**Tokenization**

**PURPOSE OF DOCUMENT:**

To act as a knowledge source for the develop of the tokenizer. This initial document is intended to be modified as more information is collected.

**DEFINITION:**

Tokenization is the act of breaking up a sequence of strings into pieces such as words, keywords, phrases, symbols and other elements called tokens.

(source: https://www.techopedia.com/definition/13698/tokenization)

**EXAMPLE OF TOKENZATION:**

FIGURE 1: Tokenization.

myFavouriteNumber = someNum + otherNum;

**Tokenized Data:**

VARIABLE myFavouriteNumber

EQUALS

OPEN\_PARENTHESIS

VARIABLE someNum

PLUS

VARIABLE otherNum

CLOSE\_PARENTHESIS

SEMICOLON

In practice, the system that is being developed would turn:

myFavouriteNumber  = someNum + otherNum;

Into something like:

Line1: VARIABLE EQUALS OPEN\_PARENTHESIS VARIABLE PLUS VARIABLE CLOSE\_PARENTHESIS SEMICOLON

The tokenized data in Figure 1 could be then be compared to other code that is tokenized in the same way.

***Implementation note***: Variable names are disregarded, however variable types may be used. For instance, the above (see Figure 1) could instead be tokenized as shown in Figure 2.

FIGURE 2: Tokenization with distinguished variables.

myFavouriteNumber = someNum + otherNum;

**Tokenized data:**

**NUMERAL**\_VARIABLE myFavouriteNumber

EQUALS

OPEN\_PARENTHESIS

**NUMERAL**\_VARIABLE someNum

PLUS

**NUMERAL**\_VARIABLE otherNum

CLOSE\_PARENTHESIS

SEMICOLON

The purpose of this would be to distinguish the above statement from others such as a String concatenation statement like:

full\_name = first\_name + last\_name;

**PRINCIPLES FOR TOKENIZATION:**

1. Strip out useless information (i.e., whitespaces).
2. Reduce code to meaningful tokens while retaining useful characteristics. See Figure 3 for an example of tokenization that is too general.
3. Keep track of line numbers for reporting.
4. The better the job that the tokenizer does at preparing the data, the better the analysis of the code will be.
5. The more generic the tokenization, the more likely a false positive match will occur (shown in Figure 1 vs. Figure 2). Conversely, the more “precise” the tokenization, the closer it will be to the original code and it will have less utility in aiding in detection of similar code.
6. A collection of lexical tokens will need to be created for each language (at minimum: C, C++, and Java) to perform the tokenization.

FIGURE 3: Tokenization that is too general.

myFavouriteNumber = someNum + otherNum;

**Tokenized data:**

VARIABLE myFavouriteNumber

OPERATOR

PUNCUATION

VARIABLE someNum

OPERATOR

VARIABLE otherNum

PUNCUATION

PUNCUATION

Note that useful information about the type of operators and punctuation is lost.

1. The tokenization procedure must be able to gracefully handle uncompilable code to account for errors such as the ones shown below in Figure 4.
   * An important consideration here is that someone may create these errors purposely, to avoid detection.
     + For instance, suppose a student is submitting an assignment and they know that the code will not be compiled (which happens from time to time for certain assignments). They could try to introduce errors to prevent detection if the above type of statements were just disregarded or handled differently from properly tokenized ones.

i.e., In Figure 4, Example 2, if the system tokenized ***intRandomNum*** differently than it would ***int RandomNum****,* the similarity could be missed.

* + - **A decision will need to be made on whether or not to report these sorts of cases to the instructor (as it is an outlier). It’s likely satisfactory to assume that instructors are responsible to ensure code runs properly and that this case doesn’t matter.**
* An intuition on the best way to deal with code that is not recognized is to leave it as the original text.

FIGURE 4: Tokenizing erroneous code.

Example 1: ublic someMethod(){};

Example 2: intRandomNum;

Example 3: ()sdjkasdasl;

ETC.

**DEALING WITH COMMENTS:**

* Removing all comments is an easy fix for removing personal information.
* Leaving comments in increases the chance of finding a copied piece of code:
  + Matching comments may sometimes be more incriminating than matching code, since there is a lot more variation in English than what will show up in tokenized code. Even though it is an outlier, it may be a nice feature to include.
  + Comments from common templates such as the ones found in most popular IDEs, or ones distributed for a class (such as in 1P02), will need to be disregarded.
  + A reasonable approach may be to ignore all comments before actual start of code for a program but analyze all comments after and check for templated code.
    - Professors can also include their templates to be ignored.
  + There would also be an additional provision to search the code for the student’s name, student ID, and username to ensure it is not anywhere else in the comments.

**CODE TO IGNORE:**

Very similar to how comments are dealt with, there is certain code that will need to be ignored:

* Code samples from an instructor
* Common constructs such as:

public static void main…

**OTHER CONSIDERATIONS:**

There are tools that exist for lexical analysis that could be investigated, such as JLex. However, for the purposes of this project, the task is likely straightforward enough to build a custom solution. There also exists a built-in tokenizer in Java (*StringTokenizer*) that may be useful.