Recursive Descent Parsing - Intro.

2016-07-18 初稿
2022-02-01 修訂
2026-02-25 修訂

本章要做的是：把輸入字串（source code）分兩步處理。
(1) Scanner：把字元串切成 token 串（像 NUM、IDENT、PLUS…）。
(2) Parser：依據文法規則，用 token 串建立/檢查語法結構（parse tree），同時做語法錯誤偵測。其中 PeekToken() 用來「只偷看不消耗」，幫助 parser 在分支處做決策。

Scanner

負責將 input 字串轉成 token 串

寫 GetToken()的注意事項：

Scanner 建議一次讀一個字元（char），原因是你需要精準掌握：

- token 的邊界（什麼時候結束）
- 行號/列號（錯誤訊息要指到正確位置）。

若一次讀一段字串，會讓「往回/偷看」與位置追蹤變得困難。

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 case，
   next token 常見可分成三類：
   (1) Identifier/Keyword：以英文字母或底線開頭（例：abc, while）
   (2) Number：以數字開頭（例：123, 3.14）
   (3) Operator/Delimiter：符號（例：+, ==, (, ;）
   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

- 從起始符號（start symbol）開始，依文法規則一步步展開，檢查 token 串是否能被文法接受。
- 若把整個推導過程畫成一棵 parse tree，那麼樹葉會是 terminal（也就是 token）。
- 由左到右讀取這些樹葉，會得到原本的 token 串。

（實作上不一定要真的建樹；很多 recursive descent parser 只做「語法檢查與對應動作」也可以。）

Syntax checking first (including lexical/syntactical error detection), evaluation later (including var. declaration)
例如 x = 1 + ;：token 可以切出來，但語法不合法（少了一個 term），所以屬於 syntax error。
例如 y = x + 1;：語法可能合法，但若 x 根本沒宣告，這是 semantic error（語意問題），通常在語法通過後才處理。

```
/*
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>)
```

Recursive descent 不能直接處理 left recursion（例如 <Exp> ::= <Exp> + <term>），否則函式會無限遞迴。所以我們要先把文法改寫成「非左遞迴」且「可用 one-token lookahead 決策」的形式。

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

Note:
Scanner（GetToken）每次回傳兩個資訊：
•   tokenType：種類（NUM / IDENT / PLUS / …）
•   tokenValue：內容（若是 NUM，回傳數值；若是 IDENT，回傳字串 lexeme）。若沒有下一個 token，回傳 EOF。PeekToken() 只看下一個 token，不前進輸入位置。

*/

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()
```