编译设计文档

编译过程简介

编译过程一般分为5个阶段：

1. 词法分析：识别单词及其类别，在本实验中主要分为无意义字符（包括空白字符、注释）、单分界符、双分界符、无符号整数、标识符、字符串常量。
2. 语法分析：根据语法规则识别语法成分，采用递归下降分析，需要消除左递归，通过向前看避免回溯。
3. 语义分析、生成中间代码：基于语法分析得出的抽象语法树进行语义分析，构造符号表，进行错误处理，生成 LLVM IR 中间代码便于优化处理。
4. 代码优化：待完成 进行机器无关优化。
5. 生成目标程序：待完成 同时进行机器相关优化。

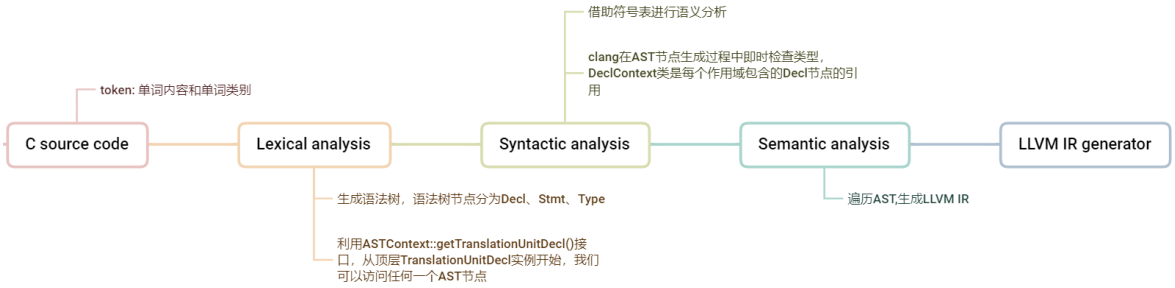
参考编译器

clang + LLVM

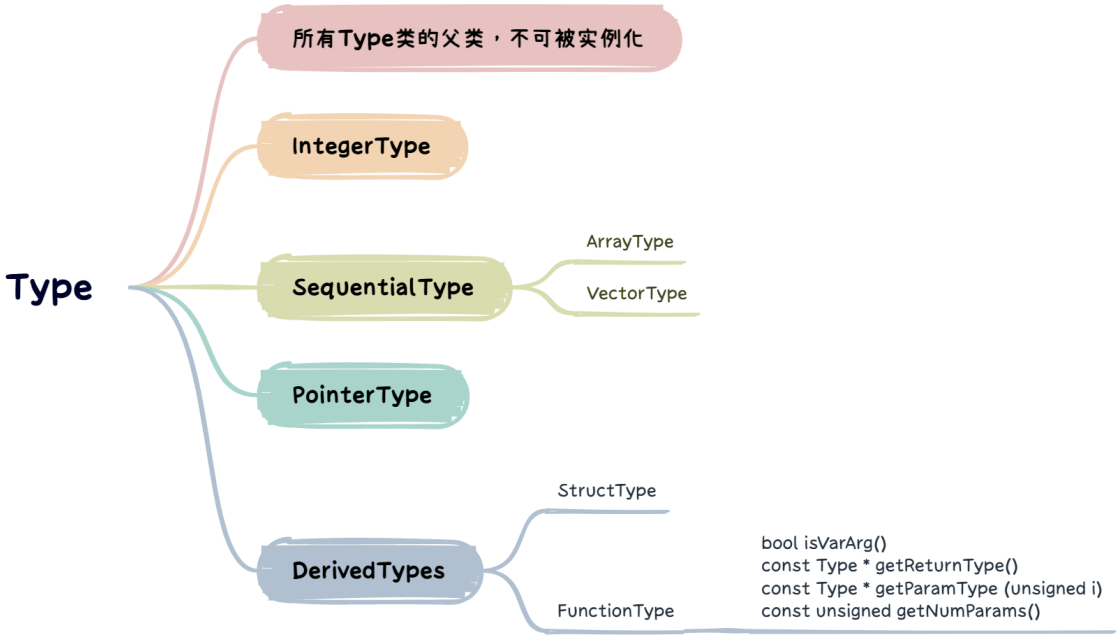
clang是将C语言转换成LLVM的前端。前端负责分析源代码，检查错误和把输入代码转换成**抽象语法树（Abstract Syntax Tree，AST）**。通过抽象语法树这种结构化表示，可以实现符号表处理、类型检查以及生成代码，因此AST是前端关注的重点之一。

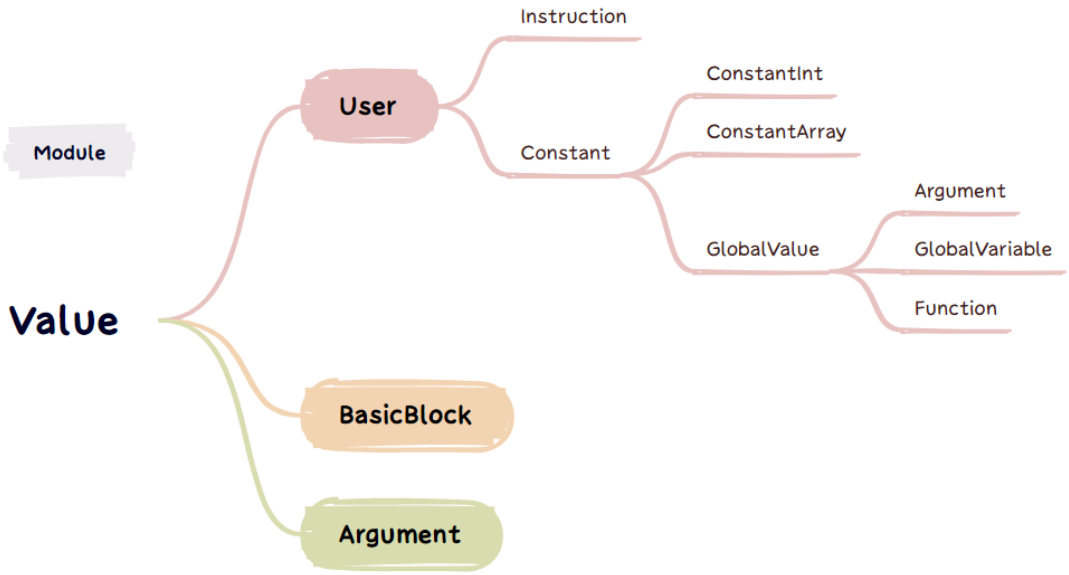
在clang中AST主要分为declaration、statement和type三类，但是它们并没有继承公共基类，每种节点都有特有的访问方式。我认为设计者主要考虑到这三类没有足够的共性，复杂的层次继承反而会降低AST设计的灵活性和可扩展性。

下为clang的流程图：



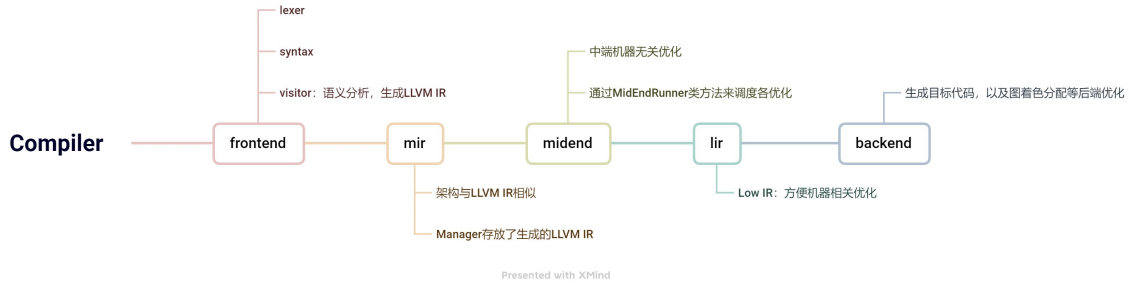
LLVM IR是一种基于静态单赋值（Static Single Assignment，SSA）的中间表示，我主要参考了LLVM IR的核心类，位于 `include/llvm/IR` 目录下：





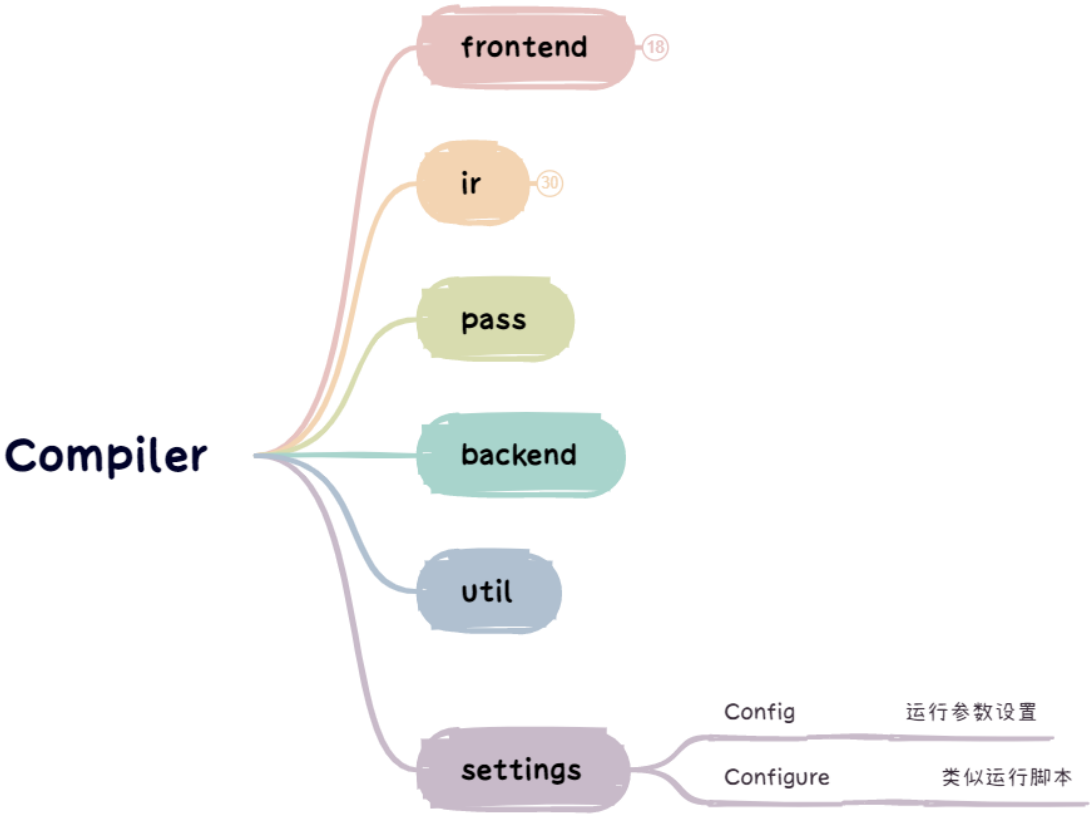
compiler2022-meowcompiler

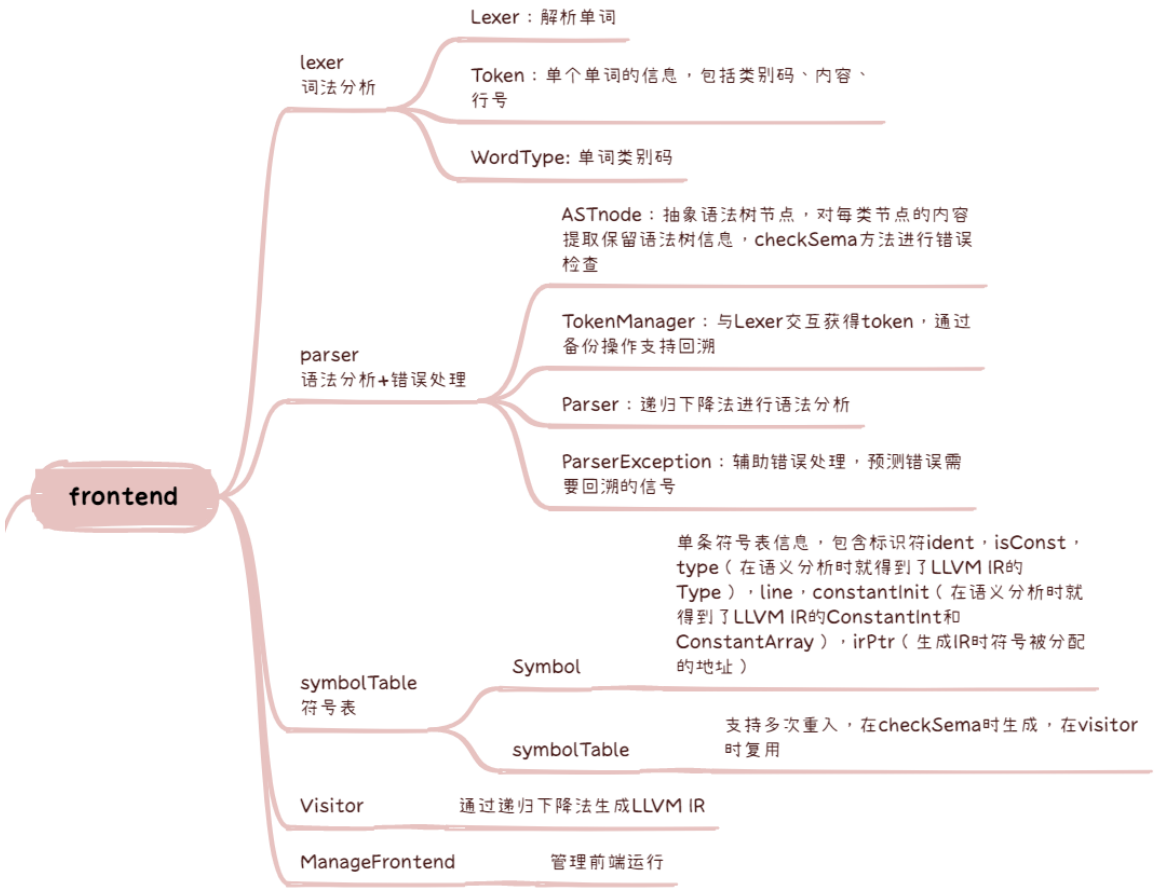
另外参考的是2022年参加编译比赛的一支队伍的compiler，架构如下：

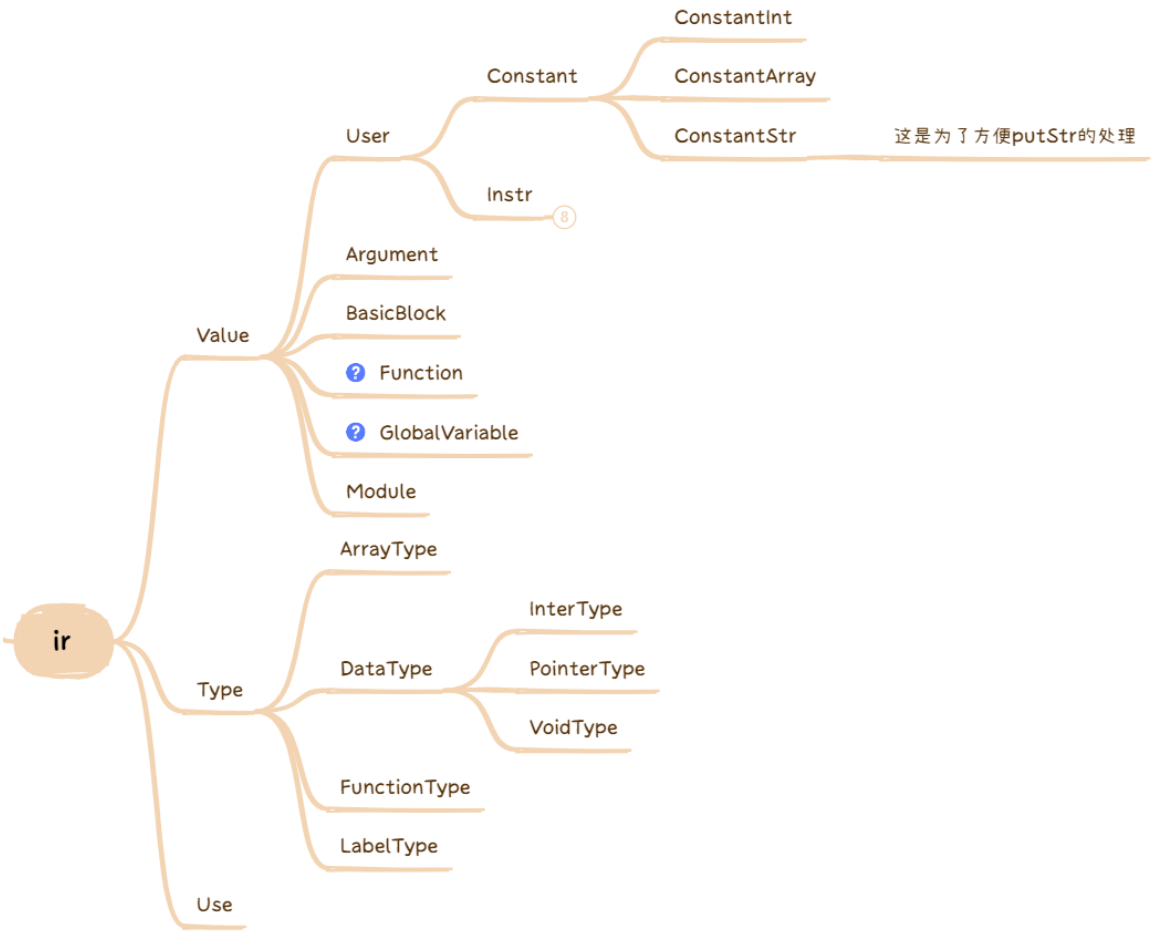



```
├─arg
├─backend # 生成目标代码arm
├─descriptor
├─exception
├─frontend
|  ├─lexer # 词法分析
|  ├─semantic # 语义分析工具（包括符号表等）
|  |  └─symbol
|  ├─visitor.java # 语义分析、生成中间代码
|  └─syntax # 语法分析
├─lir # low IR 进行与机器相关优化
├─manage
├─midend # 中端优化
├─mir # LLVM IR 架构
|  └─type
└─util # 工具
```

编译器架构（待完善）


Compiler
frontend
18
ir
30
pass
backend
util
settings
Config
运行参数设置
Configure
类似运行脚本


frontend
lexer
词法分析
Lexer：解析单词
Token：单个单词的信息，包括类别码、内容、行号
WordType: 单词类别码
parser
语法分析+错误处理
ASTnode：抽象语法树节点，对每类节点的内容提取保留语法树信息，checkSema方法进行错误检查
TokenManager：与Lexer交互获得token，通过备份操作支持回溯
Parser：递归下降法进行语法分析
ParserException：辅助错误处理，预测错误需要回溯的信号
symbolTable
符号表
Symbol
单条符号表信息，包含标识符ident，isConst，type（在语义分析时就得到了LLVM IR的Type），line，constantInit（在语义分析时就得到了LLVM IR的ConstantInt和ConstantArray），irPtr（生成IR时符号被分配的地址）
symbolTable
支持多次重入，在checkSema时生成，在visitor时复用
Visitor
通过递归下降法生成LLVM IR
ManageFrontend
管理前端运行



词法分析

词法分析阶段负责将源代码字符串分解成token。

词法分析一般有3种方式：

1. 有限状态自动机（Finite State Machine，FSM）：
2. 正则表达式：根据不同token的模式（关键字、标识符、运算符、数字）进行匹配从而识别token
3. 使用专门的词法分析工具：如Lex、ANTLR。

考虑到本课程实验采用Sysy语言，采用有限状态自动机实现。首先判断是不是注释，过滤掉注释后，先判断双分界符，再判断单分界符。

词法分析token是存入数据结构还是提供接口给语法分析，随取随用？=>后者

整体词法分析的思路同上，在编码细节处理上做了一些**修改**：

1. 为输入输出流写了一个单例模式进行封装，方便后续的读写，输入的时候采用按行读取，pos记录位置。
2. 将过滤注释和空白字符作为一个单独的方法，将有限状态自动机部分分成`ignoreBlank`和`nextToken`两个方法，对方法职责进行划分。
3. `token`分为双分界符、单分界符、无符号整数、标识符、字符串常量几类，首先进行正则表达式匹配将token划分到相应的类别，再做具体的判断。

bug记录

1. "\s"表示空字符可用于正则表达式匹配，但' '=='\s'是false。
2. formalString 的双引号内容需要进行非贪婪匹配，即 `"\".*\"` -> `"\".*?\"`

语法分析

1. 改写文法

带回溯的自顶向下分析方法，试图构造最左推导序列，然而不能处理左递归文法，同时回溯会影响效率。

对课程组提供的SysY文法进行改写，**取消左递归**，以 `AddExp` 推导为例：

```
addExp -> addExp ("+" | "-") mulExp | mulExp
改为：
addExp -> mulExp (("+" | "-") mulExp)*
```

同时语法成分输出也要做相应处理。

为了**减少回溯**，我们需要尽量保证每个非终结符的First集合不相交，这样就可以通过判断First集合来确定使用哪个产生式。然而观察文法，存在First集合相交的情况，这时需要观察相应文法的特殊性，往后继续取标志性单词观察。

2. 构造语法树节点

一开始我仅用一个 `AstNode` 类存储所有语法成分，即：

```
public class AstNode {
    private AstNodeType type; // 语法树节点类型
    private ArrayList<AstNode> nodes; // 该节点包含的子节点
    // ...
}
```

尽管在语法分析作业中这样写并没有问题，然而所有语法树节点以同一方式存储，AstNode的子节点都视为相同的元素，会丢失语法分析的一部分结果，例如：

```
语句 Stmt → LVal '=' Exp ';'
| [Exp] ';'
| Block
| 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
| 'for' '(' [ForStmt] ';' [Cond] ';' [forStmt] ')' Stmt
| 'break' ';' | 'continue' ';'
| 'return' [Exp] ';'
| LVal '=' 'getint''('')'';'
| 'printf''('FormatString{','Exp}')'';'
```

对于 `stmt` 节点，在获取子节点元素时，我们就需要手写判断语句类型，靠文法计算需要获取的子节点的下标，有时候还需要向前查看（遇到可省略语法成分时），几乎相当于又做了一遍语法分析。

因此，为了最大化保留语法分析成果，为每种语法成分建一个类，stmt也细分，每个类里个性化存储子节点，例如对于 `stmt` 中的 `for` 语句：

```
public class StmtFor extends Stmt {
    private ForStmt forStmt1 = null;
    private Cond cond = null;
    private  ForStmt forStmt2 = null;
    private Stmt stmt;
}
```

3. TokenManager - 将语法分析和与lexer交互解耦

将token的相关操作单独形成一类，与语法分析解耦，符合单一职责原则。另外，在我的设计中词法分析和语法分析一起是一遍，所以在语法分析中实现 `TokenManager` 类来进行token管理，包括控制Lexer读取token、向前查看、获取下一个token。

4. 递归下降进行语法分析

语法分析采用递归下降法，为每个非终结符构造分析函数，用向前看符号指导产生式规则。例如，对于 `CompUnit`：

```
/*
* CompUnit -> (Decl | FuncDef)* MainFuncDef
*/
public CompUnit parseCompUnit() throws ParserException {
    /* Decl */
    CompUnit compUnit = new CompUnit();
    while(!tokenManager.checkTokenType(2,WordType.LPARENT)) {
        compUnit.addDecl(parseDecl());
    }
    /* FuncDef */
    while(!tokenManager.checkTokenType(1, WordType.MAINTK)) {
        compUnit.addFuncDef(parseFuncDef());
    }
    /* MainFuncDef */
    if (tokenManager.checkTokenType(1, WordType.MAINTK)) {
        compUnit.addFuncDef(parseMainFuncDef());
    }
    LexParseLog.add(compUnit.toString());
    return compUnit;
}
```

语义分析、错误处理

符号表

SysY是分程序结构的语言，在符号表上需执行**插入**、**查表**、**定位**和**重定位**操作

- 查表是在程序的可执行语句部分读到标识符时，需要判断该标识符是否已经声明：当前所在程序单元 -> **直接外层**符号表
- 定位是在分程序的入口建立一个新的子表
- 重定位是在分程序的出口删除已处理完的分程序的标识符的子表
- 定位与重定位操作的工作方式和栈的压入和弹出操作很像。

1. 符号表存储的信息：

ident	**标识符**
isConst	是否用const限定词修饰
isGlobal	是否为全局变量
type	此处直接处理成LLVM IR中的type，用于类型检查

ident	标识符
line	符号所在行
constantInit	常量初值（若有），此处直接处理成LLVM中的Constant
irPtr	用于记录IR中符号地址（生成代码时用到）

3. 符号表的组织方式
    - 为了便于查找Symbol，我使用`HashMap<String, Symbol>`存储
    - 为了支持符号表向外层查找操作，需要记录符号表的父表`parent`
    - 该符号表在生成中间代码时需要复用，需要支持从当前符号表依次遍历子符号表，故记录子符号表数组`childTables`
4. 符号表的操作
    - 定位

      在进入分程序的入口时，创建一个新的子表，将其`parent`设置为当前符号表，将当前符号表设置为新的子表。（初始表的`parent`是`null`）

      ```
      public void locate() {
          currentTable = new SymbolTable(currentTable);
      }
      ```

    - 查表分为声明时的查表和使用时的查表
        1. 声明时仅需查找**当前层符号表**
        2. 使用时需**递归查找**当前层及直接外层符号表
    - 插入

      将Symbol插入当前层符号表
    - 重定位

      在分程序的出口时，将当前符号表设置为其`parent`。

      ```
      public void relocate() {
          currentTable = currentTable.parent;
      }
      ```

语义分析

在**语义分析阶段**，建立符号表以供查询符号表相关错误。建立一张单例模式的错误信息表`errorLog`。为了方便调用语法树各节点内方法，将`checkSema`方法分散到语法树节点类实现。

非法符号a

在词法分析的过程中，如果创建的`token`是`STRCON`类型，通过正则表达式匹配是否存在非法字符，以及`\`后不是`d`或`n`的情况。

符号表相关错误b, c, d, e, h

错误类别码	错误类型	解决思路
b	名字重定义（函数名和变量名重定义）	在当前作用域内查找
c	可执行语句使用未定义的名字	不断往直接外层查找
d	函数参数个数不匹配	Symbol记录的FunctionType的ArgumentTypes
e	函数参数类型不匹配	Symbol记录的FunctionType的ArgumentTypes
h	不能改变常量的值	Symbol记录的isConst
f	无返回值的函数存在不匹配的return语句	在`FuncDef`内进行处理
g	有返回值的函数缺少return语句（函数末尾）	

在语法分析阶段结束后，处理出符号表。为了处理出常量，如数组维度、全局变量初值、constDecl初值，需要为所有表达式及元素实现`getOpResult`方法，以及在节点与子节点之间需要传递`isGlobal`等信息。

缺少符号 i, j, k

错误类别码	错误类型
i	缺少分号
j	缺少右小括号
k	缺少又中括号

此处的难点是在语法分析的过程中有向前看的操作，若丢失关键符号，会导致进入分支错误。缺少右小括号不必担心这种情况，只需在需要右小括号但不存在的时候产生错误信息并新建一个分号token放入语法树。`tokenManager.getNextToken()`方法调用时要和所需要的类型进行匹配，否则报错。下面以缺少分号为例：

```
stmt -> 'return' [exp] ';'
stmt -> Lval '=' 'getint' '(' ')' ';' | Lval '=' Exp ';' | [Exp] ';'
```

在正常情况下，第一句中分号判断是否有exp；第二句中如果在分号前出现了等号则stmt为前两种情况。那么，

1. 在依靠分号进行分支选择的时候，先假设程序正确来分析，保存当前状态；如果进入的分支无法匹配文法，抛出异常；在原分支选择处接收异常，并进行回溯。
2. 为了能完成回溯动作，`tokenManager`新增`backupBuffer`等备份属性，在尝试匹配的时候打开`isBackup`开关，进行备份。
3. 在我的架构中，在存语法树时，我已经舍弃了一些非必要的token，例如分号、括号等，故我还需要补充一些行号记录。

其他错误 f, g, l, m

错误类别码	错误类型	解决思路
f	无返回值的函数存在不匹配的return语句	从 `FuncDef` 不断往下传 `funcType`，遇到 `return` 语句就和 `funcType` 比对。
g	有返回值的函数缺少return语句（函数末尾）	判断 `block` 的最后一个 `blockIterm` 是否为 `return` 语句。
l	printf中格式字符与表达式个数不匹配	仅需判断`stmtPrintf.getExps().getSize` 是否等于 `formatString`中 `%d` 的个数。
m	在非循环块中使用break和continue语句	`for` 设置其 `stmt` 循环块内的 `isInLoop` 均为 `true`。

生成中间代码LLVM IR

void function 行末补充return

MIPS目标代码生成