一、实验题目

小型高级语言（如简化 C 语言）分析器。

二、实验内容及要求

设计并实现一个一遍扫描的编译前端，将简化高级语言的部分语法成分（含赋值语句、分支语句、循环语句等）翻译成四元式（或三地址代码），还要求有合理的语法出错报错和错误恢复功能。

三、实验环境

编译器：Apple clang version 11.0.0 (clang-1100.0.33.12)
编程语言：C++11

四、实验说明

(1) 词法成分说明

(i) 关键字：main、if、while

注：所有关键字都是小写。

(ii) 运算符和界符：{

"+","-","*","/","<","<=",">",
">=","==","!=","=",";",
"(",")","{","}","//","#"
}

(iii) 其他单词是**标识符**(VARIABLE)和**整型常数**(DIGIT)，通过以下正规式定义：

VARIABLE = letter (letter | digit)*

DIGIT = num num*

(iv) 空格由空白、制表符和换行符组成、空格一般用来分隔标识符、整型常数、运算符、界符和关键字，词法分析阶段通常被忽略。

(2) 各种单词符号对应的种别码

单词符号	种别码
main	1
if	2
while	3
+	4
-	5
*	6
/	7
=	8
<	9
<=	10
>	11
>=	12
==	13
!=	14
;	15
(	16
)	17
{	18
}	19
//	20
VARIABLE	21
DIGIT	22

(3) 上下文无关文法说明

(i) <程序> ::= main()<语句块>

(ii) <语句块> ::= ‘{ ‘ <语句串> ’ }’

(iii) <语句串> ::= <语句>; {<语句>;}

(iv) <语句> ::= <赋值语句> | <条件语句> | <循环语句>

(v) <赋值语句> ::= VARIABLE = <表达式>

(vi) <条件语句> ::= if (条件) <语句块>

(vii) <循环语句> ::= while (条件) <语句块>

(viii) <条件> ::= <表达式> <关系运算符> <表达式>

(ix) <表达式> ::= <项> { + <项> | - <项> }

(x) <项> ::= <因子> { * <因子> | / <因子> }

(xi) <因子> ::= VARIABLE | DIGIT | (<表达式>)

(xii) <关系运算符> ::= < | <= | > | >= | == | !=

(4) 程序结构描述

程序基于面向对象编程思想，创建了以下几个类来完成所需的功能。

word：该类描述了一个单词，通过新建该类的一个实例，可以通过类中的成员变量 type_id 和 type_num 分别获取该单词的类型（关键字、运算符、标识符或整型常数）和单词的种别码，若该单词属于非法单词（不符合标识符构成规则或不属于可识别的操作符）则可通过类中成员函数 is_legal()判断。

lexical_parser：该类描述了一个词法分析的工具，如果需要单独进行词法分析，则可以新建该类的一个实例，通过调用 start()函数来分析一段程序的词法。该类有记录错误信息的一个数组，每当检测到错误时，将错误信息加入到错误信息数组中，若分析完整段代码后有错误信息，则将错误信息打印出来。

syntactic_parser：该类描述了一个语法兼语义分析的工具，该类通过继承 lexical_parser 类，可以实现扫描一个单词后先后进行词法分析、语法分析，在扫描完一个语句后对语句行进语义分析。该类继承了 lexical_parser 类中的错误信息的数组，可以在分析过程中，将分析的代码中的词法错误、语法错误都加入到该数组中，若分析完整段代码后有错误信息，则将错误信息打印出来。

(5) 语法制导翻译的算法思想描述

语法制导翻译分为两个部分：语法分析和语义分析。

(i) 语法分析

本程序使用递归下降的语法分析。用 syntactic_parser 类继承 lexical_parser 类，调用继承过来的 scan 函数来一个个读取词语，读取的同时进行词法分析，分析完没有错误就再进行语法分析，有词法错误就报错然后返回一个带非法标志的单词。

语法分析使用简单的错误恢复方法，若缺少赋值号，则循环读取下一个单词直到遇到分号（语句结束符）或变量（语句开始符）。若循环结束前没有赋值号，则报错缺少赋值号，然后开始分析下一条语句；若匹配到赋值号，则将前面出现的单词报错，然后开始分析赋值号后面的表达式。缺少分号的错误分析方法同赋值号。

(ii) 语义分析

若一个赋值语句语法分析没有错误，则在语法分析的递归中将一个表达式中的元素（包括操作数、运算符和暂时存放运算结果的一个中间元素）加入到一个五元组中，最后再将表达式赋值号左边的标识符和赋值号右边最终的中间元素（如果没有复杂运算则是一个变量或数字）放入五元组中。

如果是条件判断语句，则再调用分析语句块的函数，分析出语句块中有几条中间代码，根据语句块中中间代码的数量推出条件判断跳转的中间代码，然后再将该跳转语句的中间代码插入中间代码元组（五元组）中。

五、测试结果与分析

(1) 代码没有错误的样例

(i)，嵌套条件语句加上简单赋值语句

待分析的代码：

```
main()
{
    if (a > b) {
        a = 2 + 3 * 4; x = (a + b)/c;
    }
    while (a > 0) {
        if (a > b) {
            a = 2 + 3 * 4; x = (a + b)/c;
        }
        x = 11 ; y = (x+1*y)+(1);
    }
}      You, 3 minutes ago • Uncommitted ch
```

输出：

词法分析结果：

```
Words:
(1,main) (16,() (17,)) (18,{) (2,if)
(16,() (21,a) (11,>) (21,b) (17,))
(18,{) (21,a) (8,=) (22,2) (4,+)
(22,3) (6,*) (22,4) (15,;) (21,x)
(8,=) (16,() (21,a) (4,+) (21,b)
(17,)) (7,/) (21,c) (15,;) (19,})
(3,while) (16,() (21,a) (11,>) (22,0)
(17,)) (18,{) (2,if) (16,() (21,a)
(11,>) (21,b) (17,)) (18,{) (21,a)
(8,=) (22,2) (4,+) (22,3) (6,*)
(22,4) (15,;) (21,x) (8,=) (16,()
(21,a) (4,+) (21,b) (17,)) (7,/)
(21,c) (15,;) (19,}) (21,x) (8,=)
(22,11) (15,;) (21,y) (8,=) (16,()
(21,x) (4,+) (22,1) (6,*) (21,y)
(17,)) (4,+) (16,() (22,1) (17,))
(15,;) (19,}) (19,})
```

中间代码：

```
Intermediate Code:
(1): if a > b goto (3)
(2): goto (9)
(3): T1 = 3 * 4
(4): T2 = 2 + T1
(5): a = T2
(6): T3 = a + b
(7): T4 = T3 / c
(8): x = T4
(9): if a > 0 goto (11)
(10): goto (25)
(11): if a > b goto (13)
(12): goto (19)
(13): T5 = 3 * 4
(14): T6 = 2 + T5
(15): a = T6
(16): T7 = a + b
(17): T8 = T7 / c
(18): x = T8
(19): x = 11
(20): T9 = 1 * y
(21): T10 = x + T9
(22): T11 = T10 + 1
(23): y = T11
(24): goto (9)
(25): halt
```

（ii）简单赋值语句

待分析的代码：

```
main()
{
    x = 11 ; y = (x+1*y)+(1);
}       You, a few seconds ago
```

输出：

```
Words:
(1,main) (16,() (17,)) (18,{) (21,x)
(8,=) (22,11) (15,;) (21,y) (8,=)
(16,() (21,x) (4,+) (22,1) (6,*)
(21,y) (17,)) (4,+) (16,() (22,1)
(17,)) (15,;) (19,})

success

Intermediate Code:
(1): x = 11
(2): T1 = 1 * y
(3): T2 = x + T1
(4): T3 = T2 + 1
(5): y = T3
(6): halt
```

(2) 代码有错误的样例

(i) 缺少赋值号的赋值语句

待分析的代码：

```
1  main()
2  {
3  x  11 ; y = (x+1*y)+(1);
4  }       You, a few seconds
```

输出：

```
Words:
(1,main) (16,() (17,)) (18,{) (21,x)
(22,11) (15,;) (21,y) (8,=) (16,()
(21,x) (4,+) (22,1) (6,*) (21,y)
(17,)) (4,+) (16,() (22,1) (17,))
(15,;) (19,})

ERROR:
Line 3: lack of '='

Intermediate Code:
(1): T1 = 1 * y
(2): T2 = x + T1
(3): T3 = T2 + 1
(4): y = T3
(5): halt
```

(ii) 赋值语句中多余符号

待分析的代码：

```
main()
{
x = 11 ; y = (x+1*y)();
}
```

输出：

```
Words:
(1,main) (16,() (17,)) (18,{) (21,x)
(8,=) (22,11) (15,;) (21,y) (8,=)
(16,() (21,x) (4,+) (22,1) (6,*)
(21,y) (17,)) (16,() (17,)) (15,;)
(19,})

ERROR:
Line 3: error symbol ()

Intermediate Code:
(1): x = 11
(2): T1 = 1 * y
(3): T2 = x + T1
(4): y = T2
(5): halt
```

(iii) 赋值语句中缺少项

待分析的代码：

```
main()
{
x = 11 ; y = );
}
```

输出：

```
Words:
(1,main) (16,() (17,)) (18,{) (21,x)
(8,=) (22,11) (15,;) (21,y) (8,=)
(17,)) (15,;) (19,})

ERROR:
Line 3: lack of factor
Line 3: error symbol )

Intermediate Code:
(1): x = 11
(2): halt
```

（iv）赋值语句中多余符号

待分析的代码：

```
main()
{
x = 11 ; y)) = (x+1*y)+(1);
}
```

输出：

```
Words:
(1,main) (16,() (17,)) (18,{) (21,x)
(8,=) (22,11) (15,;) (21,y) (17,))
(17,)) (8,=) (16,() (21,x) (4,+)
(22,1) (6,*) (21,y) (17,)) (4,+)
(16,() (22,1) (17,)) (15,;) (19,})

ERROR:
Line 3: error symbol ))

Intermediate Code:
(1): x = 11
(2): T1 = 1 * y
(3): T2 = x + T1
(4): T3 = T2 + 1
(5): y = T3
(6): halt
```

（3）错误分析方法

本程序中采用简单的错误恢复方法：若赋值语句中缺少赋值号，则循环读取下一个单词直到遇到分号（语句结束符）或变量（语句开始符）。若循环结束前没有赋值号，则报错缺少赋值号，然后开始分析下一条语句；若匹配到赋值号，则将前面出现的单词报错，然后开始分析赋值号后面的表达式。缺少分号的错误分析方法同赋值号。

在缺少其他符号时（缺少左括号右括号等情况），在特定情况下特别指出这些错误。由于代码中可能出现的语法错误千变万化，暂时还无法做到将所有不合理的错误都正确恢复，只能保证一些常见的合理的错误能够被恢复并正常继续分析。