Assignment 1: Scanner

COMP3131/9102: Programming Languages and Compilers
Term 1, 2025

Worth: 12/100 marks **Due: 8:00pm Saturday 1 March 2025**

Revision Log

Nothing at this stage.

End Of Revision Log

0. FAQs

Please read the FAQs for this assignment.

1. Specification

You are to implement a scanner for VC programs manually (i.e., without using tools like JFlex or CUP). Your scanner should read characters from a VC program and produce a stream of tokens (i.e., perform lexical analysis).

After developing the scanner manually and understanding its underlying principles (as discussed in class), you will be well-prepared to build scanners using generator tools such as JLex or JavaCC.

Review the VC Language Definition carefully to learn about the the token set, its rules for forming identifiers and literals (integer, floating-point, boolean and string), including the handling of allowed escape characters (in strings).

You do not need to understand the VC grammar (to be introduced later) to complete this assignment.

The scanner operates as a subroutine called by the parser when it needs a new token in the input stream. Your scanner, when called each time, must always return the *longest possible match* in the remaining input that can be interpreted as a token.

A few examples should clarify this point:

```
INPUT                                     OUTPUT                                  COMMENTS
-------------------------------------------------------------------------------------------
<=          "<="                                                                  [ 1 token  ]
//          an end-of-line comment rather than 2 tokens "/" and "/"
1.2+2       "1.2" (float-literal),  "+" (+), "2" (int-literal)                    [ 3 tokens ]
1.2e+2      "1.2e+2"                                                              [ 1 token  ]
1.2e+ 2     "1.2" (float-literal), "e" (id), "+" (+), "2" (int-literal) [ 4 tokens ]
#           an error token                                                        [ 1 token  ]
```

Your scanner should discard all whitespace and comments found in the program.

You must return each token as an object of the class `Token` defined in the file `Token.java` and explained below.

2. Setup Your Environment

You can complete this and all subsequent assignments on any CSE machine with a Java compiler and interpreter. If you prefer working on your own computer, consider installing the Java SDK, which includes Java Swing required by our TreeDrawer package.

To prepare for this and future assignments, create a directory named `VC` with the following subdirectories:

```
                Scanner
              /  Recogniser
            .    Parser
          / .    Checker
         /  .    CodeGen
$HOME --- VC ---  ASTs
          \  .    TreeDrawer
           \ .    TreePrinter
            \.    UnParser
              \   lang
```

Each component of the VC compiler you develop this term will reside in its own package under the main package `VC`. Note that the last five components are provided for you.

3. Writing Your Scanner

1. Log in to one of the school's login computers using your CSE account:

   ```
   % ssh your-account-name@login.cse.unsw.edu.au
   ```

2. Set VC as your current working directory:

   ```
   % cd your-VC-directory
   ```

3. Copy ~cs3131/VC/Scanner/Scanner.zip to your VC directory:

   ```
   % cp ~cs3131/VC/Scanner/Scanner.zip .
   ```

4. unzip Scanner.zip

   ```
   % unzip Scanner.zip
   ```

You will find the following files unzipped:

```
================================================================================
VC
================================================================================
vc.java              The main program for the scanner
ErrorReporter.java:  a simple Error handling class

================================================================================
VC/Scanner
================================================================================

Scanner.java:        a skeleton of scanner (to be completed by you)
Token.java:          definitions for token kinds, keywords and a
                     method for classifying some "ids" into keywords
SourceFile.java:     source file handling
SourcePosition.java: positions for tokens in the program

Test Files:
                     tokens.vc tab.vc longestmatch.vc string.vc fib.vc
                     comment[1234].vc error[1234].vc

Solution Files:
                     tokens.sol tab.sol longestmatch.sol string.sol fib.sol
                     comment[1234].sol error[1234].sol
```

You must not modify the files `Token.java` and `SourcePosition.java`. Once completed, `Scanner.java` will contain the scanner for this assignment.

You can immediately compile the Java files and run the scanner on a file containing the single "(". The output will be:

```
======= The VC compiler =======
Kind = 27 [(], spelling = "", position = 0(0)..0(0)
Kind = 38 [], spelling = "", position = 0(0)..0(0)
Kind = 39 [$], spelling = "$", position = 0(0)..0(0)
```

It is clear why this output appears: the test file contains two characters—a "(" and a newline (triggering an error in this incomplete implementation)—followed by the end-of-file marker. You are to complete `Scanner.java` to build a working scanner for VC, which consists of about 300 lines of code.

Note: This partial scanner implementation is provided to help you build your scanner for this assignment. You may disregard the starter code in `Scanner.java` as long as you preserve the public interface. You are not constrained by the structure of the starter code.

As expected, this skeleton scanner does not produce yet the information as required.

In the file `Token.java`, the class `Token` is defined so that every token must be represented as an object of this class. The constructor of this class expects three kinds of information for a token:

- its kind represented as an int (e.g., Token.ID, Token.PLUS)
- its spelling represented as a string (i.e., "sum", "+")
- its position in the program represented as an object of the class `SourcePosition`.

It is simple to distinguish between identifiers and keywords. Initially you can consider all keywords as identifiers. As soon as you create an object of the class `Token` for an "identifier", the constructor of `Token` will check to see if this particular identifier is one of the keywords and return the appropriate kind of token accordingly.

In the file `SourcePosition.java`, four instance variables are defined. These variables are used to record the position of a token or a phrase in the program. In this assignment, you are only concerned with the position of a token. The position of a token is defined as follows:

```
charStart:  the column number of the char that begins this token
charFinish: the column number of the char that terminates this token
lineStart=lineFinish: the number of the line where the token is found
```

Both line and column numbers start from 1. No token can span more than one line. Hence, `lineStart = lineFinish`.

There are three special cases:

- **String literals.** In the case of a legal string literal with or without escape characters, `CharStart` and `CharFinish` should be set to the column numbers as demonstrated by the following example:

  ```
     "VC\tcomp3131\ncomp9102"
     |                      |
  CharStart              CharFinish
  ```

  That is, `CharStart` represents the column number of the opening `"' while `CharFinish` represents the column number of the closing `"'. In the case of a string with illegal escape character(s), it is up to you to decide what the lexeme (i.e., the spelling for the token) should be for the "string" and set `CharStart` and `CharFinish` appropriately. Section 5 describes how to deal with this kind of lexical errors and how to deal with unterminated strings.
- **EOF Token.** When the end of file is reached, your parser returns an EOF token. In this case, set its spelling to `$` and its charStart to 1 and its CharFinish to 1.
- **Error Token.** Set its spelling and its position as if it were a legal token. For example, if # appears in the program, your scanner will create an object of the class `Token` as:

  ```
  new Token(Token.ERROR, "#", position)
  ```

  where `position` records the position of # in the input.

Your scanner reads the stream of chars in a VC program and produces a stream of tokens. For example, let the following program be given:

```
int f;
f = 1.2e+2;
```

such that `int` appears on the first column of the first line. On this program, your scanner must return the following sequence of eight tokens, denoted `tok1, ..., tok8`, with exactly the following information:

```
tok1.kind = Token.INT
tok1.spelling=int
tok1.position.lineStart = tok1.position.lineFinish = 1
tok1.position.charStart = 1
tok1.position.charFinish = 3

tok2.kind = Token.ID
tok2.spelling=f
tok2.position.lineStart = tok2.position.lineFinish = 1
tok2.position.charStart = 5
tok2.position.charFinish = 5

tok3.kind = Token.SEMICOLON
tok3.spelling=;
tok3.position.lineStart = tok3.position.lineFinish = 1
tok3.position.charStart = 6
tok3.position.charFinish = 6

tok4.kind = Token.ID
tok4.spelling=f
tok4.position.lineStart = tok4.position.lineFinish = 2
tok4.position.charStart = 1
tok4.position.charFinish = 1

tok5.kind = Token.EQ
tok5.spelling==
tok5.position.lineStart = tok5.position.lineFinish = 2
tok5.position.charStart = 3
tok5.position.charFinish = 3

tok6.kind = Token.FLOATLITERAL
tok6.spelling=1.2e+2
tok6.position.lineStart = tok6.position.lineFinish = 2
tok6.position.charStart = 5
tok6.position.charFinish = 10

tok7.kind = Token.SEMICOLON
tok7.spelling=;
tok7.position.lineStart = tok7.position.lineFinish = 2
tok7.position.charStart = 11
tok7.position.charFinish = 11

tok8.kind = Token.EOF
tok8.spelling=$
tok8.position.lineStart = tok8.position.lineFinish = 3
tok8.position.charStart = 1
tok8.position.charFinish = 1
```

A tab size of 8 characters is assumed. **However, this does not mean that the cursor always moves by 8 characters each time you hit the tab key.** See the test case `tab.vc` and its solution `tab.sol`.

4. Testing Your Scanner

Make sure that your CLASSPATH includes the parent directory that contains the directory `VC`. Note that `VC` does not appear in the CLASSPATH.

Create and run your scanner on a UNIX-based machine as follows:

1. Compile the Java files:

   ```
   javac vc.java
   ```

   This will create all required class files for your scanner.

2. Run your scanner on a test file, say, `tokens.vc` as follows:

```
java VC.vc tokens.vc
```

In the `vc.java`, we have turned on the debugging information in the scanner by including the line:

```
scanner.enableDebugging();
```

This will enable all the token objects be printed before being returned to its caller, which is the scanner test driver program in the file `vc.java`.

To pipe the output from your scanner on `tokens.vc` to a file, say, `tokens.out`, type:

```
java VC.vc tokens.vc > tokens.out
```

To visualise the characters in `tokens.out`, particularly when it includes invisible characters, you can use the following command:

```
od –a tokens.out
```

This is particularly helpful if your solution inadvertently includes invisible characters as part of a token.

Although some test files are provided for this assignment, you are responsible for designing additional test cases to make sure your scanner works as desired.

5. Lexical Errors

You are required to detect four types of errors:

1. illegal character, i.e., a char that cannot begin any token
2. Unterminated comment
3. Unterminated string
4. Illegal escape character

In Case 1, your scanner simply returns an error token as explained previously; your scanner should not print an error message. In addition, your scanner must "consume" the error token so that the remaining input can be processed when the scanner is called next time.

In Cases 2 -- 4, your scanner must first print an appropriate error message with `"ERROR:"` appearing somewhere in the message so that your solution can be auto-marked. In addition, you can recover from such an error as follows:

- **Case 2.** It is up to you how to handle the rest of the input. Remember that the scanner does not realise a comment is unterminated until it has hit the end of the file marker.
- **Case 3.** It makes senses for the scanner to return the string token found and start processing the remaining input when called the next time. The position information for the string token can be set in the obvious way.
- **Case 4.** It is up to you to decide what string token your scanner should return. Try a C or Java compiler! In any case, your scanner should start processing the remaining input when called the next time.

The VC Language Specification requires that every line be terminated by a line terminator. It is possible to create a VC program such that its last line is followed by the EOF rather than a line terminator. Such a program is not a legal VC program. However, in all the test cases we use for marking, each line is always terminated by a '\n'. Therefore, you are not expected to deal with this special case in any special way --- it will not be assessed.

Finally, if you think that a particular lexical error does not fit precisely one of the four categories listed above, you can use the following catch-all error message:

```
ERROR: others
```

When a test case contains a lexical error, the specific tokens and their originating positions are irrelevant. The script used for marking this assignment is solely focused on identifying and validating the presence of an

appropriate error message.

6. Marking Criteria

Your scanner will be assessed only by examining how it handles various correct and incorrect inputs. You will be graded only for the correctness of your solution. However, it is in your best interest to practice good programming and software engineering principles in coding.

A large number of test cases will be used in auto marking. Some test cases carry relatively larger penalities than others depending on the severities of the errors you have committed.

7. The OpenJDK 17 for Assignment Marking

This and all subsequent assignments will be marked by using the Java compiler installed on school servers:

```
% java -version
openjdk version "17.0.13" 2024-10-15
OpenJDK Runtime Environment (build 17.0.13+11-Debian-2deb12u1)
OpenJDK 64-Bit Server VM (build 17.0.13+11-Debian-2deb12u1, mixed mode, sharing)
```

If you log in via SSH or VNC to any of our login servers, you will end up using pretty much the same software set including Java.

While you can do your assignments on any CSE machine or your own machine, you must ensure that your assignments compile and work correctly on one of these two machines.

8. Submitting Your Scanner

You are not allowed to modify `Token.java` and `SourcePosition.java`. In addition, there does not appear to be a need for you to modify `SourceFile.java`. Finally, `vc.java` is only the driver for testing your compiler. Therefore, you should modify and submit only the following file:

`Scanner.java`

You should also submit `ErrorReporter.java` if you have modified it.

No other files will be accepted. However, let me know if you really have to submit some additional files.

The command for submitting your files is:

```
give cs3131 scanner Scanner.java ... (no class files, please)
```

For more information regarding how to submit your assignments and check your marks, please read Using classrun.

You can also submit your files via our course's web site.

9. Late Penalties

This assignment is worth 12 marks (out of 100). You are strongly advised to start early and do not wait until the last minute. You will lose 2.4 marks for each day the assignment is late. The penalty will be deducted from the total rather than the actual mark received.

Given the nature of our five programming assignments (as mentioned in the course line and also our first lecture), this course adopts a non-standard penalty rate (approved by the school). Given their tight deadlines, all programming assignments will usually be marked 2 or 3 days after their due dates in order to provide some feedback to you quickly to help you start working on the following assignment almost immediately.

Extensions will not be granted unless you have legitimate reasons and have let the LIC know ASAP, preferably one week before its due date.

10. Plagiarism

As you should be aware, UNSW has a commitment to detecting plagiarism in assignments. In this particular course, we run a special program that detects similarity between assignment submissions of different students, and then manually inspect those with high similarity to guarantee that the suspected plagiarism is apparent.

If you receive a written letter relating to suspected plagiarism, please contact the LIC with the specified deadline. While those students can collect their assignments, their marks will only be finalised after we have reviewed the explanation regarding their suspected plagiarism.

Here is a statement of UNSW on plagiarism:

Plagiarism is the presentation of the thoughts or work of another as one's own.*

Examples include:

- direct duplication of the thoughts or work of another, including by copying material, ideas or concepts from a book, article, report or other written document (whether published or unpublished), composition, artwork, design, drawing, circuitry, computer program or software, web site, Internet, other electronic resource, or another person's assignment without appropriate acknowledgement;
- paraphrasing another person's work with very minor changes keeping the meaning, form and/or progression of ideas of the original;
- piecing together sections of the work of others into a new whole;
- presenting an assessment item as independent work when it has been produced in whole or part in collusion with other people, for example, another student or a tutor; and,
- claiming credit for a proportion a work contributed to a group assessment item that is greater than that actually contributed.†

Submitting an assessment item that has already been submitted for academic credit elsewhere may also be considered plagiarism. Knowingly permitting your work to be copied by another student may also be considered to be plagiarism. An assessment item produced in oral, not written form, or involving live presentation, may similarly contain plagiarised material.

The inclusion of the thoughts or work of another with attribution appropriate to the academic discipline does *not* amount to plagiarism.

Students are reminded of their Rights and Responsibilities in respect of plagiarism, as set out in the University Undergraduate and Postgraduate Handbooks, and are encouraged to seek advice from academic staff whenever necessary to ensure they avoid plagiarism in all its forms.

The Learning Centre website is the central University online resource for staff and student information on plagiarism and academic honesty. It can be located at:

www.lc.unsw.edu.au/plagiarism

The Learning Centre also provides substantial educational written materials, workshops, and tutorials to aid students, for example, in:

- correct referencing practices;
- paraphrasing, summarising, essay writing, and time management;
- appropriate use of, and attribution for, a range of materials including text, images, formulae and concepts.

Individual assistance is available on request from The Learning Centre.
Students are also reminded that careful time management is an important part of study and one of the identified causes of plagiarism is poor time management. Students should allow sufficient time for research, drafting, and the proper referencing of sources in preparing all assessment items.

* Based on that proposed to the University of Newcastle by the St James Ethics Centre. Used with kind permission from the University of Newcastle.
† Adapted with kind permission from the University of Melbourne.

Finally, and also very importantly, have fun!

Jingling Xue
Last updated 03/02/2025 22:44:44