Assignment 1: Scanner

CSI4104 Compiler Design, Fall 2019
Document Version 1.2

Due date: Saturday, October 19 2019, 23:00.

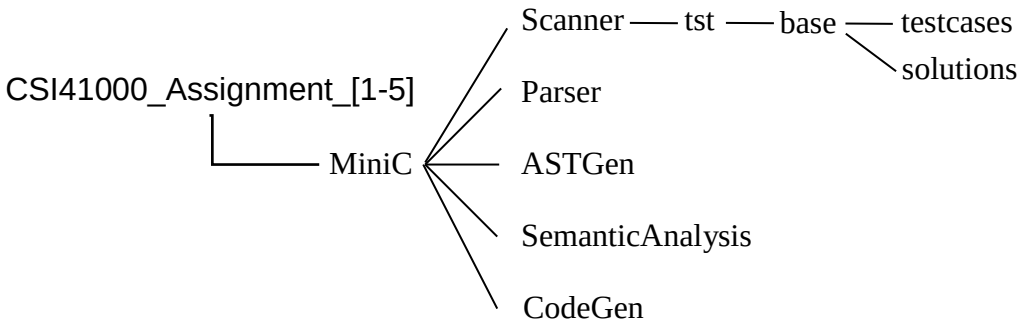
Introduction

In the first assignment we will implement a scanner by hand. This scanner will read a character stream from a MiniC source file and translate it to a sequence of MiniC tokens. This is known as the lexical analysis part of a compiler. The reason for writing the scanner by hand is that this experience will allow you to thoroughly understand the underlying principles of scanner construction. Based on this understanding, implementing a scanner using scanner generators like JLex or JavaCC will be straight-forward to you. To keep the effort reasonable, you are provided with a skeleton scanner that you can extend.

Before you start, you should carefully read the MiniC Language Specification (also available in YSCEC). The MiniC Language Specification explains in detail about the characters, tokens and comments that we want our MiniC scanner to recognize. Please note that at this stage it is not necessary to fully understand the MiniC grammar; the grammar is only provided to give you an initial idea how MiniC tokens will be composed to form MiniC programs (more on that topic in Assignment 2 on Syntax Analysis).

Environment

You can solve this assignment on any computer that has a Java compiler and a Java virtual machine (JVM) installed. However, all assignments will be tested and marked on the server **`elc1.cs.yonsei.ac.kr`**. It is therefore *highly recommended* that you test your programs on the server before submitting (see also the submission instructions below).
For this and all subsequent assignments, we'll use the following directory structure:



Every Java component of the MiniC compiler that you will develop during this semester will reside in its own package. All packages are sub-packages of the MiniC package. The MiniC package resides in directory MiniC, and all sub-packages reside in subdirectories of directory MiniC.

We provide you with a skeleton compiler that you can extend to support complete lexical analysis of MiniC source code. You will find the skeleton compiler on the server in the archive file /opt/ccugrad/Assignment1/CSI4100_Assignment_1.tgz. Copy this archive to your home-directory and untar it (e.g., in the shell, type `tar xvzf CSI4100_Assignment_1.tgz`). This will create the above directory structure in your home-directory. The directory structure is already populated with the files provided with the skeleton compiler. In the following description, all paths are relative to the top-level directory `CSI4100_Assignment_1`:

- `MiniC/MiniC.java`: this is the main program (driver) of our MiniC compiler.
- `MiniC/Scanner/Scanner.java`: a skeleton scanner to be completed by you.
- `MiniC/Scanner/Token.java`: definitions of all MiniC tokens, and a method for separating keywords from identifiers.
- `MiniC/Scanner/SourceFile.java`: handling of source files.
- `MiniC/Scanner/SourcePos.java`: a class to handle source positions (line and column numbers) of tokens.
- `build.gradle, gradle, gradlew, gradle.bat, build`: files and directories related to the Gradle open-source build automation system, which we use to compile the MiniC compiler (see below instructions on how to compile).
- `MiniC/Scanner/tst/base/testcases`: several testcases that you can use to test your scanner. You are encouraged to come up with more testcases to test your scanner in every possible way.
- `MiniC/Scanner/tst/base/solutions`: solutions for the provided testcases. Your completed scanner shall produce the output as shown in the solutions. Please note that this assignment gives some freedom with respect to lexical errors (see Section 'Lexical errors' below). Testcases containing lexical errors need not correspond byte-by-byte to the provided solutions, as long as they conform to the provided guidelines.
- `MiniC/scripts/scannertest.sh`: this is a script that you can use to automatically test your scanner. The script will run your scanner on every testcase in the Scanner/tst/base/testcases directory and compare the output to the corresponding solutions in Scanner/tst/base/solutions. Please note: this script is provided to you help debugging your scanner. The second purpose is to show you how large testing tasks can be automated by scripts. For the remaining assignments, you will have to create such scripts by yourself (e.g., by adapting the scanner testscript). **Please read the comments at the beginning of the script file so that you understand how to run the script. The script is provided as is, without guarantees about its functionality and correctness. Please use at your own risk, and do make sure that you also test**

your scanner manually, to have sufficient confidence that your scanner works correctly.

Note: for this assignment it is only necessary to modify file Scanner.java. *You must not modify Token.java or SourcePos.java, otherwise automated testing of your scanner might no longer work.* The compiler driver in MiniC.java will repeatedly call your scanner on an input file specified on the command line. With each call, your scanner is required to return the next token from the input file.

File Token.java defines a class `Token`, which we use to represent MiniC tokens. For every token in the input stream, your scanner is expected to provide the corresponding `Token` object. The constructor of the Token class expects three pieces of information:

1) the token kind, represented as an int (e.g., `Token.ID`, `TOKEN.INTLITERAL`);
2) the lexeme of the token, represented as a string (e.g., "counter", "22")
3) the position of the token in the program. Positions are represented as objects of class `SourcePos`.

Class `Token` makes it easy to distinguish keywords from identifiers: initially the scanner classifies all keywords as identifiers. The constructor of class `Token` compares the lexeme of the presumptive identifier with the list of MiniC keywords. In case the presumptive identifier is contained in the list of MiniC keywords, the constructor will change the token kind accordingly (see file Token.java).

The class `SourcePos` from file SourcePos.java contains four instance variables: `StartCol`, `EndCol`, `StartLine` and `EndLine`. These variables are used to record the position of a token within a MiniC program. In MiniC, line and column numbers start from 1. Variable `StartCol` denotes the column where a token starts, variable `EndCol` denotes the column where a token ends, and `StartLine` and `EndLine` denote the line number where the token occurred (Note: in MiniC, no token can span multiple lines, therefore `StartLine=EndLine` for every token.).

With **string literals**, `StartCol` and `EndCol` are set to the occurrences of the enclosing quotes:

```
"Hi there, this is a string token...\n"
^                                     ^
|                                     |
StartCol                              EndCol
```

If a string contains illegal escape sequences (i.e., an escape sequence other than \n), it is up to you to decide on the lexeme for this string (see also the section on lexical errors below). However, you must set `StartCol` and `EndCol` accordingly.

When the end of file is reached, the parser returns an **end-of-file token** (`Token.EOF`). This has already been implemented for you in Scanner.java.

Error Token: `Token.ERROR` is used like all other tokens: its spelling and position is set according to the input. For example, if the character @ appears in the input, the scanner will create a `Token` object using

```
new Token (Token.ERROR, "@", src_pos),
```

where src_pos specifies the position of character @ in the input.

Example

Let us assume the following MiniC input program (starting at line 1, column 1 of the input source file):

```
int x;
x = 1.4e+2;
```

Then your scanner should deliver the following sequence of tokens:

```
token1.kind = Token.INT
token1.lexeme = "int"
token1.src_pos.StartLine = 1
token1.src_pos.EndLine = 1
token1.src_pos.StartCol = 1
token1.src_pos.EndCol = 3

token2.kind = Token.ID
token2.lexeme = "x"
token2.src_pos.StartLine = 1
token2.src_pos.EndLine = 1
token2.src_pos.StartCol = 5
token2.src_pos.EndCol = 5

token3.kind = Token.SEMICOLON
token3.lexeme = ";"
token3.src_pos.StartLine = 1
token3.src_pos.EndLine = 1
token3.src_pos.StartCol = 6
token3.src_pos.EndCol = 6

token4.kind = Token.ID
token4.lexeme = "x"
token4.src_pos.StartLine = 2
token4.src_pos.EndLine = 2
token4.src_pos.StartCol = 1
token4.src_pos.EndCol = 1

token5.kind = Token.ASSIGN
token5.lexeme = "="
token5.src_pos.StartLine = 2
token5.src_pos.EndLine = 2
token5.src_pos.StartCol = 3
token5.src_pos.EndCol = 3
```

```
token6.kind = Token.FLOATLITERAL
token6.lexeme = "1.4e+2"
token6.src_pos.StartLine = 2
token6.src_pos.EndLine = 2
token6.src_pos.StartCol = 5
token6.src_pos.EndCol = 10

token7.kind = Token.SEMICOLON
token7.lexeme = ";"
token7.src_pos.StartLine = 2
token7.src_pos.EndLine = 2
token7.src_pos.StartCol = 11
token7.src_pos.EndCol = 11

token8.kind = Token.EOF
token8.lexeme = "$"
token8.src_pos.StartLine = 3
token8.src_pos.EndLine = 3
token8.src_pos.StartCol = 1
token8.src_pos.EndCol = 1
```

Longest Match

It is important for our MiniC scanner to always consume the longest possible match (maximal munch). The examples in the following table should point that out.

Input	Token(s)	Comments
>=	“>=”	One token GREATEREQ, rather than two tokens GREATER and ASSIGN.
==	“==”	One token EQ instead of two tokens ASSIGN.
//		An end-of-line comment, rather than two tokens “/” and “/”. Note: the scanner throws away comments. So everything starting from // to the end of the line should be consumed and thrown away.
@	Token.ERROR	
2.2+2	“2.2” “+” “2”	Token.FLOATLITERAL (“2.2”), Token.PLUS (“+”), Token.INTLITERAL (“2”)
2.2e+2	“2.2e+2”	Token.FLOATLITERAL (“2.2e+2”)
2.2e+ 2	“2.2” “e” “+” “2”	Token.FLOATLITERAL (“2.2”), Token.ID (“e”), Token.PLUS (“+”), Token.INTLITERAL (“2”)

The scanner should discard whitespace and comments. The scanner must always return an object of class `Token` as defined in file Token.java.

Lexical errors

A MiniC scanner should detect four kinds of lexical errors:

1) illegal characters, i.e., characters that cannot be part of any token,
2) un-terminated comments,
3) un-terminated strings,
4) illegal escape sequences in strings, e.g. "\y".

In Case (1), your scanner should simply return an error token. Please refer to the introduction of the error token above regarding the lexeme that you should provide with the error token. Your scanner must "consume" the lexeme belonging to the error token, so that scanning can proceed when your scanner is called next time.

In cases (2)-(4), your scanner must print an error message. This error message must contain the string "ERROR:" in capital letters. This is important for automated testing of your scanner. Our automated test script will look for the "ERROR:" string in your output. After printing this error message, you should proceed as follows:

Case (2): The scanner recognizes an un-terminated comment only by the time it reaches the end-of-file. You can consider the scan-job to be finished in this case and return Token.EOF.

Case (3): Note that a MiniC string is not allowed to extend to the next line. When your scanner reaches a new line while scanning a string, it should return the string token up-to-but-not-including the newline character and set the position information accordingly. When called next time, the scanner should continue with the input following the un-terminated string.

Case (4): Whatever lexeme you return with the string token, you should set the StartCol and EndCol positions accordingly. Your scanner should continue with the remaining input when called next time.

Note that with MiniC every line must be terminated by a line terminator. With all test cases, each line will be terminated by '\n'. For this reason, in your scanner you need not care about programs where the last line is terminated by EOF rather than '\n'.

Building your scanner

We use the Gradle build automation system to compile our MiniC compiler. By the end of the semester, our MiniC compiler will consist of many Java packages, and during coding and testing of such large software an incremental build system is helpful. Incremental build systems intelligently determine which parts of the source-code are up-to-date, so that only the minimum, invalidated set of source-files will be re-compiled (thus compile-time is greatly reduced!).

Gradle comes with a wrapper file which can be used to invoke Gradle. On Unix systems, the wrapper is called ``gradlew''. On Windows, the wrapper is called ``gradlew.bat''. In the following, we use `gradlew`, because our server runs the Linux operating system.

To build the MiniC compiler, navigate to the top-level directory of the Assignment 1 source tree, which is `CSI4100_Assignment_1`, and type:

```
./gradlew build
```

Note the prefix ``./'' before gradlew. This is a Unix-speciality, because on Unix systems the current working directory (./) is not included in the $PATH environment variable. If you omit ``./'', the shell won't find the gradlew wrapper script.

When you run gradlew for the first time, it will download and install Gradle in a local subdirectory of the Assignment 1 source tree. Therefore, for the first invocation you need a working Internet connection. (On our server, Gradle is already installed and set up for you. Thus on the server you can use `gradle` instead of `gradlew` to avoid the lengthy installation into your home-directory.)

Gradle's `build` task does an incremental buid of the MiniC compiler. The initial invocation has to compile all Java classes, but from then onwards only source files that you have updated or sourcefiles depending on updated source-files will be recompiled (to keep the compile-time to a minimum).

If you want to remove all build artefacts such as class files from previous builds, you should run gradle's ``clean'' task:

```
./gradlew clean
```

The clean task will remove the ``build'' subdirectory and all its contents. Running the ```build`'' task the next time will require a full build.

For further information, you can find the Gradle configuration for the MiniC compiler in file gradle.build.

Running and testing your scanner

The gradle `build` task creates a jar-file in subdirectory `build/libs`. This jar-file contains the byte-code of all Java classes of your MiniC compiler. To run the MiniC compiler, you must use the `-jar` command-line switch to provide the path to the jar-file as an argument to the JVM.

For example, assuming that you are in the top-level directory and that file `your-input-file` contains just the single character "*". Then

```
java –jar build/libs/MiniC-Scanner.jar your-input-file
```

will produce the following output:

```
********** MiniC Compiler **********
Lexical Analysis ...
token1.kind = Token.ERROR
token1.lexeme = "*"
token1.src_pos.StartLine = -1
token1.src_pos.EndLine = -1
token1.src_pos.StartCol = -1
token1.src_pos.EndCol = -1

token2.kind = Token.EOF
token2.lexeme = "$"
token2.src_pos.StartLine = -1
token2.src_pos.EndLine = -1
token2.src_pos.StartCol = -1
token2.src_pos.EndCol = -1
```

The scanner of the skeleton compiler does not recognize Token.TIMES ("*") yet, so it returns Token.ERROR. The next token is the end-of-file token Token.EOF. The scanner of the skeleton compiler does not maintain line-numbers nor column numbers yet, therefore -1 (default value) is printed.

If you want to redirect the output of your scanner to a file, use

```
java –jar build/libs/MiniC-Scanner.jar your-input-file > your-output-file
```

Marking

Your scanner will be assessed by how it handles correct and incorrect inputs. We will use a large number of test-cases for marking. The test-cases provided to you are a subset of the test-cases that we will use during marking. Testing will be done on the server, so please make sure that your scanner compiles and executes there.

Please note that all assignments are *individual* assignments. You are most welcome to discuss your ideas with your colleagues, but 'code-sharing' or 'code-reuse' is not allowed. Assignments are designed to facilitate understanding of the course contents. Assignments are also a part of the preparation for the exams.

Deliverables and Submission

You should extend the scanner in file Scanner.java so that it covers the complete MiniC specification. Note that for this assignment it is only necessary to modify file Scanner.java. Submission will therefore be restricted to this single file. If you need to submit additional files, you must let us know in advance. *You are not allowed to modify Token.java and SourcePos.java, otherwise automated testing of your scanner might no longer work.*

Warning: please make sure that the tokens emitted by your scanner contain correct source-file position information (line and column numbers). Scanners emitting incorrect position information will receive 0 score.

To submit your file Scanner.java, please log in on our server, create a submission directory, e.g., ~/myassignment1, and copy your modified Scanner.java in this directory.

After copying file Scanner.java in your submission directory, you must cd to the submission directory and type 'submit'. This will collect your file and copy it to our repository.
[you@elc1 ~] cd ~/myassignment1
[you@elc1 myassignment1] submit

Please note: There is no submission unless you issue the 'submit' command! Failing to issue the 'submit' command will result in 0 score. The timestamp of your submission will be recorded by the 'submit' command. Please note also that if you re-submit, then a previous submission will be overwritten and a new timestamp will be recorded.

If you have questions, don't hesitate to contact us. ☺