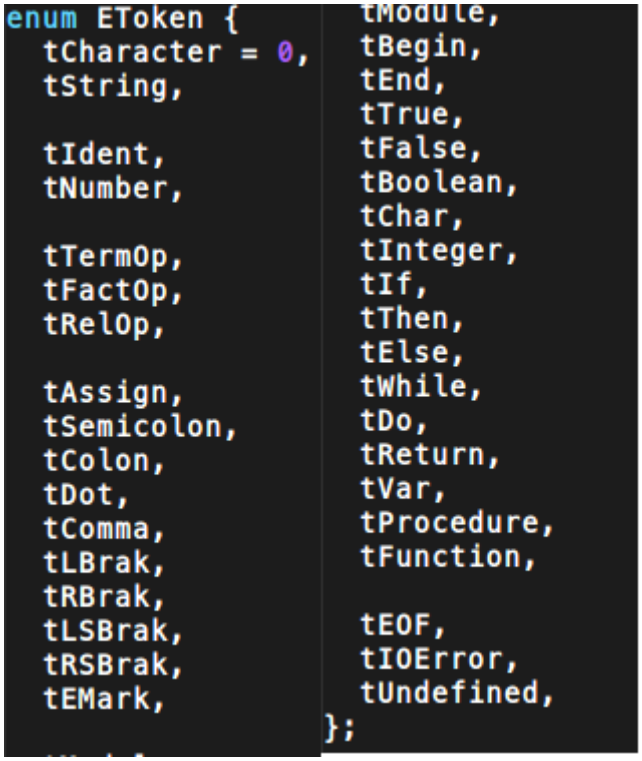
Compilers Project 1 'Scanner' Report

2013-11431 Hyunjin Jeong

The first project was to make a scanner for the SnuPL/1. The scanner accepts an input file string, and generates tokens for a parser. In this project, many parts of the scanner were already implemented in the skeleton code. Implementing the scanner is largely separated into two parts: 1. Defining token types, 2. Scanning an input string.

1. Defining token types

From the EBNF syntax definition of SnuPL/1, I defined token types. There are my token types in the following figure.



```
enum EToken {
  tCharacter = 0,
  tString,

  tIdent,
  tNumber,

  tTermOp,
  tFactOp,
  tRelOp,

  tAssign,
  tSemicolon,
  tColon,
  tDot,
  tComma,
  tLBrak,
  tRBrak,
  tLSBrak,
  tRSBrak,
  tEMark,

  tModule,
  tBegin,
  tEnd,
  tTrue,
  tFalse,
  tBoolean,
  tChar,
  tInteger,
  tIf,
  tThen,
  tElse,
  tWhile,
  tDo,
  tReturn,
  tVar,
  tProcedure,
  tFunction,

  tEOF,
  tIOError,
  tUndefined,
};
```

'tCharacter' and 'tString' are character and string types in the EBNF syntax definition. 'tIdent' and 'tNumber' are ident and number types. 'tTermOp', 'tFactOp', and 'tRelOp' are termOp, factOp, relOp types. 'tAssign' - 'tEMark' are symbols embraced by double quotation marks except 'tTermOp', 'tFactOp', and 'tRelOp'. 'tModule' – 'tFunction' are words embraced by double quotation marks in the syntax definition, and they are also called reserved keywords. 'tEOF' – 'tUndefined' are used for an error case.

2. Scanning an input string

After defining tokens, next step was generating tokens by scanning an input file string. Most cases were straight-forward to implement. The scanner just sees characters and compares them to expected ones. I will only explain how I implemented complicated cases such as 'tCharacter', 'tString', and consuming comments.

In the 'tCharacter' case, a character is valid if it seems like 'c', and c is a printable ASCII character or a valid escape character. A length of character should be one. If input is undefined escaped character like '₩p', then '₩p' is handled as '₩₩' + 'p'. So the length of '₩p' is 2, and the scanner will return tUndefined(₩'₩₩₩p₩'). Single quotation marks will be removed from token value in valid cases. In my implementation, even in error case like length is bigger than two, a scanner reads all characters until the second single quotation mark (without escape) appears. In example case 'abcd₩'edfgh', token type will be tUndefined and token value will be 'abcd₩'edfgh'. If the second single quotation mark didn't appear until EOF, all characters after the first single quotation mark will be included into the token value of tUndefined token. For example, 'abcdeEOF will be tUndefined(₩'abcde).

My implementation of 'tString' is basically identical with 'tCharacter' implementation except the scanner doesn't check the length of characters. Unlike in case 'tCharacter', If input is an undefined escaped character like '₩p', the scanner will return tString(₩₩p) because the scanner doesn't check the length of input characters. Error case handling schemes are similar, too. The scanner reads all characters until the second double quotation mark (without escape) appears. An example "abcd₩'edfgh" will be tString(abcd'edfgh) and "abcdeEOF will be tUndefined(₩"abcde).

The following figure shows my consuming comments implementation.

```
361   // Consume comments.
362   if(c == '/') {
363     while(_in->peek() == '/') {
364       while(_in->peek() != '\n') GetChar();
365       while(IsWhite(_in->peek())) GetChar();
366
367       RecordStreamPosition();
368       c = GetChar();
369       if(c != '/') {
370         if(c == EOF) return NewToken(tEOF);
371         break;
372       }
373     }
374   }
```

The while-loop in line 363-373 is used for consuming consecutive comments. The while-loop in line 364 consumes one line, and the while-loop in line 365 consumes white spaces (new-lines, tabs, spaces). In the end of line 365, a single line comment is consumed. After that, the scanner checks the first character of next line, c. If c and its next character are both '/', then another comment line will be consumed. Otherwise the consuming task is done.