Project:
https://github.com/alexpetrean80/FLCD/tree/master/lexical_analyzer

Documentation for the Lexical Analyzer

"Scanner" is the exposed interface of the lexical analyzer. Along getters for the program internal form, constants and identifiers symbol tables, here it is defined the "Scan" method, which receives the path of the source file as a parameter and returns a slice of lexical errors.

```
type Scanner interface {

   Scan(path string) []error

   GetPIF() pif.PIF

   GetConstants() symtable.SymbolTable

   GetIdentifiers() symtable.SymbolTable

}
```

"scanner" is the implementation of the "Scanner" interface. It contains 2 symbol tables, one for identifiers and one for constants, and the program internal form.

```
type scanner struct {

   identifiers symtable.SymbolTable

   constants   symtable.SymbolTable

   pif         pif.PIF

}
```

"Scan" parses the source file with the path given as a parameter.

```
func (s *scanner) Scan(inFile string) (errs []error) {

The file is read as a byte slice.

   bytes, err := ioutil.ReadFile(inFile)

   if err != nil {

      log.Fatal(err.Error())

   }
```

Tokens are retrieved as a slice of strings and the length of the slice is stored in the "pl" variable.

```
   tokens := getTokens(bytes)

   pl := len(tokens)


   var tk string
```

We loop through all the tokens.

```
   for i := 0; i < pl; i++ {

      tk = tokens[i]
```

We perform a check for operators which are made of more than one character(:=, !=, <=, >=, ==).

```
      if contains([]string{":", "!", "<", ">", "="}, tk) && i < pl-1 {

         if tokens[i+1] == "=" {

            tk += tokens[i+1]

            i++

         }


      }
```

Here we check that all strings are properly terminated with a double quote.

```
      if tk == "\"" {

         ok := false

         for i = i + 1; i < pl; i++ {

            tk += tokens[i]

            if tokens[i] == "\"" {

               ok = true

               break
```

```
            }
         }


         if !ok {
            errs = append(errs, errors.NewLexical("matching quote not
found"))
         }


      }
      Here the tokens are classified and added to the respective data
structure.
      if isReservedWord(tk) || isOperator(tk) || isSeparator(tk) {
         s.pif.Add(tk, 0)
      } else if isConstant(tk) {
         s.constants.Add(tk)
         s.pif.Add(tk, 0)
      } else if isIdentifier(tk) {
         s.identifiers.Add(tk)
         s.pif.Add(tk, 0)
      } else {
         errs = append(errs, errors.NewLexical(fmt.Sprintf("invalid
token %s\n", tk)))
      }
   }
   In the end, the slice of errors is returned.
      return
}


"getTokens" retrieves all the tokens from the source file contents passed as
a slice of bytes.
func getTokens(content []byte) []string {
```

```
   var tks []string
We define a regex for all separators, and everything against it.
   rg, err := regexp.Compile(`[\+\-
\*/%!(:=)(!=)\s\t==\n(<=)(>=)=<>\(\)\{\}\[\];,]`)
   if err != nil {
      log.Fatal(err.Error())
   }

   matches := rg.FindAllIndex(content, -1)

We iterate through all the matches.
   for i, indexes := range matches {
      start := indexes[0]
      end := indexes[1]

      var prev int
The first token requires a particular case.
      if i == 0 {
         tks = append(tks, string(content[start:end]))
         prev = 0
      } else {
         prev = matches[i-1][1]
      }

We parse the token as a string and add them to the result.
      tk := string(content[prev:start])
      if len(strings.TrimSpace(tk)) > 0 {
         tks = append(tks, tk)
      }
```

```
      sep := string(content[start:end])
      if len(sep) > 0 {
         tks = append(tks, sep)
      }

   }
   In the end, we remove all whitespace tokens and return the rest
   tks = clearWhiteSpace(tks)
   return tks
}

"clearWhiteSpace" removes all the tokens made from whitespace
characters(space, tab, newline).
func clearWhiteSpace(tks []string) []string {
   var res []string
   for _, t := range tks {
      if len(strings.TrimSpace(t)) == 0 {
         continue
      }
      res = append(res, t)
   }

   return res
}

"isReservedWord" checks if the token passed as a parameter is a reserved
word.
func isReservedWord(token string) bool {
   reservedWords := []string{"char", "else", "func", "for", "if", "int",
"main", "nil", "string", "var", "print"}
```

```
   return contains(reservedWords, token)
}

"isOperator" checks if the token passed as a parameter is an operator.
func isOperator(token string) bool {
   operators := []string{"+", "-", "*", "/", "%", "=", ":=", "==", "!=",
"<", "<=", ">", ">=", "&&", "||"}
   return contains(operators, token)
}
"isSeparator" checks if the token passed as a parameter is an separator.
func isSeparator(token string) bool {
   separators := []string{"(", ")", "[", "]", "{", "}", ";", ","}
   return contains(separators, token)
}

"isIdentifier" checks if the token passed as a parameter is an identifier.
func isIdentifier(tk string) bool {
   return checkFormat(tk, `^[_a-zA-z][a-zA-Z0-9_]*$`)
}
"isConstant" checks if the token passed as a parameter is a constant.

func isConstant(tk string) bool {
   return isInt(tk) || isString(tk) || isChar(tk)
}
"isInt" checks if the token passed as a parameter is an integer.
func isInt(tk string) bool {
   return checkFormat(tk, `^(0|[+\-]?[1-9]\d*)$`)
}
"isString" checks if the token passed as a parameter is a string.
```

```
func isString(tk string) bool {
   return checkFormat(tk, `^".*"$`)
}
"isChar" checks if the token passed as a parameter is a char.
func isChar(tk string) bool {
   return checkFormat(tk, `^'.'$`)
}


"checkFormat" is an utility procedure which verifies if a string matches a
given format.
func checkFormat(tk string, expr string) bool {
   f, err := regexp.Compile(expr)
   if err != nil {
      log.Fatal(err)
   }

   return f.MatchString(tk)
}


"contains" is an utility procedure which verifies if a given string appears
in a list.
func contains(arr []string, str string) bool {
   for _, a := range arr {
      if a == str {
         return true
      }
   }

   return false
}
```

```
"PIF" is the exposed interface of the program internal form.

This defines the method Add, and composes two aditional interfaces,
"Outputter" and "Stringer", which specify the facts that "PIF" can be
written to a file and converted to a string, respectively.

type PIF interface {

   Add(token string, index int)

   utils.Outputter

   fmt.Stringer

}


"pif" is the structure which implements the PIF interface. It contains a
slice of elements.

type pif struct {

   elems []element

}


New is the constructor of the pif struct

func New() *pif {

   return &pif{

      elems: []element{},

   }

}


 Add creates a new element and appends it to the end of the elements slice.

func (p *pif) Add(t string, i int) {

   e := element{

      token: t,

      index: i,

   }
```

```
   p.elems = append(p.elems, e)
}

This is the implementation of the file-writing operation.
func (p pif) Output(outFile string) {
   if err := os.WriteFile(outFile, []byte(p.String()), 0666); err != nil {
      log.Fatalf("cannot write pif to file: %s", err.Error())
   }
}

The “String” method returns a string containing all the necessary data from
the pif.
func (p pif) String() string {
   s := ""
   for i, e := range p.elems {
      e.index = i
      s += e.String()
      s += "\n"
   }

   return s
}

"element” is a structure representing the elements from the pif. It contains
the token and the index.
type element struct {
   token string
   index int
```

```
}
```

The "String" method converts the element to a string containing the necessary information.

```
func (e element) String() string {
   return fmt.Sprintf("token: %s;index: %d", e.token, e.index)
}
```