

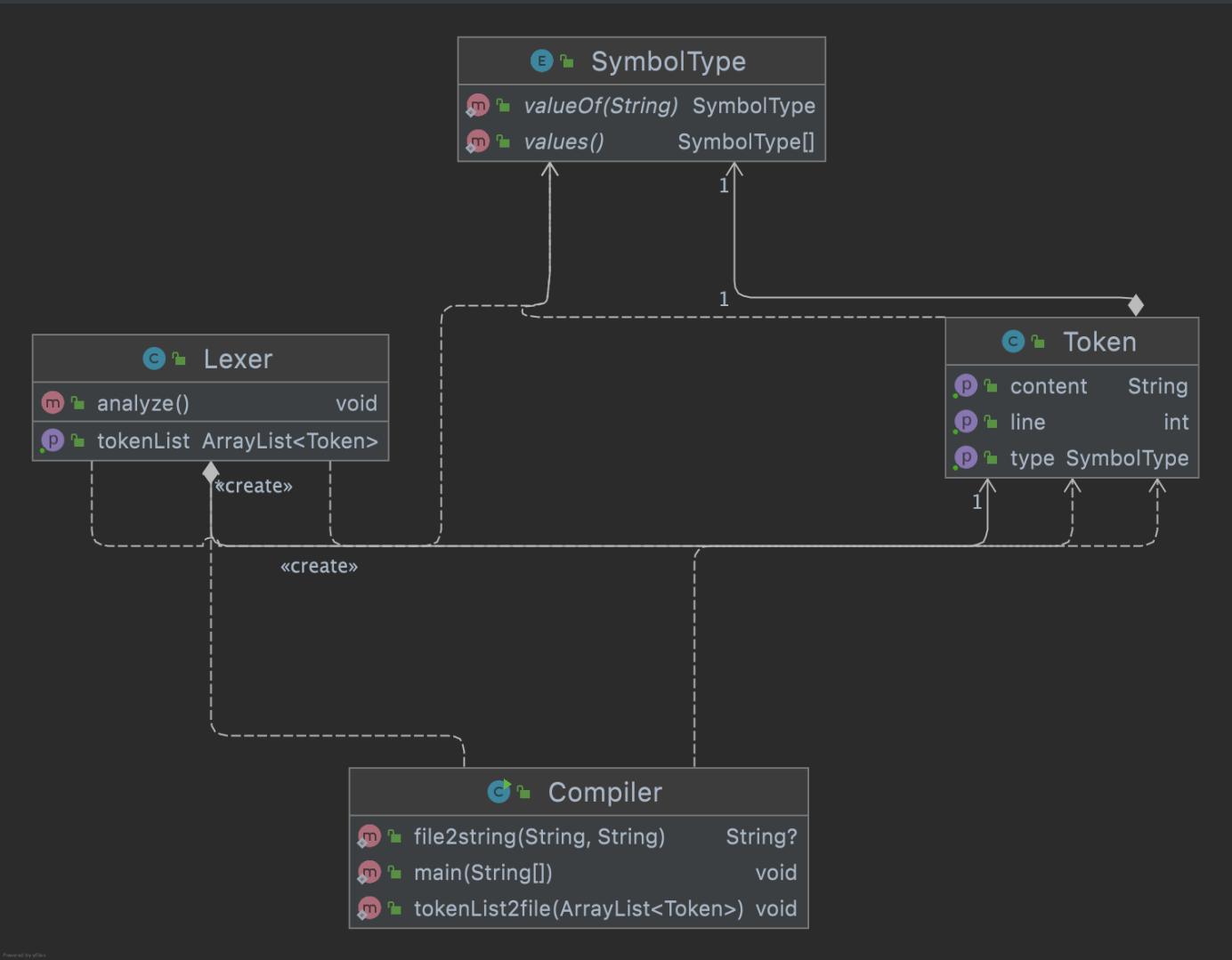
SysY语言编译器设计文档

词法分析

词法分析主要需要实现的是代码的扫描与处理，包含但不限于token的判断与存储、注释符号处理、特殊符号处理等问题。

架构设计

Diagram



Compiler类

实现两个静态方法：**file2string()** 以及 **tokenList2file()**。

file2string() 将 **testfile.txt** 文件中的代码内容拼接成一整个字符串，并且保留每行的换行符 **\n**，以便 **Lexer** 进行处理。

tokenList2file() 则将输入的 **code** 字符串进行分析，并且将分析的结果保存为 **token**，存放到 **tokenList** 并返回给 **Compiler** 进行输出。

main() 方法则是进行数据流的传递以及编码的检查，保证编译器按照正确的流程工作，具体执行方式如下：

```
public static void main(String[] args) {  
    String encoding = "utf-8";  
    String filePath = "testfile.txt";  
    String code = file2string(filePath, encoding);  
    Lexer lexer = new Lexer(code);  
    tokenList2file(lexer.getTokenList());  
}
```

Lexer类

Lexer 本质上是一个 **String** 扫描分析器，将输入的 **code** 字符串逐个进行读取，并通过关键字进行分类，主要分为 **identifier and reserved words**，**digit number**，**string"xxx"**，**! and !=**，**&&**，**||**，**+**，**-**，*****，**/ and annotation**，**%**，**< and <=**，**> and >=**，**= and ==**，**;**，**,**，**(,)**，**[,]**，**{ , }**。

在遇到 **\n \t \r \0 \space** 时，会跳过这些字符，在读取到 **\n** 时会将行数增加。

判断标识符与保留字

读取完符合标识符与保留字要求的字符之后，会对读取到的单词进行判断，如果是保留字就会存为不同的类型。

```
// identifier and reserved words
```

```

if (Character.isLetter(code.charAt(charPtr)) || code.charAt(charPtr) ==
'_' ) {
    while (charPtr < codeLen && (Character.isLetter(code.charAt(charPtr))
|| Character.isDigit(code.charAt(charPtr))) || code.charAt(charPtr) ==
'_' ) {
        tokenContent += code.charAt(charPtr);
        charPtr++;
    }
    tokenType = SymbolType.IDENFR;
    // judge for reserved words
    for (int j = 0; j < resWords.length; j++) {
        if (tokenContent.equals(resWords[j])) {
            tokenType = SymbolType.values()[j];
            break;
        }
    }
    tokenList.add(new Token(tokenContent, tokenType, curLine));
}

```

处理除法与注释

在读到 / 时，通过向后多读1~2个字符，来判断是否是注释。对于多行注释，对注释行内的
 \n 进行读取增加 curLine，确保行数的正确，并且将注释结束符作为循环结束的条件，可以做到忽略多行注释中的内容。

```

// / and annotation handle
else if (code.charAt(charPtr) == '/') {
    tokenContent = "/";
    tokenType = SymbolType.DIV;
    charPtr++;

    // handle annotation
    if (code.charAt(charPtr) == '/') {
        charPtr++;
        while (charPtr < codeLen) {
            if (code.charAt(charPtr) == '\n') {

```

```

        curLine++;
        charPtr++;
        break;
    }
    charPtr++;
}
}
else if (code.charAt(charPtr) == '*') {
    charPtr++;
    while (charPtr < codeLen - 1 && (code.charAt(charPtr) != '*' ||
code.charAt(charPtr + 1) != '/')) {
        if (code.charAt(charPtr) == '\n') {
            curLine++;
        }
        charPtr++;
    }
    if (charPtr < codeLen - 1) {
        charPtr += 2;
    }
}

else {
    tokenList.add(new Token(tokenContent, tokenType, curLine));
}
}

```

Token类

token 类本质上是一个存放读取到的单词的容器，通过记录每一个单词的内容 content ，类型 type ，所在行数 line ，方便后续语法分析、出错处理读取行数等信息。

```

public class Token {
    public String content;
    public SymbolType type;
    public int line;
}

```

```
public Token(String symbol, SymbolType type, int line) {
    this.content = symbol;
    this.type = type;
    this.line = line;
}

public String getContent() {
    return content;
}

public SymbolType getType() {
    return type;
}

public int getLine() {
    return line;
}
}
```

设计优化

SymbolType类

这是一个 java enum ，用来记录 sysY 语法中的所有符号类型，可在词法分析、语法分析、出错处理时被调用。原先设计在添加/判断 token 类型的时候，均使用字符串格式，考虑到这样的设计不够优雅、模块化，改为了 Lexer 内建 enum 。然而内建 enum 使得 Lexer 变得过于复杂，最终将此 enum 单独封装成一个类。

```

public enum SymbolType {
    MAINTK,           //main
    CONSTTK,          //const
    INTTK,            //int
    BREAKTK,          //break
    CONTINUETK,       //continue
    IFTK,             //if
    ELSETK,           //else
    ...
}

```

在关键字调用的时候也可以使用 index 对应到相应 enum 中的 type。

```

private String[] resWords = {"main", "const", "int", "break",
    "continue", "if", "else", "while", "getint", "printf", "return",
    "void"};

for (int j = 0; j < resWords.length; j++) {
    if (tokenContent.equals(resWords[j])) {
        // 与 SymbolType 中关键字的 index 相对应
        tokenType = SymbolType.values()[j];
        break;
    }
}

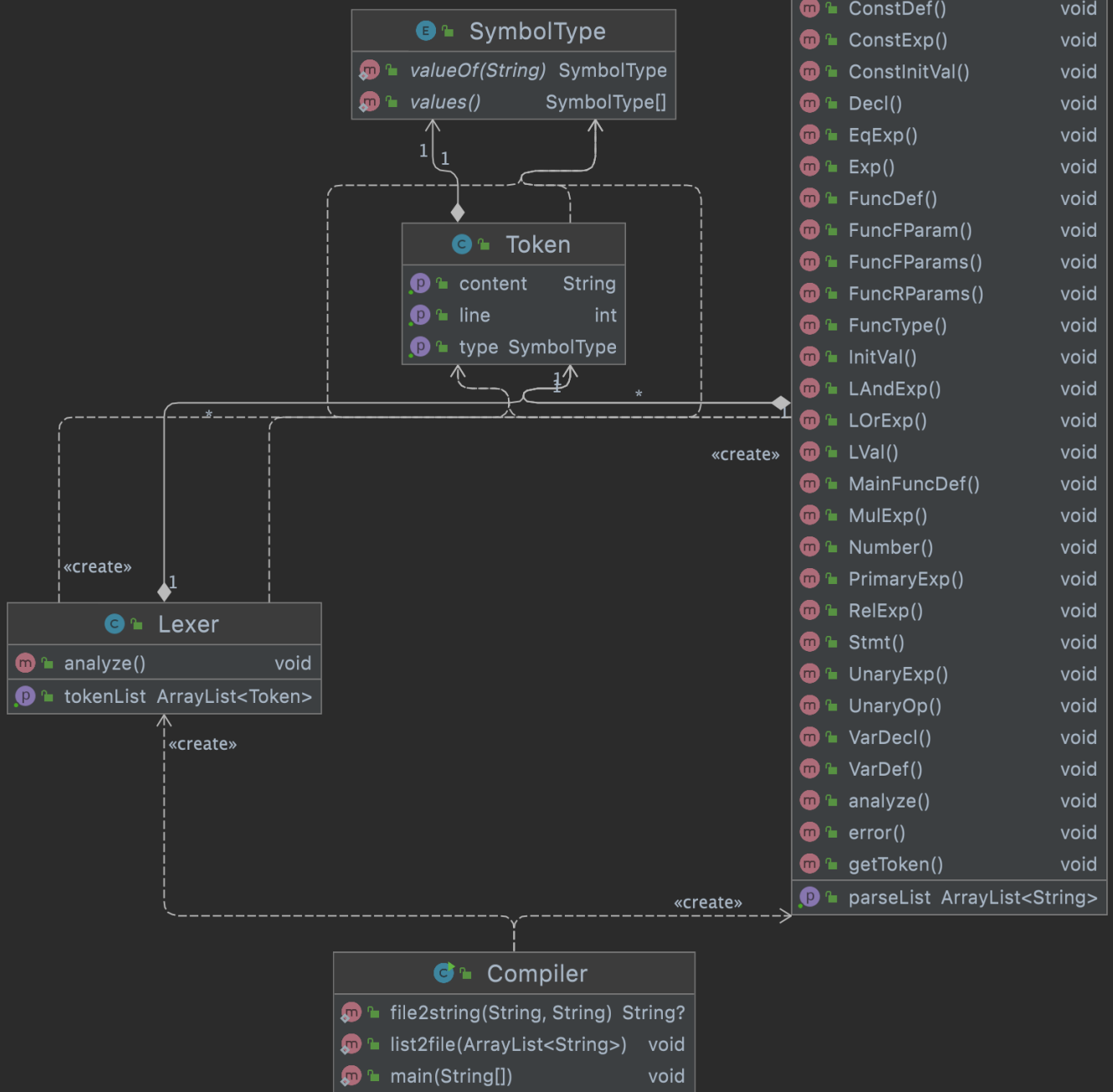
```

语法分析

在这里，采用LL(1)分析法，自顶向下分析 sysY 语法。处理过程中，需要解决一些 token 预读取、左递归文法改写等问题。

架构设计

Diagram



Parser类

在 parser 中，对文法进行分析，将每一个非终结符封装成一个分析函数，并且向下不断调用，直至遇到终结符后逐层返回。分析时使用 lexer 扫描生成的 tokenList，从产生式 CompUnit() 开始进行推导。

```
public Parser(ArrayList<Token> tokenList) {
    this.tokenList = tokenList;
    tokenList.add(new Token("end", SymbolType.END, 0));
    this.parseList = new ArrayList<>();
    this.tkPtr = 1;
    this.curToken = tokenList.get(0);
    this.analyze();
}

public void analyze() {
    CompUnit();
}
```

在处理时，为了方便读到程序结尾停止，自定义一种终结符类型 END，并添加到 SymbolType 中，并添加到要进行分析的 tokenList 当中。调用 analyze() 方法开始分析。

在处理 token 时，维护 parser 的两个属性：tkPtr 以及 curToken。curToken 指向当前 token，tkPtr 指向下一个 token 在 tokenList 中的 index，当调用 getToken() 方法时，将 curToken 内容保存到 parseList 中，并利用 tkPtr 将 curToken 移动到下一个 token，tkPtr 再增加1。

```
public void getToken() {
    parseList.add(curToken.getType() + " " + curToken.getContent());
    curToken = tokenList.get(tkPtr);
    tkPtr++;
}
```

判断下一个非终结符

在设计递归下降分析方法时，默认所有的方法进入时 `curToken` 指向当前非终结符/终结符的第一个 token，在结束时则将 `token` 指向下一个非终结符/终结符的第一个 token，并调用相关的分析方法。

以 `PrimaryExp` 为例，**`PrimaryExp`** \rightarrow **`(' Exp ') | LVal | Number`**，在调用 `PrimaryExp()` 方法时，`curToken` 指向 `PrimaryExp` 的第一个可能的 token，即为 **`(`**，**`First(LVal)`**，**`First(Number)`**。

此时根据 **`LVal`** \rightarrow **`Ident {' Exp '}`**，**`Number`** \rightarrow **`IntConst`**，不难得到 `PrimaryExp` 的第一个 token，有如下分析程序。

```
public void PrimaryExp() {
    if (curToken.type == SymbolType.LPARENT) {
        getToken();
        Exp();
        if (curToken.type == SymbolType.RPARENT) {
            getToken();
        } else {
            error();
        }
    }
    else if (curToken.type == SymbolType.IDENFR) {
        LVal();
    }
    else if (curToken.type == SymbolType.INTCON) {
        Number();
    }
    else {
        error();
    }
    parseList.add("<PrimaryExp>");
}
```

在方法结束时，将当前非终结符标识存入 `parseList`，方便输出。

改写左递归文法

sysY文法中存在左递归，这在自顶向下分析法中会造成死循环。比如， $\text{LOrExp} \rightarrow \text{LAndExp} \mid \text{LOrExp} \mid \text{LAndExp}$ ，其中存在左递归 $\text{LOrExp} \rightarrow \text{LOrExp} \mid \text{LAndExp}$ 。

将上述左递归文法改写为 $\text{LOrExp} \rightarrow \text{LAndExp} \{ \mid \text{LAndExp} \}$ ，利用 while 循环即可解决此类问题。

```
public void LOrExp() {
    LAndExp();
    parseList.add("<LOrExp>");
    while (curToken.type == SymbolType.OR) {
        getToken();
        LAndExp();
        parseList.add("<LOrExp>");
    }
}
```

需要注意的是，左递归文法改写后非终结符的输出需要添加在每次调用下一非终结符方法之后，否则会存在缺失。

设计优化

AST抽象语法树

在递归下将分析时，可以将读取到的非终结符/终结符看作树的节点进行保存，CompUnit 则为树的根节点。如此，在语法分析之后可以得到一棵抽象语法树，在进行错误处理与中间代码生成时，可以更方便的进行读取、寻找相关信息。

Node类

```

public class Node {
    private Token token;
    private Node parNode;
    private ArrayList<Node> childNode = new ArrayList<>();
    private NodeInfo nodeInfo = new NodeInfo();
    private ArrayList<Integer> paramDim = new ArrayList<>();
    ...
}

```

新建 node 类，存入每个节点的信息、父节点以及所有子节点，在输出语法分析结果时使用后续遍历即可。

对于 parser 中的每个方法，为其传入上一方法的节点 node，则此 node 则为当前方法的父节点。

```

public void Cond(Node node) {
    // 新建当前方法节点
    Node Cond = new Node("Cond");
    // 将当前方法的节点与上一方法的节点连接起来
    addRelation(node, Cond);
    if (curToken.type == SymbolType.LPARENT || curToken.type ==
        SymbolType.IDENFR || curToken.type == SymbolType.INTCON || curToken.type
        == SymbolType.PLUS || curToken.type == SymbolType.MINU || curToken.type
        == SymbolType.NOT) {
        LOrExp(Cond);
    }
    parseList.add("<Cond>");
}

```

addRelation() 方法用于连接输入的两个 node。

```
public void addRelation(Node parent, Node child) {
    parent.addChildNode(child);
    child.setParNode(parent);
}
```

同时，getToken() 在保存 curToken 的同时，将 curToken 新建为一个节点作为返回值。

```
public Node getToken(Node node) {
    parseList.add(curToken.getType() + " " + curToken.getContent());
    Node tNode = new Node(curToken);
    addRelation(node, tNode);
    curToken = tokenList.get(tkPtr);
    tkPtr++;
    return tNode;
}
```

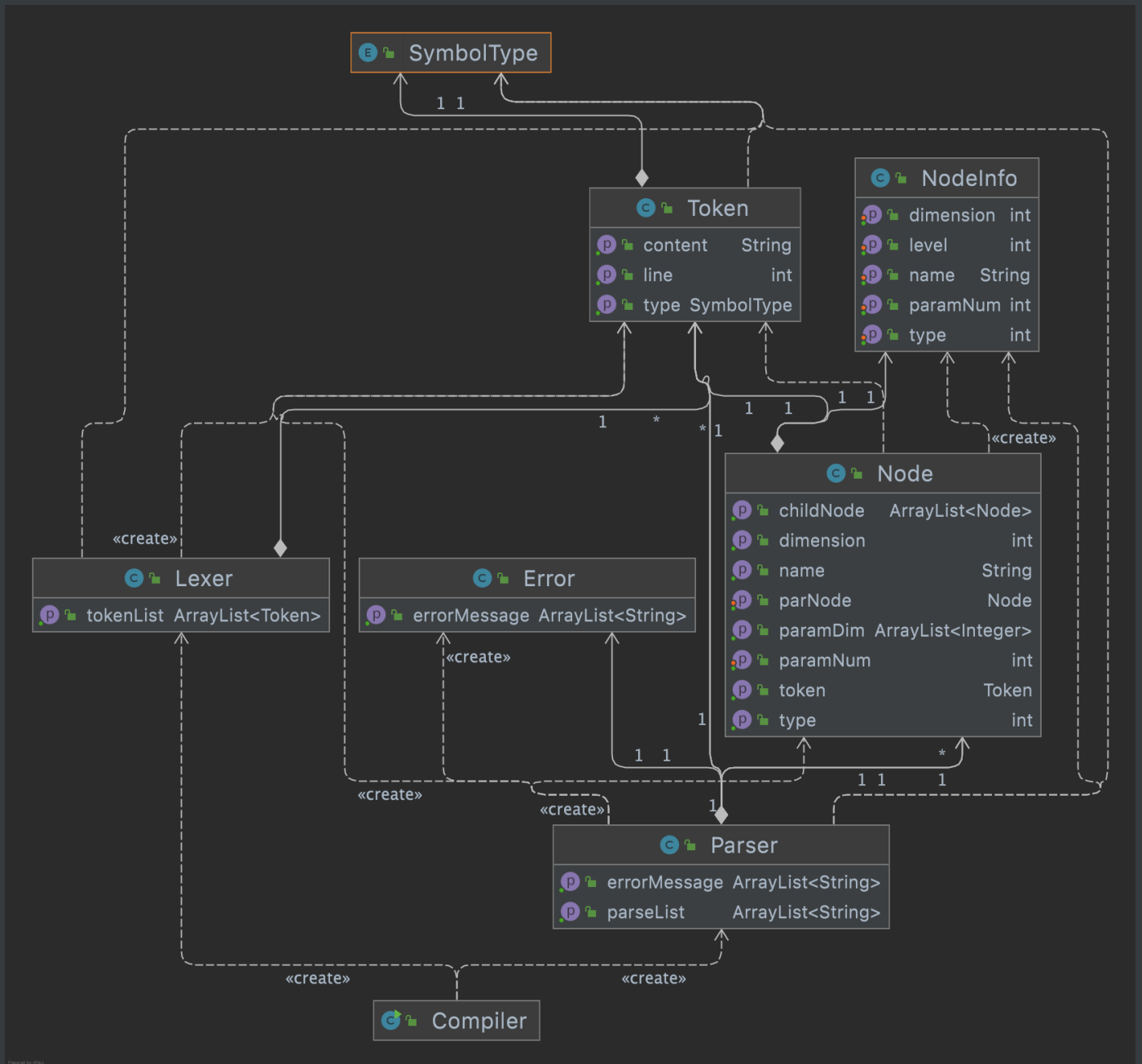
在输出语法树内容时，使用后序遍历。

```
public void recursiveOutputTree(Node node) {
    for (Node child : node.getChildNode()) {
        recursiveOutputTree(child);
    }
    System.out.println(node.getName());
}
```

错误处理

架构设计

Diagram



Error类

新增 Error 类，在 Error 中设置错误信息表，实现 `addErrorMessage()` 以及 `getMessageList()` 方法。

Parser类

针对要求实现的几种错误类型检查，作出如下新增。

构建符号表

通过 globalSymbolTable 以及 curTable 建立动态符号表。curTable 为当前局部符号表指针，同时维护一个 int 型变量 tbPtr，作为 prevadr 指针，指向当前符号表的上一个符号表首地址。

实现如下几个方法，进行局部符号表的新建，删除，以及插表前的检查工作：

```
private void insertTable(Node node) {
    curTable.add(node);
    System.out.println(node.getName() + " " + node.getType() + " tbPtr " +
tbPtr);
}

private void removeTable() {
    curTable = globalSymbolTable.get(tbPtr - 2);
    globalSymbolTable.remove(tbPtr - 1);
    tbPtr--;
}

private boolean checkTable(Node node) {
    for (Node curtbNode : curTable) {
        if (curtbNode.getName() != null &&
curtbNode.getName().equals(node.getName())
        && !(curtbNode.getType() == NodeInfo.CONST && node.getType() ==
NodeInfo.INTFUN)
        && !(curtbNode.getType() == NodeInfo.INTFUN && node.getType() ==
NodeInfo.CONST)
        && !(curtbNode.getType() == NodeInfo.CONST && node.getType() ==
NodeInfo.VOIDFUN)
        && !(curtbNode.getType() == NodeInfo.VOIDFUN && node.getType()
== NodeInfo.CONST)
        && !(curtbNode.getType() == NodeInfo.VAR && node.getType() ==
NodeInfo.INTFUN)
        && !(curtbNode.getType() == NodeInfo.INTFUN && node.getType() ==
NodeInfo.VAR)
```

```

        && !(curtbNode.getType() == NodeInfo.VAR && node.getType() ==
NodeInfo.VOIDFUN)
        && !(curtbNode.getType() == NodeInfo.VOIDFUN && node.getType()
== NodeInfo.VAR)) {
            return false;
        }
    }
    return true;
}

```

一般来说，在读取到 { 时进行局部符号表新建，在读取到 } 进行局部符号表删除。在 if 与 while 存在的某些情况下，需要进行特殊处理。

checkTable() 同时也对 node 的类型进行判断，满足 sysY 语言允许函数与变量重名的要求。

具体错误处理方法

类型a

通过对文法进行分析，可以得出所有的格式字符串必须满足如下正则表达式：

```

String NormalChar = "\\s|!|\\(|\\)|\\*|\\+|,|-|\\.|/|\\w|:|;|<|=|>|\\?|@|\\[|]|\\^|_|'|~|\\\\\\\\n";
String Char = "(" + formatChar + ")|(" + NormalChar + ")";
String formatString = "\"" + Char + "*\"";

```

通过正则匹配可以完成错误分析。

类型b

利用符号表，在 FuncDef、ConstDef、VarDef 函数中，插入符号表前先进行 checkTable() 操作，如果发现在同层次符号表中已有同名标识符，则报错；否则将当前新标识符插入到符号表中。


```

if (checkTable(idenfr)) {
    insertTable(idenfr);
} else {
    error.addErrorMessage(idenfr.getToken().getLine(), 'b');
}

```

类型c、d、e

新增 searchTable() 方法，通过不断上溯之前的所有符号表，判断是否存在“未定义的名字”、“函数参数个数不匹配”、“函数参数类型不匹配”、“不能改变常量的值”问题。

```

// 0:success 1:未定义的名字 2:函数参数个数不匹配 3:函数参数类型不匹配 4:不能改变常量的值
private int searchTable(Node node) {
    for (int i = globalSymbolTable.size() - 1; i >= 0; i--) {
        for (Node gnd : globalSymbolTable.get(i)) {
            if (gnd.getName() != null && gnd.getName().equals(node.getName()))
            {
                if (node.getType() == NodeInfo.VAR && gnd.getType() ==
NodeInfo.CONST) {
                    return 4;
                }
                if (gnd.getType() == node.getType() && node.getType() ==
NodeInfo.VAR) {
                    return 0;
                }
                if (gnd.getType() == node.getType() /*&& node.getType() ==
NodeInfo.INTFUN*/) {
                    if (gnd.getParamNum() != node.getParamNum()) {
                        return 2;
                    } else {
                        if (node.getParamNum() == 0) {
                            return 0;
                        } else {
                            for (int j = 0; j < node.getParamNum(); j++) {

```

```

        if (!Objects.equals(gnd.getParamDim().get(j),
node.getParamDim().get(j))) {
            return 3;
        }
    }
    return 0;
}
}
}
}
}
}
return 1;
}
}

```

在相关递归下降分析方法中调用 searchTable() 完成错误分析。

类型f

在 Stmt() 方法中，通过回溯当前符号表之前的所有符号表，判断包含 RETURN TK 的函数是否返回值不恰当：

```

for (int i = tbPtr - 2; i >= 0; i--) {
    if (globalSymbolTable.get(i).get(globalSymbolTable.get(i).size() -
1).getType() == NodeInfo.INTFUN
        || globalSymbolTable.get(i).get(globalSymbolTable.get(i).size() -
1).getType() == NodeInfo.VOIDFUN) {
        if (globalSymbolTable.get(i).get(globalSymbolTable.get(i).size() -
1).getType() == NodeInfo.VOIDFUN
            && hasValue) {
            error.addErrorMessage(nodePtr.getToken().getLine(), 'f');
        }
        break;
    }
}
}

```

类型g

在 Block() 中，对有返回值的函数设置 flag，如果 flag 为 true 但是分析不到 RETURN TK，则将错误信息加入到 errorMessage 中。

```
if (curToken.type == SymbolType.RBRACE) {
    Node rbrace = getToken(Block);
    if (isIntFunc) {
        boolean flag = false;
        for (Node ctNode : curTable) {
            if (ctNode.getToken().type == SymbolType.RETURN TK) {
                flag = true;
                break;
            }
        }
        if (!flag) {
            error.addErrorMessage(rbrace.getToken().getLine(), 'g');
        }
    }
    removeTable();
}
```

类型h

在 Stmt() 中如果读取到 GETINT TK 或者遇到 Stmt -> LVal '=' Exp 文法时，调用 searchTable() 进行判断。

类型i、j、k

在所有需要出现相关符号的语法分析程序末尾进行符号判断即可。

类型l

在读取到 PRINT TK 时，记录 '%d' 出现的次数 fstrNum 以及后面 exp 的个数 expNum，检查个数是否匹配。

```
while (m2.find()) {
    if (m2.group(0).equals("%d")) {
        fstrNum++;
    }
}
```

```

    }
}
getToken(stmt);
while (curToken.type == SymbolType.COMMA) {
    getToken(stmt);
    Exp(stmt, garbage);
    expNum++;
}
if (expNum != fstrNum) {
    error.addErrorMessage(nodePtr.getToken().getLine(), 'l');
}

```

类型m

break与continue的处理方式一致，均需要对符号表进行回溯查询，判断之前是否存在WHILETK。在回溯之前，可以先行判断当前BREAKTK或者CONTINUETK所在的符号表层级，如果层级存在问题，则可以直接报错。

```

if (tbPtr - 2 <= 0) {
    error.addErrorMessage(nodePtr.getToken().getLine(), 'm');
} else {
    boolean hasWhile = false;
    for (int i = tbPtr - 2; i >= 0; i--) {
        if (globalSymbolTable.get(i).get(globalSymbolTable.get(i).size() - 1).getName().equals("while")) {
            hasWhile = true;
            break;
        }
    }
    if (!hasWhile) {
        error.addErrorMessage(nodePtr.getToken().getLine(), 'm');
    }
}

```

设计优化

更好的判断封装

将涉及回溯的多种错误处理判断封装到一个方法中，即 searchTable()，

```
// 0:success 1:未定义的名字 2:函数参数个数不匹配 3:函数参数类型不匹配 4:不能改变常量的值
private int searchTable(Node node) {
    for (int i = globalSymbolTable.size() - 1; i >= 0; i--) {
        for (Node gnd : globalSymbolTable.get(i)) {
            if (gnd.getName() != null && gnd.getName().equals(node.getName()))
            {
                if (node.getType() == NodeInfo.VAR && gnd.getType() ==
NodeInfo.CONST) {
                    return 4;
                }
                if (gnd.getType() == node.getType() && node.getType() ==
NodeInfo.VAR) {
                    return 0;
                }
                if (gnd.getType() == node.getType() /*&& node.getType() ==
NodeInfo.INTFUN*/) {
                    if (gnd.getParamNum() != node.getParamNum()) {
                        return 2;
                    } else {
                        if (node.getParamNum() == 0) {
                            return 0;
                        } else {
                            for (int j = 0; j < node.getParamNum(); j++) {
                                if (!Objects.equals(gnd.getParamDim().get(j),
node.getParamDim().get(j))) {
                                    return 3;
                                }
                            }
                        }
                    }
                    return 0;
                }
            }
        }
    }
}
```

```

    }
  }
}
}
}
return 1;
}

```

在调用函数进行判断时，使用 switch-case 进行判断。

```

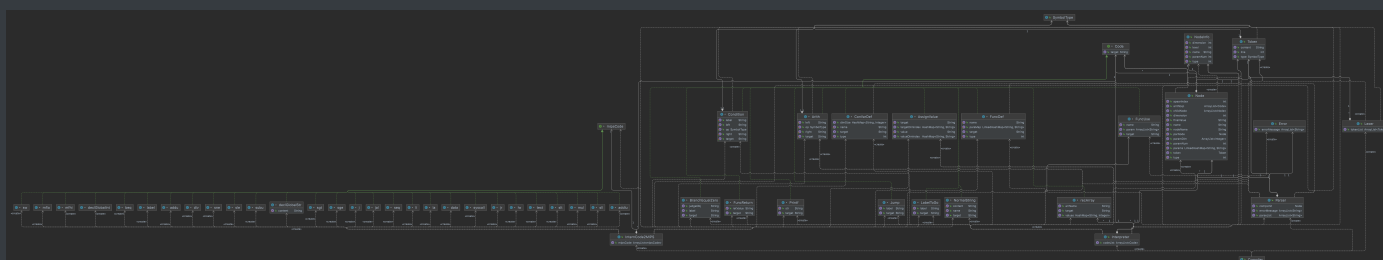
switch (searchTable(idenfr)) {
  case 1:
    error.addErrorMessage(idenfr.getToken().getLine(), 'c');
    break;
  case 2:
    error.addErrorMessage(idenfr.getToken().getLine(), 'd');
    break;
  case 3:
    error.addErrorMessage(idenfr.getToken().getLine(), 'e');
    break;
}

```

代码生成

架构设计

Diagram



中间代码结构设计

主要分为四元式计算、赋值操作、条件判断、常/变量定义、函数定义、函数返回值、跳转、打印输出等部分。下面给出样例代码对应的中间代码。

源代码：

```
int d = 4;

int hhh(int i){
    int c = i * i;
    if (i == 1) {
        return 1;
    }
    if (i == 2) {
        return 2;
    }
    c = c * c % 10;
    return hhh(i - 1) + hhh(i - 2);
}

int main () {
    int i = 2, j = 5;
    i = getint();
    j = getint();
    j = 7*5923/56*56 - hhh(hhh(3)) + (1+2-(89/2*36-53) /1*6-2*(45*56/85-56+35*56/4-9));
    int k = -+-5;
    int n = 10;
    while (n < k*k*k*k*k*k) {
        d = d * d % 10000;
        n = n + 1;
    }
    printf("%d, %d, %d\n", i, j, k);
    return 0;
}
```

对应的中间代码：

```
var int d$0
d$0 = 4
int hhh()
para int i$0
var int c$0
t$$0 = i$0 MULT i$0
c$0 = t$$0
t$$42 = i$0 null null
t$$43 = 1 null null
cond$0 = t$$42 EQL t$$43
beq cond$0, 0, if$0_end
if$0_begin:
ret 1
if$0_end:
t$$44 = i$0 null null
t$$45 = 2 null null
cond$1 = t$$44 EQL t$$45
beq cond$1, 0, if$1_end
if$1_begin:
ret 2
if$1_end:
t$$1 = c$0 MULT c$0
t$$2 = t$$1 MOD 10
c$0 = t$$2
t$$3 = i$0 MINU 1
push t$$3
call hhh
t$$4 = RET
t$$5 = i$0 MINU 2
push t$$5
call hhh
t$$6 = RET
t$$7 = t$$4 PLUS t$$6
ret t$$7
```



```
int main()
var int i$0
i$0 = 2
var int j$0
j$0 = 5
i$0 = getint
j$0 = getint
t$$8 = 7 MULT 5923
t$$9 = t$$8 DIV 56
t$$10 = t$$9 MULT 56
push 3
call hhh
t$$11 = RET
push t$$11
call hhh
t$$12 = RET
t$$15 = 89 DIV 2
t$$16 = t$$15 MULT 36
t$$17 = t$$16 MINU 53
t$$18 = t$$17 DIV 1
t$$19 = t$$18 MULT 6
t$$21 = 45 MULT 56
t$$22 = t$$21 DIV 85
t$$24 = 35 MULT 56
t$$25 = t$$24 DIV 4
t$$23 = t$$22 MINU 56
t$$26 = t$$23 PLUS t$$25
t$$27 = t$$26 MINU 9
t$$28 = 2 MULT t$$27
t$$14 = 1 PLUS 2
t$$20 = t$$14 MINU t$$19
t$$29 = t$$20 MINU t$$28
t$$13 = t$$10 MINU t$$12
t$$30 = t$$13 PLUS t$$29
j$0 = t$$30
var int k$0
```

```
t$$31 = MINU 5
t$$32 = PLUS t$$31
t$$33 = MINU t$$32
k$0 = t$$33
var int n$0
n$0 = 10
while$0_begin:
t$$34 = k$0 MULT k$0
t$$35 = t$$34 MULT k$0
t$$36 = t$$35 MULT k$0
t$$37 = t$$36 MULT k$0
t$$38 = t$$37 MULT k$0
cond$2 = n$0 LSS t$$38
beq cond$2, 0, while$0_end
while$0_stmt:
t$$39 = d$0 MULT d$0
t$$40 = t$$39 MOD 10000
d$0 = t$$40
t$$41 = n$0 PLUS 1
n$0 = t$$41
goto while$0_begin
while$0_end:
print i$0
print str$0
str$0 ,
print j$0
print str$1
str$1 ,
print k$0
print str$2
str$2 \n
ret 0
```

前期设计

生成代码部分主要的任务有 **token 转换为中间代码**，以及**中间代码转换为 MIPS 汇编代码**。主要实现通过 **Interpreter** 以及 **InternCode2MIPS** 两个模块完成。计划采用面向对象式的中间代码以及 mips 指令代码，对抽象语法树进行遍历，实现代码生成。预期难点在于数组在函数中的调用，数组维度的计算与初始化，if/while的短路断路判断。

Interpreter

由于在语法分析部分，已经实现了抽象语法树的创建，因此只需要对语法树节点进行遍历。

```

104      public void analyze() {...}
110
111  @      public void tran_CompUnit(Node node) {...}
126
127  @      public void tran_Decl(Node node) {...}
147
148      public void tran_VarDef(Node node) {...}
254
255      public void tran_ConstDef(Node node) {...}
369
370  @      public void tran_FuncDef(Node node) {...}
451
452  @      public void tran_Block(Node node) {...}
461
462  @      public void tran_BlockItem(Node node) {...}
474
475  @      public void tran_Cond(Node node, String label) {...}
503
504  @      public void tran_LAndExp(Node node, String label) {...}
512
513  @      public void tran_EqExp(Node node, String label) {...}
542
543  @      public void tran_RelExp(Node node, String label) {...}
587
588  @      public void tran Stmt(Node node) {...}
746
747  @      public void tran_MainFuncDef(Node node) {...}

```

同时，在保存中间代码信息时，设计一系列指令类并实现统一的接口 `Code`，例如 `FuncUse`、`FuncDef`、`ConVarDef` ...

- ▼ **C** Interpreter.java
 - C** Arith
 - C** AssignValue
 - C** BranchEqualZero
 - I** Code
 - C** Condition
 - C** ConVarDef
 - C** FuncDef
 - C** FuncReturn
 - C** FuncUse
 - C** Interpreter
 - C** Jump
 - C** LabelToGo
 - C** NormalString
 - C** Printf
 - C** recArray

将抽象语法树的不同子树进行转译，通过中间代码对象化切分程序，建立代码森林，最终保存入中间代码库。

InternCode2MIPS

代码翻译部分在接收到 Interpreter 发送过来的 internCode 中间代码list之后，会针对不同类的中间代码进行翻译。在 InternCode2MIPS 中转换 MIPS代码部分，将常用指令封装成类，实现统一接口，并设计重写 toString() 方法。在写 mips.txt 文件时，对最终翻译完成的 mipsCode 列表遍历调用 toString()，将语句字符串写入文件中。

在翻译中间代码时，首先将中间代码分类，分为全局变量声明、函数定义以及main函数三个部分，通过 `classify()` 方法实现分类。

```
public void classify() {
    int ptr = 0;
    boolean isDeclEnd = false;
    while(ptr != internCode.size()) {
        if
(internCode.get(ptr).getClass().getName().substring(internCode.get(ptr).
getClass().getName().lastIndexOf(".") + 1).equals("FuncDef")) {
            isDeclEnd = true;
            if (((FuncDef) internCode.get(ptr)).getName().equals("main")) {
                while (ptr != internCode.size()) {
                    ic_MainFunc.add(internCode.get(ptr));
                    ptr++;
                }
                break;
            }
            ArrayList<Code> thisFunc = new ArrayList<>();
            thisFunc.add(internCode.get(ptr));
            ptr++;
            while
(!internCode.get(ptr).getClass().getName().substring(internCode.get(ptr)
.getClass().getName().lastIndexOf(".") + 1).equals("FuncDef")) {
                thisFunc.add(internCode.get(ptr));
                ptr++;
                if (ptr == internCode.size()) break;
            }
            ptr--;
            ic_Func.add(thisFunc);
        }
        if (!isDeclEnd) {
            ic_Decl.add(internCode.get(ptr));
        }
        ptr++;
    }
}
```

```
}
```

实现分类之后，对每个模块调用存在的中间代码的翻译器，并将结果保存至 `mipsCode` 中。

```
public void translateMainFunc() {
    mipsCode.add(new label("$MainFunc$"));
    funcOffset.put("main", 0);
    for (Code ic : ic_MainFunc) {
        translateArith(ic, mainfuncSymbolTable, "MainFunc", "main");
        translateAssignValue(ic, mainfuncSymbolTable, "MainFunc", "main");
        translateConVarDef(ic, mainfuncSymbolTable, "MainFunc", "main");
        translatePrintf(ic, mainfuncSymbolTable, "MainFunc", "main");
        translateFuncUse(ic, mainfuncSymbolTable, "MainFunc", "main");
        translateBranchEqualZero(ic, mainfuncSymbolTable, "MainFunc",
"main");
        translateCondition(ic, mainfuncSymbolTable, "MainFunc", "main");
        translateJump(ic, mainfuncSymbolTable, "MainFunc", "main");
        translateLabelToGo(ic, mainfuncSymbolTable, "MainFunc", "main");
        translateFuncReturn(ic, mainfuncSymbolTable, "MainFunc", "main");
    }
    li li = new li("v0", 10);
    mipsCode.add(li);
    syscall syscall = new syscall();
    mipsCode.add(syscall);
}
```

函数调用与传参

处理函数调用时，简单的处理方法是**将参数保存至内存中**，并在保存的同时计算需要的栈空间 `offset`，将 **`$sp` 移动 `offset` 个单位**后调用 `jal` 指令跳转至函数中，之后将**函数返回值 `$v0`** 保存至当前函数的符号表中，同时**退回 `$sp` 栈指针**。更好的方法是**使用 `$a0` 等寄存器**，这样可以减少内存的压力，这种实现方法涉及代码优化，故不在这里展开。

这里存在两个难点，第一是**数组作为函数参数**怎么传入，第二是如何**区分一维数组与变量**。

数组参数传入

传入数组的时候，应该将**数组的首地址**计算出来，并压入栈中。在实现的时候，可以分出三种情况：参数为常数，参数为内存中的临时变量，参数为变量或者数组。

```
public void translateFuncUse(Code ic, HashMap<String, Integer>
symbolTable, String location, String funcName) {
    ...
    for (int i = 0; i < fu.getParam().size(); i++) {
        if (fu.getParam().get(i).contains("t$$")) ...
        else if (isDigital(fu.getParam().get(i))) ...
        else if (!fu.getParam().get(i).contains("$arr")) ...
        else {
            // 保存传入的一维/二维数组首地址
            if (symbolTable.get(fu.getParam().get(i)) == null) {
                la la = new la("t0", fu.getParam().get(i));
                mipsCode.add(la);
            } else {
                // 判断函数多重调用时，传入内层调用的参数是否是本函数的形式参数，如果是则
                // 不加ra，否则加ra
                if (Integer.parseInt(fu.getParam().get(i).split("\\$")[1]) >=
10000) {
                    int offset = symbolTable.get(fu.getParam().get(i));
                    lw lw = new lw("t0", offset, "sp");
                    mipsCode.add(lw);
                }
                else {
                    int offset = symbolTable.get(fu.getParam().get(i));
                    li li = new li("t0", offset);
                    mipsCode.add(li);
                    addu addu = new addu("t0", "t0", "sp");
                    mipsCode.add(addu);
                }
            }
        }
    }
}
```



```

}
/* 函数的多重调用
void fun1(int a, int b[], int c[]) {...}

void fun2(int d, int e[][2]) {
    int f[1] = {0};
    fun1(d, f, e[0])
    // 这里传入的数组f与e需要做不同的取址操作
}

*/

```

需要注意的是，如果在函数中调用另一个函数，即**函数的多重调用**，需要判断传入第二层函数的参数是否是本函数的形式参数。如果是的话，由于笔者传入的参数保存的都是绝对地址，因此传值的时候不需要再加上 \$sp 栈指针的值；如果是外层函数定义的新变量，则需要在外层函数符号表中取出其偏移量，加上当前 \$sp 后传入第二层函数。

区分一维变量与数组

这部分笔者在实现的时候，对于所有的数组变量加上了一个“\$arr”标签，在处理函数调用的时候只需要判断该条中间代码是否包含“\$arr”标签即可。

数组初始化

无论是全局数组还是局部数组，处理时都需要得到其需要的内存大小，由于数组维度的 ConstExp 被限定为可以求出值的表达式，因此必然可以实现。

```

private LinkedHashMap<String, Integer> constValues; // const name <-->
value
private ArrayList<recArray> arrayValues;

```

为了能够计算出数组的维度值，笔者在处理全局变量定义的同时，新建 ConstValue 这一 HashMap 记录常量的值，调用 calArith() 方法求得所有设计维度计算的中间代码。处理数组时，新建 recArray 内部类，保存已知的数组每个维度的值，以应对 a[b[0][0]] 此类情况。

初始化时，如果是全局变量，则使用 **.space** 为其开辟空间；如果是局部变量，则将当前函数的 **offset** 指针向后移动与数组大小相同的单位。

条件短路断路判断

if、while 的实现主要在处理语法树中的 Cond 结点。首先，在处理时，我们需要明白短路求值的原理。举个例子：

```
if (a < b == c > d && a == d || !c) {  
    // cond_0: a < b  
    // cond_1: c > d  
    // cond_2: cond_0 == cond_1  
    // cond_3: a == d  
    // cond_4: !c  
} else {  
  
}
```

这里首先需要弄清楚的是符号的优先级，< 和 > 的优先级大于 ==。因此，最好的方式仍然是采用递归下降的方式处理。其次应该明白运算的逻辑，运算的顺序应该是 cond_0 -> cond_2，在遇到 && 符号时应该判断当前条件是否为真，如果为真则继续判断 && 之后的条件，否则之间跳转至 || 后面的条件进行判断。

明白跳转的逻辑之后，可以梳理出如下中间代码：

```
cond$0 = a$0 LSS b$0  
cond$1 = c$0 GRE d$0  
cond$2 = cond$0 EQL cond$1  
beq cond$2, 0, if$0_case1  
t$$1 = a$0  
t$$2 = d$0  
cond$3 = t$$1 EQL t$$2  
beq cond$3, 0, if$0_case1  
goto if$0_begin  
if$0_case1:  
t$$0 = NOT c$0
```

```








beq t$$0, 0, if$0_end
if$0_begin:
goto else$0_end
if$0_end:
else$0_end

```

代码优化

架构设计

Diagram

 Simplifier		
	mipsCode	ArrayList<mipsCode>
	Simplifier(ArrayList<mipsCode>)	
	print()	void
	releaseMemory()	void
	simplify()	void
	simplifiedMIPS	ArrayList<mipsCode>

Simplifier类

为了实现代码优化与未优化部分的分离与解耦，这里将 InternCode2MIPS 类产出的 mipsCode 传入新的代码优化类，在这里对mips代码进行优化。

常见的优化主要有 SSA、DAG、乘除优化、常量计算、窥孔优化等。笔者由于时间有限以及其他课程的压力，代码优化部分仅完成了简单的窥孔优化。但是由于本身代码编写过程中，已经简化了一些计算流程，比如数组维度计算部分的 calArith() 方法在其他常量计算部分的套用；函数中各变量相对于 \$sp 的偏移量笔者也用了符号表管理机制进行记录，在一些取值操作中无需通过 mips 代码实现；... 在提交竞速排序作业后，即使未进行专门优化也排到了中位数，甚至后几个 testfile 排到了 40~50 名。

窥孔优化

做窥孔优化时，笔者注重于减少不必要的内存读写。窥孔的核心思想在于**对局部代码的优化**，因此需要对代码进行详细解读，判断出现的共性问题。笔者找到的是 sw 指令与 lw 指令的重复，以及寄存器传递值时反复擦写内存。

sw 与 lw

出现这种问题的代码如下：

```
sw $t0, 0($sp)
lw $t0, 0($sp)
# 或者
lw $t0, 0($sp)
sw $t0, 0($sp)
```

出现这种问题的原因在于，程序计算出一个中间变量后会将其保存至内存中，但是保存之后立刻又用到了这个值，又不得不将其从内存中取出来。主要特征为 sw 与 lw 的目标寄存器、偏移量以及 base 寄存器完全一致。因此可以直接删去第二条指令，具体操作如下：

```
for (int i = 0; i < mipsCode.size(); i++) {
    if (i == 0) {
        simplifiedMIPS.add(mipsCode.get(i));
        continue;
    }
}
```

```

// sw -> lw
if (mipsCode.get(i).getClass().getName().equals("lw") &&
mipsCode.get(i - 1).getClass().getName().equals("sw")) {
    lw lw = (lw) mipsCode.get(i);
    sw sw = (sw) mipsCode.get(i - 1);
    // sw $t0, -4($sp) -> lw $t0, -4($sp)
    if (lw.getRegDst().equals(sw.getRegDst()) && lw.getOffset() ==
sw.getOffset() && lw.getRegBase().equals(sw.getRegBase())) {
        if (i != mipsCode.size()) {
            i++;
        } else {
            break;
        }
    }
    simplifiedMIPS.add(mipsCode.get(i));
}

```

寄存器传递值

寄存器间传值问题的主要特征也是 lw 与 sw 指令，表现为除了目标寄存器外所有的内容均一致。

处理方法如下：

```

for (int i = 0; i < mipsCode.size(); i++) {
    if (i == 0) {
        simplifiedMIPS.add(mipsCode.get(i));
        continue;
    }
    // sw -> lw
    if (mipsCode.get(i).getClass().getName().equals("lw") &&
mipsCode.get(i - 1).getClass().getName().equals("sw")) {
        lw lw = (lw) mipsCode.get(i);
        sw sw = (sw) mipsCode.get(i - 1);
        // sw $t0, -4($sp) -> lw $t1, -4($sp)
        if (lw.getOffset() == sw.getOffset() &&
lw.getRegBase().equals(sw.getRegBase())) {

```

```
        move move = new move(lw.getRegDst(), sw.getRegDst());
        simplifiedMIPS.add(move);
        if (i != mipsCode.size()) {
            i++;
        } else {
            break;
        }
    }
    simplifiedMIPS.add(mipsCode.get(i));
}
```

写在最后

至此，笔者的 sysY 语言编译器就基本完成了。总结多次小实验的设计经验，首先是要**架构明确**，其次是**勇敢试错**。架构是程序之本，在好的架构基础上才能高效完成开发，笔者的设计仍有多处不完美之处，仍有很多的优化空间，相信在未来的不断实践中这些问题会得以解决。

感谢 **19373528 曲乐皓** 对于函数调用、函数传参部分的指导，

感谢 **19373427 刘博一** 对于代码优化部分的文献分享（虽然笔者没有来得及实现），

感谢 **19373630 曾育群** 对于笔者理解编译基本原理的帮助，

感谢 **19373185 周子航** 与笔者分享 Pcode 相关原理及其独创的中间代码 Alxa Code，

感谢 **19373529 朱祖坤** 给予笔者的部分设计灵感以及在笔者承担巨大压力时的关怀。

感谢 **李沛熙 助教**对于笔者在代码生成部分 debug 的慷慨相助，在困境中永远可以相信前辈们。

感谢 **胡春明 老师**对于编译原理理论课的精彩讲解，理论永远是实验的基石。

献给所有爱我的人与我爱的人。