PEGで構文解析をする

発表者: 次郎 (@jiro_saburomaru)

あるツールをリファクタするときにPEGが便利だった話をします。

自己紹介

Key	Value
名前	次郎
Twitter	@jiro_saburomaru
職業	SRE

目次

PEG (Parsing Expression Grammar)

PEGはWikipediaによると以下のように説明されています。(注1)

> Parsing Expression Grammar (PEG) は、分析的形式文法の一種であり、形式言語をその言語に含まれる文字列を認識するための一連の規則を使って表したものである。

ざっくり要約すると「形式言語を解析するための文法」です。

PEGにはパーサジェネレータが多数存在し(注2)、PEGの文法を食わせることで、その文法を解釈できるパーサを自動生成できます。

今回のLTではGo言語用のPEGパーサジェネレータの pointlander/peg (注3)を使います。

サンプルとしてPEGの文法に則って、以下の文法を自作してみました。
これは簡易な ini ファイル風の文法です。

```
root      <- pair+

pair      <- space key space '=' space value space delimiter

key       <- [a-zA-Z] [-_a-zA-Z0-9]*
value     <- atom

atom      <- bool / int / string
string    <- '"' ('\\' '"' / [^"])* '"'
int       <- '0' / [1-9] [0-9]*
bool      <- 'true' / 'false'

space     <- (' ' / '  ' / '\t')*
delimiter <- '\n' / ';'
```

前述の文法では、以下のテキストを解釈できます。

```
name = "test_app"
port = 1234
debug = true
```

ただし、前述の文法だけでは、構文を解析できるだけで値を取り出せません。

そこで、今回使う pointlander/peg の独自の構文を使うことで値を取り出してみます。

pointlander/peg 独自の構文を付け足したPEG文法は以下のとおりです。

```
package main

type Parser Peg {
  ParserFunc
}

root      <- pair+

pair      <- space key space '=' space value space delimiter

key       <- <[a-zA-Z] [-_a-zA-Z0-9]*>    { p.pushKey(text) }
value     <- atom

atom      <- bool / int / string
string    <- '"' <('\\' '"' / [^"])*> '"' { p.pushString(text) }
int       <- <'0' / [1-9] [0-9]*>         { p.pushInt(text) }
bool      <- <'true' / 'false'>           { p.pushBool(text) }

space     <- (' ' / '  ' / '\t')*
delimiter <- '\n' / ';'
```

これで構文解析と、値の取り出しをまとめてできるようになりました。

つまりは、字句解析と構文解析を一緒にやっています。

次に、実際に設定ファイルを読み込んで、値が取り出せるか確認してみます。

簡単な確認用のプログラムを実装してみました。
パッケージの import 部分はスライドに入り切らないので省略しています。

```
func main() {
        b, err := os.ReadFile("sample.conf")
        if err != nil {
                panic(err)
        }

        pf := ParserFunc{
                data: make(map[string]interface{}),
        }
        p := &Parser{
                Buffer: string(b),
                ParserFunc: pf,
        }
        if err := p.Init(); err != nil {
                panic(err)
        }
        if err := p.Parse(); err != nil {
                panic(err)
        }
        p.Execute()
        for k, v := range p.ParserFunc.data {
                fmt.Printf("key = %s, value = %v, type = %s\n", k, v, reflect.TypeOf(v))
        }
}
```

peg コマンドに文法を定義したpegファイルを渡すと、Goのソースコードが生成されます。ソースコード生成後にビルドして実行します。

実行結果は以下のようになります。期待通り、KeyとValueと型情報が取り出せています。

```
〉peg grammer.peg

〉go build -o app

〉cat sample.conf
name = "test_app"
port = 1234
debug = true

〉./app
key = port, value = 1234, type = int
key = debug, value = true, type = bool
key = name, value = test_app, type = string
```

ここまでで使ったサンプルコードはすべて以下のGistにまとめています。

気になった方は参考にしてみてください。

https://gist.github.com/jiro4989/668bd841e484eda7959bc027fb891da0

PEGの嬉しさ

1. パーサを自動生成できる

PEGで文法を書くだけで、パーサを自動生成できるのがとても嬉しいです。

形式がある程度定まったテキストを解析する場合は、毎回似たような構文解析プログラムを実装するのですが、その手間を省けるのは大いにメリットです。

2. 「どんなテキストを解析できるか」が分かりやすい

解析できる文法がPEGの文法として明文化されるため、PEGの文法から逆に「どういうテキストを解析できるか」を理解できます。つまり、プログラムの理解を助ける効果が期待できます。

PEGを使ってみる

PEG が構文解析に便利なことが分かったので

シェル芸人なら日常的に目にしているであろう、**アレ** の解析をやってみた。

そう、 **ANSIエスケープシーケンス** です。

textimg というコマンドのANSIエスケープシーケンスの処理部分を PEG を使って書き直した話をします。

（ここから本題）

textimg とANSIエスケープシーケンスに関する話は、2019年8月の第43回シェル芸勉強会のLTで発表しました。

気になる方は以下の資料を見てください。

「ANSIエスケープシーケンスで遊ぶ - /home/jiro4989」
https://scrapbox.io/jiro4989/ANSIエスケープシーケンスで遊ぶ

ANSIエスケープシーケンスの制御文字とエスケープシーケンスは man に書かれています。(注4)

コマンドで確認する場合は `man console_codes` を実行すれば確認できます。

グラフィック制御を行うエスケープシーケンスは **SGR (Select Graphic Rendition)** と呼びます。(注4)

textimg はこの SGR の解析処理だけを実装しています。

SGR シーケンスは `ESC [ parameters m` と定義されています。(注4)

ESCは `0x1b` を表します。

この parameters はセミコロン区切りで複数設定可能な可変長の値です。

よって、もう少し細かく SGR シーケンスを書くと以下のようになります。

```
ESC [ parameter (;parameter)* m
```

これをPEGで表現してみます。

最初に基本形です。parameterの値は 30~37、40~47、90~97、100~107 を受け付けます。この時、1の位が8の場合は書き方がさらに変化するのですが、今回は割愛します。

まず prefix color suffix という3部分を定義します。
これで `ESC[31m` (前景色が赤) のSGRを解釈できるようになりました。

```
colors <-
  prefix color suffix

prefix <-
  '\e' '['

color <-
  ([349] / '10') [0-7]

suffix <- 'm'
```

次に、セミコロン区切りで複数のparameterを指定できるようにします。
区切り文字 `delimiter` を定義し、0個以上の parameter を扱えるようにしました。

これで `ESC[31;42m` (前景色が赤、背景色が緑) のSGRを解釈できるようになりました。

```
 colors <-
-  prefix color suffix
+  prefix color (delimiter color)* suffix

 prefix <-
   '\e' '['

 color <-
   ([349] / '10') [0-7]

 suffix    <- 'm'
+delimiter <- ';'
```

これだけだと SGR しか解釈できないため、テキストも解釈できるようにします。
ESC 以外のすべての文字を `text` として定義しました。

これで最低限の SGR が含まれるテキストを解釈できる文法が整いました。

```
+root <- (colors / text)*

 colors <-
   prefix color (delimiter color)* suffix

 prefix <-
   '\e' '['

 color <-
   ([349] / '10') [0-7]

+text <- [^\e]+

 suffix    <- 'm'
 delimiter <- ';'
```

後は pointlander/peg 独自の構文を付け足して動作確認できるようにしました。
全体のコードは以下にまとめています。

https://gist.github.com/jiro4989/8c52e3264cbe98482565d7f9783e4f80

以下のテキストファイルを解析させてみます。

```
 [31;42mhello world
```

実行結果は以下のとおりです。それぞれ値が取り出せていることがわかります。

```
$ ./configfile
value = {0 31 }
value = {0 42 }
value = {1 0 hello world}
```

先程の定義だけでは、1の位が 8 の時の 256 色系や RGB 系を解釈できません。

また、SGR以外のエスケープシーケンスが現れた場合や、不完全なエスケープシーケンスが出現した場合も解釈できません。

よって、本当なら無視する文字列も定義するべきでしたが、今回は割愛しました。

こんな具合に文法を徐々に拡張していって、最終的な文法は以下のようになりました。

https://github.com/jiro4989/textimg/blob/master/parser/grammer.peg

行数だけで見ると70行程度しかないため、記述量は非常に少なく済んだと言えます。

感想

PEGを書くだけでパーサーが生成されるので実装コストがとても安いことが分かりました。

しかしながら、生成されるソースコードが非常に巨大なことも分かりました。

```
〉wc -l parser/grammer.peg.go
1300 parser/grammer.peg.go
```

前述の文法を見れば分かるように、SGR単体の文法はかなり単純です。
よって SGR のパーサを作るには、PEGはややオーバースペックだったかもしれません。

逆に、複雑な構文や、構文が今後複雑になる可能性があるなら PEG でパーサーを自動生成するのは便利そうに思います。

まとめ

以下の話をしました。

1. PEG を使うと可読性が高く、複雑な構文解析を低コストで実装できる
2. PEG をANSIエスケープシーケンスの SGR の構文解析に使ってみたら良かった
3. PEG は簡単な構文の解析にはややオーバースペックそうだけれど、構文が今後どう変化するか見えない場合には強力そう

以上

参考文献

以下は引用した資料のリンクです。

- 注1: Wikipedia. 「Parsing Expression Grammar」. https://ja.wikipedia.org/wiki/Parsing_Expression_Grammar, (参照 2022-04-28)
- 注2: Wikipedia. 「Parsing Expression Grammar」. https://ja.wikipedia.org/wiki/Parsing_Expression_Grammar#PEGパーサ生成器ほか, (参照 2022-04-28)
- 注3: GitHub. 「pointlander/peg」. https://github.com/pointlander/peg, (参照 2022-04-28)
- 注4: Linux manual page. 「console_codes(4) — Linux manual page」. https://man7.org/linux/man-pages/man4/console_codes.4.html, (参照 2022-04-28)

以下は PEG の書き方を理解するために参考にしたソースコードです。

- https://github.com/rwxrob/dtime/blob/main/grammar.peg
- https://github.com/tj/go-naturaldate/blob/master/grammar.peg