Próximo:Dividindo em listas, Anterior:Tipos simples, Acima:Tipos de personalização [Conteúdo][Índice

]

15.4.2 Tipos Compósitos

Quando nenhum dos tipos simples é apropriado, você pode usar tipos compostos, que criam novos tipos a partir de outros tipos ou de dados especificados. Os tipos ou dados especificados são chamados de *argumentos* do tipo composto. O tipo composto normalmente se parece com isso:

```
( argumentos do construtor …)
```

mas você também pode adicionar pares de valor-palavra-chave antes dos argumentos, assim:

```
( construtor { valor da palavra -chave } … argumentos …)
```

Aqui está uma tabela de construtores e como usá-los para escrever tipos compostos:

`(cons car-type cdr-type)`

O valor deve ser uma célula cons, seu CAR deve caber *car-type* e seu CDR deve caber *cdr-type* . Por exemplo, `(cons string symbol)` é um tipo de personalização que corresponde a valores como `("foo" . foo)`.

No buffer de personalização, o CAR e o CDR são exibidos e editados separadamente, cada um de acordo com seu tipo especificado.

`(list element-types…)`

O valor deve ser uma lista com exatamente tantos elementos quantos os *tipos de elementos* fornecidos; e cada elemento deve caber no *tipo de elemento* correspondente .

Por exemplo, `(list integer string function)` descreve uma lista de três elementos; o primeiro elemento deve ser um inteiro, o segundo uma string e o terceiro uma função.

No buffer de personalização, cada elemento é exibido e editado separadamente, de acordo com o tipo especificado para ele.

`(group element-types…)`

Isso funciona como `list` exceto para a formatação do texto no buffer personalizado. `list` rotula cada valor de elemento com sua tag; `group` não.

`(vector element-types…)`

Como `list` exceto que o valor deve ser um vetor em vez de uma lista. Os elementos funcionam da mesma forma que em `list`.

`(alist :key-type key-type :value-type value-type)`

O valor deve ser uma lista de cons-cells, o CAR de cada célula representando uma chave do tipo de customização *key-type* e o CDR da mesma célula representando um valor do tipo de customização *value-type* . O usuário pode adicionar e excluir pares de chave/valor e editar a chave e o valor de cada par.

Se omitido, *o tipo de chave e o tipo* de *valor são* padronizados para `sexp`.

O usuário pode adicionar qualquer chave que corresponda ao tipo de chave especificado, mas você pode dar a algumas chaves um tratamento preferencial especificando-as com `:options`(consulte Definições de variáveis). As chaves especificadas sempre serão mostradas no buffer de personalização (junto com um valor adequado), com uma caixa de seleção para incluir ou excluir ou desabilitar o par chave/valor da lista. O usuário não poderá editar as chaves especificadas pelo `:options`argumento da palavra-chave.

O argumento para as `:options`palavras-chave deve ser uma lista de especificações para chaves razoáveis na lista. Normalmente, eles são simplesmente átomos, que representam a si mesmos. Por exemplo:

```
:options '("foo" "bar" "baz")
```

especifica que existem três chaves conhecidas, a saber `"foo"`, `"bar"`e `"baz"`, que sempre serão mostradas primeiro.

Você pode querer restringir o tipo de valor para chaves específicas, por exemplo, o valor associado à `"bar"`chave só pode ser um número inteiro. Você pode especificar isso usando uma lista em vez de um átomo na lista. O primeiro elemento especificará a chave, como antes, enquanto o segundo elemento especificará o tipo de valor. Por exemplo:

```
:options '("foo" ("bar" inteiro) "baz")
```

Finalmente, você pode querer alterar a forma como a chave é apresentada. Por padrão, a chave é simplesmente mostrada como um `const`, pois o usuário não pode alterar as chaves especiais especificadas com a palavra- `:options`chave. No entanto, você pode querer usar um tipo mais especializado para apresentar a chave, como `function-item`se você soubesse que é um símbolo com uma ligação de função. Isso é feito usando uma especificação de tipo de personalização em vez de um símbolo para a chave.

```
:options '("foo"
           ((função-item alguma função) inteiro)
           "baz")
```

Muitos alistas usam listas com dois elementos, em vez de células contras. Por exemplo,

```
(defcustom list-alist
  '(("foo" 1) ("bar" 2) ("baz" 3))
  "Cada elemento é uma lista do formulário (KEY VALUE).")
```

ao invés de

```
(defcustom cons-alist
  '(("foo" . 1) ("bar" . 2) ("baz" . 3))
  "Cada elemento é uma célula cons (KEY . VALUE).")
```

Devido à forma como as listas são implementadas em cima das células contras, você pode tratar `list-alist`no exemplo acima como uma lista de células contras, onde o tipo de valor é uma lista

com um único elemento contendo o valor real.

```
(defcustom list-alist '(("foo" 1) ("bar" 2) ("baz" 3))
  "Cada elemento é uma lista do formulário (KEY VALUE)."
  :type '(alist :value-type (grupo inteiro)))
```

O `group`widget é usado aqui em vez de `list`apenas porque a formatação é mais adequada para a finalidade.

Da mesma forma, você pode ter listas com mais valores associados a cada chave, usando variações deste truque:

```
(defcustom person-data '(("brian" 50 t)
                         ("dorith" 55 zero)
                         ("ken" 52 t))
  "Alista de informações básicas sobre pessoas.
Cada elemento tem a forma (NAME AGE MALE-FLAG)."
  :type '(alist :value-type (grupo inteiro booleano)))
```

`(plist :key-type` *`key-type`* `:value-type` *`value-type`*`)`

Esse tipo de personalização é semelhante a `alist`(consulte acima), exceto que (i) as informações são armazenadas como uma lista de propriedades (consulte Listas de propriedades) e (ii) *key-type* , se omitido, assume como padrão em `symbol`vez de `sexp`.

`(choice` *`alternative-types`*`…)`

O valor deve caber em um dos *tipos alternativos* . Por exemplo, `(choice integer string)`permite um inteiro ou uma string.

No buffer de customização, o usuário seleciona uma alternativa através de um menu, podendo então editar o valor da forma usual para aquela alternativa.

Normalmente as strings neste menu são determinadas automaticamente a partir das escolhas; entretanto, você pode especificar diferentes strings para o menu incluindo a palavra- `:tag`chave nas alternativas. Por exemplo, se um número inteiro representa um número de espaços, enquanto uma string é um texto para usar literalmente, você pode escrever o tipo de personalização dessa maneira,

```
(escolha (integer :tag "Número de espaços")
        (string :tag "Texto literal"))
```

para que o menu ofereça '`Número de espaços`' e '`Texto literal`'.

Em qualquer alternativa para a qual `nil`não seja um valor válido, exceto a `const`, você deve especificar um padrão válido para essa alternativa usando a palavra- `:value`chave. Consulte Palavras-chave de tipo .

Se alguns valores estiverem cobertos por mais de uma das alternativas, customize escolherá a primeira alternativa em que o valor se encaixa. Isso significa que você deve sempre listar os tipos mais específicos primeiro e os mais gerais por último. Aqui está um exemplo de uso adequado:

```
(escolha (const :tag "Off" nil)
        símbolo (sexp :tag "Outro"))
```

Desta forma, o valor especial `nil`não é tratado como outros símbolos, e os símbolos não são tratados como outras expressões Lisp.

`(radio` *`element-types`*`…)`

Isso é semelhante a `choice`, exceto que as opções são exibidas usando botões de opção em vez de um menu. Isso tem a vantagem de exibir a documentação das opções quando aplicável e, portanto, geralmente é uma boa opção para uma escolha entre funções constantes (`function-item`tipos de personalização).

`(const` *`value`*`)`

O valor deve ser *valor* — nada mais é permitido.

O principal uso de `const`está dentro de `choice`. Por exemplo, `(choice integer (const nil))`permite um inteiro ou `nil`.

`:tag`é frequentemente usado com `const`, dentro de `choice`. Por exemplo,

```
(escolha (const :tag "Sim" t)
        (const :tag "Não" nil)
        (const :tag "Perguntar" foo))
```

descreve uma variável que `t`significa sim, `nil`significa não e `foo`significa “perguntar”.

`(other` *`value`*`)`

Essa alternativa pode corresponder a qualquer valor Lisp, mas se o usuário escolher essa alternativa, ele seleciona o valor *value* .

O principal uso de `other`é como o último elemento de `choice`. Por exemplo,

```
(escolha (const :tag "Sim" t)
        (const :tag "Não" nil)
        (other :tag "Pergunte" foo))
```

descreve uma variável que `t`significa sim, `nil`significa não, e qualquer outra coisa significa “perguntar”. Se o usuário escolher '`Perguntar`' do menu de alternativas, que especifica o valor `foo`; mas qualquer outro valor (não ou `t`) é exibido como '`nilfooPerguntar`', assim como `foo`.

`(function-item` *`function`*`)`

Como `const`, mas usado para valores que são funções. Isso exibe a string de documentação, bem como o nome da função. A string de documentação é aquela que você especifica com `:doc`, ou a string de documentação da própria *função* .

`(variable-item` *`variable`*`)`

Como `const`, mas usado para valores que são nomes de variáveis. Isso exibe a string de documentação, bem como o nome da variável. A string de documentação é aquela que você especifica com `:doc`, ou a string de documentação da própria *variável* .

`(set` *`types`*`…)`

O valor deve ser uma lista e cada elemento da lista deve corresponder a um dos *tipos* especificados.

Isso aparece no buffer de personalização como uma lista de verificação, para que cada um dos *tipos* possa ter um elemento correspondente ou nenhum. Não é possível especificar dois elementos

diferentes que correspondam ao mesmo dos *tipos* . Por exemplo, `(set integer symbol)`permite um inteiro e/ou um símbolo na lista; ele não permite vários inteiros ou vários símbolos. Como resultado, é raro usar tipos não específicos, como `integer`em um arquivo `set`.

Na maioria das vezes, os *tipos* em a `set`são `const`tipos, conforme mostrado aqui:

```
(conjunto (const :bold) (const :italic))
```

Às vezes, eles descrevem possíveis elementos em uma lista:

```
(set (cons :tag "Altura" (const height) integer)
     (cons :tag "Width" (const width) integer))
```

Isso permite que o usuário especifique um valor de altura opcionalmente e um valor de largura opcionalmente.

`(repeat` *`element-type`*`)`

O valor deve ser uma lista e cada elemento da lista deve caber no tipo *element-type* . Isso aparece no buffer de personalização como uma lista de elementos, com `'[INS]'` e `'[APAGAR]'` botões para adicionar mais elementos ou remover elementos.

`(restricted-sexp :match-alternatives` *`criteria`*`)`

Esta é a construção de tipo composto mais geral. O valor pode ser qualquer objeto Lisp que satisfaça um dos *critérios* . *critérios* devem ser uma lista, e cada elemento deve ser uma destas possibilidades:

- Um predicado — isto é, uma função de um argumento que retorna um `nil`ou não `nil`de acordo com o argumento. O uso de um predicado na lista indica que os objetos para os quais o predicado retorna não- `nil`são aceitáveis.
- Uma constante cotada - isto é, . Esse tipo de elemento na lista diz que o próprio *objeto* é um valor aceitável. `'`*`object`*

Por exemplo,

```
(restricted-sexp :match-alternatives
                 (integerp 't 'nil))
```

permite inteiros `t`e `nil`como valores legítimos.

O buffer de personalização mostra todos os valores legítimos usando sua sintaxe de leitura e o usuário os edita textualmente.

Aqui está uma tabela das palavras-chave que você pode usar em pares de valor-palavra-chave em um tipo composto:

`:tag` *`tag`*

Use *tag* como nome desta alternativa, para fins de comunicação com o usuário. Isso é útil para um tipo que aparece dentro de um arquivo `choice`.

`:match-alternatives` *`criteria`*

Use *critérios* para corresponder aos valores possíveis. Isso é usado apenas em `restricted-sexp`.

:args ***argument-list***

Use os elementos de *argument-list* como os argumentos da construção de tipo. Por exemplo, `(const :args (foo))`é equivalente a `(const foo)`. Você raramente precisa escrever `:args`explicitamente, porque normalmente os argumentos são reconhecidos automaticamente como o que segue o último par palavra-chave-valor.

Próximo:Dividindo em listas, Anterior:Tipos simples, Acima:Tipos de personalização [Conteúdo][Índice

]