Próximo:Células de Função, Anterior:Funções anônimas, Acima:Funções [Conteúdo][Índice]

13.8 Funções Genéricas

As funções definidas usando `defun`têm um conjunto codificado de suposições sobre os tipos e valores esperados de seus argumentos. Por exemplo, uma função que foi projetada para lidar com valores de seu argumento que são números ou listas de números falhará ou sinalizará um erro se chamada com um valor de qualquer outro tipo, como um vetor ou uma string. Isso acontece porque a implementação da função não está preparada para lidar com tipos diferentes daqueles assumidos durante o projeto.

Em contraste, os programas orientados a objetos usam *funções polimórficas* : um conjunto de funções especializadas com o mesmo nome, cada uma das quais foi escrita para um determinado conjunto específico de tipos de argumentos. Qual das funções é realmente chamada é decidida em tempo de execução com base nos tipos dos argumentos reais.

O Emacs fornece suporte para polimorfismo. Assim como outros ambientes Lisp, notadamente o Common Lisp e seu Common Lisp Object System (CLOS), esse suporte é baseado em *funções genéricas* . As funções genéricas do Emacs seguem de perto o CLOS , incluindo o uso de nomes semelhantes, portanto, se você tiver experiência com o CLOS , o restante desta seção parecerá muito familiar.

Uma função genérica especifica uma operação abstrata, definindo seu nome e lista de argumentos, mas (geralmente) nenhuma implementação. A implementação real de várias classes específicas de argumentos é fornecida por *métodos* , que devem ser definidos separadamente. Cada método que implementa uma função genérica tem o mesmo nome que a função genérica, mas a definição do método indica quais tipos de argumentos ele pode manipular *especializando* os argumentos definidos pela função genérica. Esses *especialistas em argumentos* podem ser mais ou menos específicos; por exemplo, um `string`tipo é mais específico do que um tipo mais geral, como `sequence`.

Observe que, diferentemente das linguagens OO baseadas em mensagens, como C ++ e Simula, métodos que implementam funções genéricas não pertencem a uma classe, eles pertencem à função genérica que implementam.

Quando uma função genérica é invocada, ela seleciona os métodos aplicáveis comparando os argumentos reais passados pelo chamador com os especialistas de argumento de cada método. Um método é aplicável se os argumentos reais da chamada forem compatíveis com os especialistas do método. Se mais de um método for aplicável, eles serão combinados usando certas regras, descritas abaixo, e a combinação tratará da chamada.

Macro: cl-defgeneric *name arguments [documentation] [options-and-methods…] &rest body*

Esta macro define uma função genérica com o *nome* e *argumentos* especificados . Se *body* estiver presente, ele fornece a implementação padrão. Se a *documentação* estiver presente (deve sempre estar), ela especifica a string de documentação para a função genérica, no formato . As *opções e métodos* opcionais podem ser uma das seguintes formas: `(:documentation` *`docstring`*`)`

`(declare` *`declarations`*`)`

Um formulário de declaração, conforme descrito em Formulário de declaração .

`(:argument-precedence-order &rest` *`args`*`)`

Este formulário afeta a ordem de classificação para combinar métodos aplicáveis. Normalmente, quando dois métodos são comparados durante a combinação, os argumentos do método são examinados da esquerda para a direita, e o primeiro método cujo especialista de argumento é mais específico virá antes do outro. A ordem definida por este formulário substitui isso, e os argumentos são examinados de acordo com sua ordem neste formulário, e não da esquerda para a direita.

`(:method [`*`qualifiers…`*`] args &rest body)`

Este formulário define um método como `cl-defmethod`faz.

Macro: cl-defmethod ***name [qualifier] arguments [&context (expr spec)…] &rest [docstring] body***

Essa macro define uma implementação específica para a função genérica chamada *name* . O código de implementação é dado por *body* . Se presente, *docstring* é a string de documentação do método. A lista de *argumentos* , que deve ser idêntica em todos os métodos que implementam uma função genérica e deve corresponder à lista de argumentos dessa função, fornece especialistas de argumento da forma , onde *arg* é o nome do argumento conforme especificado na chamada e *spec* é uma das seguintes formas especializadas: `(`*`arg spec`*`)`

`cl-defgeneric`

`type`

Este especialista requer que o argumento seja do *tipo* fornecido , um dos tipos da hierarquia de tipos descrita abaixo.

`(eql `*`object`*`)`

Este especialista requer que o argumento seja `eql`para o *objeto* dado .

`(head `*`object`*`)`

O argumento deve ser uma célula cons cujo *objeto*`car` é . `eql`

`struct-type`

O argumento deve ser uma instância de uma classe chamada *struct-type* definida com `cl-defstruct`(consulte Estruturas em Extensões Lisp Comuns para GNU Emacs Lisp), ou de uma de suas classes filhas.

As definições de método podem fazer uso de uma nova palavra-chave de lista de argumentos, `&context`, que introduz especialistas extras que testam o ambiente no momento em que o método é executado. Essa palavra-chave deve aparecer após a lista de argumentos obrigatórios, mas antes de qualquer `&rest`ou palavras- `&optional`chave. Os `&context`especialistas se parecem muito com os especialistas de argumentos regulares—(*expr spec*)—exceto que *expr* é uma expressão a ser avaliada no contexto atual, e a *especificação* é um valor para comparação. Por exemplo, `&context (overwrite-mode (eql t))`tornará o método aplicável somente quando `overwrite-mode`estiver ativado. O `&context`palavra-chave pode ser seguida por qualquer número de especialistas de contexto. Como os especialistas de contexto não fazem parte da assinatura de argumento da função genérica, eles podem ser omitidos em métodos que não os exigem.

O especialista de tipo, , pode especificar um dos *tipos de sistema* na lista a seguir. Quando um tipo pai é especificado, um argumento cujo tipo é qualquer um de seus tipos filho mais específicos, bem como netos, bisnetos, etc. também serão compatíveis. `(`*`arg type`*`)`

`integer`

Tipo pai: `number`.

number

null

Tipo pai:`symbol`

symbol

string

Tipo pai: `array`.

array

Tipo pai: `sequence`.

cons

Tipo pai: `list`.

list

Tipo pai: `sequence`.

marker

overlay

float

Tipo pai: `number`.

window-configuration

process

window

subr

compiled-function

buffer

char-table

Tipo pai: `array`.

bool-vector

Tipo pai: `array`.

vector

Tipo pai: `array`.

frame

hash-table

font-spec

font-entity

font-object

O qualificador opcional permite combinar vários métodos aplicáveis. Se não estiver presente, o método definido é um método *primário* , responsável por fornecer a implementação primária da função genérica para os argumentos especializados. Você também pode definir *métodos auxiliares* , usando um dos seguintes valores como *qualificador* :

:before

Este método auxiliar será executado antes do método primário. Mais precisamente, todos os `:before`métodos serão executados antes do primário, na primeira ordem mais específica.

`:after`

Este método auxiliar será executado após o método primário. Mais precisamente, todos esses métodos serão executados após o primário, na ordem mais específica-última.

`:around`

Este método auxiliar será executado *em vez* do método primário. O mais específico desses métodos será executado antes de qualquer outro método. Tais métodos normalmente usam `cl-call-next-method`, descrito abaixo, para invocar os outros métodos auxiliares ou primários.

`:extra` *`string`*

Isso permite adicionar mais métodos, diferenciados por *string* , para os mesmos especialistas e qualificadores.

Funções definidas usando `cl-defmethod`não podem ser interativas, ou seja, comandos (veja Definindo Comandos), adicionando o `interactive`formulário a elas. Se você precisar de um comando polimórfico, recomendamos definir um comando normal que chame uma função polimórfica definida por `cl-defgenerice` `cl-defmethod`.

Cada vez que uma função genérica é chamada, ela cria o *método efetivo* que tratará essa chamada combinando os métodos aplicáveis definidos para a função. O processo de encontrar os métodos aplicáveis e produzir o método eficaz é chamado de *despacho* . Os métodos aplicáveis são aqueles cujos especialistas são compatíveis com os argumentos reais da chamada. Como todos os argumentos devem ser compatíveis com os especialistas, todos eles determinam se um método é aplicável. Os métodos que especializam explicitamente mais de um argumento são chamados *de métodos de despacho múltiplo* .

Os métodos aplicáveis são classificados na ordem em que serão combinados. O método cujo especialista de argumento mais à esquerda é o mais específico virá primeiro na ordem. (Especificando `:argument-precedence-order`como parte das `cl-defmethod` substituições, conforme descrito acima.) Se o corpo do método chamar `cl-call-next-method`, o próximo método mais específico será executado. Se houver `:around`métodos aplicáveis, o mais específico deles será executado primeiro; ele deve chamar para executar qualquer um dos métodos `cl-call-next-method`menos específicos . `:around`Em seguida, os `:before`métodos são executados na ordem de sua especificidade, seguidos pelo método primário e, por último, os `:after`métodos na ordem inversa de sua especificidade.

Função: cl-call-next-method *&rest args*

Quando invocado de dentro do corpo léxico de um `:around`método primário ou auxiliar, chame o próximo método aplicável para a mesma função genérica. Normalmente, ele é chamado sem argumentos, o que significa chamar o próximo método aplicável com os mesmos argumentos que o método chamador foi invocado. Caso contrário, os argumentos especificados serão usados.

Função: cl-next-method-p

Essa função, quando chamada de dentro do corpo léxico de um `:around`método primário ou auxiliar, retorna non- `nil`se houver um próximo método a ser chamado.

Próximo:Células de Função, Anterior:Funções anônimas, Acima:Funções [Conteúdo][Índice]