Parsing Expression Grammars

PEGs allow you to parse data in programming languages.

Core operations:

Choice: (+ “a” “b” “c”)
Match “a” or “b” or “c”, once

Sequence: (* "1" "2" "3")
Match “1”, then “2”, then "3"

Range: (range "09" "az")
Match one of "0"-"9" or "a"-"z"

(+ (range "09") ("az"))
Match: "2", "s"
No Match: "Q", ","

We’re using Janet’s PEG syntax in this zine

Repeating matches:

Any: (any PATT)
0 or more repetitions

Some: (some PATT)
1 or more repetitions of PATT

Between: (between L H patt)
L and H repetitions of patt

(some (+ "a" "c" "e"))
Match: "a", "cce", "ace", "cea"
No Match: "", "b", "ACE"

PEGs can either be a single pattern, or a dict of named patterns, which start at :main

---

Named patterns:

```
'{
:digit (range "09")

 :2digit (between 1 2 :digit)

 :main
   (* :2digit "/" :2digit "/"
     (between 2 4 :digit))
 }
```

Matches: "12/34/1999"
No Match: "12/1/0"

PEGs with named patterns can recurse

Recusive patterns:

```
'{
:digits (some (range "09"))
:expr (+
       (* "[" :expr "]" )
       :digits
)
:main
(some (* :expr (any " ")))
}
```

Matches: "[12[34]] [55]7"
No match: "[123"

PEGs can do more than just match strings, they can also capture output

---

capture: (capture PATT)
Captures all text matched by PATT, after captures in PATT

position: (position PATT)
Captures the position at the start of PATT

cmt: (cmt PATT FUNC)
If PATT matches, calls FUNC with PATT's captures

Capture example

```
~{
 :digits (some (range "09"))
 :x (* "x:" (capture :digits))
 :y (* "y:" (capture :digits))
 :main (*
        (cmt :x ,scan-number) " "
        (cmt :y ,scan-number)
 )}
```

Matches:"x:12 y:23"→[12 23]

See if you can write PEGs that produces the shown matches

Matching challenges

123-456-9870 → ["1" "4" "9"]

12/13/2024 → ["2" "3" "4"]

(add 1 2)→[["add" "1" "2"]]

---

https://janet-lang.org/