Scanner

寫 GetToken()的注意事項

＊ 一定要一次 get 一個 char！ 千萬不要一次 get 一個 string // when getting from stdin

1. 用一個 function (姑且稱之為 F1)來「get the next char」
   此 function 要負責 keep track of
   所 get 到的 char 的 line number and column number

2. 用一個 function (姑且稱之為 F2)來「get the next non-white-space char」
   此 function 要負責「跳過 comment」
   (F2 當然該呼叫 F1 whenever F2 wants to get "the next char")

3. 呼叫 F2 以得 the next non-white-space char
   此 char 便為「next token」之始
   現在檢查此 char、判斷此「next token」是三種 case 的哪種 cas
   如果是 case 1，就叫 F3 去把「next token」"剩下的部份"去讀進來。
   如果是 case 2，就叫 F4 去把「next token」"剩下的部份"去讀進來。
   如果是 case 3，就叫 F5 去把「next token」"剩下的部份"去讀進來。

現在我們已得到此「next token」的全部了。

a. Longest match principle
b. read from stdin or vector // be aware of C vector // be aware of PAL stylecheck
c. PeekToken() vs. GetToken()

Parser -
Syntax checking first (including lexical/syntactical error detection),
evaluation later (including var. declaration)

```
/*
A recursive descend parsing algorithm.

 Original syntax :

 <BooleanExp> ::= <Exp> = <Exp>
 <Exp>        ::= <term> | <Exp> + <term> | <Exp> - <term>
 <term>       ::= <factor> | <term> * <factor> | <term> / <factor>
 <factor>     ::= NUM | IDENT | (<Exp>)

 Rewrite the original syntax (after left factoring and elimination of left recursion) :

 <BooleanExp> ::= <Exp> = <Exp>
 <Exp>        ::= <term> {+ <term> | - <term>}
 <term>       ::= <factor> {* <factor> | / <factor>}
 <factor>     ::= NUM | IDENT | (<Exp>)

Note:

You should have a scanner (lexical analyzer; GetToken()) that always returns either EOF
or NUM or IDENT or EQUAL or PLUS or ...; If it returns NUM or IDENT, then it should also
return the corresponding value (in the case of NUM) or symbol (in the case of IDENT).
What GetToken() does is that it starts from the current input, skips "white space
```

```
characters" and get the next number or identifier (if there is one). GetToken() returns
an "EOF" if there is no next input token.

You should also have a PeekToken() which only "takes a peek at" the input token but does
not "get" the token.
*/

void BooleanExp(var Bool correct)
//  <BooleanExp> ::= <Exp> = <Exp>
{
  TOLERANCE = 0.01;

  Exp(exp1Correct, exp1Value);
  if not exp1Correct
    then { correct := false; return; }

  GetToken(tokenType, tokenValue);
  if tokenType <> EQUAL
    then { correct := false; return; }

  Exp(exp2Correct, exp2Value);
  if not exp2Correct
    then { correct := false; return; }

  GetToken(tokenType, tokenValue);
  if tokenType <> EOF
    then ( correct := false; return; }

  // we do have <exp>=<exp> in the input

  if exp1Value > exp2Value
    then {
      larger := exp1Value;
      smaller := exp2Value;
    }
    else {
      larger := exp2Value;
      smaller := exp1Value;
    }

  if (larger * (1.0-TOLERANCE) <= smaller)
    then correct := true
    else correct := false;

} // BooleanExp()

void Exp(var Bool correct, var float value)
// <Exp>        ::= <term> {+ <term> | - <term>}
{
  Term(term1Correct, term1Value);
  if not term1Correct
    then { correct := false; value := 0.0; return; }

  do {

    PeekToken(tokenType, tokenValue);
    if (tokenType = EOF) or (not tokenType in [PLUS, MINUS])
      then { correct := true; value := term1Value; return; }
```

```
    // there is '+' or '-' behind the first term

    GetToken(tokenType, tokenValue);  // tokenType : PLUS or MINUS

    Term(term2Correct, term2Value);

    if not term2Correct
      then { correct := false; value := 0.0; return; }

    // second term ok.

    correct := true;

    if tokenType = PLUS
      then
        term1Value := term1Value + term2Value;
      else
        term1Value := term1Value - term2Value;

  } while TRUE;

} // Exp()

void Term(var Bool correct, var float value)
// <term>       ::= <factor> {* <factor> | / <factor>}
{
  Factor(factor1Correct, factor1Value);
  if not factor1Correct
    then { correct := false; value := 0.0; return; }

  do {

    PeekToken(tokenType, tokenValue);
    if (tokenType = EOF) or (not tokenType in [MULTIPLICATION, DIVISION])
      then { correct := true; value := factor1Value; return; }

    // there is '*' or '/' behind the first factor

    GetToken(tokenType, tokenValue);  // tokenType : MULTIPLICATION or DIVISION

    Factor(factor2Correct, factor2Value);

    if not factor2Correct
      then { correct := false; value := 0.0; return; }

    // second factor ok.

    correct := true;

    if tokenType = MULTIPLICATION
      then
        factor1Value := factor1Value * factor2Value;
      else
        factor1Value := factor1Value / factor2Value;

  } while TRUE;
```

```
} // Term()

void Factor(var Bool correct, var float value)
// <factor>     ::= NUM | IDENT | (<Exp>)
{
  GetToken(tokenType, tokenValue);

  if not tokenType in [NUM, IDENT, LEFT_PAREN]
    then { correct := false; value := 0.0; return; }

  if tokenType = NUM
    then { correct := true; value := tokenValue; return; }

  else if tokenType = IDENT
    then { correct := true; value := tokenValue ?????; return; }

  else { // tokenType = LEFT_PAREN

    Exp(expCorrect, expValue);

    if not expCorrect
      then { correct := fale; value := 0.0; return; }
      else { // expCorrect; but still neet to check RIGHT_PAREN

        GetToken(tokenType, tokenValue);

        if tokenType <> RIGHT_PAREN
          then { correct := false; value := 0.0; return; }
          else {
            correct := true; value := expValue;
            return;
          } // tokenType = RIGHT_PAREN

      } // expCorrect

  } // tokenType = LEFT_PAREN

} // Factor()
```