

Perl6 の正規表現 (+) で Lisp モドキを実装

青木大祐

筑波大学 情報科学類

2012 年 1 月 29 日

この LT の概要

- Perl6 Grammar の紹介
- Grammar を使って遊ぶ

Grammar とは？

- 一般的には解析表現文法
(*Parsing Expression Grammar*:**PEG**) と呼ぶらしい

Grammar とは？

- 一般的には解析表現文法
(*Parsing Expression Grammar*:**PEG**) と呼ぶらしい
- 小さい名前付きルールから、順に組み上げて行く
構造的な表現

Grammar とは？

- 一般的には解析表現文法
(*Parsing Expression Grammar*:**PEG**) と呼ぶらしい
- 小さい名前付きルールから、順に組み上げて行く
構造的な表現
- 正規表現と併用する事でさらに強力な記述が可能

Grammar で何ができるのか

普通の正規表現と違って...

Grammar で何ができるのか

普通の正規表現と違って...

- 構造的な文字列を表現しやすい

Grammar で何ができるのか

普通の正規表現と違って...

- 構造的な文字列を表現しやすい
- 再帰的な構造を記述できる

Grammar で何が出来なのか

普通の正規表現と違って...

- 構造的な文字列を表現しやすい
- 再帰的な構造を記述できる
 - **普通の**正規表現では再帰構造を記述できない

Grammar で何ができるのか

普通の正規表現と違って...

- 構造的な文字列を表現しやすい
- 再帰的な構造を記述できる
 - 普通の正規表現では再帰構造を記述できない
 - PCRE(Perl Compatible Regular Expressions) 系の正規表現 (?) エンジンなら可能

Grammar で何ができるのか

普通の正規表現と違って...

- 構造的な文字列を表現しやすい
- 再帰的な構造を記述できる
 - 普通の正規表現では再帰構造を記述できない
 - PCRE(Perl Compatible Regular Expressions) 系の正規表現 (?) エンジンなら可能

パーサとか書くのが楽になるかも

イメージとしてはこんな感じ

```
1 grammar Lisp {  
2     token num { \d+ }  
3     token str { '\"'\w+'\"' }  
4     token literal { <num>|<str> }  
5     token nil { 'nil'|'(' <.ws> ')' }  
6     token symbol { \w+ }  
7     token atom { <literal>|<nil>|<symbol> }  
8  
9     rule sexpr { <atom> | '\''? '(' <sexpr>+? % <.ws> ')' }  
10    rule TOP { ^^ <sexpr> $$ }  
11 }
```

イメージとしてはこんな感じ

```
1 grammar Lisp {  
2     token num { \d+ }  
3     token str { '"' '\w+' '"' }  
4     token literal { <num> | <str> }  
5     token nil { 'nil' | '(' <.ws> ')' }  
6     token symbol { \w+ }  
7     token atom { <literal> | <nil> | <symbol> }  
8  
9     rule sexpr { <atom> | '\''? '(' <sexpr>+ ? % <.ws> ')' }  
10    rule TOP { ^^ <sexpr> $$ }  
11 }
```

- 「リテラル」は「数字」と「文字列」

イメージとしてはこんな感じ

```
1 grammar Lisp {  
2     token num { \d+ }  
3     token str { '\"'\w+'\"' }  
4     token literal { <num>|<str> }  
5     token nil { 'nil'|'(' <.ws> ')' }  
6     token symbol { \w+ }  
7     token atom { <literal>|<nil>|<symbol> }  
8  
9     rule sexpr { <atom> | '\''?' '(' <sexpr>+? % <.ws> ')' }  
10    rule TOP { ^^ <sexpr> $$ }  
11 }
```

- 「リテラル」は「数字」と「文字列」
- 「アトム」は「リテラル」と「nil」と「シンボル」

イメージとしてはこんな感じ

```
1 grammar Lisp {  
2     token num { \d+ }  
3     token str { '"' '\w+' '"' }  
4     token literal { <num> | <str> }  
5     token nil { 'nil' | '(' <.ws> ')' }  
6     token symbol { \w+ }  
7     token atom { <literal> | <nil> | <symbol> }  
8  
9     rule sexpr { <atom> | '\'' '?' '(' <sexpr>+ ? % <.ws> ')' }  
10    rule TOP { ^^ <sexpr> $$ }  
11 }
```

- 「リテラル」は「数字」と「文字列」
- 「アトム」は「リテラル」と「nil」と「シンボル」
- 「S式 (S-Expressions)」は「アトム」と「S式のリスト」

イメージとしてはこんな感じ

```
1 grammar Lisp {
2     token num { \d+ }
3     token str { '"'\w+'"' }
4     token literal { <num>|<str> }
5     token nil { 'nil'|'(' <.ws> ')' }
6     token symbol { \w+ }
7     token atom { <literal>|<nil>|<symbol> }
8
9     rule sexpr { <atom> | '\''?' '(' <sexpr>+? % <.ws> ')' }
10    rule TOP { ^^ <sexpr> $$ }
11 }
```

- 「リテラル」は「数字」と「文字列」
- 「アトム」は「リテラル」と「nil」と「シンボル」
- 「S式 (S-Expressions)」は「アトム」と「S式のリスト」
 - 再帰構造

実際にパースしてみる [1]

使い方は簡単：

parse メソッドに対象の文字列を投げるだけ
受理されれば Match オブジェクトが返ってくる

```
1 my $str = "(equal (car \"(1 2)) 2)";  
2 my $m = Lisp.parse($str);
```

実際にパースしてみる [2]

```
1 => <(equal (car '(1 2)) 2)>
2  sexpr => <(equal (car '(1 2)) 2)>
3    sexpr => <equal >
4      atom => <equal>
5        symbol => <equal>
6  sexpr => <(car '(1 2)) >
7    sexpr => <car >
8      atom => <car>
9        symbol => <car>
10   sexpr => <'(1 2)>
11     sexpr => <1 >
12       atom => <1>
13         literal => <1>
14         num => <1>
15   sexpr => <2>
16     atom => <2>
17       literal => <2>
18       num => <2>
19   sexpr => <2>
20     atom => <2>
21       literal => <2>
22       num => <2>
```

更に一步

- パースするだけじゃ勿体無い

更に一步

- パースするだけじゃ勿体無い
- 手動で Match オブジェクトを走査するのは面倒

更に一步

- パースするだけじゃ勿体無い
- 手動で Match オブジェクトを走査するのは面倒
 - トークンに刻むだけなら正規表現でも (ry

更に一步

- パースするだけじゃ勿体無い
- 手動で Match オブジェクトを走査するのは面倒
 - トークンに刻むだけなら正規表現でも (ry

Action を使ってスマートに処理

Action と抽象構文木 [1]

parse メソッドは actions オプションを指定できる

```
1 my $m = Lisp.parse($str, actions => SomeAction);
```

Action と抽象構文木 [1]

parse メソッドは actions オプションを指定できる

```
1 my $m = Lisp.parse($str, actions => SomeAction);
```

- パースする際と同様に深さ優先探索でマッチングを試みる

Action と抽象構文木 [1]

parse メソッドは actions オプションを指定できる

```
1 my $m = Lisp.parse($str, actions => SomeAction);
```

- パースする際と同様に深さ優先探索でマッチングを試みる
- トークンへのマッチが成功するたびに、actions に指定したクラスの同名メソッドが呼ばれる

Action と抽象構文木 [1]

parse メソッドは actions オプションを指定できる

```
1 my $m = Lisp.parse($str, actions => SomeAction);
```

- パースする際と同様に深さ優先探索でマッチングを試みる
- トークンへのマッチが成功するたびに、actions に指定したクラスの同名メソッドが呼ばれる
 - action は深さ優先探索の戻り掛けに実行される

Action と抽象構文木 [1]

parse メソッドは actions オプションを指定できる

```
1 my $m = Lisp.parse($str, actions => SomeAction);
```

- パースする際と同様に深さ優先探索でマッチングを試みる
- トークンへのマッチが成功するたびに、actions に指定したクラスの同名メソッドが呼ばれる
 - action は深さ優先探索の戻り掛けに実行される
 - 最終的にマッチングに失敗するとしても、成功した所までは action が実行される点に注意

Action と抽象構文木 [2]

- 引数として、マッチに成功したトークンの Match オブジェクトが渡される

Action と抽象構文木 [2]

- 引数として、マッチに成功したトークンの Match オブジェクトが渡される
- Match オブジェクトを使って好きな処理をした後、抽象構文木 (*Abstract Syntax Tree*: **AST**) を作る

Action と抽象構文木 [2]

- 引数として、マッチに成功したトークンの Match オブジェクトが渡される
- Match オブジェクトを使って好きな処理をした後、抽象構文木 (*Abstract Syntax Tree*: **AST**) を作る
 - *return* で値を返すのと同じように、*make* 文を使う

Action と抽象構文木 [2]

- 引数として、マッチに成功したトークンの Match オブジェクトが渡される
- Match オブジェクトを使って好きな処理をした後、抽象構文木 (*Abstract Syntax Tree*: **AST**) を作る
 - *return* で値を返すのと同じように、*make* 文を使う
- 作られた抽象構文木は、それを含む (つまり上の階層の) Match オブジェクトから参照できる (`$m.ast`)

Action と抽象構文木 [2]

- 引数として、マッチに成功したトークンの Match オブジェクトが渡される
- Match オブジェクトを使って好きな処理をした後、抽象構文木 (*Abstract Syntax Tree*: **AST**) を作る
 - *return* で値を返すのと同じように、*make* 文を使う
- 作られた抽象構文木は、それを含む (つまり上の階層の) Match オブジェクトから参照できる (`$m.ast`)

下位要素の抽象構文木を参照しながら、上 (TOP) に向かって抽象構文木を組み立てて行く

Grammar で遊んでみる

目標:

- `(equal (car '(1 2)) 2)` を評価する

Grammar で遊んでみる

目標:

- `(equal (car '(1 2)) 2)` を評価する
- 関数は `equal` と `car` と `quote` が評価できれば良い

Grammar で遊んでみる

目標:

- `(equal (car '(1 2)) 2)` を評価する
- 関数は `equal` と `car` と `quote` が評価できれば良い
 - `equal`: 2 つの引数が等しいなら `t`、そうでないなら `nil`

Grammar で遊んでみる

目標:

- `(equal (car '(1 2)) 2)` を評価する
- 関数は `equal` と `car` と `quote` が評価できれば良い
 - `equal`: 2 つの引数が等しいなら `t`、そうでないなら `nil`
 - `car`: 引数に受け取ったリストの先頭の要素を返す

Grammar で遊んでみる

目標:

- `(equal (car '(1 2)) 2)` を評価する
- 関数は `equal` と `car` と `quote` が評価できれば良い
 - `equal`: 2 つの引数が等しいなら `t`、そうでないなら `nil`
 - `car`: 引数に受け取ったリストの先頭の要素を返す
 - `quote`: 引数に受け取った S 式をそのまま返す

Grammar で遊んでみる

目標:

- `(equal (car '(1 2)) 2)` を評価する
- 関数は `equal` と `car` と `quote` が評価できれば良い
 - `equal`: 2 つの引数が等しいなら `t`、そうでないなら `nil`
 - `car`: 引数に受け取ったリストの先頭の要素を返す
 - `quote`: 引数に受け取った S 式をそのまま返す
- 正しく評価できていれば `nil` が返るはず

実際の Action のコード

```
1 class Evaluate {  
2   method num ($/) { make $/.Str }  
3   method str ($/) { make $/.Str }  
4   method literal ($/) { make $<num>.?ast // $<str>.ast }  
5   method nil ($/) { make 'nil' }  
6   method symbol ($/) { make $/.Str }  
7   method atom ($/) {  
8     make $<literal>.?ast // $<nil>.?ast // $<symbol>.ast  
9   }  
10  ...
```

実際の Action のコード

```
1 class Evaluate {  
2   method num ($/) { make $/.Str }  
3   method str ($/) { make $/.Str }  
4   method literal ($/) { make $<num>.?ast // $<str>.ast }  
5   method nil ($/) { make 'nil' }  
6   method symbol ($/) { make $/.Str }  
7   method atom ($/) {  
8     make $<literal>.?ast // $<nil>.?ast // $<symbol>.ast  
9   }  
10  ...
```

num と str の ast

文字列そのまま

実際の Action のコード

```
1 class Evaluate {  
2   method num ($/) { make $/.Str }  
3   method str ($/) { make $/.Str }  
4   method literal ($/) { make $<num>.?ast // $<str>.ast }  
5   method nil ($/) { make 'nil' }  
6   method symbol ($/) { make $/.Str }  
7   method atom ($/) {  
8     make $<literal>.?ast // $<nil>.?ast // $<symbol>.ast  
9   }  
10  ...
```

num と str の ast

文字列そのまま

literal の ast

num か str の ast

実際の Action のコード

```
1 class Evaluate {  
2   method num ($/) { make $/.Str }  
3   method str ($/) { make $/.Str }  
4   method literal ($/) { make $<num>.?ast // $<str>.ast }  
5   method nil ($/) { make 'nil' }  
6   method symbol ($/) { make $/.Str }  
7   method atom ($/) {  
8     make $<literal>.?ast // $<nil>.?ast // $<symbol>.ast  
9   }  
10  ...
```

num と str の ast

文字列そのまま

literal の ast

num か str の ast

nil の ast

'nil'

実際の Action のコード

```
1 class Evaluate {  
2   method num ($/) { make $/.Str }  
3   method str ($/) { make $/.Str }  
4   method literal ($/) { make $<num>?.ast // $<str>.ast }  
5   method nil ($/) { make 'nil' }  
6   method symbol ($/) { make $/.Str }  
7   method atom ($/) {  
8     make $<literal>?.ast // $<nil>?.ast // $<symbol>.ast  
9   }  
10  ...
```

num と str の ast

文字列そのまま

literal の ast

num か str の ast

nil の ast

'nil'

symbol の ast

文字列そのまま

実際の Action のコード

```
1 class Evaluate {  
2   method num ($/) { make $/.Str }  
3   method str ($/) { make $/.Str }  
4   method literal ($/) { make $<num>?.ast // $<str>.ast }  
5   method nil ($/) { make 'nil' }  
6   method symbol ($/) { make $/.Str }  
7   method atom ($/) {  
8     make $<literal>?.ast // $<nil>?.ast // $<symbol>.ast  
9   }  
10  ...
```

num と str の ast

文字列そのまま

literal の ast

num か str の ast

nil の ast

'nil'

symbol の ast

文字列そのまま

atom の ast

中身の ast

S式の解釈 [1]

```
1 method sexpr ($/) {  
2   # quoted expressions  
3   if $/.Str.match(/^` \'\/) {  
4     make $/.Str.substr(1);  
5   }  
6   # atoms  
7   elsif $<atom> {  
8     make $<atom>.ast;  
9   }  
10  # other expressions  
11  else {  
12    given $<sexpr>[0].ast {  
13      when "car" {  
14        make $<sexpr>[1]<sexpr>[0].ast;  
15      }  
16      when "equal" {  
17        make $<sexpr>[1].ast eq $<sexpr>[2].ast ?? "t" !! "nil";  
18      }  
19    }  
20  }  
21 }
```

S 式の解釈 [2]

```
2  # quoted expressions
3  if $/.Str.match(/^^ \'/) {
4      make $/.Str.substr(1);
5  }
```

S 式の解釈 [2]

```
2  # quoted expressions
3  if $/.Str.match(/^^ \'/) {
4      make $/.Str.substr(1);
5  }
```

- 先頭がクォートから始まる S 式 (quote 関数)

S 式の解釈 [2]

```
2  # quoted expressions
3  if $/.Str.match(/^^ \'/) {
4      make $/.Str.substr(1);
5  }
```

- 先頭がクォートから始まる S 式 (quote 関数)
 - 引数に受け取った S 式をそのまま返す

S 式の解釈 [2]

```
2  # quoted expressions
3  if $/.Str.match(/^^ \'/) {
4      make $/.Str.substr(1);
5  }
```

- 先頭がクォートから始まる S 式 (quote 関数)
 - 引数に受け取った S 式をそのまま返す
 - クォートを外して make

S 式の解釈 [2]

```
2  # quoted expressions
3  if $/.Str.match(/^^ \'/) {
4      make $/.Str.substr(1);
5  }
```

- 先頭がクォートから始まる S 式 (quote 関数)
 - 引数に受け取った S 式をそのまま返す
 - クォートを外して make

```
6  # atoms
7  elsif $<atom> {
8      make $<atom>.ast;
9  }
```

S 式の解釈 [2]

```
2  # quoted expressions
3  if $/.Str.match(/^^ \'/) {
4      make $/.Str.substr(1);
5  }
```

- 先頭がクオートから始まる S 式 (quote 関数)
 - 引数に受け取った S 式をそのまま返す
 - クォートを外して make

```
6  # atoms
7  elsif $<atom> {
8      make $<atom>.ast;
9  }
```

- atom から成る S 式

S 式の解釈 [2]

```
2  # quoted expressions
3  if $/.Str.match(/^` \'/) {
4      make $/.Str.substr(1);
5  }
```

- 先頭がクオートから始まる S 式 (quote 関数)
 - 引数に受け取った S 式をそのまま返す
 - クオートを外して make

```
6  # atoms
7  elsif $<atom> {
8      make $<atom>.ast;
9  }
```

- atom から成る S 式
 - atom の ast をそのまま使う

S 式の解釈 [3]

```
10  # other expressions
11  else {
12      given $<sexpr>[0].ast {
13          when "car" {
14              make $<sexpr>[1]<sexpr>[0].ast;
15          }
16      }
17  }
```

S 式の解釈 [3]

```
10  # other expressions
11  else {
12      given $<sexpr>[0].ast {
13          when "car" {
14              make $<sexpr>[1]<sexpr>[0].ast;
15          }
16      }
17  }
```

その他の式: S 式の 1 番目の要素で分岐

S 式の解釈 [3]

```
10  # other expressions
11  else {
12      given $<sexpr>[0].ast {
13          when "car" {
14              make $<sexpr>[1]<sexpr>[0].ast;
15          }
16      }
17  }
```

その他の式: S 式の 1 番目の要素で分岐

- car 関数

S 式の解釈 [3]

```
10  # other expressions
11  else {
12    given $<sexpr>[0].ast {
13      when "car" {
14        make $<sexpr>[1]<sexpr>[0].ast;
15      }
16    }
17  }
```

その他の式: S 式の 1 番目の要素で分岐

- car 関数
 - 引数に受け取ったリストの先頭の要素を返す

S 式の解釈 [3]

```
10 # other expressions
11 else {
12     given $<sexpr>[0].ast {
13         when "car" {
14             make $<sexpr>[1]<sexpr>[0].ast;
15         }
16     }
17 }
```

その他の式: S 式の 1 番目の要素で分岐

- car 関数

- 引数に受け取ったリストの先頭の要素を返す
- 2 番目の S 式 (つまり引数に受け取った S 式) の 1 番目の要素の ast を返す

S 式の解釈 [4]

```
16         when "equal" {
17             make $<sexpr>[1].ast eq $<sexpr>[2].ast ?? "t" !! "nil";
18         }
19     }
20 }
21 }
```

S 式の解釈 [4]

```
16         when "equal" {  
17             make $<sexpr>[1].ast eq $<sexpr>[2].ast ?? "t" !! "nil";  
18         }  
19     }  
20 }  
21 }
```

- equal 関数

S 式の解釈 [4]

```
16     when "equal" {  
17         make $<sexpr>[1].ast eq $<sexpr>[2].ast ?? "t" !! "nil";  
18     }  
19 }  
20 }  
21 }
```

- equal 関数

- 2つの引数が等しいなら t、そうでないなら nil

S 式の解釈 [4]

```
16     when "equal" {
17         make $<sexpr>[1].ast eq $<sexpr>[2].ast ?? "t" !! "nil";
18     }
19 }
20 }
21 }
```

● equal 関数

- 2つの引数が等しいなら t、そうでないなら nil
- 2番目の S 式 (つまり第 1 引数) の ast と
3番目の S 式 (つまり第 2 引数) の ast を比較して、
等しいなら t を、そうでないなら nil を返す

実行してみる

実行部分:

実行してみる

実行部分:

コマンドライン引数に評価したい式を渡す

```
1 my $str = @*ARGS[0];  
2 my $m = Lisp.parse($str, actions => Evaluate);  
3 say $m.ast;
```

実行してみる

実行部分:

コマンドライン引数に評価したい式を渡す

```
1 my $str = @*ARGS[0];  
2 my $m = Lisp.parse($str, actions => Evaluate);  
3 say $m.ast;
```

先程の式を渡すと...

```
% perl6 lisp.pl "(equal (car '(1 2)) 2)"  
nil
```


実行してみる

実行部分:

コマンドライン引数に評価したい式を渡す

```
1 my $str = @*ARGS[0];  
2 my $m = Lisp.parse($str, actions => Evaluate);  
3 say $m.ast;
```

先程の式を渡すと...

```
% perl6 lisp.pl "(equal (car '(1 2)) 2)"  
nil
```

少し変えてみる

```
% perl6 lisp.pl "(equal (car '(1 2)) 1)"  
t
```

実行してみる

実行部分:

コマンドライン引数に評価したい式を渡す

```
1 my $str = @*ARGS[0];  
2 my $m = Lisp.parse($str, actions => Evaluate);  
3 say $m.ast;
```

先程の式を渡すと...

```
% perl6 lisp.pl "(equal (car '(1 2)) 2)"  
nil
```

少し変えてみる

```
% perl6 lisp.pl "(equal (car '(1 2)) 1)"  
t
```

ちゃんと動いてるっぽい！

喜ぶのはまだ早い？

目標の要求定義が適当だったので見逃しているが...

喜ぶのはまだ早い？

目標の要求定義が適当だったので見逃しているが...
この設計には致命的な欠陥がある！

喜ぶのはまだ早い？

目標の要求定義が適当だったので見逃しているが...
この設計には致命的な欠陥がある！

- quote: 引数に受け取った S 式をそのまま返す

喜ぶのはまだ早い？

目標の要求定義が適当だったので見逃しているが...
この設計には致命的な欠陥がある！

- quote: 引数に受け取った S 式をそのまま返す
- quote: 引数に受け取った S 式を評価せずに返す

Action の実行される順番を考える [1]

```
1 => <(equal (car '(1 2)) 1)> #22
2 sexpr => <(equal (car '(1 2)) 1)> #21
3 sexpr => <equal > #3
4 atom => <equal> #2
5 symbol => <equal> #1
6 sexpr => <(car '(1 2)) > #16
7 sexpr => <car > #6
8 atom => <car> #5
9 symbol => <car> #4
10 sexpr => <'(1 2)> #15
11 sexpr => <1 > #10
12 atom => <1> #9
13 literal => <1> #8
14 num => <1> #7
15 sexpr => <2> #14
16 atom => <2> #13
17 literal => <2> #12
18 num => <2> #11
19 sexpr => <1> #20
20 atom => <1> #19
21 literal => <1> #18
22 num => <1> #17
```

Diagram illustrating the execution order of actions for the expression `(equal (car '(1 2)) 1)`. The actions are numbered 1 to 22, and the diagram shows the sequence of actions:

- 1: `=> <(equal (car '(1 2)) 1)>` (#22)
- 2: `sexpr => <(equal (car '(1 2)) 1)>` (#21)
- 3: `sexpr => <equal >` (#3)
- 4: `atom => <equal>` (#2)
- 5: `symbol => <equal>` (#1)
- 6: `sexpr => <(car '(1 2)) >` (#16)
- 7: `sexpr => <car >` (#6)
- 8: `atom => <car>` (#5)
- 9: `symbol => <car>` (#4)
- 10: `sexpr => <'(1 2)>` (#15)
- 11: `sexpr => <1 >` (#10)
- 12: `atom => <1>` (#9)
- 13: `literal => <1>` (#8)
- 14: `num => <1>` (#7)
- 15: `sexpr => <2>` (#14)
- 16: `atom => <2>` (#13)
- 17: `literal => <2>` (#12)
- 18: `num => <2>` (#11)
- 19: `sexpr => <1>` (#20)
- 20: `atom => <1>` (#19)
- 21: `literal => <1>` (#18)
- 22: `num => <1>` (#17)

Action の実行される順番を考えてみる [2]

```
1 => <(equal (car '(1 2)) 1)> #22
2 sexpr => <(equal (car '(1 2)) 1)> #21
3 sexpr => <equal > #3
4 atom => <equal> #2
5 symbol => <equal> #1
6 sexpr => <(car '(1 2)) > #16
7 sexpr => <car > #6
8 atom => <car> #5
9 symbol => <car> #4
10 sexpr => <'(1 2)> #15
11 sexpr => <1 > #10 <== 評価している！
12 atom => <1> #9
13 literal => <1> #8
14 num => <1> #7
15 sexpr => <2> #14 <== 評価している！
16 atom => <2> #13
17 literal => <2> #12
18 num => <2> #11
19 sexpr => <1> #20
20 atom => <1> #19
21 literal => <1> #18
22 num => <1> #17
```


副作用の悪夢

もし `'(1 2)` が `'(print 1) 2)` だったら、というお話

副作用の悪夢

もし'(1 2)が'(print 1) 2)だったら、というお話

- Lispには
 - 関数 (Functions)

副作用の悪夢

もし'(1 2)が'(print 1) 2)だったら、というお話

- Lispには
 - 関数 (Functions)
 - 特殊形式 (Special Forms)
 - quote, if, cond, define, etc.

がある

副作用の悪夢

もし'(1 2)が'(print 1) 2)だったら、というお話

- Lispには
 - 関数 (Functions)
 - 特殊形式 (Special Forms)
 - quote, if, cond, define, etc.

がある

- 関数では表現できない特別な評価ルール

副作用の悪夢

もし '(1 2) が '(**(print 1)** 2) だったら、というお話

- Lisp には
 - 関数 (Functions)
 - **特殊形式** (Special Forms)
 - quote, if, cond, define, etc.

がある

- 関数では表現できない特別な評価ルール
 - ex. (if cond (expr1) (expr2))

副作用の悪夢

もし'(1 2)が'(print 1) 2)だったら、というお話

- Lispには
 - 関数 (Functions)
 - 特殊形式 (Special Forms)
 - quote, if, cond, define, etc.

がある

- 関数では表現できない特別な評価ルール
 - ex. (if cond (expr1) (expr2))
 - expr1/expr2 のどちらかの~~み~~を評価したい

副作用の悪夢

もし '(1 2) が '(**(print 1)** 2) だったら、というお話

- Lisp には
 - 関数 (Functions)
 - **特殊形式** (Special Forms)
 - quote, if, cond, define, etc.

がある

- 関数では表現できない特別な評価ルール
 - ex. (if cond (expr1) (expr2))
 - expr1/expr2 のどちらかの**み**を評価したい
 - 今の設計では両方評価してしまう

何故こうなるのか

何故こうなるのか

- 深さ優先探索の戻り掛けだから

何故こうなるのか

- 深さ優先探索の戻り掛けだから
- つまりクォートされた式の内側に居るかどうかを知る手段が無い

何故こうなるのか

- 深さ優先探索の戻り掛けだから
- つまりクォートされた式の内側に居るかどうかを知る手段が無い

やはり無理か？

解決策

解決策

方針：特殊形式の中に入る S 式を区別したい

解決策

方針：特殊形式の中に入る S 式を区別したい

```
1  token spform { 'cond' | 'if' | 'define' | 'quote' }
2
3  # sexpr with special forms
4  rule sp_sexpr { '(' <spform> [<ne_sexpr>+? % <.ws>] ')' }
5  # sexpr with no evaluate
6  rule ne_sexpr { <atom> || '(' [<ne_sexpr>+? % <.ws>] ')' }
```

解決策

方針：特殊形式の中に入る S 式を区別したい

```
1  token spform { 'cond' | 'if' | 'define' | 'quote' }
2
3  # sexpr with special forms
4  rule sp_sexpr { '(' <spform> [<ne_sexpr>+? % <.ws>] ')' }
5  # sexpr with no evaluate
6  rule ne_sexpr { <atom> || '(' [<ne_sexpr>+? % <.ws>] ')' }
```

- 特殊形式の中は別のルール (ne_sexpr) にわせる

解決策

方針：特殊形式の中に入る S 式を区別したい

```
1  token spform { 'cond' | 'if' | 'define' | 'quote' }
2
3  # sexpr with special forms
4  rule sp_sexpr { '(' <spform> [<ne_sexpr>+? % <.ws>] ')' }
5  # sexpr with no evaluate
6  rule ne_sexpr { <atom> || '(' [<ne_sexpr>+? % <.ws>] ')' }
```

- 特殊形式の中は別のルール (ne_sexpr) にわせる
- S 式と同じ構造だが、action は何もせずに文字列を返すように実装

解決策

方針：特殊形式の中に入る S 式を区別したい

```
1  token spform { 'cond' | 'if' | 'define' | 'quote' }
2
3  # sexpr with special forms
4  rule sp_sexpr { '(' <spform> [<ne_sexpr>+? % <.ws>] ')' }
5  # sexpr with no evaluate
6  rule ne_sexpr { <atom> || '(' [<ne_sexpr>+? % <.ws>] ')' }
```

- 特殊形式の中は別のルール (ne_sexpr) にわせる
- S 式と同じ構造だが、action は何もせずに文字列を返すように実装
- sp_sexpr は元の文字列を使って上手くやる

まとめ

まとめ

- Perl6 は Grammar という機能を使って構造的な文字列を簡単にパースできる

まとめ

- Perl6 は Grammar という機能を使って構造的な文字列を簡単にパースできる
- Actions を指定する事で、パースしながら抽象構文木を作れる

まとめ

- Perl6 は Grammar という機能を使って構造的な文字列を簡単にパースできる
- Actions を指定する事で、パースしながら抽象構文木を作れる
- Actions の処理は制約が大きいので、それも混みで Grammar を書いた方が最終的に楽

おまけ

全体的に上手くやってるつもり (自称) の途中実装

おまけ

全体的に上手くやってるつもり (自称) の途中実装
<http://github.com/VienosNotes/Domino>

おまけ

全体的に上手くやってるつもり (自称) の途中実装
<http://github.com/VienosNotes/Domino>

- 発表のコードはかなり簡略化したもの

おまけ

全体的に上手くやってるつもり (自称) の途中実装
<http://github.com/VienosNotes/Domino>

- 発表のコードはかなり簡略化したもの
- ちゃんと他の関数も使えるように

おまけ

全体的に上手くやってるつもり (自称) の途中実装
<http://github.com/VienosNotes/Domino>

- 発表のコードはかなり簡略化したもの
- ちゃんと他の関数も使えるように
- できれば Perl6 の関数も呼べると良いよね

おまけ

全体的に上手くやってるつもり (自称) の途中実装
<http://github.com/VienosNotes/Domino>

- 発表のコードはかなり簡略化したもの
- ちゃんと他の関数も使えるように
- できれば Perl6 の関数も呼べると良いよね

おしまい