```
File Tree (Placeholder):
  git.go
  interactive.go
  language.go
  main.go
  output.go
  pdf.go
  processor.go
  tokenizer.go
  types.go
  web.go
```

File: git.go

Tokens: 429

```go
package main

import (
    "fmt"
    "os"
    "strings"

    "github.com/go-git/go-git/v5"
    "github.com/go-git/go-git/v5/plumbing"
)

// isGitURL checks if the input string looks like a Git repository URL.
// This is a basic check and might need refinement.
func isGitURL(input string) bool {
    // Check for common Git URL schemes and the .git suffix
    return strings.HasSuffix(input, ".git") ||
        strings.HasPrefix(input, "git@") ||
        strings.HasPrefix(input, "https://") ||
        strings.HasPrefix(input, "http://") ||
        strings.HasPrefix(input, "ssh://")
    // Add other protocols like file:// if needed
}

// cloneGitRepo clones a Git repository URL into a temporary directory.
// It returns the path to the temporary directory or an error.
func cloneGitRepo(url string) (string, error) {
    // Create a temporary directory
    tempDir, err := os.MkdirTemp("", "iris-git-")
    if err != nil {
        return "", fmt.Errorf("failed to create temporary directory: %w", err)
    }

    fmt.Printf("Cloning Git repository '%s' into '%s'...\n", url, tempDir)

    // Clone the repository
    _, err = git.PlainClone(tempDir, false, &git.CloneOptions{
        URL:      url,
        Progress: os.Stdout, // Show progress during clone
        // Depth: 1, // Optional: shallow clone for faster download if history isn't needed
        ReferenceName: plumbing.HEAD, // Checkout default branch
```

```
        SingleBranch:  true,          // Only fetch the default branch
    })

    if err != nil {
        // Attempt cleanup even if clone failed
        _ = os.RemoveAll(tempDir)
        return "", fmt.Errorf("failed to clone repository '%s': %w", url, err)
    }

    fmt.Printf("Finished cloning '%s'.\n", url)
    return tempDir, nil
}
```

File: interactive.go

Tokens: 713

```go
package main

import (
    "fmt"
    "io/fs"
    "os"
    "path/filepath"

    fuzzyfinder "github.com/ktr0731/go-fuzzyfinder"
)

// runInteractiveFinder finds files/dirs and uses a fuzzy finder for selection.
func runInteractiveFinder() ([]string, error) {
    // 1. Find candidates: Walk current dir, apply basic filters (hidden, maybe gitignore?)
    // For simplicity, let's start with a basic walk respecting --hidden.
    // We won't apply include/exclude/size here, let the user pick first.
    candidates := []string{}
    root := "." // Start from current directory

    // We need a simplified walk just to get paths for the finder
    err := filepath.WalkDir(root, func(path string, d fs.DirEntry, err error) error {
        if err != nil {
            // Silently ignore errors during candidate finding?
            // Or print warnings?
            // fmt.Fprintf(os.Stderr, "Warning (interactive scan): error accessing %s: %v\n",
path, err)
            return nil // Continue walking
        }

        // Skip root
        if path == root {
            return nil
        }

        // Basic Hidden File Filter (respecting flag)
        if !showHidden && isHidden(d.Name()) {
            if d.IsDir() {
                return fs.SkipDir
            }
            return nil
        }

        // TODO: Optionally add .gitignore filtering here for a cleaner list?
        // Requires loading gitignore from "."

        candidates = append(candidates, path)
        return nil
    })

    if err != nil {
        return nil, fmt.Errorf("error scanning for files/directories: %w", err)
    }
```

```
    if len(candidates) == 0 {
        return nil, fmt.Errorf("no files or directories found to select from")
    }

    // 2. Run Fuzzy Finder
    idx, err := fuzzyfinder.FindMulti(
        candidates,
        func(i int) string {
            return candidates[i] // Display the path itself
        },
        fuzzyfinder.WithPreviewWindow(func(i, w, h int) string {
            if i == -1 { // No selection yet
                return "Select files or directories to process. Press Tab to multi-select, Enter
to confirm."
            }
            // Basic preview: show file type and size
            path := candidates[i]
            info, statErr := os.Stat(path)
            if statErr != nil {
                return fmt.Sprintf("Path: %s\nError getting info: %v", path, statErr)
            }
            fileType := "File"
            if info.IsDir() {
                fileType = "Directory"
            }
            return fmt.Sprintf("Path: %s\nType: %s\nSize: %d bytes", path, fileType, info.Size())
        }),
    )

    if err != nil {
        if err == fuzzyfinder.ErrAbort { // User pressed Esc or Ctrl+C
            fmt.Println("Interactive selection aborted.")
            return nil, nil // Return nil slice and nil error to indicate graceful exit
        }
        return nil, fmt.Errorf("fuzzy finder error: %w", err)
    }

    selectedPaths := make([]string, len(idx))
    for i, index := range idx {
        selectedPaths[i] = candidates[index]
    }

    return selectedPaths, nil
}
```

File: language.go

Tokens: 892

```go
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "strings"

    "gopkg.in/yaml.v3"
)

// LanguageInfo holds details about a specific programming/markup language.
// We only include fields relevant for file detection for now.
type LanguageInfo struct {
    Type         string    yaml:"type"  // e.g., programming, data, markup
    Extensions   []string  yaml:"extensions"
    Filenames    []string  yaml:"filenames"
    Interpreters []string  yaml:"interpreters"
    // Add other fields like color, language_id if needed later
}

// LanguageMap maps language names (e.g., "Go") to their details.
type LanguageMap map[string]LanguageInfo

// LoadedLanguageData holds the parsed language map and provides helper methods.
type LoadedLanguageData struct {
    Langs        LanguageMap
    extensionMap map[string]string // Map extension (e.g., ".go") to language name ("Go")
    filenameMap  map[string]string // Map filename (e.g., "Makefile") to language name ("Makefile"
)
}

// loadLanguageData attempts to load and parse languages.yml.
func loadLanguageData() (*LoadedLanguageData, error) {
    // Look for languages.yml in standard config paths
    configPaths := []string{}
    if home, err := os.UserHomeDir(); err == nil {
        configPaths = append(configPaths, filepath.Join(home, ".config", "iris"))
    }
    configPaths = append(configPaths, ".") // Current directory

    var langFilePath string
    for _, p := range configPaths {
        testPath := filepath.Join(p, "languages.yml")
        if _, err := os.Stat(testPath); err == nil {
            langFilePath = testPath
            break
        }
    }

    if langFilePath == "" {
        return nil, fmt.Errorf("languages.yml not found in standard config locations")
```

```
    }

    fmt.Printf("Loading language definitions from: %s\n", langFilePath)
    yamlFile, err := os.ReadFile(langFilePath)
    if err != nil {
        return nil, fmt.Errorf("error reading language file %s: %w", langFilePath, err)
    }

    var langs LanguageMap
    err = yaml.Unmarshal(yamlFile, &langs)
    if err != nil {
        return nil, fmt.Errorf("error parsing language file %s: %w", langFilePath, err)
    }

    // Build lookup maps for faster matching
    data := &LoadedLanguageData{
        Langs:        langs,
        extensionMap: make(map[string]string),
        filenameMap:  make(map[string]string),
    }

    for langName, info := range langs {
        for _, ext := range info.Extensions {
            // Ensure extension includes the dot and is lowercase for consistent matching
            lowerExt := strings.ToLower(ext)
            if data.extensionMap[lowerExt] == "" { // Avoid overwriting if multiple languages
claim same ext
                data.extensionMap[lowerExt] = langName
            }
        }
        for _, fname := range info.Filenames {
            // Match filenames case-sensitively  Linguist often does.
            if data.filenameMap[fname] == "" {
                data.filenameMap[fname] = langName
            }
        }
    }

    fmt.Printf("Loaded %d languages with %d extensions and %d specific filenames.\n", len(data.
Langs), len(data.extensionMap), len(data.filenameMap))
    return data, nil
}

// GetLanguageForFile determines the language for a given path based on loaded data.
func (ld *LoadedLanguageData) GetLanguageForFile(filePath string) (string, bool) {
    if ld == nil {
        return "", false // No language data loaded
    }

    baseName := filepath.Base(filePath)
    ext := strings.ToLower(filepath.Ext(baseName))

    // 1. Check exact filename match first (higher precedence)
    if lang, ok := ld.filenameMap[baseName]; ok {
        return lang, true
```

```
    }

    // 2. Check extension match
    if ext != "" {
        if lang, ok := ld.extensionMap[ext]; ok {
            return lang, true
        }
    }

    // Could add interpreter matching here if needed

    return "", false // No match found
}
```

File: main.go

Tokens: 4794

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "runtime"
    "sort"
    "strings"
    "sync"

    "github.com/atotto/clipboard"
    "github.com/spf13/cobra"
    "github.com/spf13/viper"
    // Tokenizer interface defined in tokenizer.go
)

var (
    // Input Sources
    inputPaths []string

    // Filtering
    includePatterns string
    excludePatterns string
    maxSizeBytes    int64
    maxDepth        int
    showHidden      bool
    noIgnore        bool

    // Output
    outputFormat    string
    outputFile      string
    printToStdout   bool // Glimpse uses '-p', but true is the default unless -f or -c is used.
Let's clarify behavior later.
    copyToClipboard bool // Glimpse uses '-c'

    // Processing
    numThreads int

    // Token Counting
    disableTokens  bool
    tokenizerType  string
    tokenizerModel string
    tokenizerFile  string

    // Web Specific
    traverseLinks bool
    linkDepth     int

    // PDF Output
    pdfOutputFile string
```

```
    // Interactive Mode
    interactiveMode bool

    cfgFile string // Variable to hold potential config file path flag (optional)

    langData *LoadedLanguageData // Global or passed around?
)

var rootCmd = &cobra.Command{
    Use:   "iris [PATHS...]",
    Short: "Iris is a tool for quickly analyzing codebases, similar to Glimpse.",
    Long:  Iris allows you to process local directories, files, Git repositories,
and web URLs to generate structure views, display content, and count tokens. `,
    Version: "0.0.1",             // Placeholder version
    Args:    cobra.ArbitraryArgs, // Allow paths to be passed as arguments
    Run: func(cmd *cobra.Command, args []string) {
        // initConfig and language loading are called via cobra.OnInitialize

        // Determine input paths: interactive or command-line args
        var finalInputPaths []string
        var err error
        if interactiveMode {
            finalInputPaths, err = runInteractiveFinder()
            if err != nil {
                fmt.Fprintf(os.Stderr, "Interactive mode error: %v\n", err)
                os.Exit(1)
            }
            if finalInputPaths == nil {
                // User aborted interactive selection
                os.Exit(0)
            }
            fmt.Printf("Processing interactively selected paths: %v\n", finalInputPaths)
        } else {
            // Use command-line arguments
            finalInputPaths = args
            if len(finalInputPaths) == 0 {
                finalInputPaths = []string{"."} // Default to current directory if no paths
provided
            }
        }

        // --- Initialize Tokenizer (if needed) ---
        var tokenizer Tokenizer // Use the interface type
        if !disableTokens {
            tokenizer, err = getTokenizer() // Returns the interface (assigns to existing err)
            if err != nil {
                fmt.Fprintf(os.Stderr, "Error initializing tokenizer: %v\n", err)
                disableTokens = true
                fmt.Fprintln(os.Stderr, "Token counting disabled due to error.")
            } else {
                // Ensure tokenizer resources are cleaned up if applicable
                defer tokenizer.Close()
            }
        }
```

```
        // --- Main Logic ---
        fmt.Println("Iris running...")

        var allFilesMaster []FileInfo // Collect files from all inputs first
        var failedPaths int
        var tempDirsToClean []string // Keep track of temp dirs for cleanup

        // Ensure temporary directories are cleaned up on exit (even if errors occur)
        defer func() {
            for _, dir := range tempDirsToClean {
                fmt.Printf("Cleaning up temporary directory: %s\n", dir)
                _ = os.RemoveAll(dir)
            }
        }()

        for _, input := range finalInputPaths {
            var filesToAppend []FileInfo
            var err error
            currentInput := input

            if isGitURL(currentInput) {
                tempDir, cloneErr := cloneGitRepo(currentInput)
                if cloneErr != nil {
                    fmt.Fprintf(os.Stderr, "Error cloning git repo %s: %v\n", currentInput,
cloneErr)
                    failedPaths++
                    continue
                }
                tempDirsToClean = append(tempDirsToClean, tempDir)
                currentInput = tempDir // Process the cloned directory path
                // Process the cloned directory as a local path, passing langData
                filesToAppend, err = processLocalPath(currentInput, langData)
            } else if isWebURL(currentInput) {
                // Process web URL
                if traverseLinks {
                    fmt.Printf("Starting web traversal from %s (max depth: %d)\n", currentInput,
linkDepth)
                    visited := make(map[string]bool)
                    // Call recursive function with specified depth
                    filesToAppend, err = processWebURLRecursive(currentInput, 0, linkDepth,
visited)
                } else {
                    // Process single web URL without traversal
                    var fileInfo FileInfo
                    fileInfo, err = processWebURL(currentInput) // Uses the non-recursive wrapper
                    if err == nil {
                        filesToAppend = []FileInfo{fileInfo}
                    }
                }
                // Error handled below
            } else {
                // Assume local path, pass langData
                filesToAppend, err = processLocalPath(currentInput, langData)
            }
```

```
        // Handle errors from processing steps
        if err != nil {
            fmt.Fprintf(os.Stderr, "Error processing %s: %v\n", input, err)
            failedPaths++
            continue
        }

        allFilesMaster = append(allFilesMaster, filesToAppend...)
    }

    // --- Parallel Token Counting (if enabled) ---
    var processedFiles []FileInfo
    if !disableTokens && tokenizer != nil { // Check if tokenizer was successfully initialized
        numWorkers := numThreads
        if numWorkers <= 0 {
            numWorkers = runtime.NumCPU()
        }
        fmt.Printf("Using %d worker(s) for token counting.\n", numWorkers)

        jobs := make(chan FileInfo, len(allFilesMaster))
        results := make(chan FileInfo, len(allFilesMaster))
        var wg sync.WaitGroup

        // Start workers
        for w := 0; w < numWorkers; w++ {
            wg.Add(1)
            // Pass the Tokenizer interface to the worker
            go tokenWorker(tokenizer, jobs, results, &wg)
        }

        // Send jobs (now includes FileInfo from web URLs)
        filesToProcess := 0
        for _, file := range allFilesMaster {
            if file.IsDir { // Directories don't need token counting
                results <- file
            } else {
                // Send files and web content (as FileInfo) to workers
                jobs <- file
                filesToProcess++
            }
        }
        close(jobs)

        // Wait for workers to finish
        wg.Wait()
        close(results)

        // Collect results
        processedFiles = make([]FileInfo, 0, len(allFilesMaster))
        for res := range results {
            processedFiles = append(processedFiles, res)
        }

    } else {
        // If token counting is disabled, just use the initially collected files
```

```
            processedFiles = allFilesMaster
        }
        // --- End Token Counting ---

        // --- Aggregation and Summary (using processedFiles) ---
        var totalFiles int
        var totalSize, totalTokens int64
        for _, file := range processedFiles {
            if !file.IsDir {
                totalFiles++
                totalSize += file.Size
                if !disableTokens {
                    totalTokens += int64(file.TokenCount)
                }
            }
        }

        summary := Summary{
            TotalFiles:  totalFiles,
            TotalSize:   totalSize,
            TotalTokens: int(totalTokens),
        }

        // --- Output Generation (using processedFiles) ---
        if pdfOutputFile != "" {
            // Prioritize PDF output if the flag is set
            err = generatePDF(processedFiles, summary, outputFormat, langData, pdfOutputFile)
            if err != nil {
                fmt.Fprintf(os.Stderr, "Error generating PDF: %v\n", err)
                // Optionally, print to stdout as fallback
            }
        } else { // Handle non-PDF output (file, clipboard, stdout)
            // Generate the output string only when not creating PDF
            var outputBuilder strings.Builder
            if outputFormat == "tree" || outputFormat == "both" {
                if len(finalInputPaths) == 1 && isDir(finalInputPaths[0]) { // Check original
single input path type
                    rootNode := buildTree(processedFiles, finalInputPaths[0])
                    outputBuilder.WriteString(printTree(rootNode))
                } else if len(processedFiles) > 0 { // Check if any files were processed
                    // If multiple inputs or single file input, show the message
                    outputBuilder.WriteString("Tree view generated for single directory input
only.\nFiles found:\n")
                    sort.Slice(processedFiles, func(i, j int) bool {
                        return processedFiles[i].Path < processedFiles[j].Path
                    })
                    for _, file := range processedFiles {
                        outputBuilder.WriteString(fmt.Sprintf("- %s\n", file.Path))
                    }
                }
                if outputFormat == "both" {
                    outputBuilder.WriteString("\n")
                }
            }
            if outputFormat == "files" || outputFormat == "both" {
```

```
                outputBuilder.WriteString(printFiles(processedFiles, !disableTokens))
            }
            // Add summary to the output string
            outputBuilder.WriteString("\n--- Summary ---\n")
            outputBuilder.WriteString(fmt.Sprintf("Total files processed: %d\n", summary.
TotalFiles))
            outputBuilder.WriteString(fmt.Sprintf("Total size: %d bytes\n", summary.TotalSize))
            if !disableTokens {
                outputBuilder.WriteString(fmt.Sprintf("Total tokens: %d\n", summary.TotalTokens))
            }
            if failedPaths > 0 {
                outputBuilder.WriteString(fmt.Sprintf("Paths failed to process: %d\n", failedPaths
))
            }

            // Declare and assign finalOutput here
            finalOutput := outputBuilder.String()

            // Now handle the destination for the generated string
            if outputFile != "" {
                // Save to text file
                err = os.WriteFile(outputFile, []byte(finalOutput), 0644)
                if err != nil {
                    fmt.Fprintf(os.Stderr, "Error writing to file %s: %v\n", outputFile, err)
                }
                fmt.Printf("Output saved to %s\n", outputFile)
            } else if copyToClipboard {
                // Copy to clipboard
                err = clipboard.WriteAll(finalOutput)
                if err != nil {
                    fmt.Fprintf(os.Stderr, "Error writing to clipboard: %v\n", err)
                    fmt.Println("\n--- Output (clipboard failed) ---")
                    fmt.Println(finalOutput)
                } else {
                    fmt.Println("Output copied to clipboard.")
                }
            } else { // Default to stdout
                fmt.Println(finalOutput)
            }
        } // End of non-PDF output handling

        // --- End Main Logic ---
    },
}

func init() {
    // Initialize config first, then languages
    cobra.OnInitialize(initConfig, initLanguages)

    // --- Flag Definitions & Viper Binding ---
    // Optional: Allow specifying config file via flag
    // rootCmd.PersistentFlags().StringVar(&cfgFile, "config", "", "config file (default is
$HOME/.config/iris/config.toml)")

    // Filtering
```

```
    rootCmd.Flags().StringVarP(&includePatterns, "include", "i", "",  Additional patterns to
include (comma-separated, e.g. *.rs,*.go) )
    viper.BindPFlag("include", rootCmd.Flags().Lookup("include"))
    rootCmd.Flags().StringVarP(&excludePatterns, "exclude", "e", "", "Additional patterns to
exclude (comma-separated)")
    viper.BindPFlag("exclude", rootCmd.Flags().Lookup("exclude"))
    viper.BindPFlag("default_excludes", rootCmd.Flags().Lookup("exclude")) // Allow config
override via default_excludes
    rootCmd.Flags().Int64VarP(&maxSizeBytes, "max-size", "s", 0, "Maximum file size in bytes (0
for no limit)")
    viper.BindPFlag("max_size", rootCmd.Flags().Lookup("max-size"))
    rootCmd.Flags().IntVar(&maxDepth, "max-depth", 0, "Maximum directory depth to traverse (0 for
no limit)")
    viper.BindPFlag("max_depth", rootCmd.Flags().Lookup("max-depth"))
    rootCmd.Flags().BoolVarP(&showHidden, "hidden", "H", false, "Show hidden files and
directories")
    viper.BindPFlag("hidden", rootCmd.Flags().Lookup("hidden"))
    rootCmd.Flags().BoolVar(&noIgnore, "no-ignore", false, "Don't respect .gitignore files")
    viper.BindPFlag("no_ignore", rootCmd.Flags().Lookup("no-ignore")) // Use snake_case for viper
key

    // Output
    rootCmd.Flags().StringVarP(&outputFormat, "output", "o", "both", "Output format: tree, files,
or both")
    viper.BindPFlag("output", rootCmd.Flags().Lookup("output"))
    viper.BindPFlag("default_output_format", rootCmd.Flags().Lookup("output"))
    rootCmd.Flags().StringVarP(&outputFile, "file", "f", "", "Save output to specified file")
    viper.BindPFlag("file", rootCmd.Flags().Lookup("file"))
    rootCmd.Flags().BoolVarP(&printToStdout, "print", "p", false, "Print to stdout (default unless
-f, -c, or --pdf used)")
    viper.BindPFlag("print", rootCmd.Flags().Lookup("print"))
    rootCmd.Flags().BoolVarP(&copyToClipboard, "clipboard", "c", false, "Copy output to clipboard"
)
    viper.BindPFlag("clipboard", rootCmd.Flags().Lookup("clipboard"))

    // Processing
    rootCmd.Flags().IntVarP(&numThreads, "threads", "t", 0, "Number of threads for parallel
processing (0 for auto)")
    viper.BindPFlag("threads", rootCmd.Flags().Lookup("threads"))

    // Token Counting
    rootCmd.Flags().BoolVar(&disableTokens, "no-tokens", false, "Disable token counting")
    viper.BindPFlag("no_tokens", rootCmd.Flags().Lookup("no-tokens"))
    rootCmd.Flags().StringVar(&tokenizerType, "tokenizer", "tiktoken", "Tokenizer to use: tiktoken
or huggingface")
    viper.BindPFlag("tokenizer", rootCmd.Flags().Lookup("tokenizer"))
    viper.BindPFlag("default_tokenizer", rootCmd.Flags().Lookup("tokenizer"))
    rootCmd.Flags().StringVar(&tokenizerModel, "model", "", "Model name for tokenizer (e.g.,
gpt-4o, gpt2)")
    viper.BindPFlag("model", rootCmd.Flags().Lookup("model"))
    viper.BindPFlag("default_tokenizer_model", rootCmd.Flags().Lookup("model"))
    rootCmd.Flags().StringVar(&tokenizerFile, "tokenizer-file", "", "Path to local tokenizer file"
)
    viper.BindPFlag("tokenizer_file", rootCmd.Flags().Lookup("tokenizer-file"))
```

```
    // Web Specific
    rootCmd.Flags().BoolVar(&traverseLinks, "traverse-links", false, "Traverse links when
processing URLs")
    viper.BindPFlag("traverse_links", rootCmd.Flags().Lookup("traverse-links"))
    rootCmd.Flags().IntVar(&linkDepth, "link-depth", 1, "Maximum depth to traverse links")
    viper.BindPFlag("link_depth", rootCmd.Flags().Lookup("link-depth"))
    viper.BindPFlag("default_link_depth", rootCmd.Flags().Lookup("link-depth"))

    // PDF Output
    rootCmd.Flags().StringVar(&pdfOutputFile, "pdf", "", "Save output as PDF")
    viper.BindPFlag("pdf", rootCmd.Flags().Lookup("pdf"))

    // Interactive Mode
    rootCmd.Flags().BoolVar(&interactiveMode, "interactive", false, "Opens interactive file picker
(? for help)")
    viper.BindPFlag("interactive", rootCmd.Flags().Lookup("interactive"))

    // Set Viper defaults (matching Glimpse example where possible)
    viper.SetDefault("max_size", 10485760) // 10MB
    viper.SetDefault("max_depth", 20)
    viper.SetDefault("default_output_format", "both")
    viper.SetDefault("default_tokenizer", "tiktoken")
    viper.SetDefault("default_tokenizer_model", "") // Rely on tokenizer specific defaults
    viper.SetDefault("traverse_links", false)
    viper.SetDefault("default_link_depth", 1)
    viper.SetDefault("default_excludes", []string{
        "**/.git/**",
        "**/target/**",
        "**/node_modules/**",
    })
    // Note: We bind the 'exclude' flag to 'default_excludes' as well,
    // so the config file setting can provide the default value for the flag.
    // If the -e flag is explicitly used, it overrides the config.

    // Set other viper defaults based on flag defaults if needed, though BindPFlag usually handles
this.
    viper.SetDefault("hidden", false)
    viper.SetDefault("no_ignore", false)
    viper.SetDefault("threads", 0)
    viper.SetDefault("no_tokens", false)
    viper.SetDefault("interactive", false)
}

// initConfig reads in config file and ENV variables if set.
func initConfig() {
    if cfgFile != "" {
        // Use config file from the flag.
        viper.SetConfigFile(cfgFile)
    } else {
        // Find home directory.
        home, err := os.UserHomeDir()
        cobra.CheckErr(err)

        // Search config in home/.config/iris directory with name "config" (without extension).
        viper.AddConfigPath(filepath.Join(home, ".config", "iris"))
```

```
        viper.AddConfigPath(".") // Also look in current directory
        viper.SetConfigName("config")
        viper.SetConfigType("toml")
    }

    viper.AutomaticEnv() // read in environment variables that match IRIS_*
    viper.SetEnvPrefix("IRIS")
    viper.SetEnvKeyReplacer(strings.NewReplacer(".", "_", "-", "_"))

    // If a config file is found, read it in.
    if err := viper.ReadInConfig(); err == nil {
        fmt.Fprintln(os.Stderr, "Using config file:", viper.ConfigFileUsed())
    } else {
        if _, ok := err.(viper.ConfigFileNotFoundError); ok {
            // Config file not found; ignore error if desired
            fmt.Fprintln(os.Stderr, "No config file found, using defaults and flags.")
        } else {
            // Config file was found but another error was produced
            fmt.Fprintf(os.Stderr, "Error reading config file: %s\n", err)
        }
    }

    // After loading config, potentially update flag variables if needed
    // Cobra/Viper binding should handle this - the flag variables like maxDepth
    // should now hold the final value from Default < Config < Env < Flag.
    // Example: Update excludePatterns based on combined sources if needed
    // The default_excludes from config will set the default for the exclude flag.
    // If -e is used, it overrides. If neither, the flag default ("" initially) is used.
    // Let's explicitly load the excludes from viper IF the flag wasn't set.
    if !rootCmd.Flags().Changed("exclude") {
        excludePatterns = strings.Join(viper.GetStringSlice("default_excludes"), ",")
    }
    // Similar logic could apply to other defaults if direct variable access is preferred over
flags.
}

// initLanguages loads the language definitions.
func initLanguages() {
    var err error
    langData, err = loadLanguageData()
    if err != nil {
        // Log error but don't necessarily fail the program?
        // Language filtering will simply not be applied.
        fmt.Fprintf(os.Stderr, "Warning: Could not load language definitions: %v\n", err)
        fmt.Fprintln(os.Stderr, "Proceeding without language-based filtering.")
        langData = nil // Ensure it's nil if loading failed
    }
}

func main() {
    // initConfig() is called via cobra.OnInitialize(initConfig)
    rootCmd.Execute()
}

// Helper function to check if a path is a directory (used in output generation)
```

```
func isDir(path string) bool {
    info, err := os.Stat(path)
    if err != nil {
        return false // Treat errors as non-directories for safety
    }
    return info.IsDir()
}

// tokenWorker now accepts the Tokenizer interface.
func tokenWorker(tk Tokenizer, jobs <-chan FileInfo, results chan<- FileInfo, wg *sync.WaitGroup)
{
    defer wg.Done()
    for file := range jobs {
        if file.IsDir {
            results <- file
            continue
        }

        var content []byte
        var readErr error

        if file.Content != nil { // Use pre-loaded content (from web processing)
            content = file.Content
        } else { // Read from disk for local files/git files
            content, readErr = os.ReadFile(file.Path)
        }

        if readErr != nil {
            fmt.Fprintf(os.Stderr, "Warning: worker could not read file %s: %v\n", file.Path,
readErr)
            file.Error = readErr
        } else if len(content) > 0 { // Only count tokens if content is available and read
successfully
            // Use the interface method to count tokens
            file.TokenCount = tk.CountTokens(string(content))
        } else {
            file.TokenCount = 0
        }
        results <- file
    }
}

// Placeholder for web URL check
func isWebURL(input string) bool {
    return strings.HasPrefix(input, "http://") || strings.HasPrefix(input, "https://")
}
```

File: output.go

Tokens: 1265

```go
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "sort"
    "strings"
)

// Node represents an entry in the directory tree structure.
type Node struct {
    Name     string
    Path     string
    IsDir    bool
    Size     int64 // Relevant for files
    Children []*Node
}

// buildTree constructs a hierarchical tree from a flat list of FileInfo.
func buildTree(files []FileInfo, rootPath string) *Node {
    // Normalize the root path
    cleanRootPath := filepath.Clean(rootPath)
    root := &Node{Name: filepath.Base(cleanRootPath), Path: cleanRootPath, IsDir: true}
    nodes := make(map[string]*Node)
    nodes[cleanRootPath] = root

    // Sort files by path to ensure parents are processed before children
    sort.Slice(files, func(i, j int) bool {
        return files[i].Path < files[j].Path
    })

    for _, file := range files {
        cleanPath := filepath.Clean(file.Path)
        parentDir := filepath.Dir(cleanPath)
        baseName := filepath.Base(cleanPath)

        parentNode, exists := nodes[parentDir]
        if !exists {
            // This might happen if a directory itself was filtered out, but not its contents.
            // Or if the input list doesn't represent a fully connected tree.
            // For simplicity, let's assume parents exist or create intermediate ones if
necessary.
            // A more robust approach might be needed depending on filtering behavior.
            fmt.Fprintf(os.Stderr, "Warning: Parent node not found for %s, skipping in tree view.
\n", cleanPath)
            continue
            // Alternatively, create missing parent nodes recursively upwards
        }

        node := &Node{
            Name:  baseName,
```

```
            Path:  cleanPath,
            IsDir: file.IsDir,
            Size:  file.Size,
        }

        parentNode.Children = append(parentNode.Children, node)
        if file.IsDir {
            nodes[cleanPath] = node // Register directory node for future children
        }
    }

    // Sort children alphabetically within each node
    sortChildren(root)

    return root
}

// sortChildren recursively sorts the children of a node alphabetically.
func sortChildren(node *Node) {
    if !node.IsDir || len(node.Children) == 0 {
        return
    }

    sort.Slice(node.Children, func(i, j int) bool {
        // Optional: Sort directories before files
        // if node.Children[i].IsDir != node.Children[j].IsDir {
        //  return node.Children[i].IsDir // true (dir) comes before false (file)
        // }
        return node.Children[i].Name < node.Children[j].Name
    })

    for _, child := range node.Children {
        sortChildren(child)
    }
}

// printTree generates the string representation of the tree.
func printTree(root *Node) string {
    var builder strings.Builder
    // Print root name separately, then start recursion for children
    builder.WriteString(root.Name)
    builder.WriteString("\n")
    printNode(&builder, root.Children, "")
    return builder.String()
}

// printNode is a helper function for recursively printing tree nodes.
func printNode(builder *strings.Builder, children []*Node, prefix string) {
    for i, node := range children {
        connector := "â”œâ”€â”€ "
        newPrefix := prefix + "â”‚   "
        if i == len(children)-1 {
            connector = "â””â”€â”€ "
            newPrefix = prefix + "    "
        }
```

```
        builder.WriteString(prefix)
        builder.WriteString(connector)
        builder.WriteString(node.Name)
        // Optionally add size or other info here
        // if !node.IsDir {
        //  builder.WriteString(fmt.Sprintf(" (%d bytes)", node.Size))
        // }
        builder.WriteString("\n")

        if node.IsDir && len(node.Children) > 0 {
            printNode(builder, node.Children, newPrefix)
        }
    }
}

// printFiles generates the string representation for the 'files' output format.
func printFiles(files []FileInfo, includeTokens bool) string {
    var builder strings.Builder
    sort.Slice(files, func(i, j int) bool { // Sort by path for consistent output
        return files[i].Path < files[j].Path
    })

    for _, file := range files {
        if file.IsDir {
            continue // Skip directories for 'files' output format
        }

        builder.WriteString(fmt.Sprintf("File: %s\n", file.Path))
        if includeTokens {
            if file.Error != nil {
                builder.WriteString(fmt.Sprintf("Tokens: Error (%v)\n", file.Error)) // Indicate
error during token count
            } else {
                builder.WriteString(fmt.Sprintf("Tokens: %d\n", file.TokenCount))
            }
        }
        builder.WriteString(strings.Repeat("=", 50))
        builder.WriteString("\n")

        // Read file content
        // Note: We re-read the file here. For efficiency, especially with parallel processing
later,
        // we could read the content once during the token counting phase in main.go and store it
in FileInfo.
        // However, this might significantly increase memory usage for large codebases.
        // Let's stick with re-reading for now, similar to Glimpse's likely approach.
        content, err := os.ReadFile(file.Path)
        if err != nil {
            // If token counting failed due to read error, file.Error might already be set.
            // We still report the error here in the content section.
            builder.WriteString(fmt.Sprintf("Error reading file: %v\n", err))
        } else {
            builder.Write(content)
        }
```

```
        // Ensure consistent line breaks after content
        if len(content) > 0 && content[len(content)-1] != '\n' {
            builder.WriteString("\n")
        }
        builder.WriteString("\n") // Add blank line between files
    }
    return builder.String()
}
```

File: pdf.go

Tokens: 2282

```go
package main

import (
    "fmt"
    "os"
    "sort"
    "strings"

    "github.com/alecthomas/chroma/v2"
    "github.com/alecthomas/chroma/v2/lexers"
    "github.com/alecthomas/chroma/v2/styles"
    "github.com/jung-kurt/gofpdf"
)

const (
    pdfPageWidth  = 210 // A4 width in mm
    pdfPageHeight = 297 // A4 height in mm
    pdfMargin     = 10  // Margin in mm
    pdfLineHeight = 5   // Line height in mm
    pdfFontSize   = 9   // Reduced font size slightly for better code fit
    pdfTabWidth   = 4   // Number of spaces for a tab
)

// generatePDF takes the collected FileInfo and Summary, generates syntax-highlighted
// PDF output according to the selected format (tree, files, both).
func generatePDF(files []FileInfo, summary Summary, outputFormat string, langData *
LoadedLanguageData, outputPath string) error {
    fmt.Printf("Generating PDF output at: %s (Format: %s)\n", outputPath, outputFormat)

    pdf := gofpdf.New("P", "mm", "A4", "") // Portrait, mm, A4, default font dir
    pdf.SetMargins(pdfMargin, pdfMargin, pdfMargin)
    pdf.SetAutoPageBreak(true, pdfMargin) // Enable auto page breaks
    pdf.AddPage()

    // Select a Chroma style
    // style := styles.Get("monokai") // Example: Monokai
    style := styles.Get("github") // Example: GitHub style
    if style == nil {
        style = styles.Fallback
    }

    // --- Output Tree (if requested) ---
    if outputFormat == "tree" || outputFormat == "both" {
        // For the tree, we don't have syntax highlighting, just print the tree structure.
        // Assuming the first input was a directory if tree format makes sense.
        // Note: This assumes tree generation is still relevant based on input types.
        // We might need to adjust how the tree string is generated/passed.
        // Let's rebuild the tree string here for simplicity.
        treeString := buildTreeString(files) // Need a helper to get tree string

        pdf.SetFont("Courier", "", pdfFontSize)
        pdf.SetTextColor(0, 0, 0)
```

```
        pdf.MultiCell(pdfPageWidth-2*pdfMargin, pdfLineHeight, treeString, "", "L", false)
        pdf.Ln(pdfLineHeight)
    }

    // --- Output Files (if requested) ---
    if outputFormat == "files" || outputFormat == "both" {
        // Sort files for consistent output
        sort.Slice(files, func(i, j int) bool {
            return files[i].Path < files[j].Path
        })

        for _, file := range files {
            if file.IsDir {
                continue
            }

            // Add File Header
            pdf.SetFont("Helvetica", "B", pdfFontSize+1)
            pdf.SetTextColor(0, 0, 0)
            pdf.MultiCell(pdfPageWidth-2*pdfMargin, pdfLineHeight, fmt.Sprintf("File: %s", file.
Path), "", "L", false)
            pdf.Ln(pdfLineHeight / 2)

            // Add Token Count if available
            if file.TokenCount > 0 || file.Error != nil { // Assuming TokenCount is populated
                pdf.SetFont("Helvetica", "", pdfFontSize-1)
                tokenStr := ""
                if file.Error != nil {
                    tokenStr = fmt.Sprintf("Tokens: Error (%v)", file.Error)
                } else {
                    tokenStr = fmt.Sprintf("Tokens: %d", file.TokenCount)
                }
                pdf.MultiCell(pdfPageWidth-2*pdfMargin, pdfLineHeight, tokenStr, "", "L", false)
                pdf.Ln(pdfLineHeight / 2)
            }

            pdf.Line(pdfMargin, pdf.GetY(), pdfPageWidth-pdfMargin, pdf.GetY()) // Separator line
            pdf.Ln(pdfLineHeight / 2)

            // Get content (read again or use pre-loaded if available)
            var content []byte
            var readErr error
            if file.Content != nil {
                content = file.Content
            } else {
                content, readErr = os.ReadFile(file.Path)
            }

            if readErr != nil {
                pdf.SetFont("Courier", "", pdfFontSize)
                pdf.SetTextColor(255, 0, 0) // Red for error
                pdf.MultiCell(pdfPageWidth-2*pdfMargin, pdfLineHeight, fmt.Sprintf("Error reading
file: %v", readErr), "", "L", false)
            } else {
                // Perform Syntax Highlighting
```

```
                err := writeHighlightedCode(pdf, style, string(content), file.Path, langData)
                if err != nil {
                    // Fallback to plain text if highlighting fails?
                    fmt.Fprintf(os.Stderr, "Warning: Syntax highlighting failed for %s: %v.
Writing plain text.\n", file.Path, err)
                    pdf.SetFont("Courier", "", pdfFontSize)
                    pdf.SetTextColor(0, 0, 0)
                    pdf.MultiCell(pdfPageWidth-2*pdfMargin, pdfLineHeight, string(content), "",
"L", false)
                }
            }
            pdf.AddPage() // Start each new file on a new page (or manage breaks better)
        }
    }

    // --- Output Summary ---
    // Ensure we are on the last page or add a new one if needed
    // (Complex page management might be needed if files output is long)
    pdf.SetFont("Helvetica", "B", pdfFontSize+1)
    pdf.SetTextColor(0, 0, 0)
    pdf.Ln(pdfLineHeight)
    pdf.MultiCell(pdfPageWidth-2*pdfMargin, pdfLineHeight, "--- Summary ---", "", "L", false)
    pdf.Ln(pdfLineHeight / 2)

    pdf.SetFont("Helvetica", "", pdfFontSize)
    summaryString := fmt.Sprintf("Total files processed: %d\nTotal size: %d bytes", summary.
TotalFiles, summary.TotalSize)
    if summary.TotalTokens > 0 { // Assuming token counting wasn't disabled
        summaryString += fmt.Sprintf("\nTotal tokens: %d", summary.TotalTokens)
    }
    // Add failed paths count? summary doesn't store it currently.
    pdf.MultiCell(pdfPageWidth-2*pdfMargin, pdfLineHeight, summaryString, "", "L", false)

    // --- Save PDF ---
    err := pdf.OutputFileAndClose(outputPath)
    if err != nil {
        return fmt.Errorf("failed to save PDF to %s: %w", outputPath, err)
    }

    fmt.Printf("Successfully saved PDF to %s\n", outputPath)
    return nil
}

// buildTreeString creates the simple text representation of the file tree.
// This is a placeholder - ideally uses the same logic as printTree in output.go
func buildTreeString(files []FileInfo) string {
    // This is inefficient - ideally, output.go provides this directly
    // For now, let's just list files as a placeholder
    var builder strings.Builder
    builder.WriteString("File Tree (Placeholder):\n")
    sort.Slice(files, func(i, j int) bool { return files[i].Path < files[j].Path })
    for _, file := range files {
        prefix := "  "
        if file.IsDir {
            prefix = "D "
```

```
        }
        builder.WriteString(fmt.Sprintf("%s%s\n", prefix, file.Path))
    }
    return builder.String()
}

// writeHighlightedCode takes code content, analyzes it, and writes it to the PDF with styles.
func writeHighlightedCode(pdf *gofpdf.Fpdf, style *chroma.Style, codeContent, filePath string,
langData *LoadedLanguageData) error {
    // 1. Determine the lexer
    lexer := lexers.Analyse(codeContent) // Try analyzing content
    if lexer == nil {
        // Fallback to guessing by filename using langData if available
        if lang, ok := langData.GetLanguageForFile(filePath); ok {
            lexer = lexers.Get(lang) // Chroma might use different names than languages.yml
        }
    }
    if lexer == nil {
        lexer = lexers.Fallback // Default plain text lexer
    }
    lexer = chroma.Coalesce(lexer)

    // 2. Tokenize
    iterator, err := lexer.Tokenise(nil, codeContent)
    if err != nil {
        return fmt.Errorf("tokenization failed: %w", err)
    }

    // 3. Iterate and Write Tokens
    pdf.SetFont("Courier", "", pdfFontSize) // Base font

    for token := iterator(); token != chroma.EOF; token = iterator() {
        entry := style.Get(token.Type)
        styleStr := ""
        if entry.Bold == chroma.Yes {
            styleStr += "B"
        }
        if entry.Italic == chroma.Yes {
            styleStr += "I"
        }
        // Underline not directly supported by basic SetFontStyle
        pdf.SetFontStyle(styleStr)

        if entry.Colour.IsSet() {
            // Use Red(), Green(), Blue() which return uint8 (0-255)
            pdf.SetTextColor(int(entry.Colour.Red()), int(entry.Colour.Green()), int(entry.Colour.
Blue()))
        } else {
            // Use default text color (e.g., black or style's default)
            fg := style.Get(chroma.Text).Colour
            if fg.IsSet() {
                pdf.SetTextColor(int(fg.Red()), int(fg.Green()), int(fg.Blue()))
            } else {
                pdf.SetTextColor(0, 0, 0) // Fallback to black
            }
```

```
        }

        // Handle background color? More complex, requires drawing rects.
        // Let's ignore background for simplicity.

        // Write the token value, handling tabs and newlines
        tokenValue := strings.ReplaceAll(token.Value, "\t", strings.Repeat(" ", pdfTabWidth))
        // gofpdf's Write handles basic line breaks within the cell width
        // Need to manage X, Y position manually for precise control over wrapping/indentation
        // Using Write for simplicity, may have wrapping issues.
        pdf.Write(pdfLineHeight, tokenValue)
    }
    pdf.Ln(-1) // Ensure we move to next line after last token

    return nil
}
```

File: processor.go

Tokens: 2384

```go
package main

import (
    "fmt"
    "io/fs"
    "os"
    "path/filepath"
    "strings"

    gitignore "github.com/monochromegane/go-gitignore"
)

// processLocalPath handles a single local file or directory path.
// It now accepts LoadedLanguageData for filtering.
func processLocalPath(path string, langData *LoadedLanguageData) ([]FileInfo, error) {
    info, err := os.Stat(path)
    if err != nil {
        return nil, fmt.Errorf("error accessing path %s: %w", path, err)
    }

    var files []FileInfo

    if info.IsDir() {
        // It's a directory, start walking
        fmt.Printf("Processing directory: %s\n", path) // Placeholder
        // Pass langData to walkDirectory
        files, err = walkDirectory(path, langData)
        if err != nil {
            return nil, err
        }
    } else {
        // It's a single file
        fmt.Printf("Processing file: %s\n", path) // Placeholder
        // Apply filters even for single files, passing langData
        keep, err := shouldKeepFile(path, info, langData)
        if err != nil {
            fmt.Fprintf(os.Stderr, "Warning: error checking file %s: %v\n", path, err)
            // Decide if we should error out or just skip
        } else if keep {
            fileInfo := FileInfo{
                Path:  path,
                Size:  info.Size(),
                Mode:  info.Mode(),
                IsDir: false,
            }
            files = append(files, fileInfo)
        } else {
            fmt.Printf("Skipping single file due to filters: %s\n", path)
        }
    }

    return files, nil
```

```
}

// parsePatterns splits a comma-separated string of patterns into a slice.
func parsePatterns(patterns string) []string {
    if patterns == "" {
        return nil
    }
    return strings.Split(patterns, ",")
}

// matchesAnyPattern checks if the given name matches any of the provided glob patterns.
func matchesAnyPattern(name string, patterns []string) (bool, error) {
    for _, pattern := range patterns {
        matched, err := filepath.Match(pattern, name)
        if err != nil {
            return false, fmt.Errorf("invalid glob pattern '%s': %w", pattern, err)
        }
        if matched {
            return true, nil
        }
    }
    return false, nil
}

// walkDirectory recursively walks a directory, respecting filters and .gitignore.
// It now accepts LoadedLanguageData for filtering.
func walkDirectory(root string, langData *LoadedLanguageData) ([]FileInfo, error) {
    var files []FileInfo
    var ignoreMatcher gitignore.IgnoreMatcher

    parsedIncludes := parsePatterns(includePatterns)
    parsedExcludes := parsePatterns(excludePatterns)
    // Check if explicit includes were provided. If not, language filtering might apply.
    hasExplicitIncludes := len(parsedIncludes) > 0

    if !noIgnore {
        // TODO: Consider handling nested .gitignore files
        // go-gitignore primarily works with one .gitignore at the root level of the match.
        // For full git compatibility, might need a more complex walker or library.
        gitIgnorePath := filepath.Join(root, ".gitignore")
        if _, err := os.Stat(gitIgnorePath); err == nil {
            matcher, err := gitignore.NewGitIgnore(gitIgnorePath)
            if err != nil {
                fmt.Fprintf(os.Stderr, "Warning: could not parse .gitignore file %s: %v\n",
gitIgnorePath, err)
            } else {
                ignoreMatcher = matcher
            }
        }
    }

    err := filepath.WalkDir(root, func(path string, d fs.DirEntry, err error) error {
        if err != nil {
            fmt.Fprintf(os.Stderr, "Warning: error accessing path %s: %v\n", path, err)
            // Optionally return err to stop walk, or fs.SkipDir for directory errors
```

```
            return nil // Report and continue
        }

        // Skip root directory itself
        if path == root {
            return nil
        }

        // --- Filtering Logic ---
        baseName := d.Name()
        isDir := d.IsDir()

        // 1. Hidden Files/Dirs
        if !showHidden && isHidden(baseName) {
            if isDir {
                return fs.SkipDir
            }
            return nil
        }

        // 2. .gitignore
        // Need the path relative to the gitignore file (usually the root)
        relPathForIgnore, _ := filepath.Rel(root, path)
        if ignoreMatcher != nil && ignoreMatcher.Match(relPathForIgnore, isDir) {
            if isDir {
                return fs.SkipDir
            }
            return nil
        }

        // 3. Max Depth
        relPath, _ := filepath.Rel(root, path)
        currentDepth := countPathSeparators(relPath)
        if maxDepth > 0 && currentDepth >= maxDepth {
            if isDir {
                return fs.SkipDir // Reached max depth, skip this directory
            }
            // If it's a file at max depth, it might still be processed below
        }

        // Apply Include/Exclude/Language Filters
        // If it's a directory, we check excludes but not includes/language yet (allow traversal)
        if isDir {
            // 4a. Exclude Pattern Match (Directories)
            excluded, err := matchesAnyPattern(baseName, parsedExcludes)
            if err != nil {
                fmt.Fprintf(os.Stderr, "Warning: error in exclude pattern matching for %s: %v\n",
path, err)
                // Decide how to handle pattern errors - skip file or ignore pattern
            }
            if excluded {
                return fs.SkipDir // Skip excluded directories
            }
            // Allow traversal of non-excluded directories
        } else {
```

```
            // Apply full filters to files
            fileName := baseName

            // 4a. Exclude Pattern Match (Files)
            excluded, err := matchesAnyPattern(fileName, parsedExcludes)
            if err != nil {
                fmt.Fprintf(os.Stderr, "Warning: error in exclude pattern matching for %s: %v\n",
path, err)
                // Decide how to handle pattern errors - skip file or ignore pattern
            }
            if excluded {
                return nil // Skip excluded files
            }

            // 4b. Include Pattern Match OR Language Match (Files)
            keepFile := false
            if hasExplicitIncludes {
                // If includes are specified, use them
                included, err := matchesAnyPattern(fileName, parsedIncludes)
                if err != nil {
                    fmt.Fprintf(os.Stderr, "Warning: error in include pattern matching for %s: %v
\n", path, err)
                }
                if included {
                    keepFile = true
                }
            } else if langData != nil {
                // If no includes specified AND langData exists, check language
                if _, knownLang := langData.GetLanguageForFile(path); knownLang {
                    keepFile = true
                }
            } else {
                // No includes, no langData -> keep all non-excluded files
                keepFile = true
            }

            if !keepFile {
                return nil // Skip files not matching includes or known languages (if applicable)
            }

            // 5. Max Size (apply only to files)
            var fileSize int64
            var fileMode fs.FileMode
            info, err := d.Info()
            if err != nil {
                fmt.Fprintf(os.Stderr, "Warning: could not get info for %s: %v\n", path, err)
                return nil // Skip file if info error
            }
            fileSize = info.Size()
            fileMode = info.Mode()
            if maxSizeBytes > 0 && fileSize > maxSizeBytes {
                return nil // Skip large files
            }

            // If file passes all filters, add it
```

```
            fileInfo := FileInfo{
                Path:  path,
                Size:  fileSize,
                Mode:  fileMode,
                IsDir: false,
            }
            files = append(files, fileInfo)
        }
        // --- End Filtering Logic ---

        return nil
    })

    if err != nil {
        return nil, fmt.Errorf("error walking directory %s: %w", root, err)
    }

    return files, nil
}

// shouldKeepFile checks if a single file (not in a walk) should be kept based on filters.
// It now accepts LoadedLanguageData for filtering.
func shouldKeepFile(path string, info fs.FileInfo, langData *LoadedLanguageData) (bool, error) {
    baseName := info.Name()

    // Hidden
    if !showHidden && isHidden(baseName) {
        return false, nil
    }

    // Gitignore - less relevant for single file args unless we load a relevant .gitignore?
    // Glimpse probably doesn't apply gitignore to explicit file args.

    // Include/Exclude/Language
    parsedIncludes := parsePatterns(includePatterns)
    parsedExcludes := parsePatterns(excludePatterns)
    hasExplicitIncludes := len(parsedIncludes) > 0

    excluded, err := matchesAnyPattern(baseName, parsedExcludes)
    if err != nil {
        return false, fmt.Errorf("exclude pattern error: %w", err)
    }
    if excluded {
        return false, nil
    }

    keepFile := false
    if hasExplicitIncludes {
        included, err := matchesAnyPattern(baseName, parsedIncludes)
        if err != nil {
            return false, fmt.Errorf("include pattern error: %w", err)
        }
        if included {
            keepFile = true
        }
```

```
    } else if langData != nil {
        if _, knownLang := langData.GetLanguageForFile(path); knownLang {
            keepFile = true
        }
    } else {
        keepFile = true // Keep if not excluded and no includes/lang specified
    }
    if !keepFile {
        return false, nil
    }

    // Max Size
    if maxSizeBytes > 0 && info.Size() > maxSizeBytes {
        return false, nil
    }

    return true, nil
}

// isHidden checks if a file path is hidden (starts with '.').
func isHidden(path string) bool {
    // Check for '.' or '..'
    if path == "." || path == ".." {
        return false
    }
    // Check if the actual base name starts with '.'
    baseName := filepath.Base(path)
    return len(baseName) > 0 && baseName[0] == '.'
}

// countPathSeparators counts the number of path separators in a relative path.
func countPathSeparators(path string) int {
    // Normalize path separators for consistency
    path = filepath.ToSlash(path)
    if path == "." || path == "" {
        return 0
    }
    // Don't count trailing slash if present
    return strings.Count(strings.Trim(path, "/"), "/")
}
```

File: tokenizer.go

Tokens: 1084

```go
package main

import (
    "fmt"
    "os"
    "strings"

    tiktoken "github.com/pkoukk/tiktoken-go"
    hf "github.com/sugarme/tokenizer"
    "github.com/sugarme/tokenizer/pretrained"
)

// Tokenizer is an interface for different tokenizer implementations.
type Tokenizer interface {
    CountTokens(text string) int
    Close() // Add a Close method for potential resource cleanup (like HF tokenizer)
}

// --- Tiktoken Wrapper ---

type TiktokenWrapper struct {
    ttk *tiktoken.Tiktoken
}

func (w *TiktokenWrapper) CountTokens(text string) int {
    if w.ttk == nil {
        return 0
    }
    tokens := w.ttk.EncodeOrdinary(text)
    return len(tokens)
}

func (w *TiktokenWrapper) Close() {
    // No explicit close needed for tiktoken-go
}

// --- HuggingFace (sugarme) Wrapper ---

type HFTokenizerWrapper struct {
    htk *hf.Tokenizer
}

func (w *HFTokenizerWrapper) CountTokens(text string) int {
    if w.htk == nil {
        return 0
    }
    en, err := w.htk.EncodeSingle(text)
    if err != nil {
        fmt.Fprintf(os.Stderr, "Warning: HF tokenizer failed to encode text: %v\n", err)
        return 0
    }
    return len(en.Tokens)
```

```
}

func (w *HFTokenizerWrapper) Close() {
    // sugarme/tokenizer doesn't seem to have an explicit Close/Free method
}

// --- Tokenizer Loading Logic ---

const defaultTiktokenModel = "gpt-4o" // Default if tokenizer is tiktoken
const defaultHFModel = "gpt2"         // Default if tokenizer is huggingface and no model
specified

// getTokenizer returns a tokenizer instance based on flags.
// It returns a Tokenizer interface.
func getTokenizer() (Tokenizer, error) {
    fmt.Printf("Initializing tokenizer (Type: %s, Model: %s, File: %s)\n", tokenizerType,
tokenizerModel, tokenizerFile)

    switch strings.ToLower(tokenizerType) {
    case "tiktoken":
        return loadTiktoken()
    case "huggingface":
        return loadHuggingFace()
    default:
        return nil, fmt.Errorf("unsupported tokenizer type: %s. Use 'tiktoken' or 'huggingface'",
tokenizerType)
    }
}

func loadTiktoken() (Tokenizer, error) {
    model := tokenizerModel
    if model == "" {
        model = defaultTiktokenModel
        fmt.Printf("No Tiktoken model specified, using default: %s\n", model)
    }

    tke, err := tiktoken.EncodingForModel(model)
    if err != nil {
        fmt.Printf("Warning: Tiktoken model '%s' not found, falling back to default '%s'. Error:
%v\n", model, defaultTiktokenModel, err)
        tke, err = tiktoken.EncodingForModel(defaultTiktokenModel)
        if err != nil {
            return nil, fmt.Errorf("failed to get tiktoken encoding for default model '%s': %w",
defaultTiktokenModel, err)
        }
    }
    return &TiktokenWrapper{ttk: tke}, nil
}

func loadHuggingFace() (Tokenizer, error) {
    if tokenizerFile != "" {
        // Load from local file
        fmt.Printf("Loading HuggingFace tokenizer from file: %s\n", tokenizerFile)
        ttk, err := pretrained.FromFile(tokenizerFile)
        if err != nil {
```

```
            return nil, fmt.Errorf("failed to load tokenizer from file %s: %w", tokenizerFile, err
)
        }
        return &HFTokenizerWrapper{htk: ttk}, nil
    } else {
        // Load from Hugging Face Hub
        model := tokenizerModel
        if model == "" {
            model = defaultHFModel
            fmt.Printf("No HuggingFace model specified, using default: %s\n", model)
        }
        fmt.Printf("Loading HuggingFace tokenizer for model: %s (this may download files)\n",
model)

        // sugarme/tokenizer uses CachedPath to download/find the tokenizer.json
        // We need the identifier used on the Hub (e.g., "bert-base-uncased")
        configFilePath, err := hf.CachedPath(model, "tokenizer.json")
        if err != nil {
            return nil, fmt.Errorf("failed to get cache path for model %s: %w", model, err)
        }

        ttk, err := pretrained.FromFile(configFilePath)
        if err != nil {
            return nil, fmt.Errorf("failed to load pretrained tokenizer for model %s (from %s):
%w", model, configFilePath, err)
        }
        return &HFTokenizerWrapper{htk: ttk}, nil
    }
}

// countTokens is now a method on the interface wrappers, no longer needed here.
/*
func countTokens(tke *tiktoken.Tiktoken, content []byte) int {
    ...
}
*/
```

File: types.go

Tokens: 184

```
package main

import "io/fs"

// FileInfo holds information about a processed file.
type FileInfo struct {
    Path       string
    Size       int64
    Mode       fs.FileMode
    Content    []byte // Content might be loaded conditionally based on output format
    TokenCount int    // Populated if token counting is enabled
    IsDir      bool   // Indicates if this is a directory entry
    Error      error  // Stores any error encountered while processing this file/dir
}

// Summary holds aggregated information about the processed items.
type Summary struct {
    TotalFiles  int
    TotalSize   int64
    TotalTokens int
}

// ProcessedItem represents either a FileInfo or a directory structure node.
// This could evolve as we implement tree/file output. For now, FileInfo covers both files and
directory entries discovered during traversal.
// We might need a separate DirectoryInfo later if we store nested structures.
```

File: web.go

Tokens: 1341

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "net/url"
    "os"
    "strings"

    md "github.com/JohannesKaufmann/html-to-markdown"
    "github.com/PuerkitoBio/goquery"
)

// processWebURLRecursive fetches content from a starting URL, converts it to Markdown,
// finds links, and recursively processes them up to maxDepth.
// It keeps track of visited URLs to avoid loops.
func processWebURLRecursive(startURL string, currentDepth, maxDepth int, visited map[string]bool)
([]FileInfo, error) {
    // Clean URL to avoid re-visiting due to fragments or slight variations
    parsedURL, err := url.Parse(startURL)
    if err != nil {
        return nil, fmt.Errorf("invalid start URL %s: %w", startURL, err)
    }
    parsedURL.Fragment = "" // Ignore fragments
    cleanURL := parsedURL.String()

    if currentDepth > maxDepth {
        fmt.Printf("Max depth (%d) reached, not processing: %s\n", maxDepth, cleanURL)
        return nil, nil
    }
    if visited[cleanURL] {
        fmt.Printf("Already visited, skipping: %s\n", cleanURL)
        return nil, nil
    }

    visited[cleanURL] = true
    fmt.Printf("Processing web URL (Depth %d): %s\n", currentDepth, cleanURL)

    // --- Fetch and Process Current URL ---
    res, err := http.Get(cleanURL)
    if err != nil {
        // Log error but continue traversal if possible? Or stop?
        fmt.Fprintf(os.Stderr, "Warning: failed to fetch URL %s: %v\n", cleanURL, err)
        return nil, nil // Skip this URL and its links on fetch error
    }
    defer res.Body.Close()

    if res.StatusCode < 200 || res.StatusCode >= 300 {
        fmt.Fprintf(os.Stderr, "Warning: failed to fetch URL %s: status code %d\n", cleanURL, res.
StatusCode)
        return nil, nil // Skip this URL
```

```
    }

    // Check content type - only parse HTML
    contentType := res.Header.Get("Content-Type")
    if !strings.Contains(strings.ToLower(contentType), "text/html") {
        fmt.Printf("Skipping non-HTML content type (%s) for URL: %s\n", contentType, cleanURL)
        return nil, nil
    }

    // Read the response body
    bodyBytes, err := io.ReadAll(res.Body)
    if err != nil {
        fmt.Fprintf(os.Stderr, "Warning: failed to read response body from %s: %v\n", cleanURL,
err)
        return nil, nil // Skip this URL
    }
    // --- End Fetch ---

    // --- Convert Current Page to Markdown ---
    converter := md.NewConverter("", true, nil)
    markdown, err := converter.ConvertString(string(bodyBytes))
    if err != nil {
        fmt.Fprintf(os.Stderr, "Warning: failed to convert HTML to Markdown for %s: %v\n",
cleanURL, err)
        // Create FileInfo with raw HTML or skip
        // Let's skip creating FileInfo if conversion fails, but still parse for links below.
    }

    var currentFiles []FileInfo
    if err == nil { // Only add FileInfo if conversion was successful
        fileInfo := FileInfo{
            Path:    cleanURL, // Use the cleaned URL
            Content: []byte(markdown),
            Size:    int64(len(markdown)),
            IsDir:   false,
        }
        currentFiles = append(currentFiles, fileInfo)
        fmt.Printf("Finished processing web URL: %s (Markdown size: %d bytes)\n", cleanURL,
fileInfo.Size)
    }
    // --- End Conversion ---

    // --- Find and Process Links (if not at max depth) ---
    if currentDepth < maxDepth {
        // Use goquery to parse the original HTML body bytes
        doc, err := goquery.NewDocumentFromReader(strings.NewReader(string(bodyBytes)))
        if err != nil {
            fmt.Fprintf(os.Stderr, "Warning: failed to parse HTML for link extraction from %s: %v
\n", cleanURL, err)
        } else {
            doc.Find("a[href]").Each(func(i int, s *goquery.Selection) {
                link, exists := s.Attr("href")
                if !exists || link == "" || strings.HasPrefix(link, "#") || strings.HasPrefix(
strings.ToLower(link), "mailto:") || strings.HasPrefix(strings.ToLower(link), "javascript:") {
                    return // Skip empty, fragment, mailto, or javascript links
```

```
                }

                // Resolve the link relative to the current page's URL
                resolvedURL, err := parsedURL.Parse(link)
                if err != nil {
                    fmt.Fprintf(os.Stderr, "Warning: could not resolve relative link '%s' on page
%s: %v\n", link, cleanURL, err)
                    return
                }

                // Only process HTTP/HTTPS URLs
                if resolvedURL.Scheme == "http" || resolvedURL.Scheme == "https://" {
                    // Recursively process the resolved link
                    linkedFiles, _ := processWebURLRecursive(resolvedURL.String(), currentDepth+1,
maxDepth, visited)
                    // We ignore the error from the recursive call to continue processing other
links
                    currentFiles = append(currentFiles, linkedFiles...)
                }
            })
        }
    }
    // --- End Link Processing ---

    return currentFiles, nil
}

// processWebURL remains as a simple, non-recursive entry point if needed,
// but the main logic will likely call processWebURLRecursive directly.
func processWebURL(url string) (FileInfo, error) {
    visited := make(map[string]bool)
    results, err := processWebURLRecursive(url, 0, 0, visited) // Call recursive with maxDepth 0
    if err != nil {
        return FileInfo{}, err
    }
    if len(results) == 0 {
        // This might happen if the initial URL fetch failed or conversion failed
        // Return an error consistent with previous behavior
        return FileInfo{}, fmt.Errorf("failed to process web URL %s (no content generated)", url)
    }
    return results[0], nil // Return the first result (the page itself)
}
```

--- Summary ---

Total files processed: 10
Total size: 58358 bytes
Total tokens: 15368