UNIVERSIDAD NACIONAL DE ASUNCIÓN

FACULTAD POLITÉCNICA – SEDE SAN LORENZO

Tokenizadores en el Procesamiento de Lenguajes Naturales (NPL)

DISCIPLINA: Diseño de Compiladores

Profesor: Sergio Andres Aranda Zeman

Alumno: Manuel René Pauls Toews

Fecha de presentación: 24 de junio de 2024

SAN LORENZO, 2024

Índice

Introducción

El procesamiento de lenguajes naturales (NPL) enfrenta varios desafíos debido a las ambigüedades y las expresiones idiomáticas propias de diferentes regiones geográficas y contextos específicos. Los analizadores léxicos deben manejar estas situaciones, apoyándose en herramientas como preprocesadores léxicos para identificar palabras. Los tokenizadores juegan un papel crucial al identificar y organizar palabras, simplificando el proceso de análisis sintáctico posterior.

El objetivo de este trabajo práctico es construir un tokenizador mínimo, MNLPTK (Minimal Natural Language Processing Tokenizer), que pueda identificar palabras (lexemas) en un texto de entrada en idioma inglés y organizarlas en una estructura de datos que sirva como diccionario. El tokenizador aceptará uno o más archivos de texto como entrada y los procesa con la ayuda del usuario del programa. En este proceso se genera un diccionario que pueda ser usado en ejecuciones futuras, como también un entrada en formato de tokens de las entradas procesadas.

Decisiones Adoptadas

Idioma

Se decidió desarrollar el trabajo práctico en inglés debido a que permite demostrar mejoras más fácilmente y porque el autor maneja mejor este idioma.

Lenguaje de Programación

Se optó por utilizar Go (Golang) debido a varias razones importantes:

- **Manejo de Strings**: Go ofrece un manejo eficiente y a la vez fácil de entender de cadenas de caracteres, lo cual es crucial para el procesamiento de lenguajes naturales.
- **Seguridad de Memoria**: La gestión de memoria en Go reduce la posibilidad de errores como los desbordamientos de buffer.
- **Eficiencia**: Go es un lenguaje compilado que produce código de máquina altamente eficiente. Esto puede ser de gran importancia, dependiendo de la longitud y el número de entradas a ser procesadas.
- **Soporte para Multithreading**: Go facilita la concurrencia y el paralelismo, lo cual es beneficioso para procesar grandes volúmenes de texto de manera rápida y eficiente. Además esto en conjunto con el uso de los channels permite la separación de lógica y UI que se explicará más adelante en este trabajo.
- **Formato CSV para Resultados**: Los resultados se guardan en formato CSV para facilitar su visualización con programas como Excel. Esto permite una fácil manipulación y análisis de los datos generados por el tokenizador.
- **Diccionario en JSON o YAML**: El diccionario puede guardarse en formatos JSON o YAML. JSON es ampliamente utilizado y compatible con muchas aplicaciones, mientras que YAML es más legible para los humanos y permite comentarios, facilitando la documentación. La flexibilidad en los formatos de salida hace que la herramienta sea adaptable a diferentes necesidades de desarrollo y análisis. La detección de formato se hace por la extensión del nombre de archivo. En caso de encontrar un formato no soportado, se avisa esto al usuario y se permite volver a elegir otro archivo o cambiar el nombre.

Mejoras Implementadas

1. **Conjugación Automática de Verbos:**
    - Los verbos se conjugan automáticamente en tercera persona singular. Por ejemplo, al agregar "fly" también se incluye "flies". Aunque se hace un esfuerzo considerable, algunas conjugaciones podrían fallar. Esto mejora la precisión del análisis léxico al considerar variaciones comunes de los verbos.
2. **Formas Plurales:**
    - Se agregan las formas plurales de los sustantivos. Por ejemplo, al agregar "fish" también se incluye "fishes". Nuevamente, se acepta que en algunas situaciones

esto puede no funcionar perfectamente. Esta mejora asegura que el diccionario de datos sea más completo y útil.

3. **Interfaz de Usuario Gráfica:**
    - Se programó una interfaz de usuario gráfica (GUI) que es amigable al usuario, permite ver mucha información al mismo tiempo y es similar a otras interfaces que un usuario debería conocer ya de otras aplicaciones.

Características Adicionales

1. **Procesamiento de Múltiples Archivos de Entrada**:
    - El tokenizador soporta la entrada de múltiples archivos, procesándolos secuencialmente. Esta capacidad permite manejar grandes volúmenes de datos de manera eficiente y fácil para el usuario.
2. **Ejecución en GUI o Consola**:
    - El programa puede ejecutarse tanto en una interfaz gráfica (GUI) como en la consola. La lógica está completamente aislada del código de visualización e interacción mediante el uso de channels. Esta separación garantiza que las mejoras o cambios en la lógica de procesamiento no afecten la interfaz de usuario y viceversa. Esta separación es posible gracias al uso de subrutinas separadas para el manejo de la interacción del usuario y el procesamiento de texto. La comunicación entre las subrutinas se hace a través de canales de Go.
3. **Librería Fyne para UI**:
    - Se utilizó la librería Fyne (fyne.io) para el desarrollo de la interfaz gráfica debido a su adaptabilidad al tamaño de pantalla, facilidad de uso, desarrollo rápido y soporte multiplataforma. Fyne permite crear interfaces de usuario modernas y adaptativas con un mínimo esfuerzo, lo cual es ideal para este proyecto.
4. **Información del Run**:
    - Al finalizar la ejecución, se muestra información relevante como la cantidad de nuevas palabras agregadas, el tiempo utilizado y más. Esta funcionalidad proporciona al usuario una visión clara del rendimiento y eficacia del tokenizador.

Metodología

Se utilizaron los conocimientos impartidos en clase para construir el tokenizador. Para la detección de palabras individuales se usó una expresión regular, donde palabras en un archivo de entrada se definen como partes de texto separados por signos de puntuación o espacios en blanco.

Para la detección de categoría de cada palabra se usa una tabla hash, ya que el patrón en este caso es exclusivamente por extensión y no por comprensión. Así la tabla hash relaciona cada palabra conocida a una categoría de palabras. Se optó por usar la función hash por defecto del compilador, ya que es razonable que un buen compilador en cada sistema operativo debería

tener una buena implementación para cadenas de texto y no se vió una necesidad de reinventar la rueda en este caso.

Programa

El código fuente completo del trabajo se encuentra disponible en https://github.com/QuisVenator/compi-tp.

Estructura del proyecto

Para la estructura del proyecto se ha seguido las mejores prácticas del lenguaje go. Este se explica a continuación:

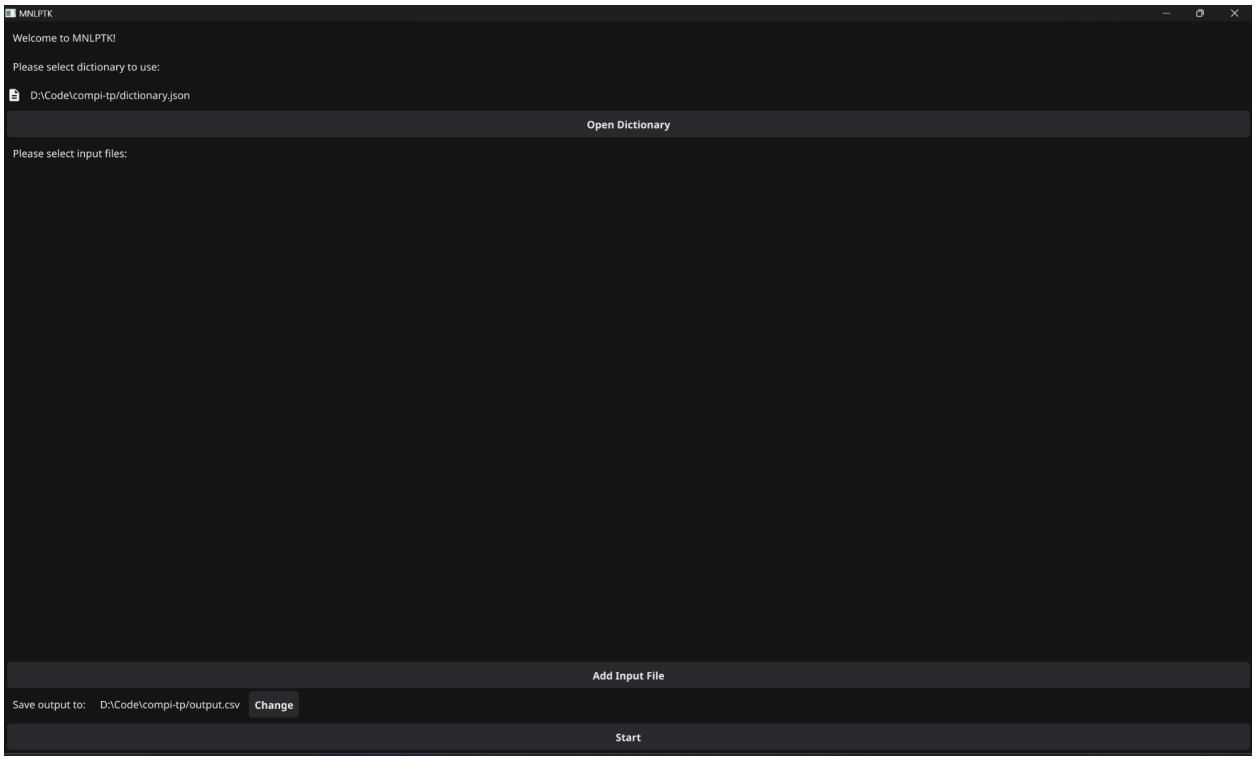
- La carpeta vendor: Esta carpeta contiene todas las dependencias del proyecto. Se cometen todas las librerías usadas al repositorio para asegurar que el proyecto sea ejecutable aún si cambian las librerías utilizadas originalmente.
- La carpeta cmd: Esta carpeta contiene los puntos de entrada “main.go”. Se tiene un punto de entrada en gui/ para la aplicación con interfaz gráfica y otro en cli/ para la aplicación en consola.
- La carpeta tokenizer:
    - tokenizer.go: Contiene la lógica para comunicar con la interfaz a través de los canales y tokenizar la entrada mediante el diccionario dado. Es el punto de entrada para la lógica de negocio del programa y encargado de guardar los resultados.
    - dictionary.go: Contiene la lógica para cargar, manipular y guardar el diccionario de palabras conocidas.
- El archivo .gitignore: Define una lista de patrones a ser ignorados por el manejador de versiones.
- El archivo dictionary.json: Un archivo que se puede usar como diccionario inicial.
- Los archivos go.mod y go.sum: Archivos autogenerados en Go, que definen los paquetes utilizados y sus versiones.
- El archivo LICENSE: Define la licencia usada del repositorio. Se debe cuidar que el projecto incluye la librería fyne.io que tiene su propia licencia, pero que es compatible con la licencia usada también.
- El archivo README.md: Archivo en formato MARKDOWN que describe brevemente cómo compilar y ejecutar el programa.

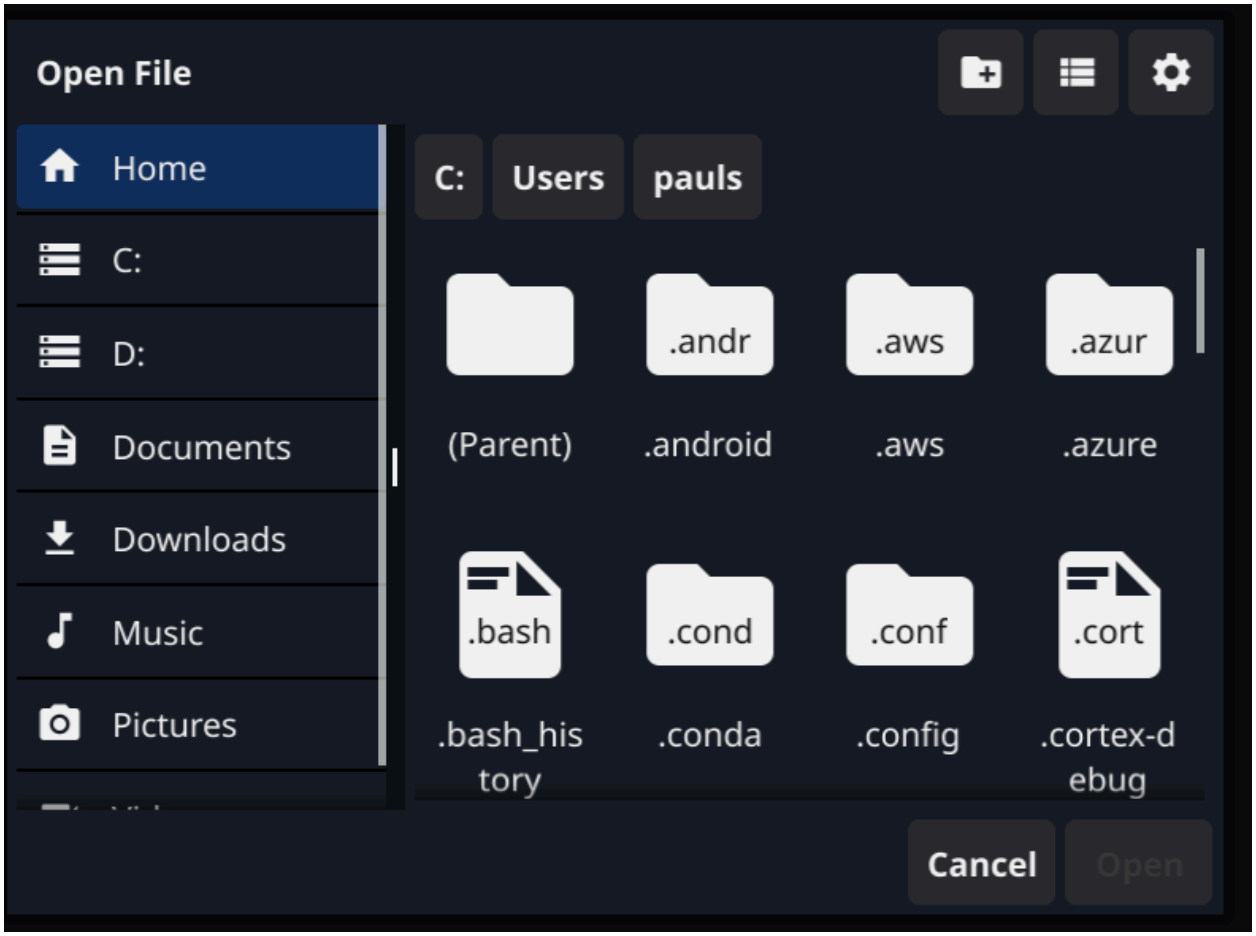
Ejemplo

Usaremos como ejemplo el texto “This is a random text”. Para el diccionario usaremos un diccionario precargado que ya tiene todas las palabras del texto, menos “random”. Cuando ejecutamos el programa en modo consola, tendremos la siguiente salida:

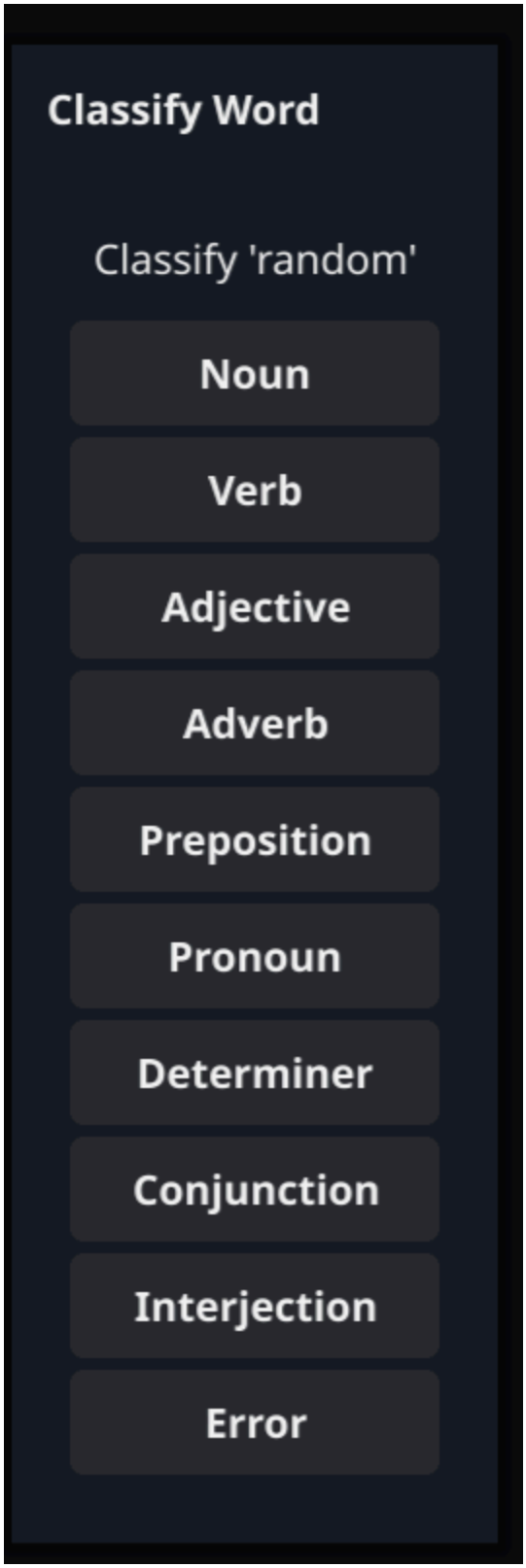
```
PS D:\Code\compi-tp> .\MNTPTK-CLI-v1.exe .\test.txt
this: Pronoun
is: Verb
Please classify the word 'random':
1: Noun
2: Verb
3: Adjective
4: Adverb
5: Preposition
6: Pronoun
7: Determiner
8: Conjunction
9: Interjection
10: Error
```

Después de ingresar 3, se procesa el texto restante.
En modo GUI, primero debemos elegir el archivo a procesar:

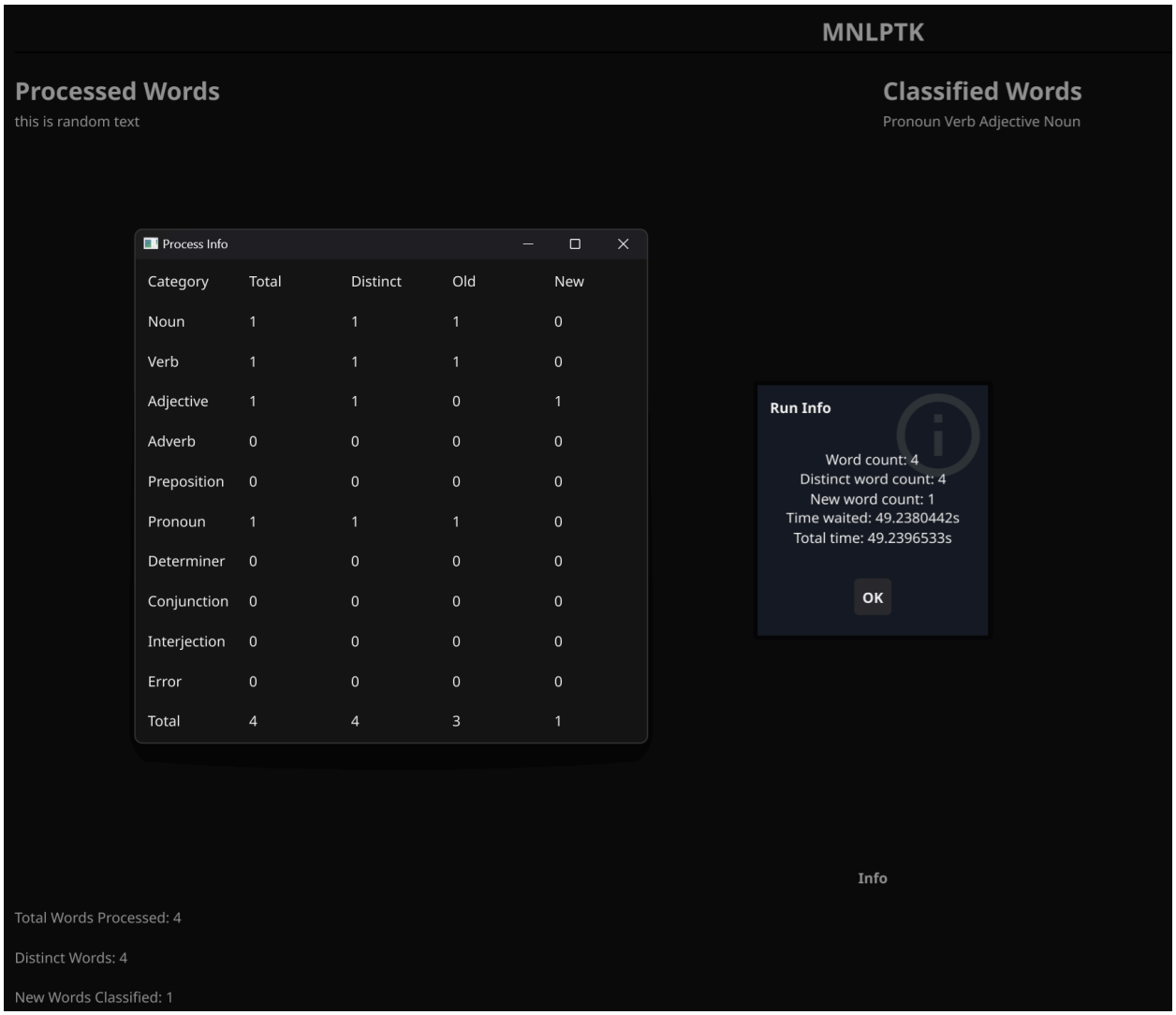




Al usuario se pide clasificar una palabra:


Classify Word
Classify 'random'
Noun
Verb
Adjective
Adverb
Preposition
Pronoun
Determiner
Conjunction
Interjection
Error

Se puede hacer la selección con el mouse o con el teclado. Después se procesa el texto restante y se da información del proceso:



El resultado se guarda en un archivo en formato de tabla .csv:

output.csv

```
TOKEN,LEXEMAS,POSICIONES
Noun;text;TXT0-4
Verb;is;TXT0-2
Adjective;random;TXT0-3
Adverb;;
Preposition;;
Pronoun;this;TXT0-1
Determiner;;
Conjunction;;
Interjection;;
Error;;



Full Tokenized Text:
Pronoun Verb Adjective Noun
```

Código fuente

Se presenta el código fuente de las lógica principal. Los archivos autogenerados y la carpeta vendor no se muestran acá, pero pueden ser visualizada en el repositorio github ya mencionado.

cmd/cli/main.go

```
package main

import (
    "fmt"
    "os"

    "github.com/QuisVenator/compi-tp/tokenizer"
)

func usage() string {
    return `
    Usage: tokenizer -o output.csv -d dictionary.json input1.txt
input2.txt ...
    The program reads an input file to analyze. Optionally the output
filename can be specified (default: output.csv) and the dictionary file
(default: dictionary.json).
    The dictionary must be in either JSON or YAML format.
    `
}

func main() {
    // Parse the command line arguments with the flags package
    if len(os.Args) < 2 {
        fmt.Println(usage())
        return
    }
    outputFilename := "output.csv"
    dictionaryFilename := "dictionary.json"
    var inputFiles []string
    for i := 1; i < len(os.Args); i++ {
        if os.Args[i] == "-o" {
            if i+1 >= len(os.Args) {
```

```
                fmt.Println("Missing output filename")
                return
            }
            outputFilename = os.Args[i+1]
            i++
        } else if os.Args[i] == "-d" {
            if i+1 >= len(os.Args) {
                fmt.Println("Missing dictionary filename")
                return
            }
            dictionaryFilename = os.Args[i+1]
            i++
        } else {
            inputFiles = append(inputFiles, os.Args[i])
        }
    }

    // Create the tokenizer
    classchan := make(chan tokenizer.Wordcategory)
    infochan := make(chan tokenizer.Runinfo)
    p, err := tokenizer.NewTokenizer(dictionaryFilename, inputFiles,
outputFilename, classchan, infochan)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer p.Close()

    // Start the tokenizer
    go p.Parse()

    // Start the display
    displayResults(p.Outchan, p.Newword, classchan, infochan)
}

// This function reads words and their classification from the output
channel and displays them on the console.
// Additionally, it reads words from the newwords channel and asks the use
to classify them. The classification is then sent on the classes channel.
```

```
func displayResults(output <-chan tokenizer.ClassifiedWord, newwords
<-chan string, classes chan<- tokenizer.Wordcategory, infochan chan
tokenizer.Runinfo) {
    for {
        select {
        case word := <-output:
            if word.Class == tokenizer.EOF {
                info := <-infochan
                fmt.Printf("Word count: %d\n", info.WordCount)
                fmt.Printf("Distinct word count: %d\n",
info.DistinctWordCount)
                for cat, count := range info.WordPerCategory {
                    fmt.Printf("Word count for category %s: %d\n", cat,
count)
                }
                for cat, count := range info.DistinctWordPerCategory {
                    fmt.Printf("Distinct word count for category %s:
%d\n", cat, count)
                }
                fmt.Printf("New word count: %d\n", info.NewWordCount)
                for cat, count := range info.NewWordPerCategory {
                    fmt.Printf("New word count for category %s: %d\n",
cat, count)
                }
                fmt.Printf("File count: %d\n", info.FileCount)
                fmt.Printf("Time spent: %s\n", info.TimeSpent)
                fmt.Printf("Time waited: %s\n", info.TimeWaited)

                for _, cat := range tokenizer.AvailableCategories {
                    fmt.Printf("Category %s:\n", cat)
                    fmt.Printf("Total: %d\n", info.WordPerCategory[cat])
                    fmt.Printf("Distinct: %d\n",
info.DistinctWordPerCategory[cat])
                    fmt.Printf("Old: %d\n",
info.DistinctWordPerCategory[cat]-info.NewWordPerCategory[cat])
                    fmt.Printf("New: %d\n", info.NewWordPerCategory[cat])
                }
                fmt.Printf("Total:\n")
                fmt.Printf("Total: %d\n", info.WordCount)
```

```
                fmt.Printf("Distinct: %d\n", info.DistinctWordCount)
                fmt.Printf("Old: %d\n",
info.DistinctWordCount-info.NewWordCount)
                fmt.Printf("New: %d\n", info.NewWordCount)

                close(infochan)
                return
            }
            fmt.Printf("%s: %s\n", word.Word, word.Class)
        case newword := <-newwords:
            fmt.Printf("Please classify the word '%s':\n", newword)
            for class := range tokenizer.AvailableCategories {
                fmt.Printf("%d: %s\n", class+1,
tokenizer.AvailableCategories[class])
            }
            var class int
            fmt.Scan(&class)
            for class < 1 || class > len(tokenizer.AvailableCategories) {
                fmt.Println("Invalid class. Please enter a valid class:")
                fmt.Scan(&class)
            }
            classes <- tokenizer.AvailableCategories[class-1]
        }
    }
}
```

cmd/gui/main.go

```
package main

import (
    "fmt"
    "os"

    "fyne.io/fyne/v2"
    "fyne.io/fyne/v2/app"
    "fyne.io/fyne/v2/container"
```

```
    "fyne.io/fyne/v2/dialog"
    "fyne.io/fyne/v2/layout"
    "fyne.io/fyne/v2/storage"
    "fyne.io/fyne/v2/theme"
    "fyne.io/fyne/v2/widget"
    "github.com/QuisVenator/compi-tp/tokenizer"
)

func main() {
    processedWords := make(map[string]struct{})
    processedWordsText := ""
    classifiedText := ""
    newWordsCount := 0
    processedWordsCount := 0

    a := app.New()
    w := a.NewWindow("MNLPTK")

    // Widgets
    header := widget.NewRichTextFromMarkdown("# MNLPTK\n___")
    header.Wrapping = fyne.TextWrapWord
    header.Segments[0].(*widget.TextSegment).Style.Alignment =
fyne.TextAlignCenter

    processedWordsTextField := widget.NewRichTextFromMarkdown("# Processed
Words\n")
    processedWordsTextField.Wrapping = fyne.TextWrapWord
    processedWordsTextField.Segments =
append(processedWordsTextField.Segments, &widget.TextSegment{Text:
processedWordsText})
    classifiedTextField := widget.NewRichTextFromMarkdown("# Classified
Words\n")
    classifiedTextField.Wrapping = fyne.TextWrapWord
    classifiedTextField.Segments = append(classifiedTextField.Segments,
&widget.TextSegment{Text: classifiedText})

    infoLabel := widget.NewLabelWithStyle("Info", fyne.TextAlignCenter,
fyne.TextStyle{Bold: true})
```

```
    processedWordsCountLabel := widget.NewLabel(fmt.Sprintf("Total Words
Processed: %d", processedWordsCount))
    distinctWordsLabel := widget.NewLabel(fmt.Sprintf("Distinct Words:
%d", len(processedWords)))
    newWordsCountLabel := widget.NewLabel(fmt.Sprintf("New Words
Classified: %d", newWordsCount))

    // Layout
    content := container.NewVBox(
        header,
        container.New(
            layout.NewGridLayout(2),
            processedWordsTextField,
            classifiedTextField,
        ),
        layout.NewSpacer(),
        infoLabel,
        processedWordsCountLabel,
        distinctWordsLabel,
        newWordsCountLabel,
    )

    w.Resize(fyne.NewSize(1920, 1080))

    // Run update loop
    updateLoop := func() {
        userChosenClassCh := make(chan tokenizer.Wordcategory)
        infochan := make(chan tokenizer.Runinfo)
        p := startup(w, userChosenClassCh, infochan)
        classifiedWordsCh := p.Outchan
        newWordsCh := p.Newword
        w.SetContent(content)
        go p.Parse()

        for {
            select {
            case word := <-classifiedWordsCh:
                if word.Class == tokenizer.EOF {
                    info := <-infochan
```

```
                    dialog.ShowInformation("Run Info", fmt.Sprintf("Word
count: %d\nDistinct word count: %d\nNew word count: %d\nTime waited:
%s\nTotal time: %s\n", info.WordCount, info.DistinctWordCount,
info.NewWordCount, info.TimeWaited.String(), info.TimeSpent.String()), w)

                    // Window for dictionary information
                    columns := []fyne.CanvasObject{
                        widget.NewLabel("Category"),
                        widget.NewLabel("Total"),
                        widget.NewLabel("Distinct"),
                        widget.NewLabel("Old"),
                        widget.NewLabel("New"),
                    }

                    for _, cat := range tokenizer.AvailableCategories {
                        columns = append(columns,
                            widget.NewLabel(string(cat)),
                            widget.NewLabel(fmt.Sprintf("%d",
info.WordPerCategory[cat])),
                            widget.NewLabel(fmt.Sprintf("%d",
info.DistinctWordPerCategory[cat])),
                            widget.NewLabel(fmt.Sprintf("%d",
info.DistinctWordPerCategory[cat]-info.NewWordPerCategory[cat])),
                            widget.NewLabel(fmt.Sprintf("%d",
info.NewWordPerCategory[cat])),
                        )
                    }
                    columns = append(columns,
                        widget.NewLabel("Total"),
                        widget.NewLabel(fmt.Sprintf("%d",
info.WordCount)),
                        widget.NewLabel(fmt.Sprintf("%d",
info.DistinctWordCount)),
                        widget.NewLabel(fmt.Sprintf("%d",
info.DistinctWordCount-info.NewWordCount)),
                        widget.NewLabel(fmt.Sprintf("%d",
info.NewWordCount)),
                    )
```

```
                    dictContent := container.NewGridWithColumns(5,
                        columns...,
                    )
                    dictW := a.NewWindow("Process Info")
                    dictW.SetContent(dictContent)
                    dictW.Show()

                    close(infochan)
                    p.Close()
                    return
                }

                processedWords[word.Word] = struct{}{}
                processedWordsText += word.Word + " "
                classifiedText += string(word.Class) + " "
                processedWordsCount++

processedWordsTextField.Segments[1].(*widget.TextSegment).Text =
processedWordsText
                processedWordsTextField.Refresh()
                classifiedTextField.Segments[1].(*widget.TextSegment).Text
= classifiedText
                classifiedTextField.Refresh()
                processedWordsCountLabel.SetText(fmt.Sprintf("Total Words
Processed: %d", processedWordsCount))
                distinctWordsLabel.SetText(fmt.Sprintf("Distinct Words:
%d", len(processedWords)))
            case word := <-newWordsCh:
                var dia *dialog.CustomDialog

                // Create classification dialog
                prompt := widget.NewLabel(fmt.Sprintf("Classify '%s'",
word))

                catBtns := make([]fyne.CanvasObject,
len(tokenizer.AvailableCategories))
                for i, cat := range tokenizer.AvailableCategories {
                    catBtns[i] = widget.NewButton(string(cat), func() {
                        userChosenClassCh <- cat
```

```
                        newWordsCount++
                        newWordsCountLabel.SetText(fmt.Sprintf("New Words
Classified: %d", newWordsCount))
                        dia.Hide()
                    })
                }

                diaContent := container.NewVBox(
                    append([]fyne.CanvasObject{prompt}, catBtns...)...,
                )

                dia = dialog.NewCustomWithoutButtons("Classify Word",
diaContent, w)
                dia.Show()
            }
        }
    }

    go updateLoop()
    w.ShowAndRun()
}

func startup(w fyne.Window, categoryCh <-chan tokenizer.Wordcategory,
infochan chan tokenizer.Runinfo) *tokenizer.Tokenizer {
    var exPath string
    ex, err := os.Getwd()
    if err != nil {
        exPath = "./"
    } else {
        exPath = ex + "/"
    }

    inputFilePaths := []string{}
    dictPath := exPath + "dictionary.json"
    outputPath := exPath + "output.csv"
    startCh := make(chan struct{})

    // Widgets
    dictionarySelectedLabel := container.NewHBox(
```

```
        widget.NewIcon(theme.DocumentIcon()),
        widget.NewLabel(dictPath),
    )
    outputLabel := widget.NewLabel(outputPath)
    inputFileList := widget.NewList(
        func() int {
            return len(inputFilePaths)
        },
        func() fyne.CanvasObject {
            return container.NewHBox(
                widget.NewIcon(theme.DocumentIcon()),
                widget.NewLabel("input.txt"),
            )
        },
        func(i widget.ListItemID, item fyne.CanvasObject) {

item.(*fyne.Container).Objects[1].(*widget.Label).SetText(inputFilePaths[i
])
        },
    )
    inputFileList.Resize(fyne.NewSize(inputFileList.MinSize().Width,
inputFileList.MinSize().Height*3))

    // Layout
    content := container.NewBorder(
        container.NewVBox(
            widget.NewLabel("Welcome to MNLPTK!"),
            container.NewVBox(
                widget.NewLabel("Please select dictionary to use:"),
                dictionarySelectedLabel,
                widget.NewButton("Open Dictionary", func() {
                    d := dialog.NewFileOpen(func(reader
fyne.URIReadCloser, err error) {
                        if err != nil {
                            dialog.ShowError(err, w)
                            return
                        }
                        if reader == nil {
                            dictPath = "dictionary.json"
```

```
                        } else {
                            dictPath = reader.URI().Path()
                            defer reader.Close()
                        }

dictionarySelectedLabel.Objects[1].(*widget.Label).SetText(dictPath)
                        dictionarySelectedLabel.Refresh()
                    }, w)

d.SetFilter(storage.NewExtensionFileFilter([]string{".json", ".yaml"}))
                    d.Show()
                }),
            ),
        ),
        container.NewBorder(
            container.NewHBox(
                widget.NewLabel("Save output to:"),
                outputLabel,
                widget.NewButton("Change", func() {
                    dialog.ShowFileSave(func(writer fyne.URIWriteCloser,
err error) {
                        if err != nil {
                            dialog.ShowError(err, w)
                            return
                        }
                        if writer != nil {
                            outputPath = writer.URI().Path()
                            defer writer.Close()
                        }
                        outputLabel.SetText(outputPath)
                    }, w)
                }),
            ),
            widget.NewButton("Start", func() {
                startCh <- struct{}{}
            }),
            nil,
            nil,
            nil,
```

```
        ),
        nil,
        nil,
        container.NewBorder(
            widget.NewLabel("Please select input files:"),
            widget.NewButton("Add Input File", func() {
                dialog.ShowFileOpen(func(reader fyne.URIReadCloser, err
error) {
                    if err != nil {
                        dialog.ShowError(err, w)
                        return
                    }
                    if reader == nil {
                        return
                    }
                    defer reader.Close()

                    inputFilePaths = append(inputFilePaths,
reader.URI().Path())
                    inputFileList.Refresh()
                }, w)
            }),
            nil,
            nil,
            inputFileList,
        ),
    )

    w.SetContent(content)

    // Run loop for startup
    for {
        <-startCh
        p, err := tokenizer.NewTokenizer(dictPath, inputFilePaths,
outputPath, categoryCh, infochan)
        if err != nil {
            dialog.ShowError(err, w)
        } else {
            return p
```

```
        }
    }
}
```

tokenizer/dictionary.go

```
package tokenizer

import (
   "encoding/json"
   "errors"
   "os"
   "strings"

   "gopkg.in/yaml.v2"
)

// We use a struct to represent a dictionary entry instead of a string of
only category
// This is to allow for future expansion of the dictionary entry
type DictEntry struct {
   Category string `json:"category" yaml:"category"`
}
type Dictionary map[string]DictEntry
type dictFileFormat int
type Wordcategory string
type ClassifiedWord struct {
   Word  string
   Class Wordcategory
}

// Enum of permissible dictionary file formats
const (
   JSON dictFileFormat = iota
   YAML
   UNKNOWN
)
```

```
const (
    NOUN         = "Noun"
    VERB         = "Verb"
    ADJECTIVE    = "Adjective"
    ADVERB       = "Adverb"
    PREPOSITION  = "Preposition"
    PRONOUN      = "Pronoun"
    DETERMINER   = "Determiner"
    CONJUNCTION  = "Conjunction"
    INTERJECTION = "Interjection"
    ERROR_LX     = "Error"
    EOF          = "EOF"
)

var AvailableCategories = []Wordcategory{NOUN, VERB, ADJECTIVE, ADVERB,
PREPOSITION, PRONOUN, DETERMINER, CONJUNCTION, INTERJECTION, ERROR_LX}

func NewDictionary() *Dictionary {
    return &Dictionary{}
}

func NewDictionaryFromFile(filename string) (*Dictionary, error) {
    var fileFormat = UNKNOWN
    if strings.HasSuffix(filename, ".json") {
        fileFormat = JSON
    } else if strings.HasSuffix(filename, ".yaml") {
        fileFormat = YAML
    } else {
        // Abort early if the file format is not recognized
        return nil, errors.New("invalid file format")
    }

    // Check if the file exists
    _, err := os.Stat(filename)
    if os.IsNotExist(err) {
        return NewDictionary(), nil
    }

    file, err := os.Open(filename)
```

```
    if err != nil {
        return nil, err
    }
    defer file.Close()

    switch fileFormat {
    case JSON:
        return NewDictionaryFromJSONFile(file)
    case YAML:
        return NewDictionaryFromYAMLFile(file)
    }

    // This should never be reached
    return nil, errors.New("invalid file format")
}

func NewDictionaryFromJSONFile(f *os.File) (*Dictionary, error) {
    decoder := json.NewDecoder(f)
    dictionary := &Dictionary{}
    err := decoder.Decode(dictionary)
    if err != nil {
        return nil, err
    }

    return dictionary, nil
}

func NewDictionaryFromYAMLFile(f *os.File) (*Dictionary, error) {
    decoder := yaml.NewDecoder(f)
    dictionary := &Dictionary{}
    err := decoder.Decode(dictionary)
    if err != nil {
        return nil, err
    }

    return dictionary, nil
}

func (d *Dictionary) SaveToJSONFile(f *os.File) error {
```

```
    encoder := json.NewEncoder(f)
    encoder.SetIndent("", "  ")
    return encoder.Encode(d)
}

func (d *Dictionary) SaveToYAMLFile(f *os.File) error {
    encoder := yaml.NewEncoder(f)
    return encoder.Encode(d)
}

func (d *Dictionary) SaveToFile(filename string) error {
    var fileFormat dictFileFormat
    if strings.HasSuffix(filename, ".json") {
        fileFormat = JSON
    } else if strings.HasSuffix(filename, ".yaml") {
        fileFormat = YAML
    } else {
        // Abort early if the file format is not recognized
        return errors.New("invalid file format")
    }

    file, err := os.Create(filename)
    if err != nil {
        return err
    }
    defer file.Close()

    switch fileFormat {
    case JSON:
        return d.SaveToJSONFile(file)
    case YAML:
        return d.SaveToYAMLFile(file)
    }

    // This should never be reached
    return errors.New("invalid file format")
}
```

```
func (d *Dictionary) AddEntry(word string, category Wordcategory, smart
bool) {
   (*d)[word] = DictEntry{Category: string(category)}

   // Some categories of the dictionary do more than just adding the word
   // For example nouns can be singular or plural, so we need to add both
   if smart {
       switch category {
       case NOUN:
           d.addNoun(word)
       case VERB:
           d.addVerb(word)
       }
   }
}

func (d *Dictionary) GetEntry(word string) (Wordcategory, bool) {
   entry, ok := (*d)[word]
   if !ok {
       return "", false
   }

   return Wordcategory(entry.Category), true
}

// Category functions

// This function will try to handle most cases of pluralizing and
singularizing nouns
// It will add the word to the dictionary if it is not already present
// A best effort is made to handle most cases, but there are some edge
cases that are not handled
// For example, "child" -> "children", "goose" -> "geese", "cactus" ->
"cacti", "deer" -> "deer" are not handled
func (d *Dictionary) addNoun(noun string) {
   var toAdd string
   if strings.HasSuffix(noun, "ies") {
       toAdd = strings.TrimSuffix(noun, "ies") + "y"
   } else if strings.HasSuffix(noun, "s") {
```

```
        if strings.HasSuffix(noun, "es") {
            // Handles cases like "buses", "foxes"
            toAdd = strings.TrimSuffix(noun, "es")
        } else {
            // Handles cases like "bananas"
            toAdd = strings.TrimSuffix(noun, "s")
        }
    } else if strings.HasSuffix(noun, "y") {
        // Handles cases like "pony"
        toAdd = strings.TrimSuffix(noun, "y") + "ies"
    } else if strings.HasSuffix(noun, "f") {
        // Handles cases like "leaf"
        toAdd = strings.TrimSuffix(noun, "f") + "ves"
    } else if strings.HasSuffix(noun, "fe") {
        // Handles cases like "wife"
        toAdd = strings.TrimSuffix(noun, "fe") + "ves"
    } else if strings.HasSuffix(noun, "o") {
        // Handles cases like "potato"
        toAdd = noun + "es"
    } else {
        // Regular case
        toAdd = noun + "s"
    }

    // We want to make sure that the word is not already in the dictionary
    if _, ok := (*d)[toAdd]; !ok {
        (*d)[toAdd] = DictEntry{Category: "noun"}
    }
}

// This function will try to handle most cases of conjugating verbs
// Similar to the addNoun function, it will add the word to the dictionary
if it is not already present
// Also similar to addNoun, a best effort is made, only that "best" is
even worse in this case
func (d *Dictionary) addVerb(verb string) {
    var toAdd string
```

```
    if strings.HasSuffix(verb, "y") && !strings.HasSuffix(verb, "ay") &&
!strings.HasSuffix(verb, "ey") && !strings.HasSuffix(verb, "iy") &&
!strings.HasSuffix(verb, "oy") && !strings.HasSuffix(verb, "uy") {
        // Handles cases like "fly" -> "flies", but not "play" -> "plays"
        toAdd = strings.TrimSuffix(verb, "y") + "ies"
    } else if strings.HasSuffix(verb, "o") || strings.HasSuffix(verb, "sh")
|| strings.HasSuffix(verb, "ch") || strings.HasSuffix(verb, "s") ||
strings.HasSuffix(verb, "x") {
        // Handles cases like "go" -> "goes", "wash" -> "washes"
        toAdd = verb + "es"
    } else {
        // Regular case
        toAdd = verb + "s"
    }

    // We want to make sure that the word is not already in the dictionary
    if _, ok := (*d)[toAdd]; !ok {
        (*d)[toAdd] = DictEntry{Category: "verb"}
    }
}
```

tokenizer/tokenizer.go

```
package tokenizer

import (
    "bufio"
    "fmt"
    "os"
    "regexp"
    "strings"
    "time"
)

var whitespaceOrPunctuation = regexp.MustCompile(`[\s\p{P}]+`)

type Tokenizer struct {
    dict     *Dictionary
```

```
    dictFile string
    input    []*os.File
    output   *os.File
    inpath   []string
    outpath  string
    Outchan  chan ClassifiedWord
    Newword  chan string
    Classes  <-chan Wordcategory
    Infochan chan<- Runinfo
}

type Runinfo struct {
    WordCount               int
    DistinctWordCount       int
    WordPerCategory         map[Wordcategory]int
    DistinctWordPerCategory map[Wordcategory]int
    NewWordCount            int
    NewWordPerCategory      map[Wordcategory]int
    FileCount               int
    TimeSpent               time.Duration
    TimeWaited              time.Duration
}

func NewTokenizer(dict string, inpath []string, outpath string, classchan
<-chan Wordcategory, infochan chan<- Runinfo) (*Tokenizer, error) {
    dictionary, err := NewDictionaryFromFile(dict)
    if err != nil {
        return nil, err
    }
    var p Tokenizer
    p.dictFile = dict
    p.dict = dictionary
    p.inpath = inpath
    p.outpath = outpath
    p.Infochan = infochan

    for _, path := range inpath {
        input, err := os.Open(path)
        if err != nil {
```

```
            return nil, err
        }
        p.input = append(p.input, input)
    }
    p.output, err = os.Create(outpath)
    if err != nil {
        return nil, err
    }

    p.Outchan = make(chan ClassifiedWord)
    p.Newword = make(chan string)
    p.Classes = classchan

    return &p, nil
}

func (p *Tokenizer) Parse() error {
    info := Runinfo{
        WordPerCategory:         make(map[Wordcategory]int),
        DistinctWordPerCategory: make(map[Wordcategory]int),
        NewWordPerCategory:      make(map[Wordcategory]int),
    }
    auxmap := make(map[string]int)
    t_start := time.Now()
    positions := make(map[Wordcategory]string)
    words := make(map[Wordcategory]string)

    fulltext := ""

    for i, input := range p.input {
        wordnum := 0
        Scanner := bufio.NewScanner(input)
        Scanner.Split(SplitWords)

        for Scanner.Scan() {
            word := Scanner.Text()
            if word == "" {
                continue
            }
```

```
            word = strings.ToLower(word)
            wordnum++
            class, ok := p.dict.GetEntry(word)
            if !ok {
                p.Newword <- word
                t_wait := time.Now()
                class = <-p.Classes
                info.TimeWaited += time.Since(t_wait)
                p.dict.AddEntry(word, class, false)

                // Info
                info.NewWordCount++
                info.NewWordPerCategory[class]++
            }
            fulltext += string(class) + " "
            // Info
            info.WordCount++
            if auxmap[word] == 0 {
                info.DistinctWordCount++
                info.DistinctWordPerCategory[class]++
            }
            info.WordPerCategory[class]++
            auxmap[word]++

            p.Outchan <- ClassifiedWord{word, class}
            positions[class] += fmt.Sprintf("TXT%d-%d,", i, wordnum)
            words[class] += word + ","
        }
    }

    p.output.WriteString("TOKEN,LEXEMAS,POSICIONES\n")
    for _, cat := range AvailableCategories {
        p.output.WriteString(string(cat))
        p.output.WriteString(";")
        p.output.WriteString(strings.TrimSuffix(words[cat], ","))
        p.output.WriteString(";")
        p.output.WriteString(strings.TrimSuffix(positions[cat], ","))
        p.output.WriteString("\n")
    }
```

```
    fulltext = strings.TrimSuffix(fulltext, " ")
    p.output.WriteString("\n\n\nFull Tokenized Text:\n")
    p.output.WriteString(fulltext)

    p.Outchan <- ClassifiedWord{"", EOF}

    // Info
    info.TimeSpent = time.Since(t_start)
    info.FileCount = len(p.inpath)

    p.Infochan <- info

    return nil
}

func (p *Tokenizer) Close() {
    for _, input := range p.input {
        input.Close()
    }
    p.output.Close()
    close(p.Outchan)
    close(p.Newword)
    err := p.dict.SaveToFile(p.dictFile)
    if err != nil {
        fmt.Println(err)
    }
}

func SplitWords(data []byte, atEOF bool) (advance int, token []byte, err
error) {
    if atEOF && len(data) == 0 {
        return 0, nil, nil
    }

    if match := whitespaceOrPunctuation.FindIndex(data); match != nil {
        return match[1], data[:match[0]], nil
    }

    if atEOF {
```

```
        return len(data), data, nil
    }

    return 0, nil, nil
}
```

Observaciones

Sería muy fácil extender el programa para usar adicionalmente un patrón por comprensión para números. Se optó por no hacer esto, para cumplir con el enunciado que pedía un patrón por extensión, además es subjetivo como se consideran los números o si forman su propia categoría.

Conclusión

El desarrollo de un tokenizador eficiente y robusto para el procesamiento de lenguajes naturales en inglés ha demostrado ser una tarea compleja pero alcanzable mediante el uso de tecnologías adecuadas y técnicas de programación modernas. La elección de Golang como lenguaje de programación principal ha permitido un manejo eficiente de cadenas de caracteres, seguridad en la gestión de memoria y un soporte sobresaliente para la concurrencia, lo cual ha sido esencial para el éxito del proyecto.

Las mejoras implementadas, tales como la conjugación automática de verbos y la generación de formas plurales, han añadido un valor significativo al proyecto, permitiendo un análisis léxico más preciso y exhaustivo. La capacidad de guardar los resultados en formatos accesibles como CSV, y de almacenar el diccionario en JSON o YAML, facilita su integración con otras herramientas y plataformas de análisis de datos.

La creación de una interfaz gráfica moderna utilizando la librería Fyne ha sido otro de los logros destacables, proporcionando una experiencia de usuario intuitiva y adaptativa, que amplía la accesibilidad del tokenizador tanto para usuarios técnicos como no técnicos. La separación clara entre la lógica de procesamiento y la interfaz de usuario, mediante el uso de channels, asegura una arquitectura de software limpia y modular, facilitando futuras mejoras y mantenimientos.

El soporte para procesar múltiples archivos de entrada y la capacidad de ejecutar el programa tanto en GUI como en consola, amplían las posibilidades de uso del tokenizador, haciéndolo una herramienta versátil y poderosa para el análisis de grandes volúmenes de texto.

En resumen, el proyecto no solo cumple con los objetivos iniciales de identificar y organizar palabras en un texto en inglés, sino que también incorpora varias mejoras y características adicionales que aumentan su utilidad. Si bien existen casos en los cuales se podrían agregar palabras equivocadas al diccionario, el trabajo cumplió con el objetivo principal de aprendizaje y las mejoras aportaron a esta meta.