

# 编译设计文档

---

## 总设计

---

采用解耦的设计，编译器在主函数分别新建一个词法分析器Lexer、一个语法分析器Parser、错误类Errors等等各子模块，再分别调用。其中主函数会将源代码转换为一个ArrayList<String>作为lexer的输入，词法分析完成后将结果再传输至parser，进行语法分析。错误处理分布在词法分析、语法分析和生成AST中，由Errors控制管理和输出。之后遍历AST输出中间代码，我完成了pcode和mips两个后续模块，因此中间代码可直接在pcode运行器中运行并输出结果，同时输入至mips模块生成对应的mips代码。

```
Lexer lexer = new Lexer(sourceCode);
lexer.beginLexer();
//lexer.printLexer();

Parser parser = new Parser(lexer.getwords());
parser.parsing();
//parser.printParser();

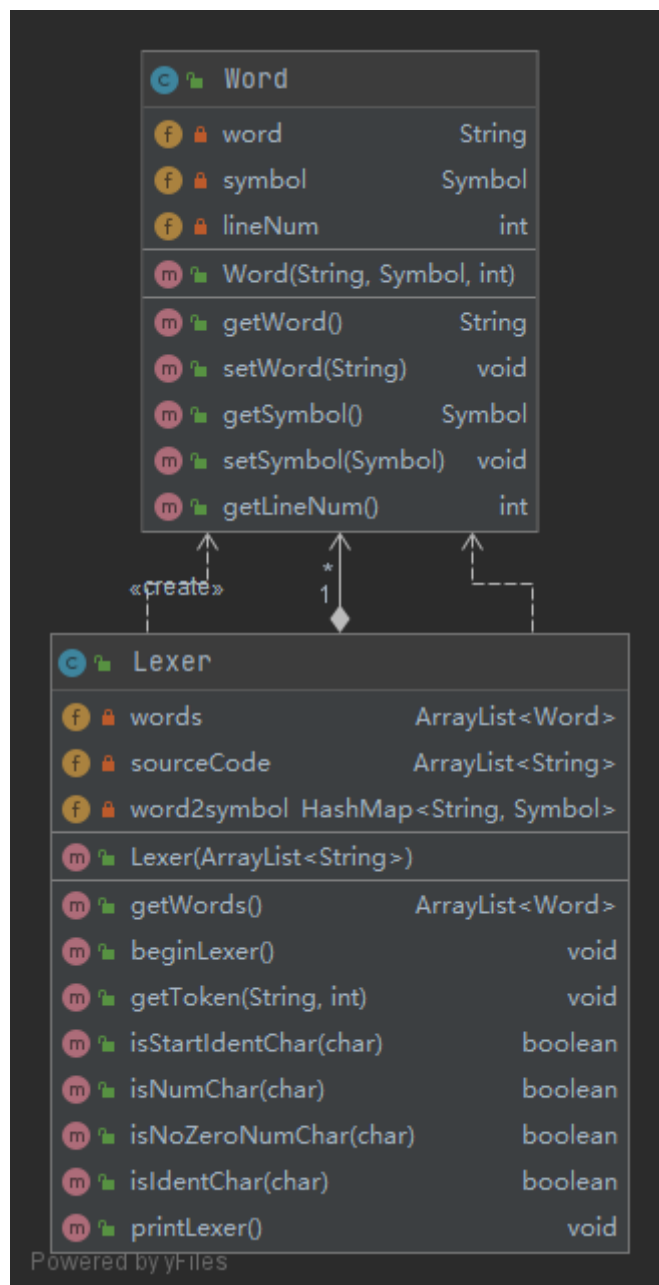
Errors.printErrors();
```

## 词法分析

---

Lexer每分析得到一个单词即创建一个其对应的Word，并将其保存至一个ArrayList用于后续语法分析。另外Symbol为所有类别码的枚举类。

由于只有三种（Ident、IntConst、FormatString）类别的单词字符串是不定的，其他的均可建立 单词名称 --> 类别码 的唯一映射，如 main --> MAINTK ``+ --> PLUS，因此建立HashMap，在读入一个单词后判断其是否为HashMap的一个键，若是则可直接建立相应的Word，否则进行其他三种类型的区分后再建立相应的Word。最后将所有的Word存入一个ArrayList，词法分析至此结束。



## Word

### 属性

- `String word`: 该单词对应的字符串
- `Symbol symbol`: 该单词对应的类别码
- `int lineNum`: 该单词的行号

### 主要函数

- `word(String word, Symbol symbol, int lineNum)`: 构造函数

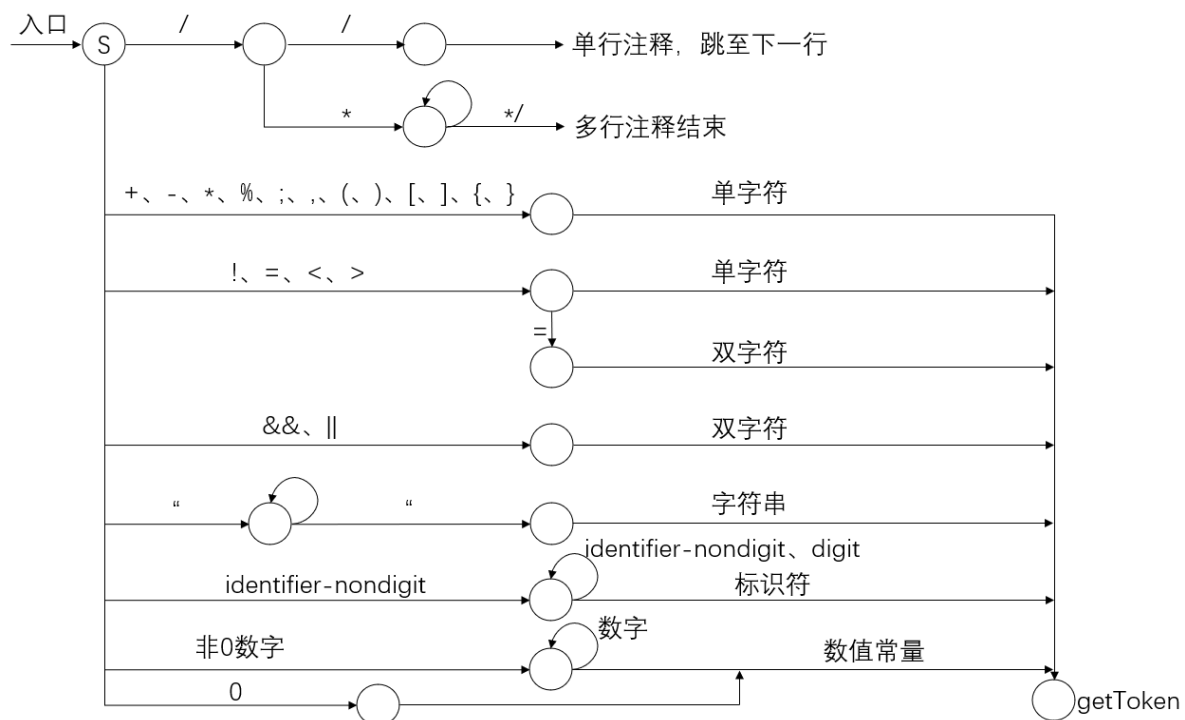
## Lexer

### 属性

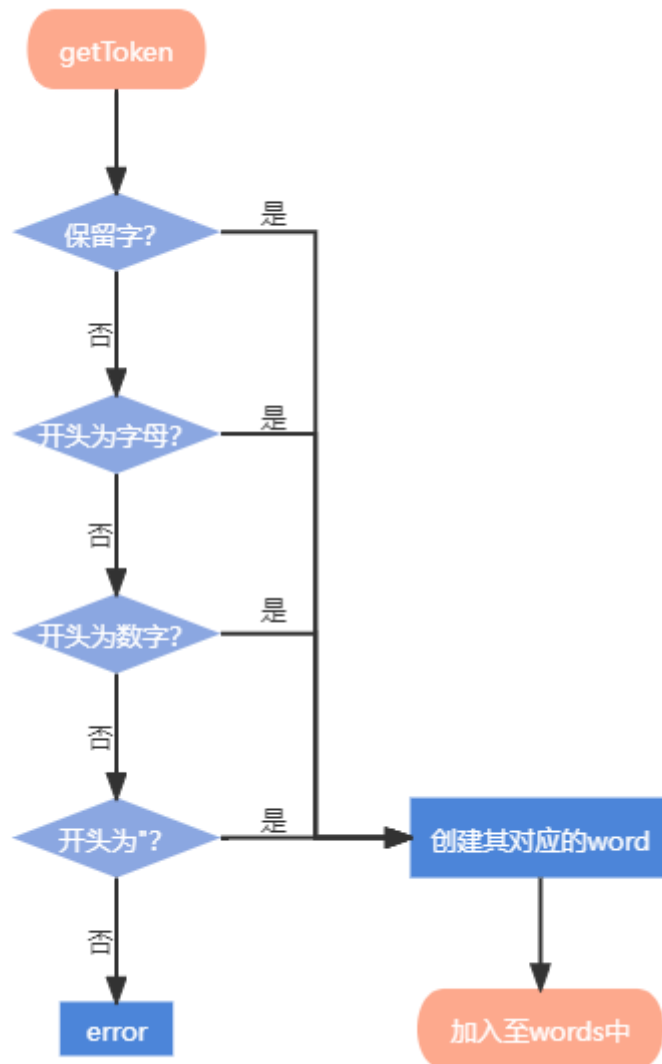
- `ArrayList<word> words`: 储存经过语法分析后得到的单词，用于后续的语法分析
- `ArrayList<String> sourceCode`: 源代码
- `HashMap<String, Symbol> word2symbol`: 初始化保留字与类别码的映射关系

## 函数

- `void beginLexer()`：词法分析函数。利用如下状态图进行，每次完成一个单词的读取后调用 `getToken` 方法。



- `void getToken(String token, int lineNum)`：将读入的单词创建一个 `word`，并存入 `ArrayList words`。具体流程为：

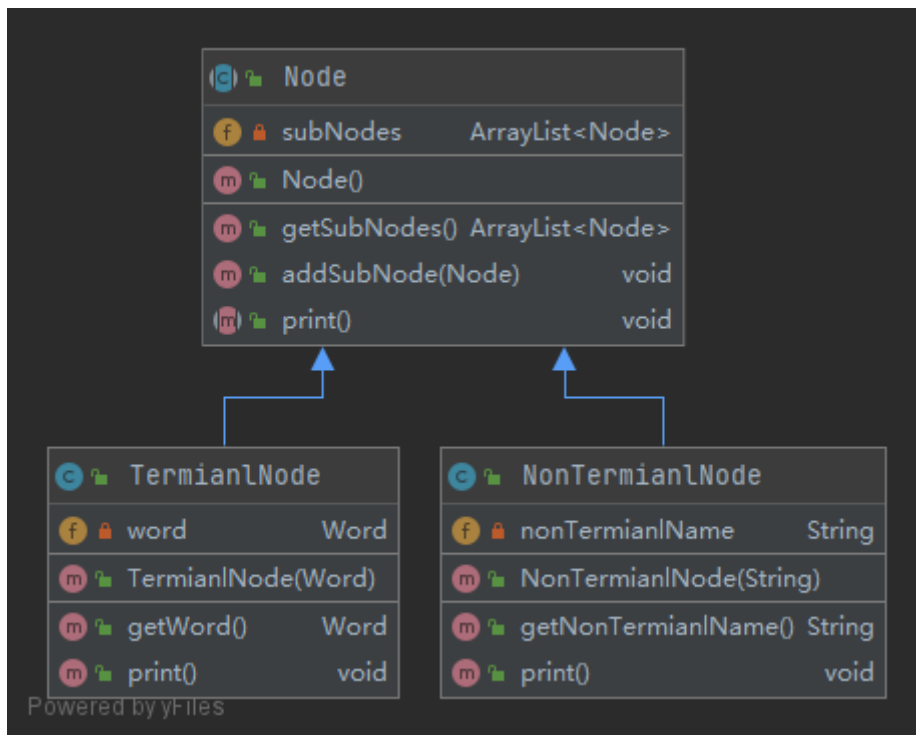


- `boolean isStartIdentChar(char c)`: 判断是否为identifier-nondigit
- `boolean isIdentChar(char c)`: 判断是否为identifier-nondigit或digit
- `boolean isNumChar(char c)`: 判断是否为digit
- `boolean isNoZeroNumChar(char c)`: 判断是否为nonzero-digit
- `void printLexer()`: 按照格式将所有词法分析得到的word输出。

## 语法分析

parser读入词法分析得到的单词，利用递归下降构建一颗CST（具体语法树），递归下降时采用向前偷看的方法解决可能出现的冲突。

## Node、TerminalNode、NonTerminalNode



如图所示，TerminalNode、NonTerminalNode继承Node，分别代表终结符节点、非终结符节点。Node包含了一个 `ArrayList<Node> subNodes` 作为其子节点。其中非终结符节点属性包含该非终结符的名字，终结符节点属性包含该终结符的信息Word。

Node有抽象方法print用于语法分析的输出。对于非终结符，需要先将其所有子节点输出后，再输出其名字；对于终结符，直接输出其对应的终结符信息。因此输出时只需要调用根节点的print方法，即可遍历整个树输出语法分析答案。

## Parser

Parser		
f	root	NonTermianlNode
f	astRoot	CompUnitNode
f	curSym	int
f	words	ArrayList<Word>
Parser(ArrayList<Word>)		
m	getRoot()	NonTermianlNode
m	printParser()	void
m	getSym(int)	Symbol
m	parsing()	void
m	FuncDef()	Node
m	FuncFParams()	Node
m	FuncFParam()	Node
m	Block()	Node
m	BlockItem()	Node
m	Stmt()	Node
m	Cond()	Node
m	LOrExp()	Node
m	LAndExp()	Node
m	EqExp()	Node
m	RelExp()	Node
m	FuncType()	Node
m	VarDecl()	Node
m	VarDef()	Node
m	InitVal()	Node
m	MainFuncDef()	Node
m	ConstDecl()	Node
m	ConstDef()	Node
m	ConstExp()	Node
m	AddExp()	Node
m	MulExp()	Node
m	UnaryExp()	Node
m	FuncRParams()	Node
m	PrimaryExp()	Node
m	UnaryOp()	Node
m	Exp()	Node
m	LVal()	Node
m	Number()	Node
m	ConstInitVal()	Node
m	error()	void
m	error(int, String)	void

## 属性

- `NonTerminalNode root`: CST的根节点
- `CompUnitNode astRoot`: AST的根节点 (在错误处理中建立, 因此在错误处理中叙述)
- `int curSym`: 语法分析当前处理的单词的序号
- `ArrayList<word> words`: 由词法分析得到的单词列表

## 函数

- `void parsing()`: 开始语法分析。新建一个根节点名为`CompUnit`, 之后开始递归下降, 由于所有的递归下降子程序都会返回一个`Node`, 因此只需要将子程序的返回值加入至当前的根节点的子节点中即可。如在`CompUnit`中加入一个常数声明子节点: `root.addSubNode(ConstDecl());`

完整的parsing如下:

```
root = new NonTerminalNode("CompUnit");
curSym = 0;
while (getSym(curSym) == Symbol.CONSTTK || getSym(curSym) ==
Symbol.INTTK || getSym(curSym) == Symbol.VOIDTK) {
    if (getSym(curSym) == Symbol.CONSTTK) {
        root.addSubNode(ConstDecl());
    }
    else if (getSym(curSym) == Symbol.INTTK && getSym(curSym+1) == Symbol.MAINTK)
    {
        root.addSubNode(MainFuncDef());
    }
    else if (getSym(curSym) == Symbol.INTTK && getSym(curSym+2) != Symbol.LPARENT)
    {
        root.addSubNode(VarDecl());
    }
    else root.addSubNode(FuncDef());
}
```

- 其他递归子程序: 每一个非终结符均有一个自己的递归子程序, 其返回代表自己的节点。在此仅挑选一般的和几类比较特殊的作详细说明。
  - 一般的子程序: 如 `LVal → Ident '[' Exp '']`, 进入子程序后首先创建一个名为`LVal`的非终结符`NonTerminalNode`, 之后按文法, 若应该读入一个终结符, 如第一个`Ident`, 则直接将该终结符加入至子节点中; 若应该读入一个非终结符, 如`Exp`, 则调用对应的子程序`Exp()`, 并加入至子节点中。

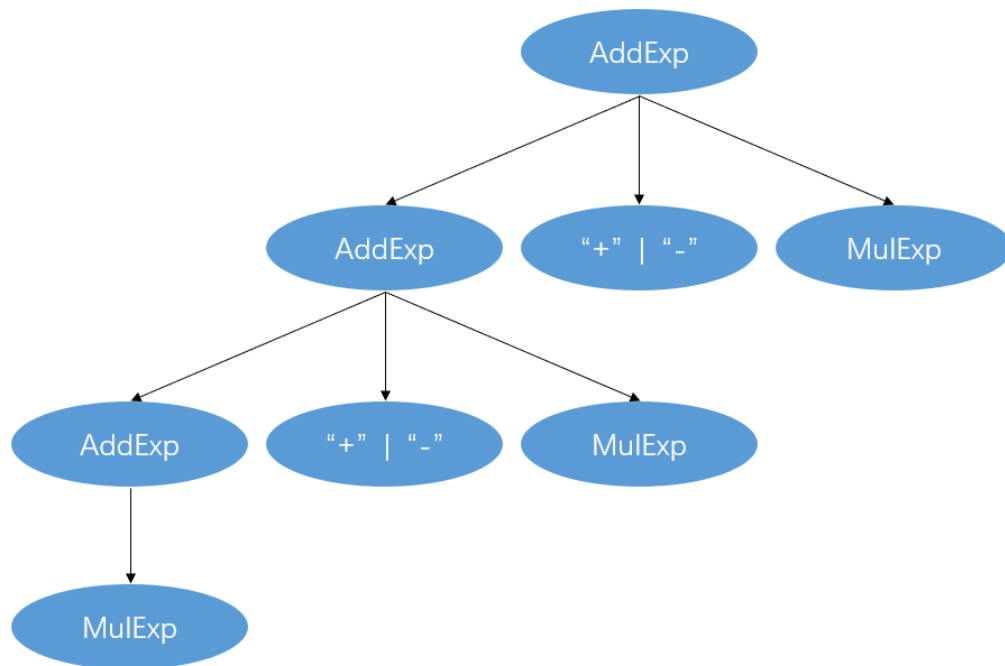
```
public Node LVal() {
    NonTerminalNode node = new NonTerminalNode("LVal");
    if (getSym(curSym) == Symbol.IDENFR) {
        node.addSubNode(new TerminalNode(words.get(curSym)));
        curSym++;
        while (getSym(curSym) == Symbol.LBRACK) {
            node.addSubNode(new TerminalNode(words.get(curSym)));
            curSym++;
            node.addSubNode(Exp());
            if (getSym(curSym) == Symbol.RBRACK) {
                node.addSubNode(new TerminalNode(words.get(curSym)));
                curSym++;
            }
        }
        else error(words.get(curSym-1).getLineNum(), "k");
    }
```

```

    }
  }
  else {
    error();
    return null;
  }
  return node;
}

```

- 需更新根节点的。AddExp  $\rightarrow$  MulExp | AddExp ('+' | '-') MulExp：假设有表达式中有俩个“+” | “-”，将其化为语法树形式即为：



可以发现原本的第一个MulExp实际上并不是我们的根节点，因此我们需要每发现一个“+” | “-”，都需要将根节点进行更新。代码如下：

```

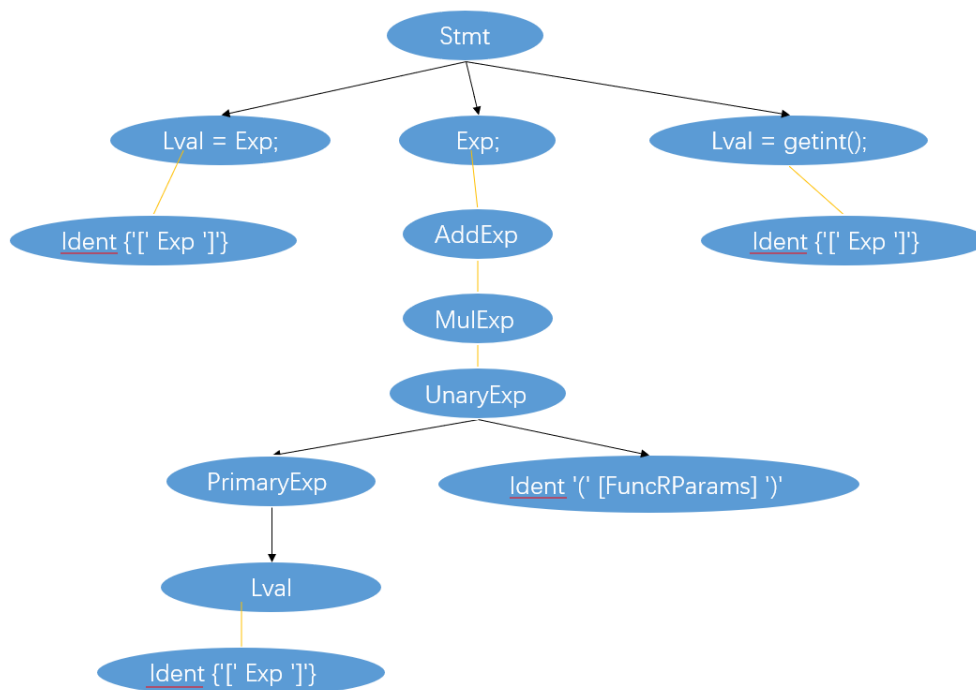
public Node AddExp(){
    NonTerminalNode node = new NonTerminalNode("AddExp");
    Node root = node;
    node.addSubNode(MulExp());
    while (getSym(cursym) == Symbol.PLUS || getSym(cursym) == Symbol.MINUS){
        node = new NonTerminalNode("AddExp");
        node.addSubNode(root); //加入老的root至最新的root中
        node.addSubNode(new TerminalNode(words.get(cursym)));
        cursym++;
        node.addSubNode(MulExp());
        root = node; //更新root为最顶上的node
    }
    if (root.getSubNodes().size()==0){
        return null;
    }
    return root;
}

```

实际上有类似的结构文法均可这样完成，如 MulExp  $\rightarrow$  UnaryExp | MulExp ('\*' | '/' | '%') UnaryExp 等等。



- `Stmt`：唯一涉及到回溯的地方。将stmt子程序发现首个单词为Ident的情况进行一个总结如下：



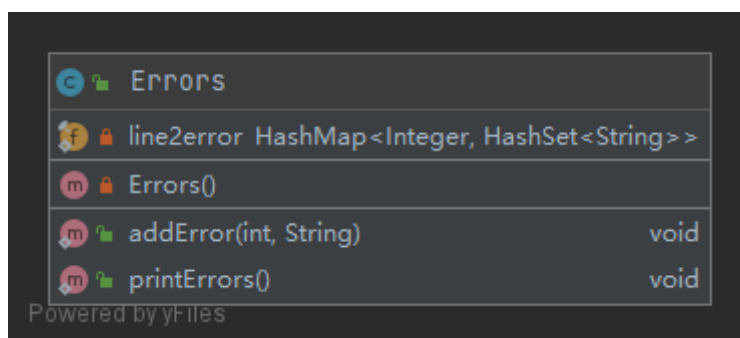
可以看到有四种情况均是以Ident起始。往前看一次，只能将 `Ident '(' [FuncRParams] ')'` 与其他三种情况分开，至于另外三种情况，可发现其均为Lval，若直接进行Lval的子程序，将导致 `Stmt -> Exp; -> AddExp -> MulExp -> UnaryExp -> PrimaryExp -> Lval` 的情况原本为Exp为根节点变为了以Lval为根节点，因此先进行Lval的预读后，若发现是上述情况，则需要回溯至读Lval之前的状态，再新建一个Exp节点，最终也将进入Lval的子程序，但是这样的根节点便是Exp了。其他两种情况可在预读Lval后判断得出，也无需回溯。

- `Symbol getSym(int index)`：获取index对应的单词的类别码，若index超出了范围，即已处理完所有单词，返回EOF。
- `void printParser()`：输出语法分析答案。只需调用 `root.print()` 即可。
- `void error()`、`void error(int line,String errorStr)`：为后续的错误处理预留的接口。

## 错误处理

为了将各部分解耦，错误处理将直接遍历语法分析得到的CST，而考虑到后续作业生成代码，CST包含了过多无用信息如`;`，且CST并没有对每一个非终结符的特征进行提取，而是将所有的子节点无差别地放入一个ArrayList中，因此在进行错误处理时，我将同时将CST转换为AST，实现信息的提取，为后续作准备。

新建了一个错误类Errors，



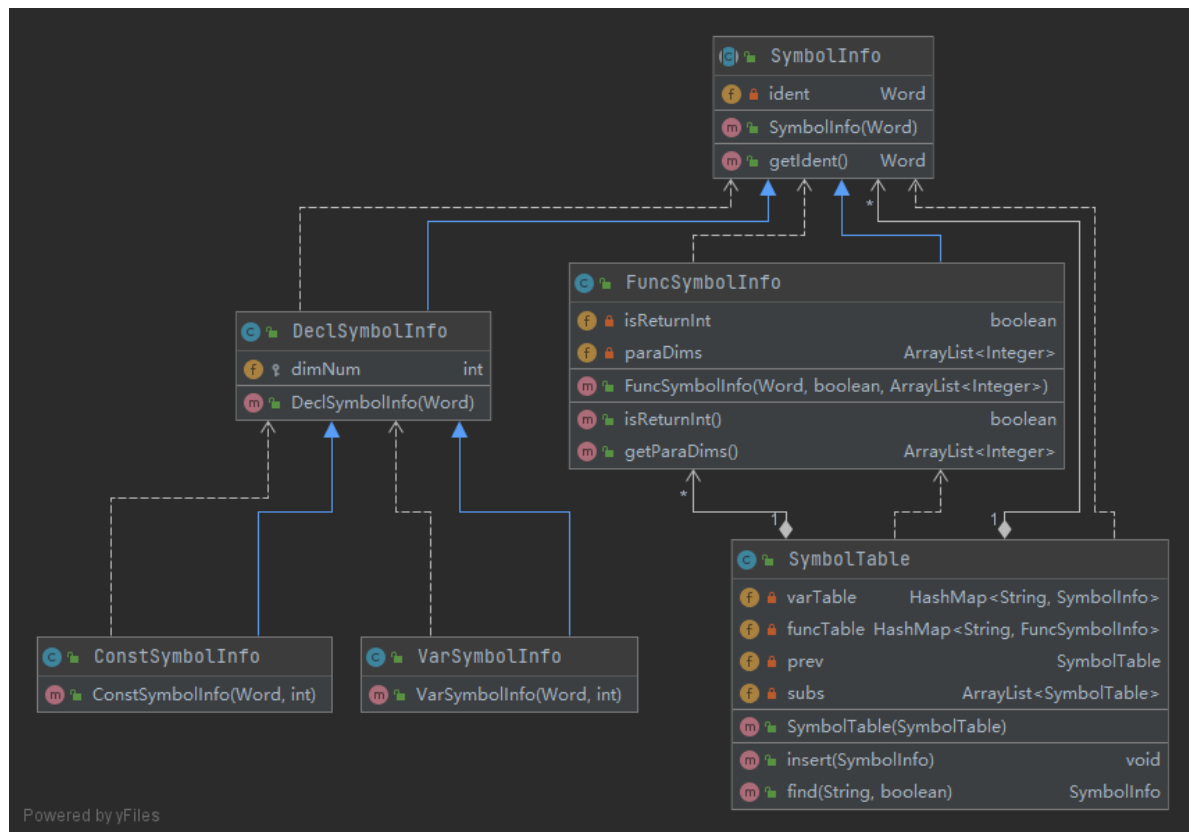
后续检测到错误时只需 `addError()` 即可。

错误处理可以分为三类：词法分析即可分析的、语法分析即可分析的、遍历CST才可分析的。

首先处理词法分析即可分析的，只有a类。这只需在Lexer中进行判断FormatString是否合法。

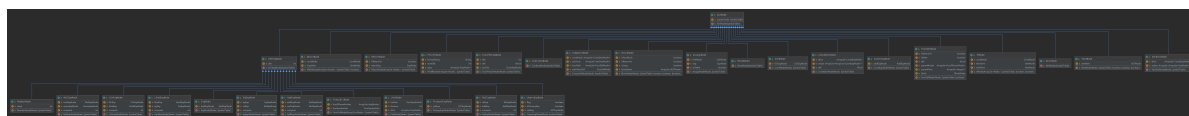
对于语法分析即可分析的，有ijk类。由于在语法分析中已经预留好了接口，只需在发现缺少`;`、`)`、`[]`时进入的error()函数传入对应的参数，即可完成ijk类的错误处理。下面主要详细说明CST的遍历和AST的生成以及这其中的错误处理。

## 符号表说明



如图所示，分为了三类信息：常数声明、变量声明、函数声明，其中常数声明与变量声明先继承DeclSymbolInfo，三类信息均继承SymbolInfo，对于不同类别给予相应的属性。由于函数和变量名可以重名，SymbolTable中分别保存函数和变量的符号信息，均由HashMap实现。SymbolTable在创建时传入其父亲SymbolTable作为prev，由此实现了符号表。

## AST结构说明



- **ASTNode**：所有的节点类均直接或间接继承ASTNode
  - **SymbolTable symbolTable**：ASTNode含有一个SymbolTable代表当前的符号表
- **ASTExpNode**：继承ASTNode，作为各种表达式节点的父类，加入了表达式维度信息
  - **int dim**：该表达式的维度
- **AddExpNode**：继承ASTExpNode，对应AddExp → MulExp | AddExp ('+' | '-') MulExp。
  - **AddExpNode addExpNode**
  - **MulExpNode mulExpNode**
  - **int compute**：0表示无addExp，1表示+，2表示-
- **MulExpNode**：继承ASTExpNode，对应MulExp → UnaryExp | MulExp ('\*' | '/' | '%') UnaryExp。

- `MulExpNode mulExpNode`
  - `UnaryExpNode unaryExpNode`
  - `int compute`: 0表示无mulExp, 1表示\*, 2表示/, 3表示%
- `UnaryExpNode`: 继承`ASTExpNode`, 对应`UnaryExp → PrimaryExp | Ident '(' [FuncRParams] ')' | UnaryOp UnaryExp`
  - `boolean flag`: 将所有UnaryOp进行合并后得到的唯一UnaryOp, true表示为正, false表示为负
  - `boolean isPrimaryExp`: 子节点是否为PrimaryExp
  - `ASTExpNode subExp`: 子节点, PrimaryExp或函数调用
- `FuncCallNode`: 继承`ASTExpNode`, 对应`Ident '(' [FuncRParams] ')'`, 为函数调用。
  - `ArrayList<ExpNode> funcRParamNodes`: 所有实参列表
  - `FuncSymbolInfo funcSymbolInfo`: 函数声明信息
- `EqExpNode`: 继承`ASTExpNode`, 对应`RelExp | EqExp ('==' | '!=') RelExp`
  - `EqExpNode eqExp`
  - `RelExpNode relExp`
  - `int compute`: 0表示无addExp, 1表示==, 2表示!=
- `PrimaryExpNode`: 继承`ASTExpNode`, 对应`PrimaryExp → '(' Exp ')' | LVal | Number`
  - `ASTExpNode subExp`: 下一级对应的节点
- `ExpNode`: 继承`ASTExpNode`, 对应`Exp → AddExp`
  - `AddExpNode addExpNode`
- `LAndExpNode`: 继承`ASTExpNode`, 对应`LAndExp → EqExp | LAndExp '&&' EqExp`
  - `LAndExpNode lAndExp`
  - `EqExpNode eqExp`
  - `int compute`: 0表示无addExp, 1表示&&
- `LOrExpNode`: 继承`ASTExpNode`, 对应`LOrExp → LAndExp | LOrExp '|' LAndExp`
  - `LOrExpNode lOrExp`
  - `LAndExpNode lAndExp`
  - `int compute`: 0表示无addExp, 1表示|
- `LValNode`: 继承`ASTExpNode`, 对应`LVal → Ident '[' Exp '']`
  - `DeclSymbolInfo lvalInfo`: 变量声明信息
  - `ArrayList<ExpNode> dims = new ArrayList<>()`: 维度信息
- `NumberNode`: 继承`ASTExpNode`, 对应`Number → IntConst`
  - `int value`: 对应的数值
- `CompUnitNode`
  - ```
private ArrayList<ConstDeclNode> constDecls = new ArrayList<>();
private ArrayList<VarDeclNode> varDecls = new ArrayList<>();
private ArrayList<FuncDefNode> funcDefs = new ArrayList<>();
private FuncDefNode mainFuncDef;
```
- `ConstDeclNode`: 继承`ASTNode`, 对应

```
ConstDecl → 'const' BType ConstDef { ',' ConstDef } ';'
ConstDef → Ident { '[' ConstExp ']' } '=' ConstInitVal
ConstInitVal → ConstExp | '{' [ ConstInitVal { ',' ConstInitVal } ] '}'
```

改写了为每一个ConstDeclNode只对应一个声明ConstDef, 出现多个逗号会将新建一个ConstDeclNode。

- `VarDeclNode`: 继承ASTNode, 对应

```
VarDecl → BType VarDef { ',' VarDef } ';'
VarDef → Ident { '[' ConstExp ']' } | Ident { '[' ConstExp ']' } '='
InitVal
InitVal → Exp | '{' [ InitVal { ',' InitVal } ] '}'
```

改写了为每一个VarDeclNode只对应一个声明ConstDef, 出现多个逗号会将新建一个ConstDeclNode。

- `ArrayList<ConstExpNode> dims`
- `ArrayList<ArrayList<ExpNode>> values`
- `word def`
- `FuncDefNode`:
  - `boolean isReturnInt;`  
`boolean isMain;`  
`word def;`  
`ArrayList<FuncFParamNode> paramNodes=new ArrayList<>();`  
`ArrayList<Integer> paramDims=new ArrayList<>();`  
`BlockNode block;`
- `FuncFParamNode`:
  - `int dim=0;`  
`word def;`  
`ConstExpNode secDim;`
- `BlockNode`
  - `boolean isFuncBlock;`  
`boolean isReturnInt;`  
`boolean isLoop;`  
`ArrayList<ASTNode> blockItems;`
- `StmtNode`
  - `ASTNode stmtItem`
- `IfNode`
  - `CondNode condNode;`  
`StmtNode thenStmt;`  
`StmtNode elseStmt;`
- `AssignNode`
  - `LValNode lValNode;`  
`ExpNode exp=null;`  
`boolean isGetInt;`
- `whileNode`
  - `CondNode condNode;`  
`StmtNode loopStmt;`

- **PrintfNode**

- String formatString;  
int numOfd;  
ArrayList<ExpNode> exps;

- **ReturnNode**

- boolean isReturnInt;  
ExpNode returnExp;

## AST的构建

将CST的相应节点作为AST节点的构造函数的参数，即可得到AST，如CompUnitNode，传入CST中对应的CompUnit的非终结符节点，之后对该根节点的子节点进行遍历，生成AST。

```
public CompUnitNode(Node root, SymbolTable symbolTable){
    super(symbolTable);
    for (Node node : root.getSubNodes()){
        if (((NonTermianlNode) node).getNonTermianlName().equals("ConstDecl")){
            for (int i = 2; i < node.getSubNodes().size(); i += 2){
                constDecls.add(new
ConstDeclNode(node.getSubNodes().get(i), symbolTable));
            }
        }
        else if (((NonTermianlNode)
node).getNonTermianlName().equals("VarDecl")){
            for (int i = 1; i < node.getSubNodes().size(); i += 2){
                varDecls.add(new
VarDeclNode(node.getSubNodes().get(i), symbolTable));
            }
        }
        else if (((NonTermianlNode)
node).getNonTermianlName().equals("FuncDef")){
            funcDefs.add(new FuncDefNode(node, symbolTable, false));
        } else {
            mainFuncDef = new FuncDefNode(node, symbolTable, true);
        }
    }
}
```

## 错误处理

除了已在词法分析和语法分析处理了的a、i、j、k。其他作逐一介绍

- b

在进行符号表的插入操作时进行判断当前符号表是否已存在同名的即可。由于函数和变量可重名，分别进行判断。

```
public void insert(SymbolInfo symbolInfo){
    if (symbolInfo instanceof FuncSymbolInfo){
        if (funcTable.containsKey(symbolInfo.getIdent().getword())){
            Errors.addError(symbolInfo.getIdent().getLineNum(), "b");
        }
    }
}
```

```

        else funcTable.put(symbolInfo.getIdent().getWord(), (FuncSymbolInfo)
symbolInfo);
    }
    else {
        if (varTable.containsKey(symbolInfo.getIdent().getWord())){
            Errors.addError(symbolInfo.getIdent().getLineNum(), "b");
        }
        else varTable.put(symbolInfo.getIdent().getWord(), symbolInfo);
    }
}

```

- c

在当前符号表以及上级所有符号表中寻找是否已定义。若找不到即返回null，添加c类错误。

```

public SymbolInfo find(String ident, boolean isFunc){
    if (isFunc){
        if (funcTable.containsKey(ident)){
            return funcTable.get(ident);
        }
        else if (prev!=null){
            return prev.find(ident, isFunc);
        }
    }
    else {
        if (varTable.containsKey(ident)){
            return varTable.get(ident);
        }
        else if (prev!=null){
            return prev.find(ident, isFunc);
        }
    }
    return null;
}

```

- d、e

在函数调用时，判断函数实参是否与函数定义时形参一致，函数定义可从符号表中找到。

```

if (funcSymbolInfo.getParaDims().size()==funcRParamNodes.size()){
    for (int i = 0 ; i<funcRParamNodes.size(); i++){
        if (funcRParamNodes.get(i).getDim() !=
funcSymbolInfo.getParaDims().get(i)){

            Errors.addError(((TermianlNode)nodes.get(0)).getWord().getLineNum(), "e");
            break;
        }
    }
}
else {
    Errors.addError(((TermianlNode)nodes.get(0)).getWord().getLineNum(), "d");
}

```

- f、g、m

每一个stmt和block都有记录“当前是否为函数体？”“当前是否为返回int的函数中？”“当前是否为循环体中？”三个标记。

因此只需要在读到return时判断当前是否是返回int的函数中，不是则报错f。

```
stmtItem =new ReturnNode(nodes,symbolTable);
if (!isReturnInt&&((ReturnNode)stmtItem).isReturnInt()){
    Errors.addError(((TermianlNode)nodes.get(0)).getWord().getLineNum(),"f");
}
```

在完成一个函数block的读入后判断最后一个语句是否为return，不是则报错g。

```
if (isFuncBlock && isReturnInt && blockItems.size() > 0 &&
blockItems.get(blockItems.size() - 1) instanceof StmtNode && !(((StmtNode)
blockItems.get(blockItems.size() - 1)).getStmtItem() instanceof ReturnNode)) {
    Errors.addError(((TermianlNode) subnodes.get(subnodes.size() -
1)).getWord().getLineNum(), "g");
}
else if (isFuncBlock && isReturnInt && blockItems.size() == 0) {
    Errors.addError(((TermianlNode) subnodes.get(subnodes.size() -
1)).getWord().getLineNum(), "g");
}
else if (isFuncBlock && isReturnInt && !(blockItems.get(blockItems.size() - 1)
instanceof StmtNode)){
    Errors.addError(((TermianlNode) subnodes.get(subnodes.size() -
1)).getWord().getLineNum(), "g");
}
```

在读入break或continue时，判断当前是否为循环体中，不是则报错m。

```
if (!isLoop){
    Errors.addError(((TermianlNode)nodes.get(0)).getWord().getLineNum(),"m");
}
```

- h

在AssignNode中进行判断，读入的变量是否为常数声明即可。

```
if (lvalInfo instanceof ConstSymbolInfo){
    Errors.addError(lvalNode.getLinenum(),"h");
}
```

- l

在printfNode中进行判断，采用使用split("%d")的方法来找到%d的个数，再与exp的个数作比较。

```
numOfd = formatString.split("%d").length - 1;
for (int i = 4;i<nodes.size()-1;i+=2){
    if (nodes.get(i) instanceof NonTermianlNode){
        exps.add(new ExpNode(nodes.get(i),symbolTable));
    }
}
if (exps.size() !=numOfd){
    Errors.addError(((TermianlNode)nodes.get(0)).getWord().getLineNum(),"l");
}
```

# 中间代码

---

## 中间代码四元式



| Op     | arg1       | arg2 | result | 说明                               |
|--------|------------|------|--------|----------------------------------|
| add    | √          | √    | √      | result=arg1+arg2                 |
| sub    | √          | √    | √      | result=arg1-arg2                 |
| mul    | √          | √    | √      | result=arg1*arg2                 |
| div    | √          | √    | √      | result=arg1/arg2                 |
| mod    | √          | √    | √      | result=arg1%arg2                 |
| neg    | √          | -    | √      | result=-arg1                     |
| func   | √          | -    | √      | arg1 result()                    |
| para   | -          | -    | √      | para int result                  |
| push   | -          | -    | √      | push result                      |
| call   | -          | -    | √      | call result                      |
| ass    | RET/getint | -    | √      | result=RET/getint                |
| ret    | -          | -    | √      | ret result                       |
| var    | -          | -    | √      | var int result                   |
| conarr | √          | √    | √      | const arr int result[arg1][arg2] |
| varass | √          | -    | √      | var int result = arg1            |
| ass    | √          | -    | √      | result=arg1                      |
| conass | √          | -    | √      | const int result = arg1          |
| label  | -          | -    | √      | result:                          |
| j      | -          | -    | √      | j result                         |
| jgt    | √          | √    | √      | jump result if arg1>arg2         |
| jge    | √          | √    | √      | jump result if arg1>=arg2        |
| jlt    | √          | √    | √      | jump result if arg1<arg2         |
| jle    | √          | √    | √      | jump result if arg1<=arg2        |
| jeq    | √          | √    | √      | jump result if arg1==arg2        |
| jne    | √          | √    | √      | jump result if arg1!=arg2        |
| arr    | √          | √    | √      | arr int result[arg1][arg2]       |
| assarr | √          | √    | √      | result[arg1] = arg2              |
| getarr | √          | √    | √      | result = arg1[arg2]              |
| getadd | √          | √    | √      | result = &arg1[arg2]             |
| seq    | √          | √    | √      | result = (arg1==arg2)?1:0        |

| Op       | arg1 | arg2 | result | 说明                        |
|----------|------|------|--------|---------------------------|
| sne      | √    | √    | √      | result = (arg1==arg2)?0:1 |
| sgt      | √    | √    | √      | result = (arg1>arg2)?1:0  |
| sge      | √    | √    | √      | result = (arg1>=arg2)?1:0 |
| slt      | √    | √    | √      | result = (arg1<arg2)?1:0  |
| sle      | √    | √    | √      | result = (arg1<=arg2)?1:0 |
| printint | -    | -    | √      | printf int                |
| printstr | -    | -    | √      | printf str                |
| strcon   | √    | -    | √      | str result = arg1         |

采用如上所示的四元式，之后遍历AST，对每一种类型的节点重写一个生成中间代码的方法，从根节点开始调用即可完成，由于在生成AST时，已经对每一个变量进行了重新编号，确保了嵌套中的重名变量序号不同，因此生成中间代码直接使用变量的编号进行翻译，下面对几处特殊处理的地方作说明：

## printf

由于mips中输出至控制台有输出字符串和数字的对应syscall，在翻译printf时，将其拆为str和int，逐个进行输出，如一个 `printf("this is a num :%d\n",a);` 将翻译为三个print，第一个print输出 `this is a num :`，第二个print输出 `a` 所对应的值，第三个print输出 `\n`，如此便完成了printf的翻译。

## 数组

我们的数组有二维数组和一维数组，但是实际上都是以一维数组的形式储存在内存中的，对于二维数组，可以先计算出偏移，再得到对应的元素，因此一条对二维数组的操作会分为3步：

```
//x = a[i][j];
mul    i    dim2    t
add    t    j        t
getarr a    t        x
```

其中dim2是第2维的大小，可从符号表中得到。其他对数组的操作均可以类似的方式进行。另外由于可以将数组作为参数传参，因此实际上我的数组是一个指针，指向数组首地址，也提供了getadd来获取该地址。

## 短路求值

采用如下的短路求值方案：  
设cond1、cond2、cond3、cond4、cond5、cond6为条件，jcond label即为满足则跳转。  
为了使得翻译时thenstmt和elsestmt的顺序不变，将所有的条件取反，则可得到下列&&短路的翻译：

```
&&短路

if (cond1&&cond2&&cond3){
    thenstmt
}
else {
```

```

    elsestmt
}

```

可翻译为:

```

j~cond1 else //第一个条件不满足立刻跳转至else
j~cond2 else //第二个条件不满足立刻跳转至else
j~cond3 else //第三个条件不满足立刻跳转至else
then:
    thenstmt
    j end
else:
    elsestmt
end:

```

同理可得到||短路:

```

||短路
if (cond1||cond2||cond3){
    thenstmt
}
else {
    elsestmt
}

```

翻译为:

```

jcond1 then //第一个条件满足立刻跳转至then
jcond2 then //第二个条件满足立刻跳转至then
jcond3 then //第三个条件满足立刻跳转至then
j else
then:
    thenstmt
    j end
else:
    elsestmt
end:

```

对于同时有&&和||的情况,与上面类似,只需加一个label在每一个||上,

```

if (con1&&con2&&con3 || con4&&con5&&con6)
    thenstmt
else
    elsestmt

    j ~cond1 label1 //不满足则立刻跳至label1 开始||之后的判断
    j ~cond2 label1 //不满足则立刻跳至label1 开始||之后的判断
    j ~cond3 label1 //不满足则立刻跳至label1 开始||之后的判断
    j then
label1:
    j ~cond4 else //不满足则立刻跳转至else
    j ~cond5 else //不满足则立刻跳转至else
    j ~cond6 else //不满足则立刻跳转至else
    j then

then:
    thenstmt

```

```
    j end
else:
    elsestmt
end:
```

在实现时只需递归调用中间代码生成函数，函数有一个参数，即为当前条件若不成立应该跳至哪个label。

## 函数调用

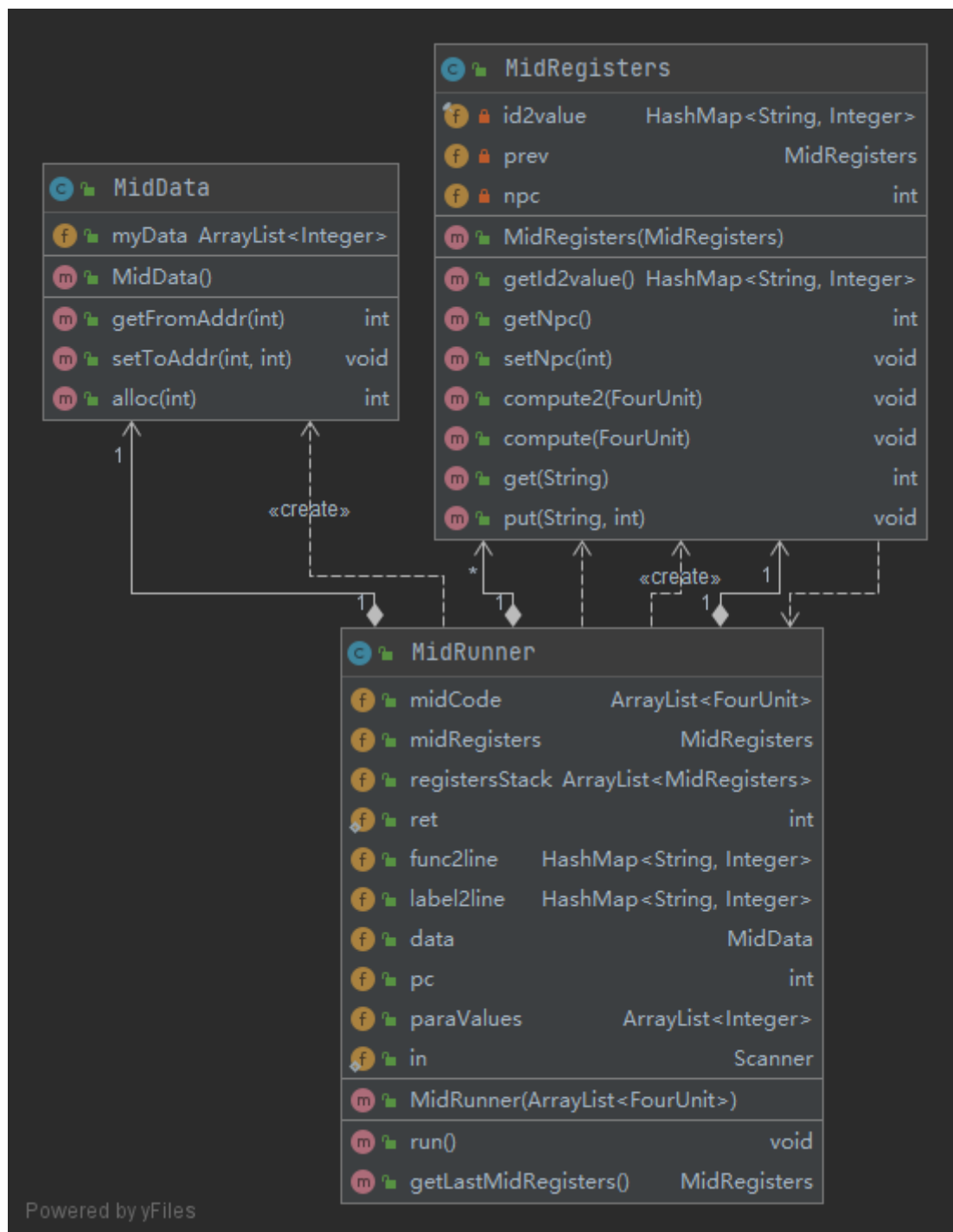
所有的函数都以相同的方式调用，无论是有返回值还是无返回值，返回值将存在于ret中，需要使从ret中取出即可。另外函数的参数以push的方式进行传递，进入函数后由para得到相应的参数。

## 简单的pcode实现

---

该部分仅用于验证中间代码的正确性，以及对优化后的中间代码进行检查是否保持原义，因此仅使用了300行左右代码完成了一个非常简洁的运行器。

分为三个部分：MidRunner、MidData、MidRegisters



MidData为模拟的内存，MidRegisters为模拟的寄存器堆。

MidRunner中各变量的含义：

- pc: 当前运行的中间代码行号
- data: 指向内存空间，可对某一个位置进行读写
- midRegisters: main函数的寄存器堆
- registersStack: 运行栈
- ret: 函数的返回值
- paraValues: 函数参数
- func2line: 保存函数对应的起始行号
- label2line: 保存label的行号

运行时首先创建一个寄存器堆作为main的现场，之后对声明进行初始化，随后将pc改为main的起始行号，开始运行，对每一种类型的四元式作相应的操作。若进行了函数的调用，则创建一个新的寄存器堆，函数返回时销毁运行栈的最顶层寄存器堆。

MidData用一个arraylist来模拟，可以对其申请一块内存、读某位置、写某位置。

MidRegisters用一个HashMap来模拟，对每一个变量设置一个寄存器，npc也即返回地址。

## 代码生成

---

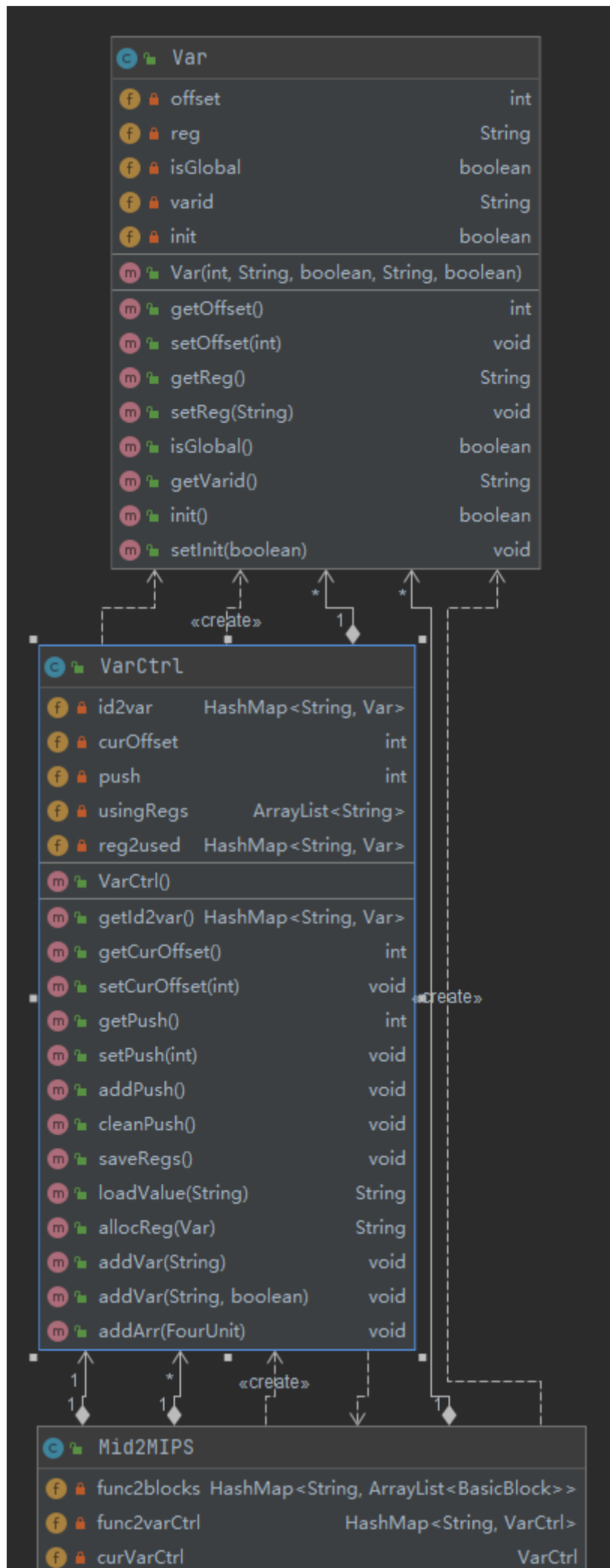
将中间代码生成mips代码。设置一个临时寄存器池管理寄存器，在对某一个变量操作时，先去查找其是否在寄存器中，不在则先从内存中进行读取并分配寄存器，之后进行相应的操作。对于大部分运算，如add、sub直接原样翻译，其他的下面作详细说明。在翻译时，首先翻译全局变量，即在.data处声明所有的全局变量，对于需要初始化的变量，将初始化操作放在main的最开始进行。在翻译完成全局变量后，先翻译main函数，确保main函数先运行，之后再翻译其他函数。


















在实现上，主要由三个类构成。

Var用于保存变量的各种信息，偏移量、当前寄存器、是否全局、变量名、是否已初始化。

VarCtrl用于维护Var，提供寄存器的管理、变量的操作等等。

Mid2MIPS用于生成代码，采用上述的顺序对每一个函数逐个基本块的进行翻译。



|                                                                                                                                    |                      |
|------------------------------------------------------------------------------------------------------------------------------------|----------------------|
|  globalVars                                        | HashMap<String, Var> |
|  Mid2MIPS(HashMap<String, ArrayList<BasicBlock>>) |                      |
|  intoMIPS()                                       | void                 |
|  initDecls()                                      | void                 |
|  genFuncs()                                       | void                 |
|  genFunc(String)                                  | void                 |
|  compute2(FourUnit)                               | void                 |
|  compute(FourUnit)                                | void                 |
|  allocArr(FourUnit)                               | void                 |
|  assarr(FourUnit)                                 | void                 |
|  getarr(FourUnit)                                 | void                 |
|  getadd(FourUnit)                                 | void                 |
|  getReg(String)                                   | String               |
|  initDecl(FourUnit)                               | void                 |
|  setDecl()                                        | void                 |
|  allocDecl(FourUnit)                              | void                 |
|  setStrs()                                        | void                 |

Powered by yFiles

## 变量管理

一个变量有状态：全局变量/局部变量，已初始化/未初始化，已分配到寄存器/未分配寄存器。

对于进行一次操作时，将首先查看其是否已获得寄存器，若已有寄存器则直接返回该寄存器，否则申请一个寄存器，再返回该寄存器。此外，若该变量未经过初始化，则保险起见将对分配的寄存器作赋0值的处理。对于全局变量，也以类似的方式进行，只是其内存位置不一样。

## 字符串

printf中的字符串，将在全局变量后进行声明，以备后续的输出。

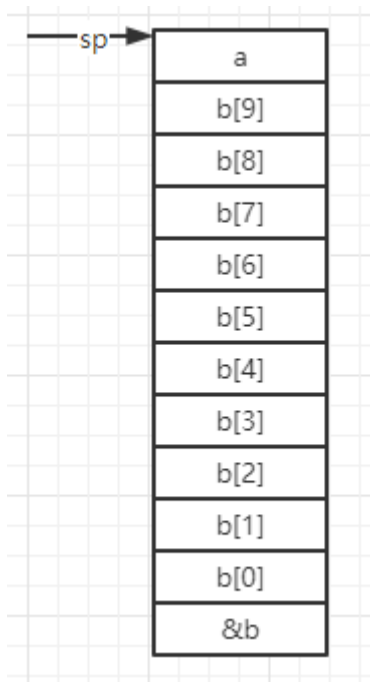
```
.data
a_0: .space 4
str_1: .asciiz ", "
str_0: .asciiz ", "
str_2: .asciiz "\\n"
```

## mips空间管理

对于全局变量，采用.data进行声明，对于局部变量，通过sp和偏移来管理，每一个变量在翻译时将获得自己与sp的偏移量，从而找到相应的内存位置。对于数组变量，会额外多出一个位置的偏移，用于保存其首地址。以下即为一个示例：

```
int main(){
    int a;
    int b[10];
}
```





## 函数调用

在进行函数调用时，首先进行push的处理，这里我将所有参数完全压入栈中，同时留一个位置用于保存ra，之后将sp移动到新的位置，这时sp上方的内存空间将无法被访问（也即调用时的现场被保存），在进入函数后，首先读入参数，由于参数已经被压进了栈中，故不需要对其进行初始化，直接增加目前的偏移量即可（避免后续的变量占用该位置）。返回后sp退回至原样，ra恢复，之后继续运行。

## 读入与输出

我的读入是如下的四元式：

```
ass getint null a_3
```

翻译时会判断其是否为getint，是的话将翻译为系统调用后再将\$v0的值赋值给a\_3所对应的寄存器

输出时也类似，直接通过str的编号，mars会将其翻译为对应的字符串地址：

```
//输出str_0  
li $v0, 4  
la $a0, str_0  
syscall
```

## 单变量操作

```
case "neg":  
case "ass":  
case "var":  
case "varass":  
case "conass":
```

对于这些都是arg2为空的四元式，提供相应的操作：如ass翻译为move。

## 双变量操作

```
case "add":
case "sub":
case "mul":
case "div":
case "mod":
case "seq":
case "sne":
case "sgt":
case "sge":
case "slt":
case "sle":
```

这些都是result为arg1和arg2共同操作得到。由于与mips中操作一致，直接翻译为mips即可。（mod将拆分为一个div和mfhi）

## 代码优化

### 临时寄存器分配

由于时间问题，仅做了临时寄存器分配，使用的策略是若有空闲寄存器则分配寄存器，若无则将最久未使用的寄存器替换（LRU）。

### 窥孔优化

- 删除无效的跳转。如

```
j label
label:
---
```

该跳转是完全无意义的，进行删除

- 删除重复赋值。如

```
ass 0 null a
ass 2 null a
```

前一个赋值可删去。

- 删除无用赋值。如

```
ass 0 null a
add b c a
```

可发现实际上前一个赋值是无效的。

### 乘除优化

仅完成了当除数和乘数为2的倍数时使用移位指令来代替。

## 高效指令

- div指令的三操作数形式会多出一条判断除数是0的分支指令，可将其替换为div+mflo的操作

|            |                  |                         |
|------------|------------------|-------------------------|
| 0x15600001 | bne \$11, \$0, 1 | 1: div \$t0, \$t1, \$t3 |
| 0x0000000d | break            |                         |
| 0x012b001a | div \$9, \$11    |                         |
| 0x00004012 | mflo \$8         |                         |

- mul指令则直接使用mul

|            |                    |                         |
|------------|--------------------|-------------------------|
| 0x712b4002 | mul \$8, \$9, \$11 | 1: mul \$t0, \$t1, \$t3 |
|------------|--------------------|-------------------------|