CSCU9Y4 Programming Language Paradigms Practical 4

Consider how you get from source code (which is just a text file, a sequence of characters) to an executing program. The first step the compiler or interpreter has to carry out is *lexical analysis* of the source code. That is, reading the text file and splitting the stream of characters into chunks that can then be associated with meanings. It does this according to the rules laid down by the programming language. These chunks are *tokens*. A *token* is a *lexical element* (as defined by the programming language), plus *literals* (i.e. numeric values, strings and so on). Examples of lexical elements are:

Identifiers In most languages, e.g. Pascal, Java and C++, identifiers begin with a letter and may contain letters, digits and underscores. The ***case*** of a letter is ***not*** significant in Pascal, while in Java and C++ it is.

Reserved words Some identifiers are ***reserved words*** and have a special meaning, e.g. **for**, **while**, **if**.

Blanks (i.e. spaces) In modern languages they are significant while in early languages, such as Fortran and Algol 60, they were ignored. Spaces can be used to *delimit* lexical elements (i.e. to determine the start and finish of a lexical element).

Comments How are these delimited? Pascal uses { ... } or (* ... *) . C uses /* ... */
What happens when you forget to close a comment? Syntax knowledgeable editors can help identify such errors.
Usually there is also special notation for comments of only one line, terminated by the end of the line. Ada uses --, Java and C++ use //

Statement termination Early languages used the end of line character to terminate a statement. Most modern languages are free format and use a semicolon to ***terminate*** (C++, Java, Ada) or to ***separate*** (Pascal) a statement.

ArrTokenTest (from practical 2) takes an input and splits it into *tokens* using what is termed a '*regular expression'.* Regular expressions come up a lot in computer science. They are a standard concise way of describing patterns in strings, including repetition and choice. Decomposition of a text into relevant parts is one of the steps used by a compiler/interpreter to understand source code. You are exploring here how that might happen, and understanding the notion of lexical elements.

This practical asks a lot of questions. Please do all the tests, and think carefully about the answers. Write them on the sheet so you can report any of them to the tutor for the checkpoint.

1. Often tokens are obtained by using white space separators. In the text area at the top of the window, type the text:

    What a lovely day.

then put a space into the second text field, labeled **Regular Expression.** What does **Split** do?

What has happened to the full stop?

Here we have a very simple language, where the lexical elements are words. TokenTest has split the English sentence into lexical elements (words) based on the space delimiter. The regular expression at this point is very simple, i.e. it is just a *space.* The program just looks for a space and splits the words up using this – the space character acts as a *delimiter*.

2. Try other examples. Does the input have to be a proper English sentence?

3. Are spaces tokens? Try adding extra spaces in the input above. What happens to the output? Try with '. .', i.e. dot space space dot. What do you expect? Is what you get what you expect?

4. Delimiters can be whatever we choose. Here, they are described by *regular expressions*. Try the input "programming,language,paradigms" with a ',' character as a delimiter. What happens? The simplest regular expression just describes exactly the pattern of characters being looked for. Try using a sequence of characters as a delimiter, e.g. xxx (with suitable input of course).

5. Regular expressions can be more complex, describing repeated patterns. They are ubiquitous in computing, unfortunately sometimes with different syntax in different contexts. Here, the syntax used is similar to that in Extended BNF. Repetition is described using '+' (one or more repetitions) and '*' (zero or more repetitions). Try using x+ and x* as delimiters (with appropriate input). Can you see the difference in what these can match?

6. What would you put in the delimiters field to treat multiple spaces as a delimiter?

7. What if we want to use more than one kind of delimiter? How can we make the program treat "What a lovely day, today!" as *five* tokens (five words without spaces or commas)? The '|' character has a special meaning in a regular expression: it is a choice operator. For example ' |,' means the delimiters are a space *or* a comma. Try it (with suitable input).

8. Try some more input. Use spaces between words. Use other delimiters (remember to separate each possible delimiter with a '|' character). What happens?

9. Now run the program with the input string: `'i*val-23 / a'` I.e. with no spaces around '*****' or '**-**', but with spaces around '**/**', and space as the delimiter. Can you explain the result? Run the program again with the same input string, but with each of **+-*/** as delimiters. Be careful about how you describe these. You will need choice, and in order to show that we wish to look for the actual character '+' or '*', rather than the '+' (one or more) or the '*' (zero or more) operator, we have to put a backslash character in front of them, like this: \+ or *. The same is also true for a full stop which we represent using '\.'.

What has now happened? Thinking about interpreting a program, why do you think that you want to keep delimiters such as '+' in your output, but not spaces?

10. Suppose you want to use sequences of '+' as delimiter, i.e. you want to split the string "Happy+++Birthday++to+You" into the four words. Which regular expression would you use?

Checkpoint

Now demonstrate to a tutor that you know how to specify the regular expression for steps 7 and 10 and 11. Complete the quiz for the checkpoint on Canvas.

Challenge: String manipulation. Suppose you wanted to ignore text inbetween "X"s in your input. One way to do this is to use the method **ReplaceAll** to manipulate the input string before passing it to **split**. First, add a call to **ReplaceAll** (before using split) in your code to replace all the text between two "X"s (including the "X"s) by a space. This needs quite a sophisticated regular expression. See the Java API guide for more detail (hint: starting from a class that uses regex, e.g. **String**, find the definition of the regex operators). You need not deal with nested comments. Use the multiple space delimiter.

Twist: what if you wanted to ignore comments in braces (the characters { and })? Modify the call to **ReplaceAll** to replace all the text between braces (including the braces) by a space. Braces are special characters within the regular expression.