OurC Project 1

For this project, you must write an interpreter that is capable of interpreting commands that are specified using the following syntax.

```
<Command>      ::= <Statement> ';' | <BooleanExp> ';' | <ArithExp> ';' | QUIT
<Statement>    ::= IDENT ':=' <ArithExp>
<BooleanExp>   ::= <ArithExp> ( '=' | '<>' | '>' | '<' | '>=' | '<=' ) <ArithExp>
<ArithExp>     ::= <Term> | <ArithExp> '+' <Term> | <ArithExp> '-' <Term>
<Term>         ::= <Factor> | <Term> '*' <Factor> | <Term> '/' <Factor>
<Factor>       ::= [ SIGN ] NUM | IDENT | '(' <ArithExp> ')'
```

where

QUIT is the word 'quit', an **IDENT** starts with a letter and is followed by digits or letters or underlines (however, an **IDENT** cannot be the word 'quit'), a **NUM** is either an integer or a float number (e.g., 35, 7, 43.8, 43., .35, 1.0 and 07) and **SIGN** is either '+' or '-'.

Apart from the above syntax for "the right input", there are also line-comments. A line-comment is any text that starts with '//' and ends with '\n'. Your program should always skip line-comments when they appear. Be aware though. Your program should only skip a line-comment and NOT "the line containing the line-comment". If there is any input that appears on the same line PRECEDEING the line-comment, your program should still read-in and process that input.

Finally, your program should terminate when the current input command is 'quit'.

It may be desirable to implement a recursive descent parser in writing your program. Therefore, a translation of the above grammar is given below. This translation ought to be "usable" in producing a recursive descent parser. However, there is no guarantee! You yourself bear the ultimate responsibility of producing a "right" translation for the above grammar (one that can be used in producing a recursive descent parser).

```
<Command> ::= IDENT ( ':=' <ArithExp> | <IDlessArithExpOrBexp> ) ';'
              | <NOT_IDStartArithExpOrBexp> ';'
              | QUIT

<IDlessArithExpOrBexp>  ::= {   '+' <Term>   | '-' <Term>
                              | '*' <Factor> | '/' <Factor>
                            }
                            [ <BooleanOperator> <ArithExp> ]

<BooleanOprator>        ::= '=' | '<>' | '>' | '<' | '>=' | '<='

<NOT_ID_StartArithExpOrBexp>
                        ::= <NOT_ID_StartArithExp>
                            [ <BooleanOperator> <ArithExp> ]

<NOT_ID_StartArithExp>  ::= <NOT_ID_StartTerm> { '+' <Term> | '-' <Term> }
<NOT_ID_StartTerm>      ::= <NOT_ID_StartFactor> { '*' <Factor> | '/' <Factor> }
<NOT_ID_StartFactor>    ::= [ SIGN ] NUM | '(' <ArithExp> ')'
<ArithExp>              ::= <Term> { '+' <Term> | '-' <Term> }
<Term>                  ::= <Factor> { '*' <Factor> | '/' <Factor> }
<Factor>                ::= IDENT | [ SIGN ] NUM | '(' <ArithExp> ')'
```

Input/Output description

Your program should be capable of doing interactive I/O. However, there is a very strict requirement on the kind of output your program should produce. Below is a sample of what your program should do. This sample is intended so that you can get a concrete feeling of what the I/O of your program should be. If there is any question about "the right I/O behavior", please post your questions on the BBS (PL-board).

Note that '>' is a prompt produced by your program. There should be a space behind '>'.

例一：

```
Program starts...
> 2+3;  // the simplest form of commands
5
> 2     +3 // a line-comment here ; useless "input" here :  5+8;
  ;       // another line-comment ;;; ('5+8;' and ';;;' should be ignored)
5
> 2
+ 3


  // Hello!  Hello! Can you do 7 + 8 ?

      ; // your program should always skip white spaces
5
>
   2   + 3

     ;      1  + 2 // no input is "got" until there is a line-enter
5
> // once a command such as '2+3;' is read in, the system
  // immediately gives a response ;
  // but then, the next command '1+2;' is already "partially read in" ;
  ;
3
> 2 + $$ - 5
Unrecognized token with first char : '$'
> 2 + * + 5 + 8
Unexpected token : '*'
> // once an input error (unrecognized token or unexpected token) is
  // encountered, the remaining input on the same
  // line is ignored ; input-processing will resume for the next line
2
    +
    3
;
```

```
5
> abc := ( 20 * 5 ) + 1 ;
101
> abc * 2 ;
202
> bcd * 2 ;
Undefined identifier : 'bcd'
> bcd := 1
;
1
> bcd * 2 ;
2
> bcd := bcd + 10 ;
11
> e := bcd

 ; e := e + 3 ;
11
> 14
> e > bcd ;
true
> e < bcd ;
false
> quit
Program exits...
```

There are three kinds of errors : error on the token level, error on the syntax level, error on the semantics level

Unrecognized token with first char : '$' // lexical error (error on the token level) -- 第一道防線
Unexpected token : '*' // syntactic error (token recognized) -- 第二道防線
Undefined identifier : 'bcd' // semantic error (grammar OK) -- 第三道防線

注意(本 project)抓 error 的**順序**如下：

第一防線： Unrecognized token with first char

第二防線(token recognized, parse grammar)： Unexpected token

第三防線(grammar OK, evaluate it)：Undefined identifier

Of course, there may be errors that are neither "unrecognized token error" nor "syntax error" nor "undefined ID error". When there is such an error (e.g., division by zero), your program should just output 'Error'. E.g.,

```
...
> 3/0
Error
> a:bc
```

Unrecognized token with first char : ':'
// 'a'是沒問題的, 問題發生於':'，是個 unrecognized token。
// 對 GetToken()而言，只有「Unrecognized token with first char」，沒有「Error」。

Due to precision considerations, we basically do not put any floating point numbers in the input. However, there can be '/' operations, which can cause floating point numbers to be computed. When your program is to compare two floating point numbers, remember to use a tolerance of 0.0001. e.g.,

(Again, tolerance for floats should be : 0.0001)

例二：

```
Program starts...
> ( 1.0 / 100000 ) = ( 1.0/10000000 ) ;  // difference is within 0.0001
true
> ( 1.0 / 100000 ) > ( 1.0/10000000 ) ;
false
> ( 1.0 / 100 ) = ( 1.0/1000000 ) ;  // difference is more than 0.0001
false
> ( 1.0 / 100 ) > ( 1.0/1000000 ) ;
true
> quit
Program exits...
```

例三： // more involved cases

```
Program starts...
> salary := 3000 ;
3000
> monthsPerYear := 12 ;
12
> income := salary * monthsPerYear ;
36000
> income * 10 ;
360000
> income * 10 > 50000 ;
true
> income * 10 > 500000 ;
false
> salary := 30000 ;
30000
> ( salary * monthsPerYear * 10 ) > 500000 ;
true
> ( salary * ( monthsPerYear - 10 ) + 20000 ) * 10 >
  ( 30000  * 2 ) * 10              + 2 * 10000 * 10
  ;
false
```

```
> ( 20000 + salary * ( monthsPerYear - 10 ) ) * 10 =
  2 * 10000 * 10 +            ( 30000  * 2 ) * 10
  ;
true
> quit
Program exits...
```

Actually, your program will be compiled and run under Linux. The input will be a file, and your program's output will be "collected" in a (different) file too.

The very first input "entry" of every input-file will be a so-called "test number". Your program should be designed in such a way that it first reads in an integer and stores it in a file-scope variable such as 'uTestNum'. Only when it has read in this "test number" should your program start to do "normal I/O".

The reason for putting a "test number" in every input file is so that you can make good use of this "feature" for debugging. (ref. : "與 bug 共舞之 run-time debug 小技巧教學" - BBS PL 版之置底文)

Therefore, the actual input and output for running your program is something that looks like the following.

例一：

```
// input 自此始，但不包括此行
1
salary := 3000 ;
monthsPerYear := 12 ;
income := salary * monthsPerYear ;
income * 10 ;
income * 10 > 50000 ;
income * 10 > 500000 ;
salary := 30000 ;
( salary * monthsPerYear * 10 ) > 500000 ;
( salary * ( monthsPerYear - 10 ) + 20000 ) * 10 >
  ( 30000  * 2 ) * 10             + 2 * 10000 * 10
  ;
( 20000 + salary * ( monthsPerYear - 10 ) ) * 10 =
  2 * 10000 * 10 +            ( 30000  * 2 ) * 10
  ;
quit
1+2;
hello, can you hear me?
// input 至此止，但不包括此 comment

// output 自此始，但不包括此行
Program starts...
> 3000
> 12
> 36000
```

```
> 360000
> true
> false
> 30000
> true
> false
> true
> Program exits...
// output 至此止，但不包括此 comment，以上各行之後無 space，但有 line-enter
```

例二：

```
// input 自此始，但不包括此行
2
( 1 / 100000 ) = ( 1/10000000 ) ;  // difference is within 0.0001
( 1 / 100000 ) > ( 1/10000000 ) ;
( 1 / 100 ) = ( 1/1000000 ) ;  // difference is more than 0.0001
( 1 / 100 ) > ( 1/1000000 ) ;
quit
To be or not to be?  That is the question.
// input 至此止，但不包括此 comment

// output 自此始，但不包括此行
Program starts...
> true
> false
> false
> true
> Program exits...
// output 至此止，但不包括此 comment，以上各行之後無 space，但有 line-enter
```

例三：

```
// input 自此始，但不包括此行
3
2+3;  // the simplest form of commands
2     +3 // a line-comment here ; useless "input" here :  5+8;
  ;       // another line-comment ;;; ('5+8;' and ';;;' should be ignored)
2
+ 3


  // Hello!  Hello! Can you do 7 + 8 ?

      ; // your program should always skip white spaces
```

```
  2   + 3

    ;      1  + 2 // no input is "got" until there is a line-enter
// once a command '2+3;' is read in, the system immediately gives an response ;
; // but then, the next command '1+2' is already "in process" ;
2 + $$ - 5
2 + * + 5 + 8
// once an input error is encountered, the remaining input on the same
  // line is ignored ; input-processing will resume for the next line
2
   +
   3
;
abc := ( 20 * 5 ) + 1 ;
abc * 2 ;
bcd * 2 ;
bcd := 1
;
bcd * 2
bcd := bcd + 10 ;
e := bcd

  ; e := e + 3 ;
e > bcd ;
e < bcd ;
quit
// input 至此止，但不包括此 comment

// output 自此始，但不包括此行
Program starts...
> 5
> 5
> 5
> 5
> 3
> Unrecognized token with first char : '$'
> Unexpected token : '*'
> 5
> 101
> 202
> Undefined identifier : 'bcd'
> 1
> 2
> 11
> 11
> 14
> true
```

> false
> Program exits...
// output 至此止，但不包括此 comment，以上各行之後無 space，但有 line-enter

For each input (test data) file, your program must produce an output file with a content that is EXACTLY THE SAME AS the expected output. This is the only way your program can "pass" any particular test.

For those that are interested, here are the commands that are used to compile and run your program on Linux.

```
g++ -Wno-deprecated -o program.out program.cpp
program.out < inputFile1
```

For your reference, here is a description of the underlying "philosophy" for testing your program. It is stated here so that you have a general idea regarding what you should do in doing your project.

In general, we will use N test data to test your project. These N test data can be arranged in such a way so that test data 2 is more difficult than test data 1, test data 3 is more difficult than test data 2, etc. Therefore, for these N test data, we will prepare N problems for you to do. You can use the same program for doing all N problems. You can also use different programs for doing different problems.

Each problem is designed as follows. (Let us call this problem "Problem No. M".) Problem No. M is for testing test-data No. M. However, there will be 3 test cases for Problem No. M, and one of these test cases will be hidden from you (you will not be able to see what the hidden data is). When you do Problem No. M, you are supposed to get a concrete feeling of what the hidden test case is by looking at the 2 test cases that the CAL system shows to you. It may happen that one of these 2 test cases is just a "trivial test case". If this happens, it is just because there is no need to use both test cases to let you know about what is to be tested in the hidden test case.

Each test data will correspond to two problems. The first problem is such that its test case 3 is "isomorphic to" test case 2. (Isomorphic df= number change and/or operator change ; e.g., 2+3 is isomorphic to 45*87). The second problem is that its test case 3 is "more or less similar to" test case 2 (perhaps test case 1 too).

Therefore, for example, Problem No. 1 and Problem No. 2 are actually intended for testing similar test data. In Problem No. 1, test case 3 is isomorphic to test case 2. In Problem No. 2, test case 3 is "somewhat similar to" test case 2.

We suggest that you do Problem No. 1 first, and then you do Problem No. 2, and then you do Problem No. 3, etc. However, this is only a suggestion. You are totally free in choosing the problem you want to do. (That is, you can do the last problem first if you want). The more problem you get it done, the more score you will get.

==========

OurC Proj1 補充說明

==========

如果 command 的第一個 token 是'quit', then "Program exits..."。

如果 command 的第一個 token 是(非'quit'的)ID

Case 1 : 此 token 之後是':='

此 ID 不須被 define 過，而文法至':='為止也沒問題

Case 2 : 此 token 之後不是':='

(a) 此 token(即 ID)之後是合於文法的 token (e.g., '+'或'-')

此 token(即 ID)必須被 define 過

(b) 此 token 之後是不合於文法的 token (e.g., '('或')')

錯誤出在「此 token(即 ID)之後的(不合於文法的)token」(e.g., '('或')')

When the token is an ID ...

Case 1 : ID is the very first token

Do according to what is stated above.

Case 2 : ID is NOT the very first token

if ID 的出現合於文法 (i.e., it is expected)

then check whether it is defined (if NOT defined then "Undefined")

else // ID 的出現不合於文法 (i.e., it is unexpected)

"Unexpected token"

ERROR 的優先順序：1.Unrecognized 2.Unexpected 3. Undefined.

依照 recursive descent parsing 過程中碰到 error 的順序，以第一個碰到的 error 為 error。

例：a + b c ;

先碰'a' : 無法確定是 error

再碰'+' : 此時要檢查'a'是否已 define

所以問題在'a'。

例：1 @ # ;

先碰'1' : OK

之後'@' : token level error (Unrecognized token with first char)

瑣碎事項：

1. 'quit' 必須出現於 cmd 的一開始才是 QUIT，在其他地方都只是個 ID

但卻勢必只能是個未定義過的 ID。

例：

```
> a:=3+(quit*5);
Undefined identifier : 'quit'
> quit:=5;
Program exits...
```

2. Project 1 不會突然出現 EOF，EOF 一定是在(有效的)'quit'之後。

但 Project 2 就不一定能如此保證了。

3. 有可能有 3.4.5, 而 unexpected token 應該是'.5'

(我們接受 3.與.5 這種 NUM，它是 token，只是未見得是 expected)

例：

```
> a:=3.4.5+(4*5);
Unexpected token : '.5'
```

4.你必須"沿路檢查 error"，error 一出現，此行(與之前尚未處理之行)就作廢。error 下面的行要算新 input

// Please bear in mind that what you are writing IS SUPPOSED TO BE an INTERACTIVE program. It is just
// somewhat unfortunate that your program has to be tested using some input file.

例：

```
> 3.4.5            // user intention: this line
Unexpected token : '.5'
> + 5;               // and this line constitute ONE cmd
5
```

5. float 印出來時是取三位小數

例：

```
> 3.0;
3.000
> 31.03125; // use something like printf( "%1.3f", a ) to print
31.031
> 31.0625;
31.063
```

6. int 與 float operate 結果是 float
(Project 1 不測 boolean 與 int or float 的+-*/)

例：

```
> 3.75+5;
8.750
```

7. variable 的型別可以變更

例：

```
> a:=5;
5
> a:=5.0;
5.000
```

其他：

```
> a:=5;
5
> a+3;
8
> (a+3)*5;
```

```
40
> a;
5
> a:=5.0;
5.000
> a;
5.000
> -3;
-3
> -a; // According to the grammar, only NUM can follow SIGN
Unexpected token : 'a'
```