天津大学

编译原理实践

组　长：　朱相举（3019244319）

组　员：　李润泽（3019244266）

递铮 （3019244106）

吴俊杰（3019207400）

高建鸿（3019244316）

任课教师：　胡静老师

2022 年 12 月 20 日

目标:

设计GO语言的编译器：
1、查找资料，给出GO语言的语法描述
2、设计中间代码表示形式，建议参考三地址指令代码。见参考课件的31页至63页
3、编译器的基本功能包括：
a）给出输入程序，可以生产其对应的语法分析树
b）给出输入程序，可以生成相应的三地址指令代码
c）给出输入程序，可以生成相应的汇编指令
4、编译器的附加加分功能：
a）给出错误的输入程序，可以进行错误分析和错误恢复
b）给出正确的输入程序，可以生成可执行文件（不用一步到位，可分步骤进行，比如先生成汇编，而后再用汇编器生成可执行文件）
c）对输入程序进行优化。（可以在中间代码层面做优化，也可以在源代码或者汇编代码层面进行）
5、提交文档要尽量涵盖整个设计过程中的细节，比如，小组讨论会议纪要、设计文档、操作步骤说明、或者在设计中遇到的问题等。
6、提交文档要写明每个人的详细分工和贡献值（百分比）

任务	进度
给出输入程序，可以生产其对应的语法分析树	已完成
给出输入程序，可以生成相应的三地址指令代码	已完成
给出输入程序，可以生成相应的汇编指令	已完成
给出错误的输入程序，可以进行错误分析和错误恢复	已完成
给出正确的输入程序，可以生成可执行文件	已完成
对输入程序进行优化	未完成

摘要

该项目旨在设计一个 GO 语言的编译器。项目使用 java 作为编程语言，结果是采用 java 编写 go 语言的编译器，并且使用 ANTLR4 作为辅助工具。

ANTLR4 能够根据用户定义的语法文件自动生成词法分析器和语法分析器，并将输入文本处理为（可视化的）语法分析树。这一切都是自动进行的，所需的仅仅是一份描述该语言的语法（grammar）文件。为此，我们也相应地编写了 GO.g4 文件用于描述 GO 语言的语法（由于词法规则和文法规则的结构相似，ANTLR 允许二者在同一个语法文件中存在。不过，由于词法分析和语法分析是语言识别过程的两个不同阶段，我们必须告诉 ANTLR 每条规则对应的阶段。它是通过这种方式完成的：词法规则以大写字母开头，而文法规则以小写字母开头）。

1、解析文本，将文本转换成一个个 token，在这一步中我们使用了 ANTLR4 工具作为辅助。我们使用 ANTLR4 的 lexer 作为词法解析器，将输入文件的字符串解析成**一系列离散的 token 符号。**

2、使用 ANTLR4 生成的 parser 作为语法分析器，分析每一个 token 之间的关系生成**最初始的语法分析树。**

3、 设计 golang 语言的抽象语法树，**通过遍历 ANTLR4 生成的语法分析树来构建我们自己的抽象语法树**，以此来达到简化语法分析树的表达形式的效果。

一、 GO 语言的语法描述

由于 ANTLR 的词法规则可以包含递归，从技术角度上看，词法分析器变得和语法分析器一样强大。这意味着我们可以甚至可以在词法分析器中匹配语法结构。或者，另外一种极端是，我们可以把字符看作词法符号，然后用语法分析器来分析字符流的语法结构(这种情况称为无扫描器的语法分析器(scannerless parser))。

划定词法分析器和语法分析器的界线位置不仅是语言的职责，更是语言编写的应用程序的职责。幸运的是，我们可以得到一些经验法则的指导。

• 在词法分析器中匹配并丢弃任何语法分析器无须知晓的东西。对于编程语言来说，要识别并丢弃的就是类似注释和空白字符的东西。否则，语法分析器就需要频繁检查它们是否存在于词法符号之间。

• 由词法分析器来匹配类似标识符、关键字、字符串和数字的常见词法符号。语法分析器的层级更高，所以我们不应当让它处理将数字组合成整数这样的事情,这会加重它的负担。

• 将语法分析器无须区分的词法结构归为同一个词法符号类型。例如，如果我们的程序对待整数和浮点数的方式是一致的，那就把它们都归为 NUMBER 类型的词法符号。没必要传给语法分析器不同的类型。

• 将任何语法分析器可以以相同方式处理的实体归为一类。例如，如果语法分析器不关心 XML 标签的内容，词法分析器就可以将尖括号中的所有内容归为一个名为 TAG 的词法符号类型。

• 另一方面,如果语法分析器需要把一种类型的文本拆开处理，那么词法分析器就应该将它的各组成部分作为独立的词法符号输送给语法分析器。例如，如果语法分析器需要处理 IP 地址中的元素，那么词法分析器就应该把 IP 地址的各组成部分(整数和点)作为独立的词法符号送入语法分析器。

1.1 词法分析

此部分主要依赖 ANTLR4 的 lexer 词法解析器工具进行词法解析生成 token。读取文件内容后将其转换成 antlr 的文件输入流，随后将文件输入流输入到 ANTLR4 生成的 lexer 工具中得到 lexer 对象，最后以 lexer 对象生成 antlr 的常规 Token 令牌流，作为语法分析器的基础组件。**在这一过程中，文件中的字符串被解析成了一个个 token 符号。**

1.1.1 数字

描述整数只是一列数字即可，而描述浮点数时，我们的规则使用了选择模式和序列模式（这里将 DIGIT 声明为 fragment 可以告诉 ANTLR，该规则本身不是一个词法符号，它只会被其他的词法规则使用）。

```
INT    :   DIGIT+;

FLOAT  :   DIGIT+ '.' DIGIT*

               |                      '.'    DIGIT+

               ;

fragment

DIGIT    :         [0-9] ;
```

1.1.2 运算符

GO 语言支持包括逻辑运算符，比较运算符，与数字运算符

逻辑运算符

符号	名称	对操作值的要求
&&	布尔与（二元）	两个操作值必须为同一布尔类型
||	布尔或（二元）	
!	布尔否（一元）	唯一的一个操作值的类型必须为布尔类型

实例（两个操作数）:

x	y	x && y	x || y	!x	!y
true	true	true	true	false	false
true	false	false	true	false	true
false	true	false	true	true	false
false	false	false	false	true	true

比较运算符

符号	名称	对两个操作值的要求

==	等于	如果两个操作数都为类型确定的，则他们的类型必须一样，或者其中一个操作数可以隐式转换为另一个操作数的类型。两者的类型必须都为可比较类型。
!=	不等于	
<	小于	两个操作值的类型必须相同并且他们的类型必须为整数类型，浮点数类型过或者字符串类型。
<=	小于或等于	
>	大于	
>=	大于或等于	

算术运算符

Go 支持五个基本二元算术运算符:

符号	名称	对两个运算数的要求
+	加法	两个运算数的类型必须相同并且为基本数值类型
-	减法	
*	乘法	
/	除法	
%	余数	两个运算数的类型必须相同并且为基本整数数值类型

Go 支持六种位运算符（也属于算术运算):

符号	名称	对两个操作数的要求以机制解释
&	位与	两个操作数的类型必须相同并且为基本整数数值类型
|	位或	
^	异或	
&^	清位	
<<	左位移	左操作数必须为一个整数，右操作数也必须为一个整数(如果它是一个常数，则它必须非负)，但它们的类型可以不同。
>>	右位移	

Go 也支持三个一元算术运算符:

符号	名称	解释
+	取正数	+n 等价于 0+n
-	取负数	-n 等价于 0-n
^	位反	^n 等价于 m^n，其中和 n 同类型并且它的二进制表示中所有比特位均为 1。比如如果 n 的类型为 int8。则 m 的值为-1;如果 n 的类型为 uint8，则 m 的值为 255

注意：

- 在很多其它流行语言中，位反运算符是用`~`表示的。
- 和一些其它流行语言一样，加号运算符`+`也可用做字符串衔接运算符（见下）。
- 和 C 及 C++语言一样，`*`除了可以当作乘号运算符，它也可以用做指针解引用运算符；`&`除了可以当作位与运算符，它也可以用做取地址运算符。
- 和 Java 不一样，Go 支持无符号数，所以 Go 不需要无符号右移运算符`>>>`。
- Go 不支持幂运算符， 我们必须使用 **math** 标准库包中的 **Pow** 函数来进行幂运算。
- 清位运算符`&^`是 Go 中特有的一个运算符。 `m &^ n` 等价于 `m & (^n)`。

在这里我们把所有情况都罗列在表达式中：

```
expr   :   (NUM | STRING | BOOL)                                    #
expr_val
       |   CHR                                          # expr_chr
       |    '-' expr
# expr_neg
       |   '(' expr ')'
# expr_ex
       |   expr op=('*'|'/'|'%'|'<<'|'>>'|'&'|'&^') expr
# expr_d
       |   expr op=('+'|'-'|'|'|'^') expr
# expr_d
       |   expr op=('=='|'>'|'<'|'>='|'<='|'!=') expr
# expr_d
       |   expr op='&&' expr                                       #
expr_d
       |   expr op='||' expr                                       #
expr_d
       ;
```

1.1.3 关键字

关键字是一些特殊的用来帮助编译器理解和解析源代码的单词。

```
grammar KeywordTest;

enumDef : 'enum' '{' … '}' ;

…

FOR : 'for' ;

…

ID : [a-zA-Z]+ ; //不会匹配'enum'和‘for’
```

由于 ID 规则也能够匹配类似 enum 和 for 的关键字，因此这意味着存在不止一种规则可以匹配相同的输入字符串。要弄清此事，我们需要了解 ANTLR 对这种混合了词法规则和文法规则的语法文件的处理机制。首先，ANTLR 从文法规则中筛选出所有的字符串常量，并将它们和词法规则放在一起。'enum'这样的字符串常量被隐式定义为词法规则，然后放置在文法规则之后、显式定义的词法规则之前。ANTLR 词法分析器解决歧义问题的方法是优先使用位置靠前的词法规则。这意味着，ID 规则必须定义在所有的关键字规则之后，在上面的例子中，它在 FOR 规则之后。ANTLR 将为字符串常量隐式生成的词法规则放在显式定义的词法规则之前，所以它们总是拥有最高的优先级。因此，在本例中，'enum'被自动赋予了比 ID 更高的优先级。下面是我们使用的一些比较常见的关键字。

```
// Keywords

BREAK                   : 'break';

DEFAULT                 : 'default';

FUNC                    : 'func';

INTERFACE               : 'interface';

SELECT                  : 'select';

CASE                    : 'case';

DEFER                   : 'defer';

GO                      : 'go';

MAP                     : 'map';

STRUCT                  : 'struct';

CHAN                    : 'chan';

ELSE                    : 'else';

GOTO                    : 'goto';

PACKAGE                 : 'package';

SWITCH                  : 'switch';

CONST                   : 'const';

FALLTHROUGH             : 'fallthrough';

IF                      : 'if';

RANGE                   : 'range';

TYPE                    : 'type';

CONTINUE                : 'continue';

FOR                     : 'for';

IMPORT                  : 'import';

RETURN                  : 'return';

VAR                     : 'var';
```

1.1.4 注释

```
WS                  : [ \t\r\n]+ -> channel(1);

COMMENT       :   '//' .*? '\n' -> channel(2);

COMMENT2           : '/*' .*? '*/' -> channel(2);
```

1.2 语法分析

此部分同样依赖于 ANTLR4 的语法分析工具。在上一步词法分析进行字符流分析的时候，Lexer 不关心所生成的单个 Token 的语法意义及其与上下文之间的关系，而这就是 Parser 的工作。**语法分析器将收到的 Tokens 组织起来，并转换成为目标语言语法定义所允许的序列。**这一步通过将上一步 lexer 解析得到的超常规 token 令牌流输入到 parser 中即可通过 antlr 工具为我们生成一个**初步的语法分析树**，这个语法分析树表述了 lexer 解析出的 token 的上下文关系。由于代码较长，我们仅仅给出了部分核心代码。详情请参见 GO.g4。

1.2.1 代码包和包导入

和很多现代编程语言一样，Go 代码包（package）来组织管理代码。 我们必须先引入一个代码包（除了 **builtin** 标准库包）才能使用其中导出的资源（比如函数、类型、变量和有名常量等）。当一个代码包被引入一个 Go 源文件时，只有此代码包中的导出资源（名称为大写字母的变量、常量、函数、定义类型和类型别名等）可以在此源文件被使用。 比如上例中的 **Println** 函数即为一个导出资源，所以它可以在上面的程序源文件中使用。

我们用一个例子来解释:

```
package main



import "fmt"

import "math/rand"



func main() {

        fmt.Printf("random number:%v。\n", rand.Uint32())

}
```

这个例子多引入了一个 **math/rand** 标准库包。 此包是 **math** 标准库包中的一个子包。 此包提供了一些函数来产生伪随机数序列。

一些解释:

- 在此例中，**math/rand** 标准库包的引入名是 **rand**。 **rand.Uint32()**函数调用将返回一个 **uint32** 类型的随机数。
- **Printf** 函数是 **fmt** 标准库包中提供的另外一个常用终端打印函数。 一个 **Printf** 函数调用必须带有至少一个实参，并且第一个实参的类型必须为 **string**。 此第一个实参指定了此调用的打印格式。此格式中的**%v** 在打印结果将被对应的后续实参的字符串表示形式所取代。 比如上列中的**%v** 在打印结果中将被 **rand.Uint32()**函数调用所返回的随机数所取代。 打印格式中的**\n** 表示一个换行符。

1.2.1 声明

声明包括了常量声明、类型声明、变量声明、函数声明。

由于前三个我们已经比较熟悉，在这里我们仅仅介绍一下函数声明;

让我们来看一个函数声明:

```
func f(num int) int {
    var res int
    if num%2==1 {
    res = num
    } else {
       res = num * num
    }
    return res
}
```

从上面的例子中，我们可以发现一个函数声明从左到右由以下部分组成：

1. 第一部分是 **func** 关键字。
2. 第二部分是函数名称。函数名称必须是一个标识符。 这里的函数名称是 **f**。
3. 第三部分是输入参数列表。输入参数声明列表必须用一对小括号括起来。 输入参数声明有时也称为形参声明，小括号内也可为空。
4. 第四部分是输出结果声明列表。在 Go 中，一个函数可以有多个返回值。 当一个函数的输出结果声明列表为空或者只包含一个匿名结果声明时，此列表可以不用一对小括号括起来；否则，小括号是必需的。
5. 最后一部分是函数体。函数体必须用一对大括号括起来。 一对大括号和它其间的代码形成了一个显式代码块。 在一个函数体内，**return** 关键字可以用来结束此函数的正常向前执行流程并进入此函数的退出阶段。

```
prog      :   ((funStat | preStat | varDefineStat |

constDefineStat)  END_STAT?)*;


preStat   :   (packageStat | importStat);


funStat       :   'func' CHR '(' parameters? ')' type? content;


importStat    :   'import' STRING;


packageStat   :   'package' CHR;
```

1.2.2 代码块

这里面包含了各种赋值、循环、分支语句语法的定义。以下仅介绍两种流程控制代码块。

1. **if-else 条件分支控制代码块**

2. **for 循环代码块**

```
stat   :  varDefineStat
       |  constDefineStat
       |  ifStat
       |       ifelseStat
       |  assignStat
       |  returnStat
       |  forStat
       |  printStat
      ;

assignStat       :  leftValueList '=' rightValueList;

varDefineStat :  'var' leftValueList type? '=' rightValueList
             |  'var' leftValueList type;

constDefineStat  :  'const'   leftValueList type? '='
rightValueList
             |  'const' '(' constList ')';




printStat       :   'putc' '(' expr ')';

returnStat       :  'return' expr?;

constList     :  constEle+;
```

1.2.3 表达式

```
expr   :   (NUM | STRING | BOOL)                          # expr_val
       |   CHR                                          # expr_chr
       |    '-' expr                                          #
expr_neg
       |   '(' expr ')'                                 # expr_ex
       |   expr op=('*'|'/'|'%'|'<<'|'>>'|'&'|'&^') expr
# expr_d
       |   expr op=('+'|'-'|'|'|'^') expr                  # expr_d
       |   expr op=('=='|'>'|'<'|'>='|'<='|'!=') expr      # expr_d
       |   expr op='&&' expr                              # expr_d
       |   expr op='||' expr                              # expr_d
       ;
```

二、 编译器设计

2.1 语法解析树的遍历

我们知道一棵解析树是包含代码所有语法信息的树型结构，它是代码的直接翻译。所以解析树，也被称为具象语法树。

ANTLR4 支持两种语法树遍历方式，第一种是监听者模式 listener，第二种是访问者 visitor，两种模式的主要区别就是监听者是处于被动方式访问语法树，每次到达一个节点的时候调用相应的监听者函数，而 visitor 则是属于主动访问语法树的方式，在每个节点中需要主动调用相应的的 visitor 函数来递归访问语法树节点。监听器和访问器机制的最大区别在于，监听器方法不负责显式调用子节点的访问方法，而访问器必须显式触发对子节点的访问以便树的遍历过程能够正常进行。因为访问器机制需要显式调用方法来访问子节点，所以它能够控制遍历过程中的访问顺序，以及节点被访问的次数。

2.2 生成相应的三地址指令代码

三地址码（Three Address Code）是一种最常用的中间语言，编译器可以通过它来改进代码转换效率。每个三地址码指令，都可以被分解为一个四元组（4-tuple）的形式：（运算符，操作数 1，操作数 2，结果）。由于每个陈述都包含了三个变量，即每条指令最多有三个操作数，所以它被称为三地址码。

中间语言（Intermediate language），有时也称为中间表示（Intermediate Rrepresentation，IR）在计算机科学中，是指一种应用于抽象机器（abstract machine）的编程语言，它设计的目的，是用来帮助我们分析计算机程序。这个术语源自于编译器，在编译器将源代码编译为目的码的过程中，会先将源代码转换为一个或多个的中间表述，以方便编译器进行最佳化，并最终产生出目的机器的机器语言。最常见的中间语言表述形式，是三位址码（Three address code），常简称为 TAC 或 3AC。

2.2.1 三地址指令代码的设计

四元式主要由四部分组成：OP，arg1，arg2，result，即（操作符，操作数 1，操作数 2，结果），其中，OP 是运算符，arg1，arg2 分别是第一和第二个运算对象，result 是编译程序为存放中间运算结果而引进的变量，常称为临时变量。当 OP 是一目运算时，常常将运算对象定义为 arg1。

例如 X = a*b+c/d 的四元式序列：

```
t1=a*b

t2=c/d

t3=t1+t2

X=t3
```

常用的三地址码（三地址码形式和四元组形式）

指令类型	指令形式	备注
赋值指令	x = y op zx = op y	op 为运算符
复制指令	x = y	
条件跳转	if x relop y goto n	relop 为关系运算符
非条件跳转	goto n	跳转到地址 n 的指令

参数传递	param x	将 x 设置为参数
过程调用	call p,n	p 为过程的名字 n 为过程的参数的个数
过程返回	return x	
数组引用	x=y[i]	i 为数组的偏移地址，而不是下标
数组赋值	x[i]=y	
地址及指针操作	x=&yx=*y *x=y	

```
    private void Assign(String str) {
        write("@ASSIGN " + str);
    }

    private void Cal(String str) {
        write("@ASSIGN " + str);
    }

    private void Check(String str) {
        write("@CHECK " + str);
    }

    private void Delete(String str) {
        write("@DELETE " + str);
    }

    private void Param(String str) {
        write("@PARAM " + str);
    }

    private void function(String str) {
        write("@FUNCTION" + str);
    }

    private void point(String str) {
```

2.2.2 三地址指令代码的生成

在三地址指令代码生成这一步，主要就是通过遍历在上一环节生成的包容器以及其中包含的 Func 抽象函数结构，相当于以函数声明节点为根节点进行抽象语法树的遍历。

在遍历过程中生成中间代码，与普通遍历抽象语法树不同的一点就是，生成中间代码的时候可以忽略如类型声明之类的不会影响值操作的语句，只需要遍历如赋值之类的操作即可，同时还有变量声明中涉及到初始化部分的节点。

```
@FUNCTION f int

@PARAM num int

@DEFINE res int

@DEFINE __TMP1 int

@ASSIGN __TMP1 = num

@DEFINE __TMP2 int

@ASSIGN __TMP2 = 2

@ASSIGN __TMP1 % __TMP2

@DELETE __TMP2

@DEFINE __TMP3 int

@ASSIGN __TMP3 = 1

@ASSIGN __TMP1 == __TMP3

@DELETE __TMP3

@TOECX __TMP1

@DELETE __TMP1

@CHECK 1

@DEFINE __TMP4 int

@ASSIGN __TMP4 = num

@ASSIGN res = __TMP4

@DELETE __TMP4

@JUMP 2

@POINT 1
```

2.3 生成相应的汇编指令

我们通过三地址指令代码生成汇编代码，首先我们通过 switch 语句进行操作符的匹配，由于代码内容过长，这里只展示一部分用于参考。在于操作符匹配成功后，我们再匹配其余的操作数，将操作数压入栈中，当需要用时再将操作数从栈中弹出到寄存器中进行计算。例如 z = x + y，如果三者皆为局部变量且类型为 int，那我们首先需要获得 x 这个操作数，根据偏移量结合寄存器 rbp 获取 x 的地址，随后将其压入栈中，y 同理。随后要做的操作就是从栈中取出操作数然后根据操作符进行相应操作。

操作数可以简单分为几种情况，第一种情况就是普通变量类型，这种情况下就是同上面例子一样，根据偏移以及寄存器获取变量的地址压入栈中。第二种情况就是临时变量类型，如果操作数是 arg1 或 arg2 则不需要进行任何操作，因为这个时候临时变量的结果就存在栈中，需要的时候直接弹栈即可。还有一种情况就是常规值操作，比如常数 1，这种时候就只需要将常数直接压入栈中即可，不需要进行其他操作。

```
switch (ss[0]) {
    case "@CALL":
        int total_params = func_param.get(ss[1]).intValue();
        for (int j = total_params - 1; j >= 0; j--)
            asm_part.add("movl " + (stack + 4 * j) + "(%rbp), " +
param_reg[total_params - j - 1]);
        asm_part.add("call " + ss[1]);
        break;
    case "@RETURN":
        asm_part.add("addq $" + (num_var * 16 + 1024) +
", %rsp");
        asm_part.add("popq %rbp");
        asm_part.add("ret");
        break;
    ……
    default:
        System.out.println("Invalid mid-code at " + i + "-th
line: " + mid.get(i));
        System.exit(0);
    }
```

```
    .globl main

    .LC0:

    .ascii "%d\n\0"

    f:

    pushq %rbp

    movq %rsp, %rbp

    subq $1144, %rsp

    movl %ecx, -4(%rbp)

    movl -4(%rbp), %eax

    movl %eax, -12(%rbp)

    movl $2, %eax

    movl %eax, -16(%rbp)

    movl -12(%rbp), %eax
```

2.4 生成相应的可执行文件

生成的汇编满足 AT&T 标准。可以直接用 gcc 得到可执行文件。

2.5 图形界面设计

2.5.1 语言框架选择

本任务是针对任务 3 实现一个可以展示的图形用户界面，涉及到 gui 框架的选用。前期考察了 java 原生 gui 工具包 swing 和 python 原生 gui 工具包 tkinter。由于负责这一块的同学之前有使用 tkinter 库的相关经验，故本次任务使用 tkinter 完成。

2.5.2 Tkinter 介绍

Tkinter 是 Python 的标准 GUI 库。Python 使用 Tkinter 可以快速的创建 GUI 应用程序。由

于 Tkinter 是内置到 python 的安装包中、只要安装好 Python 之后就能 import Tkinter 库、而且 IDLE 也是用 Tkinter 编写而成、对于简单的图形界面 Tkinter 还是能应付自如。

2.5.2 控件介绍

由于本项目的图形界面需要展示的东西并不是很多，因而所要用到的控件也不是很多，主要有如下几类：

控件名称	描述
Button	按钮控件；在程序中显示按钮。
Label	标签控件；可以显示文本和位图。
Entry	输入控件；用于显示简单的文本内容。

2.5.3 标准属性

本项目中用到的控件属性如下：

属性	描述
Dimension	控件大小
Color	控件颜色
Font	控件字体
Cursor	光标

2.5.4 具体实现

通过演示展示。

2.6 编译优化

2.6.1 死代码检测

死代码指的是程序中不可达的（**unreachable**）代码（即不会被执行的代码），或者是执行结果永远不会被其他计算过程用到的代码。去除死代码可以在不影响程序输出的前提下简化程序、提高效率。

2.6.2 不可达代码

一个程序中永远不可能被执行的代码被称为不可达代码。我们考虑两种不可达代码：

控制流不可达代码（**control-flow unreachable code**）和分支不可达代码（**unreachable branch**）。这两种代码的介绍如下。

控制流不可达代码. 在一个方法中，如果不存在从程序入口到达某一段代码的控制流路径，

那么这一段代码就是控制流不可达的。比如，由于返回语句是一个方法的出口，所以跟在它后面的代码是不可达的。例如在下面的代码中，第 4 行和第 5 行的代码是控制流不可达的：

```
int controlFlowUnreachable() {
    int x = 1;
    return x;
    int z = 42; // control-flow unreachable code
    foo(z); // control-flow unreachable code
}
```

分支不可达代码有两种分支语句：**if** 语句和 **switch** 语句。它们可能会导致分支不可达代码的出现。

对于一个 **if** 语句，如果它的条件值（通过常量传播得知）是一个常数，那么无论程序怎么执行，它两个分支中的其中一个分支都不会被走到。这样的分支被称为**不可达分支**。该分支下的代码也因此是不可达的，被称为分支不可达代码。如下面的代码片段所示，由于第 3 行 **if** 语句的条件是永真的，所以它条件为假时对应的分支为不可达分支，该分支下的代码（第 6 行）是分支不可达代码。

```
int unreachableIfBranch() {
    int a = 1, b = 0, c;
    if (a > b)
        c = 2333;
    else
        c = 6666; // unreachable branch
    return c;
}
```

对于一个 **switch** 语句，如果它的条件值是一个常数，那么不符合条件值的 **case** 分支就可能是不可达的。如下面的代码片段所示，第 3 行 **switch** 语句的条件值（变量 **x** 的值）永远是 2，因此分支 “**case 1**” 和 “**default**” 是不可达的。注意，尽管分支 “**case 3**” 同样没法匹配上条件值（也就是 2），但它依旧是可达的，因为控制流可以从分支 “**case 2**” 流到它。

```
int unreachableSwitchBranch() {
    int x = 2, y;
    switch (x) {
        case 1: y = 100; break; // unreachable branch
        case 2: y = 200;
        case 3: y = 300; break; // fall through
        default: y = 666; // unreachable branch
    }
    return y;
}
```

检测方式：为了检测分支不可达代码，我们需要预先对被检测代码应用常量传播分析，通过它来告诉我们条件值是否为常量，然后在遍历 **CFG** 时，我们不进入相应的不可达分支。

2.6.3 无用赋值

一个局部变量在一条语句中被赋值，但再也没有被该语句后面的语句读取，这样的变量和语句分别被称为无用变量（**dead variable**，与活跃变量 **live variable** 相对）和无用赋值。无用赋值不会影响程序的输出，因而可以被去除。如下面的代码片段所示，第 3 行和第 5 行的语句都是无用赋值。

```
int deadAssign() {
    int a, b, c;
    a = 0; // dead assignment
    a = 1;
    b = a * 2; // dead assignment
    c = 3;
    return c;
}
```

三、功能实现，系统流程

1.功能实现

1.1.根本内容

我们的 go 语言编译器可以完成对如下的编译

1、if else 条件判断，

2、for 循环，实现循环体及循环的判定条件

3、数组，我们可以实现有序的元素序列，

4、函数，我们的编译器可以编译包含函数的 go 语言，帮助使用模块化程序设计，同时没有数据类型方面，我们实现了基本的数据类型，是整个语言系统基础。

1.2.附加功能 1——错误分析

1. 变量未声明
2. 变量与函数名混用
3. 使用或修改未初始化的数组
4. 不同数据类型之间进行赋值

1.3.附加功能 2——生成可执行文件

1.4.附加功能 3——程序优化

1.4.1.死代码检测

去除死代码可以在不影响程序输出的前提下简化程序、提高效率。

1.4.2 不可达代码

一个程序中永远不可能被执行的代码被称为不可达代码。我们检测两种不可达代码：控制流不可达代码（**control-flow unreachable code**）和分支不可达代码（**unreachable branch**）

1.4.3 无用赋值

一个局部变量在一条语句中被赋值，但再也没有被该语句后面的语句读取，因而可以被去除。

2.系统流程

构建语法 ->go.g4 ->词法分析，token->语法分析->做出 AST->语义分析->记录信息，检查错误，名字乱用，建出符号表，分配临时变量，判断类型->三地址码 ->codegen ->汇编 ->gcc -> 可执行文件。

我们的工作从 GO 语言的语法描述开始，由于词法规则和文法规则的结构相似，ANTLR 允许二者在同一个语法文件中存在，我们使用了 GO.g4 一个文件进行了 GO 语言的语法描述，由于词法分析和语法分析是语言识别过程的两个不同阶段，我们必须告诉 ANTLR 每条规则对应的阶段。它是通过这种方式完成的：词法规则以大写字母开头，而文法规则以小写字母开头。并以此生成 ANTLR4 的词法解析器和语法分析器作为工具文件。

词法分析主要依赖 ANTLR4 的 lexer 词法解析器工具进行词法解析生成 token。读取文件内容后将其转换成 antlr 的文件输入流，随后将文件输入流输入到 ANTLR4 生成的 lexer 工具中得到 lexer 对象，最后以 lexer 对象生成 antlr 的常规 Token 令牌流，作为语法分析器的基础组件。**在这一过程中，文件中的字符串被解析成了一个个 token 符号。**

其中，符号分为关键字、标点符号、逻辑运算符、关系运算符、算术运算符号、单目运算符、混合运算符、数字、隐含符号、带反斜杠的符文、字符和字符串等。

语法分析部分同样依赖于 ANTLR4 的语法分析工具。在上一步词法分析进行字符流分析的时候，Lexer 不关心所生成的单个 Token 的语法意义及其与上下文之间的关系，而这就是 Parser 的工作。**语法分析器将收到的 Tokens 组织起来，并转换成为目标语言语法定义所允许的序列。**这一步通过将上一步 lexer 解析得到的超常规 token 令牌流输入到 parser 中即可通过

antlr 工具为我们生成一个**初步的语法分析树**，这个语法分析树表述了 lexer 解析出的 token 的上下文关系。

我们这里举很有代表性的函数声明为例

```
func f(num int) int {
    var res int
    if num%2==1 {
   res = num
    } else {
       res = num * num
    }
    return res
}
```

从上面的例子中，我们可以发现一个函数声明从左到右由以下部分组成：

1. 第一部分是 **func** 关键字。
2. 第二部分是函数名称。函数名称必须是一个标识符。 这里的函数名称是 **f**。
3. 第三部分是输入参数列表。输入参数声明列表必须用一对小括号括起来。 输入参数声明有时也称为形参声明，小括号内也可为空。
4. 第四部分是输出结果声明列表。在 Go 中，一个函数可以有多个返回值。 当一个函数的输出结果声明列表为空或者只包含一个匿名结果声明时，此列表可以不用一对小括号括起来；否则，小括号是必需的。
5. 最后一部分是函数体。函数体必须用一对大括号括起来。 一对大括号和它其间的代码形成了一个显式代码块。 在一个函数体内，**return** 关键字可以用来结束此函数的正常向前执行流程并进入此函数的退出阶段。

因此我们可以用如图所示进行定义，在这里我们只截取了有关代码部分。在与'func'关键字匹配成功后我们匹配 CHR 在这里即为函数名，匹配玩括号后括号内的参数为可有可无，随后匹配可能存在的返回值，最后 content 对应函数体，匹配有一对花括号包裹的语句。

```
CHR    :   [a-zA-Z_][a-zA-Z0-9_]*;


END_STAT  :   ';';


prog      :   ((funStat | preStat | varDefineStat |

constDefineStat)  END_STAT?)*;


funStat       :   'func' CHR '(' parameters? ')' type? content;


content   :   '{' (stat END_STAT?)* '}';
```

下一步则为抽象语法树的构建。

在构建抽象语法树环节我们通过遍历上一步 antlr 工具构建的简易语法分析树来构建抽象语法树。ANTLR4 支持两种语法树遍历方式，第一种是监听者模式 listener，第二种是访问者 visitor，两种模式的主要区别就是监听者是处于被动方式访问语法树，每次到达一个节点的时候调用相应的监听者函数，而 visitor 则是属于主动访问语法树的方式，在每个节点中需要主动调用相应的的 visitor 函数来递归访问语法树节点。监听器和访问器机制的最大区别在于，监听器方法不负责显式调用子节点的访问方法，而访问器必须显式触发对子节点的访问以便树的遍历过程能够正常进行。在这里我们采用了监听者模式。

接下来我们要设计中间代码，我们采用了其中老师推荐的表述形式：三地址指令代码。每个三地址码指令，都可以被分解为一个四元组（4-tuple）的形式：（运算符，操作数 1，操作数 2，结果）。

由于每个陈述都包含了三个变量，即每条指令最多有三个操作数，所以它被称为三地址码。在三地址指令代码生成这一步，主要就是通过遍历在上一环节生成的包容器以及其中包含的 Func 抽象函数结构，相当于以函数声明节点为根节点进行抽象语法树的遍历。

在遍历过程中生成中间代码，与普通遍历抽象语法树不同的一点就是，生成中间代码的时候可以忽略如类型声明之类的不会影响值操作的语句，只需要遍历如赋值之类的操作即可，同时还有变量声明中涉及到初始化部分的节点。在这里我们定义两个栈，其中一个用于存储临时

变量（临时变量举个例子，a = b + 1，我们首先将 b 提取出来形成临时变量 tmp1，将其与 1 做加法运算形成 tmp2，再将 tmp2 赋予 a），另一个用于存储当前函数的局部变量，在我们进入函数并遍历语句时将其逐个压栈，退出函数时将其清空。并建立哈希表存储每个变量的类型，这对于我们进行类型检查至关重要。

之后便是将三地址码转化为汇编代码，分为两个部分，第一部分用于生成汇编代码头部信息；第二部分首先识别每行三地址码的操作符，如果仅仅是@DEFINE 或者@PARAM 这类我们暂且不作处理仅仅是对于汇编代码的语句计数等进行 + 1；如果遇到标志函数开始的操作符，我们则需记录该位置，并在遇到函数截止操作符时对中间的部分即函数部分的代码调用 procedure 方法进行汇编代码的转换。通过 switch 方法对每种情况进行讨论。

例如最经典并且最常用的赋值语句中的加法。在成功匹配到@ASSIGN 操作符后，我们首先利用四元组的第二部分以及第四部分即两个操作数，将其分别赋值给变量 a,b；随后利用四元组的第三部分，即操作符。在匹配到加法操作符后，我们的操作如图所示，我们将 b 的值移动到寄存器 eax，随后我们将 eax 寄存器的值与 a 相加。对于其余的操作也是类似的道理，我们需要参考寄存器的使用方式进行代码的编写。