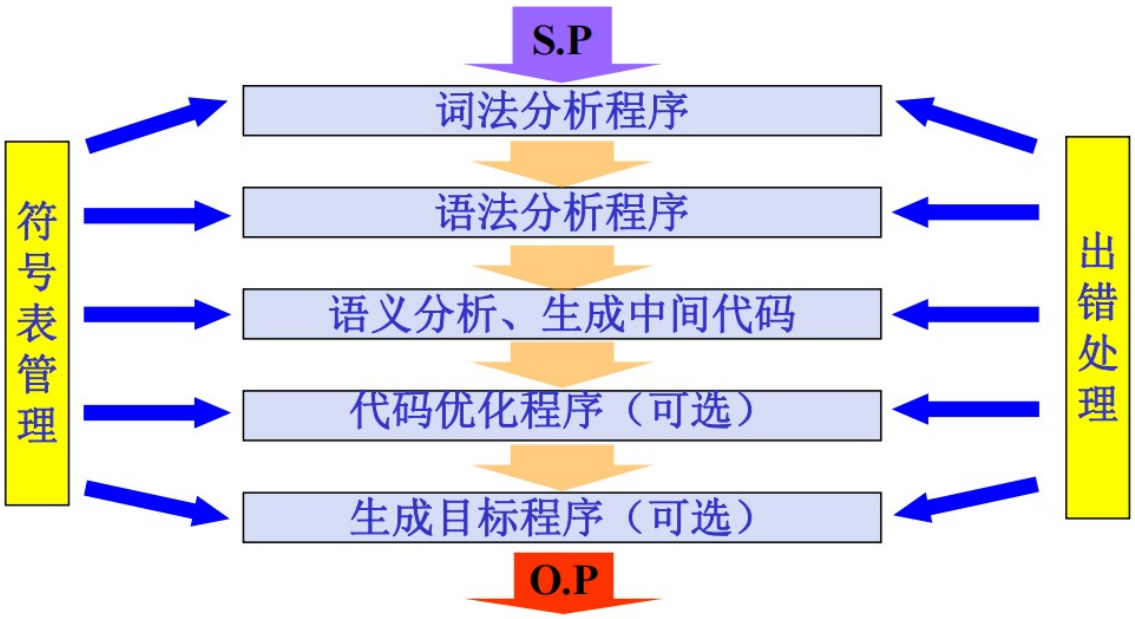
一、参考编译器介绍

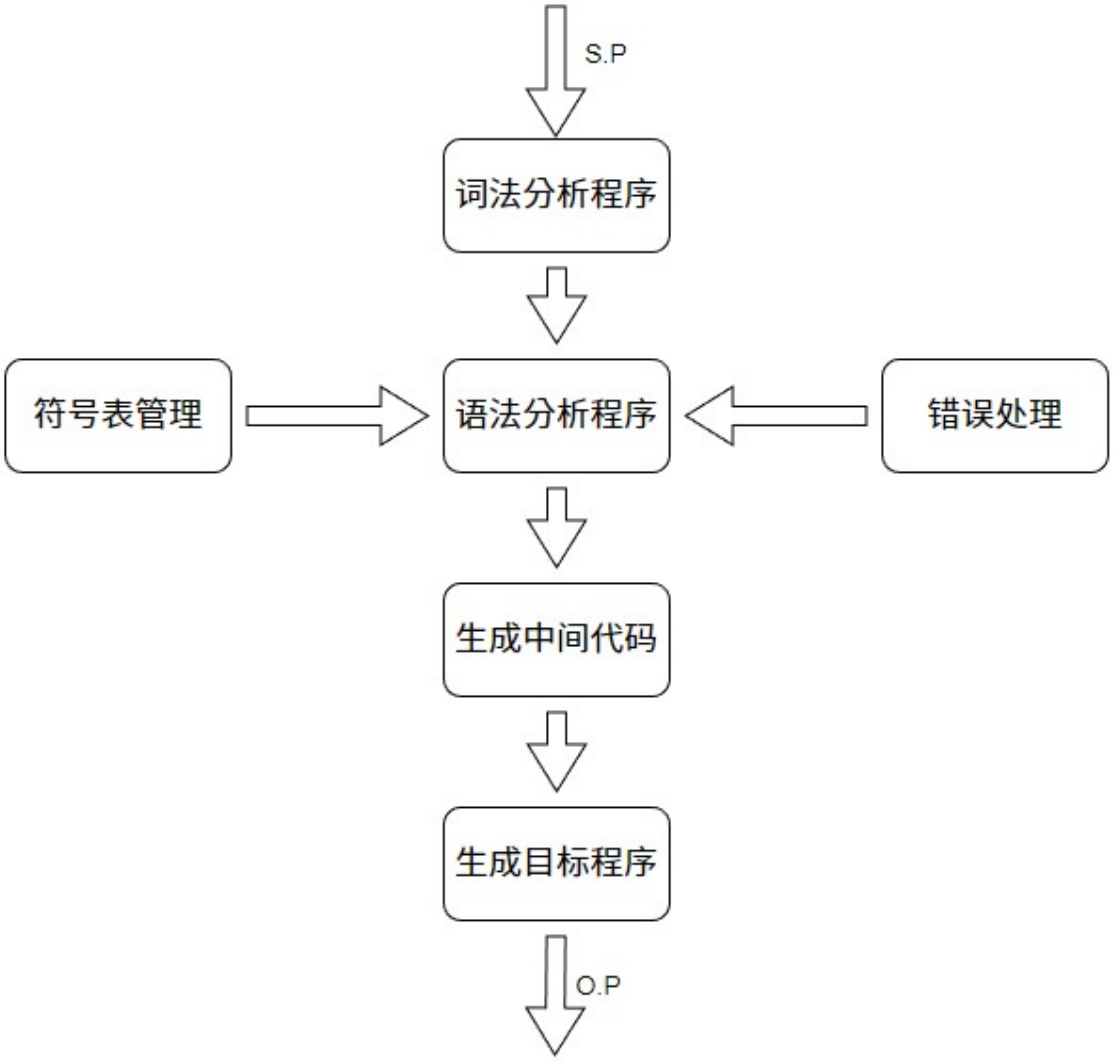
总体结构



我参考了手册、pdf、上学期oo课所学到的递归下降结构以及上一届学长的中间代码设计，我并没有阅读一个完整的编译器，很多地方都是以自己目前的了解先写，写着写着发现了很多问题，当时想的解决方法也不长远，因此在debug的路上越走越远，这也让我长了经验，以后写代码会考虑的更多，同时需要参考一些编译器代码结构，让自己有个良好的整体思路。

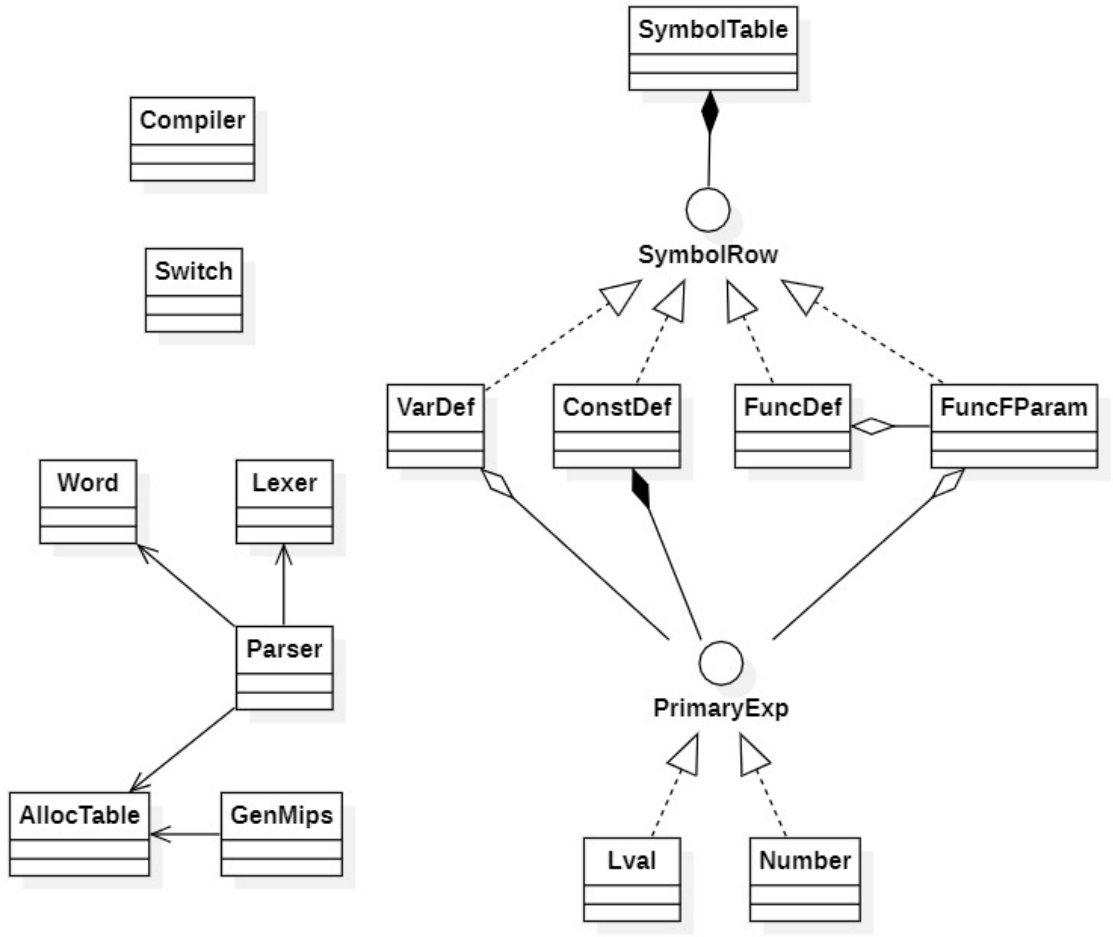
二、编译器总体设计

总体结构



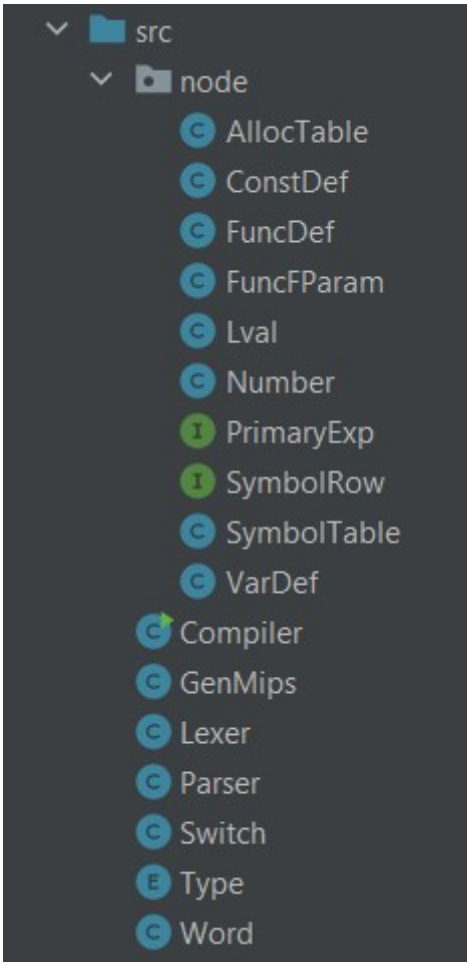

```
词法分析程序：在Lexer类分析。
语法分析程序：在Parser类分析。
生成中间代码：在Lexer类生成。
生成目标程序：在GenMips类生成。
符号表管理：在Parser类建立符号表并管理。
错误处理：在Parser类找出错误并处理。
```

接口设计




```
Lval类：变量，实现PrimaryExp接口。
Number类：常量，实现PrimaryExp接口。
PrimaryExp接口：基本表达式。
VarDef类：变量定义，实现SymbolRow接口。
ConstDef类：常量定义，实现SymbolRow接口。
FuncDef类：函数定义，实现SymbolRow接口。
FuncFParam类：函数形参，实现SymbolRow接口。
SymbolRow接口：符号表中的一行。
SymbolTable类：符号表，记录标识符，类型，层号等，供查找。
AllocTable类：分配空间表，记录全局以及每个函数里所有数组需要的空间大小。
```

文件组织



node文件夹下的是依赖于递归下降所建立的节点(除AllocTable，SymbolRow，SymbolTable之外)，记录了一些值以方便中间代码的生成。
Compiler类：程序运行的入口。
GenMips类：中间代码生成MIPS代码。
Lexer类：词法分析。
Parser类：语法分析。
Switch类：生成txt文件的开关。
Type类：是一个枚举类，所有类别码的类型，可省略，没用到。
Word类：在词法分析时，存储单词的行号，类别码，字符串。

三、词法分析设计

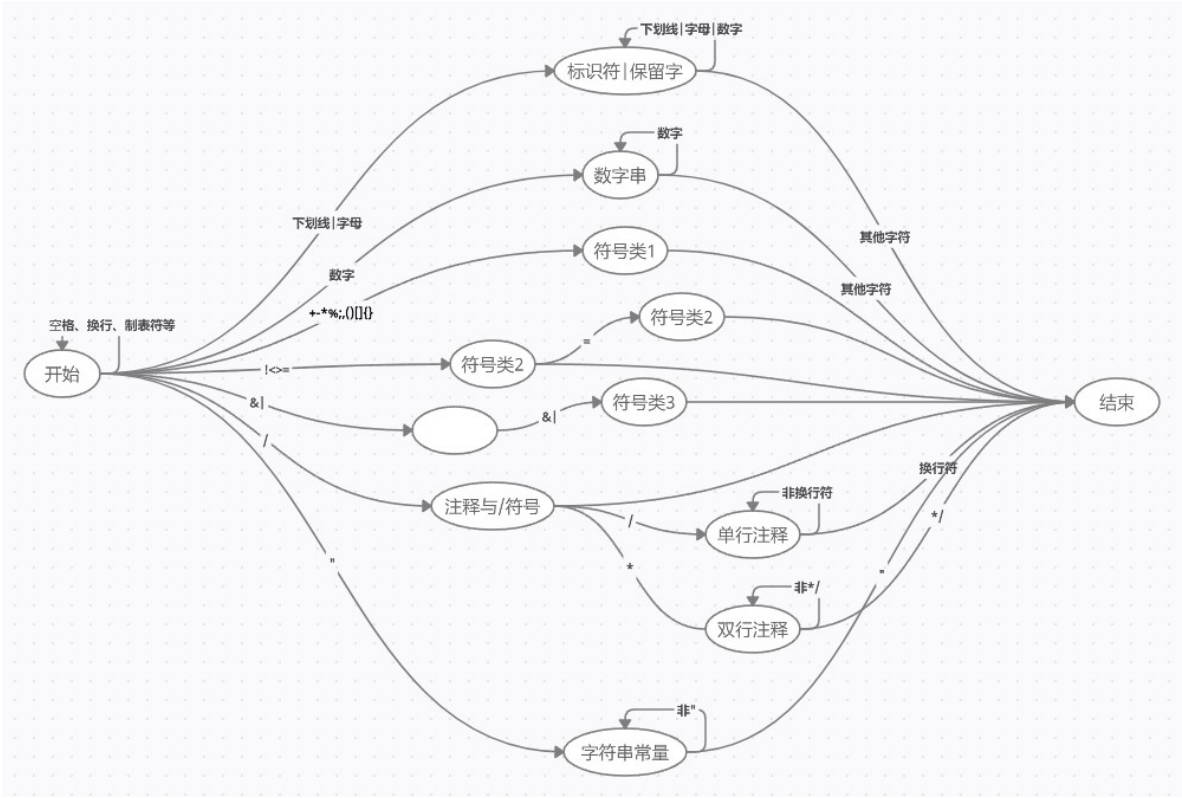
编码前的设计

建立Word类如下，存取所检索到的单词名称，类别码以及行号。

```
public class Word {
    private int line;
    private String categoryCode;
    private String name;
    ......
}
```

在Lexer类建立reservedWord表保存保留字名称和对应的类别码，方便获取所检索到的单词所对应的类别码，同理symbol表保存符号名称和对应的类别码。

根据下图开始编码。



完成词法分析需要进行两遍，第一遍是一次性读取`testfile.txt`内容并交给`Lexer`类处理，第二遍是`Lexer`类进行处理以识别所有单词。

编码完成之后的修改

没修改，但是遇到了一些bug。

bug：

- '/'符号和符号类1放在一起了，应该和注释放在一起判断。
- 判断条件写的逻辑出现错误`!(input.charAt(pos) == '*' && input.charAt(pos + 1) == '/')`写成`(input.charAt(pos) != '*' && input.charAt(pos + 1) != '/')`
- pos++忘记了

四、语法分析设计

编码前的设计

使用递归下降分析法编码。

文法较多较复杂，先把简单的文法完成。

剩下复杂的文法会遇到的问题以及解决方法如下：

- 对于左递归问题，举例：

  将`AddExp => MulExp | AddExp ('+' | '-') MulExp`改写为`AddExp => MulExp {('+' | '-') MulExp}`，以消除左递归，需要注意的地方在于`<AddExp>`的输出，不要遗漏

- 对于FIRST相交问题，可以使用超前扫描，举例：

```
CompUnit => {Decl} {FuncDef} MainFuncDef
VarDecl => BType VarDef { ',' VarDef } ';'
        => 'int' Ident { '[' ConstExp ']' } { ',' VarDef } ';'
FuncDef => FuncType Ident '(' [FuncFParams] ')' Block
        => 'int' Ident '(' [FuncFParams] ')' Block
MainFuncDef => 'int' 'main' '(' ')' Block
```

可以发现进入VarDecl、FuncDef和MainFuncDef分支的前提都是'int'。因此可以预读下一个单词，若为'main'则进入MainFuncDef分支，若为Ident需再预读下一个单词，若为"[,;"之一则进入VarDecl，若为'('则进入FuncDef分支

- 对于不知道要进入哪条分支问题，举例：

```
Stmt => LVal '=' Exp ';' | LVal '=' 'getint''('')'';' | [Exp] ';'
```

可以pos++直到遇到'='或';'，若先遇到'='则进入LVal分支；若先遇到';'则进入[Exp];分支。**使用当前这个方法的前提是LVal和Exp无法推出'='和';'**

- 对于可有可无的问题，如`'return' [Exp] ';'`的`[Exp]`，可以建立如下函数（`FIRST(<Exp>)`）来判断是否有`Exp`从而决定是否进入`Exp`分支。

```
private boolean isExp() {
    return word.getCategoryCode().equals("LPARENT") ||
            word.getCategoryCode().equals("IDENFR") ||
            word.getCategoryCode().equals("INTCON") ||
            word.getCategoryCode().equals("PLUS") ||
            word.getCategoryCode().equals("MINU") ||
            word.getCategoryCode().equals("NOT");
}
```

完成语法分析需要进行两遍，第一遍是一次性读取`testfile.txt`内容并交给`Lexer`类处理，第二遍是`Parser`类向`Lexer`类发起取单词请求，`Lexer`类识别单词并返回给`Parser`类。

编码完成之后的修改

不需要输出终结符即`<Ident>, <IntConst>, <FormatString>`。我忽略掉了，把它们都输出出来了，在这上面做了修改。除此之外，在设计上没做修改，但是遇到了一些bug。

bug：

- 忘记输出`<PrimaryExp>`
- 忽略了`constInitVal(initVal)`的'{'，以至于将`constInitVal(initVal)`改写了
- `word.getCategoryCode().equals("LPARENT")`写成了`word.getCategoryCode().equals("(")`
- `Stmt => Block => '{' { BlockItem } '}'`

```
private void _stmt() throws IOException {
    if (word.getCategoryCode().equals("LBRACE")) {
        getWord();
        _block();
    }
    ......
}
private void _block() throws IOException {
    if (word.getCategoryCode().equals("LBRACE")) {
        getWord();
        ......
    }
}
```

_stmt函数不小心getWord()了，导致进入_block函数缺失"{"

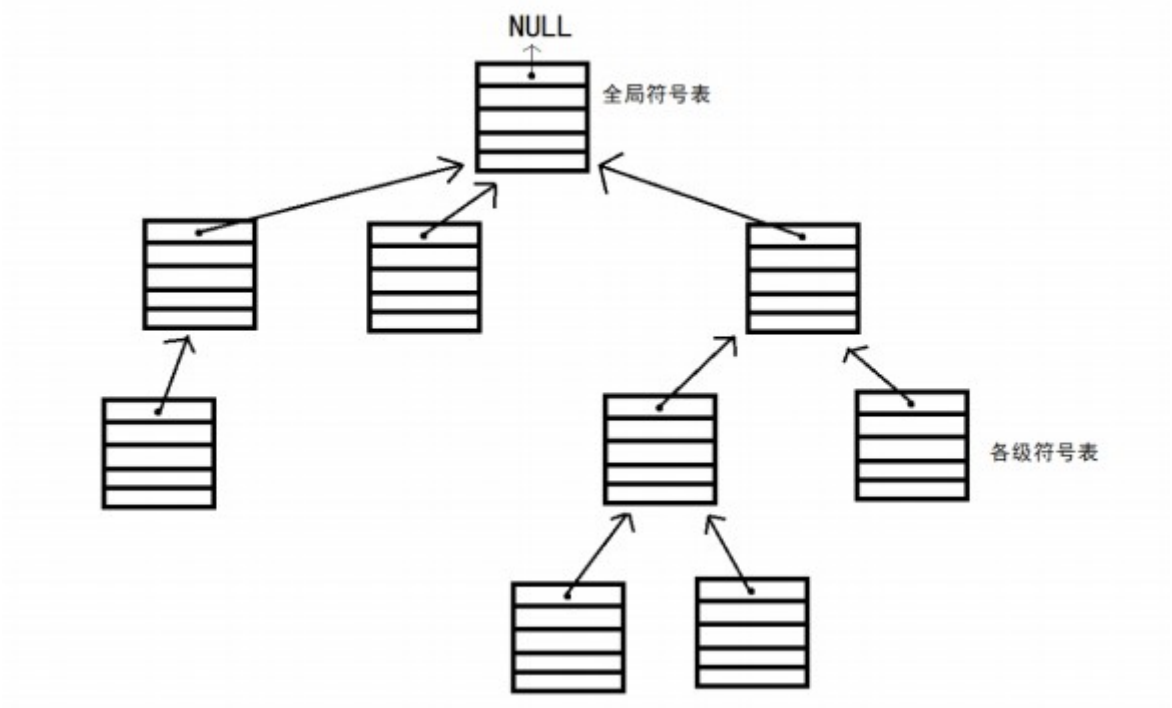
- 在判断`isStmt`时漏了`continue`

五、错误处理

编码前的设计

符号表

使用了参考手册提供的方案1创建了符号表。



符号表中的某一行可以是`ConstDef`，`VarDef`，`FuncDef`，`FuncFParam`类的其中一个，即`ConstDef`，`VarDef`，`FuncDef`，`FuncFParam`类实现`SymbolRow`接口。

错误处理

具体思路：建立全局变量存对应的错误行号，若发现错误立即输出行号与错误类别码。

错误a：一个接一个地检查字符，若出现非法字符则抛出a类错误。

错误b：查本级符号表同时计数，若查找成功则+1，次数大于等于2时抛出b类错误。

错误c：从本级符号表开始，自底向上查找，若查找失败则抛出c类错误。

错误d+e：函数形参与实参数量不等则抛出d类错误。函数形参与实参类型不同则抛出e类错误，需考虑的类型有变量、一维数组、二维数组、函数调用的返回值。

错误f+g：记录目前函数的返回值类型(isReturn)以及是否有返回值(hasReturnValue)，即`return Exp;`语句，目前函数结束时判断错误f和g。

```
// f
if (!isReturn && hasReturnValue) {
    writeError(line, "f");  // 抛出f类错误
}

// g
if (isReturn && !hasReturnValue) {
    writeError(line, "g");  // 抛出g类错误
}
```

错误h：从本级符号表开始，自底向上查找，若目前的`lval`属于`ConstDef`类则抛出h类错误。

错误ijk：在语法分析过程中，若缺少`;`则抛出i类错误，j类错误和k类错误同理。

错误l：扫描`FormatString`语句，若`%d`个数与`Exp`个数不同则抛出l类错误。

错误m：定义一个全局变量`isLoop`，进入`while`时`isLoop`赋值为`true`，出去`while`时`isLoop`赋值为`false`。若使用`break`或`continue`语句且`isLoop`为`false`则抛出m类错误。

编码完成之后的修改

- 对错误类别码a做了更改，即%与%d，\与\n，需多加注意。

六、代码生成

代码生成 - 作业一

程序中仅涉及常量声明、变量声明、读语句、写语句、赋值语句，加减乘除模除等运算语句、函数定义及调用语句。

编码前的设计

先前准备

- 创建一个全局以及多个局部数组表，一个局部数组表(AllocTable)属于一个函数，存所有声明过的数组名及数组大小。方便存进mips的data区。
- 创建一个字符串表(strings)，记录所有需要`printf`的字符串。方便存进mips的data区。

中间代码设计

- 标识符携带了一些信息，由java正则表达式表示：`"([0-9a-zA-z_\\$]+)@([0-9]+)((&|~)([0-9]+))?"`，$符号开头的是临时变量，@后面跟着的是level，&和~跟着的是下标(未×4)。&表示取数组地址，~表示取数组元素。

常量、变量声明中间代码：

```
const int a = 2, b[2][2] = {{1, 2}, {3, 4}};
转换成
decl int a@1 2
decl int b@1 1 2 3 4
即
decl <type> <ident> <values ...>
```

读语句中间代码：

```
printf("a=%d", a);
转换成
printf "a="
printf a@1
即
printf <value>
```

写语句中间代码：

```
a = getint();
转换成
a@1 = getint()
即
<ident> = getint()
```

赋值语句、加减乘除模除等运算语句中间代码：

```
第一种：
e = b;
转换成
e@1 = b@1
即
<ident> = <ident>

第二种：
d = -e;
转换成
$0@1 = - e@1
d@1 = $0@1
即
<ident> = <op> <ident>

第三种：
d = a * b + c * a;
转换成
$0@1 = a@1 * b@1
$1@1 = c@1 * a@1
$0@1 = $0@1 + $1@1
d@1 = $0@1
即
<ident> = <ident> <op> <ident>
```

函数定义语句中间代码：

```
int func(int a, int b[]) {
    return a * b[1];
}
```
转换成
```
proc int func@0 2
para int func@0 a@1
para int func@0 b@1
$0@1 = a@1 * b@1~1
ret $0@1
```
即
```
proc <type> <ident> <FParamCount>
para <type> <funcName> <FParam>
ret <value>
```

函数调用语句中间代码：

```
c = func(a, b);
```
转换成
```
call func@1 a@1 b@1
RET $0@1
c@1 = $0@1
```
即
```
call <funcName> <RParams ...>
RET <value>
```

目标代码设计

- 将`$v0,$v1,$a0,$a1,$a2,$a3`作为一个临时寄存器，暂时存储一些中间变量。

常量、变量声明目标代码：

```
.data
# b@1
arr_0: .space 16

.text
# decl int a@1 2
addi $v0, $0, 2     # $v0
sw $v0, -4($sp)

# decl int b@1 1 2 3 4
la $a1, arr_0       # $a1
addi $v0, $0, 1     # $v0
sw $v0, 0($a1)
addi $v0, $0, 2
sw $v0, 4($a1)
addi $v0, $0, 3
sw $v0, 8($a1)
addi $v0, $0, 4
sw $v0, 12($a1)
```

读语句目标代码：

```
# printf "a="
la $a0, str_0
li $v0, 4
syscall

# printf a@1
lw $a0, -4($sp)
li $v0, 1
syscall
```

写语句目标代码：

```
# a@1 = getint()
li $v0, 5
syscall
sw $v0, -4($sp)
```

赋值语句、加减乘除模除等运算语句目标代码：

```
# e@1 = b@1
lw $a1, -8($sp)
sw $a1, -20($sp)

# $0@1 = - e@1
lw $a1, -20($sp)
sub $a1, $0, $a1
move $t0, $a1

# d@1 = $0@1
sw $t0, -16($sp)

# $0@1 = a@1 * b@1
lw $a1, -4($sp)
lw $a2, -8($sp)
mul $a3, $a1, $a2
move $t0, $a3

# $1@1 = c@1 * a@1
lw $a1, -12($sp)
lw $a2, -4($sp)
mul $a3, $a1, $a2
move $t1, $a3

# $0@1 = $0@1 + $1@1
add $a2, $t0, $t1
move $t0, $a2

# d@1 = $0@1
sw $t0, -16($sp)
```

函数定义语句目标代码：

```
# proc int func@0 2
func_func_0:
```

```
# para int func@0 a@1
lw $t0, 0($sp)

# para int func@0 b@1
lw $t1, 4($sp)

# $0@1 = a@1 * b@1~1
lw $s0, 4($t1)
mul $a2, $t0, $s0
move $t2, $a2

# ret $0@1
move $v1, $t2
jr $ra
```

函数调用语句目标代码：

```
# call func@1 a@1 b@1
addi $sp, $sp, -8   # 变量
addi $sp, $sp, -4   # 寄存器
sw $ra, 0($sp)      # 寄存器压栈
addi $sp, $sp, -8   # 实参
lw $v0, 16($sp)
sw $v0, 0($sp)      # 实参压栈
la $a0, arr_0
sw $a0, 4($sp)      # 实参压栈
jal func_func_0     # 跳转到函数
addi $sp, $sp, 8    # 实参
lw $ra, 0($sp)      # 寄存器弹栈
addi $sp, $sp, 4    # 寄存器
addi $sp, $sp, 8    # 变量

# RET $0@1
move $t0, $v1

# c@1 = $0@1
sw $t0, -8($sp)
```

完成生成代码需要进行三遍，第一遍是一次性读取`testfile.txt`内容并交给`Lexer`类处理，第二遍是`Parser`类向`Lexer`类发起取单词请求，`Lexer`类识别单词并返回给`Parser`类，`Parser`类在进行语法分析的同时会记录信息并生成中间代码文件`MidCode.txt`，第三遍`GenMips`类读取`midCode.txt`并生成目标代码文件`mips.txt`。

编码完成之后的修改

在设计过程中， 由于很多因素未考虑仔细，修改的次数很多，几乎是边写边修改，因此花费的时间也很多。吸取到的经验是不要急于下手，可以先去理解源程序、中间代码和目标代码之间的对应关系，以及去了解源程序中的一些特性，比如数组什么时候取地址，什么时候取数值，还有数值在什么地方是变量，在什么地方是常量。

一开始我建符号表的时候只想到了数组变量常量都是先赋值了才会使用，所以在做运算语句时应当就能获取一个准确的值，于是直接在转换中间代码时就做了计算，并将所得的值赋值给左值，后来才发现忽略了函数的形参和返回值应该是一个变量，无法获取一个准确值。所以在处理运算语句时就进行了多次更改。

还有就是数值上的处理没有进行深入的考虑，比如`a[2][2] = b[1][1]`，左值是一个取地址操作，右值是一个取数组元素值操作。再比如

```
void func(int a[], int b) {
    ...
}
int main() {
    int a[2][2] = {{1,2},{3,4}}, b[2][2] = {{22,33},{44,55}};
    func(a[1], b[1][1]);
}
```

`func(a[1], b[1][1]);`第一个实参是取地址操作，第二个实参是取数组元素值操作。

最后，寄存器分配也出了点问题，比如寄存器不够用以及寄存器进行运算操作时覆盖了原先寄存器保存的值导致出现错误。`add $t0,$t0,$t1`，对于这种操作应该注意`$t0`是一个临时变量还是已存取了其它变量的寄存器。我目前的解决方法是声明的所有变量、常量都放进栈里，放进寄存器的仅有实参、临时变量以及返回值。

注意事项：

- Stmt的Lval是左值，PrimaryExp的Lval的是右值。
- 左值可以取变量常量所存的栈位置、取数组地址（&）、取实参、返回值、临时变量所存的寄存器，右值可以取变量常量、取数组元素（~），实参可以取变量常量、取数组地址（&）、取数组元素（~）。
- 临时寄存器的分配，在一个赋值语句里尽量做到互斥，避免寄存器里的数值被覆盖。

代码生成 - 作业二

在代码生成作业一的代码上处理条件语句、循环语句。

编码前的设计

条件表达式语句中间代码：

```
int a = 5;
if (a > 3) {
}
转换成
$0@0 = a@1 > 3
cmp $0@0 0 bne if_1_0
jump j if_1_1
if_1_0:
jump j if_1_end
if_1_1:
if_1_end:
即
cmp <ident> <ident> <op> <label>
```

条件表达式语句目标代码：

```
# $0@0 = a@1 > 3
lw $a1, -4($sp)
sgt $a2, $a1, 3
move $t0, $a2

# cmp $0@0 0 bne if_1_0
```

```
bne $t0, $0, if_1_0

# jump j if_1_1
j if_1_1

# if_1_0:
if_1_0:

# jump j if_1_end
j if_1_end

# if_1_1:
if_1_1:

# if_1_end:
if_1_end:
```

对于短路求值：

```
if (a && b || c && d) {
}
添加label以方便跳转，如下。
a && b
cond_if_1_0_0:
c && d
cond_if_1_0_1:
j if_1_1
if_1_0:
j if_1_end
if_1_1:
if_1_end:

解析：
若a为false，则跳转到cond_if_1_0_0，否则往下执行。
若b为true，则跳转到if_1_0执行if内部代码，否则往下执行。
若c为false，则跳转到cond_if_1_0_1，否则往下执行。
若d为true，则跳转到if_1_0执行if内部代码，否则往下执行。
```

编码完成之后的修改

由于我在写代码作业时并没有为下一次作业着想，因此在做代码生成作业二时改动很大。

改了常量、变量声明中间代码：

```
const int a = 2, b[2][2] = {{1, 2}, {3, 4}};
转换成
decl const a@1 2
decl const b@1&0 1
decl const b@1&1 2
decl const b@1&2 3
decl const b@1&3 4
即
decl <type> <ident> <value>
```

在`if`和`while`加入后很多值变得无法确定（编译过程是从上到下的），因此代码生成作业一查符号表获取确定值的功能被彻底打乱。好在我`de`了几天的`bug`之后，代码还能继续用，只是看着有点屎山。在这基础上也算完成了“常量传播”。

新增了两条中间代码，以解决代码生成作业一的bug：

```
<la reg:ident@lev arrName index:ident@lev>
<lw reg:ident@lev arrName index:ident@lev>
```

同时我又捋了捋数据存放的位置：
全局变量和所有数组（对于局部数组，会有一个表记录它和函数的对应关系）放在.data区；
全局或局部常量不存放，可在编译过程中通过查符号表获取值，即在中间代码时就已替换成数值；
局部变量放在栈里；
寄存器放临时变量，实参（数组地址，数组元素，变量），返回值。

七、代码优化

在代码生成阶段时，由于对中间代码设计以及常量变量数组应该存放的位置欠缺考虑，因此很多时间都花在了debug上，中间代码设计的不好也不利于代码优化，所以我最终没进行代码优化设计。