

```
class ConstValue {
public:
    bool ConstTag;
    union ConstVal {
        bool BoolVal;
        int IntVal;
        float FloatVal;
        double DoubleVal;
    } val;
    std::string GetConstValueInfo(Type ty);
    ConstValue() {
        val.IntVal = 0;
        ConstTag = false;
    }
};
```

定义了常量型，里面含有bool BoolVal;int IntVal;float FloatVal;double DoubleVal;

```
class VarAttribute {
public:
    Type::ty type;
    bool ConstTag = 0;
    std::vector<int> dims{};    // 存储数组类型的相关信息
    std::vector<int> IntInitVals{};
    std::vector<float> FloatInitVals{};

    // TODO():也许你需要添加更多变量
    VarAttribute() {
        type = Type::VOID;
```

```
        ConstTag = false;
    }
};
```

定义变量类型。

- type表示变量类型的枚举类型成员变量
- ConstTag表示变量是否为常量
- dims表示一个整数向量，用于存储数组类型变量的维度信息
- IntInitVals表示整数类型变量的初始值
- FloatInitVals表示浮点数类型变量的初始值

```
class NodeAttribute {
public:
    int line_number = -1;
    Type T;
    ConstValue V;
    std::string GetAttributeInfo();
};
```

定义了节点的属性，行号初始化为-1，相当于行号是未知的（啥意思？）

语法分析总流程

首先是进入main.cc文件，前面跟lexer是一样的，不多提

进入main函数

进入parser

```
    if (strcmp(argv[step_tag], "-parser") == 0) {
        ast_root->printAST(fout, 0);
        fout.close();
        return 0;
    }

    ast_root->TypeCheck();
    if (error_msgs.size() > 0) {
        for (auto msg : error_msgs) {
            fout << msg << std::endl;
        }
        fout.close();
        return 0;
    }
```

从root节点开始打印抽象语法树的操作，然后递归调用，直到把整个树打印完

然后关闭fout文件流，开始执行类型检查，检查节点的信息是否正确。如果有error的话就报错，直接退出程序

SysY_parser.tab.cc文件中，定义了程序ast_root

```
Program ast_root;
```

调用printAST函数

```
void __Program::printAST(std::ostream &s, int pad) {
    s << std::string(pad, ' ') << "Program\n";
    if (comp_list != nullptr) {
        for (auto comp : (*comp_list)) {
            comp->printAST(s, pad + 2);
        }
    }
}
```

相当于，程序去打印最小编译单元，然后comp调用printAST，前面先空两格

```
    std::vector<CompUnit> *comp_list;
```

相当于是CompUnit类型的

然后调用CompUnit_FuncDef和CompUnit_Decl

```
void CompUnit_Decl::printAST(std::ostream &s, int pad) { decl->printAST(s, pad);
}

void CompUnit_FuncDef::printAST(std::ostream &s, int pad) { func_def->printAST(s,
pad); }
```

Questions

请说明yylval，yytext 以及词法分析中的返回值在语法分析阶段是如何被使用的。(如果被使用到的话)

返回值

词法分析阶段，我们返回的是字符的类型yytokentype

```
/* Token kinds.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
  enum yytokentype
  {
    YYEMPTY = -2,
    YYEOF = 0,                     /* "end of file"  */
    YYerror = 256,                 /* error  */
    YYUNDEF = 257,                 /* "invalid token"  */
    STR_CONST = 258,               /* STR_CONST  */
    IDENT = 259,                   /* IDENT  */
    FLOAT_CONST = 260,             /* FLOAT_CONST  */
    INT_CONST = 261,               /* INT_CONST  */
```

```
    LEQ = 262,                     /* LEQ  */
    GEQ = 263,                     /* GEQ  */
    EQ = 264,                      /* EQ  */
    NE = 265,                      /* NE  */
    AND = 266,                     /* AND  */
    OR = 267,                      /* OR  */
    ADDASSIGN = 268,               /* ADDASSIGN  */
    MULASSIGN = 269,               /* MULASSIGN  */
    SUBASSIGN = 270,               /* SUBASSIGN  */
    DIVASSIGN = 271,               /* DIVASSIGN  */
    MODASSIGN = 272,               /* MODASSIGN  */
    CONST = 273,                   /* CONST  */
    IF = 274,                      /* IF  */
    ELSE = 275,                    /* ELSE  */
    WHILE = 276,                   /* WHILE  */
    NONE_TYPE = 277,               /* NONE_TYPE  */
    INT = 278,                     /* INT  */
    FLOAT = 279,                   /* FLOAT  */
    FOR = 280,                     /* FOR  */
    RETURN = 281,                  /* RETURN  */
    BREAK = 282,                   /* BREAK  */
    CONTINUE = 283,                /* CONTINUE  */
    ERROR = 284,                   /* ERROR  */
    THEN = 285                     /* THEN  */
  };
  typedef enum yytokentype yytoken_kind_t;
#endif
```

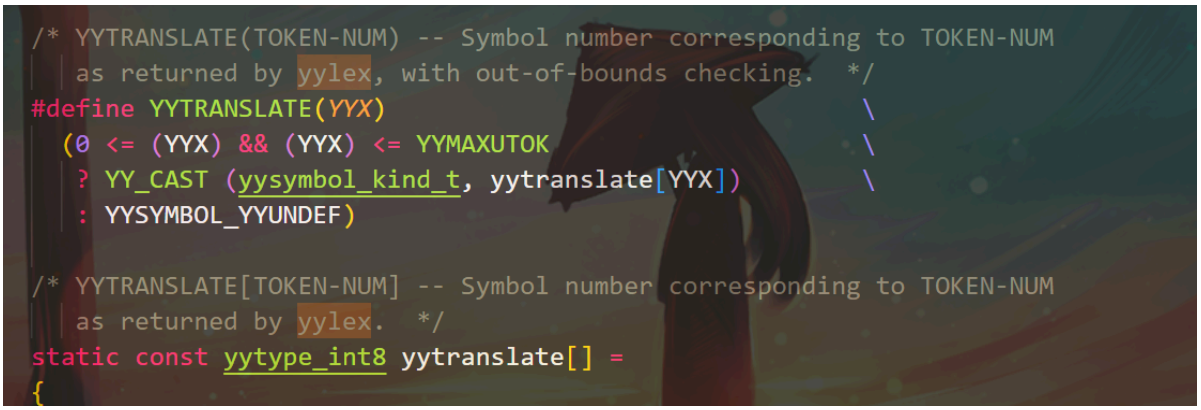
首先，用yychar记录yylex的返回值

```
  if (yychar == YYEMPTY)
    {
      YYDPRINTF ((stderr, "Reading a token\n"));
      yychar = yylex ();
    }

  if (yychar <= YYEOF)
    {
      yychar = YYEOF;
      yytoken = YYSYMBOL_YYEOF;
      YYDPRINTF ((stderr, "Now at end of input.\n"));
    }
  else if (yychar == YYerror)
    {
      /* The scanner already issued an error message, process directly
         to error recovery.  But do not keep the error token as
         lookahead, it is too special and may lead us to an endless
         loop in error recovery. */
      yychar = YYUNDEF;
      yytoken = YYSYMBOL_YYerror;
      yyerror_range[1] = yylloc;
      goto yyerrlab1;
    }
  else
    {
      yytoken = YYTRANSLATE (yychar);
      YY_SYMBOL_PRINT ("Next token is", yytoken, &yylval, &yylloc);
    }
```

当yychar为空的时候，记录返回值

调用YYTRANSLATE（550行）



```
/* YYTRANSLATE(TOKEN-NUM) -- Symbol number corresponding to TOKEN-NUM
   as returned by yylex, with out-of-bounds checking.  */
#define YYTRANSLATE(YYX)                                \
  (0 <= (YYX) && (YYX) <= YYMAXUTOK                     \
   ? YY_CAST (yysymbol_kind_t, yytranslate[YYX])        \
   : YYSYMBOL_YYUNDEF)

/* YYTRANSLATE[TOKEN-NUM] -- Symbol number corresponding to TOKEN-NUM
   as returned by yylex.  */
static const yytype_int8 yytranslate[] =
{
```

如果在范围内，就用

这个yytranslate表，将我们原来的词法分析的yytokentype转化为yysymbol_kind_t（映射）

```
static const yytype_int8 yytranslate[] =
{
       0,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,    42,     2,     2,     2,    45,     2,     2,
      33,    34,    43,    40,    32,    41,     2,    44,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,    31,
      46,    35,    47,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,    38,     2,    39,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,    36,     2,    37,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     2,     2,     2,     2,
       2,     2,     2,     2,     2,     2,     1,     2,     3,     4,
       5,     6,     7,     8,     9,    10,    11,    12,    13,    14,
      15,    16,    17,    18,    19,    20,    21,    22,    23,    24,
      25,    26,    27,    28,    29,    30
};
```

yylval

我们使用$$等内容宏定义的时候，就是yylval是一个union型的。从里面取出相应类型的元素

yytext

未使用

请说明你是如何处理 if 和 if-else 的移进-规约冲突的，举一个会出现移进-规约冲突的 if-else 例子。

声明优先级：

```
%precedence THEN
%precedence ELSE
```

优先级的话，else比then要高

```
|IF '(' Cond ')' Stmt %prec THEN
{
    $$ = new if_stmt($3,$5);
    $$->SetLineNumber(line_number);
}
|IF '(' Cond ')' Stmt ELSE Stmt
{
    $$ = new ifelse_stmt($3,$5,$7);
    $$->SetLineNumber(line_number);
}
```

你可能会对代码中的%prec THEN 感到疑惑，这是为了解决悬空-else 文法的二义性问题, 考虑下面的 if 语句文法：

$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt$

$stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt$

在语法分析处于如下状态时:

$\textbf{if}\ expr\ \textbf{then if}\ expr\ \textbf{then}\ stmt \cdot \textbf{else}\ stmt$

我们可以将终结符 **else** 移入，也可以使用产生式 $stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt$ 进行归约，这时发生了移入/归约冲突，而正确的做法是将 **else** 移入。

在 yacc 中我们可以给终结符声明优先级：

%precedence **then**

%precedence **else**

这样终结符 **else** 的优先级高于终结符 **then**。产生式的优先级和右部最后一个终结符的优先级相同，即产生式 $stmt \rightarrow \textbf{if}\ expr\ \textbf{then}\ stmt$ 的优先级和终结符 **then** 的优先级相同。在发生移入/归约冲突时，通过比较向前看符号和产生式的优先级来解决冲突，若向前看符号的优先级更高，则进行移入，若产生式的优先级更高，则进行归约。这里 **else** 的优先级更高，因此会将 **else** 移入。

SysY 语言中的 if 语句并没有终结符 **then**, 在 yacc 中我们可以使用%prec 关键字, 将终结符 **then** 的优先级赋给产生式。

这个意思是，识别到if then if then else的时候，我们由于会先去把else放进去，然后完成后面的匹配工作

请说明语法树的根节点是什么类型，该根节点的子结点可能有哪些类型。

Program类型

```
Program ast_root;

    std::vector<CompUnit> *comp_list;

    __Program(std::vector<CompUnit> *c) { comp_list = c; }
```

子节点：就是comp_list的类型，是CompUnit

CompUnit分为**CompUnit_FuncDef和CompUnit_Decl**

- CompUnit_Decl：最小编译单元中的量的声明（变量或者常量）
- CompUnit_FuncDef：函数
    - 首先打印函数名
    - 然后打印返回类型
    - 遍历参数表
    - 然后打印block

SysY_parser.y 中，%union，%token, %type 分别是什么意思，对应上下文无关文法的哪些部分（如果有对应的话）

```
%union{
    char* error_msg;
    Symbol symbol_token;
    double float_token; // 对于SysY的浮点常量，我们需要先以double类型计算，再在语法树节点创建的时候转为float
    int int_token;
    Program program;
    CompUnit comp_unit;  std::vector<CompUnit>* comps;
    Decl decl;
    Def def;  std::vector<Def>* defs;
    FuncDef func_def;
    Expression expression;  std::vector<Expression>* expressions;
    Stmt stmt;
    Block block;
    InitVal initval;  std::vector<InitVal>* initvals;
    FuncFParam formal;   std::vector<FuncFParam>* formals;
    BlockItem block_item;   std::vector<BlockItem>* block_items;
```

```
}
```

类型的提前声明

这与文法中的 **属性** 相关，特别是每个非终结符的属性值。这些属性可以是不同类型的变量，用于存储解析过程中的信息，比如语法树节点的信息。

```
%token <symbol_token> STR_CONST IDENT
%token <float_token> FLOAT_CONST
%token <int_token> INT_CONST
%token LEQ GEQ EQ NE // <=   >=   ==   !=
%token AND OR // &&    ||
%token ADDASSIGN MULASSIGN SUBASSIGN DIVASSIGN MODASSIGN  //added
%token CONST IF ELSE WHILE NONE_TYPE INT FLOAT FOR
%token RETURN BREAK CONTINUE ERROR
```

这个相当于上下文无关文法的**终结符**

```
//give the type of nonterminals
%type <program> Program
%type <comp_unit> CompUnit
%type <comps> Comp_list
%type <decl> Decl VarDecl ConstDecl
%type <def> ConstDef VarDef
%type <defs> ConstDef_list VarDef_list
%type <func_def> FuncDef
%type <expression> Exp LOrExp AddExp MulExp RelExp EqExp LAndExp UnaryExp
PrimaryExp
%type <expression> ConstExp Lval FuncRParams Cond
%type <expression> IntConst FloatConst StringConst
%type <expression> Array_Dim_Number ConstArray_Dim_Number
%type <expressions> Array_Dim_Number_list ConstArray_Dim_Number_list Exp_list;
%type <stmt> Stmt
%type <block> Block
%type <block_item> BlockItem
%type <block_items> BlockItem_list
%type <initval> ConstInitVal VarInitVal
%type <initvals> VarInitVal_list ConstInitVal_list
%type <formal> FuncFParam
%type <formals> FuncFParams
```

这个是上下文无关文法中的**非终结符**

变量定义的数组维度可以出现0次或多次，变量声明decl的vardef可以出现一次或多次，但是不能0次。你在编写SysY_parser.y 时是怎么处理这种语法的。

变量定义的数组维度：

```
VarDef
:IDENT Array_Dim_Number_list '=' VarInitVal
{
    ($$) = new VarDef($1,$2,$4);
    ($$)->SetLineNumber(line_number);
}
|IDENT Array_Dim_Number_list
{
    ($$) = new VarDef_no_init($1,$2);
    ($$)->SetLineNumber(line_number);
}
|IDENT '=' VarInitVal
{
    ($$) = new VarDef($1,nullptr,$3);
    ($$)->SetLineNumber(line_number);
}
|IDENT
{
    ($$) = new VarDef_no_init($1,nullptr);
    ($$)->SetLineNumber(line_number);
}
;
```

维度为0的话，相当于用后两个文法。1或者多就用前两个，用list的定义

```
Array_Dim_Number_list
:Array_Dim_Number
{
    $$ = new std::vector<Expression>;
    ($$)->push_back($1);
}
|Array_Dim_Number_list Array_Dim_Number
{
    ($1)->push_back($2);
    $$ = $1;
}
;
```

list的定义相当于1或者多

变量声明decl的vardef:

```
VarDef_list
:VarDef
{
    ($$)= new std::vector<Def>;
    ($$)->push_back($1);
}
|VarDef_list ',' VarDef
{
    ($1)->push_back($3);
    ($$) = ($1);
}
;
```

必须使用一个vardef，前面还能加list，递归，所以最少是一次，最多无上限

请说明文法设计时为什么要使用addExp，mulExp，relExp 等一系列 Exp 并加上复杂的语法推导，而不直接使用一个Exp，然后 Exp→Exp('+' | '–' | '*' | '&&' | ...) Exp

导致运算顺序的错误

优先级的问题

举个例子 a+b*c，如果使用Exp的话，就变成了(a+b)*c

答辩提问

1.讲一讲你设计的FuncDel，增加了什么变量类型的文法？——在int类型的基础上增加了float类型和void类型

2.讲一讲你设计的文法，怎么实现的识别单独的分号语句——在stmt中定义了该文法

3.讲一讲sysytree的递归逻辑——这个简单