Próximo:Erros, Anterior:Pegar e jogar, Acima:Saídas não locais [Conteúdo][Índice]

11.7.2 Exemplos de `catch`e `throw`

Uma maneira de usar `catch`e `throw`é sair de um loop duplamente aninhado. (Na maioria das linguagens, isso seria feito com a `goto`.) Aqui calculamos para *i* e *j* variando de 0 a 9: `(foo i j)`

```
(defun search-foo ()
  (capturar 'loop
    (seja ((i 0))
      (enquanto (< i 10)
        (seja ((j 0))
          (enquanto (< j 10)
            (se (foo ij)
                (lance 'loop (lista ij)))
            (setq j (1+ j))))
        (setqi (1+i)))))))
```

Se `foo`alguma vez retornar não- `nil`, paramos imediatamente e retornamos uma lista de *i* e *j* . Se `foo`sempre retorna `nil`, `catch`retorna normalmente, e o valor é `nil`, já que esse é o resultado do `while`.

Aqui estão dois exemplos complicados, ligeiramente diferentes, mostrando dois pontos de retorno ao mesmo tempo. Primeiro, dois pontos de retorno com a mesma tag, `hack`:

```
(defun catch2 (tag)
  (pegue a etiqueta
    (jogue 'hack' sim)))
⇒ pegar2


(pegue 'hack
  (imprimir (catch2 'hack))
  'não)
-| sim
⇒ não
```

Como ambos os pontos de retorno possuem tags que correspondem ao `throw`, ele vai para o interno, aquele estabelecido em `catch2`. Portanto, `catch2`retorna normalmente com value `yes`, e esse valor é impresso. Finalmente, a segunda forma de corpo no exterior `catch`, que é `'no`, é avaliada e retornada do exterior `catch`.

Agora vamos mudar o argumento dado para `catch2`:

```
(pegue 'hack
  (imprimir (catch2 'quux))
  'não)
⇒ sim
```

Ainda temos dois pontos de retorno, mas desta vez apenas o externo tem a tag `hack`; o interior tem a etiqueta `quux`em vez disso. Portanto, `throw`faz com que o externo `catch`retorne o valor `yes`. A função `print`nunca é chamada e a forma do corpo `'no`nunca é avaliada.

Próximo:Erros, Anterior:Pegar e jogar, Acima:Saídas não locais [Conteúdo][Índice]