本科生实验报告

实验课程：编译原理实验

实验名称：编译器构造实验

专业名称：计算机科学与技术（人工智能与大数据方向）

学生姓名：马浩宇

学生学号：19335154

实验地点：实验室/课后

实验成绩：

报告时间：2022.7.3

1、实验要求：

实验目的

1. 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；
2. 培养学生独立分析问题、解决问题的能力，以及系统软件的设计的能力；
3. 提高程序设计能力、程序调试能力；

具体要求

前端

一个简单文法的编译器前端的设计与实现：

- 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）；
- 扫描器设计实现；
- 符号表系统的设计实现；
- 语法分析器的设计实现；
- 中间代码设计；
- 中间代码生成器设计实现；

后端

一个简单文法的编译器后端的设计与实现：

- 中间代码的优化设计与实现（可选）；
- 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；

2、设计细节

文法设计

按照实验要求，本次实验设计的文法包含变量说明语句、算术运算表达式和赋值语句，初次之外还扩展了包括if、else语句和基础的包括与或非的逻辑运算表达式，其中设计的文法如下所示：

实验设计文法.

<program> → <state_seq>

<state_seq> → <declaration>; <state_seq> |
<assignment>; <state_seq> |
<state_if> <state_seq> | ε

<declaration> → <type> <id_list>

<type> → int | bool | double

<id_list> → id <option_assignment> <id_list'>

<option_assignment> → = <expression> | ε

<id_list'> → , <id_list> | ε

<state_if> → if (<state_if'>

<state_if'> → <expression>) { <state_seq> } <option_else>

<option_else> → else { <state_seq> } | ε

<assignment> → id = <expression>

<expression> → <simple_expression> <expression'>

<expression'> → w_0 <simple_expression> <expression'> | ε

<simple_expression> → - <term> <simple_expression'> |
<term> <simple_expression'>

<simple_expression'> → w_1 <term> <simple_expression'> | ε

<term> → <factor> <term'>

<term'> → w_2 <factor> <term'> | ε

<factor> → (<expression>) | ~<factor> | id | num

其中 id 表示标识符
num 表示数字
w_0 包含 {>, <, ≤, ≥, ==}
w_1 包含 {+, -, ||}
w_2 包含 {*, /, &&}

文法设计主要包括4个主要部分：变量声明部分、赋值语句部分、if语句部分和表达式语句部分。文法的设计基于c语言的常见文法表达，语句之间通过`;`分割，语句块中通过`{}`中括号来进行划分。为了后续扫描器、语法分析器等部分的设计方便，所以还需要规定好本次实验支持的关键词表和界符表：

关键字	界符	界符
int	;	*
double	,	==
bool	=	<=
if	-(取负)	>=
else	+	<
	-（减）	>
	/	{
	&&	}
	||	~

因为采用的是最终定义语法的关键字和界符如上所示而设计词法分析器时对应的可识别的关键字和界符包含了很广，所以每个符号对应的编号在这里没有给出。

扫描器（词法分析）

本次实验中的词法分析器采用的为实验1中设计的词法分析器，虽然最终没有实现全部的关键字和界符的功能但是其能识别到的界符和关键字如下图所示：

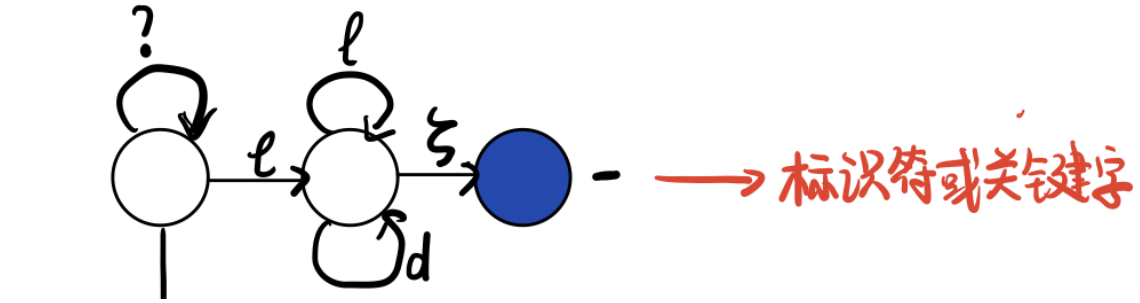
```
map<string, int> kt = {
        {"auto",4},{"void",5},{"char",6},{"const",7},{"double",8},{"float",9},
        {"int",10},{"long",11},{"if",12},{"else",13},{"for",14},{"while",15},{"continue",16},
        {"break",17},{"switch",18},{"case",19},{"main",20},{"bool",21}};

map<string, int> pt = {
        {"(",30},{")",31},{"[",32},{"]",33},{"{",34},{"}",35},{"+",36},{"++",37},{"+=",38},{"-",39},
        {"--",40},{"-=",41},{"*",42},{"*=",43},{"/",44},{"/=",45},{"%",46},{"<",47},{">",48},{"=",49},
        {"<=",50},{"==",51},{">=",52},{"&",53},{"|",54},{"^",55},{"&&",56},{"||",57},{"//",58},{"/*",59},
        {"*/",60},{",",61},{";",62},{"<<",63},{">>",64},{"->",65},{"-d",66},{"~",67}};
```

将对应的界符表和关键字表通过`map<string,int>`的数据结构进行存储，方便查询对应符号的编号。除此之外，将标识符的种类码定义为00，字符常量的种类码定为01，字符串变量的种类码定为02，数字常量的种类码定义为03。

在定义好界符、关键字等的种类码后，接下来进行自动机的设计，对应的自动机可以识别的部分包括：**普通标识符、设定好的关键字及界符、数字常量（整数、小数、科学计数法）、行注释及块注释、字符串和字符常量的识别。**虽然这些功能某些并没有在后续的语法和语义分析中利用到，但还是将其尽可能完善的设计，（其中只有标识符、关键字及界符、以及数字常量在后续的设计中继续使用到，其他的功能只有词法分析能够完成，后续部分无法完成，特此声明），其中具体每个部分的思路如下所示：

标识符及关键字

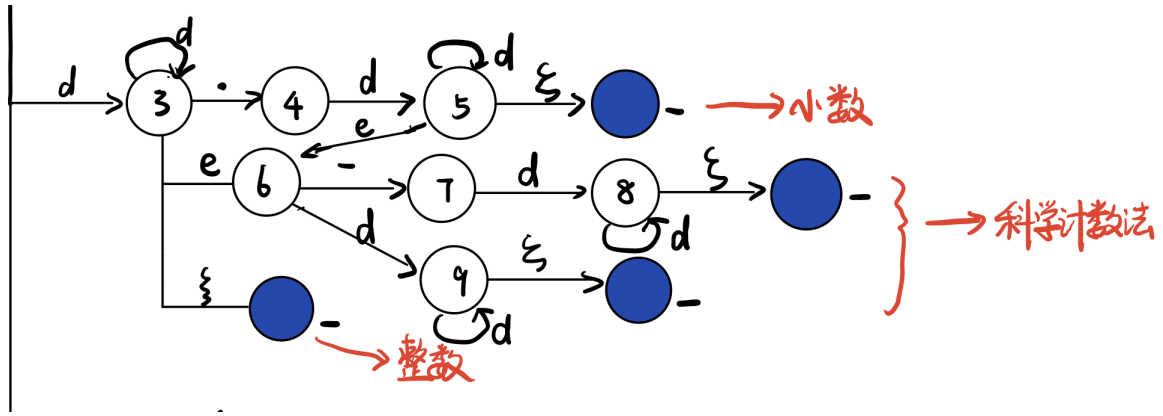
由C语言设计标准中标识符只能由下划线、字母和数字构成，并且标识符不能和已经设定好的关键字重复，所以在设计词法分析器过程中可以将关键字看作为一种特殊的标识符，两者的自动机表示相同，其对应的自动机设计如下图所示：



其中对应的？为空格、换行符等需要跳过的词，l表示字母，d表示数字，ζ表示除了表示出的字符外其他的字符，该自动机的含义为当处理字符串过程中如果读取到字母或者下划线则进入下一个状态，并且可以重复读取字母、下划线或者数字，当读到除这些字符以外的字符时，则进入终止态。当获取了这样一个字符序列后，首先默认其为标识符，接下来通过`map.count()`函数来判断该字符串是否在关键词表中出现过即是否是关键词，如果是关键词则通过map对应的键值对找到其对应的种别码进行储存，如果不是则通过`find()`函数查看该标识符是否出现在iT表中，如果是首次出现，则还需要将其储存进入对应的iT表中。

数字常量

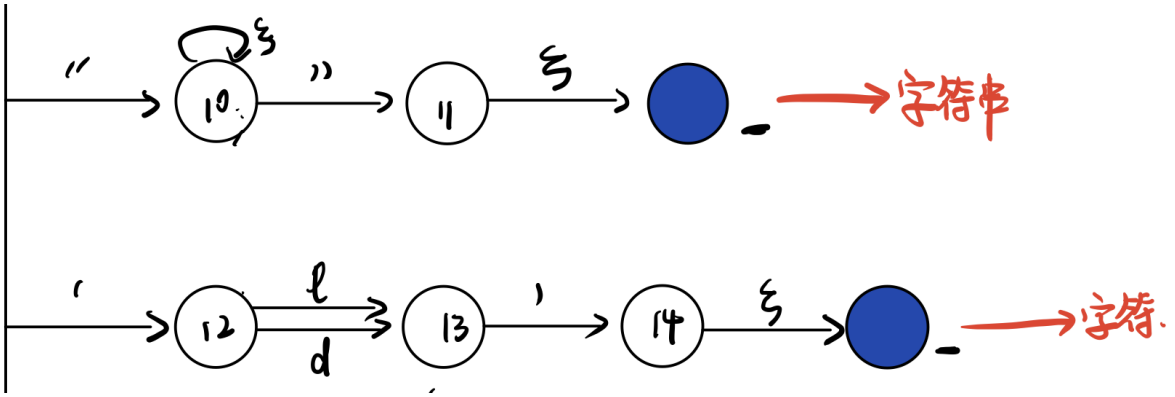
可以识别出的数字常量包括整数、小数和科学计数法表示，对应的自动机设计如下所示：



对应首先从初始状态1读取到数字则进入状态2可以重复读取数字，如果此时读取到不是小数点、字符e、数字，则表示该数字读取结束为一个整数，则进入终结态。接下来如果读取到小数点则进入状态4，在该转态如果继续读取到数字则表示该数字常量表示一个小数，如果读取到的不是一个数字的话，则该格式不符合C语言的要求，进入报错状态，进行报错提示。同样在状态5可以反复读取数字，接下来如果读取到字符e，则进入状态6，表示为一个科学计数法，如果是其他字符则进入终止态。同样在状态3读取到字符e也可以进入状态6同样表示科学计数法，在e后面可以紧跟着一个负号，接下来即重复的读取数字，直到识别到的字符不为数字表示该数字常量读取结束。读取到的数字常量，同样除了按照对应的类别码储存到tokens结果数组中，还需要储存到对应的CT表中。

字符串及字符常量

可以识别到“ ”，' '对应的字符串（包括空串）和字符，其对应的自动机设计如下图所示：



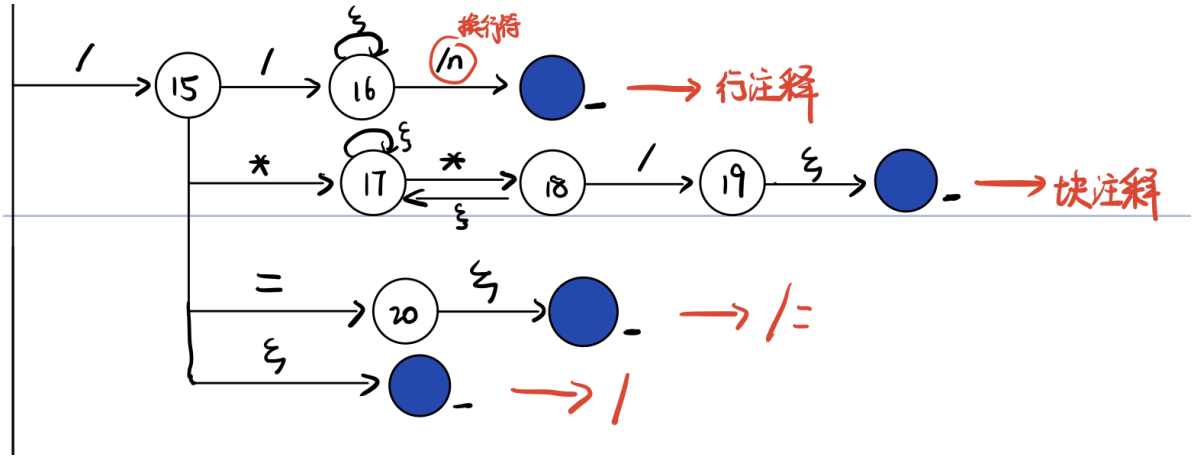
其中设计和ppt上略有不同，ppt上的字符串识别自动机无法识别到空串。所以在进行字符串的识别进行了一些修改，对应的从初始状态读取到`"`或者`'`时分别进入状态10和12，值得注意的是字符串可以使任意长度的，而字符常量只能由一个字符组成，否则则会产生报错提示。分别在读取到下一个双引号或者单引号时准备进入终止状态。

在设计读取字符串时，比如要考虑到转义字符存在的情况，所以当在读取字符串和字符时如果识别到`\`转义字符时，需要自动将下一个字符不能将其当做具有状态跳转意义双引号或者单引号进行读取。进而实现了可以识别转义字符的功能。

与上面相似的，在完成对应的字符串和字符读取后，除了按照对应的种别码储存在tokens结果数组中之外，还需要将其储存在sT和cT数组中。

行注释及块注释

在本次实验设计标准中，注释中的内容不会被当做标识符、关键字或界符等进行记录，这就要求当进入了读取注释的状态时，要忽略后面的字符内容，直到从对应的注释状态中退出后，才能重新进行记录，具体关于注释的自动机如下所示：

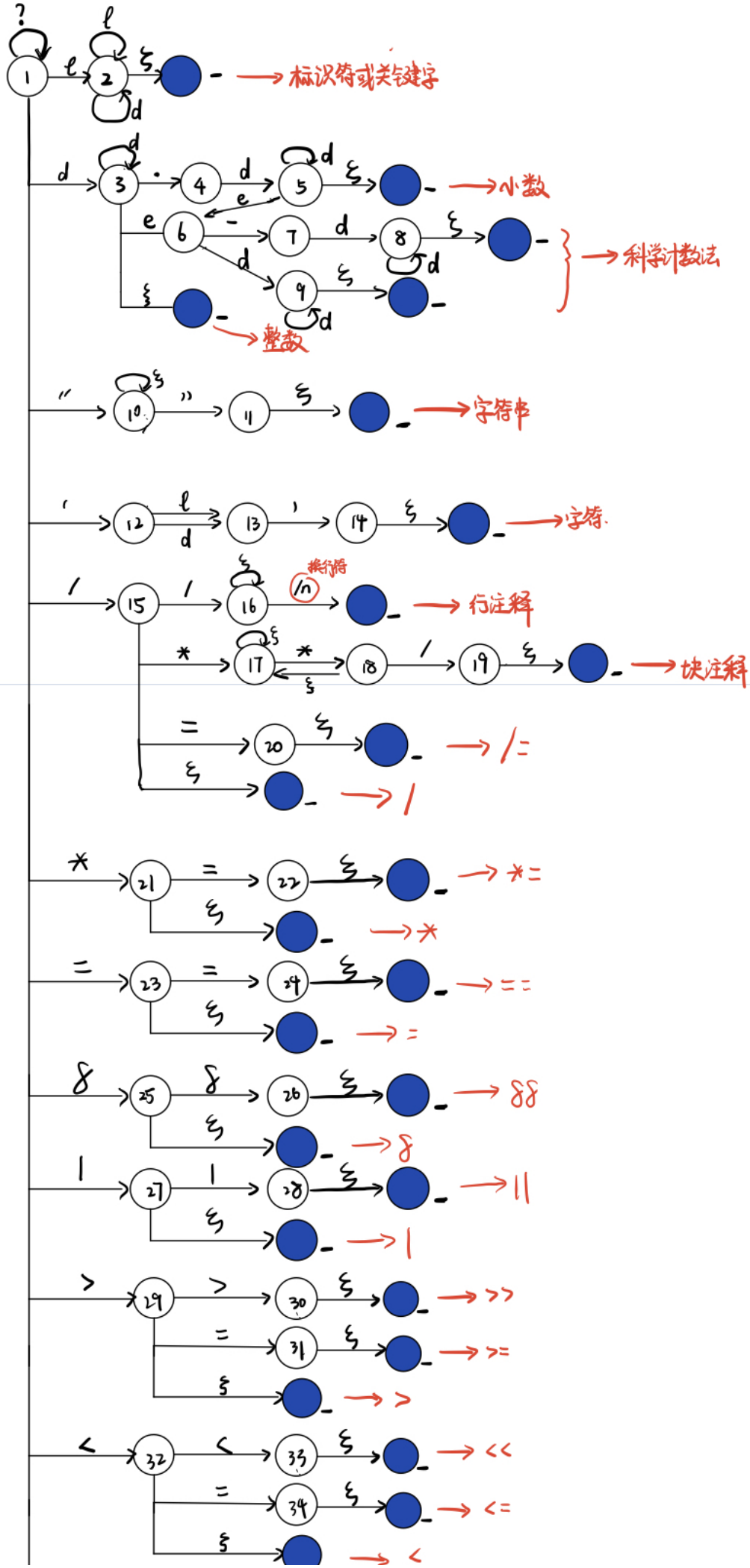


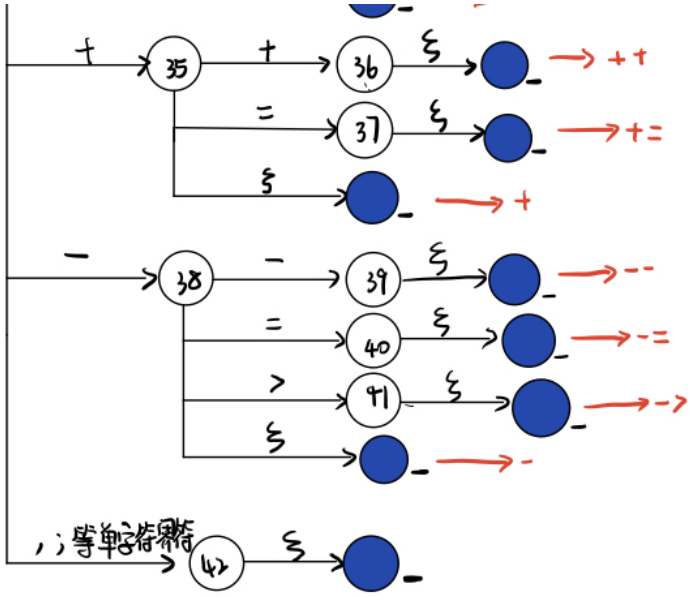
从上面自动机可以看出，从初始态如果连续读到字符//或者/*时分别进入行注释状态16和块注释状态17，在注释状态时，无论读到任何字符都不进行记录，直到读到换行符即退出行注释状态，而连续读到*/时字符时，结束块注释阶段，在对应的终止态时分别将对应的行注释符和块注释符和对应种别码存入tokens。

一般界符的识别

一般界符是指基本运算符，识别的大体思路类似，基本上为按流程识别到对应的单目运算符后，在根据后续的字符判断是否为双目运算符，并对应不同的终止态，但是在通过代码实现时，发现可以各个运算符共享识别到单词后缀符进入终止态的状态，因为在原本的设计思路中需要通过不同的终止态来判断对应的运算符分别是什么，而在实现过程中通过map储存所有界符的方式，只需要在终止态通过对应界符的储存的字符串即可得到对应的种别码，并储存到tokens中。

完整的自动机模型如下图所示：


标识符或关键字
小数
科学计数法
整数
字符串
字符
行注释
块注释
/=
/
*=
*
==
=
&&
&
||
|
>>
>=
>
<<
<=
<



符号表系统

符号表系统的目标是完成标识符表、常数表的设计，便于在语法及语义以及后续目标代码生成中利用标识符及数字常量的属性信息。

首先定义表示符的属性信息为一个新的类`id_attribute`，其中包含的信息包括标识符类型、种类、地址以及后续目标代码生成中需要利用到的活跃状态信息等4个属性。而符号表系统的定义通过`map<string,id_attribute>`，其中string对应为标识符的名称，对应的value为其属性，每个标识符在声明变量时即定义好其在内存中的相对地址，便于目标代码生成过程中的读写寻址操作。值得注意的是因为要将自定义的类定义为map的value，所以需要定义其排序规则，通过重载<操作符，根据两个变量的地址ADD比较大小。

常数表储存的为变量的值和其对应的类型，通过`map<string,string>`来实现，key对应为常量的数值，而value对应为常量的类型（如int或double或bool）。

语法分析器

语法分析器的任务是给定对应的程序，可以根据定义的文法判断该程序是否符合该文法。

本次编译器构造实验的语法分析器通过递归子程序下降法完成，根据已经定义好的文法设计子程序即可，因为后续对应的翻译文法的语义分析器也是通过递归下降子程序法实现，而语义分析器即为在语法分析器上添加了语义动作的过程。所以对应的每个子程序的设计图将在后续中间代码生成部分的语义分析器进行展示。

每次为语法分析器加载对应的Token序列，整个语法分析过程主要包括两个过程：取下一个字符以及匹配lookahead和期望的字符。具体代码实现通过一个全局指针`int token_pointer = 0;`来选取当前的Token，并通过一个匹配函数`match()`来判断当前lookahead是否与期望的字符匹配，如果不匹配则报错，并通过`exit()`终止程序。每次分析时首先调用第一个子程序`program()`开始递归分析过程。

中间代码生成

语义分析器的任务是根据词法给定的token序列，以及规定好的翻译文法，执行对应的翻译文法中包含的语义动作。语义动作基本分为两个部分：有关中间代码生成的语义动作和有关符号表内容填写的语义动作，首先需要将给定的文法变为对应的翻译文法，将本次实验设计的文法转换为对应的翻译文法如下图所示：

实验翻译文法

<program> → <state_seq>

<state_seq> → <declaration>; <state_seq> |
<assignment>; <state_seq> |
<state_if> <state_seq> | ε

<declaration> → <type> <id_list>

<type> → int{change_type} | bool{change_type} | double{change_type}

<id_list> → id{write_IDtable} <option_assignment> <id_list'>

<option_assignment> → = {push} <expression>{ASSIGN} | ε

<id_list'> → , <id_list> | ε

<state_if> → if (<state_if'>

<state_if'> → <expression>){IF} | <state_seq>} <option_else>{IE}

<option_else> → else{EL}{ <state_seq> } | ε

<assignment> → id = <expression>

<expression> → <simple_expression> <expression'>

<expression'> → w_0 <simple_expression>{GEQ(w_0)} <expression'> | ε

<simple_expression> → - <term>{GEQ1(-)} <simple_expression'> |
<term> <simple_expression'>

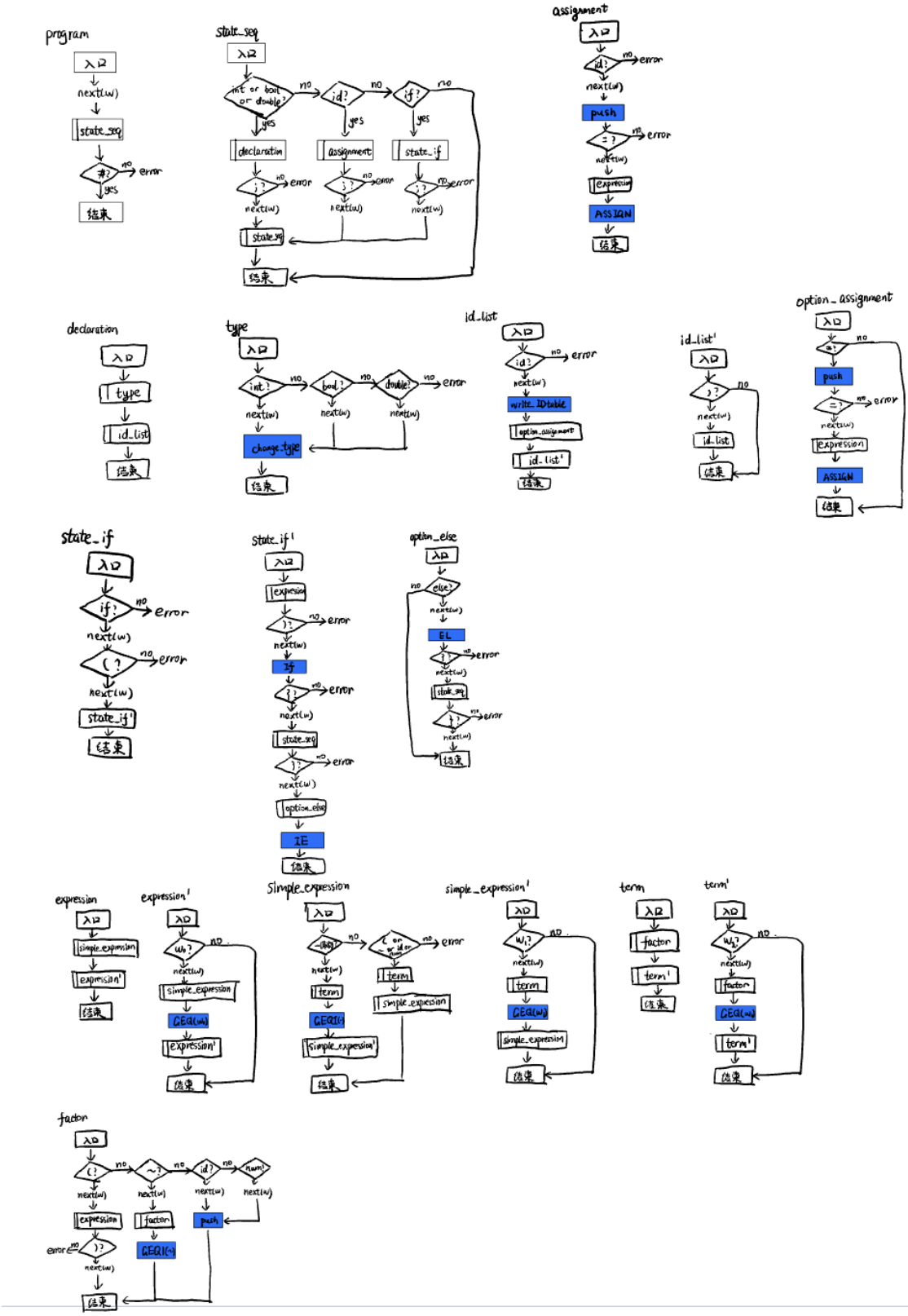
<simple_expression'> → w_1 <term>{GEQ(w_1)} <simple_expression'> | ε

<term> → <factor> <term'>

<term'> → w_2 <factor>{GEQ(w_2)} <term'> | ε

<factor> → (<expression>) | ~ <factor>{GEQ1(~)} | id{push} | num{push}

本次实验采用了递归下降子程序法实现，对应的子程序如下图所示：



具体的代码实现在语法分析器的基础上加入了对应语义动作对应的函数，其中子程序图中对应的蓝色部分对应为语义动作，可以从子程序图中看出主要包括`change_type()`,`write_IDtable()`,`ASSIGN`,`GEQ`,`GEQ1`,`push`,`IF`,`EL`,`IE`。

- `change_type()`函数的任务是改变全局变量type的类型和数据大小，这个函数主要是因为在写入符号表时需要写入对应的Type，所以通过全局变量来记录当前声明变量的type。
- `write_IDtable()`函数的任务是将声明的变量信息写入符号表，其中变量的类型通过全局变量type获得，此次完成的编译器支持的功能只支持对应的种类为"v"，而对应的地址通过全局变量address获得，而每次需要将address加上当前变量的大小，可以通过type全局变量获取。
- `ASSIGN`函数的任务主要是完成赋值四元式的生成，取标识栈顶的两个元素，其中第一个元素为赋值操作数，第二个元素为赋值的结果变量，按照对应格式写入四元式。

- `GEQ`函数的任务是完成一般符号的四元式生成，根据传入的符号，取栈顶的前两个元素作为操作数，然后生成一个临时变量来储存结果，同时需要将临时变量写入符号表，临时变量的类型根据具体操作的属性来判断。
- `GEQ1`函数的任务为对应的1元操作数生成四元式。
- `push`将上一个lookahead压入标识符栈中。
- `IF`生成IF四元式。
- `EL`生成EL四元式。
- `IE`生成IE四元式。

中间代码设计为四元式，具体实现时通过声明一个新的`quart`类来作为四元式，其中包含4个属性分别为操作符、操作数1、操作数2、结果变量，均以string的格式进行存储，默认初始化为"_"，若对应元素没有则不进行更新。

目标代码生成

本次实验的目标代码采用mips汇编语言，目标代码生成的任务主要为将对应的中间代码四元式转换为目标代码mips指令语言。

本次实验中设计支持的部分包括**整数的表达式运算，大小比较，以及if else语句**的设计。

整个目标代码生成的过程为：

1. 将中间代码划分为基本块
2. 每次取一个基本块更新其中变量的活跃信息
3. 将对应基本块的中间四元式代码转换为目标代码
4. 释放除存储活跃变量之外的寄存器
5. 重复2-4步直到所有基本块被处理完毕

基本块划分

所以首先需要做的就是将中间代码四元式划分为对应的基本块，因为本次实验设计的编译器能够识别的语句中只有if语句会导致基本块的划分，所以根据四元式中`if,el,ie`操作符出现的位置划分为基本块即可。

通过一个`vector<int>`向量来储存对应的基本块的分界线，在通过循环遍历该vector中的值即可处理所有的基本块，在每个基本块结束时遍历RDL，将其中的不活跃变量重新写回内存来释放寄存器。在处理完所有的基本块后释放所有的寄存器内容，将其写回内存对应的位置，方便结果的查看。

活跃信息更新

每次获得基本块后，都需要从基本块最后一条四元式语句反向遍历整个基本块的代码，以此来更新对应的变量的活跃信息，更新变量活跃信息的规则如下所示：

1. 初始化时将定义的变量的状态都设定为`(y)`，而中间表达式产生的临时变量则初始化为`(n)`；
2. 反向遍历基本块，根据符号表中的状态信息更新四元式；
3. 更新表中的活跃状态信息，每当变量出现在结果位置时将表中状态信息更新为`(n)`，每当变量出现在操作数位置时将表中信息更新为当前四元式的序号，代表该变量下一次应用的位置；

更新了四元式信息后，接下来就遍历基本块中的四元式表达式来进行汇编代码的转化。

寄存器使用规则

mips汇编语言可以使用的寄存器为16个，标号为8-24，规定第24个寄存器为专用的寄存器，所以本次设计中可以自由使用的寄存器数量为15个，然后对应的内存位置使用Mars软件测试运行mips语句，发现对应的数据内存从8192开始，所以根据其符号表中的相对地址得到真实地址，对应的转换方式为$address = 8192 + IDtable[id].ADD$。通过两个vector，分别储存当前寄存器中的变量信息和变量的状态信息，初始状态为“ ”。

规定一个寻址函数`LOCR()`，遍历RDL，如果该变量出现在寄存器中，返回对应的寄存器编号，如果该变量未出现在寄存器中则返回该变量对应的内存地址。在转换代码时寄存器使用的规则遵循课件上的3个原则：

- 主动释放：如果该四元式某个操作数变量在寄存器中，则将该寄存器作为分配给结果变量的寄存器，如果该操作数仍未活跃变量则将其写回内存，如果不是活跃变量则不做处理。
- 选空闲者：遍历RDL寄存器，如果RDL中有空闲的寄存器则返回该寄存器作为分配的寄存器。
- 强迫释放：遍历RDL中变量的状态，选择不活跃的变量或者距离活跃点最远的变量所在的寄存器为分配的寄存器，然后如果为活跃变量还需要通过`sw/sb`指令将其写回自己对应的内存位置。

通过一个函数`set_res_rd`将对应寄存器的储存变量信息和变量状态信息都转化为结果变量。

当操作数如果不在寄存器中时，可以通过`load_id()`函数将对应的变量加载到寄存器中，并且根据需要将该寄存器的状态设置为结果变量或者是对应的变量。

目标代码转换

本次实验中只设计了对于整数和bool类型的运算操作，bool变量只能进行逻辑运算，否则根据设计会在语义分析阶段进行报错。

因为mips指令，对于寄存器之间的运算和寄存器与立即数的运算是进行区分的，所以对于不同的情况应该分开讨论。生成代码时将操作数为常数或者为变量进行分开的处理。其中对于大多数操作符来说，mips中都有对应的指令，只需要根据相关的指令进行转换即可，比如加法对应的`add，addi`减法对应的`sub`乘法对应的`mul`，在这里展示几种不同的操作数的变换。

第一种特殊的为减法操作和除法操作，因为在进行该操作时两个操作数的顺序是会对结果产生影响的，所以在选定第二个操作数作为结果寄存器时需要注意要保证原有的计算顺序，在进行减法计算时，当第一个操作数为常数第二个操作数为变量时，因为没有对应的指令，所以采用的方法为将第二个操作数的按位取反然后加一完成取负操作，接着通过`addi`指令将第一个操作数加上即可完成对应运算。

第二种特殊的为比较大小的操作，因为mips指令中与比较大小指令相关的指令多为跳转指令，所以需要通过跳转指令和标签的配合完成对应的判断大小关系的指令，首先将两个操作数进行sub指令得到两者的差，通过`bgtz/bgez/bltz/blez`等指令比较两操作数的差与0的关系来判断两者的关系，如果不满足则跳转到对应的false标签，将结果寄存器设为0，如果满足则继续执行将结果寄存器设为1，然后跳转到false标签对应块后面的true标签继续执行操作。

第三种特殊的为取负操作，根据mips指令将对应操作数通过`not`指令按位取反后，接着将其加1完成取负的操作。

第四种特殊的操作为if/else等跳转操作，需要通过一个栈来完成当发生递归if/else时的操作，每次出现if操作符的四元式，生成一个标签FJ_i将其作为跳转的目的地，然后将该标签入栈，通过`beq`比较操作数和0的关系来判断是否发生跳转。每次出现el操作符的四元式时，就取栈顶元素将其出栈，作为接下来else块的开始标签，对应当不满足if条件时进行else操作，同时为了让if操作块与else操作块分离，需要在声明标签之前，通过一个跳转指令跳转到else语句执行结束，表示if操作执行完成，通过声明另一个标签ie_j，将其作为if/else语句结束的表示，每次出现el操作符的四元式时，就将栈顶元素出栈并声明该标签，表示if/else语句结束。

3、实验测试

为了测试本次实验中设计的全部功能，设计的测试代码如下所示：

```
int a,b,c;
int x,y,z;
b = 3 * 2;
c = 2 + b;
a = (b+c)/2;
```

```
if((b<c)&&(a>5)){
    x = -a;
    if(-x>0){
        z = 10;
    }
}
else{
    y = a-b;
}
```

该代码比较了基本的算术表达式，以及嵌套的if/else语句以及条件判断语句和基本逻辑操作的正确执行，该代码执行结果应该为a=7，b=6，c=8，x=-7，z=10，y=0；

词法分析结果

运行编译器程序可以得到一个tokens.txt储存着词法分析的结果，上述代码的词法分析结果部分如下图所示：

tokens.txt

```
<int ,10>
<a ,0>
<, ,61>
<b ,0>
<, ,61>
<c ,0>
<, ,61>
<d ,0>
<; ,62>
<int ,10>
<x ,0>
<, ,61>
<y ,0>
<, ,61>
<z ,0>
<; ,62>
<b ,0>
<= ,49>
```

可以看到可以正确的识别到对应的词法部分，也能够正确区分减号和取负操作符为不同的界符。

语法分析结果

上述代码语法分析结果会返回正确，而如果语法分析判断正确的情况下会在控制台输出completely right的提示，对应上面的代码出现下面指令而且代码成功运行到结束，表示语法分析功能出现，为了验证完整性，测试在识别到不正确的语法时会出现报错，采用下列代码测试：

```
e=a+
```

```
PS D:\course\2\compiler\final_assignment> ./compiler.exe
read finish!
unmatch#with62
```

从结果中可以看出此时识别到了语法的错误。

语义分析结果（中间代码+符号表）

对应上述代码生成的四元式序列如下所示：

```
QT: 21
(*,3,2,t1)
(=,t1,_,b)
(+,2,b,t2)
(=,t2,_,c)
(+,b,c,t3)
(/,t3,2,t4)
(=,t4,_,a)
(<,b,c,t5)
(>,a,5,t6)
(&&,t5,t6,t7)
(if,t7,_,_)
(-,a,_,t8)
(=,t8,_,x)
(-,x,_,t9)
(>,t9,0,t10)
(if,t10,_,_)
(=,10,_,z)
(ie,_,_,_)
(el,_,_,_)
(=,a,_,y)
(ie,_,_,_)
```

可以观察到为正确的四元式序列，而此时对应的语义分析结束时的符号表中的信息如下表所示：

```
a v int 0 (y)
b v int 4 (y)
c v int 8 (y)
d v int 12 (y)
t1 v int 28 (n)
t10 v bool 64 (n)
t2 v int 32 (n)
t3 v int 36 (n)
t4 v int 40 (n)
t5 v bool 44 (n)
t6 v bool 48 (n)
t7 v bool 52 (n)
t8 v int 56 (n)
t9 v int 60 (n)
x v int 16 (y)
y v int 20 (y)
z v int 24 (y)
```

可以观察到正确储存了每个变量的属性，并且给每个变量在初始化时即分配了正确的相对地址。符号表系统的设计正确，同时可以在此时进行一些报错提醒，比如当赋值操作数左右段的操作数类型不同时：

```
PS D:\course\2\compiler\final_assignment> ./compiler.exe
read finish!
completely right
Types are different in assignment operation!
```

可以看到报错提醒赋值操作两端的种类不同，而当使用bool变量参与加减乘除运算时：

```
PS D:\course\2\compiler\final_assignment> ./compiler.exe
read finish!
completely right
bool variable can't + - * / !
```

会报错提醒bool变量不能进行加减乘除等基本表达式运算，而当使用整数操作进行与或非操作时：

目标代码生成

生成的目标代码会写进文件`mips.asm`并且生成在当前的文件夹下，对应上述的代码对应生成的mips代码文件(部分)如下所示：

mips.asm

```
.mips
addi $8, $0, 3
mul $8, $8, 2
sw $8, 8220
sw $8, 8196
addi $8, $8, 2
sw $8, 8224
sw $8, 8200
lw $9, 8196
add $8, $8, $9
sw $8, 8228
div $8, $8, 2
sw $8, 8232
sw $9, 8196
lw $10, 8200
sub $9, $9, $10
bgez $9,false0
addi $9, $0, 1
j true0
false0:
addi $9, $0, 0
true0:
sw $8, 8192
```

接下来在Mars中执行该代码并观察对应内存中的内容是否为预计的内容：

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+12)	Value (+16)	Value (+20)	Value (+24)	Value (+28)
8192	7	0	8	-7	0	10	6	8
8224	14	7	1	1	1	-7	7	1
8256	0	0	0	0	0	0	0	0
8288	0	0	0	0	0	0	0	0
8320	0	0	0	0	0	0	0	0
8352	0	0	0	0	0	0	0	0
8384	0	0	0	0	0	0	0	0
8416	0	0	0	0	0	0	0	0
8448	0	0	0	0	0	0	0	0
8480	0	0	0	0	0	0	0	0

0x00002000 (.data) Hexadecimal Addresses Hexadecimal Values ASCII

可以从中看出对应的内存位置第一行前6个分别对应a，b，c，x，y，z，可以从结果中看出其符合预期的结果，表示编译器的功能正确运行，能够正确地将源程序编译成目标程序并成功执行。

4、总结与心得

本次实验的全部源代码可以在（网址）上查看。

通过本次大作业完整的设计实现了一个简单的编译器，能够将源语言成功编译成汇编指令并成功运行。本次实验综合性极强，结合了本学期中学习的所有内容，在实验的过程中充满了挑战但同时也收获很多。

本次实验需要从文法设计开始完整的设计一个编译器，需要完成词法分析、语法分析、语义分析、目标代码生成等复杂部分，每个环节都紧密相关，通过该实验将一整个学期的理论课学习和实验课学习串联起来，实践能力进一步加强，同时也对课上的内容有了更深的理解。除此之外，通过本次实验回顾了有关汇编语言mips的内容，同时从前端到后端也更加熟悉了编译器每个子系统的功能以及设计技巧，大大

加强了代码的组织能力和编程能力。

但是本次实验中还是有一些遗憾，尽管词法分析器设计的功能比较全面，但随着一步步推进和时间因素的影响，最终实现的功能还是比较简单的赋值语句、算术表达式、if/else语句和简单的逻辑操作，像一些复杂功能并没有实现，比如说并不支持浮点数对应的目标指令的生成，以及while语句的实现，这些将在后续的学习中找时间进一步改进。

总而言之，本次实验基本完成了预期的目的和要求，为本学期的编译原理课程学习进行了全面的总结，同时也为后面的学习工作奠定了良好的基础。