

合肥工业大学

系统软件综合设计报告

编译原理分册

设计题目 _____

学生姓名 _____

学 号 _____

专业班级 计算机科学与技术 _____

指导教师 _____

完成日期 2020.8.22 _____

报告目录

一、设计目的及设计要求.....	3
二、开发环境描述	3
三、设计内容、主要算法描述.....	4
四、设计的输入和输出形式.....	30
五、程序运行（测试、模拟）的结果（屏幕拷贝、生成结果的打印输出）	31
六、总结（体会）	37
七、源程序清单（部分核心代码）作为报告的附件。	38

一、设计目的及设计要求

设计目的

简单 C 语言编译器是本人参考课本和编译原理龙书，结合自己所学知识，并且查阅网上内容后自己完成的一个比较简易的编译器，该程序可以对部分 C 语言语句进行词法分析，语法分析，语义分析并产生三地址代码和四元式

设计要求

过递归下降方法完成对部分 C 语言语句的词法分析，语法分析，语义分析，以及中间代码生成（三地址代码和四元式）

二、开发环境描述

Windows 版本

Windows 10 家庭中文版
© 2019 Microsoft Corporation。保留所有权利。



系统

制造商:	ASUSTek Computer Inc.
处理器:	Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz 2.20 GHz
已安装的内存(RAM):	24.0 GB (23.9 GB 可用)
系统类型:	64 位操作系统，基于 x64 的处理器
笔和触控:	没有可用于此显示器的笔或触控输入



ASUSTek Computer Inc. 支持

网站: [联机支持](#)

windows 10 家庭中文版

处理器: Intel i7-8750H

内存: 24GB

开发软件: IntelliJ IDEA 2019.1.3 x64

使用语言: Java

三、设计内容、主要算法描述

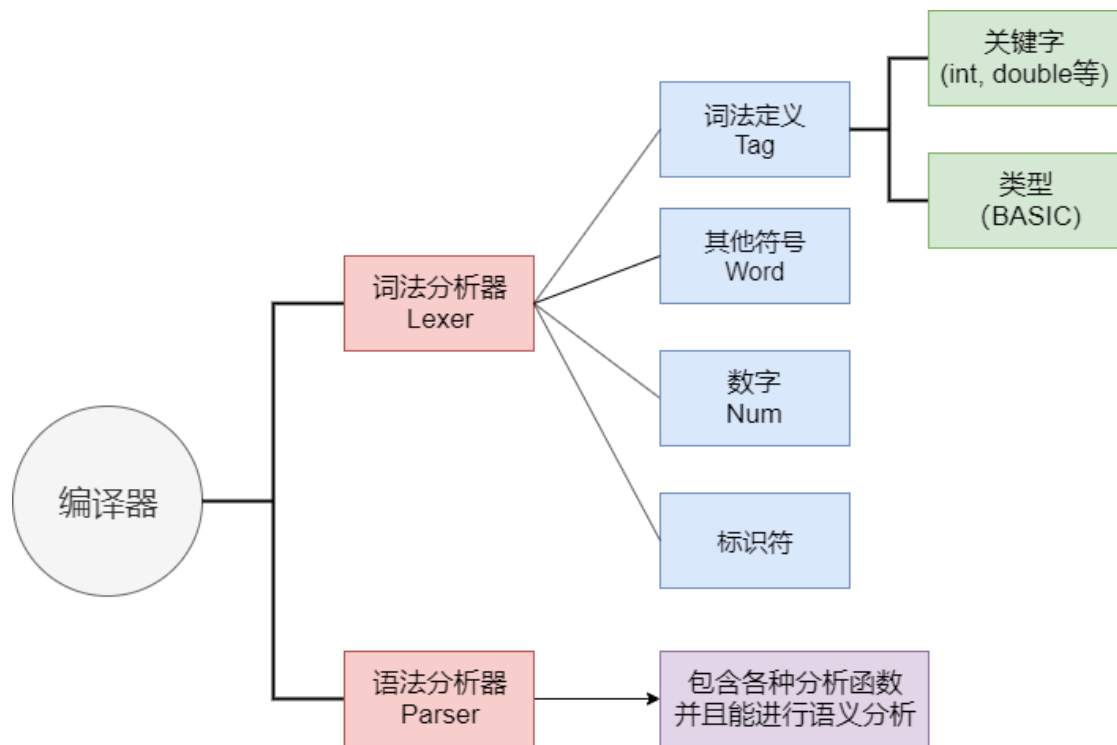
因为在实验实现了 LL(1),LR(1), 并且在前一个课设题目实现了 LALR(1),因此这次的综合性自拟题目就想用新方法来完成。新方法就是递归下降分析法。通过对词法定义, 随后通过递归下降对语法进行分析, 从而得出抽象语法树。随之产生三地址代码和四元式。最终完成对 C 语言的编译。

●递归下降分析法思想

递归下降分析法是自顶向下分析算法的一部分, 通过将各个非终结符的分析函数进行递归调用, 从而实现功能。分析高效, 并且容易实现, 并且错误定位明确。这些都是本人决定采用自顶向下分析法的原因

●编译器总体架构

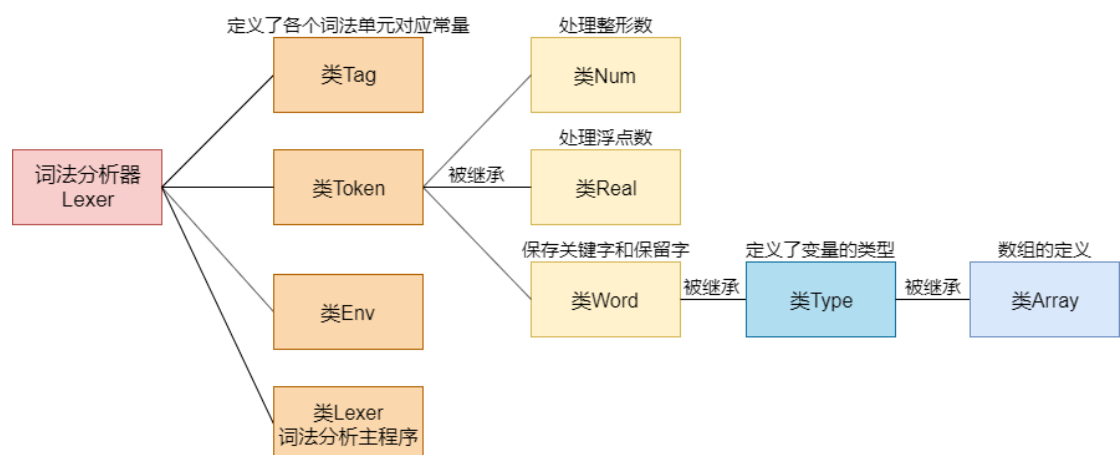
一个编译的过程需要进行词法分析, 语法分析, 语义分析及中间代码生成, 优化和目标代码生成, 而本编译器因为比较简便, 因此只具有前三个阶段, 大致组成如下图表所示



C语言编译器 词法分析与语法分析

●词法分析

C 语言是一门面向过程, 抽象化的通用程序设计语言。所使用的关键字较少, 并具有丰富的数据类型和运算符。本次编译器的设计从以下几个方面入手



C语言编译器 词法分析器结构(包括package symbols与lexer)

◎关键字

本编译器一共包含 11 种关键字，其中包含多种控制语句要用到和定义变量要用的关键字，依次是 for, while, true, false, do, if, else, int , char, float, bool 其中 while,do 可组成 do-while 和 while 语句，for 即 for 语句， if, else 可组成 if 语句，if-else 语句 和 if-else if-else 语句。这些控制语句足以实现大部分简单的程序功能。 除此之外还有定义变量的四个类型，int, char, float 和 bool

关键字	对应常量
for	FOR(801)
while	WHILE(802)
true	TRUE(831)
false	FALSE(832)
do	DO(805)
if	IF(806)
else	ELSE(807)
int	BASIC(825)——INT
char	BASIC(825) ——CHAR
float	BASIC(825) ——FLOAT
bool	BASIC(825) ——BOOL

◎数字

在先前的关键字中有定义来了四种变量的类型 int, char, float 和 bool，这就意味着本次编译器需要包含整型数和浮点数的定义，依次将其定义为 Num 类和 Real 类。浮点数和整型数在定义上没有太大的差别，只是用于存储数值的 value

分别是 int 和 float (Java 中的类型) 类型

Real 类 : public final float value;

Num 类: public final int value;

这样对数字有一个较好的定义

◎类型(Type)

贯彻始终的类型在先前定义了关键字后最终在这里进行统一的定义，Type 类继承自 Word 类，在此基础上加入了机器字长，分别遵循相应的 C 语言规则

类型	机器字长
Int	4
char	1
float	8
bool	1

同时，在该类中还定义有函数判断一个变量是否可数(numeric),这在后期对数字和其他类型的分辨和判断式子中计算是否存在语义上的错误起着关键作用

此外最重要的一个应该就是类型之间的转换了，如果一个算式 $a = b + c$

b 是 float 类型而 c 是 int 类型那么结果是什么类型，这里遵循下列规则

①两个运算操作数都是 int 或者都是 float 类型，则结果还是对应类型

②如果两个运算操作符是 int 和 float 类型的话，则结果为 int 类型

◎数组(array)

数组作为 C 语言中特别常用的一个数据结构自然也是需要定义的，数组的 Array 类继承自 Type 类，在其基础上增加了数组元素个数 size 和 数组的元素类型 of

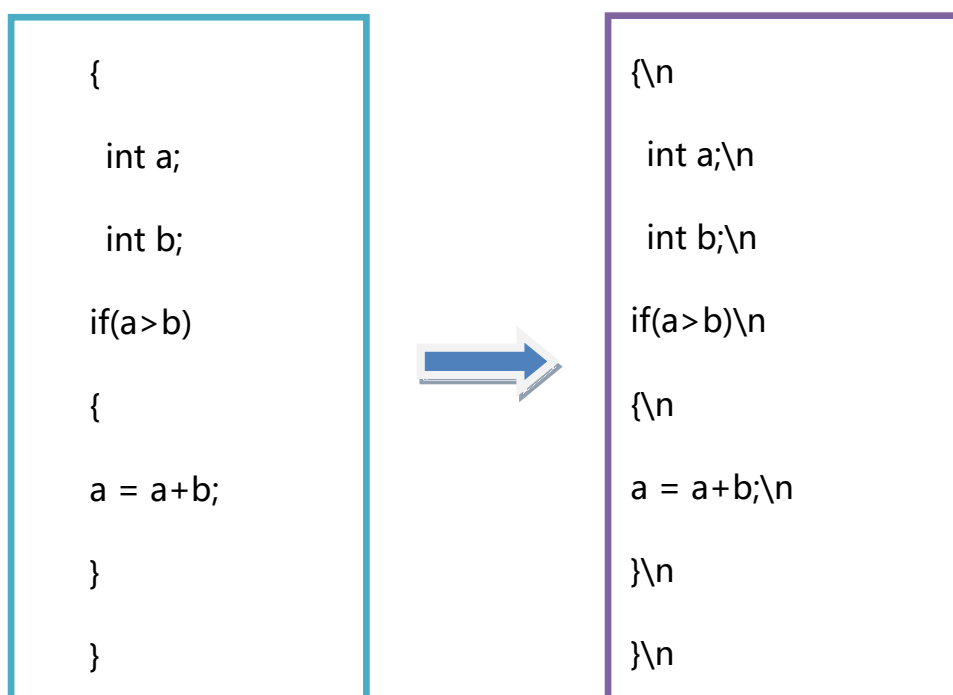
◎Env

◎词法分析主程序(Lexer)

程序可以看作是一个很长的字符串, 其中包含很多换行符, 制表符和空格等。而词法分析器主要功能就是分析当前每个词的属性, 配合语法分析器一起使用。

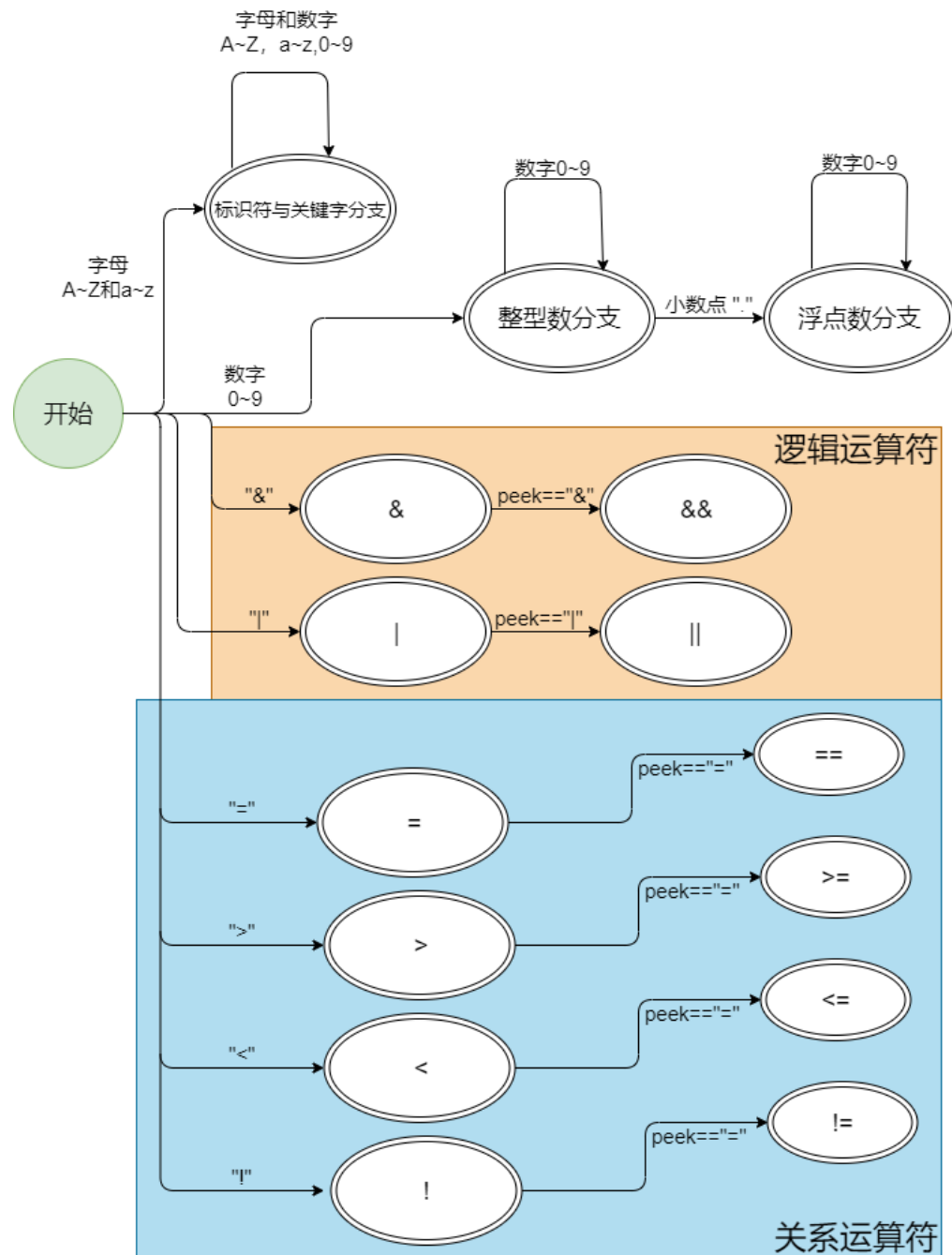
首先在词法器实例中添加入预先的关键字, 这里使用 reserve 函数对其进行添加。关键字可以使用 Hashtable 进行存储。定义完毕后就可以开始编写扫描函数了, 现有如下程序段 (左侧)

实际上这个程序段也可以表示为(右侧)



因此“ \n”也成了程序编译时判定行数的依据, 而相对无用的“ /t” 和“
”在识别的时候可以直接进行跳过

可设定一个变量 peek 存储当前遍历到的位置的字符，然后对这个字符进行分析。这里本人运用到了正则表达式进行配对。并且由于配对过程用文字描述过于繁杂，因此这边使用一张图表进行描述



C语言编译器 词法分析DFA

在进入标识符与关键字分支之后，会声明一个 StringBuffer 变量，随后只要后续一直是字符或者数字，就会一直 append 到 StringBuffer 变量中，最终

就形成了一个最终的标识符/关键字。随后只要将该 StringBuffer 的内容在先前设置好的关键字 Hashtable 中进行查找即可，如果查找成功，就以查找到的关键字 token 进行返回，否则以标识符 Id 形式的 token 进行返回

进入数字识别模块后，首先进入的是整型数分支，往后不断扫描。如果都是数字则继续进行，如果扫描到小数点则进入浮点数分支。直到末尾后分别按照当前到达的 token 进行返回。

同时还设置了对部分逻辑运算符和关系运算符的词法识别，特别说明的是，如果违背了这里面的任何一项规则，都会被视为错误，会进行报错

● 语法分析

在完成了大部分词法分析的工作后，接下来最关键的就是语法分析的设计。因为我们使用的是递归下降分析方法，因此如何对文法进行设计，又如何将设计好的文法转化为功能函数，是实现语法分析功能的关键

在先前词法分析中识别到各类关键字后将会返回一个 token，而语法分析器正是凭借着这个 token 从而明白接下来要对应做什么

◆ 代码块 block

首先引入的是代码块 block 的概念，这里的代码块指的是由一个 “{” 和一个 “}” 组成的若干行代码的集合，因此文法可以设计成代码块识别 “{” 后引入一个非终结符代表代码块的内容，我们这边这个非终结符设置为 B

随后我们对代码块内的内容进行研究。一个完整的程序，如果要用到变量，肯定要在开头进行定义，本人将这些定义语句视为定义代码块 define，引入新非终结符 D_1 ，随后就是各类语句，这里将这类不确定类型的需要后续继续扫描的语句称为剩余语句 stmts，引入新非终结符 S_1 表示，因此目前的文法可用

表示如下

$$\begin{aligned} S &\rightarrow \{B\} \\ B &\rightarrow D_1 S_1 \mid \varepsilon \end{aligned}$$

◆定义代码块 define

因为本编译器中实现的有单纯的变量以及数组，因此需要完成的就是分别对变量和数组定义过程即可，将定义语句的格式规定如下

```
int value;      char value;    float value;    bool value;

int [N] array;

char [N][N] array;
```

相应的文法也可以很快拟定出来，因为语句第一位肯定是类型关键字，而后面则是标识符或者数组的定义，接上一段文法中的非终结符 D_1 (代表定义代码块)，首先就是引入一个新的非终结符代表类型关键字(BASIC)，我们使用 B_1 代表，随后另一个非终结符也被引入，即 I ，用于代表所定义的标识符。最终还需要定义数组的定义，引入另外一个非终结符 D_2 代表数组定义，而数组的定义就是在普遍的变量定义上加上一组中括号，外加上面的数字(代表数组组数)，所获得的文法如下所示

$$\begin{aligned} D_1 &\rightarrow B_1 D_2 I \mid B_1 I \\ B_1 &\rightarrow \text{int} \mid \text{bool} \mid \text{char} \mid \text{float} \\ D_2 &\rightarrow [N] D_2 \mid \varepsilon \end{aligned}$$

◆剩余语句 stmts

随着文法的进一步定义，语法解析也越来越清晰，接下来就是对这个剩余语句 stmts 进行分析。实际上这个剩余语句中不仅可能包含更多语句，还可能有包

含新的代码块 block,而这些语句中有可能是各类分支语句或者赋值语句, 例如 do-while,if-else 语句等, 而这些分支语句也有着各自对应的文法, 引入一个非终结符 S_2 代表这一系列分支语句和赋值语句

①for 语句

C 语言中 for 语言的格式如下(范例)

```
for(i=0 ; i<N ; i=i+1)
{
}
```

从规范中可以看出, 括号内的内容可以表示为三部分, 而第一个和第三个实际上可以视作普通的语句, 而第二个是一个典型的布尔表达式。我们引入一个新的非终结符 B_2 用于表示表达式, 而表达式中便包含布尔表达式 (在后面的内容会讲到), 这样就成功获得了 for 语句的文法

$$F \rightarrow \text{for}(S_2; B_2; S_2)B$$

在 for 语句之后是一个 block 对 for 语句内内容进行配对

②if 语句, if-else 语句, if-else if-else 语句

C 语言中的 if 语言格式如下(范例)

```
if(i==0){}
else if(i==5){}
else{}
```

因此在设计 if 系列语句的时候要考虑一系列内容, 因为一个 if 语句内可能包含其他 if 语句或者 else if 语句, 这里引入一个新非终结符 E 用于表示 else, 以避免文法大小写混合引起的歧义, 而在每个语句背后都有一个 stmt 用于匹配大括

号内的语句内容。这里使用 I_1 非终结符引出 if 的文法

$$\begin{aligned} I_1 &\rightarrow \text{if}(B_2)S_2 \mid \text{if}(B_2)S_2ES_2 \\ E &\rightarrow \text{else} \end{aligned}$$

③while 语句 与 do-while 语句

C 语言中的 while 和 do-while 语言格式如下(范例)

```
while(i>0){  
do{  
}while(i<0)
```

在 while 关键字之后紧跟着的就是小括号以及里面的布尔表达式，因此 while 语句可以使用以下文法规则定义（为了 do-while 语句中的 while 不会引起歧义，这里将 while 定义为非终结符 W_1 ）

$$W \rightarrow W_1(B_2)S_2$$

do-while 语句中的 do 关键字和 while 同理为了不引起歧义，do 关键字后跟的是一个代码块，随后跟随的是 while 关键字以及括号中的布尔表达式，因为道理相同所以不多加赘述，以非终结符 D_3 引出 do-while 语句的文法

$$D_3 \rightarrow D_4S_2W_1(B_2)$$

最终综合起来的所有 while 类型文法如下

$$\begin{aligned} W &\rightarrow W_1(B_2)S_2 \\ D_3 &\rightarrow D_4S_2W_1(B_2) \\ W_1 &\rightarrow \text{while} \\ D_4 &\rightarrow \text{do} \end{aligned}$$

④break 语句

break 语句相对来说比较简单，因为 break 的语句只有

break;

因此文法实现比较简单，这里以 B_3 作为 break 文法插入口

$$B_3 \rightarrow break;$$

⑤赋值语句

待标识符声明之后，是需要进行赋值的，这里分为变量的赋值和数组元素的赋值

范例: $a=4;$ $array[2][4] = 43;$

按照两种语句的格式规律来说，变量赋值语句在匹配到表式符后，紧接着就会匹配到 "=", 因此以这个作为分辨两种赋值语句的依据，倘若识别的是不是 "=", 则自动视为赋值的是数组。进入数组赋值模块，而考虑到数组可能有多维的情况，因此也设置了一个循环判别，以 "[" 作为切入口，这样的话文法就已经很清晰了，我们引入若干个非终结符，A 代表赋值语句块， A_1 代表数组赋值模块，因此所得文法如下

$$\begin{aligned} A &\rightarrow IA_1 = B_2 \\ A_1 &\rightarrow [B_2]A_1 \mid \varepsilon \end{aligned}$$

至此，所有控制语句和赋值语句的文法已经设计完成，汇总文法如下

$$\begin{aligned}
S &\rightarrow \{B\} \\
B &\rightarrow D_1 S_1 \mid \varepsilon \\
D_1 &\rightarrow B_1 D_2 I \mid B_1 I \\
B_1 &\rightarrow \text{int} \mid \text{bool} \mid \text{char} \mid \text{float} \\
D_2 &\rightarrow [N] D_2 \mid \varepsilon \\
S_1 &\rightarrow F \mid I_1 \mid W \mid D_3 \mid B_3 \mid A \mid \varepsilon \\
F &\rightarrow \text{for}(S_2; B_2; S_2) B \\
I_1 &\rightarrow \text{if}(B_2) S_2 \mid \text{if}(B_2) S_2 E S_2 \\
E &\rightarrow \text{else} \\
W &\rightarrow W_1 (B_2) S_2 \\
D_3 &\rightarrow D_4 S_2 W_1 (B_2) \\
W_1 &\rightarrow \text{while} \\
D_4 &\rightarrow \text{do} \\
B_3 &\rightarrow \text{break}; \\
A &\rightarrow I A_1 = B_2 \\
A_1 &\rightarrow [B_2] A_1 \mid \varepsilon
\end{aligned}$$

S : 起始非终结符
 B : 代码块block
 D_1 : 定义代码块
 S_1 : 剩余语句
 I : 标识符
 B_1 : 类型关键字
 D_2 : 数组(定义)
 N : 数字
 F : for语句块
 I_1 : if语句块
 W : while语句块
 D_3 : do – while语句块
 B_3 : break语句
 A : 赋值语句块
 B_2 : 表达式
 A_1 : 数组赋值模块

◆表达式

程序中除了各类语句之外，还存在着各种各样的表达式。这些表达式可以用于赋值，或者用在控制语句的条件判定中

诸如 “a>b” , “!b” , “stmt1 && stmt2” 等等的语句都可以视为表达式

①或运算表达式：该表达式完成的是一个或运算，运算符号为 ||

②与运算表达式：该表达式完成的是一个与运算，运算符号为 &&

③等值/不等值表达式：该表达式包含符号 “==” 或者 “!=” ,用于确定运算符两侧的内容是否相等或者不等

④比较表达式：包含符号 “>” , “<” , “>=” , “<=” ,专门用于比较两侧内容的大小

⑤加减法表达式：该表达式完成加减法运算，包含 “+” 与 “-”

⑥乘除法表达式：该表达式完成乘除法运算，包含 “*” 和 “/”

⑦单目运算表达式：该表达式完成的是一个单目运算，因为该运算只需要一个操作数，比如非运算和自减运算

①其他表达式：表达式也是可以进行嵌套的，并且一个数，或者是单纯的 true,false 也可以是表达式，因为表达式内容过多，但是规律简单，完整阐述会占用大量不必要的篇幅，因此这里只列举其中一个文法的设计过程

范例：加减法表达式

首先 “+” 和 “-” 可以首先提出来，这是十分明显的，剩下只需要对两侧的操作数进行分析，因为操作数可以是数，可以是 true/false,也可以是另外一个表达式，因此我们定义一个非终结符 F_2 用来表示这类操作数，称为 factor，因此文法就可以很顺利的设计出来

$$\begin{aligned} B_2 &\rightarrow F_2 + F_2 \mid F_2 - F_2 \\ F_2 &\rightarrow (B_2) \mid N \mid true \mid false \mid I \mid A_1 \end{aligned}$$

按照这个规则，剩下其他表达式的文法就应运而生了，最终生成的表达式完整文法如下

$$\begin{aligned} B_2 &\rightarrow F_2 + F_2 \mid F_2 - F_2 \mid F_2 * F_2 \mid F_2 / F_2 \mid F_2 \& \& F_2 \mid F_2 \parallel F_2 \mid ! F_2 \mid -F_2 \\ F_2 &\rightarrow (B_2) \mid N \mid true \mid false \mid I \mid A_1 \end{aligned}$$

◆汇总与集成

将所有文法设计好之后，就可以根据文法设计对应的功能函数了，并且各个函数之间可以互相调用。扫描不断进行。因为使用的是递归下降分析法，因此函数递归调用是非常普遍的，功能函数由于行数较多，如想要知道如何实现，可以查看源文件 Parser.java

◆错误处理

在语法分析过程中可能会出现错误,平常在 IDE 写代码的时候编译器也经常会报错,因此报错功能对于编译器同样很重要。在本编译器中,本人通过编写一个 error 函数来对这些错误进行一个报告。error 函数实质上是一个输出/报错函数,它将收集到的错误信息(通过函数参数引入)通过控制台,GUI 控制台或者抛出错误的方式来报告错误

目前本编译器实现的报错类型共有以下若干种

①变量未定义错误:如果一个变量未经过定义就使用,那么就会报错。报错格式:“未定义变量” undeclared

②语法错误:这是最常见的一个错误,往往是程序语句不符合编译器文法定义引起的语法分析错误,格式:syntax error

并且先前词法分析器中记录的行数这里也起到了作用,可以精确定位到是第几行代码出现了错误,格式:near line x

这样就可以快速定位错误代码,并让 coder 能及时修改代码

至此,语法分析的大致内容就全部介绍完毕,接下来将介绍最后一个大步骤,语义分析与中间代码产生,是构建抽象语法树,三地址代码和四元式的重要过程

●语义分析与中间代码产生

语义分析是编译过程的一个逻辑阶段,主要是对通过语法分析的源程序进行上下文有关性质的筛查,审查源程序有没有语义错误。该阶段是编译器最实质性的工作,因为其第一次对源程序的语义作出解释,并且在过程中生成中间代码,中间代码的好处主要是方便移植和方便目标代码产生。

本编译器主要采用语法制导翻译,通过生成抽象语法树,产生三地址代码和四元式,在四元式的生成过程中还用到了拉链回填等技术

◆抽象语法树节点 Node 定义

抽象语法树中的每一个节点都是 Node,而不同类型的节点又要具备不同的属性,因此该 Node 类只是一个父类,后边还会有很多子类继承 Node 类并增添更多属性与方法

Node 类的构造函数中只有一个参数,就是当前节点所在代码行数,方便报错函数报告错误发生于第几行

◆表达式节点 Expr, 语句节点 Stmt 定义

Expr 类用于代表一系列表达式节点,该类继承 Node 类,并且在其基础上增添了几个属性,分别是运算符 op(token 类型)和运算类型 type(Type 类型),语句节点也是一种类型的节点,通常被用于表示一个语句,其子类便是各类控制语句,诸如 while, if, for 等等,相比于父类 Node 其增加了 after 和 Enclosing 等属性,after 用于存储下一条指令的标号,而 Enclosing 则用于 break 语句

◎ gen 函数与 reduce 函数

本函数在一定程度上参考了龙书《编译原理》的思路

gen 函数返回一个参数,该参数在未来三地址代码的构造中可作为其右部,并且可以在其子类中被多次改写

reduce 函数将一个表达式规约成为一个单一的参数,返回一个常量,标识符或者临时变量(Temp),该函数对于 Op 类的子类(表示单目运算符的子类 Unary 等)有着十分重要的作用,并且都会被重写

例: $a = b * c$

按照四元式(在后面会讲到)来说,并不是单纯的生成 $(*, b, c, a)$ 就行了

首先要将 $b*c$ 的运算结果存放在一个临时变量 t1 中

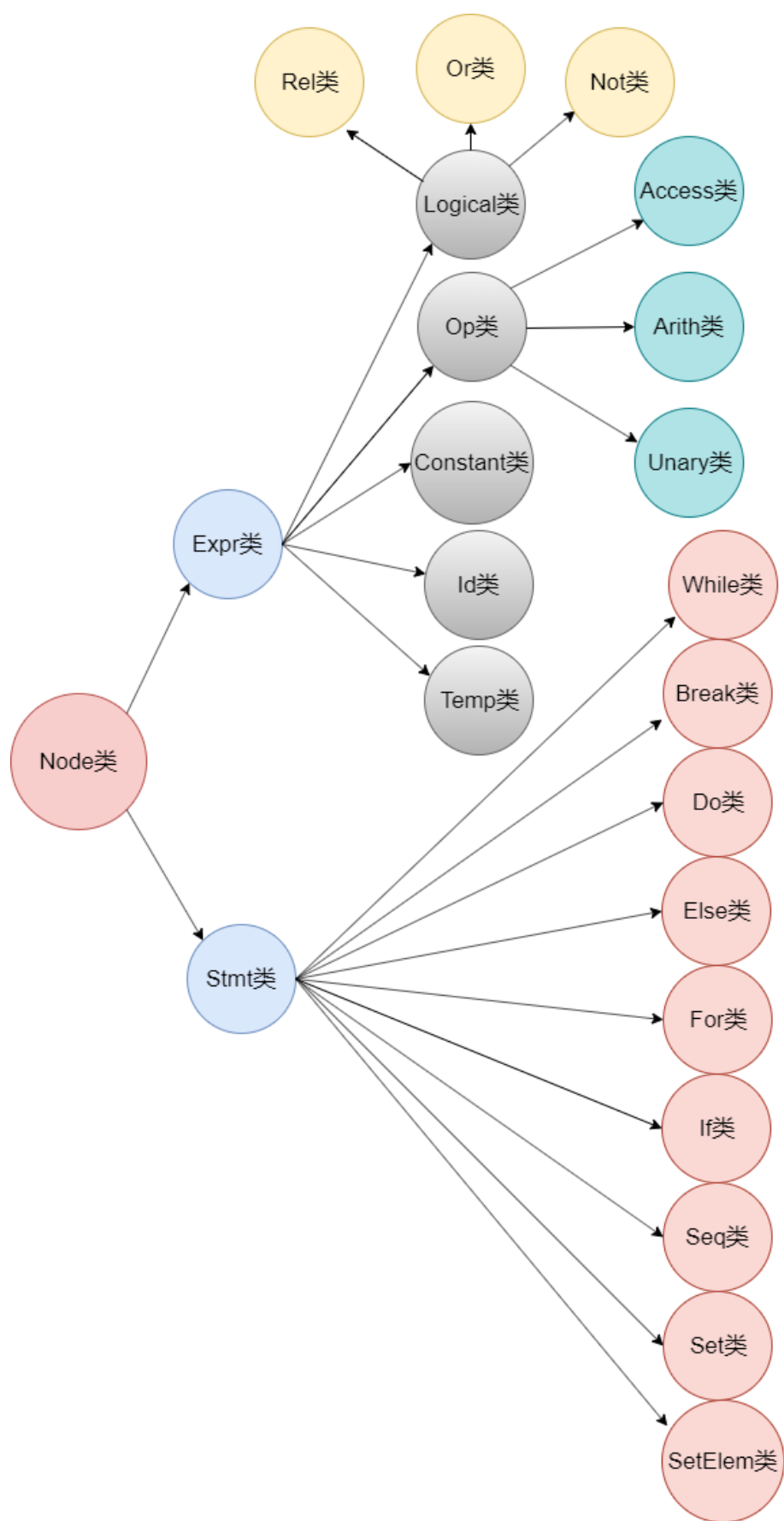
即(*, b, c, t1),然后才能将该值传给目标变量, (:= , t1, -, a)

◎ transforToString 函数

是一个信息输出/展示函数, 将当前得到的部分信息转换成字符串, 是后面生成三地址代码的重要功能函数

◆三地址代码

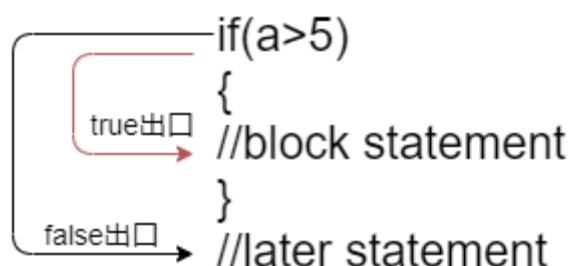
三地址码是本编译器要实现的第一种中间语言, 编译器通过它可以改进代码的转换效率, 在上述描述的 gen,reduce 和 transforToString 函数下, 根据父类 Expr 和 Stmt 的定义, 在此基础上再定义其他用于生成三地址代码的具体子类, 基本定义如下图所示



C语言编译器 语义分析各功能类

◎ jumping 函数

因为语言中包含各类控制语句，因此就涉及到各种跳跃与标号的保存
诸如 while, if 等语句都包含条件判定，根据条件中的布尔表达式的真假从而决定是否进入代码块。因此这就涉及到一个概念，就是真(true)出口和假(false)出口



因此引入函数 jumping,也就是跳跃函数，其包含两个参数，分别是标号 t 和标号 f,标号 t 代表着 true 出口的标号，f 代表 false 出口的标号，该 jumping 函数在不同的控制语句中被不同程度的调用，而在最原始的 jumping 函数中还调用了另外一个函数 emitjumps，该函数与 emits 函数一同完成了三地址代码的产生

◎ emitjumps 与 emit 函数

上述两个函数用于三地址代码的生成，emitjumps 用于判断包含跳转标号的 goto L 语句生成，因为标号大于 0 的缘故，因此如果 true 出口或者 false 出口为 0 的话说明 true 出口或者 false 出口不存在，这里将这两种标号的存在情况分为三种，这里设置 t 为 true 出口标号，f 为 false 出口标号

①true 出口标号和 false 出口标号同时存在 ,按照下列格式进行输出

ifA Expr goto L t

goto L f

②只有 true 出口标号存在 ,按照下列格式进行输出

ifB Expr goto L t

②只有 false 出口标号存在 ,按照下列格式进行输出

iffalse Expr goto L f

其中②, ③相对较常用, ②广泛用于 while,for,if 语句中, ③主要用于 do-while 语句

范例: 将一个嵌套有 if-else if-else 语句的 for 语句翻译成三地址代码 (标号从 7 开始)

```
for(a=2;a<5;a=a+1)
{
a=a+1;
if(a<5)
{
break;
}
else if(a>5)
{
a=2;
}
else{
a=4;}}
```



```
L7: a=2
    iffalse a < 5 goto L2
L9: a=a + 1
L10: iffalse a < 5 goto L12
L11: goto L0
    goto L7
L12: iffalse a > 5 goto L14
L13: a=2
L15: goto L7
L14: a=4
L16: a=a + 1
    goto L7
L2:
```

而 emit 函数主要用于对那些普通的语句进行三地址代码的产生，相对而言处理起来就没有这么麻烦

emit 函数传入一个字符串型参数，这个参数会经过多次调用。由多个分散的字符串逐渐组合成一个完整的字符串，比如表达式 $a+b$ ，编译器将其视作两个操作数和一个算符。这是因为编译器不知道这两个操作数是标识符还是另一个表达式，因此通过这种递归调用（抽象语法树）可以使得编译器最大程度的确保翻译的正确性

范例：复杂表达式的翻译

$$a = a+b-c/e+(b+a*e-b);$$

可被翻译为以下三地址代码

L1: $t1 = a + b$

$t2 = c / e$

$t3 = t1 - t2$

$t4 = a * e$

$t5 = b + t4$

$t6 = t5 - b$

$a = t3 + t6$

L2:

通过对临时变量的使用，从而很好表达了这个复杂的表达式

三地址代码的产生原理基本就介绍到这里，这是一种很好的中间语言，对后期目

标代码产生有着很大的作用，下面还会介绍另外一种中间语言四元式的构造

◆四元式

- 四元式也是一种中间语言，其格式通常为

(OP, arg1, arg2, result)

实际上完成了对三地址代码的构造后，四元式的构造原理也基本相似，本人在后期完成四元式构建后四元式和三地址代码的产生是写在同一批函数中的，但是虽然原理有相似之处，但是四元式相比三地址代码在很多地方还是有较大差异的，尤其是在标号和部分回填方面相比三地址代码要实现的难度大得多

◎ 四元式数据结构构建

为了更好地对四元式进行实现，本人将四元式语句都封装成了一个数据结构，这样方便后期语句中各个参数和方法的调用

```
class quadruple
{
    quadruple(String o,String a1,String a2,String r)
    quadruple(){ }

    String op;

    String arg1;

    String arg2;

    String result;

}
```

其中包含两个构造函数用于不同实例化构造的使用（带参和不带参），以及四元式的四个基本属性

◎ 四元式标号

四元式和三地址代码最大的区别之一就是在标号处理方面，对于相同的一段代码，翻译成四元式和三地址代码标号往往截然不同，比如之前拿出来讨论的那个复杂表达式 $a = a + b - c / e + (b + a * e - b)$ ；如果是三地址代码的话就只需要消耗两个标号，然而如果是四元式的话，就要消耗多达七个标号

```
L1: t1 = a + b
```

```
t2 = c / e
```

```
t3 = t1 - t2
```

```
t4 = a * e
```

```
t5 = b + t4
```

```
t6 = t5 - b
```

```
a = t3 + t6
```

```
L2:
```

三地址代码

```
0 : +,a,b,t1
```

```
1 : /,c,e,t2
```

```
2 : -,t1,t2,t3
```

```
3 : *,a,e,t4
```

```
4 : +,b,t4,t5
```

```
5 : -,t5,b,t6
```

```
6 : +,t3,t6,a
```

```
7 :
```

四元式

这就要求在四元式构造时对标号的精细处理，因为任何一个差错都可能导致四元式产生的完全错误，因此，与使用 emit 和 emitjumps 的三地址代码不同，四元式的产生通过离散的实例化语句进行，这样的优点就是可以在任何需要的地方进行四元式产生，比方重在 while 语句中的跳转四元式

```
quad[quadNum++] = new quadruple("j", "-", "-", Integer.toString(sta));
```

◎ 拉链回填和真假链

本部分是四元式构造中最为关键的一部分，因为除了普通的语句生成之外，还有众多的布尔表达式和控制语句要进行生成。而现在最关键的问题就是，四元式不能像三地址代码那样遇到控制语句就扔个标号出去等着接 true 出口/假出口，因为后面还有多少代码在编译器看来是完全未知的，而四元式就是要将每一个单元都设置标号（详情可见上一页的三地址代码和四元式比较框图），因此如何确定这个跳转的标号就是我们所要解决的问题，这里引入拉链回填的概念

拉链回填的主要思路就是将一个由跳转指令组成的列表以综合属性的形式进行传递，也就是这些指令暂时先不指定跳转的目标，而是保存在一个列表中，等到能够确定正确的目标标号时才进行填充操作，本人在此次回填操作中设置三个列表，分别为真链 truelist,假链 falselist 和紧跟链 nextlist,这三个节点作为综合属性定义给每一个节点，使得它们能在翻译的过程中得以传递或者操作

以 while 语句作为范例，使用以下规则用于 while 的翻译和回填

$S \rightarrow \text{while } M_1(B)M_2S_1$	$\text{backpatch}(S_1.\text{nextlist}, M_1.\text{instr})$ $\text{backpatch}(B.\text{truelist}, M_2.\text{instr});$ $S.\text{nextlist} = B.\text{nextlist};$
--	---

在这个规则中，通过 backpatch 这个函数将 B 的 truelist 和 S_1 的 nextlist 分别回填为 M_1 和 M_2 的标号，同时使得 S 的 nextlist 等于 B 的 nextlist

在这个过程中的两个 M 代表着不同的非终结符，事实上每条控制语句都有着对应的翻译规则，这里部分规则参考了龙书上的语句翻译规则，因此过多规则不再陈述，下面使用一个范例来解释拉链回填的实际运用

本次距离仍然使用先前的嵌套 if-else if-else 语句的 for 语句，如果不使用拉链回填的话，经过常规的四元式翻译过程后，四元式如右侧所示（标号从 11 开始）

```
for(a=2;a<5;a=a+1)
```

```
{
```

```
  a=a+1;
```

```
  if(a<5)
```

```
  {
```

```
    break;
```

```
  }
```

```
  else if(a>5)
```

```
  {
```

```
    a=2;
```

```
  }
```

```
  else{
```

```
    a=4;}}
```

```
11 : :=,2,-,a
```

```
12 : j<,a,5,null
```

```
13 : j,-,-,26
```

```
14 : +,a,1,a
```

```
15 : j<,a,5, null
```

```
16 : j,-,-, null
```

```
17 : j,-,-, null
```

```
18 : jeee,-,-, null
```

```
19 : j>,a,5, null
```

```
20 : j,-,-, null
```

```
21 : :=,2,-,a
```

```
22 : jeee,-,-, null
```

```
23 : :=,4,-,a
```

```
24 : +,a,1,a
```

```
25 : j,-,-, null
```

```
26 :
```

因此在复杂的控制语句中，如果没有
拉链回填，四元式的翻译效果相当糟糕

因为一大部分的跳转四元式都会缺乏目标标号，这对后面目标代码生成影响极大

①在翻译进行到标号 12 的四元式时，这时候对应的语句是一个 $A \text{ rel } B$ 的句型，

这时候构造 B 的 truelist,falselist,分别等于当前标号+1 和当前标号+2, 因为紧接着的下一个四元式是一个无条件跳转四元式, 该四元式是该布尔表达式的真出口, 紧接着还有一个无条件跳转四元式, 代表假出口

②翻译到第一个 if 语句时, 操作和上述类似, 需要注意的是因为该 if 内包含一个 break 语句, 因此还需要额外增添一个无条件跳转语句(跳转到 for 循环外面), 因此该标号也一并被添加到 B 的 falselist 中

③针对 else 语句有特别的四元式语句 jeee,根据对 if-else 语句的翻译以及处理(规则如下)

$S \rightarrow \text{if } (B) M \ S_1$	$\text{Backpatch}(B.\text{truelist}, M.\text{instr})$ $S.\text{nextlist} = \text{merge}(B.\text{falselist}, S_1.\text{nextlist});$
$S \rightarrow \text{if } (B) M \ S_1 \text{Nelse} M_2 S_2$	$\text{backpatch}(B.\text{truelist}, M_1.\text{instr})$ $\text{backpatch}(B.\text{truelist}, M_2.\text{instr});$ $\text{temp} = \text{merge}(S_1.\text{nextlist}, N.\text{nextlist});$ $S.\text{nextlist} = \text{merge}(\text{temp}, S_2.\text{nextlist});$

其中的 merge 函数用于将两个列表进行合并, 将结果作为参数进行返回

经过一系列的 backpatch 之后, 所有的目标标号就都回填完毕, 这时候所得到的就是完整正确的四元式代码

该部分可以说是本人在制作这个编译器时工作量最大的部分之一了, 因为想要将该四元式的生成和三地址代码的生成代码结合在一起还着实有点小麻烦。不过在经历了无数次调试以及测验之后, 还是成功的完成了该部分的设计

回填完之后的完整四元式如下图所示

```
11 : :=,2,-,a
12 : j<,a,5,14
13 : j,-,-,26
14 : +,a,1,a
15 : j<,a,5,17
16 : j,-,-,19
17 : j,-,-,26
18 : jeee,-,-,24
19 : j>,a,5,21
20 : j,-,-,23
21 : :=,2,-,a
22 : jeee,-,-,24
23 : :=,4,-,a
24 : +,a,1,a
25 : j,-,-,12
26 :
```

至此，四元式构造的核心部分和问题已经全部阐述完毕，本人认为四元式虽然构造相比于三地址代码更为复杂，但是对于目标代码产生来说，四元式比三地址代码更好

◆ 错误处理

①未闭合 break 错误: 如果 break 语句出现在了非循环体的代码块内, 将会报此类错误 (因为 break 一般搭配循环体), 报错格式: unclosed break

②类型错误: 如果运算时两个操作数类型不一致或者是类型不匹配, 则报错
报错格式: type error

③分支语句条件错误: 分支语句条件一般要求都是布尔表达式, 如果检测出不是布尔表达式则会报错, 报错格式: Boolean required in while/if/do

至此, 本编译器的全部核心代码和问题全部阐述完毕

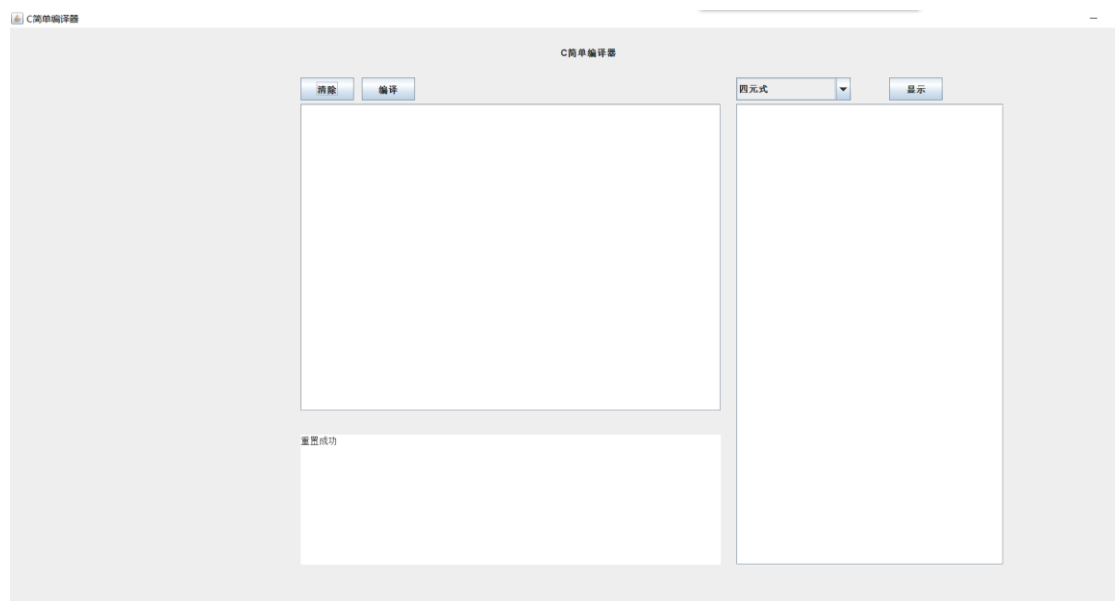
四、设计的输入和输出形式

设计输入: 简单的 C 语言程序段

设计输出: 经过编译后生成的三地址代码和四元式

五、程序运行（测试、模拟）的结果（屏幕拷贝、生成结果的打印输出）

①项目 UI 如下图所示，在左上方的大框框里输入需要编译的 C 语言程序，左下方模拟的是控制台输出，右侧的框框用于输出四元式和三地址代码，并且有下拉条可以进行选择

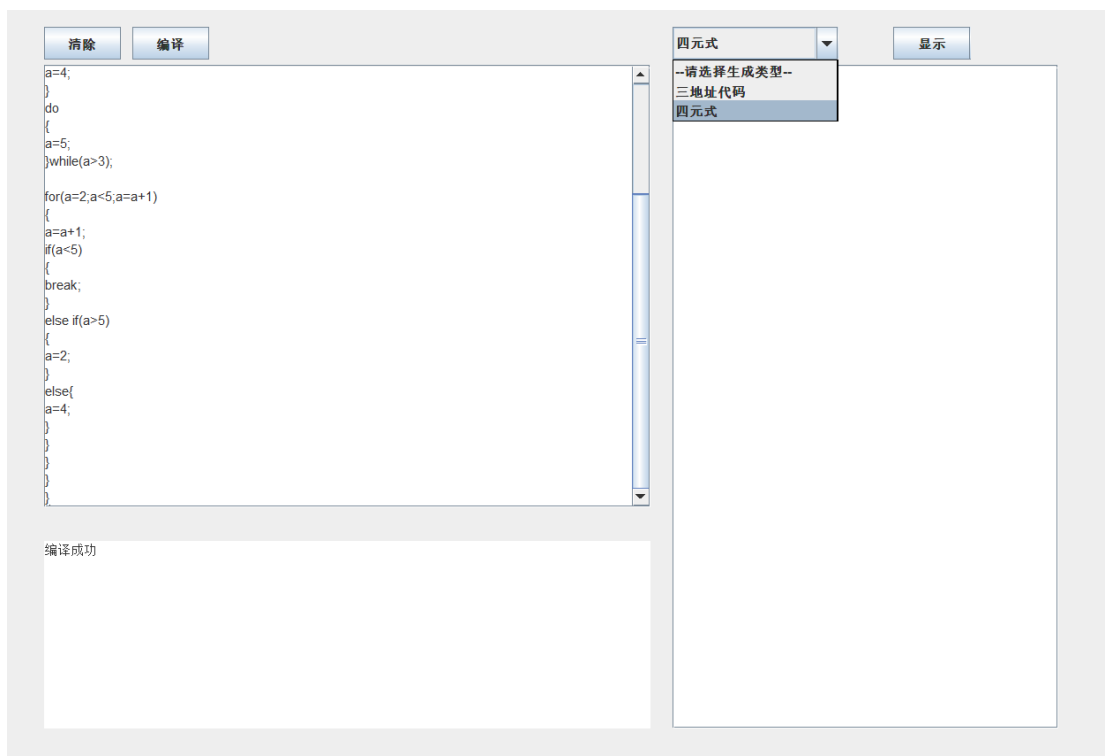


②首先进行控制语句测试，使用的语句如下（因为程序过长，故使用 IntelliJ 风格）

```
{  
    int a;  
    int b;  
    while (a > 0)  
    {  
        a = 1;  
    }  
    if (a < 2)  
    {  
        a = 4;  
    }  
    do  
    {  
        a = 5;  
    } while (a > 3);  
}
```

```
for (a = 2; a < 5; a = a + 1)
{
a = a + 1;
if (a < 5)
{
break;
}
else if (a > 5)
{
a = 2;
}
else {
a = 4;
}
}}
```

③点击“编译”按钮后，伪控制台显示“编译成功”，并且可以在右侧进行选择



④选择“三地址代码”选项，点击“显示”按钮，输出三地址代码

三地址代码

▼

显示

L1:

iffalse a > 0 goto L3

L4:

a=1

goto L1

L3:

iffalse a < 2 goto L5

L6:

a=4

L5:

a=5

L8:

ifB a > 3 goto L5

L7:

a=2

iffalse a < 5 goto L2

L9:

a=a + 1

L10:

iffalse a < 5 goto L12

L11:

goto L0

goto L7

L12:

iffalse a > 5 goto L14

L13:

a=2

L15:

goto L7

L14:

a=4

L16:

a=a + 1

goto L7

L2:

⑤选择“四元式”选项，点击“显示”按钮，输出四元式

四元式

▼

显示

0 : j>,a,0,2
1 : j,-,-,4
2 : :=,1,-,a
3 : j,-,-,0
4 : j<,a,2,6
5 : j,-,-,7
6 : :=,4,-,a
7 : j,-,-,8
8 : :=,5,-,a
9 : j>,a,3,7
10 : j,-,-,11
11 : :=,2,-,a
12 : j<,a,5,14
13 : j,-,-,26
14 : +,a,1,a
15 : j<,a,5,17
16 : j,-,-,19
17 : j,-,-,26
18 : jeee,-,-,24
19 : j>,a,5,21
20 : j,-,-,23
21 : :=,2,-,a
22 : jeee,-,-,24
23 : :=,4,-,a
24 : +,a,1,a
25 : j,-,-,12
26 :

⑥接下来测试复杂布尔表达式，结合 if-else 语句一同进行

```
{  
    int a;  
    int b;  
    int c;  
    a=2; b=4; c=43;  
    if (a<b&& a==b&& a<=b&& a>=b&& a>b)  
    {  
        c=7;  
    }  
    else {  
        c=5;  
    }  
}
```

```
}  
}  
}
```

⑦其他过多操作不再重复展示，选择“三地址代码”选项，点击“显示”按钮，输出三地址代码

三地址代码		显示
L1:	a=2	
L3:	b=4	
L4:	c=43	
L5:	iffalse a < b goto L7	
	iffalse a == b goto L7	
	iffalse a <= b goto L7	
	iffalse a >= b goto L7	
	iffalse a > b goto L7	
L6:	c=7	
	goto L2	
L7:	c=5	
L2:		

⑧选择“四元式”选项，点击“显示”按钮，输出四元式

四元式		显示
0	::=,2,-,a	
1	::=,4,-,b	
2	::=,43,-,c	
3	:j<,a,b,5	
4	:j,-, -,15	
5	:j==,a,b,7	
6	:j,-, -,15	
7	:j<=,a,b,9	
8	:j,-, -,15	
9	:j>=,a,b,11	
10	:j,-, -,15	
11	:j>,a,b,13	
12	:j,-, -,15	
13	::=,7,-,c	
14	:jeeee,-, -,16	
15	::=,5,-,c	
16	:	

⑨最后对实际运用的代码进行测试，这里选择测试冒泡排序代码

```
{  
    int i;  
    int j;  
    int g;  
    int n;  
    int temp;  
    int [15] arr;  
    n=10;  
    g=53;  
    for(i=0;i<n-1;i=i+1)  
    {  
        for(j=i;j<n;j=j+1)  
        {  
            if(arr[i]>arr[j])  
            {  
                temp = arr[i];  
                arr[i] = arr[j];  
                arr[j] = temp;  
            }  
        }  
    }  
}
```

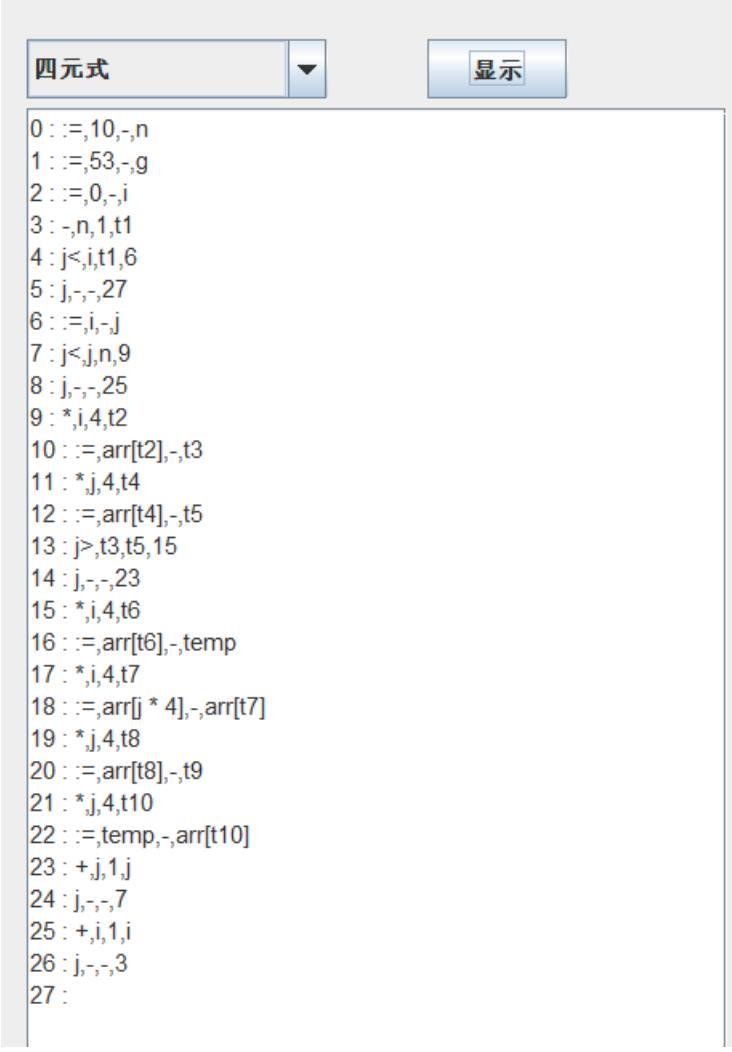
⑩选择“三地址代码”选项，点击“显示”按钮，输出三地址代码

三地址代码

显示

L1:	n=10
L3:	g=53
L4:	i=0
	t1 = n - 1
	iffalse i < t1 goto L2
L5:	j=i
	iffalse j < n goto L4
L6:	t2 = i * 4
	t3 = arr[t2]
	t4 = j * 4
	t5 = arr[t4]
	iffalse t3 > t5 goto L5
L7:	t6 = i * 4
	temp=arr[t6]
L8:	t7 = i * 4
	t8 = j * 4
	t9 = arr[t8]
	arr[t7] =t9
L9:	t10 = j * 4
	arr[t10] =temp
L10:	j=j + 1
	goto L5
	i=i + 1
	goto L4
L2:	

①①选择“四元式”选项，点击“显示”按钮，输出四元式



```
0 :=, 10, -, n
1 :=, 53, -, g
2 :=, 0, -, i
3 :=, n, 1, t1
4 : j<, i, t1, 6
5 : j, -, -, 27
6 :=, i, -, j
7 : j<, j, n, 9
8 : j, -, -, 25
9 : *, i, 4, t2
10 :=, arr[t2], -, t3
11 : *, j, 4, t4
12 :=, arr[t4], -, t5
13 : j>, t3, t5, 15
14 : j, -, -, 23
15 : *, i, 4, t6
16 :=, arr[t6], -, temp
17 : *, i, 4, t7
18 :=, arr[j * 4], -, arr[t7]
19 : *, j, 4, t8
20 :=, arr[t8], -, t9
21 : *, j, 4, t10
22 :=, temp, -, arr[t10]
23 : +, j, 1, j
24 : j, -, -, 7
25 : +, i, 1, i
26 : j, -, -, 3
27 :
```

至此，演示全部顺利结束

六、总结（体会）

本次项目可以说是本人对所学过的编译原理的知识的一次大总结，从词法分析，经过语法分析，语义分析到最后中间代码产生。参考了无数教材和网上的内容，包括刘春林等人著的《编译原理》，和编译巨作龙书《编译原理》，并且也在CSDN，stack-overflow, github 等网站上查阅了很多资料，获取了很多内容

中间遇到了很多很多问题，有时候解决一个小问题甚至要到凌晨才能解决，解决不了也睡不着，但是最后还是通过自己的努力写出了这么个成品，虽然目标

代码生成来不及实现，但是本人也十分满足了，以前也听过别人说编译原理课写自己的编译器，如果不算上目标代码生成的话，我也算是拥有了自己的一个编译器了。

也相信自己寒假-暑假努力锻炼出来的工程和代码能力能对以后更多大项目的实现乃至工作有帮助

七、源程序清单（部分核心代码）作为报告的附件。

路径下的 src 文件夹包含两个文件夹

①其中的 CParser 文件夹包含目前项目的所有源代码，因为文件过多，因此不一一阐述

①Server 文件夹包含待开发的网页应用型代码（暂时没有用）