~~Course Design of Compiling.

Autumn 2022. BUAA.

author:szy

编译器设计文档

前言

参考编译器介绍

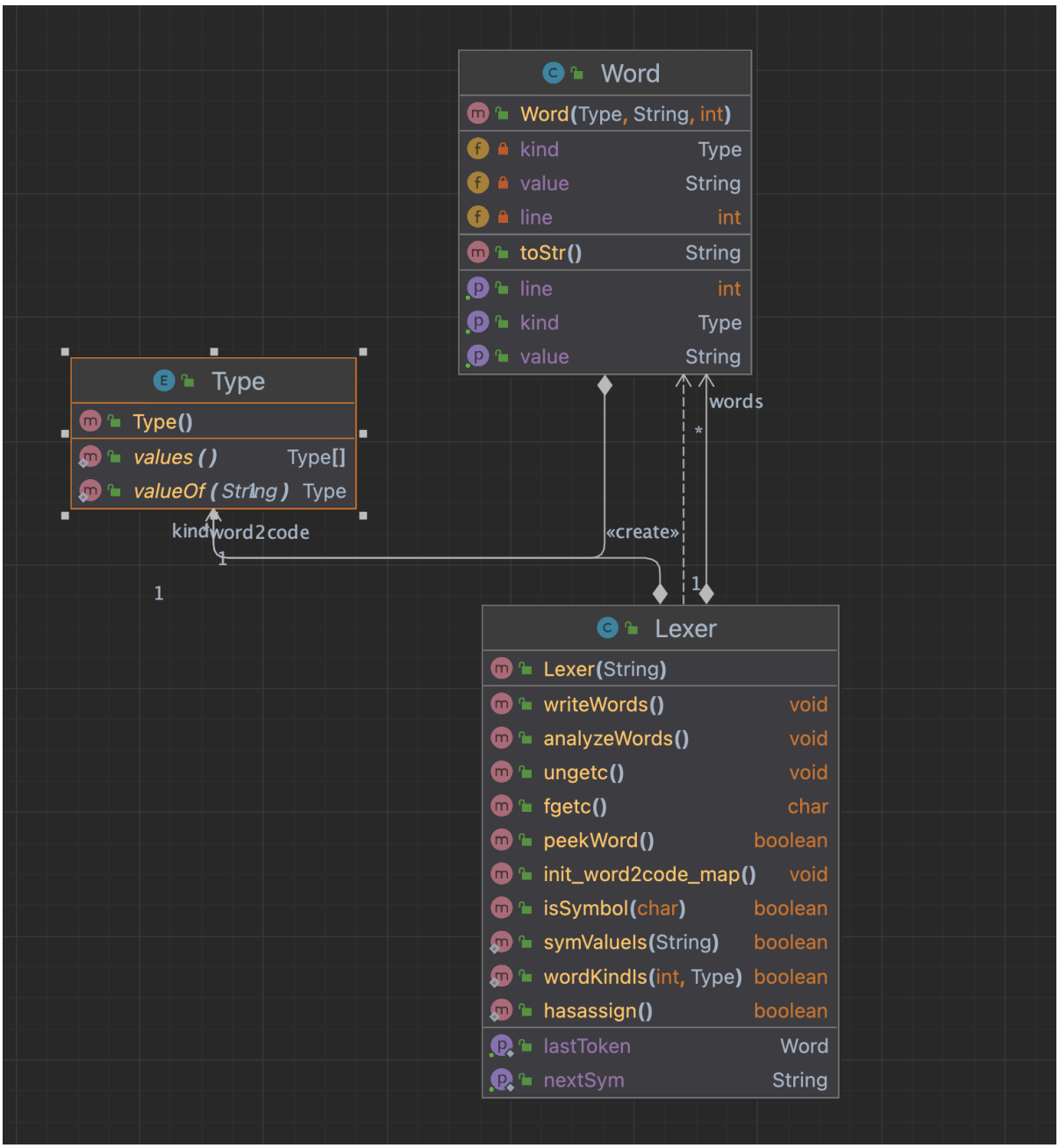
本人主要参考的是PL0编译器，该编译器是课程组下发的，生成的目标代码是Pcode，我在进行词法分析的时候参考了该编译器，获得的一定的思路，他的词法分析程序是getsym，这个程序不断读取最小元token来进行词法分析，读完这个程序之后我便顺利的写出了词法分析，但由于之后我要写的目标代码是mips，所以从语法分析之后我没有再参考该编译器，而且Pascal语言读起来也很不流畅，所以之后的代码设计我基本上是与同学们讨论和自己构想出的。

编译器总体设计

编译器在编写完毕后主要由词法分析器Lexer，语法分析器Grammer，错误处理管理器Errorchecker，以及中间代码生成器Midcodegenerator，目标代码生成器Mipsgenerator五部分构成。五部分基本是顺序递进，词法分析器的分析结果将tokens交给语法分析器，语法分析器进行语法分析同时使用错误处理管理器进行错误识别，语法分析器同时建立AST语法树，将AST语法树交给中间代码生成器产生midcodes，midcodes最终被传入目标代码生成器，生成mips码。

1.词法分析

1.1 架构设计



1.1.1 Type Class

该类为枚举类,枚举类主要枚举了所有要识别的单词名称对应的类别码（包括下表中的所有类别码）。之所以使用枚举类，一方面可以统一的记录所有的类别码类型，我们在Word类中记录单词对应类别码时，可以直接调用已经固定好的Type内的所有名称类别，避免我们自己写类别码String时打错，造成低级Bug。另一方面，使用枚举类更可以大大增强代码的可读性。

单词名称	类别码	单词名称	类别码	单词名称	类别码	单词名称	类别码
Ident	IDENFR	!	NOT	*	MULT	=	ASSIGN
IntConst	INTCON	&&	AND	/	DIV	;	SEMICN
FormatString	STRCON	||	OR	%	MOD	,	COMMA
main	MAINTK	while	WHILETK	<	LSS	(	LPARENT
const	CONSTTK	getint	GETINTTK	<=	LEQ	)	RPARENT
int	INTTK	printf	PRINTFTK	>	GRE	[	LBRACK
break	BREAKTK	return	RETURNTK	>=	GEQ	]	RBRACK
continue	CONTINUETK	+	PLUS	==	EQL	{	LBRACE
if	IFTK	-	MINU	!=	NEQ	}	RBRACE
else	ELSETK	void	VOIDTK				

- Word类

该类记录单词的String值，单词类别码（Type类），line行数（行数是在错误处理的时候加的，因为错误处理时需要输出错误所在的行数）。

- Lexer类

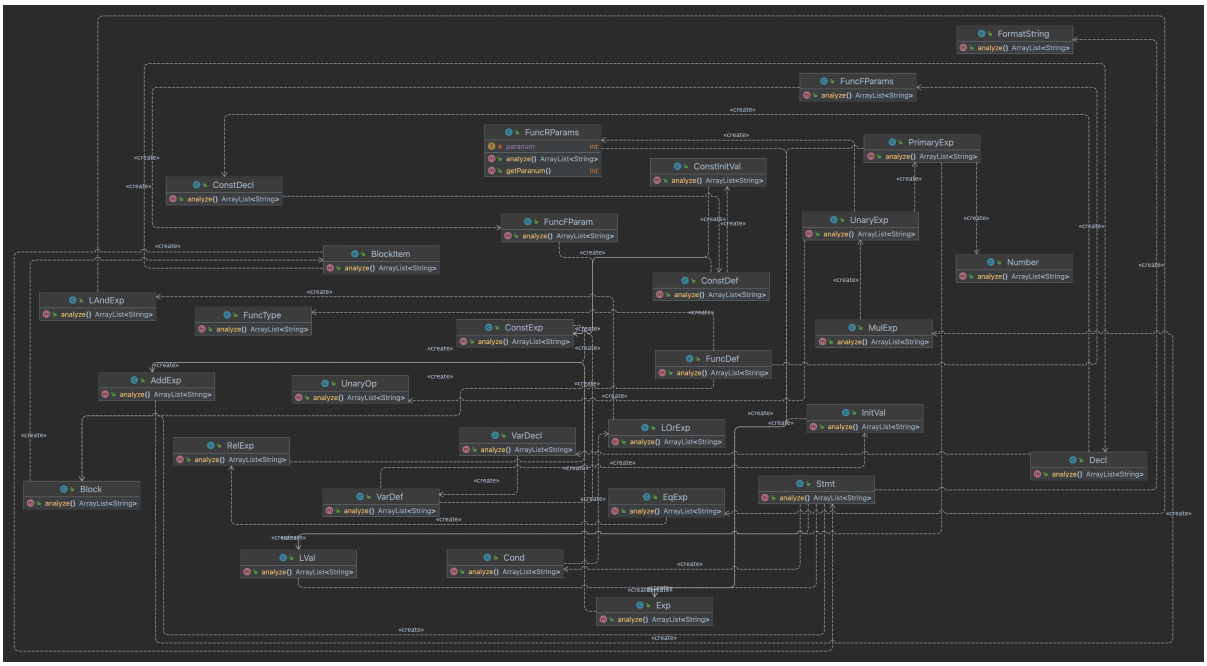
词法解析器类，该类实现词法解析的主要功能，主要采用流式解析的方式，分为数字，字母，操作符（特殊字符）三类进行解析。主要分为以下三步

1. 读取文件中的所有字符，组成一个字符ArrayList，在ArrayList最后引入特殊符号～代表文件结束（EOF）。
2. 检测到符合词法的单词后，为每个单词生成Word类并存入wordList中；之后继续读取、处理后续字符。检测合法单词的顺序如下
    - 跳过空白符号，记得遇到\n换行
    - 如果遇到～符号，即终止符号，停止循环
    - 检测到字母或”_“开头,说明是Ident或者字符串类型的保留字(通过word2code是否含有该key来区分)
    - 检测到引号“，处理Formatstring
    - 检测到数字打头，处理数字
    - 检测到字符`!&|+-*/%<>=;,(){}`任一
        - 是`/`,是两种注释或者是除号，处理注释的时候记得换行时，保持line更新
        - 是`&`,检测到`&&`
        - 是`|`，检测到`||`
        - 是`<>=!`，看看后面还有没有`=`，分两种情况处理
        - 其他的`+ - * % ; , ( ) [ ] { }`没有特殊情况，就一个单字符，一起处理掉
    - 循环重复上述过程
3. 额外的，在处理过程中如果遇到"\n"换行符，那么在Lexer中记录行号line，在创建word的时候可以该单词所处的line也记录到word类中。

1.2 设计修改

- 在进行语法分析设计时，我们把当前单词的提取，回退，类别码判断，单词值判断等功能方法全都交给了词法分析器Lexer执行，这些功能与语法分析器相分离，更好的降低了耦合度。在进行错误处理时，由于我们需要输出行数，因此在Word类中新增了line属性，用于记录每个单词所在行数。
- 初始的设计存在一个小Bug，即遇到`/*/`注释开头的时候，我的程序会过早的结束对于注释的读取，这与我的注释读入循环的终止思路有一定关系，在语法分析的时候进行了修复。

2. 语法分析



2.1架构设计

总体下降采用递归下降的方法，从UML类图可以看到，分别给每一个语法成分编写了其对应的analyze解析函数，只需要根据文法进行对应的语法成分解析函数调用即可，思路非常简单。但问题在于，使用递归下降的方法解析意味着文法中不能出现左递归的情况存在，因此我们对文法进行了一定的改写，消除了左递归，经过观察我们发现左递归全部出现在表达式的部分文法中，只需要对这部分文法进行修改即可。此外在进行架构设计的时候，还出现了其他几个问题，统一总结如下。

2.1.1 文法左递归问题

我们利用理论课的两条规则消除左递归。

> 规则一（提因子）:U::=xy|xw|...|xz ⇒U::=x (y|w|...|z)
>
> 规则二：U::=x|y|...|z|Uv ⇒U::=(x|y|...|z) {v}

修改前

```
<MulExp>        := <UnaryExp> | <MulExp> ( '*' | '/' | '%' ) <UnaryExp>
<AddExp>        := <MulExp> | <AddExp> ( '+' | '-' ) <MulExp>
<RelExp>        := <AddExp> | <RelExp> ( '<' | '>' | '<=' | '>=' ) <AddExp>
<EqExp>         := <RelExp> | <EqExp> ( '==' | '!=' ) <RelExp>
<LAndExp>       := <EqExp> | <LAndExp> '&&' <EqExp>
<LOrExp>        := <LAndExp> | <LOrExp> '||' <LAndExp>
```

修改后

```
<MulExp>        := <UnaryExp> { ('*' | '/' | '%') <UnaryExp> }
<AddExp>        := <MulExp> { ('+' | '-') <MulExp> }
<RelExp>        := <AddExp> { ('<' | '>' | '<=' | '>=') <AddExp> }
<EqExp>         := <RelExp> { ('==' | '!=') <RelExp> }
<LAndExp>       := <EqExp> { '&&' <EqExp> }
<LOrExp>        := <LAndExp> { '||' <LAndExp> }
```

其余文法与原来保持不变。还有一个注意的的要点在于，我们解析的时候虽然改写了文法，但是在输出的时候还需要跟原来的文法保持一致。比如我们改写了MulExp，正常来说我们只需要在最后加`add("<MulExp>");`,把输出加进去，但是这并不是原文法的输出，我们需要在（1）出也加上，才是原文法的输出。

```
public class MulExp extends GrammarElement {
    @Override
    public ArrayList<String> analyze(){
        addAll(new UnaryExp().analyze());
        while (Lexer.symValueIs("*") || Lexer.symValueIs("%")
        || Lexer.symValueIs("/")){
            add("<MulExp>"); // (1) .是上一个的语法父节点，这里也要加
            add(Lexer.getNextSym()); //* / %
            addAll(new UnaryExp().analyze());
        }
        add("<MulExp>");
        return sublist;
    }
}
```

2.2.2 向前看问题

在文法中遇到"|","{}","[]"，就会存在向前看问题，因为这里存在不确定性，我们需要向前看来判断这里到底是来确定到底走哪个分支，或者这里到底是否存在这个非终结符。

2.2.3 Stmt解析中Lval开头还是Exp开头

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

这里我们通过查看分号前有没有等号出现，如果Lval开头则必然会有一个等号，如果不是Lval开头，以Exp开头，Exp内部不存在等号。

```
public static boolean hasassign() {    //查找分号前有无等号
        int offset = 1;
        while (index + offset < words.size()) {
            Word newsym = words.get(index + offset);
            if (newsym.getValue().equals(";")){
                break;
            } else if (newsym.getValue().equals("=")) {
                return true;
            }
            offset += 1;
        }
        return false;
    }
```

2.2.4 递归下降架构的设计

设计递归下降时，为了程序的架构更加明晰，我将每个左部的非终结符都建立了类，这些类都继承一个父类GrammarElement。该父类包含sublist数组属性，这个数组包含该非终结符根据文法推出的所有终结符以及非终结符字符串，用于记录和输出。我们使用add和addAll方法向这个数组里添加元素，在递归调用的过程中，我们不断的add，addAll，便可以将sublist补充完毕。每一个非终结符都有sublist，这也是为什么我要创建这个父类的原因，减少代码冗余，最后CompUnit里面的字符串元素就是我们要输出的结果。同时该父类有一个analyze方法，没个非终结符子类需要重写该方法，以实现每个非终结符的解析。

```
public class GrammarElement {
    protected ArrayList<String> sublist;

    public GrammarElement(){
        sublist = new ArrayList<>();
    }
    public ArrayList<String> analyze(){
```

```
            return null;
        }
        public void add(String s){
            sublist.add(s);
        }
        public void addAll(ArrayList<String> ss){
            sublist.addAll(ss);
        }

        public void error() {
            System.err.println("Error!");
        }
    }
```

2.2 设计修改

与设计时最初的架构基本无变化。

3. 错误处理

3.1 架构设计

下表为课程组给出的a到m这几种错误类型，包含了类别码、解释与文法中可能存在的地方，下文将对每一错误类别介绍具体错误检查的设计方案。

错误类型	错误类别码	解释	对应文法及出错符号(...省略该条规则后续部分)
非法符号	a	格式字符串中出现非法字符报错行号为所在行数。	→ “”{}””
名字重定义	b	函数名或者变量名在**当前作用域**下重复定义。注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。报错行号为所在行数。	→ ...→ ... | ... → ... → ...
未定义的名字	c	使用了未定义的标识符报错行号为所在行数。	→ ...→ ...
函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。报错行号为函数调用语句的**函数名**所在行数。	→‘(’[FuncRParams]‘)’
函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。报错行号为函数调用语句的**函数名**所在行数。	→‘(’[FuncRParams]‘)’
无返回值的函数存在不匹配的return语句	f	报错行号为**‘return’**所在行号。	→‘return’ {‘[’Exp‘]’}‘;’
有返回值的函数缺少return语句	g	只需要考虑函数末尾是否存在return语句，**无需考虑数据流**。报错行号为函数**结尾的****’}’**所在行号。	FuncDef → FuncType Ident ‘(’ [FuncFParams] ‘)’ BlockMainFuncDef → 'int' 'main' '(' ')' Block
不能改变常量的值	h	为常量时，不能对其修改。报错行号为所在行号。	→‘=’ ‘;’|‘=’ ‘getint’ ‘(’ ‘)’ ‘;’
缺少分号	i	报错行号为分号**前一个非终结符**所在行号。	,及中的';’
缺少右小括号’)’	j	报错行号为右小括号**前一个非终结符**所在行号。	函数调用()、函数定义()及中的')’
缺少右中括号’]’	k	报错行号为右中括号**前一个非终结符**所在行号。	数组定义(,,)和使用()中的']’
printf中格式字符与表达式个数不匹配	l	报错行号为**‘printf’**所在行号。	Stmt →‘printf’‘(’FormatString{,Exp}’)’‘;’
在非循环块中使用break和continue语句	m	报错行号为**‘break’**与‘**continue’**所在行号。	→‘break’‘;’|‘continue’‘;’

分别处理a-m问题

- A

- 问题:格式字符串中出现非法字符报错行号为FormatString所在行数
  解决：formatstring类中直接进行检查

```
public static void checkA(String formatstring){ //去除引号传进来
    int index = 0;
    while (index < formatstring.length()) {
        int a = formatstring.charAt(index);
        if (a == 32 || a == 33 || (a >= 40 && a <= 126 && a != 92)) {
            index += 1;
        } else if (a == 37) {
            if (index + 1 < formatstring.length() &&
formatstring.charAt(index + 1) == 'd') {
                index += 2;
            } else {
                add(new Error(Lexer.sym,"a"));
                break;
            }
        } else if (a == 92) {
            if (index + 1 < formatstring.length() &&
formatstring.charAt(index + 1) == 'n') {
                index += 2;
            } else {
                add(new Error(Lexer.sym,"a"));
                break;
            }
        } else {
            add(new Error(Lexer.sym,"a"));
            break;
        }
    }
}
```

- BC

名字重定义或未定义的情况：都是关于定义相关错误，处理方式为建立符号表，对于B类重定义的情况，每次调用该标识符时检查在当前作用域下是否已定义具有相同名字的函数名或变量名,对于C类错误，则检查表中所有数据检测是否有未定义的情况。

```
public static boolean checkB(Symbol asymbol){
        for (Symbol symbol : Parser.table.getSymbols()) {
            //flag2 = symbol.getTableType().equals(TableType.FUNC)? 1:0;
            if(symbol.getName().equals(asymbol.getName())){ //&& flag1 ==
flag2
                //add error
                add(new Error(asymbol.getWord(),"b"));
                return true;
            }
        }
```

```
        return false;
    }

    public static Symbol checkC(Word w,TableType tableType){ //未定义
        Symbol checkresult = null;
        //System.out.println("check "+w.getValue()+"
"+tableType.toString());
        if(!tableType.equals(TableType.FUNC)){
            checkresult = lookallTableFindsamename(w);
        }else {
            checkresult = lookGlobalTAbleFindSamename(w);
        }

        if( checkresult == null){
            add(new Error(w,"c"));
        }
        return checkresult;
    }
```

- DE
- 对于d类错误及函数参数个数不匹配的情况，分两个部分进行处理，首先在函数定义时为函数记录下该函数名拥有的参数以及其类型、个数等特征，之后在函数调用时对参数个数进行计数，若不匹配则抛出D类错误。
- 对于e类函数参数类型不匹配的情况，在每一个函数参数进行检查时标记该参数的维度，由于仅有二维数组、一维数组、整数，以及void等类型的情况，且仅需考虑数据的维度是否匹配，因此分别将这些参数的维度定义为2、1、0以及任意负数，检测函数对应位置参数的维度与此时传入参数的维度是否相同，若不相同则抛出e类错误

```
public static boolean checkD(Word w,Symbol func,int paramnum){
    int shouldhave = func.getParamsLen();
    //System.out.println("should have param num:"+shouldhave);
    if(shouldhave != paramnum){
        add(new Error(w,"d"));
    }
    return shouldhave != paramnum;
}

public static void checkE(Word w,Symbol symbol,ArrayList<Integer>
dimensions){ //dimensions是调用时的实际dimension
    ArrayList<Integer> rightDimensions = symbol.getRightParamsDimentions();
    for(int i = 0;i<dimensions.size();i++){
        if(!dimensions.get(i).equals(rightDimensions.get(i))){
            add(new Error(w,"e"));
            break;
        }
    }
}
```

- F
- f类错误为无返回值的函数存在不匹配的return语句，报错的行号为return所在行数。无返回值的函数即为void的类型的函数，由于允许return后面直接加分号的形式存在，因此当检测到了return时不能直接认定存返回值，同时还要检查return后面是否有Exp()类型的表达式语句，若无表达式语句，可以认为仍然是存在一个无返回值的返回语句，并不予报错，若一直到void函数末尾均未检测到return类型，则符合无返回值函数的要求。

```
public static void checkF(Word w){ //无返回值函数存在return exp; 报错报到return
行
    if( Parser.funckind.equals(FuncKind.VOID) && Parser.blockhasReturn){
        add(new Error(w,"f"));
    }
}
```

- G
- g类错误则为有返回值的函数缺少return语句，报错的行号为结尾右大括号所在行数。对于有返回值的函数，由于对于if、else、while等结构体内嵌套的返回值均不予承认，因此需要检测的是函数block结构体内最后一句是否为return类型的语句，因此需要设立布尔变量来判断函数最后一句的语句类型，若最后一句不为return语句或return语句中无表达式，则认为此时发生该类错误，该函数无返回值。

```
public static void checkG(Word w){ // 判断有返回值的最后一句是不是return exp;
    if(Parser.funckind.equals(FuncKind.INT) && !Parser.lastIsReturn){
        add(new Error(w,"g"));
    }
}
```

- H
- h类错误为修改了常量的值：检测方式为每当对LVal进行赋值时，检查是否该标识符为Const的常量，不可改变值，若符合，则报错。

```
public static void checkH(Word w){
    Symbol symbol = lookallTableFindsamename(w);
    if(symbol == null){
        System.out.println("no define but need to check h?????????????");
        return;
    }
    if(symbol.getConstType().equals(ConstType.CONST)){
        add(new Error(w,"h"));
    }
}
```

- IJK
- i、j、k类错误，为缺少分号小括号中括号的情况，对于这几类情况在语法分析过程中可顺便处理。当检测到缺少的符号为这几种符号的情况，可直接抛出对应类型的错误及错误码，从而完成错误检测，此处用的是match函数，match函数中处理逻辑如下：

```
public boolean match(String s){
    if (!Lexer.symValueIs(s)) {
        switch (s) {
            case ";":
                ErrorChecker.add(new Error(Lexer.getLastToken(),"i"));
                break;
            case ")":
                ErrorChecker.add(new Error(Lexer.getLastToken(),"j"));
                break;
            case "]":
                ErrorChecker.add(new Error(Lexer.getLastToken(),"k"));
                break;
            default:
                break;
        }
        return false;
    }
    add(Lexer.getNextSym());
    return true;
}
```

- L
- L类错误与a类错误类似，均是对字符串检查，对应的检查逻辑为在检查过程中分别计数，print语句中有多少个应当输出的参数，以及逗号后面实际传入参数的个数检查，二者不匹配时抛出l类错误

```
public static void checkL(Word printf,String formatstr,int num){
    int shouldhavenum = formatstr.split("%d").length-1;
    if(shouldhavenum != num){
        add(new Error(printf,"l"));
    }
}
```

- M
- m类错误为非循环体内具有break与continue语句，对于这两种情况，则需要设置一个全局变量存储此时是否为函数循环函数体，若在循环函数体内调用break给continue依据则认为是正常逻辑，否则抛出错误。此处设置了一个cycleDepth参数，标识深入的循环结构层数，当进入一个循环体时计数+1，退出时计数-1，为0则说明此时未在循环体内

```
public static void checkM(Word w){
    if(Parser.intoWhile == 0){
        add(new Error(w,"m"));
    }
}
```

3.2 设计修改

- 开始设计符号表的时候考虑不周，漏掉了很多重要的属性，这些属性在之后生成中间代码的时候非常重要，比如说数组维度等，因此这一部分的符号表相关代码在之后代码生成部分做了进一步完善，记录了更多的信息。

4. 代码生成

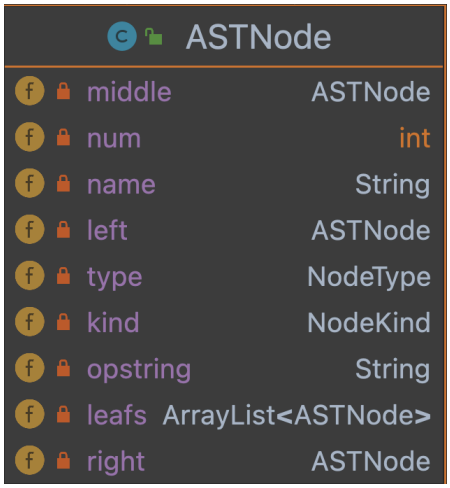
4.1 语法树AST

将建立AST的过程融入到了语法分析的过程中，在第一遍语法分析的过程中就将AST建立，AST树采用三叉树的方式记录。

- 所有节点的类型如下

```
public enum NodeType {
    1 usage
    CompUnit,Decl,Func,FuncDef,FuncFParams,Block,BlockItem_Decl,
    2 usages
    BlockItem_Stmt,VarDecl,VarDef,ConstDecl,ConstDef,InitVal,ConstInitVal,
    1 usage
    FuncRParams,IfStatement,While,Break,Continue,Return,Printf,ExpList,
    2 usages
    Assign_getint,Assign_value,Ident,Exp,Number,FormatString,Getint,OP,OR,AND
}
```

- ASTNode类，记录节点类型，节点名称，节点数字，左子树，右子树，中子树，叶子节点等信息。



4.2 中间代码设计

4.2.1 函数相关

与课程组给出的中间代码推荐格式基本相同

函数声明

```
int foo(int a, int b) {
  // ...
}
```

```
int foo()
para int a
para int b
```

函数调用

```
i = tar(x, y);
```

```
push x
push y
call tar
i = RET
```

函数返回

```
return x + y;
```

```
t1 = x + y
ret t1
```

4.2.2 变量常量声明

- 常量没有显式输出常量数值，但数值计算并存储在了在符号表中
- 变量也没有显式输出变量初始化值，而是拆解为了一条定义中间代码与赋值中间代码，如下：

```
int a = 1;
const int b = 2;
```

```
INT a
a = 1
CONSTINT b
```

4.2.3 分支和跳转

- 分支部分难点在于处理短路求值，部分逻辑的核心代码如下：

```
if (type.equals(NodeType.AND)) {
    parseCond(n.getLeft(), jinlabel, joutlabel);
} else {
    String orLabel = "orLabel_" + orcnt;
    orcnt += 1;

    parseCond(n.getLeft(), jinlabel, orLabel);
    createMidCode(MidType.jump, jinlabel);

    createMidCode(MidType.Label, orLabel);
}
parseCond(n.getRight(), jinlabel, joutlabel);
```

生成的中间代码如下

```
if (a || !a || a == 1 && b <= b >= a == a != 2) {
    // ...
}
```

```
beq a, 0, orLabel_1
j into_if1
orLabel_1:
bne a, 0, orLabel_0
j into_if1
orLabel_0:
bne a, 1, end_if1
sle #tmp1, b, b
sge #tmp2, #tmp1, a
seq #tmp3, #tmp2, a
beq #tmp3, 2, end_if1
into_if1:
#Out Block
end_if1:
```

- 跳转部分难点在于处理break和continue
- 对每一处if或while导致的分支跳转，拆分出了begin、into、end三部分，分别对应含判断的Cond开始前、进入Stmt块、分支跳转结束三个位置，用于满足不同情况下的跳转

```
while (1) {
    if (1) {
        break;
    }
    if(0){
        continue;
    }
}
```

```
begin_loop1:
beq 1, 0, end_loop1
intostmt_loop1:
beq 1, 0, end_if1
into_if1:
#Out Block While
j end_loop1
#Out Block
end_if1:
beq 0, 0, end_if2
into_if2:
#Out Block While
j begin_loop1
#Out Block
end_if2:
#Out Block
j begin_loop1
end_loop1:
```

4.2.4 数组处理

- 数组定义

```
int a[3][4] = {{1, 2, 3, 4},{0,0,0,0}, {0,0,0,0}};
```

```
array int a[3][4]
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
a[4] = 0
a[5] = 0
a[6] = 0
a[7] = 0
a[8] = 0
a[9] = 0
a[10] = 0
a[11] = 0
```

- 数组存取

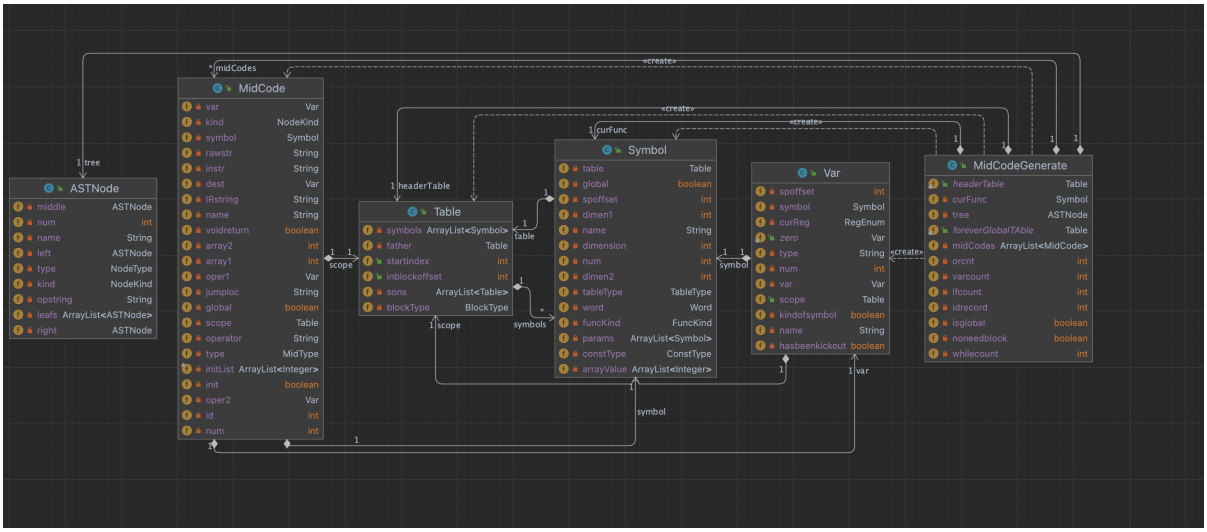
因为之前符号表，语法树处理二维数组时都已经直接把数组处理为了一维数组，所以数组存取时需要直接酸楚数组下标，并适当利用临时变量进行存取。

```
a[2][0] = -1;
z = a[2][0];
```

```
a[8] = -1
#tmp1 = a[8]
z = #tmp1
```

4.3 中间代码生成

- 代码生成主要类及其依赖关系



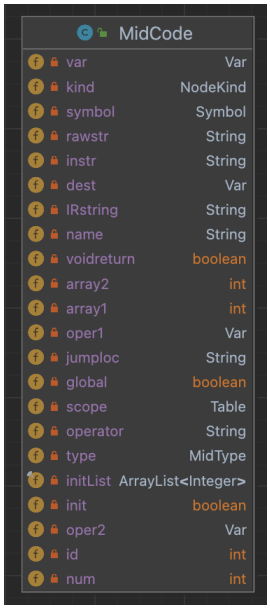
4.3.1 主要类描述

- **中间代码类型枚举类**

```
public enum MidType {
    15 usages
    Note,Label,call,Return,Printf,Push,Getint,assign_ret,assign,assign2,
    6 usages
    intDecl,arrayDecl,funcDecl,funcPara,jump,branch,setcmp
}
```

中间代码一共分为16类，分别对应每一类主要的中间代码语句，例如注释，标签，函数调用，声明等等类型的中间代码类型，这些类型在目标代码生成的时候将分别对应进行生成。

- **中间代码类**

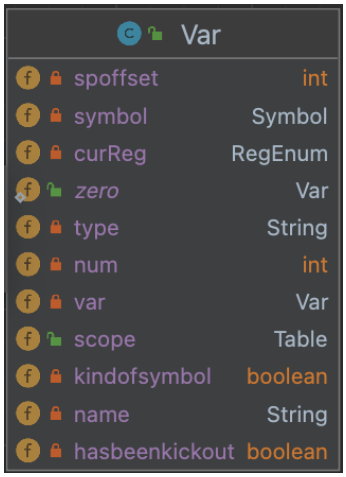


中间代码包装为了MidCode类，内部记录了中间代码大量信息，主要包括

- 中间代码类型
- 中间代码包括的符号表符号
- 数组维度
- 数组变量下表
- 数值
- 表达式的操作符
- 表达式的操作数

等等这些信息，每一类的中间代码会用到不同的信息，这些信息在目标代码生成的时候都将得到利用。

- **变量类**



变量类主要是由于生成中间代码的过程中需要递归处理嵌套的表达式，因此为统一函数传参类型与返回值，设立了Var变量，该变量会在多种类型的中间代码中用到。

4.3.2 中间代码核心实现代码

- 变量常量声明及初始化

```
public void parseDef(ASTNode n) {
    ASTNode ident = n.getLeft();
    String name = ident.getName();
    NodeKind kind = ident.getKind();

    //符号表
    ASTNode init = n.getRight();
    Symbol symbol = insertTosymbolTable(ident, init, false, false);


    //数组
    if (kind.equals(NodeKind.CONSTARRAY) || kind.equals(NodeKind.ARRAY)) {
        parseArrayDef(n);
    } else {
        //symbol.setIrindex(midCodes.size());//todo 干啥的
        //const int,int
        if (init != null) {
            if (kind.equals(NodeKind.CONSTINT) || isglobal) { //isglobal 也
直接计算初值，因为这块不可能有函数
                int constinitnum = symbol.getNum();
                MidCode ir = new MidCode(MidType.intDecl, kind, name);
                ir.setNum(constinitnum); //初值
                ir.setInitIsTrue();
                ir.setSymbol(symbol);
                ir4init(ir);
            } else { //如果是var int ，需要计算表达式，采取一个生成两条的策略
                //第一条声明
                MidCode numInitir = new MidCode(MidType.intDecl, kind,
name);
                numInitir.setSymbol(symbol);
                ir4init(numInitir);

                //第二条赋值
                //startindex = midCodes.size(); //初始化复制语句开始位置 todo
干啥的

                Var intinitVar = parseExp(init);
                Var intinitLval = new Var("var", name);

                intinitLval.setSymbol(symbol);
                intinitLval.setiskindofsymbolTrue();//todo 这个干啥的

                //checkIfAssignGlobalValue(intinitVar);//todo 真的感觉这个没啥
用，不用不生成也没提高性能啊
```

```
                MidCode ir = new MidCode(MidType.assign2, intinitLval,
intinitVar);
                ir4init(ir);

            }
        } else { //没有初值，可以直接构建
            MidCode ir = new MidCode(MidType.intDecl, kind, name);
            ir.setSymbol(symbol);
            ir4init(ir);
        }
    }
}
```

- 数组声明及初始化

```
public void parseArrayDef(ASTNode n) {

    ASTNode ident = n.getLeft();
    String name = ident.getName();
    Symbol symbol = lookallTableFindsamename(name);
    assert symbol != null;
    NodeKind kind = ident.getKind();

    //数组identnode 左右节点分别为dimen1，dimen2，
    //在写语法分析的symbol里面当时并没有设置这两个值，需要在这里算一下两个维度
    ASTNode dimen1Node = ident.getLeft();
    ASTNode dimen2Node = ident.getRight();
    int dimen1num = dimen1Node.calcuValue();
    int dimen2num = 0;
    if (dimen2Node != null) {
        dimen2num = dimen2Node.calcuValue();
    }

    MidCode midCode = new MidCode(MidType.arrayDecl, name, dimen1num,
dimen2num);
    midCode.setSymbol(symbol);

    //开始设置初值，如果是const array 或者global我们直接存储到中间代码arraylist里面
    //如果是int类型数组，我们需要生成多条assign2语句，因为这里可能不能直接得到初值，需
要运行时计算
    ASTNode init = n.getRight();

    //没初值，直接打包返回
    if (init == null) {
        ir4init(midCode);
        return;
    }
```

```
    midCode.setInitIsTrue();
    if (symbol.isConst() || symbol.isGlobal()) {
        //这里直接把初值存到midCode，方便生成

        parseArrayInitNums(midCode, symbol, name, init, dimen1num,
dimen2num, true);
        ir4init(midCode);
    } else {
        ir4init(midCode);
        parseArrayInitNums(midCode, symbol, name, init, dimen1num,
dimen2num, false);
    }
}
```

- if与while

```
public void parseIfStatement(ASTNode n, int localwhilecount) {
    ifcount += 1;

    String end_ifLabel = "end_if" + ifcount;
    String end_elseLabel = "end_else" + ifcount;
    String intoblocklabel = "into_if" + ifcount;      //主要用于短路求值直接跳
入

    parseCond(n.getLeft(), intoblocklabel, end_ifLabel);

    //进入If基本块范围，打标签，建立表
    createMidCode(MidType.Label, intoblocklabel);
    MidCodeGenerate.openTable(BlockType.IF);
    noneedblock = true;

    //if stmt
    parseStmt(n.getMiddle(), localwhilecount);
    createMidCode(MidType.Note, "#Out Block");

    //else stmt
    if (!n.haselse()) { //如果没有else
        MidCodeGenerate.closeTable();
        noneedblock = false;
        createMidCode(MidType.Label, end_ifLabel);
    } else {
        createMidCode(MidType.jump, end_elseLabel);
        MidCodeGenerate.closeTable();
        noneedblock = false;
        createMidCode(MidType.Label, end_ifLabel);

        //开始解析else
        noneedblock = true;
```

```
        MidCodeGenerate.openTable(BlockType.ELSE);
        parseStmt(n.getRight(), localwhilecount);
        createMidCode(MidType.Note, "#Out Block");
        MidCodeGenerate.closeTable();
        noneedblock = false;

        createMidCode(MidType.Label, end_elseLabel);
    }

}
```

```
public void parseWhileLoop(ASTNode n) {
    //将whilecount本地化，防止while嵌套，生成break，continue的时候没法找到原来的标签
    int localwhilecnt = whilecount;


    String beginloop_label = "begin_loop" + whilecount;
    String endloop_label = "end_loop" + whilecount;
    String intoloop_label = "intostmt_loop" + whilecount;
    createMidCode(MidType.Label, beginloop_label);

    whilecount += 1;

    parseCond(n.getLeft(), intoloop_label, endloop_label);

    //进入while基本块
    MidCodeGenerate.openTable(BlockType.WHILE);
    noneedblock = true;
    createMidCode(MidType.Label, intoloop_label);

    parseStmt(n.getRight(), localwhilecnt);

    createMidCode(MidType.Note, "#Out Block");
    createMidCode(MidType.jump, beginloop_label);
    MidCodeGenerate.closeTable();
    noneedblock = false;
    createMidCode(MidType.Label, endloop_label);

}
```

- if和while语句的cond

```
public void parseCond(ASTNode n, String jinlabel, String joutlabel) {

    NodeType type = n.getType();
    String opstring = n.getOpstring();

    // > < >= <=
```

```
    if (OperDiction.isnumcmp(opstring)) {

        Var leftexp = parseRelExp(n.getLeft());
        Var rightexp = parseRelExp(n.getRight());

        if (opstring.equals(">=")) {
            createMidCode(MidType.branch, "blt", joutlabel, leftexp,
rightexp);
        } else if (opstring.equals("<=")) {
            createMidCode(MidType.branch, "bgt", joutlabel, leftexp,
rightexp);
        } else if (opstring.equals(">")) {
            createMidCode(MidType.branch, "ble", joutlabel, leftexp,
rightexp);
        } else {
            createMidCode(MidType.branch, "bge", joutlabel, leftexp,
rightexp);
        }
    } else if (OperDiction.iseqcmp(opstring)) {

        Var lefteq = parseEqExp(n.getLeft());
        Var righteq = parseEqExp(n.getRight());
        createMidCode(MidType.branch, opstring.equals("!=") ? "beq" :
"bne", joutlabel, lefteq, righteq);
    } else if (type.equals(NodeType.OR) || type.equals(NodeType.AND)) {
        //重点在于短路求值怎么处理
        if (type.equals(NodeType.AND)) {
            parseCond(n.getLeft(), jinlabel, joutlabel);
        } else {
            String orLabel = "orLabel_" + orcnt;
            orcnt += 1;

            parseCond(n.getLeft(), jinlabel, orLabel);
            createMidCode(MidType.jump, jinlabel);

            createMidCode(MidType.Label, orLabel);
        }
        parseCond(n.getRight(), jinlabel, joutlabel);
    } else {
        if (opstring.equals("!")) {
            Var notexp = parseExp(n.getLeft());
            createMidCode(MidType.branch, "bne", joutlabel, notexp, new
Var("num", 0));
        } else {
            Var exp = parseExp(n);
            createMidCode(MidType.branch, "beq", joutlabel, exp, new
Var("num", 0));
        }
    }
```

```
}
```

- printf
- 该部分处理核心是将原始字符串按照%d分割为了多条子串，%d的部分则调取表达式的值，其余部分按原样输出字符串内容，这些直接输出的字符串之后会放到mips的data段

```
private void parsePrintf(ASTNode n) {
    String formatString = n.getLeft().getName();
    formatString = formatString.substring(1, formatString.length() - 1);


    createMidCode(MidType.Note, "#Start Print");

    if (n.getRight() != null) {
        String[] splits = formatString.split("%d", -1);
        ASTNode explist = n.getRight();
        int i = 0;
        while (i < splits.length) {
            String splitstr = splits[i];
            if (!splitstr.equals("")) {
                Var var_splitstr = new Var("str", splitstr);
                createMidCode(MidType.Printf, var_splitstr);
            }
            //防止越界
            if (explist.getLeafs() == null || i > explist.getLeafs().size()
- 1) {
                break;
            }

            ASTNode oneexp = explist.getLeafs().get(i);
            Var printexp = parseExp(oneexp);
            createMidCode(MidType.Printf, printexp);
            i += 1;
        }
    } else {
        Var var_formatString = new Var("str", formatString);
        createMidCode(MidType.Printf, var_formatString);
    }

    //todo 11.26 处理了printf函数先执行返回后再一起执行printf，解决printf的顺序问题
    //todo 主要采取移动printf顺序的方法，函数执行放前面，所有printf按照顺序放在最后面
    int record_start = -1;
    for (int i = midCodes.size() - 1; i >= 0; i--) {
        if(midCodes.get(i).getIRstring() == null){
            continue;
        }
        if (midCodes.get(i).getIRstring().equals("#Start Print")) {
```

```
                record_start = i;
                break;
            }
        }
        if (record_start != -1) {
            //先去掉，然后重新排序,把函数调用相关的放在前面，最后全部放printf
            int record_end = midCodes.size() - 1;
            ArrayList<MidCode> tmplist = new ArrayList<>();
            //正序倒出来
            for (int i = record_start; i <= record_end; i++) {
                tmplist.add(midCodes.get(i));
            }
            //原来的删掉
            if (record_end >= record_start) {
                midCodes.subList(record_start, record_end + 1).clear();
            }
            //开始往里面加
            ArrayList<Integer> printfid = new ArrayList<>();
            for(int i =0 ;i < tmplist.size();i++){
                MidCode mid = tmplist.get(i);
                if(!mid.getType().equals(MidType.Printf)){
                    midCodes.add(mid);
                }else {
                    printfid.add(i);
                }
            }
            for (Integer id : printfid) {
                midCodes.add(tmplist.get(id));
            }
            createMidCode(MidType.Note, "#End Print");
        } else {
            MyError.errorat("Midcodegen", 621, "record_start = -1不可能");
        }
    }
```

- 函数调用部分

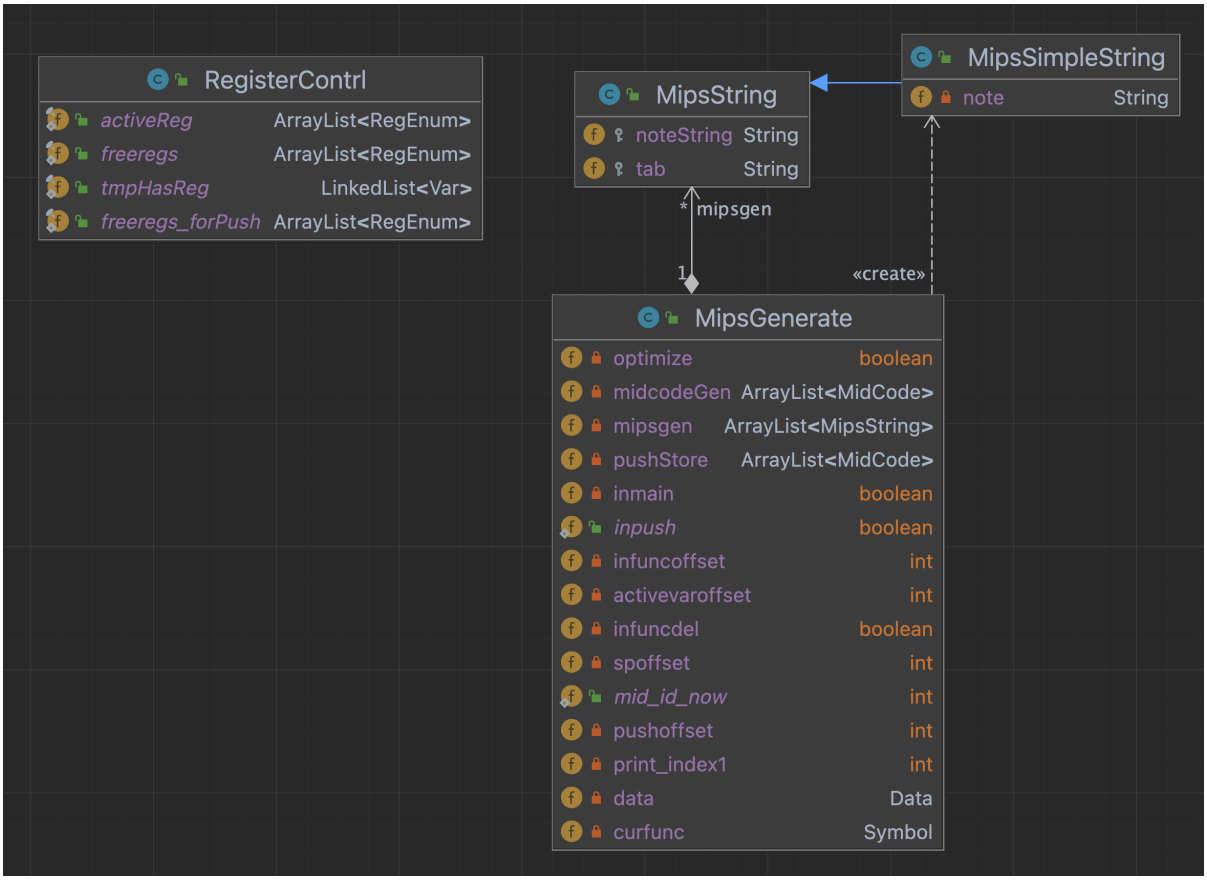
```
public void parseFuncDecl(ASTNode n) {
    ASTNode identnode = n.getLeft();
    String funcname = identnode.getName();
    NodeKind functype = identnode.getKind();


    //填符号表
    curFunc = insertTosymbolTable(identnode, null, true, false);


    MidCodeGenerate.openTable(BlockType.FUNC);
```

```
    createMidCode(MidType.funcDecl, functype, funcname);

    ASTNode params = identnode.getLeft();
    if (params != null) {
        for (ASTNode leaf : params.getLeafs()) {
            parseFuncFParam(leaf);
        }
    }

    parseBlock(n.getRight(), -404);
    createMidCode(MidType.Note, "#end a func"); //todo 这些注释没用的话其实也可
以删了
    MidCodeGenerate.closeTable();
    curFunc = null;
}

private void parseFuncFParam(ASTNode n) {//这里都是identnode
    String name = n.getName();
    Symbol symbol = insertTosymbolTable(n, null, false, true);
    curFunc.addParam(symbol);

    //todo 这里好像可以集到concatraw那里，不管了先理解吧,这里确实可以
    StringBuilder sb = new StringBuilder();
    sb.append("para int ");
    if (n.getKind().equals(NodeKind.INT) ||
n.getKind().equals(NodeKind.CONSTINT)) {
        sb.append(name);

    } else if (n.getKind().equals(NodeKind.ARRAY) ||
n.getKind().equals(NodeKind.CONSTARRAY)) {
        int dimen = n.getNum();
        if (dimen == 1) {
            sb.append(name).append("[]");
        } else if (dimen == 2) {
            sb.append(name).append("[][]");
        }
    } else {
        MyError.errorat("Mipsgenrater", 300);
    }
    //
    createMidCode(MidType.funcPara, sb.toString());
}
```

4.4 Mips目标代码生成

4.4.1 目标代码生成基本架构

架构除了利用到了之前的中间代码架构中的类（不再在图中显示），还用到了新增的寄存器类，寄存器控制器类，目标代码类等。



4.4.2 寄存器分配

首先定义了寄存器枚举类，方便后面直接引用寄存器名称与序号。

```
public enum RegEnum {
    zero,at,v0,v1,a0,a1,a2,a3,t0,t1,t2,t3,t4,t5,t6,t7,s0,
    s1,s2,s3,s4,s5,s6,s7,t8,t9,k0,k1,gp,sp,fp,ra,wrong,none,none2
    ,regisempty
}
```

寄存器分配采用的策略是：无论是临时变量还是符号表内变量都不长时间持有寄存器，而是在使用后立即释放的策略。对于可能出现的寄存器不够用的情况我们把目前占有寄存器的临时变量kickout，保存到栈里，从而产生空闲的寄存器。

- 释放寄存器

```
public static void FreeReg(RegEnum regnum){ //寄存器释放函数
    if(MipsGenerate.inpush){
        freereg_toPushReg(regnum);
        return;
```

```
    }
    int regno = regnum.ordinal();

    if(regno == 28 || regno==29 || regno == 30 || regno == 31 || regno < 8
|| regno>31){
        MyError.errorat("Refisterctrl",34,"没有分配这个"+regno+"寄存器，无法把他
释放");
        //todo 1127
        System.exit(0);
        freeregs.add(RegEnum.none);
        freeregs.add(RegEnum.wrong);
        freeregs.add(RegEnum.wrong);
        freeregs.add(RegEnum.wrong);
        activeReg.remove(RegEnum.none);
    }else {
        freeregs.add(regnum);
        //新增
        ArrayList<Var> record_var_need_del = new ArrayList<>();
        for (Var var : tmpHasReg) {
            if(var.getCurReg().equals(regnum)){
                record_var_need_del.add(var);
            }
        }
        for (Var var : record_var_need_del) {
            tmpHasReg.remove(var);
        }
        activeReg.remove(regnum);
    }

}
```

- 获取寄存器

```
public static RegEnum GetregFrom_pushReg(){
    if(!freeregs_forPush.isEmpty()){
        RegEnum regnum = freeregs_forPush.get(0);
        freeregs_forPush.remove(0);
        return regnum;
    }else {
        return RegEnum.wrong;
    }
}
```

- 踢出临时变量占有寄存器

```
public static  Var kickoutAtmp(){
    if(tmpHasReg.isEmpty()){
        MyError.errorat("Registerctrl",30,"kick不出去");
```

```
        //System.out.println(MipsGenerate.mid_id_now);
    }
    Var tmp = tmpHasReg.removeFirst();
    if(tmp.getName().equals("#tmp78")){
        System.out.println(MipsGenerate.mid_id_now);
        System.out.println("here");
    }
    System.out.println("kick out "+tmp.getName());
    RegEnum has =  tmp.getCurReg();
    FreeReg(has);
    return tmp;
}
```

4.4.3 目标代码生成具体实现

- data段保存

data段需要收集所有的全局变量，以及printf内部的符号串，核心函数如下。

```
public StringBuilder toMipsString() {
    StringBuilder ans = new StringBuilder(".data\n");
    for (MidCode midCode : printfStringMap.keySet()) {
        ans.append(tab).append(printfStringMap.get(midCode)).append(":")
                .append(" .asciiz").append(tab).append("\"")

.append(midCode.getVariable().getName()).append("\"").append("\n");
    }
    for (MidCode midCode : globalInitArr) {
        ans.append(tab).append("Global_")
                .append(midCode.getName()).append(": ").append(".word ");
        //初始化数组
        if(midCode.getType().equals(MidType.intDecl)){
            if(midCode.isInit()){
                ans.append(midCode.getNum());
            }else {
                ans.append("0");
            }
        }else{
            if (midCode.isInit()){
                for (Integer num : midCode.getInitList()) {
                    ans.append(num).append(",");
                }
                ans.deleteCharAt(ans.length()-1);//去掉最后一个逗号
            }else {
                int size = midCode.getArraySize();
                ans.append("0:").append(size);
            }
        }
        ans.append("\n");
```

```
    }
    return  ans;
}
```

- 数组存取

在生成mips码时多次涉及对于数组存取，所以对于这类操作封装成了一个统一的函数，对于数组为全局数组，常量数组，变量数组，局部数组的不同情况做了不同分类，只要涉及数组的存取直接调用即可。

```
public void LwOrSW_between_Reg_Array(String instrction, RegEnum reg, Var
arraynum) {
    if (!arraynum.getType().equals("array")) {
        MyError.errorat("Mipsgen", 842, "这里必须传array var");
    }
    Symbol arraysymbol = arraynum.getSymbol();
    Var index = arraynum.getVar();
    String arrayname = arraynum.getName();

    //global 数组
    if (arraysymbol.isGlobal()) {
        if (index.getType().equals("num")) {
            //todo bug A3 全局数组la访问错误
            int offsetnum = index.getNum();
            if (instrction.equals("lw")) {
                genaddSwLwLa("la", reg, "Global_" + arrayname);
                genaddSwLwLa(instrction, reg, offsetnum * 4 + "($" + reg +
")");
            } else if (instrction.equals("sw")) {
                RegEnum tmp = getReg_kickouttmp_necessary(arraynum,false);
                genaddSwLwLa("la", tmp, "Global_" + arrayname);
                genaddSwLwLa(instrction, reg, offsetnum * 4 + "($" + tmp +
")");
                RegisterContrl.FreeReg(tmp);
            } else {
                MyError.errorat("Mipsgen", 918, "没有sw，lw以外的");
            }


        } else if (index.getType().equals("var")) {
            RegEnum indexreg = getVarReg(index, true, false);
            genaddSll(indexreg, indexreg, 2);
            genaddSwLwLa(instrction, reg, "Global_" + arrayname + "($" +
indexreg + ")");
            //不做优化时，申请必须释放
            RegisterContrl.FreeReg(indexreg);
        }
    } else {
        //下面计算两种情况，分别在函数内和main内，主要是地址计算有差别
```

```
        if (infuncdel) {
            //函数内
            ArrayList<Integer> res = curfunc.hasthisPara(arrayname);
            int isAParam = res.get(0);
            if (isAParam == 1) {
                //是函数参数,函数参数中的函数需要取基地址，因为数组传参数传的是地址
                //todo 这里如果不给arraynum设置iskindofsymbol 获得不了正确的基地址
                //todo 到底要不要设置，还得看Visit那边有没有什么关系，待定，如果不设置，给lwSymbolFromStackToReg加个参数也行
                RegEnum arraybasetmp = lwSymbolFromStackToReg(arraynum, null, true,false);
                if (index.getType().equals("num")) {
                    int offset = index.getNum() * 4;
                    genaddSwLwLa(instrction, reg, offset + "($" + arraybasetmp + ")");

                } else if (index.getType().equals("var")) {

                    RegEnum varindex_reg = getVarReg(index, true, false);
                    genaddSll(varindex_reg, varindex_reg, 2);
                    genadd("add", arraybasetmp, arraybasetmp, varindex_reg);
                    genaddSwLwLa(instrction, reg, "($" + arraybasetmp + ")");
                    RegisterContrl.FreeReg(varindex_reg);
                } else {
                    MyError.errorat("Mipsgen", 732, "没有这种类型");
                }
                RegisterContrl.FreeReg(arraybasetmp);
            } else {
                //是函数内部数组，内部数组不需要取基地址，直接算偏移然后用sp取
                //todo 有问题
                //int localbaseadr = Integer.parseInt(baseaddress);
                if (index.getType().equals("num")) {
                    String arraddress = getSymbolAdressAtStack(arraysymbol, index.getNum() * 4);
                    genaddSwLwLa(instrction, reg, arraddress);
                } else if (index.getType().equals("var")) {
                    RegEnum varindex_reg = getVarReg(index, true, false);
                    genaddSll(varindex_reg, varindex_reg, 2);
                    genadd("add $" + varindex_reg + ", $" + varindex_reg + ", $sp", true);
                    int arraybaseoffset = getSymbolOffsetInfunc(arraysymbol, 0);
                    genaddSwLwLa(instrction, reg, arraybaseoffset + "($" + varindex_reg + ")");
                    RegisterContrl.FreeReg(varindex_reg);
```

```
                } else {
                    MyError.errorat("Mipsgen", 745, "没有这种类型");
                }
            }
        } else {
            //main内
            if (index.getType().equals("num")) {
                String arradress = getSymbolAdressAtStack(arraysymbol,
index.getNum() * 4);
                genaddSwLwLa(instrction, reg, arradress);
            } else if (index.getType().equals("var")) {
                String arradress = getSymbolAdressAtStack(arraysymbol, 0);
                RegEnum varindex_reg = getVarReg(index, true, false);
                genaddSll(varindex_reg, varindex_reg, 2);
                genaddSwLwLa(instrction, reg, arradress + "($" +
varindex_reg + ")");
                RegisterContrl.FreeReg(varindex_reg);

            } else {
                MyError.errorat("Mipsgen", 762, "没有这种类型");
            }
        }
    }
}
```

- 变量存取

对于Var类的存取，无论是对符号表内变量，临时变量的存取都有相似的过程，因此统一封装为这两个函数。只要涉及对于Var的存取都可以直接调用。

```
public RegEnum lwSymbolFromStackToReg(Var var, RegEnum toReg, boolean
arraymustneed,boolean tmpmustneed) {
    //计算地址，将stack中的变量取出
    Symbol s = var.getSymbol();
    if ((var.isKindofsymbol() || arraymustneed) && s != null) {
        String address = getSymbolAdressAtStack(s, 0);
        if (toReg == null) {
            RegEnum reg = getReg_kickouttmp_necessary(var,false);
            genadd("lw $" + reg + " ," + address, true);
            return reg;
        } else {
            genadd("lw $" + toReg + " ," + address, true);
            return toReg;
        }

    } else if(tmpmustneed){
        String tmpvarAddr = getTmpVarAdressAtStack(var);
        genadd("lw $" + toReg + " ," + tmpvarAddr, true);
```

```
        return toReg;
    }else {
        MyError.errorat("Mipsgen", 739, "symbol有问题，找不到");
    }
    return null;
}
```

```
public RegEnum saveRegToVarable(Var dest, RegEnum rightReg) {  //var =
right
    Symbol destsymbol = dest.getSymbol();
    //存到左边
    if (dest.getType().equals("var")) {
        //理论来讲左边一定是symbol类的VAriable
        if (!dest.isKindofsymbol()) {
            //todo 这个理论上来讲只能在assign return或者数组赋值出现，因为左值是一个
临时变量
            RegEnum retreg = getVarReg(dest, false, false);
            genadd("move", retreg, rightReg);
            return retreg;
        } else {
            if (destsymbol.isGlobal()) {
                //全局变量改Global
                //add("sw $" + regForOper1 + ", Global_" + globalvarname);
                genaddSwLwLa("sw", rightReg, "Global_" +
destsymbol.getName());
            } else {
                //局部变量sw存回stack
                genaddSwLwLa("sw", rightReg,
getSymbolAdressAtStack(destsymbol, 0));
            }
        }
    } else {
        MyError.errorat("Mipsgen", 362, "左值不可能不是tmp var");
    }
    return RegEnum.wrong;
}
```

- 函数调用

函数调用主要涉及中间代码种类包括call,push两类，因为push类中间代码在call之前，而执行call时需要先将活跃变量入栈再进行push参数，因此对于push中间代码先进行存储，到生成call时，在活跃变量入栈后再生成push类的中间代码。

```
public RegEnum genPush(MidCode midCode) {
    pushoffset -= 4; // -4,-8,-12这样
    Var pushVar = midCode.getVariable();

    if (midCode.getId() == 44) {
```

```
            System.out.println("here");
        }

        if (pushVar.getType().equals("array")) {
            //array push的是数组地址
            RegEnum pusharray_baseaddr_reg =
getReg_kickouttmp_necessary(pushVar,false);
            writeArrayAddrToReg(pushVar, pusharray_baseaddr_reg);
            genaddSwLwLa("sw", pusharray_baseaddr_reg, pushoffset + "($sp)");
            RegisterContrl.FreeReg(pusharray_baseaddr_reg);

        } else {
            RegEnum pushReg = getVarReg(pushVar, true, false);
            //push的sp统一下降放到call
            genaddSwLwLa("sw", pushReg, pushoffset + "($sp)");
            //todo 这里的tmpvar，不放，先活跃变量入栈，再push，push这里放掉，活跃变量出寄
存器的时候顺序不对应了
            //todo 但是tmpvar也不能不放
            //todo 目前方案,不是临时变量的放掉，是临时变量的不放，返回
            if (pushVar.getType().equals("num") ||
                    (pushVar.getType().equals("var") &&
pushVar.isKindofsymbol())) {
                RegisterContrl.FreeReg(pushReg);
            } else {
                //tmp变量
                if(pushVar.isHasbeenkickout()){
                    RegisterContrl.FreeReg(pushReg);
                }else {
                    return pushReg;
                }

            }
        }
        return RegEnum.none;
    }
```

进行call时先推活跃变量入栈，再生成push参数，最后将活跃变量取出栈恢复原寄存器。

```
public void genCall(MidCode midCode) {
        if (midCode.getId() == 131 || midCode.getId() == 134) {
            System.out.println("here");
        }
        String funcname = midCode.getIRstring();
        Symbol funcsymbol =
MidCodeGenerate.lookGlobalTableFindsamename(funcname);
        if (funcsymbol == null) {
            MyError.errorat("Mipsgen", 421, "没有这个函数");
        }
```

```
        assert funcsymbol != null;
        int paranum = funcsymbol.getPamaNum();

        pushoffset = 0;
        //先推活跃变量到本地栈(倒推正取)
        activevaroffset = 0;
        ArrayList<RegEnum> activeRegList = RegisterContrl.activeReg; // 这里
用深克隆
        int preactiveoffset = 0;

        //活跃变量全部推入，不管
        for (int i = activeRegList.size() - 1; i >= 0; i--) {
            RegEnum activereg = activeRegList.get(i);
            genaddIm("addi", RegEnum.sp, RegEnum.sp, -4);
            genaddSwLwLa("sw", activereg, "($sp)");
            preactiveoffset += 4;
        }
        activevaroffset = preactiveoffset;


        //再推函数参数到函数栈,并将push从pushstore删除
        ArrayList<Integer> removepush = new ArrayList<>();
        ArrayList<RegEnum> should_free_aft_push = new ArrayList<>();
        inpush = true;
        for (int i = pushStore.size() - paranum; i < pushStore.size(); i++)
{
            RegEnum should_free = genPush(pushStore.get(i));
            if (!should_free.equals(RegEnum.none)) {
                should_free_aft_push.add(should_free);
            }
            removepush.add(i);
        }
        inpush = false;
        for (int j = removepush.size() - 1; j >= 0; j--) {
            int i = removepush.get(j);
            pushStore.remove(i);
        }
        activevaroffset = 0;

        //推ra到函数栈
        genaddIm("addi", RegEnum.sp, RegEnum.sp, pushoffset - 4);//再减4存ra
        genaddSwLwLa("sw", RegEnum.ra, "($sp)");

        //跳转
        genadd("jal Function_" + midCode.getIRstring(), true);

        //栈维护，提取ra
        genaddSwLwLa("lw", RegEnum.ra, "($sp)");
        genaddIm("addi", RegEnum.sp, RegEnum.sp, -(pushoffset - 4));
```

```
        //活跃寄存器全部提取
        for (RegEnum activeReg : activeRegList) {
            genaddSwLwLa("lw", activeReg, "($sp)");
            genaddIm("addi", RegEnum.sp, RegEnum.sp, 4);
        }

        //todo 这个不要了，push改成均不释放，到call完然后活跃寄存器按顺序出栈以后，放掉该放掉的
        for (RegEnum shoud_free : should_free_aft_push) {
            RegisterContrl.FreeReg(shoud_free);
        }
    }
```

- 其他中间代码由于在生成中间代码的时候记录的信息已经比较完善，在生成目标代码时只需要转化为对应的跳转，运算等指令即可，在此不做赘述。

5. 优化设计

> 因为能力有限而且时间不太够，所以简单做了两个优化，没有做中间代码层次的优化。一个是运算方面对于乘除法，模运算进行了形式转化并优化，第二个小优化是在生成中间代码的时候结合符号表，进行常量直取，包括遇到常量、常量数组直接取出数字，如果表达式只包括常量之间运算直接算出结果等优化，将运算和数值提取直接放到代码生成阶段，不必放到mips运行阶段。

1 运算优化

1.1 乘法优化

- 对2的次方的数的乘积直接替换为sll指令减少乘法指令
- 如果有一乘数为0直接li为0，减少mult指令

```
                if (isPowerOfTwo(rightnum)) {
                    //2的次方
                    int mi = (int) (Math.log(rightnum) / Math.log(2));
                    genaddSll(destreg, leftreg, mi);
                } else if (rightnum == 0) {
                    genli(destreg, 0);
                } else {
                    genli(RegEnum.fp, rightnum);
                    genadd("mult", leftreg, RegEnum.fp);
                    genadd("mflo", destreg);
                }
```

1.2 除法优化

- 当两个操作数中一个除数为常数时，进行优化，用位移与加减运算代替除法
- 相关实现参考了一篇论文《Division by Invariant Integers using Multiplication》中的计算公式，论文来源于一篇学长的优化文档，对其进行了参考实现：

procedure CHOOSE_MULTIPLIER(**uword** d, **int** $prec$);
Cmt. d – Constant divisor to invert. $1 \le d < 2^N$.
Cmt. $prec$ – Number of bits of precision needed, $1 \le prec \le N$.
Cmt. Finds m, sh_{post}, ℓ such that:
Cmt. $2^{\ell-1} < d \le 2^{\ell}$.
Cmt. $0 \le sh_{\text{post}} \le \ell$. If $sh_{\text{post}} > 0$, then $N + sh_{\text{post}} \le \ell + prec$.
Cmt. $2^{N+sh_{\text{post}}} < m * d \le 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})$.
Cmt. Corollary. If $d \le 2^{prec}$, then $m < 2^{N+sh_{\text{post}}} * (1 + 2^{1-\ell})/d \le 2^{N+sh_{\text{post}}-\ell+1}$.
Cmt. Hence m fits in $\max(prec, N - \ell) + 1$ bits (unsigned).
Cmt.
int $\ell = \lceil \log_2 d \rceil$, $sh_{\text{post}} = \ell$;
udword $m_{\text{low}} = \lfloor 2^{N+\ell}/d \rfloor$, $m_{\text{high}} = \lfloor (2^{N+\ell} + 2^{N+\ell-prec})/d \rfloor$;
Cmt. To avoid numerator overflow, compute m_{low} as $2^N + (m_{\text{low}} - 2^N)$.
Cmt. Likewise for m_{high}. Compare m' in Figure 4.1.
Invariant. $m_{\text{low}} = \lfloor 2^{N+sh_{\text{post}}}/d \rfloor < m_{\text{high}} = \lfloor 2^{N+sh_{\text{post}}} * (1 + 2^{-prec})/d \rfloor$.
while $\lfloor m_{\text{low}}/2 \rfloor < \lfloor m_{\text{high}}/2 \rfloor$ **and** $sh_{\text{post}} > 0$ **do**
    $m_{\text{low}} = \lfloor m_{\text{low}}/2 \rfloor$; $m_{\text{high}} = \lfloor m_{\text{high}}/2 \rfloor$; $sh_{\text{post}} = sh_{\text{post}} - 1$;
end while; /* Reduce to lowest terms. */
return (m_{high}, sh_{post}, ℓ); /* Three outputs. */
end CHOOSE_MULTIPLIER;

Figure 6.2: Selection of multiplier and shift count

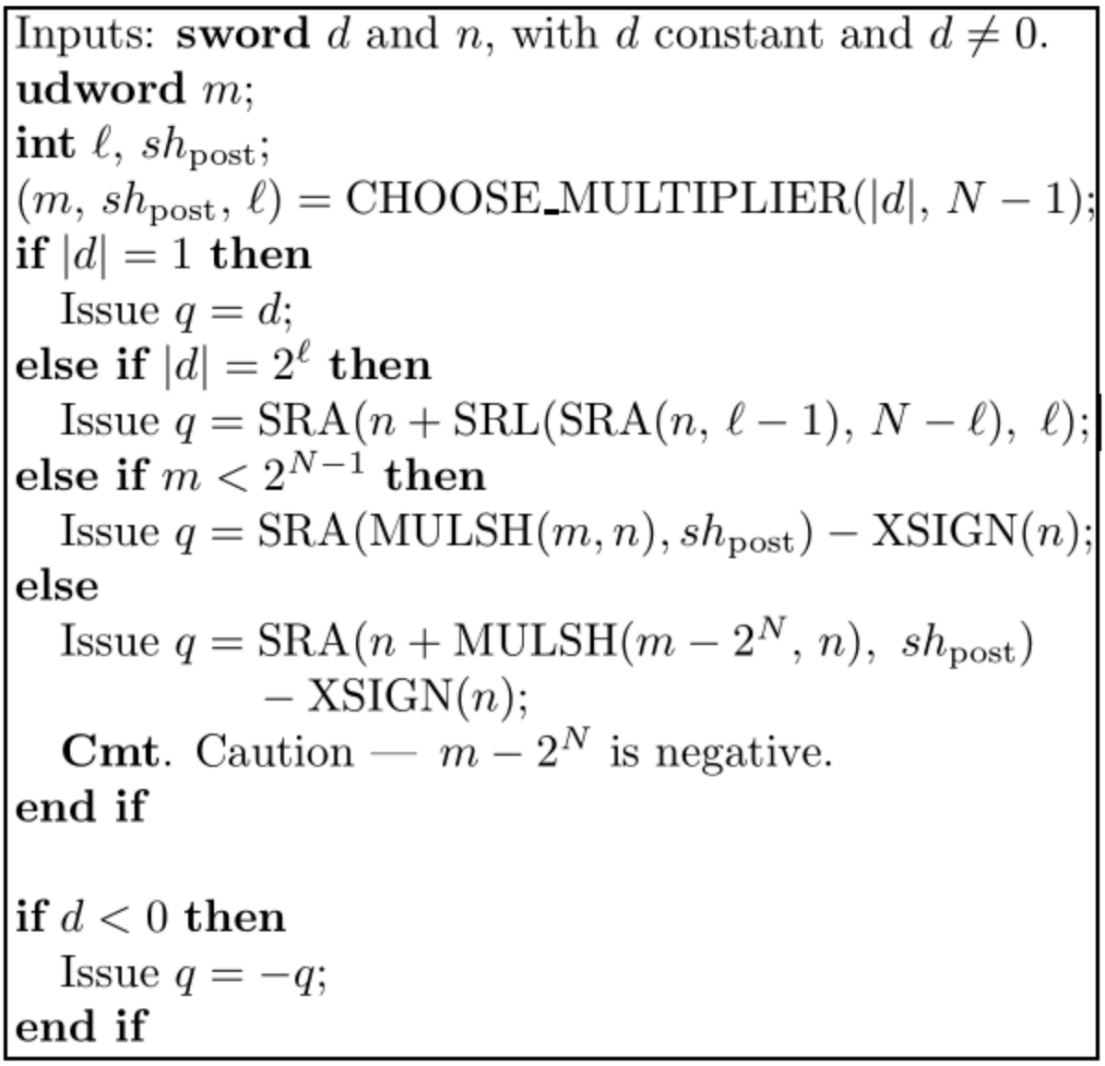


Figure 5.2: Optimized code generation of signed $q = \text{TRUNC}(n/d)$ for constant $d \neq 0$

- 除法优化代码摘录

```
    public void DivOptimize(RegEnum dest, RegEnum left, int rightnum) {
        Divopt ans = ConstValue.ChooseMultiplier(Math.abs(rightnum));
        //System.out.println("here"+ans.get(0) + " "+ans.get(1)+"
"+ans.get(2));
        long m = ans.m_high;
        int sh_post = ans.sh_post;
        //MyError.errorat(m + sh_post+" haha",1174);
        if (Math.abs(rightnum) == 1) {
            genadd("move", dest, left);

        } else if (isPowerOfTwo(Math.abs(rightnum))) {
            int mi = (int) (Math.log(rightnum) / Math.log(2));
            genaddSraSrl("sra", dest, left, (mi - 1));
            genaddSraSrl("srl", dest, dest, (32 - mi));
            genadd("add", dest, dest, left);
```

```
            genaddSraSrl("sra", dest, dest, mi);


        } else if (m < Math.pow(2, 31)) {  // q = SRA(MULSH(m, n), shpost)
- XSIGN(n)

            genli(RegEnum.a3, m);
            genadd("mult", left, RegEnum.a3);
            genadd("mfhi", dest);
            genaddSraSrl("sra", dest, dest, sh_post);

            genaddSraSrl("slti", RegEnum.a3, left, 0);
            genadd("add", dest, dest, RegEnum.a3);


        } else {
            genli(RegEnum.a3, (int) (m - Math.pow(2, 32)));
            genadd("mult", left, RegEnum.a3);
            genadd("mfhi", dest);
            genadd("add", dest, dest, left);
            genaddSraSrl("sra", dest, dest, sh_post);

            genaddSraSrl("slti", RegEnum.a3, left, 0);
            genadd("add", dest, dest, RegEnum.a3);
        }

        if (rightnum < 0) {
            genadd("sub", dest, RegEnum.zero, dest);
        }
    }
```

1.3 模运算优化

- 利用上述除法优化的基础，进行取余数优化： 将a % b 翻译为 a - a / b * b。实现代码如下
- 总体思路便是利用了除法优化基础，进一步进行乘积减法运算，额外的如果模数为1的话直接li为0，减少模运算。

```
if (rightnum == 1) {
    genli(destreg, 0);
} else {
    DivOptimize(destreg, leftreg, rightnum);
    genli(RegEnum.fp, rightnum);
    genadd("mult", destreg, RegEnum.fp);
    genadd("mflo", RegEnum.fp);
    genadd("sub", destreg, leftreg, RegEnum.fp);
}
```

2 常量直取

常量直取在生成中间代码时初步实现，在建立符号表时遇到const常量无论是int还是数组都直接将其初值存储在符号表以及中间代码类之中，以便之后生成目标代码时直接取用。

- 代码实现如下，在遇到const int和const array时直接存储其值，以便后续生成

```
private Var parseIdent(ASTNode n) {
    //MyError.sout("In parseIdent " + n.getName() + " " + n.getKind());
    NodeKind kind = n.getKind();

    if (kind.equals(NodeKind.FUNC)) {   //函数调用！！！！！！！
        String funcname = n.getName();
        ASTNode rparams = n.getLeft();


        Symbol func =
MidCodeGenerate.lookGlobalTableFindsamename(funcname);


        if (rparams != null) {      //函数有参数则push
            for (int i = 0; i < rparams.getLeafs().size(); i++) {
                ASTNode para = rparams.getLeafs().get(i);

                Symbol fparami = func.getParams().get(i); //函数的第i个参数类
型
                int arraydimen = fparami.getDimension();

                if (fparami.isArray()) {       //如果是array类型的函数参数,传参
数的时候用
                    Var paraexp = parseParaArray(para, arraydimen);
//需返回array类型
                    createMidCode(MidType.Push, paraexp);

                } else {
                    Var paraexp = parseExp(para);      //正常的var类型exp
                    createMidCode(MidType.Push, paraexp);
                }
```

```
            }
        }

        //call
        MidCode ir = new MidCode(MidType.call, funcname);
        ir.setSymbol(func);
        ir4init(ir);

        //ret
        if (func.getFuncKind() != null &&
!func.getFuncKind().equals(FuncKind.VOID)) {
            Var tmpvar = getTmpVar();
            createMidCode(MidType.assign_ret, tmpvar);
            return tmpvar;
        }
        return null;


    } else if (kind.equals(NodeKind.INT) || kind.equals(NodeKind.CONSTINT))
{
        if (kind.equals(NodeKind.CONSTINT)) {     //优化 constint直接存储
            String name = n.getName();
            Symbol symbol = MidCodeGenerate.lookallTableFindsamename(name);
            assert symbol != null;
            int num = symbol.getNum();
            return new Var("num", num);

        }

        Var intvar = new Var("var", n.getName());
        intvar.setiskindofsymbolTrue();//设置成是符号表里面的东西
        return intvar;


    } else if (kind.equals(NodeKind.CONSTARRAY)) {
        //return parseArrayVisit(n);
        Var arrnnum = parseArrayVisit(n);
        Var tmp = getTmpVar();
        createMidCode(MidType.assign2, tmp, arrnnum);
        return tmp;
    } else if (kind.equals(NodeKind.ARRAY)) {
        //这里必须分两类
        //我们要把临时数组的访问最终聚合为临时变量

        Var arrnnum = parseArrayVisit(n);
        Var tmp = getTmpVar();
        createMidCode(MidType.assign2, tmp, arrnnum);
        return tmp;
    }
```

```
    MyError.errorat("Midgen", 801, "不知道ident是什么类型");
    return null;
}
```

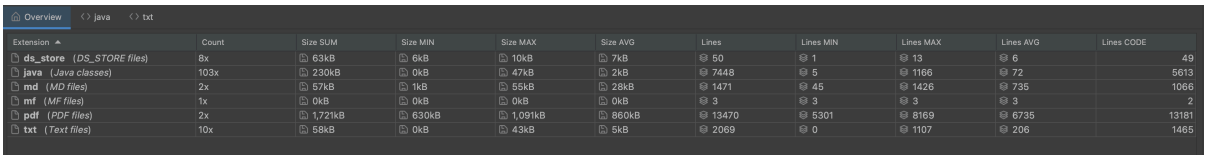
- 在建立语法树的时候如果遇到数字之间的加减法，不生成中间变量而是直接进行运算得到最终数字,以下是代码实现

```
public int calcuValue() {
    if (OperDiction.hasOperator(opstring)) {
        switch (opstring) {
            case "+":
                if (right != null) {
                    return left.calcuValue() + right.calcuValue();
                }
                return left.calcuValue();
            case "-":
                if (right != null) {
                    return left.calcuValue() - right.calcuValue();
                }
                return -left.calcuValue();
            case "*":
                return left.calcuValue() * right.calcuValue();
            case "/":
                return left.calcuValue() / right.calcuValue();
            case "%":
                return left.calcuValue() % right.calcuValue();
            default:
                MyError.errorat("ASTnode",151);
                return 0;
        }
    }else if(type.equals(NodeType.Number)){
        return num;
    }else if(type.equals(NodeType.Ident)){
        if(kind.equals(NodeKind.FUNC)){
            //func 不再这里算，这里只能算一些简单的加减乘除
            MyError.errorat("ASTNODE",158,"这里不应该有这种情况，不能在编译时计算
函数");
        } else if
(kind.equals(NodeKind.INT)||kind.equals(NodeKind.CONSTINT)) {
            Symbol symbol = MidCodeGenerate.lookallTableFindsamename(name);
            return symbol.getNum();
        }else
if(kind.equals(NodeKind.ARRAY)||kind.equals(NodeKind.CONSTARRAY)){
            Symbol symbol = MidCodeGenerate.lookallTableFindsamename(name);
            if(symbol != null){
                if(right != null){ //二维
                    int dimen2 = symbol.getDimen2();
                    int index1 = left.calcuValue();
```

```
                    int index2 = right.calcuValue();
                    int index = index1*dimen2 + index2;
                    return symbol.getArrayValue().get(index);
                }else {//一维数组
                    int index = left.calcuValue();
                    return symbol.getArrayValue().get(index);
                }
            }else {
                MyError.errorat("ASTNODE",179,"符号表没找到，但理应之前定义");
            }
        }else {
            MyError.errorat("ASTNODE",178);
        }

    }else {
        MyError.errorat("ASTNODE",182);
    }
    return -92929299;
}
```

在最终生成目标mips码时，将表达式中所有const常量直接用常量的值替换，并且我们将表达式中数字和数字之间的直接运算已经在建立语法树时运算完毕，也直接用数值替换，这样可以减少很多中间临时变量的使用。以上时所有的常量直取的相关优化。

6.代码量及总结



最后的代码量在5600多行，总之本学期的编译课程中学到了编译的相关知识，也学到了形式语言等等相关知识，总而言之收获还是非常多的。最后最重要的也是最有满足感的还是自己完成了一个mips的编译程序，从词法分析，语法分析到错误处理，到最后的中间代码生成与目标代码生成，整个过程体验下来虽然压力很大，但是直到做完之后的满足感还是非常不错的。