Special Topic: Functional Programming
Source Code Similarity Tool: Zim

Zane Keeley
School of ICT
Griffith University

October 21, 2019

Contents

Abstract

This report details the Zim project for the Advanced Topics: Functional Programming course. The purpose of the project was to develop a tool which could be used to detect similarities / plagiarism between files, producing a report summarizing the results and outlining any matches found between the compared files. By the end of the project, all of the core requirements for the project were completed, with the tool being able to detect similiarities between files. The core algorithm for the tool ended up being quadratic with a complexity of $O(N^2)$, where N represents the number of files being compared. A few extras were also implemented within the summarizing report to help support the detection of plagiarism, which are particularly useful when comparing large files or a large number of files.

1 Introduction

1.1 Scope and Requirements

The initial scope of the project was to develop a tool for detecting similarities between files written using the JavaScript language, thereby detecting possible plagiarism amongst students.

The core requirements for the project were as follows:

1. To be able to compare multiple JavaScript files, detecting any sections of code between the files which match.
2. The tool would be able to detect matches even when identifiers, whitespace and comments within the code had been changed.
3. To provide a report each time the tool was used which summarizes the results of the files that were compared and outlines any matches that were found. This report should also be easy to read and navigate.
4. The tool would have the ability to be easily expanded into further programming languages.

2 Design

During the early design stage of Zim, we decided that we would follow the technical report of Sim as a guide, which is a similar tool used for detecting the similarities between files. The algorithm used in Sim consists of generating a list of tokens using a lex-generated scanner for the input, then comparing the tokens between inputs to determine any matching runs, where a matching run is a substring of consecutive matching tokens between the inputs with a length equal to or greater than a required minimum length [1].

For the design of Zim, we determined that the main program would be broken up into 4 key sections: which were lexing a file; tokenizing a file; comparing the files; and outputting the results.

A lexer would be used to separate the contents of each file into its associated lexemes and positions, which would allow easy expansion into further programming languages, as each lexer would be responsible for separating a file into its associated language elements. The lexemes would then be translated into tokens during the tokenizing phase of Zim, which would encapsulate the lexemes of the programming language into identifiers, literals, and keywords (including the operators and separators), making comparisons between files and determining the file positions of any matches easier. Once files are tokenized, we can compare the tokens between files, detecting any sections of code where the tokens match and collecting the results to provide within the report. The phases of the Zim tool and the order they are performed can be seen in figure 1.

We decided that an HTML format for the report would be best as it could be easily navigated by the user and they would only have to store the reports they're interested in, such as when the tool detects plagiarism the user would like to keep a record of. To aid in detecting plagiarism, we wanted the report to provide the file paths, an indication of the length of each match and a

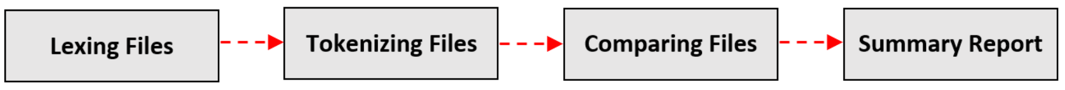


Figure 1: The phases of the Zim tool.

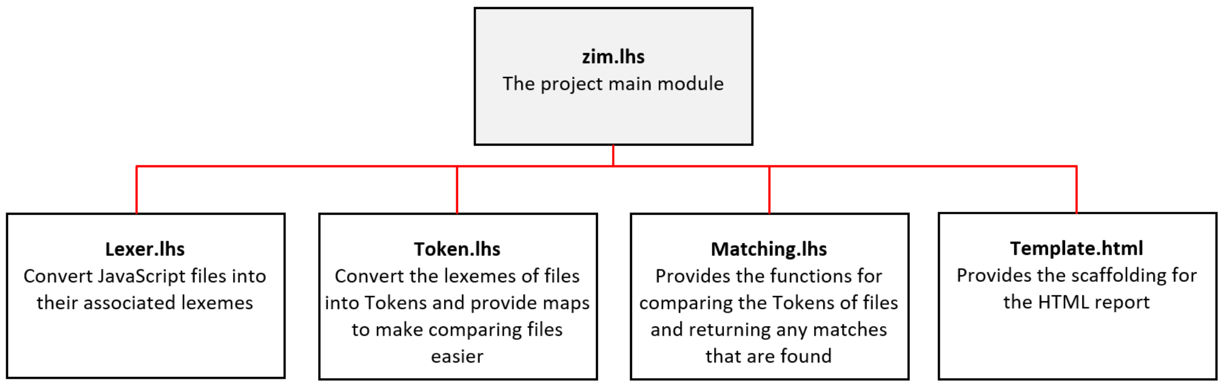

zim.lhs
The project main module
Lexer.lhs
Convert JavaScript files into their associated lexemes
Token.lhs
Convert the lexemes of files into Tokens and provide maps to make comparing files easier
Matching.lhs
Provides the functions for comparing the Tokens of files and returning any matches that are found
Template.html
Provides the scaffolding for the HTML report

Figure 2: The module hierarchy of Zim.

side by side comparison of any sections of code where a match was detected, which would allow the user to easily inspect the code and determine whether actual plagiarism has occured.

3 Implementation

In this section we present how the phases of Zim were implemented, outling the modules that were developed to perform the tasks discussed during the design. The module hierarchy for the implementation of Zim can be seen in figure 2.

3.1 Zim

`Zim` is the main module for the project, which provides the main function and is responsible for integrating the various modules of the project with the user input data. The goals of the Zim module are to:

1. Define and detect the command line options that will be used when comparing the collection of files (such as the programming language to be used).

2. Compare a collection of files provided as command line arguments and return a summary of the results as an HTML report.

```
module Main (main) where

import System.Environment
import System.Exit
import System.IO
import Control.Monad
import Data.Tuple
import Data.Typeable
import Data.List as L
import Data.Char
import Data.Map.Strict as M
import Data.Maybe
import Data.Set as S
import ABR.Util.Pos
import ABR.Util.Args as A
import ABR.Control.Check
import ABR.Parser
import ABR.Parser.Checks
import ABR.Text.Configs as C
import ABR.Text.String
import ABR.Text.Markup

import Zimm.Lexer
import Zimm.Token
import Zimm.Matching
```

`Source` is a type alias for a String, representing the source code for a file.

```
type Source = String
```

`readLex path` reads the file given by the `path`, lexs it, then returns the (path, source, tlps) – where tlps are a list of tagged lexemes and their respective positions within the lexed file, with the tag being used to identify the different types of lexemes.

```
readLex :: FilePath -> IO (FilePath, Source, TLPs)
readLex path = do
   putStrLn $ "File: " ++ path
   source <- readFile path
   case checkLex lexerL source of
      CheckPass tlps -> return (path, source, tlps)
      CheckFail msg  -> do hPutStr stderr msg
                           exitWith $ ExitFailure $ -1
```

`snippet startPos endPos source` returns the lines of `source` between the `startPos` and `endPos` (inclusive).

```
snippet :: Pos -> Pos -> Source -> Source
snippet (l1, _) (l2, _) source = unlines $
   L.take (l2 - l1 + 1) $
   L.drop l1 $
   lines source
```

`snippetTokens p1 p2 tlps` returns the lines of `tlps` between `p1` and `p2` (inclusive).

```
snippetTokens :: Pos -> Pos -> TLPs -> Source
snippetTokens p1 p2 tlps = f (fst (snd (head tlps))) $
   L.takeWhile (\((_, _), p) -> p <= p2) $
   L.dropWhile (\((_, _), p) -> p < p1) tlps
   where
   f :: Int -> TLPs -> Source
   f line tlps = case tlps of
      [] -> ""
      ((_, lexeme), (l, c)) : tlps'
         | l == line -> ' ' : makeHTMLSafe lexeme ++ f line tlps'
         | otherwise -> '\n' : makeHTMLSafe lexeme ++ f l tlps'
```

`findCrossTotals runs` returns the total length of the matching runs found between each pair of compared files where at least 1 matching run occured.

```
findCrossTotals :: [(FilePath, FilePath, [Run])] -> [(FilePath, FilePath, Int)]
findCrossTotals runs = L.sortBy (\(_,_,tc1) (_,_,tc2) -> compare tc2 tc1)
                       $ L.map (\(f1, f2, rs) -> (f1, f2, sum
                       $ L.map (len) rs)) runs
```

`findClusters sets` takes a list of Sets and returns a list of the merged Sets, where a pair of Sets are merged whenever their intersection is not the empty list (i.e they share atleast one element) – where the function will continue to process the sets until no further merges can be achieved.

```
findClusters :: Ord a => [Set a] -> [Set a]
findClusters sets | L.length sets < 2 = sets
                  | otherwise         = findClusters' (head sets) (tail sets)
    where checkSets :: Ord a => Set a -> Set a -> Bool
          checkSets set1 set2 = S.null (S.intersection set1 set2)
          findClusters' :: Ord a => Set a -> [Set a] -> [Set a]
          findClusters' set' sets' | L.null sets' = [set']
                                   | otherwise    =
            let (p1, p2) = L.partition (checkSets set') sets'
            in case p2 of
                [] -> [set'] ++ findClusters' (head p1) (tail p1)
                _  -> if (L.null p1)
                      then [S.unions ([set'] ++ p2)]
                      else findClusters' (S.unions ([set'] ++ p2)) p1

main :: IO ()
main = do
   -- check command line arguments. If any invalid command line options are
   -- provided, exit the program displaying the invalid options
   args <- getArgs
   let (options, fileNames) = findOpts
          [ParamS "lang", ParamS "run", FlagS "list", FlagS "desc"] args
       bads = L.filter (\(c:_) -> c `elem` "+-") fileNames
       lang = L.map toLower $ trim $
          (A.lookupParam "lang" options "JavaScript")
   unless (L.null bads) (do
      putStrLn $ "bad options: " ++ show bads
      exitWith $ ExitFailure $ -1
     )
   unless (lang == "JavaScript") (do
      putStrLn $ "language type not supported: " ++ show lang
      exitWith $ ExitFailure $ -1
     )
   let minRunLen :: Int
       minRunLen = read $ A.lookupParam "run" options "20"
       listSources = A.lookupFlag "list" options False
       listOrder = A.lookupFlag "desc" options True
   -- read in the files that were provided and lex them, and the template
   pathSourceTlpss <- mapM readLex fileNames
   template <- readFile "template.html"
   let
      indexPathSourceTlpss = L.zip [1..] pathSourceTlpss
      sourceMap :: M.Map FilePath Source
      sourceMap = M.fromList $ L.map (\(path, source, _) -> (path, source))
         pathSourceTlpss
```

```
tlpsMap :: M.Map FilePath TLPs
tlpsMap = M.fromList $ L.map (\(path, _, tlps) -> (path, tlps))
   pathSourceTlpss
filePathMap :: M.Map FilePath Int
filePathMap = M.fromList $ L.zip (L.map (\(path,_,_)
      -> path) pathSourceTlpss) [1..]
keyMap = makeMapsKey $ keywords ++ separators ++ operators
tlpss = L.map (\(_, _, tlps) -> tlps) pathSourceTlpss
keyLitMap = makeMapsLiteral keyMap tlpss
pathTokenss = [(path, tokens) | path <- fileNames,
               let tlps = fromJust $ M.lookup path tlpsMap,
               let keyLitIdMap = makeMapsId keyLitMap tlps,
               let tokens = tokenizeFile keyLitIdMap tlps]
allRuns :: [(FilePath, FilePath, [Run])]
allRuns = L.concat [[(path1, path2, runs)
                 | (path2, tokens2) <- pts,
                   let runs = filterRuns $ findRuns minRunLen tokens1 tokens2,
                   not (L.null runs)]
                 | (path1, tokens1) : pts <- L.tails pathTokenss]
matches = if listOrder
          then L.sortBy (\(_,_,r1) (_,_,r2) -> compare (len r2) (len r1))
               $ [(path1, path2, run)
                 | (path1, path2, runs) <- allRuns, run <- runs]
          else [(path1, path2, run)
                 | (path1, path2, runs) <- allRuns, run <- runs]
crossTotals = findCrossTotals allRuns
indexMatches = L.zip [1..] matches
clusters = L.map (S.toList) $ findClusters (L.map (\(p1,p2,_)
           -> S.fromList [(filePathMap ! p1), (filePathMap ! p2)]) allRuns)
listFlag = if listSources then [CParam "listFlag" "true"] else []
reportData = listFlag ++ [
   CList "files"
      [[CParam "number" (show n),
        CParam "path" path,
        CParam "tokensNum" (show (L.length tlps)),
        CParam "fullSource" (makeHTMLSafe source)]
      | (n, (path, source, tlps)) <- indexPathSourceTlpss],
   CList "matches"
      [[CParam "file1" (show (filePathMap ! path1)),
        CParam "file2" (show (filePathMap ! path2)),
        CParam "pos1" (show (startPos1 run) ++ " - " ++ show (endPos1 run)),
        CParam "pos2" (show (startPos2 run) ++ " - " ++ show (endPos2 run)),
        CParam "matchingTokens" (show (len run)),
        CParam "matchNum" (show n),
        CParam "source1" (makeHTMLSafe (snippet (startPos1 run)
               (endPos1 run) (sourceMap ! path1))),
        CParam "source2" (makeHTMLSafe (snippet (startPos2 run)
               (endPos2 run) (sourceMap ! path2))),
        CParam "tokens1" (snippetTokens (startPos1 run) (endPos1 run)
               (tlpsMap ! path1)),
        CParam "tokens2" (snippetTokens (startPos2 run) (endPos2 run)
               (tlpsMap ! path2))]
      | (n, (path1, path2, run)) <- indexMatches],
        CList "crossTotals"
           [[CParam "file1" (show (filePathMap ! path1)),
             CParam "file2" (show (filePathMap ! path2)),
             CParam "tokens" (show tokens)]
            | (path1,path2,tokens) <- crossTotals],
        CList "clusters"
           [[CParam "cluster" (show cluster)]
            | cluster <- clusters]
        ]
      report = popTemplate reportData template
   writeFile "report.html" report
```

3.2 Lexer

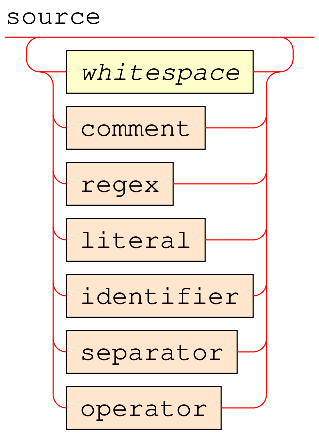
The `Lexer` module provides the lexical analysis of a JavaScript source through grouping the characters of a file into the lexemes associated with the JavaScript language. The Lexer module utilizes the ABR.Parser, ABR.Util.Pos and ABR.Parser.Lexers libraries to lex files and provide the positions for the lexemes [2].

```
module Zimm.Lexer (
      lexerL, keywords, separators, operators
   ) where

import Data.Char

import ABR.Util.Pos
import ABR.Parser
import ABR.Parser.Lexers
```

3.2.1 Overall lexer

`lexerL` is the lexer that recognises all the tokens in a JavaScript program.

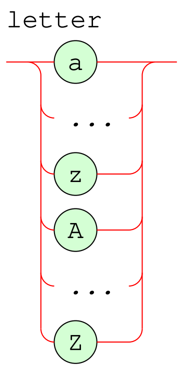
```
source ::= {$whitespace$ | comment | regex | literal | identifier |
           separator | operator}
```



```
lexerL :: Lexer
lexerL =
   tagKeywords keywords $ dropWhite $ nofail $ total $
      listL [whitespaceL, commentL, regexL, mLiteralL, identifierL,
         separatorL, operatorL]
```

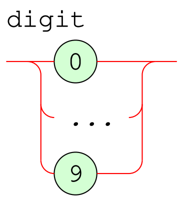
3.2.2 Language Element Lexers

```
letter ::= ("a" | $...$ | "z" | "A" | $...$ | "Z")
```



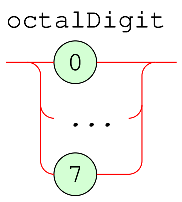
```
letterL :: Lexer
letterL = satisfyL isAlpha "letter"
```

```
digit ::= ("0" | $...$ | "9")
```



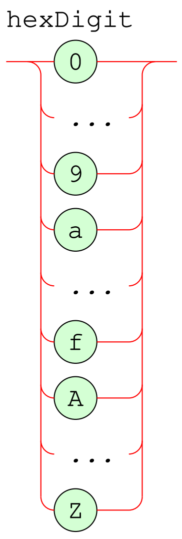
```
digitL :: Lexer
digitL = satisfyL isDigit "digit"
```

```
octalDigit ::= ("0" | $...$ | "7")
```



```
octalDigitL :: Lexer
octalDigitL = satisfyL isOctDigit "octalDigit"
```

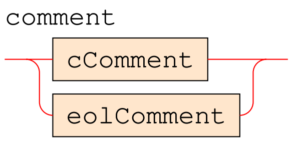
```
hexDigit ::= ("0" | $...$ | "9" | "a" | $...$ | "f" | "A" | $...$ | "Z")
```



```
hexDigitL :: Lexer
hexDigitL = satisfyL isHexDigit "hexDigit"
```

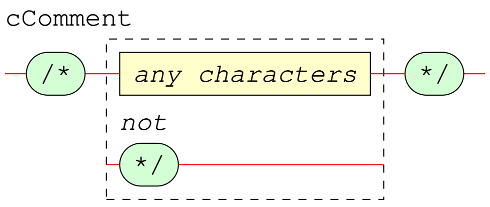
Implemented in `ABR.Parser`.

```
comment ::= (cComment | eolComment)
```



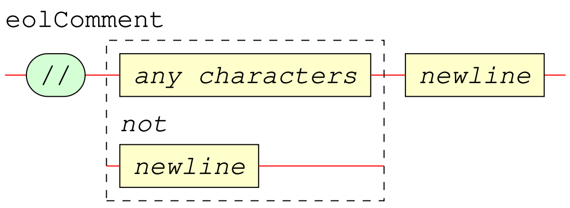
```
commentL :: Lexer
commentL = (cCommentL <|> eolCommentL) %> " "
```

```
cComment ::= "/*" < $any characters$ ! "*/" > "*/"
```



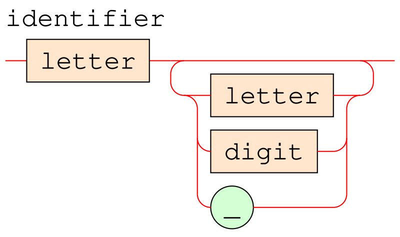
```
cCommentL :: Lexer
cCommentL =
        tokenL "/*"
   <&&> nofail' "incomplete comment" cCommentEndL
   %> "cComment"
   where
   cCommentEndL :: Lexer
   cCommentEndL =
          tokenL "*/"
      <|>      satisfyL (const True) ""
          <&&> cCommentEndL
```

```
eolComment ::= "//" < $any characters$ ! $newline$ > $newline$
```



```
eolCommentL :: Lexer
eolCommentL =
   tokenL "//"
   <&&> (many (satisfyL (/= '\n') "") &%> "")
   <&&> (optional (literalL '\n') &%> "")
   %> "eolComment"
```

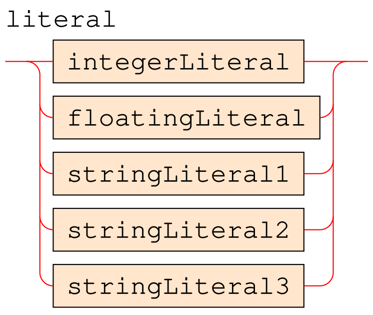
```
identifier ::= letter {letter | digit | "_" }
```



```
identifierL :: Lexer
identifierL =
   ((letterL <|> literalL '_' <|> literalL '$') <&&>
    (many (letterL <|> digitL <|> literalL '_' <|> literalL '$')
     &%> ""))
   %> "identifier"
```

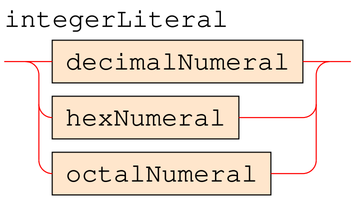
Implement boolean literals as keywords.

```
literal ::= (integerLiteral | floatingLiteral | stringLiteral1 |
stringLiteral2 | stringLiteral3)
```



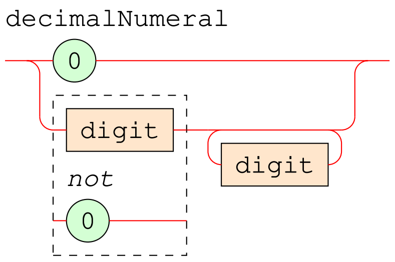
```
mLiteralL :: Lexer
mLiteralL =      floatingLiteralL
            <|> integerLiteralL
            <|> stringLiteral1L
            <|> stringLiteral2L
            <|> stringLiteral3L
```

```
integerLiteral ::= (decimalNumeral | hexNumeral | octalNumeral)
```



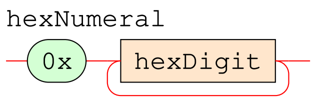
```
integerLiteralL :: Lexer
integerLiteralL =
   (     hexNumeralL
     <|> octalNumeralL
     <|> decimalNumeralL
   )
   %> "integerLiteral"
```

```
decimalNumeral ::= "0" | <digit ! "0"> {digit}
```



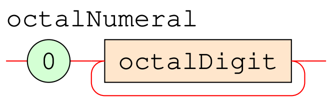
```
decimalNumeralL :: Lexer
decimalNumeralL = (
          literalL '0'
     <|>       digitL `alsoNotSat` literalL '0'
          <&&> (many digitL &%> "")
   ) %> "decimalNumeral"
```

```
hexNumeral ::= "0x" {hexDigit}+
```



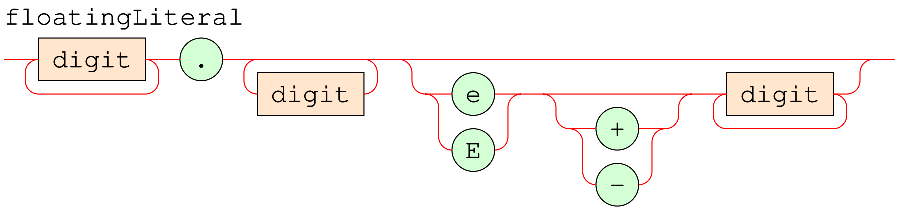
```
hexNumeralL :: Lexer
hexNumeralL =
        tokenL "0x"
   <&&> (many hexDigitL &%> "")
   %> "hexNumeral"
```

```
octalNumeral ::= "0" {octalDigit}+
```



```
octalNumeralL :: Lexer
octalNumeralL =
        literalL '0'
   <&&> (many octalDigitL &%> "")
   %> "octalNumeral"
```

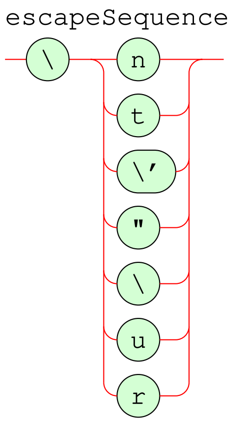
```
floatingLiteral ::= {digit}+ "." {digit} [("e" | "E") [("+" | "-")]
{digit}+]
```



```
floatingLiteralL :: Lexer
floatingLiteralL =
        (some digitL &%> "")
   <&&> literalL '.'
   <&&> (many digitL &%> "")
   <&&> (optional (
                (literalL 'e' <|> literalL 'E')
           <&&> (optional (     literalL '+'
                            <|> literalL '-')
                 &%> "")
           <&&> (nofail' "exponent expected."
                   (some digitL) &%> "")
        ) &%> "")
   %> "floatingLiteral"
```

Implement boolean literals as keywords.

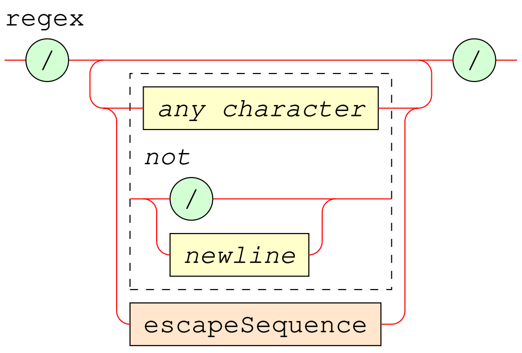
```
escapeSequence ::= "\\" ("n" | "t" | "\'" | "\"" | "\\" | "u" | "r")
```



```
escapeSequenceL :: Lexer
escapeSequenceL =
        literalL '\\'
   <&&> nofail' "bad character escape sequence" (
                literalL 'n'
            <|> literalL 't'
            <|> literalL '\''
            <|> literalL '"'
            <|> literalL '\\'
            <|> literalL 'u'
```

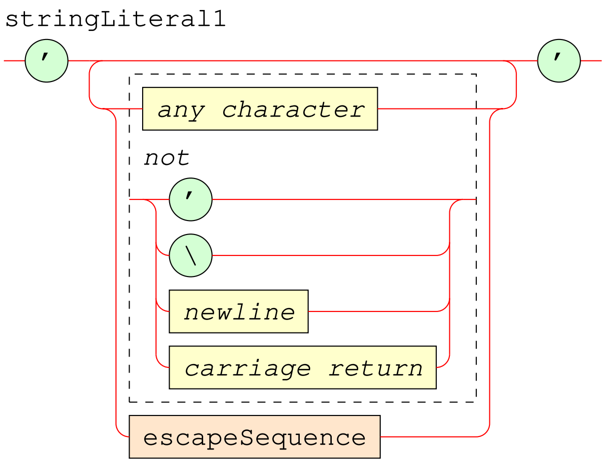
```
          <|> literalL 'r'
        )
   %> "escapeSequence"
```

```
regex ::= "/" {< $any character$ ! "/" | $newline$> | escapeSequence} "/"
```



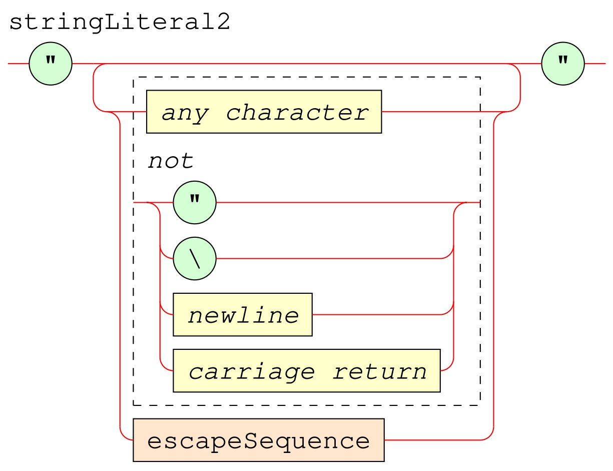
```
regexL :: Lexer
regexL =
        literalL '/'
   <&&> (many (
                  satisfyL (`notElem` "\n/") ""
              <|> escapeSequenceL
           ) &%> "")
   <&&> (literalL '/')
   %> " "
```

```
stringLiteral1 ::= "'" {< $any character$ ! "'" | "\\" | $newline$ |
$carriage return$> | escapeSequence} "'"
```



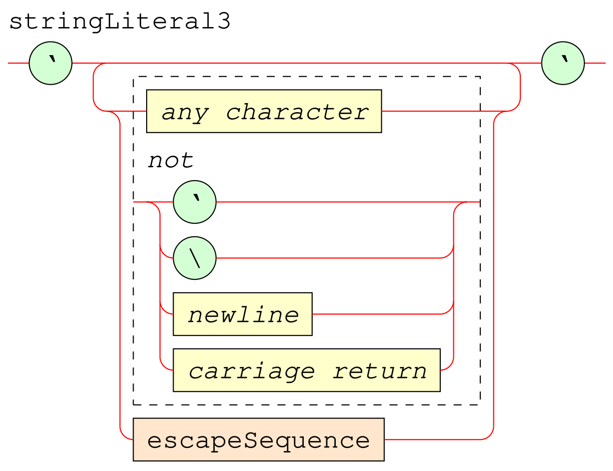
```
stringLiteral1L :: Lexer
stringLiteral1L =
        literalL '\''
   <&&> (many (
                  satisfyL (`notElem` "\'\\\n\r") ""
              <|> escapeSequenceL
           ) &%> "")
   <&&> nofail (literalL '\'')
   %> "stringLiteral"
```

```
stringLiteral2 ::= "\"" {< $any character$ ! "\"" | "\\" | $newline$ |
$carriage return$> | escapeSequence} "\""
```



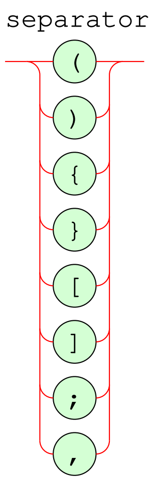
```
stringLiteral2L :: Lexer
stringLiteral2L =
        literalL '\"'
   <&&> (many (
                  satisfyL (‘notElem‘ "\"\\\n\r") ""
              <|> escapeSequenceL
           ) &%> "")
   <&&> nofail (literalL '\"')
   %> "stringLiteral"
```

```
stringLiteral3 ::= "‘" {< $any character$ ! "‘" | "\\" | $newline$ |
$carriage return$> | escapeSequence} "‘"
```



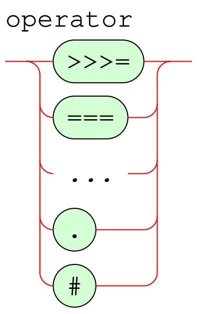
```
stringLiteral3L :: Lexer
stringLiteral3L =
        literalL '‘'
   <&&> (many (
                  satisfyL (‘notElem‘ "‘\\\n\r") ""
              <|> escapeSequenceL
           ) &%> "")
   <&&> nofail (literalL '‘')
   %> "stringLiteral"
```

```
separator ::= ("(" | ")" | "{" | "}" | "[" | "]" | ";" | ",")
```



```
separatorL :: Lexer
separatorL = (
          literalL '('
      <|> literalL ')'
      <|> literalL '{'
      <|> literalL '}'
      <|> literalL '['
      <|> literalL ']'
      <|> literalL ';'
      <|> literalL ','
   ) %> "separator"
```

```
operator ::= (">>>=" | "===" | $...$ | "." | "#")
;level="lexical".
```



```
operatorL :: Lexer
operatorL = (
          tokenL ">>>="
      <|> tokenL "==="
      <|> tokenL "!=="
      <|> tokenL "<<="
      <|> tokenL ">>="
      <|> tokenL "**="
      <|> tokenL ">>>"
      <|> tokenL "++"
      <|> tokenL "--"
```

```
    <|> tokenL "+="
    <|> tokenL "-="
    <|> tokenL "*="
    <|> tokenL "/="
    <|> tokenL "%="
    <|> tokenL "!="
    <|> tokenL "<="
    <|> tokenL ">="
    <|> tokenL "&="
    <|> tokenL "|="
    <|> tokenL "^="
    <|> tokenL "&&"
    <|> tokenL "||"
    <|> tokenL "**"
    <|> tokenL "=="
    <|> tokenL "<<"
    <|> tokenL ">>"
    <|> literalL '\\'
    <|> literalL '+'
    <|> literalL '-'
    <|> literalL '*'
    <|> literalL '/'
    <|> literalL '%'
    <|> literalL '='
    <|> literalL '>'
    <|> literalL '<'
    <|> literalL '!'
    <|> literalL '&'
    <|> literalL '|'
    <|> literalL '~'
    <|> literalL '^'
    <|> literalL '?'
    <|> literalL ':'
    <|> literalL '.'
    <|> literalL '#'
  ) %> "operator"
```

3.2.3 Keywords

`keywords`, `separators` and `operators` are the keywords, operators and separators within the JavaScript environment language respectively – which are used when generating the `Token` maps for the JavaScript language, allowing these elements to be identified and mapped to their respective `Tokens` when tokenizing a file.

```
keywords :: [String]
keywords =
  ["await", "break", "case", "catch", "class", "const", "continue",
   "debugger", "default", "delete", "do", "else", "enum", "export",
   "extends", "false", "finally", "for", "function", "if", "implements",
   "import", "in", "instanceof", "interface", "let", "new", "null", "package",
   "private", "protected", "public", "return", "super", "switch", "this",
   "throw", "try", "true", "typeof", "var", "void", "while", "with",
   "yield"]
```

```
separators :: [String]
separators =
   ["(", ")", "{", "}", "[", "]", ";", ","]

operators :: [String]
operators = [">>>=", "===", "!==", "<<=", ">>=", "**=", ">>>", "++",
             "--", "+=", "-=", "*=", "/=", "%=", "!=", "<=", ">=", "&=",
             "|=",  "^=", "&&", "||", "**", "==", "<<", ">>", "+", "-",
             "*", "/", "%", "=", ">", "<", "!", "&", "|", "~", "^", "?",
             ":", ".", "\\", "#"]
```

`tagKeywords` retags identifiers as keywords.

```
tagKeywords :: [String] -> Lexer -> Lexer
tagKeywords keywords = (@> map tagKeyword)
   where
   tagKeyword :: ((Tag,Lexeme),Pos) -> ((Tag,Lexeme),Pos)
   tagKeyword tlp = case tlp of
      (("identifier", l), p)
         | l `elem` keywords -> (("keyword", l), p)
         | otherwise         -> tlp
      _                      -> tlp
```

3.3 Token

The `Token` module provides the functions required for the tokenization of program sources. Tokenization involves converting the lexemes of a file into `Tokens`, which encapsulate all the lexemes of a language into one of 3 types (keywords, literal values and identifiers) – which allow for more effective comparison of files to detect similarities.

```
module Zimm.Token (
      Token(..), makeMapsKey, makeMapsLiteral, makeMapsId, tokenizeFile
   ) where

import Data.List as L
import Data.Map.Strict as M

import ABR.Util.Pos
import ABR.Parser
```

A `Token` is used to identify the type of a lexeme (keywords, literal values and identifiers). The keyword variant includes all the keywords, separators and operators associated with the language.

```
data Token =   Key Int
             | Literal Int
             | Id Int
   deriving(Eq, Ord, Show)
```

A `TokenMap` is a map between the Lexemes of a file and their associated `Token`.

```
type TokenMap = M.Map Lexeme Token
```

`makeMapsKey lexs` returns a `TokenMap` for all of the keywords, separators, and operators of the language - where `lexs` contains a list of all the keywords, separators and operators for the language. All files will share the same tokens for these language elements.

```
makeMapsKey :: [Lexeme] -> TokenMap
makeMapsKey lexs = fromList $ L.zip lexs $ L.map Key [0..]
```

`makeLiteralToken n m (tag, lex)` adds a new `Token` to the `TokenMap` if `(tag, lex)` is a new numeric constant - where `m` is the `TokenMap` and `n` is the next token value to use. `makeLiteralToken` then returns the new map and the next token value to use. All files should have common tokens for numeric constants.

```
makeLiteralToken :: Int -> TokenMap -> (Tag, Lexeme) -> (Int, TokenMap)
makeLiteralToken n m (tag, lex) =
   let addOne = case M.lookup lex m of
          Nothing -> (n + 1, M.insert lex (Literal n) m)
          Just _  -> (n, m)
   in case tag of
      "integerLiteral"  -> addOne
      "floatingLiteral" -> addOne
      _                 -> (n, m)
```

`makeMapsLiteral m tlpss` returns a new `TokenMap` where all of the constants from all lexed files have been added to the map, where `m` is the incomplete `TokenMap` and `tlpss` are the `TLPs` from all files that were lexed.

```
makeMapsLiteral :: TokenMap -> [[((Tag, Lexeme), Pos)]] -> TokenMap
makeMapsLiteral m tlpss =
   let addOne (n, m) tlp = makeLiteralToken n m (fst tlp)
       addFile (n, m) tlps = L.foldl addOne (n, m) tlps
   in snd $ L.foldl addFile (1, m) tlpss
```

`makeIdToken n m (tag, lex)` adds a new `Token` to the `TokenMap` if `(tag, lex)` is a new identifer - where `m` is the incomplete `TokenMap` and `n` is the next token value to use. `makeIdToken` then returns the new `TokenMap` and the next token value to use.

```
makeIdToken :: Int -> TokenMap -> (Tag, Lexeme) -> (Int, TokenMap)
makeIdToken n m (tag, lex) =
   let addOne = case M.lookup lex m of
          Nothing -> (n + 1, M.insert lex (Id n) m)
          Just _  -> (n, m)
   in case tag of
      "identifier"  -> addOne
      _            -> (n, m)
```

`makeMapsId m tlps` returns a new `TokenMap` where all of the identifiers for a single lexed file have been added to the map, where `m` is the incomplete `TokenMap` and `tlps` are the tagged lexemes / positions for the associated file.

```
makeMapsId :: TokenMap -> [((Tag, Lexeme), Pos)] -> TokenMap
makeMapsId m tlps =
   let addOne (n, m) tlp = makeIdToken n m (fst tlp)
   in snd $ L.foldl addOne (0, m) tlps
```

`tokenizeFile m tlps` returns the list of `Tokens` and their positions for a single lexed file, where `m` is a completed `TokenMap` linking all the lexemes for a single file to their respective `Tokens` and `tlps` are the tagged lexemes / positions of the file to be tokenized. String literals all have the same token (Literal 0) (not in the map).

```
tokenizeFile :: TokenMap -> [((Tag, Lexeme), Pos)] -> [(Token, Pos)]
tokenizeFile m tlps =
   let tokenize ((tag, lex), pos) = case tag of
         "stringLiteral" -> (Literal 0, pos)
         _               -> case M.lookup lex m of
            Nothing -> error $ "tokenizeFile: unknown lex: "++ show tag ++
               show lex ++ show pos
            Just t  -> (t, pos)
   in L.map tokenize tlps
```

3.4 Matching

The `Matching` module provides the cross-matching of tokenised files, detecting any sequences of consecutive matching `Tokens` between tokenized files whose length is greater than or equal to a minimum required length.

```
module Zimm.Matching (
      Run(..), findRuns, filterRuns
   ) where

import Data.List as L
import Data.Map.Strict as M
import Data.Set as S

import ABR.Util.Pos

import Zimm.Token
```

`match ts ts'` compares 2 lists of `Tokens` and determines the number of matching consecutive `Tokens` from the start of each list, returning the (len, lastPos1, lastPos2) – where len is the length of matching tokens and lastPos1 / lastPos2 are the last matching positions in the respective lists. If the first `Tokens` in each list do not match, (0, (-1, -1), (-1,-1)) will be returned. When comparing `Tokens`, identifiers are mapped to an index according to the order they appear within their respective list of `Tokens`, which is a form of unification that allows the algorithm to detect matches even when identifier names have been changed between files.

```
match :: [(Token, Pos)] -> [(Token, Pos)] -> (Int, Pos, Pos)
match ts ts' =
   let len = match' 0 ts 0 M.empty ts' 0 M.empty
       lastPos1 | len == 0  = (-1, -1)
                | otherwise = snd $ ts !! (len - 1)
       lastPos2 | len == 0  = (-1, -1)
                | otherwise = snd $ ts' !! (len - 1)
   in (len, lastPos1, lastPos2)
   where
   match' :: Int -> [(Token, Pos)] -> Int -> M.Map Int Int
              -> [(Token, Pos)] -> Int -> M.Map Int Int -> Int
   match' n t1s v1 m1 t2s v2 m2
      | L.null t1s || L.null t2s = n
      | otherwise                =
         let reassign :: Token -> Int -> M.Map Int Int ->
                         (Token, Int, M.Map Int Int)
             reassign t v m = case t of
                Id i -> case M.lookup i m of
```

```
                  Nothing -> (Id v, v + 1, M.insert i v m)
                  Just j  -> (Id j, v, m)
               _ -> (t, v, m)
            (t1, _) : tl1 = t1s
            (t2, _) : tl2 = t2s
            (t1', v1', m1') = reassign t1 v1 m1
            (t2', v2', m2') = reassign t2 v2 m2
        in if t1' == t2'
             then match' (n + 1) tl1 v1' m1' tl2 v2' m2'
             else n
```

A `Run` represents a continuous stream of matching `Tokens` that were found between 2 files, where `len` represents the number of matching `Tokens` and `startPos1`, `endPos1`, `startPos2`, `endPos2` represent the positions in the respective files where the stream of matching `Tokens` start and finish.

```
data Run = Run {
      len :: Int,
      startPos1 :: Pos,
      endPos1 :: Pos,
      startPos2 :: Pos,
      endPos2 :: Pos
   }
```

`findRuns n f1 f2` returns all the matching runs of at least `n` `Tokens` from the tokenized files `f1` and `f2`.

```
findRuns :: Int -> [(Token, Pos)] -> [(Token, Pos)] -> [Run]
findRuns n f1 f2 =
   [Run m p1 ep1 p2 ep2
   | t1@((_, p1) : _) <- L.tails f1, t2@((_, p2) : _) <- L.tails f2,
     let (m, ep1, ep2) = match t1 t2, m >= n]
```

`runInclues r r'` checks whether the start and ending positions of run `r'` are encapsulated within the run `r` (meaning that `r'` is effectively a redundant match), returning `True` when the positions of `r'` are encapsulated within the run `r` and `False` otherwise.

```
runIncludes :: Run -> Run -> Bool
runIncludes r r' =
   len r' < len r &&
   startPos1 r < startPos1 r' && startPos2 r < startPos2 r' &&
   endPos1 r >= endPos1 r' && endPos2 r >= endPos2 r'
```

`filterRuns runs` checks a list of runs for any that are redundant (any run having start and end positions which are encapsulated within the positions of another run), returning the list of runs where all redundant runs have been removed.

```
filterRuns :: [Run] -> [Run]
filterRuns runs = case runs of
   []     -> []
   r : rs -> r : filterRuns (L.filter (\r' -> not (runIncludes r r')) rs)
```

3.5 template.html

The template.html file is used to provide the scaffolding for the HTML report that summarises the results of compared files, which was achieved through utilizing the ABR.Text.Markup and ABR.Text.Configs libraries [2].

3.6 Zim Installation

In order to install Zim on your machine, please follow these steps:

1. Download the Zim package files on your local machine.
2. Install the most recent Glasgow Haskell Compiler (GHC) on your system, which includes cabal.
3. Install the System.FilePath.Glob package on your system using the command `cabal install glob`.
4. Enter the zim/src directory within your local shell console and enter the command `make`. This will compile all the modules required to use `Zim`.

3.7 Zim User Guide

Before attempting to use Zim, please ensure all modules have been compiled (see Zim Installation).

3.7.1 Command Syntax

To use Zim to compare files, navigate to the zim/src directory and use the following command structure within your shell console:

```
zim [-run <run length>] [<+->desc] [-lang <language>] <filepaths>
```

where [ ] indicate options / flags and <> denotes user input.

Command Options / Flags:

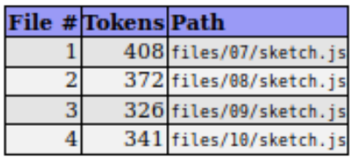
1. `run <run length>` – option used to determine the required minimum length of consecutive matching tokens before it is considered a matching run. The default value is 20.
2. `+desc` – flag which will order the matches table of the HTML report in descending order according to the number of tokens which matched between the files. This is the default for the desc flag.
3. `-desc` – flag which will order the matches table of the HTML report in the order that they were detected by Zim.
4. `lang <language>` – option used to determine the programming language which will be used as the basis for comparison. The default value is JavaScript (currently the only language supported).

Note that all input files should match the programming language indicated, otherwise it will cause a mismatch between the elements of the language and those of the files compared, resulting in ineffective comparisons between files.

3.7.2 Example Commands:

```
zim -run 50 -desc -lang JavaScript files/*.js
```

Compares all JavaScript files within the files/ directory (relative to current directory), using a run length of 50 (meaning atleast 50 consecutive tokens are required to match), ordering the matches table of the HTML report in descending order of matching tokens and using the JavaScript programming language as the basis for comparison.



File #	Tokens	Path
1	408	files/07/sketch.js
2	372	files/08/sketch.js
3	326	files/09/sketch.js
4	341	files/10/sketch.js

Figure 3: A sample list of input files which have been compared by Zim which are presented within the HTML report.

```
zim dir1/file1.js dir2/file2.js dir3/file3.js
```

Compares 3 JavaScript files, using a run length of 20, the JavaScript programming language as the basis for comparison, and ordering the matches table of the HTML report in descending order of matching tokens (all default values for the options / flags).

4 Results

In this section we will present the results of the completed Zim tool, providing examples of the tool being used to detect similiarities between files, and providing the results of the testing we performed to determine the performance of the tool.

4.1 Function

To test the functionality of Zim, we compared 4 files using the Zim tool to detect similarities between them. The results of the matches found between the files are provided within the HTML report and are discussed in the subsections below.

4.1.1 Listing the Files

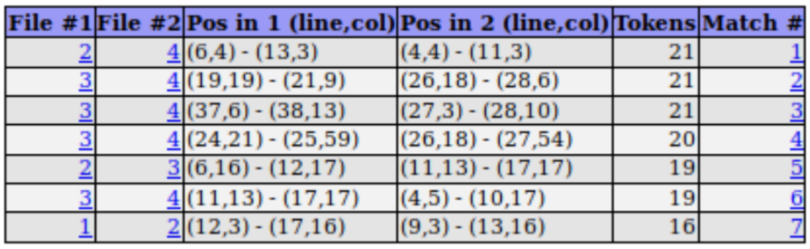
The first section of the HTML report presents: a number which maps each of the file paths to an index; the total number of tokens contained within each file; and the file paths read in by the tool. The indices provide a reference to each file, which are linked to throughout the later sections of the report and will navigate the user back to the associated file when clicked. See figure 3.

4.1.2 Matches

The Matches section of the HTML report presents the matches found between the files compared, providing: the indexes for the files where each match was found; the positions in each file where the start of the matching run occurs; the length of matching tokens for each match; and an index for each match that was found, which can be clicked on to view the source code and tokens between the files that were compared. An example of the Matches table can be seen in figure 4.

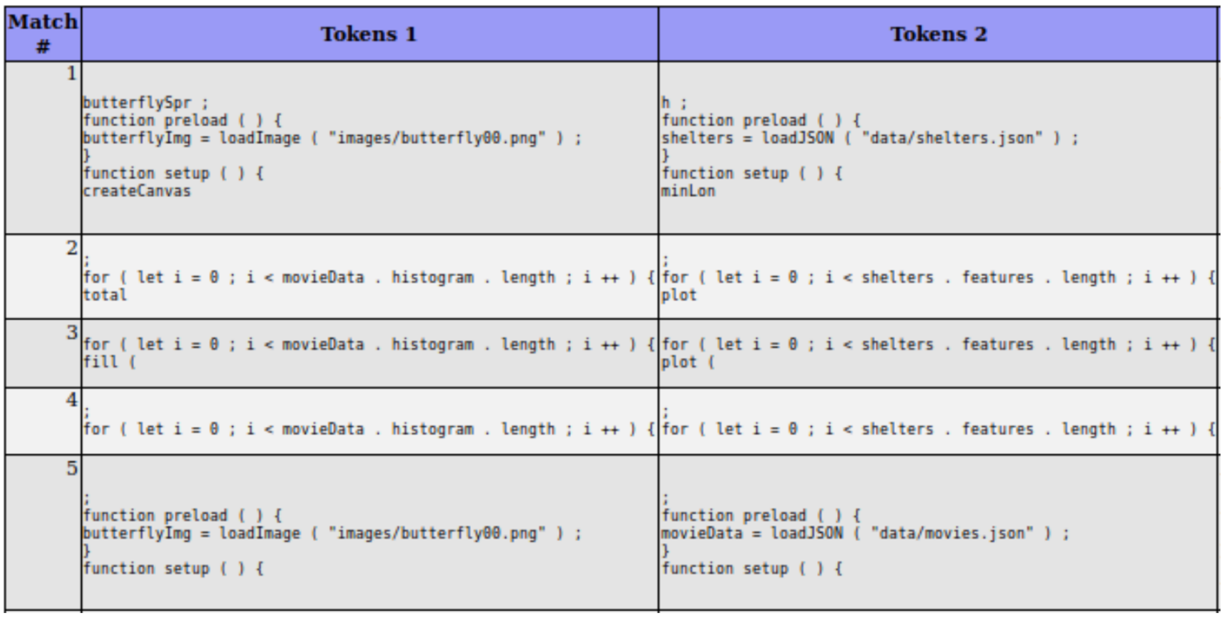
4.1.3 Sources

The Sources section of the HTML report presents the sections of source code and tokens for both files where a matching run occured, presented as a side by side comparison. This allows easy inspection of the matches to determine whether actual plagiarism between the files has occured. The token side by side comparison shows the source code reconstructed from the tokenized stream of TLPs (which ignores comments and unnecessary whitspace), whereas the source code side by side comparison provides the raw source code with all comments and whitespace left intact. An example of the token comparison for a match can be seen in figure 5 and the source code comparison can be seen in figure 6.



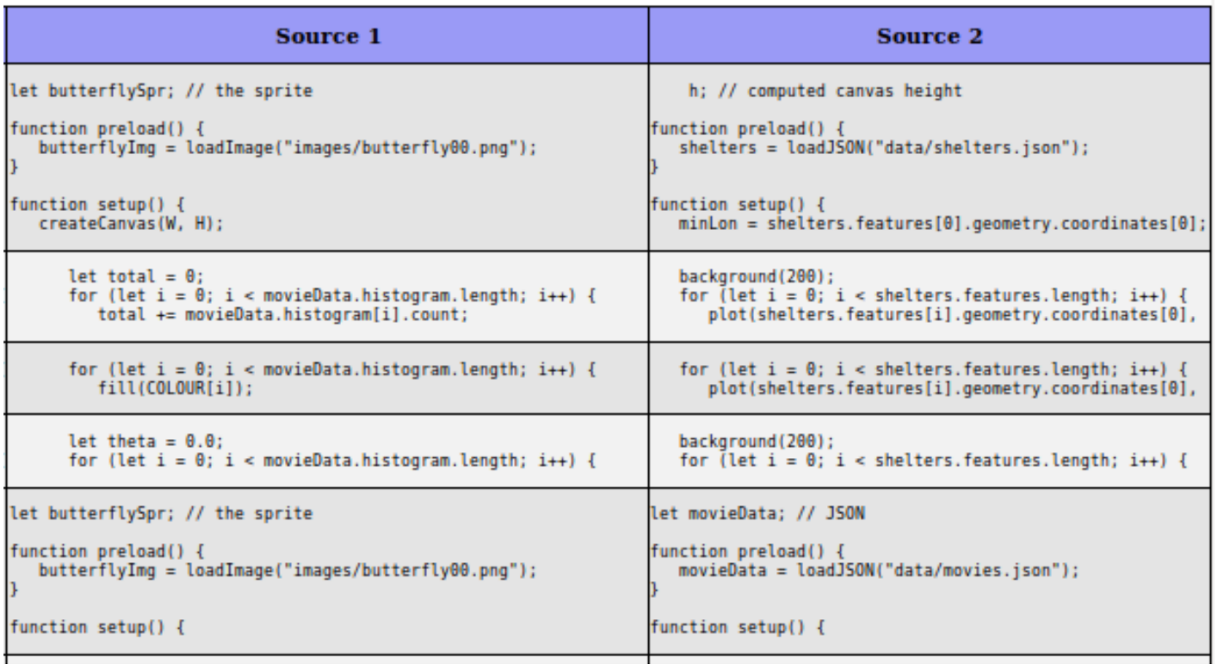
File #1	File #2	Pos in 1 (line,col)	Pos in 2 (line,col)	Tokens	Match #
2	4	(6,4) - (13,3)	(4,4) - (11,3)	21	1
3	4	(19,19) - (21,9)	(26,18) - (28,6)	21	2
3	4	(37,6) - (38,13)	(27,3) - (28,10)	21	3
3	4	(24,21) - (25,59)	(26,18) - (27,54)	20	4
2	3	(6,16) - (12,17)	(11,13) - (17,17)	19	5
3	4	(11,13) - (17,17)	(4,5) - (10,17)	19	6
1	2	(12,3) - (17,16)	(9,3) - (13,16)	16	7

Figure 4: A sample list of matches found between files compared by Zim which are presented within the HTML report.



Match #	Tokens 1	Tokens 2
1	butterflySpr ; function preload ( ) { butterflyImg = loadImage ("images/butterfly00.png") ; } function setup ( ) { createCanvas	h ; function preload ( ) { shelters = loadJSON ("data/shelters.json") ; } function setup ( ) { minLon
2	; for (let i = 0 ; i < movieData . histogram . length ; i ++) { total	; for (let i = 0 ; i < shelters . features . length ; i ++) { plot
3	for (let i = 0 ; i < movieData . histogram . length ; i ++) { fill (	for (let i = 0 ; i < shelters . features . length ; i ++) { plot (
4	; for (let i = 0 ; i < movieData . histogram . length ; i ++) {	; for (let i = 0 ; i < shelters . features . length ; i ++) {
5	; function preload ( ) { butterflyImg = loadImage ("images/butterfly00.png") ; } function setup ( ) {	; function preload ( ) { movieData = loadJSON ("data/movies.json") ; } function setup ( ) {

Figure 5: A sample of the token comparison for matching runs found between files which are presented within the HTML report.



Source 1 | **Source 2**

```
let butterflySpr; // the sprite

function preload() {
   butterflyImg = loadImage("images/butterfly00.png");
}

function setup() {
   createCanvas(W, H);
```

```
  h; // computed canvas height

function preload() {
   shelters = loadJSON("data/shelters.json");
}

function setup() {
   minLon = shelters.features[0].geometry.coordinates[0];
```

```
      let total = 0;
      for (let i = 0; i < movieData.histogram.length; i++) {
         total += movieData.histogram[i].count;
```

```
   background(200);
   for (let i = 0; i < shelters.features.length; i++) {
      plot(shelters.features[i].geometry.coordinates[0],
```

```
      for (let i = 0; i < movieData.histogram.length; i++) {
         fill(COLOUR[i]);
```

```
   for (let i = 0; i < shelters.features.length; i++) {
      plot(shelters.features[i].geometry.coordinates[0],
```

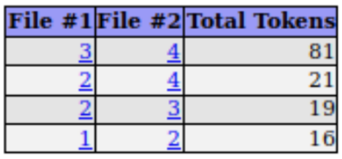
```
      let theta = 0.0;
      for (let i = 0; i < movieData.histogram.length; i++) {
```

```
   background(200);
   for (let i = 0; i < shelters.features.length; i++) {
```

```
let butterflySpr; // the sprite

function preload() {
   butterflyImg = loadImage("images/butterfly00.png");
}

function setup() {
```

```
let movieData; // JSON

function preload() {
   movieData = loadJSON("data/movies.json");
}

function setup() {
```

Figure 6: A sample of the source code comparison for matching runs found between files which are presented within the HTML report.



File #1	File #2	Total Tokens
3	4	81
2	4	21
2	3	19
1	2	16

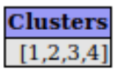
Figure 7: A sample of the Cross Totals which presents the total length of matching runs found between files and are presented within the HTML report.

4.1.4 Cross Totals

The Cross Totals section of the HTML report presents the total length of matching runs that were found between each of the compared files (for files where at least 1 matching run occured), providing a quick indication of the files which have the more severe cases of matching runs. An example of the Cross Totals can be seen in figure 7.

4.1.5 Clusters

The Clusters section of the HTML report presents the groups of files where content may have been shared. Clusters are determined on the basis of transitivity, meaning that if file 1 has matches with file 2 and file 2 has matches with file 3, a cluster (1,2,3) is formed. For our example which compared 4 files, all of the compared files resulted in a cluster. See figure 8.



Clusters
[1,2,3,4]

Figure 8: A sample of the Clusters table which presents groups of files which share similiarities on the basis of transitivity and are presented within the HTML report.

Table 1: Benchmarking results for increasing numbers of files.

Files (number)	Elapsed Time (seconds)	Average File Size (tokens)
2	22	4739
5	151	4012
10	491	3391
20	2286	3391

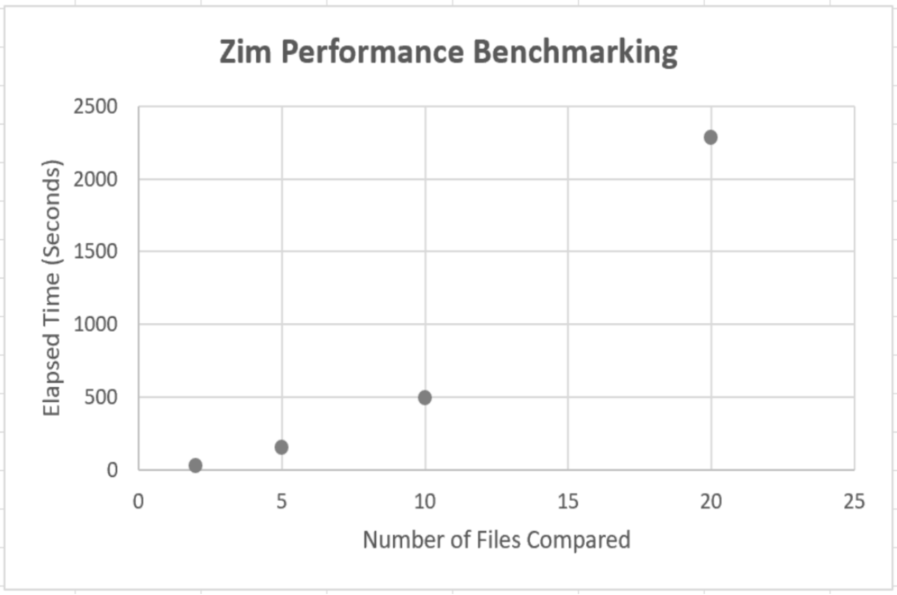


Figure 9: Benchmarking results for increasing numbers of files.

4.2 Performance

To collect result for the performance of the Zim tool, we tested Zim with different numbers of files and recorded the elapsed time it took to compare all the files. This allowed us to analyse how an increasing number of files impacts the performance of the program. The raw results for the benchmarking tests can be seen in table 1 and figure 9.

The results for these tests show that the algorithm used by Zim is quadratic with a complexity of $O(N^2)$ (where N is the number of compared files), which was determined through acquiring a slope of 1.99 when comparing the logarithms of the raw results. See figure 10.

5 Conclusion and Reflections

5.1 Discussion

With regards to the project, Zim is successfully able to detect similarities between files and provides a conclusive summary of the results within the HTML report, which has proven to be an effective aid for detecting plagiarism. The asymptotic performance for the algorithm used in Zim ended up being quadratic with a complexity of $O(N^2)$, which was the expected result based on our project following a similar algorithm to the Sim tool (which was also quadratic) [1].

The core requirements for the project were completed in addition to a few extras to help aid in the detection of plagiarism, such as the option to order the matches of the HTML report according to the number of tokens that matched, the Cross Totals for identifying the total length

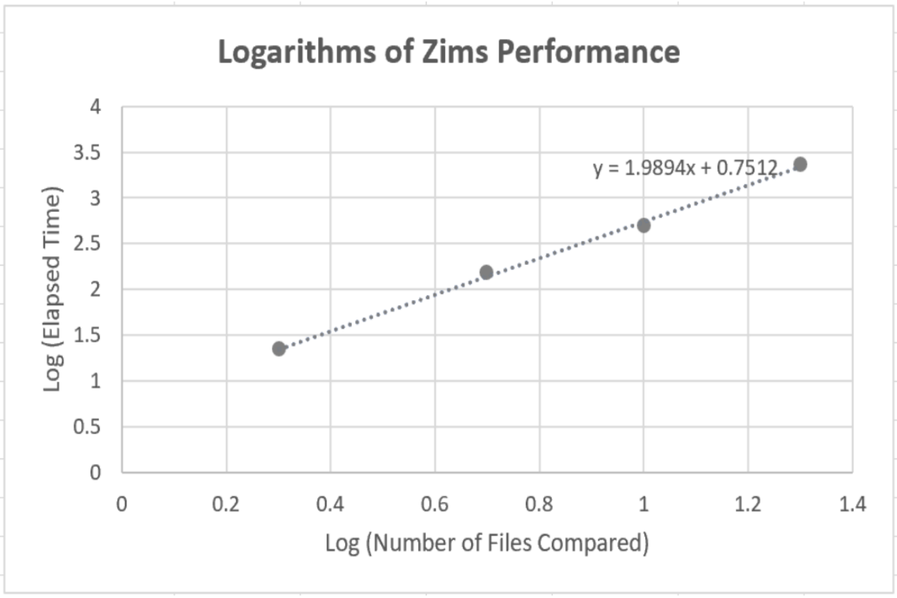


Figure 10: Logarithms of the Zim benchmarking results.

of matches found between similar files, and the Cluster table to identify groups of files which are connected through similarities between them.

5.2 Future Work

The Zim tool was developed with the potential for future expansion into further programming languages. In order for a new language to be added to the tool, an additional Lexer module will be required to translate files from the associated language into its lexemes. The new Lexer module should then allow for files of the new language to be lexed and tokenized, which can then be compared by the Zim tool.

The Zim tool can also be improved to make the algorithm compare files faster, as the current algorithm finds redundant matches (matches where the start and end positions are encapsulated within a previously discovered match) before discarding redundant matches once all the matches for a pair of files have been found. The algorithm would perform faster through being able to skip redundant matches during the comparison, rather then finding them then having to discard them later.

Lastly, we believe the algorithm used for detecting clusters of plagiarised files could be further improved in the future. Currently Zim detects clusters based on transitivity, meaning that if file 1 has matches with file 2 and file 2 has matches with file 3, they form a cluster (1,2,3). However this method doesn't take into account the sections of code that matched. Even though this method is still definitely useful for detecting groups of students who share work amongst each other, we believe the algorithm could be further expanded to detect clusters when the *matches* of tokens between files are similar, which would be an effective way to detect cases where a section of code is copied amongst several files, which is a common case amongst plagiarising students.

5.3 Reflection

Heading into the Functional Programming Course at the start of the trimester, I was only really expecting to learn a new 'skill' that I could apply within my programming career (how to code using functional programming languages), but I actually ended up learning more then I could have ever anticipated.

I was able to learn functional programming, how to write makefiles, and LaTeX documents, but my most significant takeaway from the course was the profound impact it had on my perspective towards different programming concepts and paradigms, which I am confident have significantly improved my capabilities as a programmer.

Programming in Haskell really helped me to understand the pitfalls of Object Oriented Programming languages, in particular, how bugs and defects are introduced into Object Oriented programs through the use of shared mutable state (entities which can change their value during runtime) and how shared mutable state makes it difficult to follow large sections of code (due to difficulty tracking the state of an entity). Furthermore, the use of shared mutable state frequently tends to make it very difficult to maintain large libraries of code, which can often lead to code having to be completely rewritten when bugs are introduced or when modifications are required.

Understanding the impact of shared mutable state has greatly improved the way I design and write code in Object Oriented languages, as I am now constantly mindful of how the code I write may impact external programs, which leads me to writing code that is better encapsulated, more predictable, easier to follow and easier to modify.

Another significant benefit I gained from the course was the impact it had on my perspective towards writing functions. Haskell has a distinct way of incentivising *small* functions (as opposed to large functions that perform a large amount of operations). Although I had to adjust initially (as I was used to writing large functions), after a while writing small functions became *natural*. Now when thinking about a problem, I find myself mentally breaking it down into smaller functions / sub-problems, which has made writing code and solving problems a lot easier. Furthermore, after I had been writing small functions in Haskell for a while, I began to notice and really appreciate just how robust and reusable the code I was writing was.

Going through the course also helped me to gain a much better understanding of all the concepts and paradigms of functional programming and how to utilize them effectively. In particular, higher order functions, pattern matching, immutable state, predicates, list comprehensions, recursion, lambda functions, auxiliary functions, auxiliary expressions, referential transparency, monads, functors and lastly what I call 'pipelines' (transforming an input through piping it through multiple functions to acquire the desired result).

I feel Haskell was a key aspect of the course as it is a *pure* functional programming language (meaning that it enforces the use of functional paradigms). Past courses introduced a handful of functional programming concepts, but I feel were not taught effectively due to the courses using multi-paradigm languages such as Swift or Python, which meant I could easily revert to Object Oriented style programming whenever I wanted. Not having this option in Haskell forced me to break out of that comfort zone and practice the functional programming paradigms, which contributed significantly to my learning and personal growth.

Haskell is also a very strict language, with a lot of rules being enforced by the compiler. During the early stages of the course this was difficult to deal with, mainly because the compiler would often complain about issues surrounding concepts that I had not yet been taught (such as pattern matching), which in turn made it difficult to address or understand these issues within my code. But as I became familiar with the concepts, not only did these issues become easier to address but I began to appreciate the strictness of the language and how useful this was as a tool for writing good code, where the only errors I would come across were of my own doing and not a fault of the language.

My biggest personal difficulty with the course was when Monads were introduced, where reading up on the concept / listening to people trying to explain it didn't seem to help - but as I began to write more and more code the concept finally started to make sense.

Even though the aforementioned challenges naturally presented a very large learning curve for me, I can now fully appreciate how necessary they were for me to gain the growth I was able to achieve throughout the course.

Lastly, I was really pleased that the Zim project was completed to a standard where it could be used to achieve a useful purpose (detecting plagiarism), and was not just a 'gimmick' designed for learning purposes. I feel projects like this provide me with better preparation for post-graduate work in the IT industry, as they give a more realistic expectation of the type of work I could be asked to undertake (i.e products which are intended to be used by a customer).

To summarize I feel extremely lucky and grateful to have participated in this course, where I would go so far as to say this course has offered the most value from all the courses I have participated in throughout my degree.

References

[1] Dick Grune. Concise Report on the Algorithms in Sim. https://dickgrune.com/Programs/similarity_tester/TechnReport. Accessed: 2019-10-18. 2, 5.1

[2] Andrew Rock. ABR Haskell Libraries. https://www.ict.griffith.edu.au/arock/ABRHLibs/doc/ABRHLibs.pdf. Accessed: 2019-10-18. 3.2, 3.5