申优文章

19231258 陈思翰

一.前言

编译课程为提高系统能力的核心课程之一，其挑战性是不言而喻的，它要求我们不仅仅是掌握理论知识，还需要自己动手亲自实践（就个人而言"知道"和"做出来"是完全不同的难度）。而为了更好更有效地培养我们的系统能力，课程组也几乎不给出架构上的设计建议，但是我们在刚开始学习的时候应该是对整个架构甚至是到底要干什么都是不太清楚的，这就注定了我们的架构设计大概率会存在缺陷，很可能会经历重构的过程，不过这也是我们系统能力提高的必经之路。

除此之外，做一个小编译器（尽管经过了课程组简化）难度也还是很高的，主要体现在其工程量之大。本学期最后做完优化我的编译器包括各种`debug`的代码，共有`8000`行左右的。代码量大带来的问题就是，一旦出了`bug`，找起来将十分困难，并且容易忘记之前的代码细节，迭代时稍不留神就出现隐藏很深的`bug`，其中让我感触最深的就是错误处理时`bug`死活查不出来，结果发现是词法分析的`bug`.

与课本上的`5`阶段（`词法分析`,`语法分析`,`语义分析和中间代码生成`,`代码优化`,`目标代码生成`)不同，我认为我们实验中编译器的的构造分为以下几个部分，分别为：`词法分析`，`语法分析`，`语义分析和错误处理`，`中间代码和目标代码生成`，`代码优化`。由于代码优化的难度比较大、工作比较多，新增代码的地方比较分散，并且实际上实现编译器的基本功能（产生目标代码）并不需要经过代码优化阶段，所以我将代码优化分为独立的一个部分进行讨论，将其他`阶段放在系统架构部分进行讨论

整个编译器采用`Java`实现

二.系统架构

一.词法分析

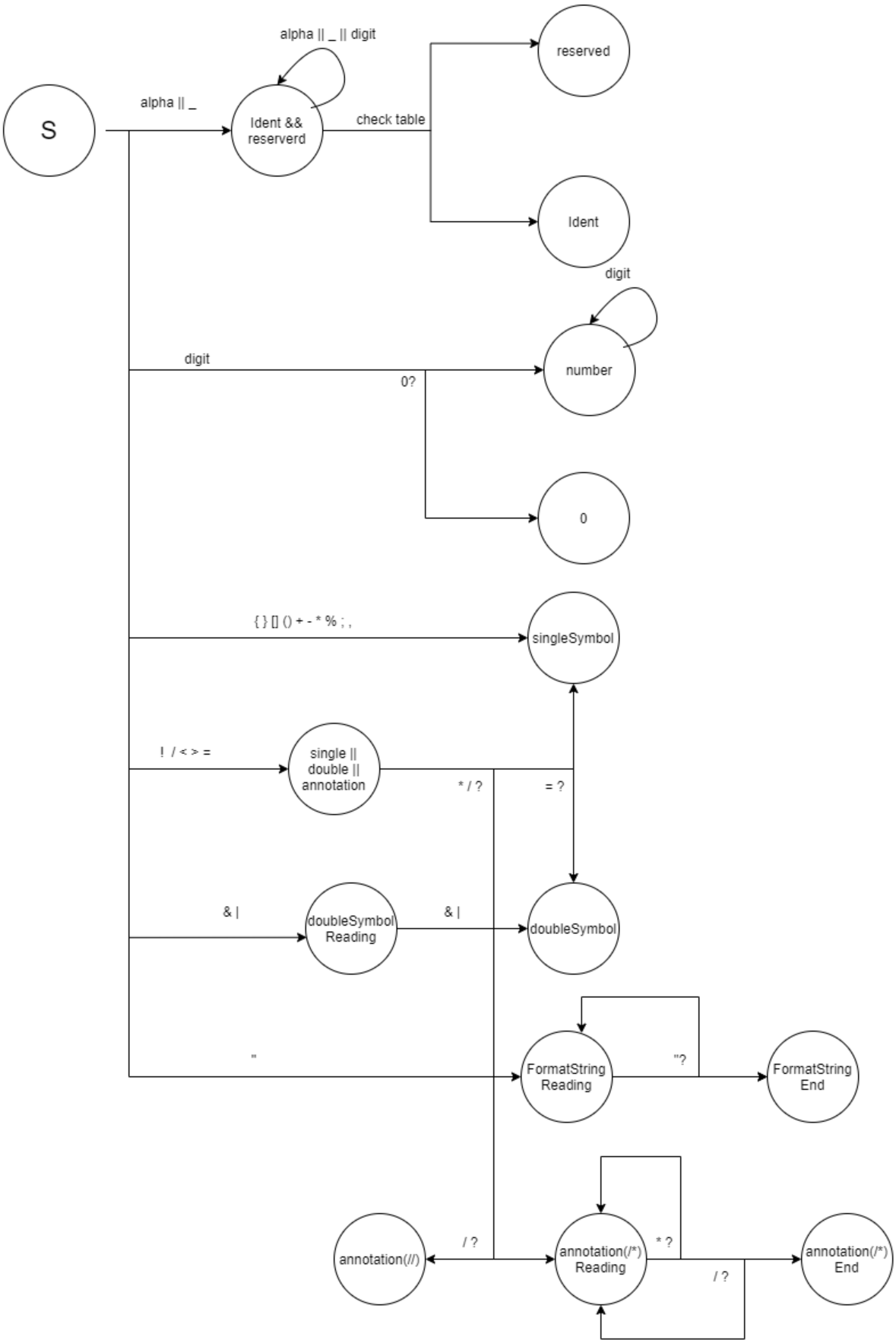
1.模块设计

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				

为了进行语法分析，我创建了一个专门的`WordAnalysis`模块负责语法分析，具体功能就是逐个读取字符，按照状态机的原理解析出对应的单词，添加到`ArrayList`中，最后一起返回，对于注释`//`,`/**/`，我选择直接去掉，不加入到`ArrayList`中

建立存储字符信息的类`word`,包括`单词值`,`单词类别`,`行号`等信息

2.状态机



注:

`?`表示选择 例如 `/ ?`表示当前读入的字符是否为 `/`

`||`表示或者

3.实现细节

- Java不像`C/C++`可以有指针回退函数,于是自己实现了一个`retract`

```
public void retract(){
    try {
        raf.seek(raf.getFilePointer() - 1);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

- `Java`不像`C/CPP` `char`和`int`能直接隐式转换，在比较的时候很不方便，在`WordAnalysis`中增设两个属性

```
    private int sym;//symbol的值
    private char c;//symbol的值
```

- 返回`"EOF"-""`键值对（单词类-单词值）的表示读到EOF

二.语法分析

1.模块设计

经过`词法分析模块`，得到了一个包含所有单词的`ArrayList`,`语法分析模块`需要对其进行递归下降处理，需要注意的有两点：

- 消除左递归
    - 改写文法，之后根据文法采用循环解决
- 当不能确定分支时，不断向前预读，直到能区分分支

我创建了`GrammarAnalysis`模块进行语法分析，对于每一条文法推导规则的非终结符建立一个解析函数，自顶向下递归解析，约定进入非终结符解析函数的时候，`symbol`中已经读入了新的单词，出来的时候要读入新的单词

同时预留错误处理的接口，具体来说就是当某个解析程序中解析到不是的期望单词时应该怎么办

2.实现细节

First集处理

非终结符	First集
Decl	const|int
ConstDecl	const
VarDecl	int
VarDef	ident
ConstDef	ident
ConstInitVal	(|intconst|ident（必须是常量）|+|-|{
ConstExp	(|intconst|ident（必须是常量）|+|-
InitVal	(|intconst|ident|+|-|{
Exp	(|intconst|ident|+|-
AddExp	(|intconst|ident|+|-（如果是逻辑Exp则还有！）
MulExp	(|intconst|ident|+|-（如果是逻辑Exp则还有！）
UnaryExp	(|intconst|ident|+|-（如果是逻辑Exp则还有！）
PrimaryExp	(|ident|intconst
Number	intconst
FuncDef	void|int
FuncType	void|int
MainFuncDef	int
FuncFParams	int
FuncFParam	int
FuncRParam	(|intconst|ident|+|-
Block	{
BlockItem	const|int|ident|;|(|intconst|+|-|{|if|while|break|continue|return|printf
Stmt	ident|;|(|intconst|ident|+|-|{|if|while|break|continue|return||printf
LVal	ident
Cond	(|intconst|ident|+|-|!
LOrExp	(|intconst|ident|+|-|!
LAndExp	(|intconst|ident|+|-
EqExp	(|intconst|ident|+|-
RelExp	(|intconst|ident|+|-
UnaryOp	+|-|!

相交情况	处理策略
int和FuncDef相交	FuncDef int后两个必是(
\	\
\	\
文法前半部分可提取	提取文法
\	\
\	\
\	\
\	\
\	\
\	\
\	\
Ident和PrimaryExp下的LVal相交	Ident之后必有(
\	\
\	\
int Func和int main相交	主函数有main
\	\
int Func和int main相交	主函数有main
\	\
\	\
\	\
\	\
\	\
两个LVal分支，一个Exp可能出现的分支都有ident	Exp下的ident后面没有 = 只有一个LVal后面有getint
\	\
\	\
\	\
\	\
\	\
\	\
\	\

表格放不下我拆成两块

显然文法中存在某几个非终结符分支的First集有交集的情况，为了继续用递归下降处理，需要进行预读，直到能区分分支，举个例子

$$Exp \rightarrow AddExp$$
$$AddExp \rightarrow MulExp|AddExp('+'|'-')MulExp$$
$$MulExp \rightarrow UnaryExp|MulExp('*'|'/'|'\%')$$
$$UnaryExp \rightarrow PrimaryExp|Ident'('[FuncRParams]')'|UnaryOpUnaryExp$$
$$PrimaryExp \rightarrow' ('Exp')'|LVal|Number$$
$$LVal \rightarrow Ident'['Exp']'$$
$$Number \rightarrow IntConst$$

可以看到最终存在5个不同的起始符，当下降到`Exp`上一层，并且读到5个起始符之一时（上一层其他路径如果存在与该`Exp`路径`First`集相交的情况应该在上一层考虑所以本层一定时5个起始符之一），采取预读的方式往后读，如果后面一个还是一样，就再往后，直到读到不一样的单词

将前缀(`First`)判断封装成函数，逻辑更清晰，关键是期中期末新增文法时比较方便

```
public Boolean isExpPrefix() {
        //ExpPrefix不含! 只有+ -
        return getSymbolClass().equals("IDENFR") ||
getSymbolClass().equals("PLUS") || getSymbolClass().equals("MINU")
                || getSymbolClass().equals("LPARENT") ||
getSymbolClass().equals("INTCON");
    }
```

- 错误处理的过程中发现，对于`LVal`和`AssignStmt`无法靠预读知道到底需要进入哪个分支（不知道`;`是否会缺失），解决方法是进行"假读"并回溯(之后需要重新下降此部分)，即先默认进入`LVal`,如果后面紧接着没有`=`，就说明不是`LVal`分支

注意进行封装：

```
public String getSymbolClass() {
        return symbol.get("class");
    }
    public void getSymbol() {
        if (symbol != null) {
            wordAndGrammar.add(symbol);
        }
        //刚开始的时候 symbol为null
    .   //将存储答案加在getSymbol中 防止遗忘
        do {
            symbol = wordAnalysis.getSymbol(false);
        } while (symbol.get("class").equals("ANNO"));
        //过滤注释
    }
    public String getPeekSymbolClass(int i) {
        //peek 并不改变指针 只有get才会改变指针 注意 peek指针指在当前指针的下一个
        peekSymbol = wordAnalysis.peekSymbol(i);
        return peekSymbol.get("class");
    }
    public void addNonTermimal(String nonTermValue) {
        nonTermimal = new HashMap<>();
        nonTermimal.put("class","NONTER");
```

```
            nonTermimal.put("word",nonTermValue);
            wordAndGrammar.add(nonTermimal);
        }
        //非终结符记录
```

三.语义分析和错误处理

1.模块设计

1.AST设计

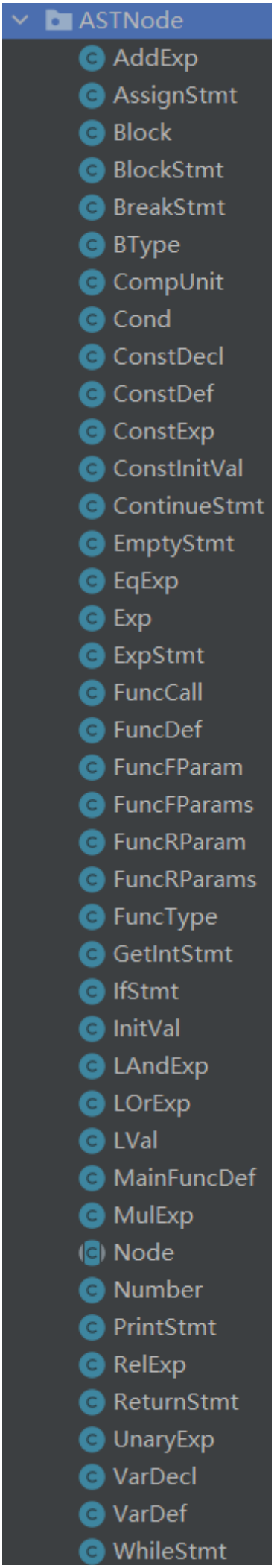
个人理解语义分析就时在词法分析和语法分析的基础上将信息有效地组织起来，方便后面地模块取用

之前已经通过递归下降对源代码进行了语法分析，我们只需要把语义分析的过程加进递归下降即可，关键是如何把信息有效地组织起来

我的方法是建立`AST`(抽象语法树)，`AST`的构造和文法高度对应，但也不是完全一致，存在文法中有的非终结符`AST`中没有，`AST`中有的`文法`中却没有的情况

为了保持信息、结构的完整性，我的`AST`各与文法基本相同,此外，对文法中没有细分的语法成分重新建立了不同节点加以区分，所有节点都是非终结符，终结符作为属性存储在对应的非终结符节点中

并建立抽象类`Node`,有`link`,`getLine`,`checkError`等公共方法,`AST`的各个节点都继承自`Node`


ASTNode
AddExp
AssignStmt
Block
BlockStmt
BreakStmt
BType
CompUnit
Cond
ConstDecl
ConstDef
ConstExp
ConstInitVal
ContinueStmt
EmptyStmt
EqExp
Exp
ExpStmt
FuncCall
FuncDef
FuncFParam
FuncFParams
FuncRParam
FuncRParams
FuncType
GetIntStmt
IfStmt
InitVal
LAndExp
LOrExp
LVal
MainFuncDef
MulExp
Node
Number
PrintStmt
RelExp
ReturnStmt
UnaryExp
VarDecl
VarDef
WhileStmt

实际上我们隐约能感受到，语法分析的过程实际上就是在`dfs`一颗多叉树，这颗多叉树本质上就是我们的语法树

注意为了能清楚的知道当前节点在树的什么位置，我们要依据文法建立对应的类，然后通过`instance of`继续判断，父节点和子节点能进行双向访问，至此我们就将所有的源程序所有信息存进`AST`，有效地管理起来了

之后的错误处理（除了检查符号缺失），中间代码生成，目标代码生成甚至是优化模块都和语法分析模块没有关系了，实现了真正意义上地解耦

2.符号表设计

建立`SymbolTableEntry`类，并实现针对`函数`，`变量`和`常量`各自的构造器，及相应的查询方法，为了区分不同的表项，建立`DataType`,`DeclType` `枚举类型`分别表示数据类型和声明类型：

`SymbolTable`符号表采用`ArrayList` + `HashMap`的结构：

全局声明

采用`ArrayList`进行存储

变量（常量）

name	DeclTyp	DataType	line	addr	value	length1	length2	size
	var|const	int|int_array_1D|int_array_2D		相对gp的offset				

函数：

name	DeclTyp	DataType	FParam
	func	int|void	

局部声明

采用`HashMap<String,ArrayList>`进行存储

name	DeclType	DataType	layer	line	addr	value	lenght1	length2	size
	param	int|int_array_1D|int_array_2D	0		相对sp的offset		\		4
	var|const	int|int_array_1D|int_array_2D			相对sp的offset				size
	temp	int	\	\	相对sp的offset		\	\	4

```
public enum DataType {
    INT,
    VOID,
    INT_ARRAY_1D,
    INT_ARRAY_2D,
    UNDEFINED,
}

public enum DeclType {
    VAR,
    CONST,
    FUNC,
}
```

3.错误处理设计

非法符号	a	格式字符串中出现非法字符报错行号为**<FormatString>**所在行数。	<FormatString> → “‘{<Char>}’”
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**<Ident>**所在行数。	<ConstDef>→<Ident> ... <VarDef>→<Ident> ... |<Ident> ... <FuncDef>→<FuncType><Ident> ... <FuncFParam> → <BType> <Ident> ...
未定义的名字	c	使用了未定义的标识符报错行号为**<Ident>**所在行数。	<LVal>→<Ident> ... <UnaryExp>→<Ident> ...
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。	<UnaryExp>→<Ident>‘(’[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。	<UnaryExp>→<Ident>‘(’[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。	<Stmt>→‘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] ‘)’ Block MainFuncDef → 'int' 'main' '(' ')' Block
不能改变常量的值	h	<LVal>为常量时，不能对其修改。报错行号为**<LVal>**所在行号。	<Stmt>→<LVal>‘=’ <Exp>‘;’|<LVal>‘=’ ‘getint’ ‘(’ ‘)’ ‘;’
缺少分号	i	报错行号为分号**前一个非终结符**所在行号。	<Stmt>,<ConstDecl>及<VarDecl>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(<UnaryExp>)、函数定义(<FuncDef>)及<Stmt>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(<ConstDef>,<VarDef>,<FuncFParam>)和使用(<LVal>)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。	<Stmt>→‘break’‘;’|‘continue’‘;’

`dfs`根节点即可，相比在递归下降中进行错误处理，全局变量明显减少，参数传递的情况更少出现，重载的`checkError`（需要传参）也都包含在各个类中，并且与语法分析模块无关，实现了解耦

- 对于`i,j,k`类错误直接在语法分析的过程中解决
- 对于`a,l`错误只需对`PrintStmt`中国你相应的语法成分进行检查
- `b,c,d,e,h`:当遇到对应的语法成分时先进行查表，进行异常判断，如果需要写入信息，再考虑将信息写入符号表

如果是函数声明，则通过函数名在`HashMap`中找到用于记录函数体的`ArrayList`，进行类似的查表写入操作，除此之外,碰到`Block`节点层数就增加，出`Block`节点层数就减少等

- `f`:`AST`当遇到`ReturnStmt`就进行检查，如果是`void`类型则不应该返回具体值（可能出现多个`ReturnStmt`）.
- `g`:对于`int`类型的函数直接查看最后一个`Block`的最后一个子节点，如果不是`ReturnStmt`并且有返回`Exp`就异常

- `m`:当遇到`ContinueStmt`,`BreakStmt`时向上不断搜索父节点，如果找不到`WhileStmt`就异常

2.实现细节

每个类都对`checkError`进行重写，部分需要传参的类需要进行重载

对根节点`CompUnit`调用`checkError`方法时，各子节点会自顶向下进行对`checkError`进行递归地调用，当到达需要进行错误处理的节点，先进行该节点负责的错误处理之后递归地向下调用子节点的`checkError`方法

由于是递归的进行错误处理，到部分子节点随时需要查表，并记录错误，个人选择将`ErrorAnalysis`模块中的`getSymbolTable`和`addError`方法设为静态方法

建立`SymbolTable`类,并将查表填表等对符号表的操作封装在符号表内，如果发生异常，则返回失败标志，在调用的地方进行错误的记录，注意查表的时候要注意相应的作用域

建立`Error`类，建立`ErrorType`枚举类型，构造器中能根据枚举类型自动填充`msg`错误信息，方便debug，重载`compareTo`方法，方便排序后输出有序的答案,方便`debug`

四.中间代码生成

1.模块设计

由于之前已经建了`AST`,所有信息都存在了对应的节点上，能够很方便的取出来，因此只需要遍历`AST`，如果遇到了"需要产生中间代码"地节点，相应地生成即可

创建`MidCodeGener`模块，将`AST`传入此模块,建立`MidCodeEntry`类，以方便中间代码的管理，具体属性包括”四元式“的各个操作数

最终输出`ArrayList<MidCodeEntry>`

中间代码采用”四元式“的形式：

Name:全局变量的名字

FuncName:函数的名字

T此处可能出现临时变量

V:此处可能出现变量

C:此处可能出现常量

():括号中的内容可能会出现可能不出现

中间代码采用"四元式"的形式：

op	r1	r2	r3	dst
GLOBAL_DECLARE				Name
FUNC_DECLARE				FuncName
PUSH_PARAM	T/V/C	paramNum		FuncName
STORE_RET				T
LOAD_ARRAY_1D	ArrayName	T/V/C		T
STORE_ARRAY_1D	ArrayName	T/V/C		T/V/C
LOAD_ARRAY_2D	ArrayName	T/V/C	T/V/C	T
STORE_ARRAY_2D	ArrayName	T/V/C	T/V/C	T/V/C
LOAD_ADDRESS	ArrayName	(T/V/C)		T
ASSIGN	V			T/V/C
PRINT_STRING				PrintContent
PRINT_INT				T/V/C
RET_VALUE				T/V/C
RET_VOID				
GETINT				T
PREPARE_CALL				FuncName 保护ra s,t a
CALL				FuncName 拉栈跳转
FIN_CALL				FuncName 还原现场
ADD	T/V/C	T/V/C		T
SUB	T/V/C	T/V/C		T
MULT	T/V/C	T/V/C		T
DIV	T/V/C	T/V/C		T
MOD	T/V/C	T/V/C		T
NEG	T/V			T

op	r1	r2	r3	dst
SLT	T/V/C	T/V/C		T
SLE	T/V/C	T/V/C		T
SGT	T/V/C	T/V/C		T
SGE	T/V/C	T/V/C		T
SEQ	T/V/C	T/V/C		T
SNE	T/V/C	T/V/C		T
NOT	T/V			T
LABEL_GEN				label
BEQZ	T/V/C			label
BNEZ	T/V/C			label
GOTO				label

2.实现细节

具体做法是在每个节点类中实现`genMidCode`方法，对`root`节点调用`genMidCode`，就能递归地产生所有中间代码了(本质上是把信息的存储方式转换一下,方便后续处理)，遍历顺序实际和递归下降地顺序相同

五、目标代码生成

由于中间代码已经产生了所有的中间代码组成的列表，将此`ArrayList`传入`TargetCodeGener`模块，进行目标代码的生成，逐条翻译每一条中间代码为目标代码

REGISTER	NAME	USAGE
0|zero	常量0	
1|at	保留给汇编器	
$2-3$	$v0-$v1	函数调用返回值
$4-7$	$a0-$a3	函数调用参数
$8-15$	$t0-$t7	临时寄存器
$16-23$	$s0-$s7	全局寄存器
$24-25$	$t8-$t9	临时寄存器
28|gp	全局指针(Global Pointer)	
29|sp	堆栈指针(Stack Pointer)	
30|fp	帧指针(Frame Pointer)	
31|ra	返回地址(return address)	

1.模块设计

在此阶段我除了必须的寄存器只使用了`t0,t1,t2,t3`寄存器进行运算，所有的变量都通过内存访问

全局变量的处理

首先需要弄懂这个`gp`寄存器是用来干什么的：由于`lw`,`sw`等指令中最后`16`位为立即数的长度，也就是说只能访问到某个地址前后的`-32~32`KB的空间，那么在运行过程中想要直接访问全局变量只借助`sp`是不行的

所以我们把`gp`固定在一个位置，这个位置要刚好在这64KB的中间，实现对前后32KB的全局变量的随机访问，`mars`中`gp`被初始化为`0x10008000`，不难想到`0x10000000~0x10010000`，刚好就是这`64`KB，所以我们需要从`0x10000000`开始存储全局变量，然后通过`gp`进行访问，如果超过`64`KB，`mars`会自动进行指令分解，不会产生错误

但是`mars`中.data默认不是从`0x10000000`开始的，经过本人研究发现以下做法可以改变起始位置

```
.data 0x10000000
a:.word 3

b:.word 0

c:.word 1,2,3,4,5
```

令人欣喜的是，这些指令都是不算入竞速指标的，此外相比于开头直接更改`gp`也更加优雅、合理，显然这是处理全局变量的最优策略

函数调用的处理

编译阶段：

- 所有的`常量`，`全局变量`预先算出 存到对应的符号表里
- 所有数组的length(`1D` `2D`)预先算出 放到对应的符号表里

- `const`进行替换

运行阶段：

- 对于局部变量：
    - 如果是数组全部放在栈里（`local`），用的时候从对应的`addr`取，算完之后立马存到对应的`addr`
    - 如果是局部变量或者临时变量，在此时通过内存进行使用：使用时`load`出来，更新后`save`回去
- 函数调用过程：(注意sp改变时符号表里的offset要相应变化)
    - 进函数前：

      1.保护 ra

      2.保护已经使用的`reg`（此时没有）

      3.sp -- funcsize

      4.跳转 jal
    - 返回前:

      1.存储v0

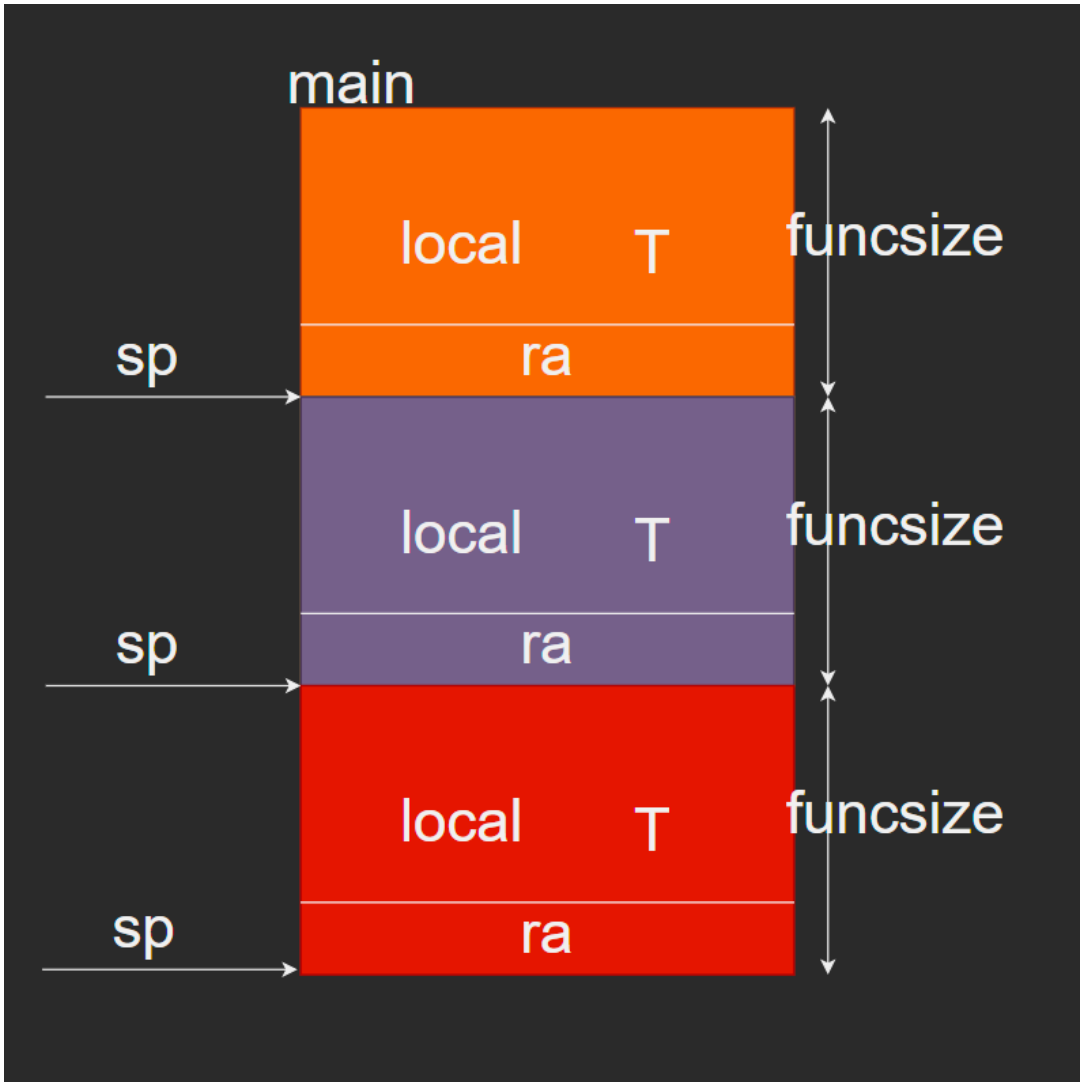
      2.返回 jr
- 函数调用返回后：

  1.sp ++funcsize

  2.之前保护的`reg`(此时没有)

  3.还原 ra

  具体内存结构



短路处理：

为了避免结构过于复杂，并且兼顾一定的性能，按照预算的优先级，我进行了如下定义：

`Cond`：具有原子性，只可能出现0/1两种情况，由 `>=` `==` 等表达式自左向右运算得到，不使用跳转语句处理以上表达式

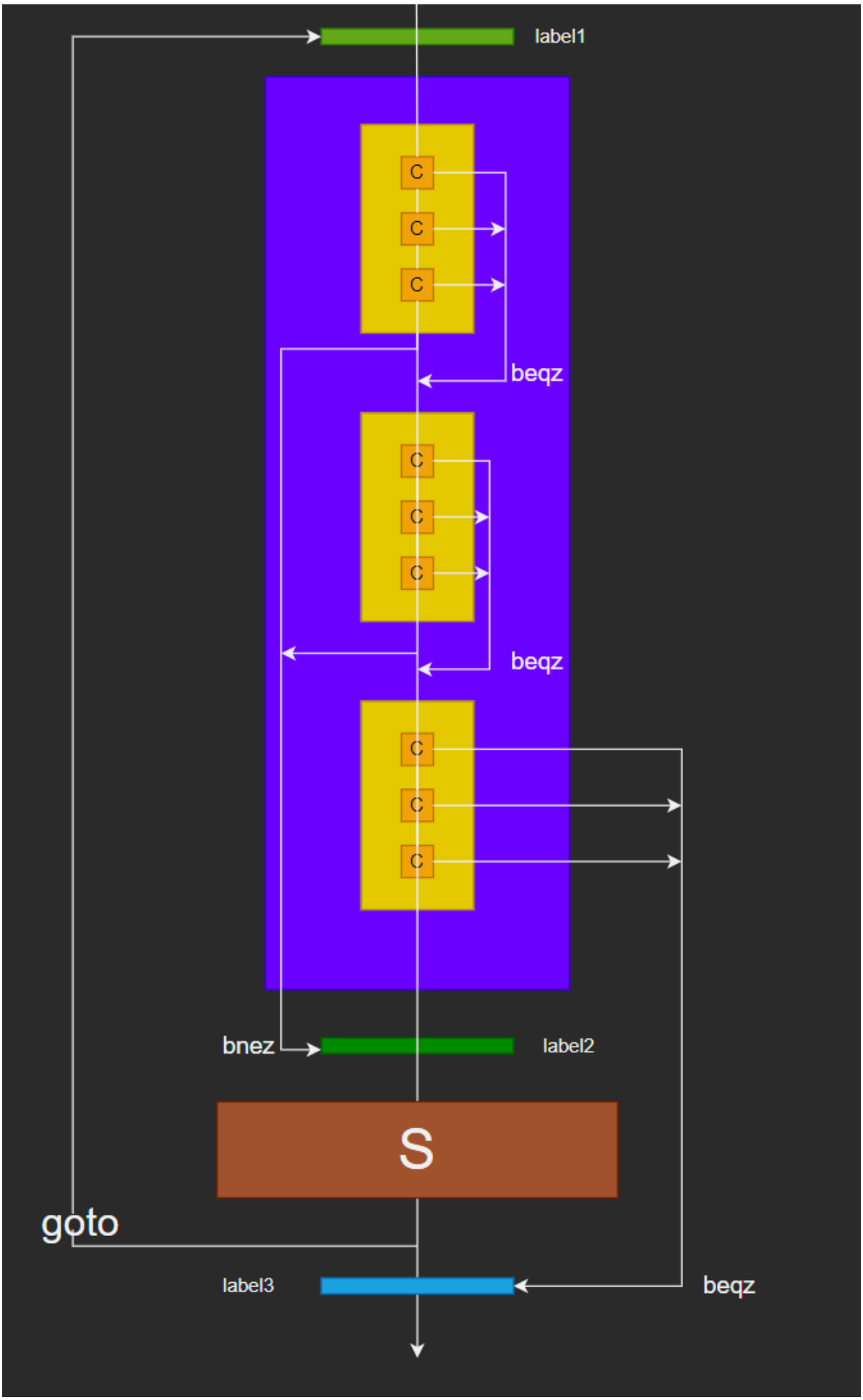
`And` 块：里面的元素均为 `Cond`，`&&` 连接相邻的 `Cond`

`Or` 块：里面的元素均为 `And` 块，`||` 连接相邻的 `And` 块

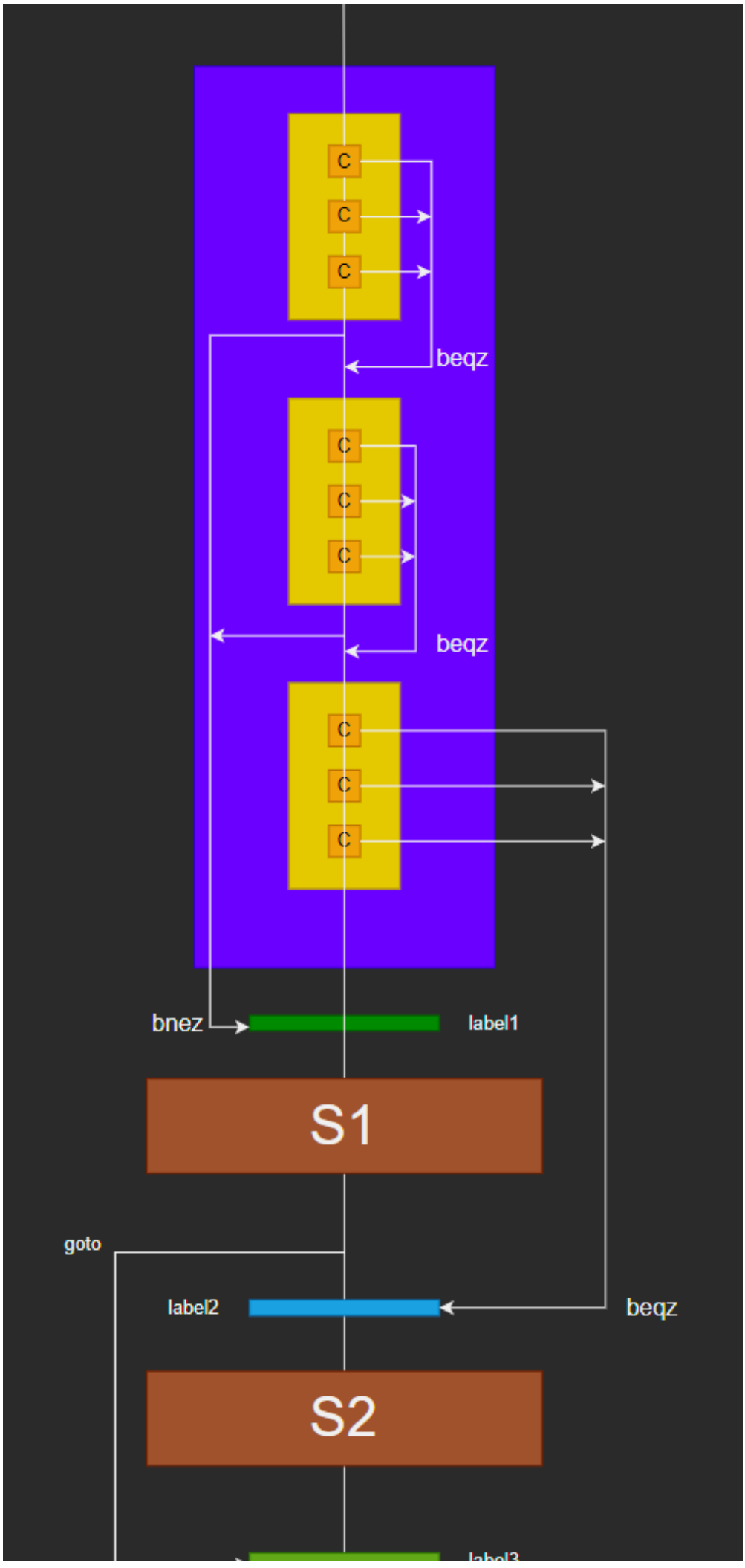
通过观察可以发现一旦 `And` 块中只要有一个 `Cond` 为 `0` 就可以直接跳到下一个 `And` 块了，如果 `And` 块中当前 `Cond` 是 `1`,则考察下一个 `Cond` 的值，`if`,`while` 和中类似

需要注意的是 `And` 块和 `Or` 块中凡是最后一个元素均要进行特殊处理，以避免冗余的 `label` 产生影响后续模块，具体处理方式是设置标记，在生成相应代码的时候查询父节点的标记

while


label1
C
C
C
beqz
C
C
C
beqz
C
C
C
bnez
label2
S
goto
label3
beqz

if


C
C
C
beqz
C
C
C
beqz
C
C
C
bnez
label1
S1
goto
label2
beqz
S2

2.实现细节

`MidCodeEntry`需要实现`toTargetCode`方法，将不同的中间代码翻译并输出到文件即可，注意翻译的过程要注意封装函数，因为`mips`汇编逻辑比较繁琐，否则会出现各种各样的`bug`，之后改成使用`reg`也不好过渡，最后预留接口

```
public String toTargetCode() {
    curCode = "";
    String start = String.format("##############") + toString() +
String.format("##############\n");
    String end = "\n";
    curCode += start;
    if (opType == OpType.GLOBAL_DECLARE) {
        genStoreGlobal();
    }else if (opType == OpType.FUNC_DECLARE) {
        MidCodeGener.startFuncDef(dst);
        genFuncLabel();
    } else if (opType == OpType.PUSH_PARAM) {
        genPushParam();
    } .....
    curCode += end;
    return curCode;
}
```

三.优化部分

优化的主要目标是运行的更快，主要涉及到中间代码和目标代码的优化，我所做的优化包括以下内容

我花了很久去看书上的内容，开始想将各种优化都做到尽量完美，直到上手才发现书上讲的和实验做的难度差异略大，限于时间原因，我采取性价比最高的优化方案，即部分优化方法没有完全和书本所教的一样，虽然效果上可能会略差，但更容易实现，性价比更高，最终总体优化效果较令人满意

1.划分基本块

由于中间代码之间缺少了执行控制流等信息，我们需要划分基本块，基本块可以说是优化的基础，之后很多优化中间代码的操作都是建立在中间代码的基础之上的，我划分基本块的方法如下：

- 跳转型中间代码`goto`,`beq`,`bne`的下一句为入口点，`return,exit`的下一句为入口点
- `label`型中间代码为入口点,考虑到可能出现连续的`lablel`，将所有连续的`label`当作一个入口点
- 每个函数有一个`FuncBlock`,函数下的`BasicBlock`放在`FuncBlock`中，每个`BasicBlcok`下包含多条中间代码，不包括产生`label`的中间代码，声明全局变量和函数的中间代码.`FuncBlock`还包括`HeadBlock`,`EndBlock`，均继承自`BasicBlcok`用来标记起止位置，对后面做数据流分析也提供了方便
- 对于不可达基本块（没有前驱），需要直接删去

2.数据流分析

我们需要知道为什么需要做数据流分析，我个人的理解就是要弄清楚在某个地方使用的某个变量它是从哪来的（在哪定义的），经过某个地方之后，后面还会不会用到这个变量

活跃变量分析

活跃变量需要从后往前算，如果`in`集合中存在某个变量，说明在此基本块（或者中间代码）及以后，某个基本块（或者中间代码）会用到此变量，如果这个变量在寄存器中，那么这个寄存器在这条路径上就不能被其他的变量再次使用，如果经过某个基本块显然，我们可以利用活跃变量分析不同变量之间的冲突关系，根据此分配寄存器

由于已经建立了基本块，接下来以基本块为单位算出`def`,`use`集合，通过不断迭代直到每个基本块的`in`,`out`也收敛，然后以中间代码为单位算出`def`,`use`集合（实际操作时可以和算基本块的`def`,`use`同时进行），在基本块内不断迭代使得中间代码的`in`,`out`收敛

中间代码的`def`,`use`：

op	r1	r2	r3	dst
PUSH_PARAM	use			
STORE_RET				def
LOAD_ARRAY_1D		use		def
STORE_ARRAY_1D		use		def
LOAD_ARRAY_2D		use	use	def
STORE_ARRAY_2D		use	use	def
LOAD_ADDRESS		(use)		def
ASSIGNV	def			use
PRINT_STRING				
PRINT_INT				use
RET_VALUE				use
RET_VOID				
GETINT				def
PREPARE_CALL				
CALL				
FIN_CALL				
EXIT				
ADD	use	use		def
SUB	use	use		def
MULT	use	use		def
DIV	use	use		def
MOD	use	use		def
NEG	use			def
SLT	use	use		def

op	r1	r2	r3	dst
SLE	use	use		def
SGT	use	use		def
SGE	use	use		def
SEQ	use	use		def
SNE	use	use		def
NOT	use			def
BEQZ	use			label
BNEZ	use			label

3.图着色

经过活跃变量分析，已经知道了哪些变量之间会发生冲突（不能同时使用一个寄存器），具体来说就是如果两个变量同时出现在某个中间代码的`out`集合中，那么他就是冲突的，当然还可以细化变量之间的冲突关系，但这就需要定义到达分析了，限于时间本人没有完成，只做了基本的图着色，效果也还行

有了冲突图之后按照书上的启发式算法进行寄存器分配，最终得到`变量->寄存器`的对应关系表,之后生成目标代码决定是否分配寄存器，分配哪个寄存器的时候查询这个对应关系表就行了

我并没有严格区分`s`,`t`寄存器的使用，前面也提到了我把`T/V`看作地位相等，除了预留`a0,v0,ra,sp,gp,t0,t1,t2,t3,0,at`等寄存器，其他寄存器全部进入寄存器池

另外需要注意，为了避免过于复杂，我没有对全局变量分配寄存器，参数、局部变量、临时变量都是分配寄存器的对象

4.常量折叠

对于表达式计算，如果两个操作数是常数/常量（`const`)显然可以提前在编译阶段算出来，例如：

```
const int a = 3;
//原代码
int x = a + a;

//中间代码
ASSIGN x3 6
```

5.常量传播/复写传播

限于时间原因，我做的是基本块内的常量传播和复写传播，对于常量传播，它与常量折叠最大的区别在于，常量折叠不需要数据流分析，本质上是一种窥孔优化，必须要求操作数都是常量(`const`)或者`Number`，对于以下情况不能做出优化

```
int a = 3;
int b = a + 3;
//假设后面都不会用到a
//不能优化为中间代码
ASSIGN a 3
ASSIGN b 6
```

但是常量传播可以，在每个基本块中建立一个`spreadMap`，顺序遍历基本块中的中间代码：

如果某个中间代码的`def`中出现某个变量`T/V`，那么就建立映射`T/V -> val`，`val`可以是`C`

同理复写传播也类似，只是`val`可能是`T/V/C`

具体算法是：

- 顺序遍历中间代码，检查如果该中间代码使用到的变量在`spreadMap`中存在映射就用`val`替换
- 如果该中间代码产生了`def`,如果该`def`在`spreadMap`中存在，便更新映射；如果不存在便添加映射

6.死代码删除

常量传播/复写传播的真正目的是删除死代码（基本块内），所谓死代码就是指存在于程序中，但执行了却和没执行效果一样，可以选择不执行的代码，例如

```
int main() {
    int x = 3+ 3 * 5;
    int z = 3;
    int y = x*z + x*z + x * z + 7;
    int i = 1;
    printf("%d",y);
    return y;
}
//优化前
FUNC_DECLARE main
ASSIGN x5 18
ASSIGN z6 3
MULT x5 z6 @T_0
MULT x5 z6 @T_1
ADD @T_0 @T_1 @T_2
MULT x5 z6 @T_3
ADD @T_2 @T_3 @T_4
ADD @T_4 7 @T_5
ASSIGN y7 @T_5
ASSIGN i8 1
PRINT_INT y7
EXIT
//优化后的中间代码
FUNC_DECLARE main
PRINT_INT 169
EXIT
```

之前做过活跃变量分析，如果某个中间代码的产生的`def`在`out`中不存在，那么可以认为该中间代码为死代码，当然某些特殊的中间代码需要特殊考虑：

- 全局变量的存储
- 函数调用涉及到的中间代码

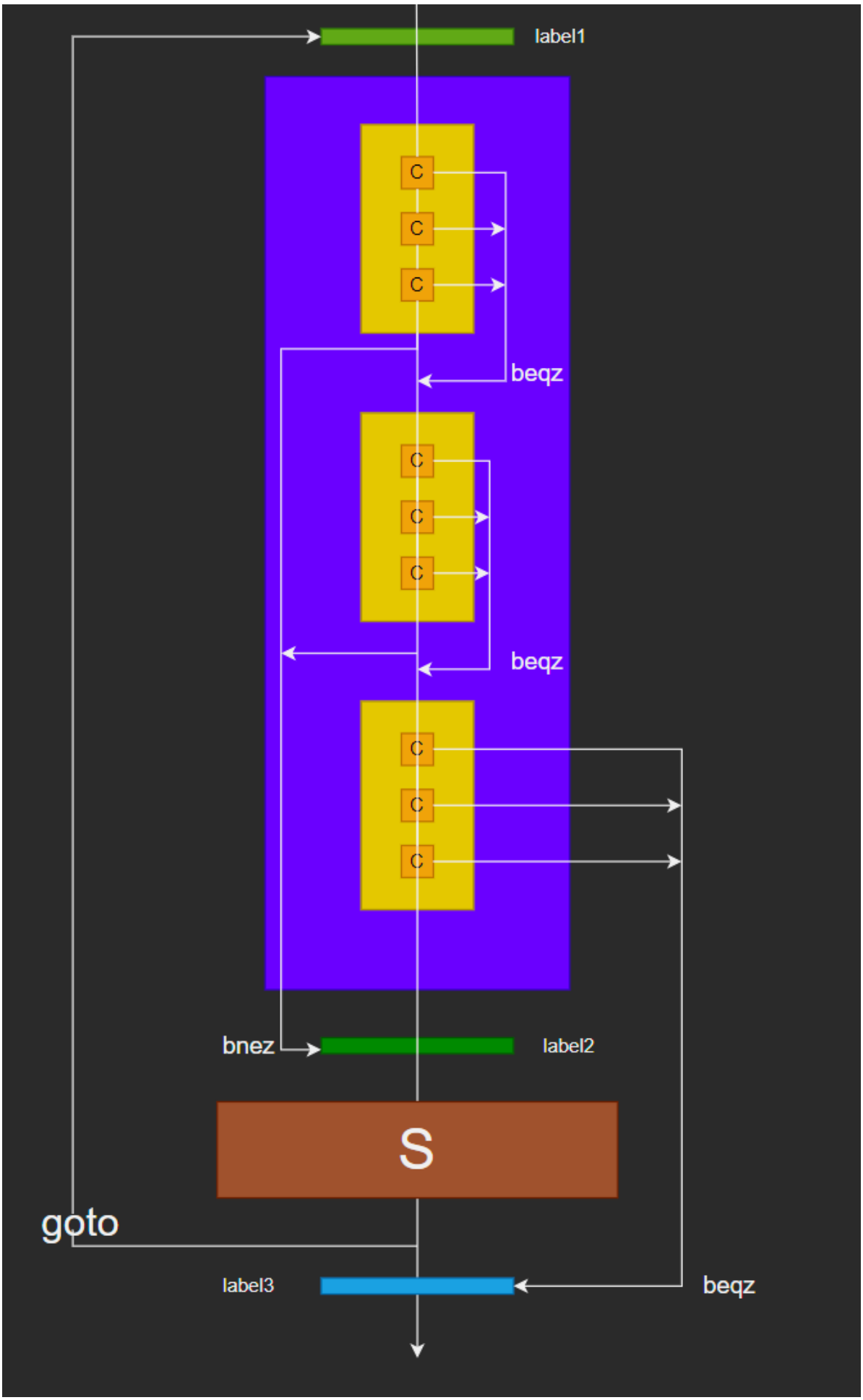
- 输入语句
- 输出语句
- 跳转语句
- label生成

可能满足删除条件的中间代码如下所示：

op	r1	r2	r3	dst
STORE_RET				T
LOAD_ARRAY_1D	ArrayName	T/V/C		T
LOAD_ARRAY_2D	ArrayName	T/V/C	T/V/C	T
LOAD_ADDRESS	ArrayName	(T/V/C)		T
ASSIGN	V			T/V/C
ADD	T/V/C	T/V/C		T
SUB	T/V/C	T/V/C		T
MULT	T/V/C	T/V/C		T
DIV	T/V/C	T/V/C		T
MOD	T/V/C	T/V/C		T
NEG	T/V			T
SLT	T/V/C	T/V/C		T
SLE	T/V/C	T/V/C		T
SGT	T/V/C	T/V/C		T
SGE	T/V/C	T/V/C		T
SEQ	T/V/C	T/V/C		T
SNE	T/V/C	T/V/C		T
NOT	T/V			T

7.while结构优化

原while结构：



可以看到每次在执行完 `s` 之后需要无条件 `j` 回 `label1` 进行循环条件是否仍满足的判断，但如果在 `s` 执行完之后再执行一遍紫色框住的部分就可以省去一条 `j`，令人欣喜的是任何地方的每一次while循环都能省一次 `j` 指令，优化是巨大的，用C表达类似于：

```
if(!cond) {
    do {

    } while (cond);
}
....
```

8.窥孔优化

中间代码

- 加减乘除模中操作数为0，+1，-1的处理

目标代码

- 表达式计算时，如果有一个操作数是常数，如果存在`i`型指令例如：`addiu`,`slti`等，不要用`li`+`addu`或`li`+`slt`
- 用`addiu`代替`subiu`，`slt`+`seq`代替`sge`
- 乘法优化：
    - 用`mul $t3,$t2,$t1`
    - 如果两操作数中有一个存在常数则检查是否为2的次幂，如果是用`sll`代替
- 除法优化：
    - 用`div $t2,$t1,mflo $t3`,可以避免除数为`0`检查耗费的指令
    - 如果除数是`2`的次幂，可以进行优化，注意`mars`中被除数的正负会影响到运算结果，简单来说被除数是负数时行为与给定预期不同，如果被除数是负数，需要将被除数加上`2^k - 1`（判断正负用分支指令）；除数的正负可以通过设置标志，最终将商取反，显然如果两次取反，可以不取反
    - 经查阅资料发现，如果除数不是`2`的次幂，而是一些特殊数也可以进行优化，如下表：

  中间一列是`magic_number`，右边一列是`shift(s)`

	Signed	
d	M(hex)	s
-5	99999999	1
-3	55555555	1
-2^k	7FFFFFFF	k-1
1	-	-
2^k	80000001	k-1
3	55555556	0
5	66666667	1
6	2AAAAAAB	0
7	92492493	2
9	38E38E39	1
10	66666667	2
11	2E8BA2E9	1
12	2AAAAAAB	1
25	51EB851F	3
125	10624DD3	3
625	68DB8BAD	8

但是需要注意如果`magic_number`太大了，可能会发生溢出

所以我只选择特判下面的正数（不包括7）

```
li dividend,magic_number
mfhi dst
sra dst,dst,s
bgtz dst,label
addiu dst,dst,1
label:
```

- 模优化：
  - 一个直观的做法是将除法和乘法优化结合，被除数减去商和除数的积就是模
  - 这里需要注意被除数和商可能分配了同一个寄存器，这种情况下要提前保存被除数对应寄存器中的值
- 计算地址时进行优化：
  - 一维地址 $addr = base + i * 4$,如果`i`是常数，只需要一条`addiu`
  - 二维地址 $addr = base + (i * l + j) * 4$，分别讨论`i`,`j`是否常数 并进行常数优化

9.无用循环去除

由于进阶死代码删除实现上比较困难，但是不得不承认，根据公开的`test1`,存在一种循环，循环体中不断修改的变量,但在后面根本没有使用到

```
int d = 4;
int main () {
    int i = 2,j = 5;
```

```
    i = getint();
    j = getint();
    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;
}
```

可以考虑去掉无用循环，注意这种循环结构拥有`唯一`前驱和后继（不算本身）

具体做法是先找到循环体对应的基本块，如果基本块中存在以下中间代码就不考虑去除：

- 输出
- 输入
- 函数相关
- 更改了全局数组

我的架构中凡是要改变变量（局部变量和全局变量）的值必须通过`ASSIGN`，所以对于循环中的`ASSIGN`要分两种情况考虑

具体方法为：

- 如果`ASSIGN`对局部变量赋值，该局部变量不能出现在循环块的唯一后继的`in`中
- 如果`ASSIGN`对全局变量赋值，该全局变量不能出现在循环块`以后`所有的中间代码的`use`中
- 如果`ASSIGN`对全局变量赋值且碰到函数调用，遍历被调用函数的所有中间代码，中间代码的`use`中不能有该全局变量，由于可能再次调用函数，所以需要递归的进行检查