

Introduction à Go – TD1: Calculatrice

Franklin "Snaipe" Mathieu

2023

Dans ce TD, nous allons implémenter un calculateur d'entiers en ligne de commande implémentant une version simplifiée de Lisp.

Par exemple, évaluer l'expression `(+ 1 2 3)` devrait afficher 6.

Étant un outil en ligne de commande, l'interface proposée de l'outil serait:

- `calc --input=<fichier>`: Ouvre le fichier et lance REPL (read-eval-print loop) dessus. Chaque ligne du fichier est évaluée, et le résultat est affiché.
- `calc`: équivalent à `calc --input=-`: l'entrée standard est utilisée à la place d'un fichier.
- `calc --eval=<expression>`: évalue une expression et affiche le résultat. Équivalent à `echo 'expression' |`

Au premier abord, nous devons écrire un interpréteur capable de prendre une string et de l'évaluer. Ceci est typiquement fait en 3 étapes: l'analyse lexicale, qui consiste à transformer le texte en entrée en une séquence de jetons; puis l'analyse syntaxique, qui consiste à transformer cette séquence de jetons en un arbre syntaxique; et finalement l'évaluation, qui évalue l'arbre syntaxique.

Fort heureusement, les dialectes de Lisp sont construits de manière à ce que l'analyse syntaxique et l'évaluation peuvent être faits en même temps sans représentation intermédiaire: en effet, la séquence de jetons est déjà structuré comme un arbre.

La partie souvent difficile, si vous n'avez jamais écrit ce genre de programmes, est l'analyse lexicale; nous allons voir comment l'approcher en Go.

1 Mise en place

1.1 CLI

Commençons par créer un dossier de projet:

```
$ go mod init example.com/calc
```

Pour ce projet, nous utiliserons <https://github.com/alecthomaskong> pour prendre en charge le CLI:

```
$ go get github.com/alecthomaskong
```

Cette bibliothèque nous permet de facilement définir des options. Créez un fichier `main.go`:

```

package main

import (
    "fmt"

    "github.com/alectho/kong"
)

type CLI struct {
    Input string `default:"-" help:"evaluate input from file"`
    Eval  string `help:"evaluate expression instead or running the REPL"`
}

func main() {
    var cli CLI
    kong.Parse(&cli)

    fmt.Printf("%#v\n", cli)
}

```

Ce programme affiche simplement les arguments qui ont été interprétés; vous pouvez tester son fonctionnement avec `go run`:

```

$ go run . -h
Usage: calc

Flags:
  -h, --help           Show context-sensitive help.
  --input="-"          evaluate input from file
  --eval=STRING        evaluate expression instead or running the REPL

$ go run . --eval='(+ 1 2 3)'
main.CLI{Input:"-", Eval:"(+ 1 2 3)"}

$ go run . --input /dev/null
main.CLI{Input:"/dev/null", Eval:""}

```

1.2 Lire un fichier

En Go, le paquet standard `os` met à disposition des abstractions pour travailler avec le système. `*os.File` est le type standard représentant un fichier.

Pour ouvrir un fichier en lecture seule:

```

f, err := os.Open("/chemin/vers/fichier")
if err != nil {
    // Gérer l'erreur
}

```

Un fichier possède une méthode `Read` pour lire du contenu:

```

data := make([]byte, 4096)
n, err := f.Read(data) // Lis au plus 4096 octets, retourne le nombre d'octets lus
if err != nil {
    // Gérer l'erreur
}

```

... et finalement, un fichier doit être fermé pour en libérer les ressources:

```
if err := f.Close(); err != nil {  
    // Gérer l'erreur; en général uniquement nécessaire quand  
    // le fichier est ouvert en écriture.  
}
```

Notez cependant que si `f.Read` retourne un erreur, et que vous décidez de retourner une erreur en retour, alors vous quitterez la fonction sans avoir appelé `f.Close()`. Pour palier à ce genre de situation, il est courant d'utiliser le mot-clé `defer`:

```
f, err := os.Open("/chemin/vers/fichier")  
if err != nil {  
    // Gérer l'erreur  
}  
defer f.Close() // sera exécuté à la sortie de la fonction
```

Ce faisant il n'est donc pas nécessaire de manuellement appeler `f.Close()` sauf si vous voulez vérifier les erreurs (lorsque le fichier est ouvert en écriture).

`os.Stdin`, `os.Stdout` et `os.Stderr`, représentant respectivement l'entrée standard, la sortie standard, et la sortie d'erreur en Go, sont des variables globales de type `*os.File`, et implémentent donc ces mêmes méthodes.

Utiliser `Read` directement n'est cependant pas, en général, la manière la plus simple de lire des fichiers. Il s'agit d'un bloc de base utilisé par d'autres abstractions implémentant des comportements plus généraux.

Pour lire le fichier ligne-par-ligne, il est plus facile de faire appel au type `bufio.Scanner`. Le package `bufio` implémente des primitives pour lire et écrire avec du "buffering".

Changeons notre `main.go` pour lire le fichier spécifié par `cli.Input` et en afficher le contenu:

```

--- a/main.go
+++ b/main.go
@@ -1,7 +1,9 @@
package main

import (
+   "bufio"
+   "fmt"
+   "os"

   "github.com/alecthomas/kong"
)
@@ -15,5 +17,27 @@ func main() {
    var cli CLI
    kong.Parse(&cli)

-   fmt.Printf("%#v\n", cli)
+   var in *os.File
+   if cli.Input == "-" {
+       in = os.Stdin
+   } else {
+       f, err := os.Open(cli.Input)
+       if err != nil {
+           fmt.Fprintln(os.Stderr, err)
+           os.Exit(1)
+       }
+       defer f.Close() // sera fermé en sortant du main()
+       in = f
+   }
+
+   scanner := bufio.NewScanner(in)
+
+   for scanner.Scan() { // Continue de scanner jusqu'à EOF
+       line := scanner.Text()
+       fmt.Println(line)
+   }
+   if err := scanner.Err(); err != nil { // S'il y a une erreur, l'afficher
+       fmt.Fprintln(os.Stderr, err)
+       os.Exit(1)
+   }
}

```

Vous pouvez le voir en action avec `go run`:

```

$ echo -e "foo\nbar\nbaz" | go run .
foo
bar
baz

```

2 Analyse lexicale

Passons aux choses sérieuses: la transformation de texte en lexèmes. Dans une expression telle que `(+ a 2)`, il convient de produire les jetons `"(", "+", "a", "2", ")"`. Si nous classifions ces jetons en différents types, nous obtenons 5 types généraux: les parenthèses gauche et droite, les symboles (`+` et `a`), et les nombres.

`<"(", LPAREN>, <"+", SYMBOL>, <"1", NUMBER>, <"2", NUMBER>, <">", RPAREN>.`

Représentons ces tokens dans notre code; dans un fichier nommé token.go:

```
package main

type TokenType int

const (
    TokenEOF TokenType = iota
    TokenSymbol
    TokenNumber
    TokenLParen
    TokenRParen
)

type Token struct {
    Type TokenType
    Val string
}
```

Il ne reste qu'à écrire notre analyseur lexical. Pour ce faire, définissons un type `Lexer` dans un fichier `lex.go`:

```

package main

type Lexer struct {
    in string // le texte en entrée

    start int // index du début du token en cours
    index int // index du curseur

    // TODO - ajouter d'autres champs si nécessaire

    tokens []Token // les tokens produits lors de l'analyse lexicale
}

func MakeLexer(in string) Lexer {
    return Lexer{in: in}
}

// next consomme une rune, avançant le curseur vers l'avant, et retourne la rune consommée.
// next retourne 0 s'il n'y a plus de rune de disponible (fin du texte).
func (l *Lexer) next() (r rune) {
    return 0 // TODO
}

// back reviens à la rune précédente, annulant l'effet d'un next()
func (l *Lexer) back() {
    // TODO
}

// peek retourne la rune qui suit sans la consommer (ou 0 en fin de texte)
func (l *Lexer) peek() rune {
    return 0 // TODO
}

// emit ajoute dans l.tokens un nouveau Token de type "typ", et de valeur l.in[l.start:l.index], et
// met à jour l.start pour démarrer un nouveau token
func (l *Lexer) emit(typ TokenType) {
    // TODO
}

// discard avance l.start pour ne pas mettre le caractère en cours dans le token produit par emit.
func (l *Lexer) discard() {
    // TODO
}

```

Il n'est pas nécessaire de les implémenter pour le moment. Pour comprendre à quoi nous servent ces utilitaires, nous allons écrire une machine à état simple. En Go, il est assez facile d'en créer avec un type fonction:

```

type stateFunc func() stateFunc

```

En d'autres termes, `stateFunc` est une fonction qui retourne une `stateFunc` de l'état suivant. Pour exécuter cette machine à état, il suffit de continuellement évaluer cette fonction:

```
func (l *Lexer) Lex() []Token {
    state := l.lexMain
    for state != nil {
        state = state()
    }
    return l.tokens
}
```

Ici, l'état initial est la méthode `lexMain`, et tant qu'il y a un état suivant, on appelle `state` continuellement, en réassignant son résultat dans elle-même.

La fonction `lexMain` est responsable de l'analyse lexicale du texte de manière générale; écrivons-la:

```
func (l *Lexer) lexMain() stateFunc {
    for {
        n := l.next()
        switch {
        case n == 0: // EOF
            return nil // On arrête la machine à état
        case n == '(':
            l.emit(TokenLParen)
        case n == ')':
            l.emit(TokenRParen)
        case n == '-':
            if strings.IndexRune("0123456789", l.peek()) == -1 {
                // n est un tiret qui n'est pas suivi d'un chiffre; c'est le début d
                return l.lexSymbol
            }
            fallthrough
        case strings.IndexRune("0123456789", n) != -1:
            return l.lexNumber
        case unicode.IsSpace(n):
            l.discard() // On exclue les espaces de nos jetons
        default:
            return l.lexSymbol
        }
    }
}
```

Le but de `lexMain` est de reconnaître le début d'un jeton en fonction de certains critères et de déléguer le lexing plus spécialisé de certains jetons à d'autres fonctions d'état. Par exemple, si le caractère en cours est '(', nous pouvons directement émettre le jeton `LPAREN`, tandis que si le caractère en cours est '4', alors nous avons le début d'un nombre, et `lexMain` retourne `l.lexNumber`, qui est spécialisé dans le lexing d'un nombre. Écrivons `lexNumber` et `lexSymbol`:

```

func (l *Lexer) lexSymbol() stateFunc {
    for {
        r := l.next()
        if r == 0 || unicode.IsSpace(r) {
            break
        }
    }
    l.back()
    l.emit(TokenSymbol)
    return l.lexMain
}

func (l *Lexer) lexNumber() stateFunc {
    for {
        r := l.next()
        if r == 0 || strings.IndexRune("0123456789", r) == -1 {
            break
        }
    }
    l.back()
    l.emit(TokenNumber)
    return l.lexMain
}

```

`lexSymbol` et `lexNumber` font deux choses très similaires: elles consomment l'entrée caractère par caractère jusqu'à ce que le caractère trouvé ne remplissent pas la définition d'un symbole (ou nombre), reviennent un caractère en arrière pour ne pas le compter dans le jeton, puis finalement émettent le jeton avant de revenir sur `l.lexMain`.

Une fois fini, il est temps d'écrire quelques tests; créez un fichier `lex_test.go`, et ajoutez quelques cas de tests. Par exemple, `"(+ 1 2)"` devrait causer `Lex()` à retourner la slice suivante:

```

expected := []Token{
    {Type: TokenLParen, Val: "("},
    {Type: TokenSymbol, Val: "+"},
    {Type: TokenNumber, Val: "1"},
    {Type: TokenNumber, Val: "2"},
    {Type: TokenRParen, Val: ")"},
}

```

Ces tests devraient tous rater; en effet, aucune des fonctions utilitaires de `Lexer` n'ont été implémentées, mais vous avez maintenant un environnement correct pour implémenter ces méthodes et en valider le comportement via `go test`.

Modifions aussi `main.go` pour utiliser notre `Lexer`:


```

--- a/main.go
+++ b/main.go
@@ -32,9 +32,17 @@ func main() {

    scanner := bufio.NewScanner(in)

-   for scanner.Scan() { // Continue de scanner jusqu'à EOF
+   for {
+       if cli.Input == "-" {
+           fmt.Fprint(os.Stderr, "\x1B[94;1m \x1B[0m ")
+       }
+       if !scanner.Scan() {
+           break
+       }
        line := scanner.Text()
-       fmt.Println(line)
+       lex := MakeLexer(line)
+       fmt.Println(lex.Lex())
    }
    if err := scanner.Err(); err != nil { // S'il y a une erreur, l'afficher
        fmt.Fprintln(os.Stderr, err)
    }
}

```

Profitez-en pour tester votre REPL:

```

$ go run .
1 2 3
[{2 1} {2 2} {2 3}]
^D

```

3 Évaluation

Nous avons maintenant une séquence de jetons à évaluer, il ne nous reste plus qu'à définir une fonction pour l'évaluer. Créez un fichier eval.go:

```

package main

func Eval(tokens []Token) ([]int, error) {
    return nil, nil // TODO
}

```

Eval consomme des jetons du paramètre `tokens` et retourne une slice des éléments évalués, ou une erreur si besoin.

Par exemple, si la séquence de jetons donnée à Eval correspond à `1 2 3`, Eval devrait retourner `[]int{1, 2, 3}`. Pour `(+ 1 2 (+ 3 4))`, 42, Eval devrait retourner `[]int{10, 42}` (i.e. la somme `1+2+(3+4)`, suivi du nombre 42). Finalement, pour `(foo 1)`, eval devrait retourner une erreur disant que "foo" n'est pas une opération définie.

Vous pouvez créer des erreurs via l'intermédiaire de `fmt.Errorf`. La fonction se comporte comme `fmt.Printf` et retourne une erreur formatée avec des arguments:

```
package main

func Eval(tokens []Token) ([]int, error) {
    return nil, fmt.Errorf("bad input %v", tokens)
}
```

Dans l'exemple plus haut, appeler Eval avec 1 2 3 retournerait une erreur avec comme message bad input [1 2 3].

C'est à vous de jouer! Implémentez les opérateurs suivants:

- + - somme des arguments
- * - produit des arguments
- - - Si un seul argument, retourne son négatif. Sinon, retourne argument 1 moins le reste des arguments; par exemple (- 1 2 3) retourne 1 - 2 - 3.
- - / - Retourne argument 1 moins divisé par le reste des arguments; par exemple (- 1 2 3) retourne 1 / (2 * 3).

Pensez à changer votre fonction main pour utiliser Eval et afficher chaque résultat sur une ligne à part, ou l'erreur le cas échéant.

Bonus 1: modifiez Eval pour que la fonction prenne une `map[string]int` de variables à substituer dans l'expression, et ajoutez une option `-arg nom=valeur` dans le CLI. Par exemple `calc --eval='(+ a b)' --arg a=1 --arg b=2` devrait afficher 3.