《编译原理期末课程设计》

实验报告

学 院 名 称 ：	计算机学院
专业（班级）：	计算机科学与技术（人工智能+大数据）
学 生 姓 名 ：	何鸿荣
学 号 ：	19335052
时 间 ：	2022 年 7 月 4 日
Github：	(hhr247/compile (github.com))

成 绩 ：

实验题目

一个简单文法的编译器前段的设计与实现

- 定义一个简单程序设计语言文法（包括变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、If语句、While语句等）
- 扫描器设计实现；
- 符号表系统的设计实现；
- 语法分析器设计实现；
- 中间代码设计；
- 中间代码生成器设计实现。

一个简单文法的编译器后段的设计与实现

- 中间代码的优化设计与实现
- 目标代码的生成（使用汇编语言描述，指令集自选）；
- 目标代码的成功运行。

实验目的

通过编译器相关子系统的设计，进一步加深对编译器构造的理解；

培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力；

提高程序设计能力、程序调试能力

设计方案

在本次实验中，主要将其分成4个模块：

1. 符号表
2. 词法分析器
3. 语法/语义分析器
4. 目标代码生成器

这四个模块中，互相关联，词法分析器的输出是语法/语义分析器的输入，而语法/语义分析器又是目标代码生成器的输入。符号表则是贯穿了整个设计的过程。

在符号表中，主要涉及符号表的查询与管理

在词法分析器中，主要构造了能够识别token的自动机，并且将token输出。

在语法/语义分析过程中，主要是语法的构造与识别，符号表内容的填充与中间代码的生成。

在最后的目标代码生成中，主要是将四元式转换成汇编语言的形式。

方案实现

符号系统的设计

在这里，我们同时需要设定符号表的内容。

在符号表中，我主要实现了两个类型的表以及一个总表：

- 第一种是存储变量名的表，名为variable table

    - 在其中，主要记录了该变量的类型，该变量的所需要的空间，变量名。
- 第二种是存储函数名的表，名为function table
    - 在其中，主要记录了函数名，函数返回类型。
- 在总表中，记录了所有的标识符，并且给标识符一个类型，这个类型将会在语义分析中使用到。

在符号表系统中，主要有将目标变量名连接到对应的符号表位置的功能。

符号表的设计如下所示：

```
class Symbol_table {
public:
    Symbol_table();
    ~Symbol_table();
    int add_master(string a);
    bool connect_func(string name, string return_type);
    bool connect_variable( string name, string tval, int len);
    int find(string a);
    int findf(string a);
    int findv(string a);
    string getname(string id);
    void print_table();
    vector<func_table> func;
    vector<constant_table> constant;
    vector<variable_table> variable;
    vector<master_table> symbol;
};
```

在符号表中，主要有函数表，常数表，变量表

以及一个符号总表。

在我实现的部分中，由于只有一个main函数。

因此符号总表能用作为描述main函数的符号表。

词法分析

在词法分析中，我们除了需要识别对应的单词以外，还需要将标识符填入符号表。

但是具体的符号表细节将会在语义识别过程中完成。

在词法分析中，我们实现的自动机能够识别大部分C语言的内容，包括标识符，常数，运算符等。

所有能识别的内容如下所示：

按照c语言语法的设计，首先确定关键字表内容，参照c语言的标准， 一共有30个关键字，将其列表，如下所示：

auto	**union**	**volatile**	**while**
short	typedef	void	goto
int	const	if	continue
long	unsigned	else	break
float	signed	switch	sizeof
double	externed	case	return
char	register	for	
struct	static	do	

确定了关键字表后，还需要确定界符表与运算符表，确定剩余的类型。将实现的界符与运算符如下所示：

{	**}**	**(**	**)**
[	]	/*	*/
//	'	"	<
<=	>	>=	==
!=	=	+	++
+=	-	--	-=
*	*=	/	/=
%	%=	<<	<<=
>>	>>=	&	&=
&&	|	|=	||
^	^=	,	?
:	!	~	;
.			

需要注意的是，在我们的标识符与关键字一致的时候，会优先将其识别为关键字。

当我们将token识别为关键字时，我们需要查找符号表：

- 若是符号表中出现过：将该关键字在符号表中的位置写入到token所对应的val中
- 若是符号表中没有出现过：加入符号表中，并且将该关键字在符号表中的位置写入到token所对应的val中
- 符号表中的内容将会在语义分析中进行。

在词法分析中，需要设计一个自动机来实现字符串的识别。

自动机设计

首先我们设定初始状态为1，结束状态为0，异常状态为-1。

在初始状态中，遇到空白符，如空格符，换行符等，则维持不变，否则将会执行状态转换。

标识符|关键字

在初始状态遇到字母的时候，可以确定此时只有两种情况，要么是标识符，要么是关键字。因此此时可以继续读入字母或者数字，或者是下划线，其余情况都属于终结符。

实数

在初始状态遇到数字的时候，也可以分成多种情况：十进制的数字、八进制数字、二进制数字、十六进制。

除一种情况：0.x的小数 以外，十进制的开头必然是1-9之间的数字。其余进制的开头是0x，0，0b。因此可以确定当前的数字的取值的范围，按照进制的表示数来读取字符，如十进制能够表示的范围只有0-9，八进制表示的范围只有0-7等。

除此之外，十进制的数字还会有多种表示方法，一种是浮点数，一种是整数，还有一种是科学计数法。因此需要判断是否存在‘ . ’，来确定是否属于小数，需要判断是否存在‘ e|E ‘来确定是否属于科学计数法表示。

小数的表示方法包括".1" "0.1" "11.1"等。

最后当我们遇到其他的字符的时候，意味着该状态的终结。这其中的所有的状态都属于实数。

字符|字符串

在初始状态遇到字符或者字符串所对应的字符如 ' 与" 时，进入字符的匹配状态。

字符：

- 在遇到 ’ 的时候，字符只能有一个，也就是说其中最多只能存一个字符。但是有一种情况除外，转义符 \，此时就可以存储两个字符，其中转义符因为其转义特性，并不能算作一个符号。但是在词法分析的过程中，并不需要考虑，只需要将其存储下来。在字符串中，若是已经存在一个字符，则下个字符必须是 ’，否则应该转到-1状态，代表出现异常。遇到 ‘ 字符后，就能够

字符串：

- 在遇到 “ 的时候，字符串可以是任意的字符。当我们遇到转义符的时候，后面是允许添加任意字符，哪怕是 ” 也允许继续视为字符串。
- 最后当我们遇到单独的 “ 的时候，就视为字符串的结束。
- 需要注意的是，需要考虑换行符\n，因为在字符串中，除非是遇到转义符\，否则是不允许换行的。字符串中也不允许直接进行换行。因此遇到\n的时候，会跳转到-1，也就是异常状态。

注释

在初始状态只会在遇到 / 才有可能进入注释，其中注释有两种方式，一种是行注释，另一种是块注释。其中行注释所对应的界符为//，块注释所对应的界符为/* */。

行注释：

- 进入行注释后，可以往后读任意的字符，但是有两种可能的跳出方式。
- 第一种是读到EOF也就是文件的结束。
- 另一种是读到换行符。
- 只需要在读取到该两种符号的时候跳出，即可完成行注释的识别功能。

块注释：

- 进入块注释后，可以往后读任意的字符，有两种跳出方式。
- 第一种是遇到EOF。
- 第二种是同时遇到 */ 。
- 因此需要读取到上述两种符号的时候跳出块注释。

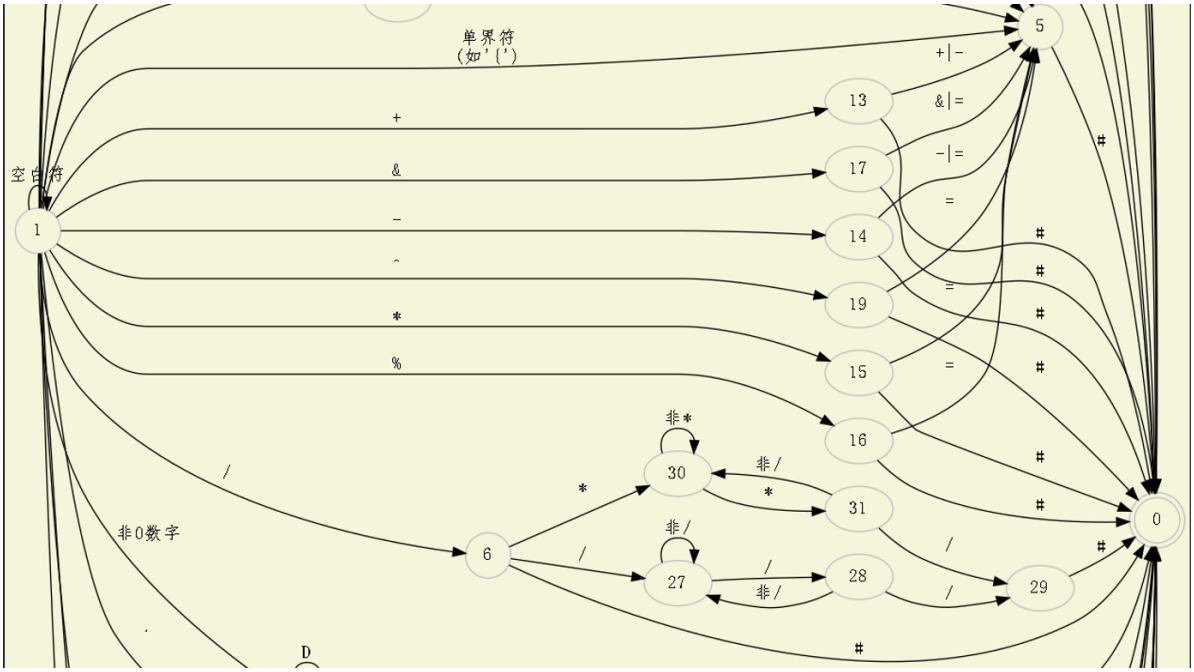
界符|运算符

我们能够将界符|运算符分成两种，一种是不管后续跟着的是什么字符，都无法在扩展新状态。另一种是可以扩展新状态的符号。

在第一种的情况，我们使用单独的状态记录，读取到下一个字符的时候就直接跳转为结束状态。

在第二种的情况，先扩展出所有的可能，并且判断扩展出的新的符号属于第一种情况还是第二种情况。若是此时的符号不属于当前所有的扩展可能，则可以跳转到终止状态。

由于最终生成的自动机比较长，因此本处只展示部分的自动机内容，完整的自动机的图示在”sources/DFA.png"中



该自动机可以通过一个函数实现：

```
int state_change(int state, char ch);
```

在该函数中，输入当前状态，输入字符。

能够返回转换后的状态。

实现了自动机的功能。

token识别：

将token读取出来，并且按照不同的终结状态前的状态进行分类，可以分成注释、实数、字符、字符串、标识符、关键字、界符、运算符共8种。其中注释是不需要记录token的。但是我们仍然需要记录行数。便于出现错误的时候进行提醒所在行数。

先对输入的状态进行分类，判断属于哪种类别。

若是属于注释，则直接返回注释所对应的代号。

若是属于实数，则将token放入实数表，返回实数的代号。

若是属于字符，则将token中的字符内容放入字符表，并且返回字符的分隔符所对应的代号，字符串的代号，以及字符的分隔符的代号。共三个。

若是属于字符串，则将token中的字符串内容放入字符串表，并且返回字符串分隔符代号，字符串代号，字符串分隔符代号，共三个。

若是属于标识符，则将token放入标识符表。并返回标识符代号

若是属于关键字，则返回该关键字所对应的代号。

若是属于界符或者运算符，则返回token中所对应的代号。

代号输出

从状态读取阶段中读出了token对应的代号之后，需要将其记录下来，按照本次的任务，需要将其读入文件中，因此建立一个文件，用于记录token所对应的内容，并且将对应的代号记录下来。需要用到文件的读写功能。

将所有的词素输出到token.txt中，用于后续的语法分析的使用。

语法分析

实验思路

在本次实现的语法中，主要实现了LL1分析法。

在LL1分析法中，需要有LL1的文法，而在设计LL1文法之前，需要先将原文法写出来，随后将其转换成LL1文法。除此之外，为了执行语义分析的步骤，还需要对添加语义动作。

实现功能

在本次实现的内容中，识别的语句包括**if-else语句**， **while语句** ，**函数定义的语句** ，**变量定义的语句**，**分号分割的表达式**，**赋值表达式**，**二元运算符算术表达式**，以及部分的**一元运算符表达式。**

在这里我们规定使用到的终结符：

终结符表如下所示：

符号	对应终结符	符号	对应终结符	符号	对应终结符
++ -- ! ~ + -	ω_0	* / %	ω_1	+ -	ω_2
<< >>	ω_3	< <= > >=	ω_4	== !=	ω_5
= -= += ... ^=	ω_6	||	"||"	&&	'&&'
|	"|"	^	"^"	&	"&"
(	"("	)	")"	;	";"
常数	c	标识符	v	字符串	s
void,int,float,char	$type$	if	if	else	$else$
{	"{"	}	"}"	while	$while$
,	','				

在实现过程中，考虑到实现的C语言文法的运算符优先级，在实现LL1分析法的语法分析器中，会考虑运算符的优先级。

实现的运算符按照优先级表示如下所示：

运算符	描述
++ -- ! ~ + -	一元运算符
* / %	乘除法运算符
+ -	加减法运算符
<< >>	移位运算符
< <= > >=	关系运算符
== !=	相等运算符
&	位与运算符
^	位异或运算符
|	位或运算符
&&	逻辑与运算符
||	逻辑或运算符
= -= += *= /= %= &= ^= |= <<= >>=	赋值运算符

实现语法分析器主要有以下步骤：

1. 确定文法，并且构造出扩展版本的文法。
2. 利用对应的文法构建分析表
3. 根据语义分析器的流程， 编写代码。

接下来我们主要描述如何实现上述的步骤。

初始文法构建

在C语言中，我们所能够实现的文法能够支持多个表达式，通过分号来分割，表达式中支持使用赋值表达式，算术表达式。由此我们开始构建C语言的文法规则。在具有运算符的文法的末尾添加动作函数GEQ。在if语句等部分，需要产生一个标记用四元式，标记此处为if语句的开始。

if语句：

- 在if语句中，需要注意的时if-else的判断，在这里，我将if-else语句定义如下：
- **IfStatement** -> if "(" **E** ")" {IF(if)} **Statement** {EL(el)} {IE(ie)}

  | if "(" **E** ")" **Statement** $else$ {EL(el)} **Statement** {IE(ie)}
- 其中因为可能会出现不被括号括起来的单语句或者使用括号括起来的多语句，而只要我们使用Statement，就能够将二者表示出来。
- 通过这个方法，还能够有效的保证else的作用域。
- 需要注意的是，为了保证一个if对应最多一个else，规定每一次都需要执行EL，哪怕最终并没有else语句，这样子就能够在生成四元式的时候，清楚的看到一个if对应一个else。

while语句：

- 在while语句中，需要注意循环的起止点。在这里，我将循环语句定义如下：
- **While_Statement** -> $while$ {WH()} '(' **Expression** ')' {DO(do)} **Statement** {WE(we)}
- 在该部分中，主要是将while分成三块，
  - while关键字，标记该语句的开始
  - 循环的条件，循环的条件应该是一个表达式，我们通过计算该表达式的结果判断是否应该终止循环。
  - 循环体：循环执行的内容。

定义语句

- 定义的语句分成两种，一种是函数的定义，一种是变量的声明与定义。
- **External_Declaration** -> **Function_Definition** [IDF()] {function entry}| **Declaration** ";"
- 在函数的定义中，我们主要关心函数的返回值，函数名，在我所实现的版本中，控制函数的参数为0；除此之外，还有函数的具体内容，通过compound_statement实现，compound_statement是被花括号括起来的一些语句。这里必须要有花括号。
- **Function_definition** -> $type$ [addT] **Direct_declarator** **Compound_statement**
- 在变量的声明与定义中，主要关心变量的类型以及变量名。其中变量名可能是通过逗号分隔开的变量名，因此有可能是一连串的变量名。
- **Declaration** -> $type$ [add_type] **Variable_List**
- **Variable_List** -> **Variable_List** ',' **init_declarator** | **init_declarator**
- 初次之外，在声明变量的时候，也有可能会对变量直接进行定义，如

```
int a = 10;
```

- 因此在声明变量的时候，还有可能添加定义的内容：
- **init_declarator** -> v [IDV] | v "=" **Assignment_expression** [GEQ w6]

表达式语句

- 在表达式语句中，主要可以分成三种语句
  - 赋值表达式：如

```
a = 10;
```

  - 二元运算符表达式：

```
c + d;
c == d;
```

  - 一元运算符表达式：

```
++b;
```

- 这三种语法可能会出现在同一条语句中，如
  -

```
a = c + d + ++b;
```

- 因此在设计的过程中，可以根据优先级关系以及左/右结合性来对其进行设计语法。
- 下面给出部分的有关优先级的表示：
  - **add_expression** -> **add_expression** ω_2 **mul_expression** {GEQ ω_2}| **mul_expression**
  - **mul_expression** -> **mul_expression** ω_1 **unary_expression** {GEQ ω_1} | **unary_expression**

LL1文法构建

由于上述初始的文法不属于LL文法，因此我们需要对文法进行一定的修改，使其符合LL文法，这样才能够使用LL(1)分析法实现语法分析器。

- 存在直接左递归，因此我们需要消除左递归
- 同一非终结符的多个候选式存在共同前缀，因此需要提取左公因子，直到非终结符中多个候选式不存在共同后缀为止。

转换成LL1文法后，需要计算每一个非终结符的first集和follow集。

计算出来之后，可以通过LL1文法的产生式以及first集合follow集，构建出LL(1)分析表。

由于生成的LL(1)文法与LL(1)分析表比较大， 因此此处不展示。

所有的文法在文件"sources/文法.md"以及"sources/文法.pdf"中

LL1分析表在文件"sources/LL1分析表.xlsx"中。

然后根据LL(1)分析表中构建执行LL1分析法的程序；

在程序中，需要通过当前符号和语义栈中的栈顶元素匹配，若是匹配失败，则说明语义分析失败。

分析法过程如下：

- 首先向语义栈压入初始非终结符P
- 执行循环
  - 从token中取出一个终结符n
  - 执行当语义栈不为空时的循环
    - 判断栈顶元素为终结符集合还是非终结符集合
    - 若是属于终结符集合
      - 栈顶与终结符n不匹配时，说明输入的语法有错误
      - 栈顶与终结符匹配时，若是都为#说明分析结束，否则可以将栈顶元素弹出
    - 若是属于非终结符
      - 判断栈顶非终结符的select集中是否包含n
      - 若是不包含，说明输入的语法有误
      - 否则将返回的产生式逆序压入。

伪代码如下：

```
stack.push(P)
while(true){
    n = next(w)
    while(!stack.empty())
        if stack.top() in terminal_set:
            if n != stack.top():
                return error
            if n == stack.top() == #:
                return success
            stack.pop()
```

```
        if stack.top() in Not_terminal_set:
            if garmmar = analize_list(n,stack.top()):
                stack.pop()
                stack.push(garmmar)
            else:
                return error
}
```

通过该代码，我们能够判断是否符合语法。

语义分析以及中间代码生成

在这一部分，主要是通过构造翻译制导文法来实现的四元式的结果生成，同时还能够判断是否符合语义：如运算表达式中的参数类型是否合法，避免出现char与int相加的操作。

而需要通过添加语义动作符号来实现上述部分。

语义动作构建

在语义动作的构建中

主要分成以下几种：

- 函数与变量定义：需要将对应的内容放入符号表中。
  - 在函数定义中，我们关心的是函数的返回值，函数名。知道这两个信息之后，就能够将其填入符号表
  - 在变量定义中，我们关心的是变量类型，长度，变量名。知道这几个信息之后，就能够将其填入符号表
- 表达式的四元式生成。
  - 表达式四元式生成中，我们需要注意操作符类型，操作对象。一般来说可以在符号表中找到对应内容。
- 跳转语句的四元式生成。
  - 在跳转语句的生成中，我们关心结束位置以及开始位置，跳转位置。

所有的语义动作如下

动作函数	动作	动作函数	动作
add_type	定义类型	add_name	将标识符添加到列表中
IDF	将列表中标识符添加到函数表中	IDV	将列表中标识符添加到变量表中
GEQ	运算符表达式的动作函数	IF	IF语句开头
IE	IF语句结束	EL	else语句开头
WH	while语句开头	DO	while执行
WE	while结束		

定义语句

在定义语句中，我们只关心标识符的名字，标识符的内容，因此能够使用任意一个数据结构将其保存下来，并且在执行的过程中写入到符号表中。

由于标识符可以是变量名，也可以是函数名，因此动作函数能够包含以下内容

- IDV：用于将变量名添加到变量表中：首先需要判断符号表中该变量是否已经被定义，若是已经被定义，我们则视为重定义，并返回错误。
- IDF：用于将函数名添加到函数表中：首先需要判断符号表中该函数是否已经被定义，若是已经被定义，我们则视为重定义，并返回错误。

在定义变量名的过程中，还需要查看是否通过赋值语句来对变量进行赋值。

运算语句

其中需要注意的是赋值语句的动作函数的位置，单目运算符的动作函数的位置。这两种文法的位置与二元的运算符的位置不同，动作函数的内容也不同。除此之外，还需要结合符号表的内容，需要判断变量是否在符号表中出现过，若是没有出现过，则说明该变量是违法的，不能直接使用。

我们将所有的运算符动作函数分成三种：

- 单目运算符的动作函数：GEQ(op0)，单目运算符包括++，--，+，-等。
    - 对于第一种类型，GEQ(op0)，我们的动作函数的执行如下：
    - 从SEM栈中取出一个栈顶元素 arg1。
    - 从OPS栈中取出一个运算符 op。
    - 记运算结果为result。
    - 需要注意的是，对于前自增或者前自减，会先执行操作，然后再返回结果，因此我们会将arg1视作运算结果，也就是说此时的result为arg1.
    - 四元式的表示为(op, arg1, _, result)
    - 并且将此时的result压入SEM栈。
    - 此时单目运算符的四元式已经能够正常的表示出来。
- 二元运算符的动作函数：GEQ(op1)，二元运算符包括+, -, *, / 等。
    - 对于第二种类型，GEQ(op1)，我们的动作函数的执行如下：
    - 从SEM栈中取出两个栈顶元素 arg1与arg2。
    - 从OPS栈中取出一个运算符 op。
    - 记运算结果为result。
    - 四元式的表示为(op, arg1, arg2, result)
    - 并且将此时的result压入SEM栈。
    - 需要注意的是，此时需要考虑两个栈顶元素之间，是否属于同一种类型，若是属于不同的类型，则会返回错误。
    - 此时二元运算符的四元式已经能够正常的表示出来。
- 赋值运算符的动作函数：GEQ(op2)，赋值运算符包括=，+=，-=等。
    - 对于第三种类型，GEQ(op2)，我们的动作函数的执行如下：
    - 从SEM栈中取出两个栈顶元素 arg1与arg2。
    - 从OPS栈中取出一个运算符 op。
    - 四元式的表示为(op, arg1, _, arg2)
    - 需要注意的是，此时需要考虑两个栈顶元素之间，是否属于同一种类型，若是属于不同的类型，则会返回错误。
    - 并且将此时的arg2压入SEM栈。
    - 此时赋值运算符的四元式已经能够正常的表示出来。

if语句

- 在if语句中，主要涉及跳转的指令，在四元式中，我们只需要将其表现出来即可，在后续的汇编代码生成的时候，就能够将其转换成对应的标记位置。
- 还需要注意else的生成，我们在此规定每一个四元式（IE,_,_,_）都存在一个对应的（EL,_,_,_），尽管可能在输入的程序中并不包含else，但是通过这个EL的生成，能够更好的判断IF的位置。

while语句

- 为了确保每次循环都能够计算一次循环条件，循环的起点应该在while关键字后，计算循环条件前。
- 而循环的末尾除了是标记while的终止以外，还需要执行无条件跳转，跳转到循环的开始。
- 只有通过判断循环条件，才能够跳转到while的终止点。
- 其中循环的条件判断应该是需要计算括号内的表达式的结果来确定的，表达式的结果会存在SEM栈中，因此只需要将SEM栈中的元素取出即可。
- 四元式表示为（WH,_,_,_）（DO,arg,_,_）（WE,_,_,_）

除此之外，当我们遇到运算符或者运算对象的时候，需要将运算符压入OPS栈，运算对象压入SEM栈。

然后是关于分号的处理，我们在赋值语句中可以看到，当我们使用分号的时候，已经与上一句的内容无关了，因此此时我们需要将SEM栈以及OPS栈内容全部清空。尽管栈内容清空了，但是因为我们是基于实验2的内容进行修改的语义分析器，所以不会影响实验二中语法的判断，因此也就不会影响语法的正确性。

最终，会将四元式结果输出到quadruple-output.txt中

目标代码生成

目标代码生成中，主要关注如何将四元式转换成汇编语言。

在此，我选择使用MIPS汇编语言生成。

部分使用到的汇编语言的介绍

下面是关于使用到的汇编语言，按照字母排列。

```
add $t1,$t2,$t3         Addition with overflow : set $t1 to ($t2 plus $t3)
addi $t1,$t2,-100       Addition immediate with overflow
and $t1,$t2,$t3         Bitwise AND : Set $t1 to bitwise AND of $t2 and $t3
beq $t1,$t2,label       Branch if equal
bgtz $t1,label          Branch if greater than zero
blez $t1,label          Branch if less than or equal to zero
bltz $t1,label          Branch if less than zero
bne $t1,$t2,label       Branch if not equal
div.d $f2,$f4,$f6       Floating point division double precision
lb $t1,-100($t2)        Load byte : Set $t1 to sign-extended 8-bit value from
effective memory byte address
lw $t1,-100($t2)        Load word : Set $t1 to contents of effective memory word
address
mul $t1,$t2,$t3         Multiplication without overflow
nor $t1,$t2,$t3         Bitwise NOR : Set $t1 to bitwise NOR of $t2 and $t3
or $t1,$t2,$t3          Bitwise OR : Set $t1 to bitwise OR of $t2 and $t3
ori $t1,$t2,100         Bitwise OR immediate
sb $t1,-100($t2)        Store byte : Store the low-order 8 bits of $t1 into the
effective memory byte address
sub $t1,$t2,$t3         Subtraction with overflow : set $t1 to ($t2 minus $t3)
sw $t1,-100($t2)        Store word : Store contents of $t1 into effective memory
word address
xor $t1,$t2,$t3         Bitwise XOR (exclusive OR) : Set $t1 to bitwise XOR of
$t2 and $t3
```

在生成过程中，需要关注符号表的使用以及寄存器的分配。

首先是对这一模块的一些关键数据结构的设计：

数据结构设计

寄存器管理

- 为了实现寄存器的分配，需要使用到两个内容
  - regs：记录当前寄存器储存的内容
  - regs_times：记录当前寄存器距离上次被访问的时间
- 在寄存器的分配中，我选择使用LRU(最少最近被使用）方式来管理寄存器。
- 对于变量以及临时变量，一共有8个寄存器可供使用。分别是$t0-$t7寄存器

$t0	8
$t1	9
$t2	10
$t3	11
$t4	12
$t5	13
$t6	14
$t7	15

- 对于常数，考虑到最多有两个参数，因此可以使用额外的两个寄存器：$s0和$s1

$s1	17
$s2	18

因为需要使用LRU方式来管理寄存器，因此使用数组的数据结构来对寄存器的使用情况进行控制。

跳转目标管理

在本次的设计中，因为涉及到比较的结果，因此使用分支类型的指令来实现比较

而在执行跳转的时候，就需要对标签进行管理。

由于本次实验中主要涉及if语句，while语句的跳转，结合比较的结果，一共有三种产生比较的语句。

- 比较语句，使用”bii“控制
- IF语句，使用”IFi“，”ELi“，”IEi“控制
- while语句，使用”WHi“和”IWi“控制。

其中，if和while因为涉及嵌套的可能性，需要使用栈的数据结构来控制各自的范围。

代码设计

在MIPS中，可以分成数据段.data和代码段.text

在数据段.data中，我们可以存放变量，本次实验的变量存放在这个部分，这个部分的数据来自于符号表。

而在代码段.text中，我们存放的内容是代码段的内容，也就是我们四元式的部分。

因此在生成目标代码的过程中，可以被拆分成两部分，数据段的目标代码和代码段的目标代码

数据段的目标代码

在数据段中，我们主要考虑变量的声明

- 生成变量的存储空间：
  - 首先需要根据符号表中的内容，确定总共有多少个变量被使用，然后再进行空间分配：
  - 这里通过函数"gen_data"来实现对数据段的内容的生成。
  - 由于我们在语义分析中已经实现了对变量的长度的判断，因此只需要判断对应的变量的代码长度，就能够实现对变量的数据分配。

```
void ASS::gen_data(Symbol_table* Sym) {
    outfile << ".data" << endl;
    for (int i = 0; i < Sym->variable.size(); i++) {
        outfile << Sym->variable[i].name << ":\t.space\t" << Sym-
>variable[i].len << endl;
    }
    outfile << ".text" << endl;
    for (int i = 0; i < Sym->func.size(); i++) {
        outfile << Sym->func[i].name <<":"<< endl;
    }
}
```

- 在上述代码中，首先需要输出".data"，然后才是我们需要定义的内容。

代码段的目标代码

在代码段中，我们主要关心两个内容

- 如何将变量或者临时变量放到寄存器中并且被识别使用
- 如何将四元式的内容转换成汇编语言的代码。

为了解决上述的问题，我设计了两个函数，一个用于处理变量与寄存器之间的关系，另一个用于处理运算操作和跳转操作。

变量与寄存器

在管理变量与寄存器时，我们遵循下列的规定：

- 每有一次使用寄存器的操作，所有寄存器的时间计数增加，被使用的寄存器的时间计数被置为0。
- 若是变量已经在寄存器中，返回寄存器编号。
- 若是变量不在寄存器中，选择一个最近最少被使用的寄存器，并且检查它里面的内容是否是变量
  - 若是内容为变量，将目前的内容保存到变量中，执行存储操作，在执行存储指令时，需要判断变量的长度，若是属于4，则使用sw指令，若是属于1，则使用sb指令。
  - 若是内容为临时变量，则可以考虑将该临时变量删除，而不需要写入任何的地方。
- 将该变量通过加载指令放到选择的寄存器中，寄存器的时间计数置为0。在执行加载指令时，需要判断变量的长度，若是属于4，则使用lw指令，若是属于1，则使用lb指令。
- 最后将该寄存器的编号返回，得到被操作的寄存器。

运算操作与跳转操作

在运算操作中，主要需要注意与四元式中的内容吻合，因此不是所有的操作都存在3个地址，比如说在赋值语句以及单元运算符中的语句，其四元式只有2个地址，因此需要使用if语句进行判断。

除此之外，对于大部分的运算操作，MIPS中已经有对应的指令，下列将对应指令与对应的运算符通过表格形式展示出来，对于小部分的运算操作，则在后续进行说明：

下列表格中，基本上所有的操作都符合 res := arg1 op arg2

操作	指令	操作	指令	操作	指令
*	mul	/	div.d	+(二元运算符)	add
-(二元运算符)	sub	<<	sllv	>>	srlv
|	or	&	and	^	xor

剩余的部分，因为需要执行特殊的操作，因此放到下面进行说明

- 自增/自减：
  - 由于是经过加一或者减一操作，因此可以使用addi或者addv指令
  - 格式为 addi $t0,$t1,1
- 单目运算符的+与-：
  - 由于是对自身进行操作，因此等价于0+自身或者0-自身，结果如下：
  - ```
    outfile << "\t" << "add" << "\t" << res << ',' << "$0" << "," << arg1 << endl;
    ```
- 比较符号：
  - 在比较符号中，由于需要产生比较的结果，在此可以通过分支指令b来实现。
- 在分支指令中，有如下指令
- 通过上述的指令，我们就能实现比较符号的汇编代码：
  - 可以构建如下的结构：
    - 分支符号，成功则跳转bi0
    - 将结果置为0，代表比较结果是False；并跳转到bi1
    - bi0:
  - 将结果置为1，代表比较结果是true；
    - bi1:
  - 之后的代码....
  - 通过该结构，能够将比较结果放到res中。
- 赋值符号 =：
  - 赋值符号可以看作 res = arg1 + 0
  - 因此可以通过add指令来实现，同单目运算符+与-的计算。
- 带运算的赋值符号
  - 带运算的赋值运算符，可以近似的看作 res = res op arg1；
  - 因此可以仿照表格中的运算符进行设计。
- 逻辑与 逻辑或操作：
  - 实际上只需要考虑左部是否为0和右部是否为0。
  - 因此通过分支运算来实现，MIPS指令中有能够与0比较的指令
  - 通过上述的指令，我们能够实现下列步骤
  - 逻辑与：
    - 比较左边是否为0，是0则跳转到bi0
    - 比较右边是否为0，是0则跳转到bi0
    - 将结果赋值为1，代表此时的左部和右部都不为0，即与操作为真，跳转到bi1；
    - bi0：
    - 将结果赋值为0，代表此时的左部和右部存在一个值为0，即与操作为假。
    - bi1：
    - 后续代码...
  - 逻辑或：
    - 比较左边是否为0，不是0则跳转到bi0
    - 比较右边是否为0，不是0则跳转到bi0
    - 将结果赋值为0，代表此时的左部和右部都为0，即或操作为假，跳转到bi1；
    - bi0：
    - 将结果赋值为1，代表此时的左部和右部至少存在一个值为1，即或操作为真
    - bi1：
    - 后续代码...
    ```

if语句

在IF的四元式之前，按照语法规定，已经将结果计算好，并且作为参数arg1传入，因此只需要判断该arg1的值是否为0，就能够执行跳转。

需要注意的是，因为涉及IE和IF，因此需要在不成立的时候，将结果跳转至ELi。

而我们在设计ELi的时候，需要在EL前一句加入 b IEi 跳出if语句，否则代码会继续执行else部分的内容。

while语句

在设计while语句，需要关注while语句的起始地址，执行跳转的过程同样可以通过branch的指令来执行。

while语句的结构大致可以设置如下：

- WHi： 作为起点
- E：计算表达式的值
- 若是表达式的值为0，跳转到WEi；
- 循环体的末尾，无条件跳转回WHi的起点。
- WEi：作为while语句末尾
- 后续代码...

跳转的寄存器考虑

因为跳转后，寄存器取值可能不同，因此需要进行处理。

假如跳转回去while中，还需要再特定的位置中加入寄存器的存储。比如说在循环判定条件前和循环判定时，使用了a，此时a是不需要读入寄存器的，已经存在于寄存器t0中。但是在循环的内容中因为a没有被使用，因此被换出，此时执行跳转到循环开始时，则会出现while的判定失败的问题，因为此时寄存器t0存的值并不是a所对应的值。

这里可以通过在每个代码块内，默认寄存器初始化，需要重新读入，这样子能够减少大部分的问题，但是相反的，代码会变得更加复杂，加入了存入和读取的过程。

```
void ASS::reg_claer(Symbol_table* sym) {
    store_data(sym);
    for (int i = 0; i < 8; i++) {
        regs[i] = "empty";
        regs_times[i] = 10;
    }
}
```

也就是说将代码块分块后，每一个代码块的结束位置，比如说在if分支语句前，while语句开始前以及的结束位置，都需要将结果保存下来。这样子能够确保每次进入代码块时，都是属于相同的状态。

实验结果

测试说明

在本次的程序中，需要输入源代码文件名。如demo.c。

demo.c中的代码如下：

```
int main() {
    int a = 1, b = 0, c = 20;
    a = 1;
    b = a + c;
    c = 0;
    while (a <= b && a <= 40) {
```

```
        a *= 2;
        b = b * 2 - c;
        if (c < 10) {
            ++c;
        }
    }
    if (b < 2 || a > 30) {
        b = 4;
    }
    else if (b <= 4) {
        b = c ^ b;
        if (a > b) {
            b = a | b;
        }
        else {
            a = b;
        }
    }
    b <<= 4;
    int d = a * 1 + b * 2 + c * 3;
}
```

在其中，包含了基本的四则运算之外，还有if语句，while语句的实现，算术与，算术或的运算，位移运算，赋值运算，自增运算等。

变量c在循环中，充当了计数器的作用，用于计算执行了多少次循环操作。

词法分析

token的表示形式为<val, code, line>

对于上述的代码，我们能够得到token串如下(只列出部分内容)：

```
1   < int, 7, 1>
2   < 0, 4, 1>
3   < (, 39, 1>
4   < ), 40, 1>
5   < {, 37, 1>
6   < int, 7, 2>
7   < 1, 4, 2>
8   < =, 54, 2>
9   < 1, 1, 2>
10  < ,, 79, 2>
```

此部分内容包含了开始部分的几个token串。

完整的token串在文件"result/token.txt"中

语法分析

在语法分析过程中，会展示出成功或者失败，而如果在遇到失败情况下，会显示出部分的错误。

如语法分析中，LL(1)分析表中不匹配时，会有匹配内容的错误之处：

假如将第二行中代码，修改如下：

```
int main() {
    int a = 1, b = 0, ;
```

则会在运行时，展示如下错误：

```
error: terminals "v" and ";" isn't match.
```

提示说需要一个变量v，但是此时只有分号"；"。因此会导致出错。

而在进行语法分析的过程中，会将所有的LL1语义栈打印出来。

部分LL1语义栈展示如下：

```
Stack: # P
Stack: # P1 ExD
Stack: # P1 ExD1 v type
Stack: # P1 ExD1 v
Stack: # P1 ExD1
Stack: # P1 Func1
Stack: # P1 CS {IDF()} ) (
Stack: # P1 CS {IDF()} )
Stack: # P1 CS {IDF()}
Stack: # P1 CS
Stack: # P1 } CS1 {
Stack: # P1 } CS1
Stack: # P1 } Ss
Stack: # P1 } Ss S
Stack: # P1 } Ss ; Declaration
Stack: # P1 } Ss ; VL type
Stack: # P1 } Ss ; VL
Stack: # P1 } Ss ; VL1 {IDV()} v
Stack: # P1 } Ss ; VL1 {IDV()}
Stack: # P1 } Ss ; VL1
```

该部分内容是关于从初始P进入到函数的过程。

所有的LL1语义栈在文件“result/LL1-Stack output.txt”中

语义分析

在语义分析中，除了会生成四元式，还会检查运算过程中类型是否匹配。

若是出现不匹配的情况，则会将其打印出来。

如将代码中第三行改为a = 0.1;

```
int main() {
    int a = 1, b = 0, c = 20;
    a = 0.1;
```

此时会有类型错误的说明，如下所示：

```
Error: Different type can't be calculate.
```

除此之外，还会根据语义动作，完善符号表以及生成四元式。

动作过程：

```
action:
action: GEQ(op1)
action:
action: next(w)
action: next(w)
action: action:
action:
action: next(w)
```

动作过程展示了在每一步骤中，选择了执行什么语义动作。

完整的动作过程在文件"result/action.txt"中

符号表内容如下所示：

```
SYMBOL TABLE
            name            type      character         address
            main               f             id               0
               a               v             id               0
               b               v             id               1
               c               v             id               2
               d               v             id               3


VARIABLE TABLE
            name            tval             len
               a             int               4
               b             int               4
               c             int               4
               d             int               4


FUNCTION TABLE
            name    return_type
            main            int

```

该符号表中，记录了所有遇见的标识符的信息

符号表的展示在文件“result/Symbol_table.txt”中。

部分四元式的生成如下所示：

```
(= 1 _ S1)
(= 0 _ S2)
(= 20 _ S3)
(= 1 _ S1)
(+ S1 S3 t0)
(= t0 _ S2)
(= 0 _ S3)
(WH _ _ _)
(<= S1 S2 t1)
(<= S1 40 t2)
(&& t1 t2 t3)
(DO t3  _ _)
```

该展示部分显示了程序开始的部分，包括a，b，c的初始化和while语句的入口。

完整的四元式生成在文件“result/quadruple-output.txt”中

目标代码生成

在目标代码生成的过程中，直接输出了对应的目标代码文件

部分汇编代码文件如下：

```
.data
a:  .space  4
b:  .space  4
c:  .space  4
d:  .space  4
.text
main:
    ori $s0,$0,1
    lw  $t0,a
    add $t0,$s0,$0
    ori $s0,$0,0
    lw  $t1,b
    add $t1,$s0,$0
    ori $s0,$0,20
    lw  $t2,c
    add $t2,$s0,$0
    ori $s0,$0,1
    add $t0,$s0,$0
    add $t3,$t0,$t2
    add $t1,$t3,$0
    ori $s0,$0,0
    add $t2,$s0,$0
```

该文件能够在MARS中运行。

执行该汇编语言文件，通过观察对应的abcd的值，来判断汇编语言是否能够成功按照指定的代码流执行。

在初始位置，程序尚未执行的时候，对应的位置的内容如下所示。

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x00002000	0	0	0	0
0x00002020	0	0	0	0

点击运行程序后

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)
0x00002000	64	64	6	210
0x00002020	0	0	0	0

data段中的数据发生改变，a变为64，b变为64，c变为6，d变为210。

作为对比，使用gcc编译C文件，并且在文件的最后将其结果输出：

```
PS F:\C++>  & 'c:\Users\hhr\.vscode\e
er.exe' '--stdin=Microsoft-MIEngine-I
-MIEngine-Error-0rb2xlb4.uck' '--pid=
ter=mi'
a=64
b =64
c =6
d =210
```

可以发现，运行的结果是一致的，说明了此时我们的编译器能够生成Mips汇编代码，并且能够成功的运行。

生成的汇编代码文件在“result/target.asm”中。

课程设计总结：

本次的实验，结合了以往的所有实验部分，并且需要进行融合起来，使得最后能够生成汇编程序。在这个过程中，需要对编译原理的内容比较熟悉，对每个部分的内容要足够了解，否则在写代码的过程中容易出现错误。

总结

词法分析

在词法分析过程中，主要是需要注意自动机的设计，需要考虑的内容有很多，包括标识符的识别，关键字的识别。在实数的范围内尤其需要注意，在一开始的设计中，只是简单的识别了10进制的整数，但是在进行测试的时候，注意到当我们使用浮点数，或者用16进制表示的时候，就会出现读取错误的情况，这个时候就需要考虑进制的表示方法，并且按照规则去读取。

字符串的识别也是比较大的问题，在读取的时候，因为需要考虑转义符'\'，并非遇到对应的字符的时候就能够进入终止态，而是需要判断是否存在转义符，才能进入终结态。

使用文件指针读取实现字符的读取，因此在遇到token的结束符的时候，还需要将文件指针修改到前一个位置。这样子才能够避免出现漏读的现象。

在获取到token后，还需要将其填入符号表中，用于后续的判定作用。

语法分析

在语法分析中，遇到了不少的问题，因为多实现了关于赋值表达式的规则，而赋值表达式按照C语言的规则，左值是不允许作为表达式出现的，所以需要对左边的元素个数进行限制，因此，可能会出现单个元素单走的情况，这个时候，我们难以判断是使用算数表达式计算，还是用赋值表达式计算，因此在这里需要考虑后续接的内容是什么，当我们只有单个的时候，我们需要引入空边，允许我们后续接的内容是UnE的follow集，这个时候我们还要分成几种情况，若是后续接入的内容为等号，我们就视为赋值表达式，若是后续接入的内容为运算符，我们就视为二元表达式的计算，若是后续接入的内容为分号或者括号，这种情况就视为空边。通过一段时间的研究，我才能够实现相关的表达式实现。

在本次实验中的语法分析，仍然发现有不足的地方，比如说可以减少文法的复杂度，目前的文法相对来说，规模比较大，应该式存在方法能够较少文法的复杂度。

除此之外，在进行设计的过程中，还学会了LL1文法的变换过程，能够亲自将原石文法其变换成LL1文法。

语义分析

语义分析的工作主要在语法分析基础上进行，只需要在合适的地方加入语义动作，构建出翻译制导文法。

在添加表达式动作函数的时候，需要注意的地方有动作函数的辨别，在这个过程中，其实对于每个不同的运算符应该都有一个不同的动作函数，但是因为其中涉及了大约20种的运算符，因此我决定将其分类。正如我上述所说的将动作函数的分类分成上述的3种：单目运算符，二元运算符，赋值运算符。

在这种相同类别的动作函数中，他们也执行的内容几乎是一致的，唯一的区别是运行的运算符不同。就是说他们可以共用同一套模板，因此难点变为了如何确定此时使用的是什么运算符，比如说加号与乘号都属于二元运算符的类型，他们执行该类型的动作函数，使用的是同一个规则，但是我需要知道加法的时候使用的是加号，乘法的时候使用的是乘号。因此这个时候我们就引入了一个操作符栈OPS，该OPS能够存放遇见的操作符。而当我们需要执行动作函数的时候，就能够从OPS中取出操作符。这样子就能够解决了确定运算符的问题。而且按照动作函数在文法中的位置，必定是选择最近的操作符来执行，也就是说不会发生乱序的情况。

除此之外，还需要判断是否存在相同的类型，使得两个运算符之间能够进行运算。

在添加定义变量的表达式中，主要是和符号表共同使用，但实际上，也可以将定义语句转换成四元式，然后保存其地址，通过栈指针来分配空间。

而在控制跳转的变量表达式中，主要是需要判断

因为与之前的语法分析器相比，我们只是往其中加入了动作函数的功能，所以不会出现文法识别错误的情况，依旧能够保证有正常的语法识别功能。也就是说能够识别出当前的表达式中的语法错误。

语义分析主要基于语法分析的代码。由于我在语法分析中的文法涉及到了优先级的比较，因此在添加动作函数的时候，并不需要太多的工程量，只需要在相对应的文法上添加动作函数GEQ(op)，然后再分析的过程中添加读取到动作函数时的执行内容，整个过程相对来说比较轻松。

目标代码生成

在目标代码生成中，其实关注的内容只有寄存器的分配与使用，由于我们寄存器数量是有限的，什么时候需要将变量或者临时变量存入寄存器，什么时候不需要使用寄存器，这是需要考虑的内容。

在考虑这部分的时候，还需要考虑跳转的问题。这部分需要进行处理，否则会出现很大的麻烦，代码运行会出现出错。比如说我认为此处是变量a，但是实际上临时变量t1。而通过代码块处理后，能够变成合适的代码。

这里可以通过在每个代码块内，默认寄存器初始化，需要重新读入，这样子能够减少大部分的问题，但是相反的，代码会变得更加复杂，加入了存入和读取的过程。

```
void ASS::reg_claer(Symbol_table* sym) {
    store_data(sym);
    for (int i = 0; i < 8; i++) {
        regs[i] = "empty";
        regs_times[i] = 10;
    }
}
```

该部分是存在于每一个代码块的结束时，比如说在if分支语句前，while语句开始前以及的结束位置。这样子能够确保每次进入代码块时，都是属于相同的状态。

确定了逻辑后，再决定目标代码的生成就相对轻松许多了。只需要查阅对应的指令，来使得能够实现对应功能的代码。

收获

经过本次课程设计，完整的实现了从源代码到目标代码的生成过程，在这个过程中，需要结合在本学期中的课堂上学习到的内容进行设计。经过本次的设计，对于编译原理有了更大的收获，理解了如何读入token并且设计文法将对应语言转换成中间代码。而在中间代码的处理过程中，还需要对汇编语言有一定的了解。

美中不足的时，由于时间问题，并没能将浮点数的内容也一并实现，本次实验中主要是关于整数的操作，实际上在浮点数操作中，需要在浮点数的寄存器$f0,$f1等进行操作，然而在浮点数寄存器的管理上，需要分辨每一个变量的类型，还需要判断什么时候能用浮点数与整数进行操作，什么时候不能用浮点数与整数操作。这都是需要考虑的问题，在本次实验中没能完善。但是在本次实验中，几乎所有关于整数的操作都能够实现了，包括常规的四则运算，if else语句的使用，while语句的识别，以及位运算和逻辑运算操作。都已经能够实现了。

建议

在实验开始的时候，应该从符号表开始设计，因为整个编译过程中都涉及到了符号表的管理，调整一下顺序，先说明符号表的管理，然后才是词法分析等内容，否则先写完词法分析，在说符号表管理时，会导致在后续修改代码的时候出现比较多的问题，从完整的架构上说，符号表也是编译原理中关键的部分。

附录：

代码文件在src中。

实验结果在result中。

一些关键的内容，在本次实验中未提到的内容在sources中，包括自动机，文法，LL1分析表等。