Próximo:Estendendo pcase, Acima:Condicional de correspondência de padrões [Conteúdo][Índice]

11.4.1 A `pcase`macro

Para mais informações, consulte Condicional de correspondência de padrões .

Macro: expressão pcase *e cláusulas de descanso*

Cada cláusula em *cláusulas* tem a forma: . (*`pattern body-forms`…*)

Avalie a *expressão* para determinar seu valor, *expval* . Encontre a primeira cláusula em *cláusulas* cujo *padrão* corresponda *a expval* e passe o controle para os *formulários do corpo* dessa cláusula .

Se houver uma correspondência, o valor de `pcase`é o valor da última das *formas do corpo* na cláusula de sucesso. Caso contrário, `pcase`avalia como `nil`.

Cada *padrão* deve ser um *padrão pcase* , que pode usar um dos padrões principais definidos abaixo ou um dos padrões definidos via `pcase-defmacro`(consulte Estendendo pcase).

O restante desta subseção descreve diferentes formas de padrões centrais, apresenta alguns exemplos e conclui com importantes advertências sobre o uso do recurso let-binding fornecido por alguns formulários de padrões. Um padrão de núcleo pode ter as seguintes formas:

_

Corresponde a qualquer *expval* . Isso também é conhecido como *don't care* ou *wildcard* .

'val

Corresponde se *expval* for igual a *val* . A comparação é feita como se por `equal`(veja Predicados de Igualdade).

keyword

integer

string

Corresponde se *expval* for igual ao objeto literal. Este é um caso especial de , acima, possível porque objetos literais desses tipos são autocitantes. *`'val`*

symbol

Corresponde a qualquer *símbolo expval* e, adicionalmente, let-binds *a expval* , *de* modo que essa associação esteja disponível para *formulários de corpo* (consulte Dynamic Binding).

Se o *símbolo* fizer parte de um padrão de sequenciamento *seqpat* (por exemplo, usando `and`, abaixo), a ligação também estará disponível para a parte de *seqpat* seguindo a aparência do *símbolo* . Este uso tem algumas ressalvas, veja ressalvas .

Dois símbolos a serem evitados são `t`, que se comporta como _ (acima) e está obsoleto, e `nil`, que sinaliza um erro. Da mesma forma, não faz sentido vincular símbolos de palavras-chave (consulte Variáveis constantes).

(pred *function*)

Corresponde se a *função* de predicado retornar non- `nil` quando chamada em *expval* . *a função* de predicado pode ter uma das seguintes formas:

nome da função (um símbolo)

Chame a função nomeada com um argumento, *expval* .

Exemplo:`integerp`

expressão lambda

Chame a função anônima com um argumento, *expval* (consulte Expressões Lambda).

Exemplo:`(lambda (n) (= 42 n))`

chamada de função com *n* argumentos

Chame a função (o primeiro elemento da chamada de função) com *n* argumentos (os outros elementos) e um argumento *n* +1-th adicional que é *expval* .

Exemplo: `(= 42)`
Neste exemplo, a função é `=`, *n* é um, e a chamada de função real se torna: . `(= 42 expval)`

`(app function pattern)`

Corresponde se a *função* chamada em *expval* retornar um valor que corresponda a *pattern* . *A função* pode assumir uma das formas descritas para `pred`, acima. Ao contrário `pred`de , no entanto, `app`testa o resultado em relação a um *padrão* , em vez de um valor de verdade booleano.

`(guard boolean-expression)`

Corresponde se *a expressão booleana* for avaliada como não `nil`.

`(let pattern expr)`

Avalia *expr* para obter *exprval* e corresponde se *exprval* corresponder a *pattern* . (É chamado `let`porque o *padrão* pode vincular símbolos a valores usando *símbolo* .)

Um *padrão de sequenciamento* (também conhecido como *seqpat*) é um padrão que processa seus argumentos de subpadrão em sequência. Existem dois para `pcase`: `and`e `or`. Eles se comportam de maneira semelhante aos formulários especiais que compartilham seu nome (consulte Combinando Condições), mas em vez de processar valores, eles processam subpadrões.

`(and pattern1…)`

Tenta corresponder ao *pattern1* …, na ordem, até que um deles falhe. Nesse caso, `and`também não corresponde e o restante dos subpadrões não é testado. Se todos os subpadrões corresponderem, `and`corresponde.

`(or pattern1 pattern2…)`

Tenta corresponder *pattern1* , *pattern2* , …, em ordem, até que um deles seja bem-sucedido. Nesse caso, `or`as correspondências também e o restante dos subpadrões não são testados. (Observe que deve haver pelo menos dois subpadrões. Simplesmente sinaliza erro.) `(or pattern1)`

Para apresentar um ambiente consistente (veja Intro Eval) para *formas de corpo* (evitando assim um erro de avaliação na correspondência), se algum dos subpadrões deixar-vincular um conjunto de símbolos, todos eles *devem* vincular o mesmo conjunto de símbolos.

`(rx rx-expr…)`

Faz a correspondência de strings com o regexp *rx-expr* …, usando a `rx`notação regexp (consulte Rx Notation), como se fosse por `string-match`.

Além da `rx`sintaxe usual, *rx-expr* … pode conter as seguintes construções:

(let *ref rx-expr…*)

Vincule o símbolo *ref* a uma subcorrespondência que corresponda *a rx-expr …* . *ref* é vinculado em *formas de corpo* à string da subcorrespondência ou nil, mas também pode ser usado em `backref`.

(backref *ref*)

Como a `backref`construção padrão, mas *ref* também pode ser um nome introduzido por uma construção anterior. `(let *ref* …)`

Exemplo: Vantagem Sobre`cl-case`

Aqui está um exemplo que destaca algumas vantagens `pcase` sobre `cl-case` (veja Condicionais em Extensões Common Lisp).

```
(pcase (get-return-code x)
  ;; corda
  ((e (pred stringp) msg)
   (mensagem "%s" mensagem))
  ;; símbolo
  ('sucesso (mensagem "Concluído!"))
  ('would-block (mensagem "Desculpe, não posso fazer isso agora"))
  ('somente leitura (mensagem "O shmliblick é somente leitura"))
  ('acesso negado (mensagem "Você não tem os direitos necessários"))
  ;; predefinição
  (código (mensagem "código de retorno desconhecido %S"))))
```

Com `cl-case`, você precisaria declarar explicitamente uma variável local `code`para manter o valor de retorno de `get-return-code`. Também `cl-case`é difícil de usar com strings porque usa `eql`para comparação.

Exemplo: usando`and`

Um idioma comum é escrever um padrão começando com `and`, com um ou mais subpadrões de *símbolo* fornecendo ligações para os subpadrões que se seguem (assim como para as formas do corpo). Por exemplo, o padrão a seguir corresponde a inteiros de um dígito.

```
(e
  (pred integerp)
  n; vincular na expval
  (guarda (<= -9 n 9)))
```

Primeiro, `pred`corresponde se for avaliado como não- . Em seguida, é um padrão de *símbolo* que corresponde a qualquer coisa e se vincula a *expval* . Por fim, corresponde se a expressão booleana (observe a referência a) for avaliada como não- . Se todos esses subpadrões corresponderem, corresponde. `(integerp *expval*)nilnnguard(<= -9 n 9)nniland`

Exemplo: Reformulação com`case`

Aqui está outro exemplo que mostra como reformular uma tarefa de correspondência simples de sua implementação tradicional (function `grok/traditional`) para uma usando `pcase`(function `grok/pcase`). A docstring para ambas as funções é: “Se OBJ for uma string no formato "key:NUMBER", retorne

NUMBER (uma string). Caso contrário, retorne a lista (padrão "149").” Primeiro, a implementação tradicional (veja Expressões Regulares):

```
(defun grok/tradicional (obj)
  (se (e (stringp obj)
          (string-match "^key:\\([[:digit:]]+\\)$" obj))
      (correspondência de string 1 obj)
    (lista "149" 'padrão)))


(grok/tradicional "chave:0") ⇒ "0"
(grok/tradicional "chave: 149") ⇒ "149"
(grok/tradicional 'monólito) ⇒ ("149" padrão)
```

A reformulação demonstra a vinculação de *símbolos* , bem como `or`, `and`, `pred`, `appe` `let`.

```
(defun grok/pcase (obj)
  (pcase obj
    ((ou ; linha 1
      (e ; linha 2
       (pred stringp) ; linha 3
       (pred (string-match ; linha 4
              "^key:\\([[:digit:]]+\\)$")) ; line 5
       (app (match-string 1) ; line 6
            val)); line 7
      (let val (list "149" 'default))); line 8
     val))); linha 9


(grok/pcase "chave:0") ⇒ "0"
(grok/pcase "chave:149") ⇒ "149"
(grok/pcase 'monólito) ⇒ ("149" padrão)
```

A maior parte de `grok/pcase`é uma única cláusula de um `pcase` formulário, o padrão nas linhas 1-8, a forma de corpo (único) na linha 9. O padrão é `or`, que tenta combinar seus subpadrões de argumento, primeiro `and`(linhas 2- 7), depois `let`(linha 8), até que um deles tenha sucesso.

Como no exemplo anterior (consulte o Exemplo 1), `and`começa com um `pred`subpadrão para garantir que os subpadrões a seguir funcionem com um objeto do tipo correto (string, neste caso). Se retorna , falha e, portanto , falha também. `(stringp` *`expval`*`)nilpredand`

O próximo `pred`(linhas 4-5) avalia e combina se o resultado for não- , o que significa que *expval* tem a forma desejada: . Novamente, falhando nisso, falha e também. `(string-match RX` *`expval`*`)nilkey:NUMBERpredand`

Por último (nesta série de `and`subpadrões), `app` avalia (linha 6) para obter um valor temporário *tmp* (ou seja, a substring “NUMBER”) e tenta combinar *tmp* com o padrão (linha 7). Como esse é um padrão de *símbolo* , ele corresponde incondicionalmente e, adicionalmente, vincula -se a *tmp* . `(match-string 1` *`expval`*`)valval`

Agora que `app`correspondeu, todos `and`os subpadrões corresponderam e, portanto, `and`correspondem. Da mesma forma, uma vez `and`combinado, `or`corresponde e não prossegue para tentar o subpadrão `let`(linha 8).

Vamos considerar a situação em que `obj`não é uma string, ou é uma string, mas tem a forma errada. Nesse caso, um dos `pred`(linhas 3-5) não corresponde, portanto `and`(linha 2) não corresponde, portanto `or`(linha 1) prossegue para tentar o subpadrão `let`(linha 8).

Primeiro, `let`avalia `(list "149" 'default)` como get `("149" default)`, o *exprval* e, em seguida, tenta corresponder *exprval* com o pattern `val`. Como esse é um padrão de *símbolo* , ele corresponde incondicionalmente e, adicionalmente, vincula -se `val`a *exprval* . Agora que `let`combinou, `or`combina.

Observe como ambos os padrões `and`e `let`subpadrões terminam da mesma maneira: tentando (sempre com sucesso) corresponder ao padrão de *símbolo*`val` , no processo de vinculação `val`. Assim, `or`sempre combina e o controle sempre passa para a forma do corpo (linha 9). Como esse é o último formulário do corpo em uma `pcase`cláusula com correspondência bem-sucedida, é o valor de `pcase`e, da mesma forma, o valor de retorno de `grok/pcase`(consulte O que é uma função).

Advertências para *símbolo* em padrões de sequenciamento

Todos os exemplos anteriores usam padrões de sequenciamento que incluem o subpadrão de *símbolo* de alguma forma. Aqui estão alguns detalhes importantes sobre esse uso.

1. Quando o *símbolo* ocorre mais de uma vez em *seqpat* , a segunda e as ocorrências subsequentes não se expandem para religação, mas sim para um teste de igualdade usando `eq`.

   O exemplo a seguir apresenta um `pcase`formulário com duas cláusulas e dois *seqpat* , A e B. A e B primeiro verificam se *expval* é um par (usando `pred`) e, em seguida, associam símbolos ao `care` `cdr` de *expval* (usando um `app`cada).

   Para A, como o símbolo `st`é mencionado duas vezes, a segunda menção torna-se um teste de igualdade usando `eq`. Por outro lado, B usa dois símbolos separados, `s1`e `s2`, ambos se tornam ligações independentes.

   ```
   (defun grok (objeto)
     (objeto pcase
       ((e (pred consp) ; seqpat A
             (aplicativo carro st); primeira menção: st
             (aplicativo cdr st)); segunda menção: st
        (lista 'eq st))
       ((e (pred consp); seqpat B
             (aplicativo carro s1); primeira menção: s1
             (aplicativo cdr s2)); primeira menção: s2
        (lista 'não-eq s1 s2))))

   (deixe ((s "uau!"))
     (grok (cons ss))) ⇒ (eq "yow!")
   (grok (cons "yo!" "yo!")) ⇒ (não-eq "yo!" "yo!")
   (grok '(4 2)) ⇒ (não-eq 4 (2))
   ```

2. *O símbolo* de referência de código de efeito colateral é indefinido. Evitar. Por exemplo, aqui estão duas funções semelhantes. Ambos usam `and`, *símbolo* e `guard`:

   ```
   (defun quadrado-dois dígitos-p/CLEAN (inteiro)
     (pcase (* inteiro inteiro)
       ((e n (guarda (< 9 n 100))) (lista 'sim n))
       (desculpe (lista 'não desculpe)))))
   ```

```
(quadrado de dois dígitos-p/CLEAN 9) ⇒ (sim 81)
(quadrado de dois dígitos p/CLEAN 3) ⇒ (nº 9)


(defun quadrado duplo dígito-p/TALVEZ (inteiro)
  (pcase (* inteiro inteiro)
    ((e n (guarda (< 9 (incf n) 100))) (lista 'sim n))
    (desculpe (lista 'não desculpe)))))

(quadrado de dois dígitos-p/TALVEZ 9) ⇒ (sim 81)
(quadrado de dois dígitos-p/TALVEZ 3) ⇒ (sim 9) ; ERRADO!
```

A diferença está na *expressão booleana* em `guard`: `CLEAN`referências `n`simples e diretas, enquanto `MAYBE`referências `n`com efeito colateral, na expressão `(incf n)`. Quando `integer`é 3, eis o que acontece:

- O primeiro `n`liga-o a *expval* , ou seja, o resultado da avaliação `(* 3 3)`, ou 9.
- *expressão booleana* é avaliada:

  ```
  início: (< 9 (incf n) 100)
  torna-se: (< 9 (setq n (1+ n)) 100)
  torna-se: (< 9 (conjunto n (1+ 9)) 100)
  torna-se: (< 9 (setq n 10) 100)
                                      ; efeito colateral aqui!
  torna-se: (< 9 n 100) ; nagora limitado a 10
  torna-se: (< 9 10 100)
  torna-se: t
  ```

- Como o resultado da avaliação não é `nil`, `guard`correspondências, `and`correspondências e controle passam para os formulários do corpo dessa cláusula.

Além da incorreção matemática de afirmar que 9 é um número inteiro de dois dígitos, há outro problema com `MAYBE`. A forma do corpo faz referência `n`mais uma vez, mas não vemos o valor atualizado – 10 – de forma alguma. O que aconteceu com isso?

Para resumir, é melhor evitar totalmente as referências de efeitos colaterais aos padrões de *símbolos , não apenas na expressão booleana* (in `guard`), mas também em *expr* (in `let`) e *function* (in `prede app`).

3. Na correspondência, as formas do corpo da cláusula podem fazer referência ao conjunto de símbolos que o padrão let-binds. Quando *seqpat* é `and`, este conjunto é a união de todos os símbolos que cada um de seus subpadrões let-binds. Isso faz sentido porque, para `and`corresponder, todos os subpadrões devem corresponder.

   Quando *seqpat* é `or`, as coisas são diferentes: `or`corresponde ao primeiro subpadrão que corresponde; o resto dos subpadrões são ignorados. Não faz sentido para cada subpadrão deixar vincular um conjunto diferente de símbolos porque as formas do corpo não têm como distinguir qual subpadrão corresponde e escolher entre os diferentes conjuntos. Por exemplo, o seguinte é inválido:

   ```
   (requer 'cl-lib)
   (pcase (read-number "Digite um inteiro: ")
   ```

```
  ((ou (e (pred cl-evenp))
           e-num); ligar e-numa expval
       o-num); vincular o-numa expval
   (lista e-num o-num)))


Digite um número inteiro: 42
error→ O valor do símbolo como variável é nulo: o-num
Digite um número inteiro: 149
error→ O valor do símbolo como variável é nulo: e-num
```

A avaliação da forma corporal sinaliza erro. Para distinguir entre subpadrões, você pode usar outro símbolo, idêntico em nome em todos os subpadrões, mas diferente em valor. Retrabalhando o exemplo acima: `(list e-num o-num)`

```
(requer 'cl-lib)
(pcase (read-number "Digite um inteiro: ")
  ((and num ; line 1
        (or (and (pred cl-evenp) ; line 2
                 (let spin 'even))); line 3
            (let spin 'odd))) ; line 4
   (list spin num))); linha 5


Digite um número inteiro: 42
⇒ (mesmo 42)
Digite um número inteiro: 149
⇒ (ímpar 149)
```

A linha 1 “fatora” a ligação *expval* com `and`e *símbolo* (neste caso, `num`). Na linha 2, `or`começa da mesma forma que antes, mas em vez de vincular símbolos diferentes, usa `let`duas vezes (linhas 3-4) para vincular o mesmo símbolo `spin`em ambos os subpadrões. O valor de `spin`distingue os subpadrões. A forma do corpo faz referência a ambos os símbolos (linha 5).

Próximo:Estendendo pcase, Acima:Condicional de correspondência de padrões [Conteúdo][Índice]