

# 任务三会议纪要

## 本文档说明

本小组项目的学习以周期性会议的形式开展，因此可能会出现某次会议的末尾和下一次会议的开始都是类似学习内容的情况。故文档以项目的学习内容为架构，不以会议时间等为标题。但会在任务全部完成后，在本文档的开头汇总每次会议的纪要以总结体现小组认真的态度。而之后细节的工作记录也有会议时间的体现，细致记录了每次的会议和学习内容。

由于在任务三开展阶段实训的开始，以及本身难度较大，导致进度放缓。最终，小组实现如下功能：

go语言语法文件的收集；语法分析树的生成；三地址码的设计与转换；汇编指令的设计与实现；

附加内容中，实现了错误分析、二进制文件的生成与部分代码优化。

会议纪要记录人：胡轶然

会议纪要审核人：李研

## 会议记录汇总

1. 任务三第一次会议：2022.3.26 14:00-17:00 项目制3的第7次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），开启任务三，精读书本第九章和第十五章并详细记录，研读任务三参考PPT。
2. 任务三第二次会议：2022.4.4 9:00-12:00 项目制3的第8次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），获得GO语言的g4文件。可以成功生成语法分析树；学习了三地址代码的分类和内涵（遇到的问题是发现最开始找的语法文件不符合要求，在定义函数时，{}前后必须加上空格/tap键才可以等。之后进行更换）。
3. 任务三第三次会议：2022.4.11 14:30-17:30 项目制3的第9次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）学习三地址码的概念，集体安装go语言环境，解决环境配置问题，学习基础go语言编程，汇总go语言的常用语句，以帮助之后任务的实现。
4. 任务三第四次会议：2022.4.18 9:00-12:30 项目制3的第10次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）进行go语言编程，实现语法分析树的转换。
5. 任务三第五次会议：2022.4.15 14:00-16:30 项目制3的第11次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）复习三地址码的表达形式，设计自己任务的三地址码的具体形式，并简单实现了赋值语句转换为三地址码的过程。
6. 任务三第六次会议：2022.5.2 14:00-16:30 项目制3的第12次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）参考网络学习资料以及现有编译器转换成三地址码的实现过程，继续实现三地址码转换的过程。
7. 任务三第七次会议：2022.5.9 18:00-20:30 项目制3的第13次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）最终实现转换为三地址码的过程。
8. 任务三第八次会议：2022.5.16, 14:30-16:30，项目制3的第14次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）复习第九章关于错误处理的内容，自行查找学习资料，实现部分go语言程序转换为三地址码过程中错误分析的处理；
9. 任务三第九次会议：2022.5.22, 14:00-16:30，项目制3的第15次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）继续完善错误分析的过程。
10. 任务三的第十次会议：2022.6.3, 9:30-12:00，项目制3的第16次会议（参会人：胡轶然、许振东、沈昊、胡书豪）暂时结束错误分析的任务，并分工了解汇编指令转换的过程原理；
11. 任务三的第十一次会议：2022.6.8, 14:00-16:30，项目制3的第17次会议（参会人：胡轶然、许振东、沈昊、胡书豪）选定使用MIPS指令集作为汇编指令的参考，并针对之前已经实现的三地址码转换功能，开展汇编指令转换的代码编写；

12. 任务三的第十二次会议：2022.6.12，14:00-16:30，项目制3的第18次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）针对三地址码转汇编进行小组分工，完成对已实现的三地址码转换为汇编的功能，并开展PPT的制作；
13. 任务三的第十三次会议：2022.6.18，9:00-12:00，项目制3的第19次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪）查找资料，学习相关Yacc有关知识，进一步完善错误处理的部分。
14. 任务三的第十四次会议：2022.6.24，19:00-21:00，项目制3的第20次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），调整修改PPT，书写展示文稿，并进行修改调整。
15. 任务三的第十五次会议：2022.6.24，15:00-17:00，项目制3的第21次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），分工制作IDE展示界面和二进制文件的生成。
16. 任务三的第十六次会议：2022.6.27，15:00-17:00，项目制3的第21次会议（参会人：胡轶然、许振东、沈昊、李研、胡书豪），继续制作IDE展示界面和二进制文件的生成，并完善PPT与讲稿。

部分会议照片：



zima-blue@11:~/learning/Project3\_ubuntu/tmp/go/final/pythonProject/test\$ pip3 install PyQt5  
... Collecting PyQt5  
Using cached PyQt5-5.15.7-cp37abi-manylinux1\_x86\_64.whl (8.4 MB)  
Requirement already satisfied: PyQt5>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5) (52.11.M)  
Requirement already satisfied: PyQt5-QT5>=5.15.0 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-QT5) (52.11.M)  
Requirement already satisfied: PyQt5-sip>=13.2>12.11 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-sip) (52.11.M)  
Requirement already satisfied: PyQt5-tools>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-tools) (52.11.M)  
Requirement already satisfied: PyQt5-qtcore>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtcore) (52.11.M)  
Requirement already satisfied: PyQt5-qtgui>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtgui) (52.11.M)  
Requirement already satisfied: PyQt5-qtlocation>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtlocation) (52.11.M)  
Requirement already satisfied: PyQt5-qtprint>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtprint) (52.11.M)  
Requirement already satisfied: PyQt5-qtquick>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtquick) (52.11.M)  
Requirement already satisfied: PyQt5-qtserialport>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtserialport) (52.11.M)  
Requirement already satisfied: PyQt5-qtwebchannel>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtwebchannel) (52.11.M)  
Requirement already satisfied: PyQt5-qtwebkit>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtwebkit) (52.11.M)  
Requirement already satisfied: PyQt5-qtxmlpatterns>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtxmlpatterns) (52.11.M)  
Requirement already satisfied: PyQt5-sip>=13.2>12.11 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-sip) (52.11.M)  
Requirement already satisfied: PyQt5-tools>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-tools) (52.11.M)  
Requirement already satisfied: PyQt5-qtcore>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtcore) (52.11.M)  
Requirement already satisfied: PyQt5-qtgui>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtgui) (52.11.M)  
Requirement already satisfied: PyQt5-qtlocation>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtlocation) (52.11.M)  
Requirement already satisfied: PyQt5-qtprint>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtprint) (52.11.M)  
Requirement already satisfied: PyQt5-qtquick>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtquick) (52.11.M)  
Requirement already satisfied: PyQt5-qtserialport>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtserialport) (52.11.M)  
Requirement already satisfied: PyQt5-qtwebchannel>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtwebchannel) (52.11.M)  
Requirement already satisfied: PyQt5-qtwebkit>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtwebkit) (52.11.M)  
Requirement already satisfied: PyQt5-qtxmlpatterns>=5.15.7 in /home/zima-blue/anaconda3/lib/python3.9/site-packages (From packages: PyQt5-qtxmlpatterns) (52.11.M)



zima-blue@11:~/learning/Project3\_ubuntu/tmp/go/final/pythonProject/test\$  
base) zima-blue@11:~/learning/Project3\_ubuntu/tmp/go/final/pythonProject/test\$  
base) zima-blue@11:~/learning/Project3\_ubuntu/tmp/go/final/pythonProject/test\$

详细分工

胡轶然：整理会议纪要，协助完成三地址码的生成和过程中的错误分析处理；协助完成汇编指令的生成与代码优化；整理制作PPT展示以及讲稿文档；审核设计文档；安装运行IDE演示界面；24%

许振东：制作IDE界面进行展示，实现语法分析树的生成，帮助代码优化的实现；提供工作总结和协助PPT的制作；19%

沈昊：合作实现汇编指令的生成，帮助实现程序二进制文件的生成和代码优化，制作项目整体的脚本文件，提供工作总结和协助PPT的制作；19%

李研：协助完成三地址码的生成过程中的错误处理，提供工作总结和代码的解释文档；帮助制作IDE界面进行展示，审核会议纪要；19%

胡书豪：帮助设计三地址码的表达形式和三地址码的生成，提供工作总结和协助PPT的制作；撰写设计文档；19%

## -----知识学习-----

### Chapter 9: 错误报告与恢复

本章主要介绍ANTLR自动生成的语法分析器使用的自动错误报告和恢复策略。与此同时，还介绍了如何修改默认的错误处理机制使之符合典型情况下的需求，以及如何在特定程序中定制错误信息。

#### 9.1 错误处理入门

本节介绍了ANTLR的基本错误信息，包括：通用错误信息（针对一般性错误），无可行备选分支错误（针对在语法分析器决策的关键位置剩余的输入文本不符合规则的任何一个备选分支的情况），词法错误（即无法将一个或多个字符匹配为词法符号的情况），单词法符号移除（针对单个不符合语法的多余词法符号），单词法符号补全（针对单个缺失的词法符号）等。

#### 9.2 修改转发ANTLR的错误信息

本节介绍了通过自定义实现接口ANTLRErrorListener来改变错误消息的目标输出和内容的方法。

首先，在添加自定义的错误监听器之前，我们必须移除原有的输出目标是控制台的内置错误监听器以免重复。

同时，通过Java Swing技术，我们还可以修改syntaxError()方法，将错误消息使用对话框来进行显示。

此外，默认错误监听器ConsoleErrorListener不会对有歧义的输入序列向控制台打印错误信息。尽管如此，语法分析器检测到有歧义的输入序列的时候仍然会通知错误监听器。因此，如果使用者（即我们）希望得到自己有关语法歧义的通知，需要添加一个DiagnosticErrorListener的实例来告知语法分析器，并且通过如下的方法告知语法分析器对所有的歧义警告都感兴趣：

```
parser.getInterpreter()  
    .setPredictionMode(PredictionMode.LL_EXACT_AMBIG_DETECTION);
```

同时，作者同时推荐在开发过程中使用上面提到的DiagnosticErrorListener来检测歧义。

在ANTLR4中，其语法分析器会尝试在子规则的识别前和识别过程中进行重新同步，避免草率地丢弃词法符号并退出当前规则，从而提供了更优秀的内置错误恢复机制，并允许开发者更加容易地修改错误处理策略。

#### 9.3 自动错误恢复机制

本节详细讲述了ANTLR提供的错误恢复机制，包括：

1. 通过扫描后续词法符号来恢复。也就是在面对真正的非法输入时，语法分析器选择向后查找和（有可能的）不断消耗词法符号，直到它认为自己重新同步（找到重新同步集合中的词法符号），再返回原先被调用的规则。这是最基本，最激进的策略，也是ANTLR在试图在规则内部恢复无效后的后备方案。

p.s. 重新同步集合是调用栈中所有规则的后续符号集合的并集，而后续符号集合是能够鼓励和延续该条规则，从而无须离开当前规则的词法符号集合

2. 从不匹配的词法符号中恢复。这种方式下，语法分析器重新同步有三种选择：移除一个词法符号、补全一个词法符号或者跑出一个异常来启动同步-返回机制（也即方法1）语法分析器会首先使用移除当前词法符号的方式——这是重新同步最容易的方式，而在移除词法符号无法同步的情况下，它会转而尝试补全一个词法符号。这一过程中，错误处理器会创建一个语法分析器所期望类型的词法符号，以及与当前词法符号相同的行列位置信息。
3. 从子规则的错误中恢复。在任意子规则的起始位置，语法分析器会且仅会尝试进行单词法符号移除，如果子规则是一个循环结构，在遇到错误时，语法分析器会尝试进行积极的恢复，使得自己能留在循环的内部。成功匹配到循环的某个备选分支后，词法分析器会持续消费词法符号，直到发现满足下列条件之一的词法符号为止：
  - (a) 循环的另一次迭代
  - (b) 紧跟在循环之后的内容
  - (c) 当前规则的重新同步集合中的元素
4. 错误恢复机制的防护措施。ANTLR语法分析器具有内置防护措施以保证错误恢复过程正常结束。如果在相同的语法分析位置遇到了相同的输入情况，语法分析器会在尝试进行恢复之前强制消费一个词法符号。

#### 9.4 勘误备选分支

对一些常见的语法错误可以对其进行特殊处理。只需增加一些备选分支，匹配此类常见错误即可。样例如下：

```
errors/Call.g4
stat: fcall ';' ;
fcall
: ID '(' expr ')'
| ID '(' expr ')' ')' {notifyErrorListeners("Too many parentheses");}
| ID '(' expr           {notifyErrorListeners("Missing closing ')'"});
|
expr: '(' expr ')'
| INT
|
```

#### 9.5 修改ANTLR的错误处理策略

我们有时会遇到一些非典型的，需要修改默认机制的场景，例如希望关闭某些会带来额外运行负担的默认的错误处理功能，或是希望语法分析器遇到第一个语法错误时就退出。

ANTLRErrorStrategy接口已经有了一个实现类DefaultError-Strategy，该类完成了全部的默认错误处理工作。

这一接口中主要被介绍的方法有：

- (a) reportError()，用于进行错误报告，根据抛出异常类型，把报告错误的职责委托给另三个方法之一（这三个方法的具体细节似乎暂时不需要关心）
- (b) recover()，用于完成同步-返回功能，如果需要在第一个错误时直接报错并退出，可以覆盖这一方法并直接抛出一个RuntimeException。
- (c) sync()，遇到错误时，语法分析器会调用这个方法尝试重新同步，如果我们需要关闭错误处理功能，可以通过继承DefaultError-Strategy类并使用空方法覆盖sync()的方式进行。
- (d) recoverInline()，用于执行行内恢复，如果需要在第一个错误处报错并退出，同样需要使用一个抛出异常的方法覆盖之。

以上的方法只针对语法错误的处理策略，如果需要修改语法分析器对词法错误的处理策略，需要覆盖Lexer类里的方法。

## Chapter 15: Grammar Reference

第十五章的内容包括对ANTLR语法及其关键语义的参考和总结，以此为基础开始对GO语言的语法设计是十分必要的。

### 15.1 语法词汇表

ANTLR的词汇表遵循了C语言及其继承者的句法规则，并引入一些扩展用于对语法进行描述。

关于注释：ANTLR支持单行、多行以及Javadoc风格的注释（此种注释不会被忽略并将被送入语法分析器）。

关于标志符：语法符号名和词法规则名总是以大写字母开头；文法规则总是以小写字母开头；首字母之后的字符可以是大小写字符、数字和下划线；允许出现Unicode字符；若语法文件编码不是UTF-8，需在ANTLR工具中使用-encoding选项。

关于文本常量：ANTLR与大多数其他语言一样不区分字符常量和字符串常量，所有的文本常量都是由单括号括起来的字符串，且不支持正则表达式。文本常量可以包含\uXXXX（十六进制的Unicode字符值）形式的Unicode转义序列。ANTLR生成的识别器假定语法中的字符都是Unicode字符，而运行库会根据目标语言对输入文件的编码作出假设。

关于动作：动作的格式为由花括号包围的任意文本；右花括号位于字符串/注释中或花括号平衡时无须转义，其他情况下需使用反斜杠转义。内嵌代码可以出现在以@header和@member命名的动作、词法和文法规则、指定异常捕获区、文法规则的属性区域（返回值、参数以及局部变量），以及一些规则元素的选项（判定）中。

关于关键字：不要使用目标语言中的关键字作为词法符号、标签或规则名。保留字列表如下：import、fragment、lexer、parser、grammar、returns、locals、throws、catch、finally、mode、options、tokens（另有rule不是关键字，但应避免将它作为规则或备选分支名）。

### 15.2 语法结构

一份语法由一个语法声明和紧随其后的若干条规则构成，通用形式如下：

```
/** Optional Javadoc-style comment */
① grammar Name;
options {...}
import ... ;
tokens {...}
@actionName {...}

«rule1» // parser and lexer rules, possibly intermingled
...
«ruleN»
```

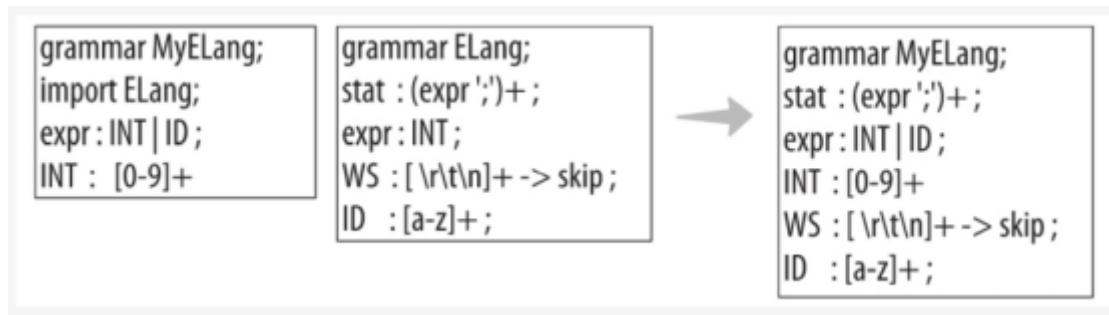
包含语法X的文件必须被命名为X.g4。options、import、token声明（此三项可有可无且最多出现一次）以及动作可以以任意次序出现。

文件头以及至少一条规则必须存在。

可在grammar前加parser或lexer创建近允许文法规则出现的语法或纯词法语法。只有词法规则语法才能包含模式说明。

关于语法导入：语法导入允许将词法分解为可复用的逻辑单元。一个语法会从其导入的语法中继承所有规则、词法符号声明和具名的动作；位于“主语法”中的规则将会覆盖其导入的语法中的规则以实现继承机制。

在处理一份主语法的过程中，ANTLR工具将所有被导入的语法加载到一起，然后将其中的规则、词法符号类型以及具名动作合并到主语法中。



如图，展示了导入Elang语法后的MyELang语法。

并非每种类型的语法都能导入其他类型的语法：词法语法能导入此法语法，句法语法能导入句法语法，混合语法能导入词法语法或者句法语法。

ANTLR将被导入的规则放置在主语法的词法规则列表末尾，这使得主语法中的词法规则具有比被导入语法中规则更高的优先级。

关于词法符号声明：tokens区域存在的意义在于，它定义了语法所需而未在本语法中列出对应规则的词法符号。实际上，tokens区域仅仅是一些会被合并到整体词法符号集合中的词法符号定义。

关于语法级别的动作：两种动作，header和members，前者用于将代码注入生成的识别类的类声明之前，后者用于将代码注入为识别类的字段和方法。

对于混合语法，ANTLR同时将这些代码注入到词法分析器和语法分析器中；需使用@parser:: name或者@lexer:: name使代码只出现在语法分析器或词法分析器中。

其余内容不做详细记录，详情查看参考书。

## 课程PPT学习

三地址码学习：

[三地址码学习](#);

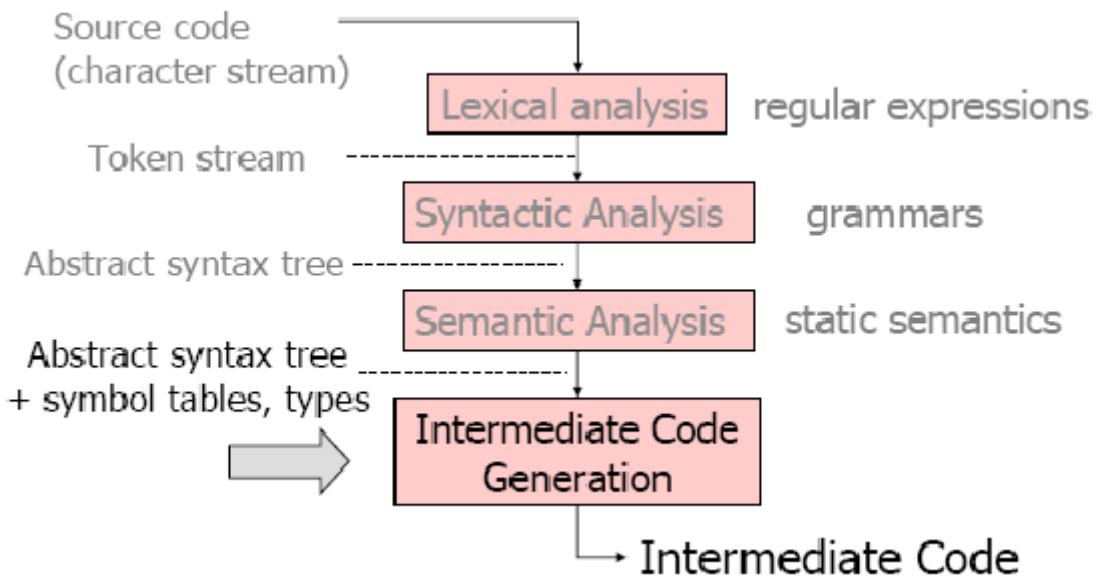
## 语义分析

紧跟在语法分析之后，编译程序要做的工作就是进行静态语义检查和翻译。编译器必须要检查源程序是否符合源语言规定的语法和语义要求。这种检查称为静态检查，检查并报告程序中某些类型的错误。



语义分析的检查包括：检查在词法分析和语法分析中发现不了的错误、范围错误、类型错误、类型检查、也有控制流错误。

## 中间代码生成



- 常用源语言的中间表示形式

- 抽象语法树

- 后缀式

- 三地址代码（包括三元式、四元式、间接三元式）

- DAG图表示

### 三地址指令代码

- 三地址代码的形式

三地址码的得名原因是每条语句通常包含三个地址，两个是操作数地址，一个是结果地址。三地址代码一般是下列形式的语句序列：

$x := y \text{ op } z$ , 其中,  $x$ 、 $y$ 和 $z$ 是名字, 常量或编译器生成的临时变量,  $\text{op}$ 代表任何操作符（定点运算符、浮点运算符、逻辑运算符等）。这里不允许组合的算术表达式，因为语句右边只有一个操作符。

也就是说,  $\$x+y * z\$$ 这样的表达式是不被允许的，在三地址代码中要拆分成这样的形式

- $T1 := y * z$

- $T2 := x + T1$

其中 $T1$ ， $T2$ 为编译时产生的临时变量。

这种复杂算术表达式和嵌套控制流语句的拆解使得三地址码适用于目标代码生成及优化。三地址码的得名原因是每条语句通常包含三个地址，两个是操作数地址，一个是结果地址。

- 三地址语句的类型

形如 $x := y \text{ op } z$ 的赋值语句，其中 $\text{op}$ 为二元算术算符或逻辑算符

形如 $x := \text{op } y$ 的赋值语句，其中 $\text{op}$ 为一元算符。

形如 $x := y$ 的复制语句，将 $y$ 的值赋给 $x$

形如 $\text{goto } L$ 的无条件跳转语句，即下一条将被执行的语句是带有标号 $L$ 的三地址语句

- 生成三地址码的S-属性文法：

产生式	语义规则
$S \rightarrow id := E$	$S.code := E.code \parallel gen(id.place := E.place)$
$E \rightarrow E_1 + E_2$	$E.place := newtemp;$
	$E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place + E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := newtemp;$
	$E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place * E_2.place)$
$E \rightarrow - E_1$	$E.place := newtemp;$
	$E.code := E_1.code \parallel gen(E.place := 'uminus' E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place$ $E.code := ''$

### · 三地址语句的实现

三地址语句是中间代码的一种抽象形式，这些语句可以以带有操作符和操作数域的记录来实现。四元式、三元式及简介三元式是三种这样的表示。

### · 四元式

一个四元式是带有四个域的记录结构，这四个域分别称为op, arg1, arg2及result。域op包含一个代表运算符的内部码，三地址语句 $x:=y op z$ 通过将y放入arg1, z放入arg2, 并且将x放入result, :=为算符。

像 $x:=y$ 或 $x:=-y$ 这样的一元操作符语句不使用arg2, 像param这样的运算符仅使用arg1域。条件和无条件语句将目标标号存入result域，临时变量也要填入符号表中。

$a := b * -c + b * -c$				
	op	arg1	arg2	result
(0)	uminus	c		T <sub>1</sub>
(1)	*	b	T <sub>1</sub>	T <sub>2</sub>
(2)	uminus	c		T <sub>3</sub>
(3)	*	b	T <sub>3</sub>	T <sub>4</sub>
(4)	+	T <sub>2</sub>	T <sub>4</sub>	T <sub>5</sub>
(5)	assign	T <sub>5</sub>		a

	op	arg1	arg2
(0)	uminus	c	
(1)	*	b	0
(2)	uminus	c	
(3)	*	b	2
(4)	+	1	3
(5)	assign	a	4

### · 三元式

为了避免把临时变量填入符号表，我们可以通过计算这个临时变量的语句的位置来引用这个临时变量。这样三地址代码的记录只需要三个域op, arg1和arg。对于一目运算符op, arg1和arg2只需用其一。我们可以随意选用一个。

$x[i] := y$				$x := y[i]$			
	op	arg1	arg2		op	arg1	arg2
(0)	=[]	x	i	(0)	=[]	y	i
(1)	assign	(0)	y	(1)	assign	x	(0)

### · 间接三元式

为了便于代码优化处理，有时不直接使用三元式表，而是另设一张指示器（称为间接码表），它将运算的先后顺序列出有关三元式在三元表中的位置。换句话说，我们用一张间接码表辅以三元式表的办法来表示中间代码。这种表示方法称为间接三元式。

◆  $X := (A + B) * C$      $Y := D \wedge (A + B)$

间接代码

(1) (2) (3) (1) (4) (5)

	op	arg1	arg2
(1)	+	A	B
(2)	*	(1)	C
(3)	:=	X	(2)
(4)	^	D	(1)
(5)	:=	Y	(4)

当在代码优化过程中需要调整运算顺序时，只需重新安排间接码表，无需改动三元式表

对于间接三元式表示，语义规则中应增添产生间接码表的动作，并且在向三元式表填进一个三元式之前，必须先查看一下此式是否已在其中，就无须填入。

### · 表示方法的比较

三元式与四元式的差异可以看作在表示中引入了多少间址。

使用四元式表示，定义或使用临时变量的三地址语句可通过符号表直接访问该临时变量的地址

使用四元式的一个更重要的好处体现在优化编译器中。在三元式中，如果要移动一条临时值的语句需要改变arg1和arg2数组中对该语句的引用。

间接三元式没有上述问题，间接三元式看上去和四元式非常相似，他们都需要大约相同的存储空间，并且对代码重新排序的效率相同。

对于普通三元式，必须将对那些临时变量的存储分配推迟到代码生成阶段。

### 三地址代码翻译生成资料

#### · 常用的三地址码

序号	指令类型	指令形式	备注
1	赋值指令	$x = y \text{ op } z$	$\text{op}$ 为运算符
2	复制指令	$x = y$	
3	条件跳转	$\text{if } x \text{ relop } y \text{ goto } n$	$\text{relop}$ 为关系运算符
4	非条件跳转	$\text{goto } n$	跳转到地址n的指令
5	参数传递	$\text{param } x$	将x设置为参数
6	过程调用	$\text{call } p, n$	p为过程的名字n为过程的参数的个数
7	过程返回	$\text{return } x$	
8	数组引用	$x=y[i]$	i为数组的偏移地址，而不是下标
9	数组赋值	$x[i]=y$	
10	地址及指针操作	$x=&y$ $x=*y$ $*x=y$	

· 将上表的三地址指令用四元式表示

$x = y \text{ op } z$	$(\text{op}, y, z, x)$
$x = \text{op } y$	$(\text{op}, y, \_, x)$
$x = y$	$(=, y, \_, x)$
$\text{if } x \text{ relop } y \text{ goto } n$	$(\text{relop}, x, y, n)$
$\text{goto } n$	$(\text{goto}, \_, \_, n)$
$\text{param } x$	$(\text{param}, \_, \_, x)$
$\text{call } p, n$	$(\text{call}, p, n, \_)$
$\text{return } x$	$(\text{return}, \_, \_, x)$
$x=y[i]$	$(=[], y, i, x)$ ps: y为基址，i为偏移地址
$x[i]=y$	$([], =, y, x, i)$
$x=\&y$	$(\&, y, \_, x)$
$x=^y$	$(=^, y, \_, x)$
$*x=y$	$(*=, y, \_, x)$

· 中间代码生成的例子

```

while a<b do
    if c<5 then
        while x>y do
            z=x+1;
    else x=y;

```

100到112为指令的编码，从100到112顺序执行。

代码行	解读
100: ( j<, a, b, 102 )	如果a<b ,那么跳转到102指令，否则继续执行101指令
101: ( j, -, -, 112 )	该指令为无条件指令，跳转到112
102: ( j<, c, 5, 104 )	如果c<5 ,那么跳转到104指令，否则继续执行103指令
103: ( j, -, -, 110 )	该指令为无条件指令，跳转到110
104: ( j>, x, y, 106 )	如果x>y ,那么跳转到106指令，否则继续执行105指令
105: ( j, -, -, 100 )	该指令为无条件指令，跳转到100
106: ( +, x, 1, t1 )	x+1的值赋值给t1
107: ( =, t1, -, z )	t1的值赋值给z, 106和107完成了一条语句
108: ( j, -, -, 104 )	该指令为无条件指令，跳转到104
109: ( j, -, -, 100 )	该指令为无条件指令，跳转到100
110: ( =, y, -, x )	把y赋值给x, 然后执行111指令
111: ( j, -, -, 100 )	该指令为无条件指令，跳转到100
112: return	结束

## Yacc错误处理学习

Yacc中，语法说明文件（即Yacc源文件，一般表示为xxx.y的格式）分为三个部分：

第一部分：

定义段，可分为两部分，第一部分用 '%{' 和 '%}' 包围起来，其内部是C语法写的一些定义和声明。

第二部分是对文法的终结符和非终结符做一些相关声明，声明中包含其部分特性，包括是否为终结符，有无结合性，若有，是左结合还是右结合，等等。

第二部分：

规则段，其定义了文法的非终结符以及产生式集合，以及当归约整个产生式时应该执行的操作。例如：

```

1   expr: expr PLUS term      {语义动作}
2       | term                {语义动作}
3       ;

```

产生式后大括号括起的语义动作表示该条产生式归约时应当执行的操作。（按照这条规则归约时，程序实际要做些什么）

第三部分：

辅助函数部分，使用C语言语法来写，一般是规则段中或者语法分析器的其他部分用到的函数。

“一般来说，除规则段用到的函数外，辅助函数段一般包括如下一些例程：yylex(), yyerror(), main()。”（摘自Yacc手册，也是使用代码中不规范的一点：yylex()和yyerror()似乎被写在了定义段）

（关于yylex()的功能在此暂且不表）

yyerror()是Yacc提供的错误信息报告程序，其库中源程序如下：

```
#include <stdio.h>
yyerror (s) char * s{
    fprintf (stderr, "%s\n", s);
}
```

如果用户需要更复杂的自定义错误信息报告程序，可以自己覆写这一函数。

而Yacc处理错误的方法是：在其发现语法错误时，抛弃那些导致错误的符号来调整状态栈，然后从出错处的后一个符号处或是跳过若干符号直到遇到指定的某符号时继续分析。（即在遇到错误时直接开始抛弃符号，直到出错处全部清除或者遇到用户指定的某个符号才继续）

Yacc拥有一个保留终结符error，不需声明也可使用。将其写在某个产生式（或者说某条规则）右部，Yacc就认为这个地方是可能出错的，如果确实出错，就进行上述错误处理。

例如：

```
stat: error;
```

这使得Yacc在分析stat推导出的句型时可以在遇到语法错误时直接跳过出错部分继续分析（但仍然会打印语法错误信息）

又例如

```
stat: error';' ;
```

这使得Yacc在分析stat推导出的句型时，在遇到语法错误时开始不断抛弃符号流中的符号，直到找到下一个分号为止。

有时我们会需要用到'yyerrok'这一语句，用于在一个操作中将解析器重置为常规模式，从而避免编译器在重新同步的过程中错误地在删除不合法的token时不报错，这使得一些错误无法在一次编译尝试中被看到。

## -----任务实现-----

### 生成语法分析树

简而言之，首先，计算机先用一个词法分析器(Lexer)对文本(Source Code)进行词法分析，生成Token。接下来是将它传给一个解析器，然后检索生成语法分析树。词法分析器用来将字符序列转换为单词(Token)。词法分析主要是完成：

1. 对源程序的代码进行从左到右的逐行扫描，识别出各个单词，从而确定单词的类型；
2. 将识别出的单词转换为统一的机内表示——词法单元 (Token) 形式。

token是一种类型Map的key/value形式，它由<种别码，属性值>组成，种别码就是类型、属性值就是值。例如下述代码。

```
go package main
const s = "foo"

PACKAGE(package)
IDENT(main)
CONST(const)
IDENT(s)
ASSIGN(=)
STRING("foo")
EOF()
```

我们设计了Go语言的解析器使用了 **LALR(1)** 的分析法来解析词法分析过程中输出的Token序列。语法分析器最终会使用不同的结构体来构建语法分析树中的节点，根节点和其它节点如下：

```
//根节点
type File struct {
    PkgName *Name
    DeclList []Decl
    Lines   uint
    node
}

//非根节点
type (
    Decl interface {
        Node
        aDecl()
    }
    FuncDecl struct {
        Attr map[string]bool
        Recv *Field
        Name *Name
        Type *FuncType
        Body *BlockStmt
        Pragma Pragma
        decl
    }
)
```

操作指令：

```
antlr4 Golang.g4
javac Golang*.java
grun Golang sourceFile -gui example.go
```

## go语言安装

安装步骤在此不详述，可以使用 `go env`, `go version`, `go run xxx.go` 来验证自己的g语言是否安装成功。

# 生成三地址码

在原始工程中，文件架构为：

```
src
|__lexer.l
|__node.h
|__parserIR.y
|__symtab.h
test
test_new
Makefile
README.md
```

在 `parserIR.y` 中，定义终结符：

在其中，定义终结符：

```
%token <sval> PACKAGE IMPORT FUNC BREAK CASE CONST CONTINUE DEFAULT
%token <sval> ELSE FOR GO IF RANGE RETURN STRUCT SWITCH TYPE VAR VAR_TYPE
%token <sval> BOOL_CONST NIL_VAL IDENTIFIER BYTE STRING ELLIPSIS
%token <sval> SHL SHR INC DEC
%token <sval> INTEGER
%token <sval> FLOAT
%left <sval> ADD SUB MUL QUO REM
%token <sval> ASSIGN AND NOT DEFINE AND_NOT
%left <sval> OR XOR ARROW //Identifier
%right <sval> ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN QUO_ASSIGN REM_ASSIGN
%right <sval> AND_ASSIGN OR_ASSIGN XOR_ASSIGN
%right <sval> SHL_ASSIGN SHR_ASSIGN AND_NOT_ASSIGN COLON
%left <sval> LAND LOR EQL NEQ LEQ GEQ SEMICOLON
%left <sval> GTR LSS LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA PERIOD
```

非终结符：

```
%type <nt> StartFile Expression Expression1 Expression2 Expression3 Expression4
Expression5
%type <nt> Block StatementList Statement SimpleStmt int_lit
%type <nt> EmptyStmt /*ExpressionStmt*/ IncDecStmt OPENB CLOSEB
%type <nt> Assignment Declaration ConstDecl varSpec string_lit
%type <nt> Signature Result Parameters ParameterList ParameterDecl
%type <nt> ConstSpec MethodDecl Receiver TopLevelDecl TopLevelDeclList
LabeledStmt
%type <nt> ReturnStmt BreakStmt ContinueStmt StructType
%type <nt> FunctionDecl FunctionName TypeList
%type <nt> Function FunctionBody FunctionCall ForStmt ForClause /*RangeClause*/
InitStmt ArgumentList Arguments
%type <nt> PostStmt Condition UnaryExpr PrimaryExpr
%type <nt> Selector Index /*slice */TypeDecl TypeSpecList TypeSpec VarDecl
%type <nt> TypeAssertion ExpressionList ArrayType assign_op rel_op add_op mul_op
unary_op Tag byte_lit float_lit Label
//%type <nt> ExprCaseClauseList ExprCaseClause
%type <nt> Operand Literal BasicLit OperandName ImportSpec IfStmt
%type <nt> ImportPath
%type <nt> PackageClause PackageName PackageName2 ImportDecl ImportDeclList
ImportSpecList
```

```
%type <nt> FieldDeclList FieldDecl Type TypeLit ArrayLength
%type <nt> TypeName
%type <nt> /*QualifiedIdent*/ PointerType IdentifierList IdentifierLIST BaseType
```

剩下大多数代码的核心思想代码的核心思想遵循LALR (1) 分析法，在遇到传过来的Token是非终结符时，编写代码将其进行拆解为终结符。

在lexer.l中，三地址码的生成一共实现以下字符：

```
"="          {yyval.sval = strdup(yytext); return ASSIGN;}
"+"
"-"
"*"
"/"
"%"

"&"
"|"
"^"
"<<"
">>>"
"&^"

"+="
"-="
"*="
"/="
"%="

"&="
"|="
"^="
"<=>"
">>=>"
"&^="

"&&"
"||"
"->"
"+"
"--"

"=="
">>
"<"
"!/"

"!="
"=<"
">=>"
":="
"..."
```

```
{yyval.sval = strdup(yytext); return ADD;}
{yyval.sval = strdup(yytext); return SUB;}
{yyval.sval = strdup(yytext); return MUL;}
{yyval.sval = strdup(yytext); return QUO;}
{yyval.sval = strdup(yytext); return REM;}

{yyval.sval = strdup(yytext); return AND;}
{yyval.sval = strdup(yytext); return OR;}
{yyval.sval = strdup(yytext); return XOR;}
{yyval.sval = strdup(yytext); return SHL;}
{yyval.sval = strdup(yytext); return SHR;}
{yyval.sval = strdup(yytext); return AND_NOT;}

{yyval.sval = strdup(yytext); return ADD_ASSIGN;}
{yyval.sval = strdup(yytext); return SUB_ASSIGN;}
{yyval.sval = strdup(yytext); return MUL_ASSIGN;}
{yyval.sval = strdup(yytext); return QUO_ASSIGN;}
{yyval.sval = strdup(yytext); return REM_ASSIGN; }

{yyval.sval = strdup(yytext); return AND_ASSIGN;}
{yyval.sval = strdup(yytext); return OR_ASSIGN;}
{yyval.sval = strdup(yytext); return XOR_ASSIGN;}
{yyval.sval = strdup(yytext); return SHL_ASSIGN;}
{yyval.sval = strdup(yytext); return SHR_ASSIGN;}
{yyval.sval = strdup(yytext); return AND_NOT_ASSIGN; }

{yyval.sval = strdup(yytext); return LAND;}
{yyval.sval = strdup(yytext); return LOR;}
{yyval.sval = strdup(yytext); return ARROW;}
{yyval.sval = strdup(yytext); return INC;}
{yyval.sval = strdup(yytext); return DEC; }

{yyval.sval = strdup(yytext); return EQL;}
{yyval.sval = strdup(yytext); return GTR;}
{yyval.sval = strdup(yytext); return LSS;}
{yyval.sval = strdup(yytext); return NOT; }

{yyval.sval = strdup(yytext); return NEQ;}
{yyval.sval = strdup(yytext); return LEQ;}
{yyval.sval = strdup(yytext); return GEQ;}
{yyval.sval = strdup(yytext); return DEFINE;}
{yyval.sval = strdup(yytext); return ELLIPSIS; }

{yyval.sval = strdup(yytext); return LPAREN;}
{yyval.sval = strdup(yytext); return RPAREN;}
{yyval.sval = strdup(yytext); return LBRACE;}
{yyval.sval = strdup(yytext); return RBRACE;}
```

```

"["          {yyval.sval = strdup(yytext); return LBRACK;}
"]"          {yyval.sval = strdup(yytext); return RBRACK;}
","          {yyval.sval = strdup(yytext); return COMMA;}
";"          {yyval.sval = strdup(yytext); return SEMICOLON;}
":"          {yyval.sval = strdup(yytext); return COLON;}
"."          {yyval.sval = strdup(yytext); return PERIOD;}

"package"      {yyval.sval = strdup(yytext); return PACKAGE;}
"import"       {yyval.sval = strdup(yytext); return IMPORT;}
"func"         {yyval.sval = strdup(yytext); return FUNC;}
"break"        {yyval.sval = strdup(yytext); return BREAK;}
"case"         {yyval.sval = strdup(yytext); return CASE;}
"const"        {yyval.sval = strdup(yytext); return CONST;}
"continue"     {yyval.sval = strdup(yytext); return CONTINUE;}
"default"      {yyval.sval = strdup(yytext); return DEFAULT;}
"else"         {yyval.sval = strdup(yytext); return ELSE;}
"for"          {yyval.sval = strdup(yytext); return FOR;}
"go"           {yyval.sval = strdup(yytext); return GO;}
"if"           {yyval.sval = strdup(yytext); return IF;}
"range"        {yyval.sval = strdup(yytext); return RANGE;}
"return"        {yyval.sval = strdup(yytext); return RETURN;}
"struct"       {yyval.sval = strdup(yytext); return STRUCT;}
"switch"       {yyval.sval = strdup(yytext); return SWITCH;}
"type"         {yyval.sval = strdup(yytext); return TYPE;}
"var"          {yyval.sval = strdup(yytext); return VAR;}
{VAR_TYPE}      {yyval.sval = strdup(yytext); return VAR_TYPE;}
{BOOL_CONST}    {yyval.sval = strdup(yytext); return BOOL_CONST;}
{NIL_VAL}       {yyval.sval = strdup(yytext); return NIL_VAL; }

{LETTER}({LETTER}|{DIGIT})*      {yyval.sval = strdup(yytext); return IDENTIFIER;}
{DIGIT}+                  {yyval.sval = strdup(yytext); return INTEGER;}
{DIGIT}+.{DIGIT}+          {yyval.sval = strdup(yytext); return FLOAT;}
\'{UNICODE_CHAR}'          {yyval.sval = strdup(yytext); return BYTE;}
\"([\\\\\\n\"]|\\.\\.)*\"  {yyval.sval = strdup(yytext); return STRING;}

```

中间代码的设计方式，依照四元式的方式进行定义，基本形式为：(op, result, arg1, arg2)，及<操作符>，<结果>，<操作数1>，<操作数2>。将操作数1和操作数2参与的op操作的结果，存储到results的位置。其他形式：

(return, value)相当于空出了result和arg2的位置。一般适用于单目运算符。除此之外，对于有些运算符不生成结果的，例如print，则将result位置空出。

中间代码的部分具体形式如下：

```

//1
(op, c, a, b)
//op为运算符如“+、-、*、/、%、^、==、<、>”等，单目运算arg2位置空出。例如：
(+, t1, a, b)//即t1 = a + b
(-, t2, c)//即t2 = -c

//2
(call, function_name, return_value)
//调用函数function，如果有返回值，会储存在return_value中。例如：
(call, main, 0)//调用main函数，返回值为0

//3

```

(exit) //退出函数体或调用函数等（注意没有操作数和结果，位置空出）

//4

(return, value) //函数对value进行返回。例如：

(return, t1) //即 return t1

//5

(print, value) //将操作数value打印输出，例如：

(print, t1) //将操作数t1打印输出

//6

(label, label\_name)

//将当前位置命名为label\_name，配合goto和ifgoto使用

//7

(goto, label\_name) //无条件跳转至label\_name所在的位置。例如：

17行：(goto label\_hahaha)

... ...

20行(label, label\_hahaha) //从17行跳转到20行

//8

(ifgoto, op, args1, args2, label\_name) //有条件地跳转至label\_name所在的位置，args1 op

args2为真才进行跳转，其中op为=、>、<、<=、>=等。例如：

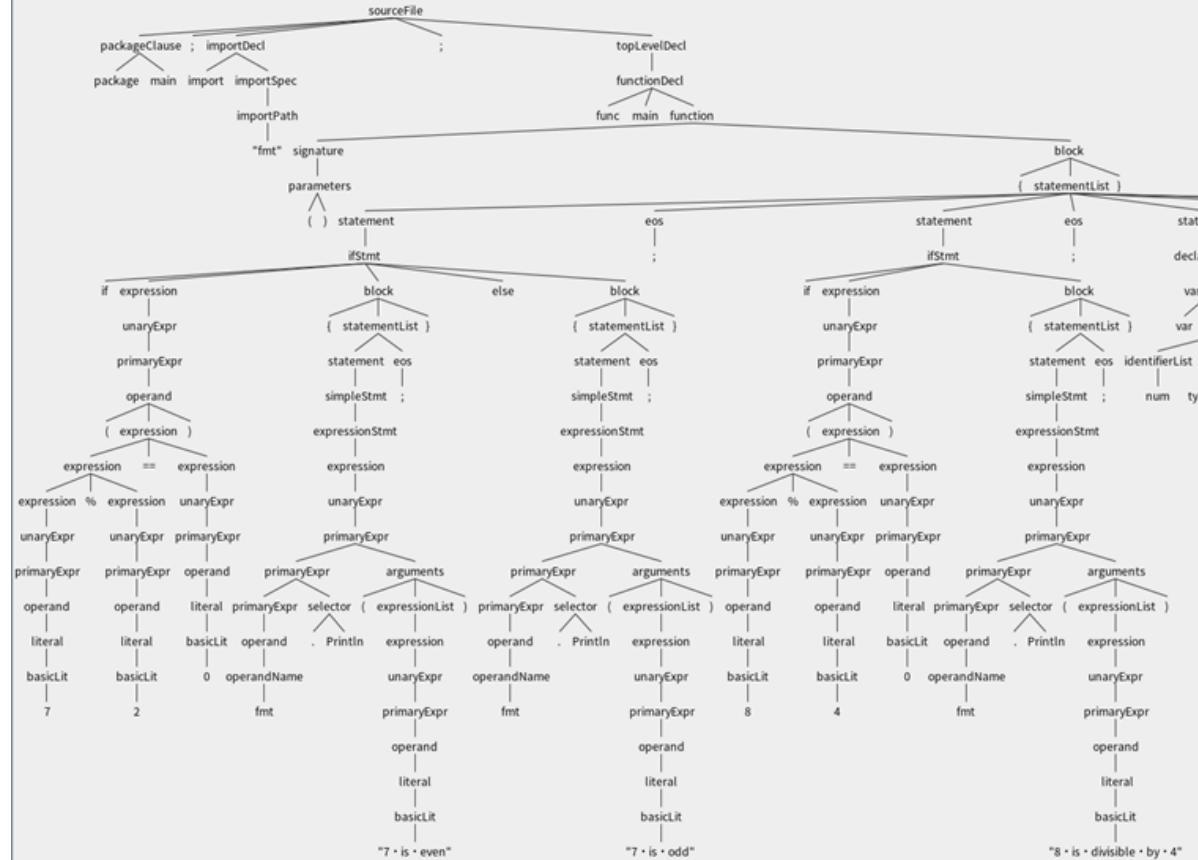
17行：(ifgoto, =, t3, 1, label\_name)

... ...

20行：(label, label\_name)

//如果t3 = 1则从17行直接跳转到20行执行

程序示例：



```
1,call,main,0
2,exit
3,function,main
4,%,t0,7,2
5,==,t1,t0,0
6,ifgoto,=,t1,1,label0
7, goto, label1
8, label, label0
9, string, t_str0, "7 is even"
10, print, t_str0
11, goto, label2
12, label, label1
13, string, t_str1, "7 is odd"
14, print, t_str1
15, goto, label2
16, label, label2
17, %, t2, 8, 4
18, ==, t3, t2, 0
19, ifgoto, =, t3, 1, label3
20, goto, label4
21, label, label3
22, string, t_str2, "8 is divisible by 4"
23, print, t_str2
24, label, label4
25, =, num, 45
26, <, t4, num, 0
27, ifgoto, =, t4, 1, label5
```

### 三地址码生成中的错误处理

在 `symtab.h` 中，主要解决变量未声明的问题。

定义是否处于错误状态的变量：`bool error_status = 0;`

变量未定义的错误：

```
void check_entry(string ident, table *t){
    if(!lookup(ident, t)){
        cout << "ERROR: variable " << ident << " NOT declared.\n";
        error_status = 1;
    }
}
```

其中的 `lookup` 函数如下：

```

bool lookup(string ident, table * t1){
    int check = 0;
    //while(t1!=NULL){
    table * temp1 = t1;
    //cout << "LOOKING FOR " << ident << endl;
    while(temp1->table_type!=1){
        if(look_in_map(ident,temp1->symtab)==1){
            check=1;
            break;
        }
        temp1=temp1->parent;
    }
    return check;
}

```

`look_in_map`是我们定义的一个布尔函数：

```

bool look_in_map(string ident,symboltable *symtab1){
    int check = 0;

    symboltable &symtab = *symtab1;
    unordered_map<string, s_entry>:: iterator itr;
    if(symtab.size()==0){
        check=0;
    }
    else{
        for(itr=symtab.begin();itr!=symtab.end();itr++){
            if(itr->first==ident){
                check = 1;
                break;
            }
        }
    }
    return check;
}

```

`table`的结构：

```

typedef struct table{
    symboltable* symtab;
    table *parent; //=NULL;
    int table_type=0;
    //table *child=NULL;
}table;

```

而在 `parserIR.y` 文件中，关于错误处理的部分，Yacc自己提供的错误信息报告函数为`yyerror()`，源程序如下：

```

void yyerror (const char *s)
{
    fprintf (stderr, "%s\n", s);
}

```

利用此函数，即可帮助实现任务所需的错误分析与恢复。

# 生成汇编指令

汇编码设计参考了MIPS，包括R型、I型和J型指令。转换分为三个关键部分：

第一部分为指令结果生成，我们规定了用于不同类型指令和数据段的汇编码生成方法，程序执行过程中会检查三地址码类型并进行相应转换。

第二部分为符号表生成，我们将三地址码划分为基本块，并记录基本块中每条语句的活动及下一步使用信息，逐块生成汇编代码；对于每个基本块中的变量及对应值，都会建立相应的符号表。

第三部分为寄存器描述符的设置，我们的程序会启发式地为每条指令查找合适寄存器，并实时更新寄存器描述符。

我们的主汇编码生成程序，包含：

1. 一个包含用于不同类型指令和数据段的x8代码生成方法的codegenerator类，它能够检查指令类型并进行相应的转换。
2. Next\_use函数，它通过向后传递来确定基本块中每条语句的活动和下一步使用信息。然后，它通过向前传递为每个基本块生成汇编代码。最后，它将所有寄存器内容保存到内存中，并重置符号表的活动和下一次使用信息。
3. 主要功能是读取输入的ir文件并调用代码生成器的方法。Get\_Reg包含为使用next\_use启发式生成的下一个x86指令查找合适寄存器的代码。除此之外，它还包括用于寄存器描述符更新的代码，用于跳转调用的上下文保存等。

**代码架构的解析（主要包含四个文件）：**

**syntab.h:** 定义空寄存器，符号表，寄存器、地址以及代码段的map。

**tac.h:** main()函数的初始化，另含判断操作数是变量还是整数的isInteger()函数。

**tac.cpp:** 符号表生成以及寄存器描述符的设置。

根据三地址码运算符 (op) 确定操作类型，判断操作数个数并赋值（记为in1、in2和out）。

寄存器生成部分，包括寄存器的初始化、分配以及清空 (\$a1-\$a3, \$t1-\$t9, \$s1-\$s7)。

符号表生成部分，包括基本块划分（每块起始与终止位置的确定）、三地址与汇编码逐行map、变量赋值与map、内存地址计算与map、语句下一步使用情况的记录。

**translate.cpp:** 指令结果生成。

初始化blockcode（基本块），调用tac.cpp的方法向blockcode里添加汇编语句。

分为赋值、运算以及跳转三个部分（均由大量if实现）。

赋值：支持寄存器、存储器赋值以及立即数赋值，使用isInteger函数判断赋值方式。对于寄存器 (reg类型)、存储器赋值 (mem类型)，调用符号表syntab得到所需值。

运算：支持+-\*%/^&|^ (加减乘除整除与或非)。汇编码直接翻译即可，例：三地址码op1 out in1 in2直接转为汇编码op2 reg1 reg2 reg3。

跳转：支持无条件跳转goto：直接翻译为j target；

有条件跳转ifgoto：根据语句上下文得到跳转条件，并翻译为助记符，包括ble、bge、blt、bgt、beq、bne；

标签设置label：初始设置为label0；

函数调用call以及返回return（立即数、reg、mem）：保存现场（记录当前地址）并跳转至符号表对应的地址处。

此外，包含getreturn以及print。

目前可实现的部分三地址转换为汇编指令如下：

1. decl\_array,arr[N]

    定义数组

    arr-> 数组名

    N -> 数组大小

2. =,x, <symbol>

    赋值运算

    x = <symbol> symbol可以是数字、变量或数组元素

注：只允许以下两种操作：

    a[i] = t

    t = a[i]

3. call,func\_name,optional\_var

    调用函数func\_name, 如果有返回值，会储存在optional\_var中

4. ret,optional\_return\_var

    函数返回， optional\_return\_var为返回的可选变量

5. op,a,b,c     op in [&, |, ^, +, -, %, /, \*, <<, >>]

    算数及按位逻辑操作， 允许的操作符如上

6. ifgoto, leq,a,b,<line no. or label>

    跳转至某一行或某个标签

    支持无条件跳转和有条件跳转

7. print,n

    打印整数n

8. input,n

    从stdin中扫描整数n

9. op,a,b     op in [~, !, ++, --]

    对b进行一元操作并把结果储存在a中

10. label, label\_name

设置标签，功能不变

## 二进制可执行文件生成

因为软件和硬件之间隔着一个操作系统，我们将我们写好的源代码编译生成了汇编代码，但是不可执行，因为不同的操作系统操作要求不同，这时我们就需要通过汇编得到二进制文件，进而链接生成可执行文件来实现不同操作系统需要的要求。IDE设计

### 机器语言的格式

1. 机器语言指令有操作码(OP)和地址码两部分组成

|\_\_\_\_OP\_\_\_\_|d|w\_|

|\_\_\_\_OP\_\_\_\_|s|w\_| <--此格式用于立即寻址方式

在多数操作码中，常使用某些位来指示某些信息：

如图上结构里的： w = 1 时 对字来操作

w = 0 时 对字节来操作

d值在双操作数指令中才有效

当 d = 1 时 有且只有一个寄存器用于目的操作数

d = 0 时 有且只有一个寄存器用于源操作数

s = 1 时 立即数为8位，但要求扩展成16位数

s = 0 时 当指令作字节操作/有16位立即数

2. 寻址方式的机器语言表示：

| mod | reg | r/m |

|\_-|\_-|\_-|\_-|\_-|\_-|

reg 表示寄存器方式，在不包括立即数的双操作数指令的情况下，规定必须有一个操作数在寄存器中，该寄存器由reg字段指定，并与操作码字节中的w位相组合确定的寄存器。

### 汇编语言转化为机器语言

事实上，因为汇编代码和机器代码基本一一对应，这个过程并不困难。通过搜索linux的调试工具,radare2有附加的工具来转化。执行代码如下：

```
rasm2 -L #查看architectures  
rasm2 -a x86 -b 32 'mov eax, 33' #转机器码  
echo 'push eax;nop;nop' | rasm2 -f -
```

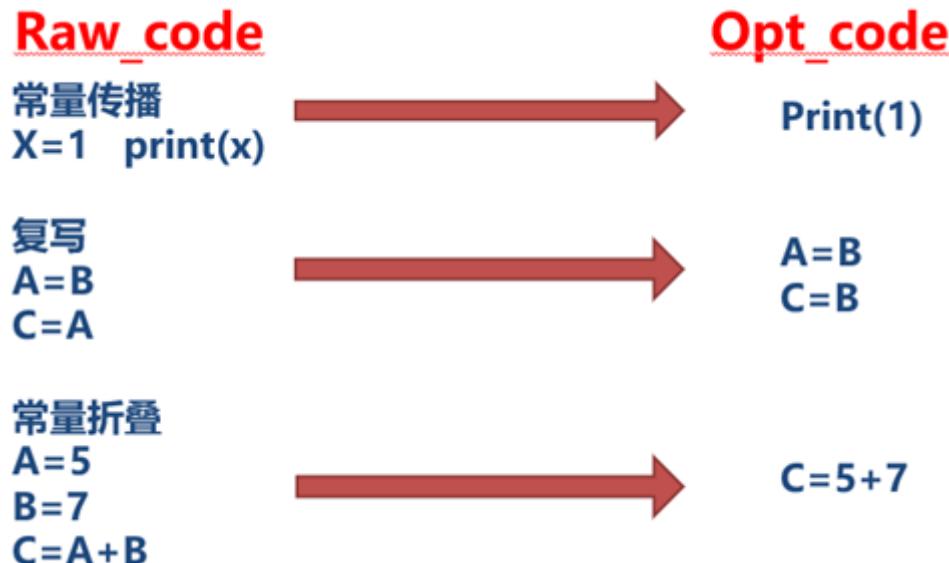
## 链接

链接过程就是要把编译器生成的一个个目标文件链接成可执行文件。最终得到的文件是分成各种段的，比如数据段、代码段、BSS段等等，运行时会被装载到内存中。各个段具有不同的读写、执行属性，保护了程序的安全运行。具体步骤如下：

1. 链接器收集main包引用的所有其它包中的符号信息，并将它们装载到一个大的字节数组中
2. 对于每个符号，链接器计算它在（数组）镜像中的地址。
3. 然后他为每个符号应用重定位，
4. 链接器准备所有ELF格式（linux系统中）文件所需的文件头。然后它再生成一个可执行的文件。

## 代码优化部分：

代码优化部分主要有下列三个板块：



可是存在的困惑是，代码优化的最终目标之一，是加快程序的执行速度，缩短程序的执行时间。但是因为小组实现的编译器还不够完善与成熟，很难确保在源程序的基础上进行代码优化的总执行时间会快于原始的程序执行时间。因此这里是需要进一步提升的地方。

## IDE设计

使用PyQt5技术。

```
"""
文件对话框 QFileDialog
最常用的是打开文件和保存文件对话框
"""

# 需求:
# 1. 打开文件，显示到窗口上
# 2. 打开文本文件，将文本文件的内容显示到窗口上

import sys
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
import subprocess

class QFileDialogDemo(QWidget):
```

```
def __init__(self):
    super(QFileDialogDemo, self).__init__()
    self.initUI()

# 编写初始化方法
def initUI(self):
    # 设置窗口标题
    self.setWindowTitle('GO COMPILER')
    # 创建垂直布局
    layout = QVBoxLayout()
    # 创建button1控件，用于加载图片
    self.button1 = QPushButton('run')
    # 创建label控件，把图像显示到label控件上
    self.imageLabel = QLabel()
    # 创建button2控件，用于加载文件
    self.button2 = QPushButton('加载文本文件')
    # 创建QTextEdit控件，来显示文本加载的内容
    self.contents = QTextEdit('显示文本加载内容')
    # 连接信号槽
    self.button1.clicked.connect(self.runTest)
    self.button2.clicked.connect(self.loadText)
    # 把控件添加到垂直布局里
    layout.addWidget(self.button1)
    layout.addWidget(self.imageLabel)
    layout.addWidget(self.button2)
    layout.addWidget(self.contents)
    # 应用于垂直布局
    self.setLayout(layout)

# 槽方法
def runTest(self):
    #print("wuhuqifei")
    #generate final code
    ss = self.contents.toPlainText()
    gogo = open('./go/gogo.go', encoding='utf-8', mode='w+')
    gogo.write(ss)
    gogo.close()
    print(ss)
    #execute test.sh
    subprocess.call("./test.sh", shell=True)

def loadText(self):
    # 直接创建QFileDialog，第二种方法
    # 创建对象
    dialog = QFileDialog()
    # 设置文件创建模式
    dialog.set FileMode(QFileDialog.AnyFile)
    # 选择文件
    dialog.setFilter(QDir.Files)

    #打开文件
    if dialog.exec():
        # 如果打开成功
        filename = dialog.selectedFiles()
        print(filename)
        # 打开文件，可以打开多个，取第一个
        f = open(filename[0], encoding='utf-8', mode='r')
```

```

# cmd = open('./test.sh',encoding='utf-8',mode='w+')
# cmd.write('#! /bin/bash\n')
# cmd.write('cd ./go\n')
# cmd.write('sh compile.sh ')
# cmd.write(filename[0])
# cmd.close()

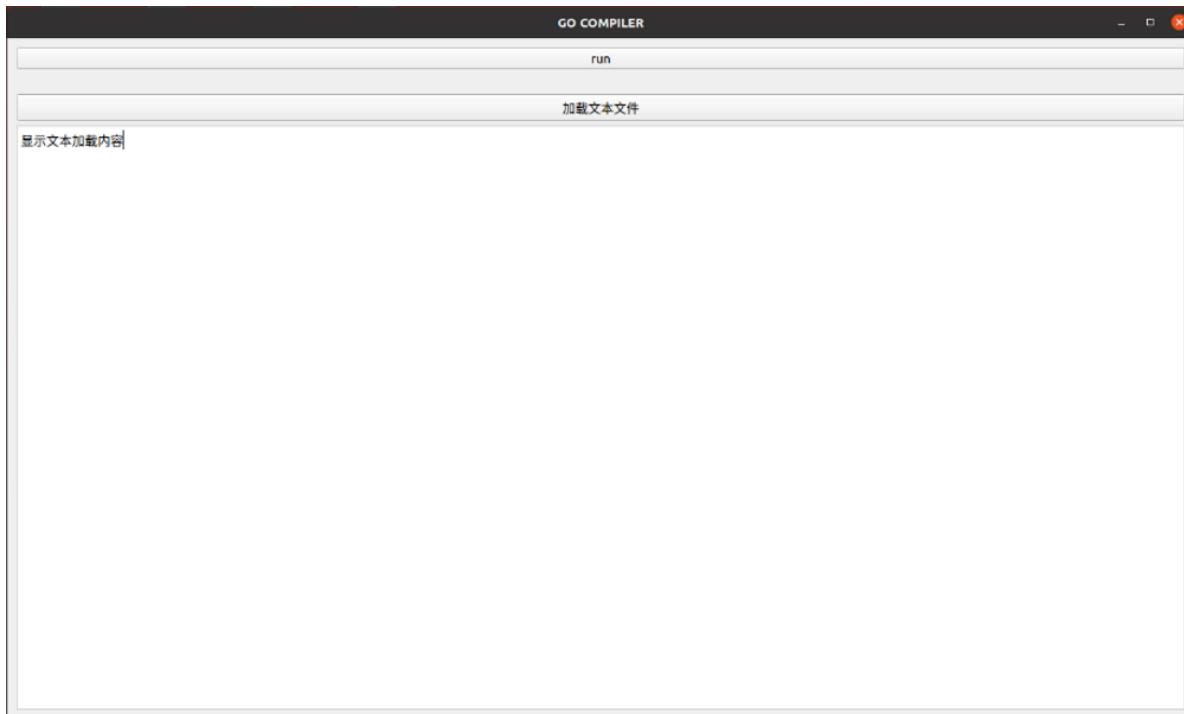
# 读取
# 使用with的原因，自动关闭，当with读取结束后，会自动调用f里面的close方法关闭文
档

with f:
    data = f.read()
    self.contents.setText(data)

# 防止别的脚本调用，只有自己单独运行时，才会调用下面代码
if __name__ == '__main__':
    # app实例化，并传递参数
    app = QApplication(sys.argv)
    # 创建对象
    main = QFileDialogDemo()
    # 创建窗口
    main.show()
    # 进入程序的主循环，通过exit函数
    sys.exit(app.exec_())

```

在终端执行 `python test1.py` 即可进行执行与演示。示例如下：



## 工程演示示例

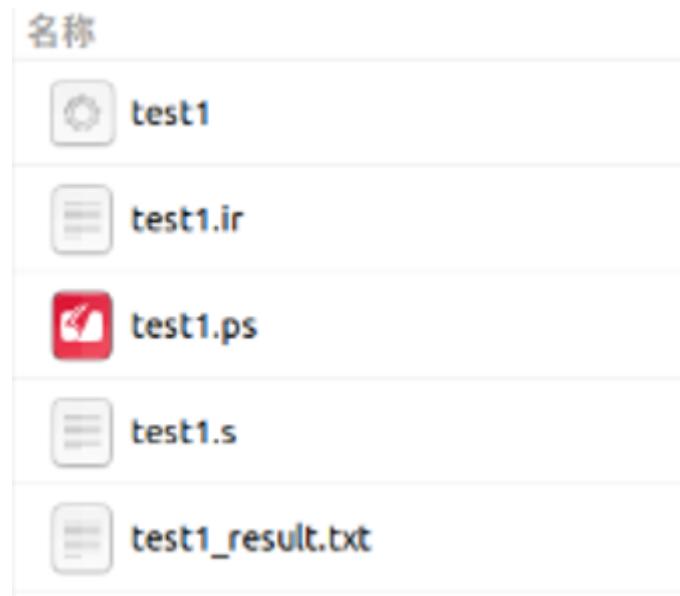
`test1.go (正确的文件)`

```
package main;
import "fmt";
func main() {

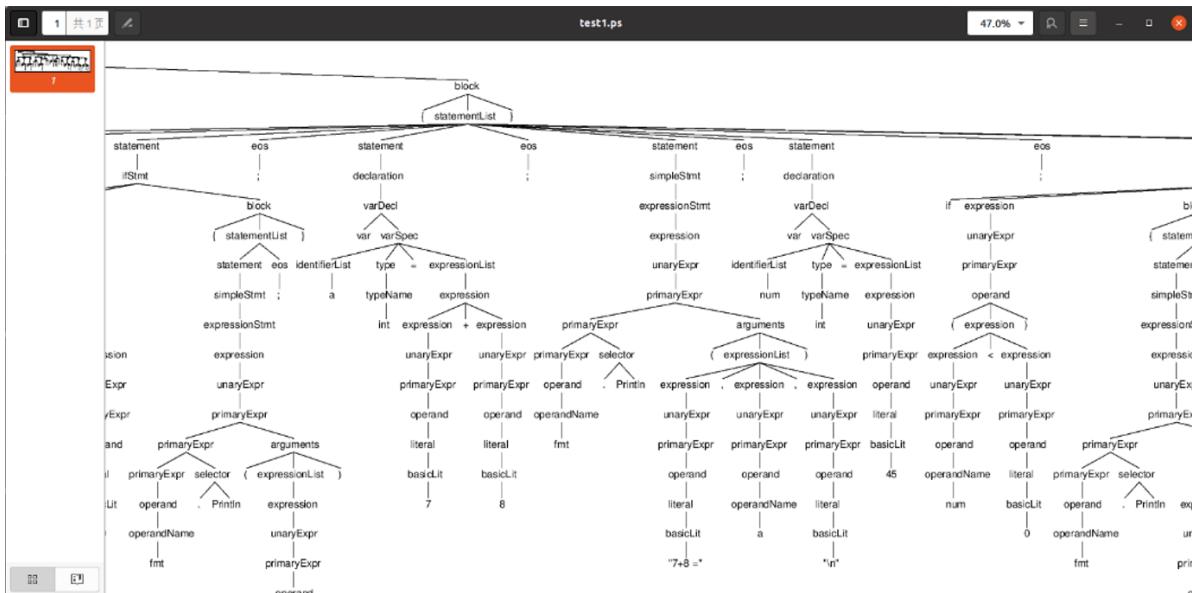
    if (7%2 == 0){
        fmt.Println("7 is even\n");
    }else {
        fmt.Println("7 is odd\n");
    }

    if (8%4 == 0) {
        fmt.Println("8 is divisible by 4\n");
    }
    var a int = 7+8;
    fmt.Println("7+8 =",a,"\\n");
    var num int = 45;
    if (num < 0) {
        fmt.Println(num, " is negative\\n");
    } else if (num < 10) {
        fmt.Println(num, " has 1 digit\\n");
    } else {
        fmt.Println(num, " has multiple digits\\n");
    }
}
```

在运行之后，会生成如下文件：（从上到下，分别为二进制文件、中间代码文件、语法分析树矢量图、汇编指令文件、程序的结果输出文件）



test1.ps:语法分析树



## test1.ir: 中间代码

The screenshot shows a graphical user interface with three windows:

- test1.ir**: The leftmost window displays the source code in a syntax-highlighted text editor. The code consists of numbered lines of assembly-like pseudocode, including labels, jumps, prints, and arithmetic operations.
- test1.s**: The middle window shows the assembly translation of the pseudocode, with lines starting with numbers followed by assembly instructions.
- test1\_result.txt**: The rightmost window displays the execution results of the program, showing the output of the print statements.

## test1.s:汇编指令

The screenshot shows a window with three tabs: 'test1.ir' (assembly source), 'test1.s' (assembly output), and 'test1\_result.txt' (execution results). The assembly code includes various instructions like .word, .asciz, li, rem, seq, sw, lw, beq, and syscall, along with comments explaining the purpose of certain values (e.g., "has 1 digit\n", "has multiple digits\n"). The assembly output file 'test1.s' contains the generated assembly code. The results file 'test1\_result.txt' contains the output of the program execution.

```
1 .data
2 t4: .word 0
3 a: .word 0
4 t1: .word 0
5 t0: .word 0
6 t2: .word 0
7 t3: .word 0
8 t6: .word 0
9 t5: .word 0
10 num: .word 0
11 t_temporary_string6: .asciz      " has 1 digit\n"
12 t_temporary_string4: .asciz      "\n"
13 t_temporary_string7: .asciz      " has multiple digits\n"
14 t_temporary_string3: .asciz      "7+8 ="
15 t_temporary_string2: .asciz      "8 is divisible by 4\n"
16 t_temporary_string5: .asciz      " is negative\n"
17 t_temporary_string1: .asciz      "7 is odd\n"
18 t_temporary_string0: .asciz      "7 is even\n"
19 .text
20 main:
21 l1:    main_prog
22 j     main_prog
23 main_prog:
24     li      $t8 , 7
25 rem   $a1, $t8, 2
26 seq   $a2, $a1, 0
27 sw    $a2, t1
28 sw    $a1, t0
29 lw    $a3, t1
30 beq  $a3, 1, label0
31 l6:
32 j     label1
33 label0:
34     li      $v0, 4
35 la    $a0, t_temporary_string0
36 syscall
37 j     label2
38 label1:
39     li      $v0, 4
40 la    $a0, t_temporary_string1
41 syscall
42 j     label2
```

test1\_result.txt: 执行结果文件

This screenshot shows the same multi-tabbed editor as above, but the contents of the tabs have been cleared or are empty. The tabs are labeled 'test1.ir', 'test1.s', and 'test1\_result.txt'.

test\_incorrect.go: 存在错误的go语言程序

```
package main;
import "new";
var global int;

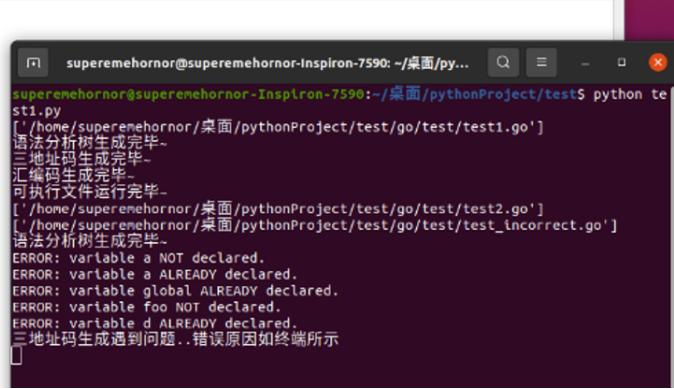
func foo2(){
    var i int;
    a++;
}

func main() {

    var a int;
    const a int; //working for now
    var global int;
    //a = 5;
    foo();
    if(a<6){
        var d string = "he";
    };
    var d *int;
    var d int;
    d += 5;
}
```

执行结果:

```
package main;
import "new";
var global int;
func foo2(){
    var i int;
    a++;
}
func main() {
    var a int;
    const a int; //working for now
    var global int;
    //a = 5;
    foo();
    if(a<6){
        var d string = "he";
    };
    var d *int;
    var d int;
    d += 5;
}
```



A terminal window titled 'superemehornor@superemehornor-Inspiron-7590: ~/桌面/pythonProject/test\$ python test1.py' displays the following output:

```
superemehornor@superemehornor-Inspiron-7590: ~/桌面/pythonProject/test$ python test1.py
[~/home/superemehornor/桌面/pythonProject/test/go/test/test1.go]
语法分析树生成完毕-
三地址码生成完毕-
汇编码生成完毕-
可执行文件运行完毕-
[~/home/superemehornor/桌面/pythonProject/test/go/test/test2.go]
[~/home/superemehornor/桌面/pythonProject/test/go/test/test_incorrect.go]
语法分析树生成完毕-
三地址码生成完毕-
汇编码生成完毕-
可执行文件运行完毕-
ERROR: variable a NOT declared.
ERROR: variable a ALREADY declared.
ERROR: variable global ALREADY declared.
ERROR: variable foo NOT declared.
ERROR: variable d ALREADY declared.
三地址码生成遇到问题..错误原因如终端所示
```

至此，任务三工作到此结束。