Aplicação em LISP e introdução ao CLOS

Diego D. B. Carvalho, Vinicius A. Carlos

Ciências da Computação 4ª fase, 2002
Departamento de Informática e Estatística.
Universidade Federal de Santa Catarina (UFSC), Brasil, 88040-900
Fone (0XX48)333-9999, Fax (0XX48)333-9999
diegodbc@inf.ufsc.br, vcarlos@inf.ufsc.br

Resumo

Este trabalho discute o código fonte de uma aplicação de jogo da velha em LISP capaz de jogar contra o usuário, introduz conceitos de CLOS (Common LISP Object System) e discorre sobre a versatilidade da linguagem funcional LISP e suas implementações .

Palavras-chave: Programação Funcional, LISP, CLOS.

Abstract

This papers discusses the source code of a pplication of tic-tac-toe in LISP capable to play against the user, it introduces concepts of CLOS (Common LISP Object System) and it discourses about functional language LISP versatily and its implementations.

Key-words: Functional Programming, LISP, CLOS.

Introdução

Este trabalho tem como objetivo aprofundar os conceitos de LISP demonstrando uma pequena aplicação, falar a respeito de orientação a objetos em LISP (CLOS) e como este conceito se relaciona com a programação funcional.

História

No fim dos anos 70 os conceitos da programação orientada a objetos começavam a ter um forte impacto no LISP. No MIT, certas ideias do SmallTalk caminharam para sistemas de programação largamente usados. Flavors, um sistema de programação orientada a objetos com herança múltipla, foi desenvolvida no MIT para a comunidade LISP por Howard Cannon e outros. Na Xerox, a experiência com o Smalltalk e Knowledge Representation Language (KRL) levaram ao desenvolvimento do LISP Object Oriented Programming System (LOOPS) e mais tarde ao Common LOOPS.

CLOS

Estes sistemas influenciaram o desenvolvimento do Common LISP Object System (CLOS). CLOS foi desenvolvido especificamente para este esforço de normalização e foi separadamente escrito no {Common LISP Object System Specification}. No entanto, pequenos detalhes do seu projeto foram ligeiramente mudados desde essa publicação, que não deve ser tomada como uma referencia autoritária na semântica de orientação a objeto.

Características

CLOS é a parte do Common LISP responsável pelas características usuais de uma linguagem de programação orientada a objetos como classes, métodos, herança etc.

Para a implementação de CLOS em Commom LISP foram criadas macros para definir classes, métodos criar instâncias. Abaixo veremos as três macros responsáveis por toda estrutura de CLOS.

-Definindo uma Classe

(DEFCLASS nome-da-classe (nome-da-superclasse*)
(descrição-de-slots*)

```
    opções-de-classe*
  )
```

Exemplo:

```
(defclass pessoa ()
    ((nome :accessor nome-pessoa
          :initform ‘joao
          :initarg :nome
     )
     (idade :accessor idade-pessoa
          :initform 15
          :initarg :idade
  ) ))
```

Com DEFCLASS definimos os slots e também já podemos definir os que seriam os métodos de acesso para os slots apenas colocando a opção :ACCESSOR *(nome-do-método)*, seus valores iniciais, setados através da opção :INITFORM *valor* e se esses slots serão iniciados ao chamar a macro MAKE-INSTANCE, colocando a opção :INITARG *:nome-do-slot*..

Observe que os slots são como atributos nas classes que definimos em outras linguagens de programação.

Pode-se criar uma instância de uma classe com MAKE-INSTANCE. É similar às funções MAKE-x definidas por DEFSTRUCT, porém é permitido passe a classe a ser instanciada como argumento:

```
  (MAKE-INSTANCE class {initarg value}*)
```

Ao invés do objeto classe (é o que o símbolo de nome de classe está denotando: uma variável global que representa o objeto-classe) pode-se simplesmente usar o seu nome. Por exemplo:

```
  (make-instance 'pessoa :idade 20)
```

Esta pessoa teria idade 20 e nome diego por default.
É muitas vezes uma boa definir os seus próprios métodos construtores, ao invés de chamar MAKE-INSTANCE diretamente, porque assim pode-se encapsular detalhes de implementação e não precisa utilizar parâmetros-de-palavra-chave para nada. Você poderia desejar definir:

```
 (defun make-pessoa (nome idade)
   (make-instance 'pessoa
                  :nome nome
                  :idade idade))
```

caso queira que o nome e a idade sejam requeridos através de parâmetros posicionais.

Os métodos de acesso podem ser usados para pegar e setar valores de slots (variáveis de instância), como nos exemplos abaixo:

```
<cl> (setq p1 (make-instance 'pessoa :nome 'maria
:idade 50))
#<pessoa @ #x7bf826>

<cl> (nome-pessoa p1)
maria

<cl> (idade-pessoa p1)
50

<cl> (setf (idade-pessoa p1) 101)
101

<cl> (idade-pessoa p1)
101
```

Para definir uma subclasse basta incluir o nome da superclasse entre parênteses na definição da classe.

```
  <cl> (defclass jogador-futebol (pessoa)
       ((time :accessor time-jogador
              :initarg :criciúma
     ) ))
```

CLOS também possui suporte a herança múltipla, bastando incluir mais de uma classe entre parênteses.

-Definindo um Método ou Função Genérica

```
(DEFMETHOD nome-funcao-generica lista-
lambda
   corpo-funcao*
  )
```

Métodos são a maneira pela qual os objetos se comunicam. Em LISP As mensagens/operações são funções genéricas cujo comportamento pode ser definido para instâncias de classes particulares através da definição de métodos.

Exemplo:
Este é um método utilizado para atribuir um novo valor ao slot jogador-futebol-time

```
(defmethod mudar-time ((time jogador-futebol)
                       (novo-time string))
  (setf (jogador-futebol-time time) novo-time)
)
```

Aplicações com LISP e CLOS

Há um certo estigma com relação a LISP no que diz respeito aos tipos de aplicações em que LISP é utilizado. Muito se fala de programas para a área de Inteligência Artificial, pela facilidade de se programar e pela característica marcante de LISP poder alterar seu comportamento durante a execução de um programa, porém, LISP é utilizado também em aplicações industriais. Em http://www.lisp.org/table/good.htm pode-se encontrar bons motivos para se usar LISP e onde aplicações em LISP são encontradas.

Um exemplo interessante é o de uma empresa desenvolvedora de jogos californiana Naugth Dog, criou um ambiente baseado em CLOS e no ambiente multiplatorma Alegro CL, chamada GOOL, particular para desenvolvimento de games.

Neste ambiente foram criados jogos para o sistema Playstation e Playstation 2, este último possui um hardware extremamente complexo, com diversos processadores. O argumento utilizado foi que apesar de a aplicação gerada tenha um desempenho menor do que se fosse programada em C++ por exemplo, ganha-se muito em portabilidade e facilidade de compreensão do código, resultando em uma grande economia em equipamentos e pessoal.

Aplicação: Jogo da Velha

A aplicação desenvolvida é bem simples, é um jogo da velha jogado contra o computador. A maior diferença é que ele foi desenvolvido com interface gráfica. Para isso foi utilizado uma das implementações para Common LISP chamada Allegro Common LISP 6.02. Apesar de não ser uma implementação gratuita de Common LISP ele permite licença por tempo indeterminado para as funções básicas.

Allegro Common LISP 6.02 prove um ambiente de desenvolvimento gráfico (IDE) com seus componentes baseados em CLOS. Desse modo, a parte de CLOS que há na aplicação do jogo da velha é definida pelos componentes gráficos da aplicação.

A dinâmica do jogo é toda desenvolvida pelas definições de funções básicas relacionadas ao tratamento do tabuleiro que podem ser utilizadas fora do Allegro sem maiores problemas. A exceção diz respeito ao tratamento de eventos que é feita com o apoio do Allegro.

O tabuleiro é tratado como uma variável global setada no início da aplicação via função DEFVAR

```
(defvar *tabuleiro* '(board 0 0 0 0 0 0 0 0 0))
```

O tabuleiro é uma lista contendo as posições de 1 a 9, correspondente as casas do tabuleiro, que se está ocupada por uma jogada do humano é mudado para 1 seu valor e quando corresponde a uma jogada do computador é setado para 10. Assim o tabuleiro vai se modificando durante a partida.

A idéia para a programação foi dividir cada atividade em uma função, que é a idéia da programação funcional.

As duas principais funções do jogo são jogada-oponente e jogada-computador que é mostrada abaixo:

```
(defun jogada-computador (tabuleiro window)
  (let* ((melhor-jogada (escolha-melhor-jogada tabuleiro))
         (pos (first melhor-jogada))
         (*tabuleiro* (fazer-jogada *computador* pos tabuleiro)))
    (cond ((vencedor-p *tabuleiro*) (escreva-venceu window pos))
          ((tabuleiro-cheio-p *tabuleiro*) (escreva-empatou-ultima window pos))
          (t (escreva-jogada window pos))
    )
  )
)
```

Essa função recebe como parâmetro o tabuleiro do jogo modificado e a janela onde o jogo está sendo apresentado graficamente. Conhecendo-se as funções let* e cond se torna muito fácil de entender o resto do código. As funções definidas com –p ao final do nome são chamadas de predicado e retornam valor booleano. Então o que essa função faz é escolher a melhor jogada possível para o computador, atrelá-la a melhor-jogada, modificar o tabuleiro com essa jogada e verificar se o jogo acabou ou pelo fato do computador ter vencido ou se empatou, caso nenhum desses caso tenha acontecido o jogo fica esperando pela jogada do humano.

A função jogada-oponente é muito similar a essa apresentada acima.

A inteligência do computador está na função escolha-melhor-jogada que recebe o tabuleiro, analisa e retorna a posição onde deve ser feita a jogada.

No caso do jogo da velha não é necessário um algoritmo minimax para decidir pela melhor jogada, uma vez que as possibilidades são poucas, então a idéia utilizada foi definir estratégias de jogo muito comuns de ocorrer.

A função responsável pelo carregamento do jogo é rodar-jogo que inicializa todos os objetos gráficos existentes na aplicação e chama a função jogar que inicializa a dinâmica do jogo propriamente dita.

O código fonte da aplicação pode ser encontrado no endereço http://www.inf.ufsc.br/~vcarlos/jogodavelha/

Um pouco de LISP

LISP tem muitas funções pré-definidas e querer falar sobre todas elas não é o objetivo desse artigo. Porém LISP têm muitas características interessantes que podem ser observadas através dessas funções.

Uma das mais interessantes é o fato de LISP permitir que funções possam ser passadas como parâmetros para outras funções. Isso dá uma grande habilidade para programas em LISP que podem se modificar através da mudança das funções que podem ser passadas para outras funções.

Todas as funções criadas a partir de funções são criadas a partir de uma função primitiva chamada FUNCALL.

Com FUNCALL é possível chamar uma função passando alguns argumentos para a mesma.

Exemplo:
(funcall #´cons ‘a ‘b) ➔ (a . b)

É possível fazer também:

(setf fn #´cons) onde fn recebe a função cons e depois fazer:

(funcall fn ‘c ‘d) ➔ (c . d)

Com o operador MAPCAR é possível passar uma lista para uma função de modo que a função vai ser aplicada a cada elemento da lista. Por exemplo:

(defun quadrado (n) (* n n))

(mapcar #´square ‘(1 2 3 4 5)) ➔ (1 4 9 16 25)

Não é preciso definir a função previamente para usá-la com o MAPCAR, pode-se passar a definição diretamente para o MAPCAR via uma expressão lambda, como no exemplo abaixo:

(mapcar #´(lambda (n) (* n n)) ‘(1 2 3 4 5))
(1 4 9 16 25)

Existem outros operadores como FIND-IF, REMOVE-IF, REDUCE que também se aplicam a outras funções e listas.

O mais interessante, no entanto é poder escrever seu próprio operador que recebe uma função como argumento, como por exemplo:

```
(defun direitos-legitimos (fn)
  (funcall fn ‘(vida liberdade e felicidade)))
```

Ao chamar essa função é só passar como argumento uma função que ela se aplicará à lista:

(direitos-legitimos #´lenght) ➔ 4

(direitos-legitimos #´reverse) ➔
(felicidade e liberdade vida)

(direitos-legitimos #´first) ➔ vida

Conclusões

Estudar LISP, para quem já tem uma história em programação orientada à objeto como Java, C++ ou Object Pascal, não é tão fácil de assimilar, apesar da sintaxe clara da linguagem, que justifica o pouco contato com LISP por parte dos alunos de gradução do curso de Ciências da Computação da Universidade Federal de Santa Catarina.

Estudar LISP mais afundo é intrigante pois podemos perceber que a linguagem é mais que uma ferramenta importante em aplicações para Inteligência Artificial, ela é uma linguagem madura, altamente portável, que vários desenvolvedores a utilizam em várias áreas de programação.

A versatilidade do LISP nos surpreendeu de tal maneira que, ao ter contato com as mais diferentes implementações para LISP, seus vários pacotes e ferramentas, testamos um pouco de tudo porém não foi possível desenvolver uma aplicação mais robusta.

Esses pacotes e extensões existem para os mais diferentes tipos de aplicação, desde aplicações gráficas até as que envolvem criação de músicas.

Portanto, aplicações em LISP são totalmente possíveis, viáveis e factíveis.

Referências

[1] T. Hasemer and J. Domingue, "Common Lisp Programming for Artificial Inteligence", Adison-Wesley Publishers Ltd. 1989.

[2] D. S. Touretzky, "Common Lisp: A Gentle Introduction to Symbolic Computation", The Benjamin/Cummings Publishing Company 1990.

[3] www.franz.com

[4] www.LISP.org

[5] www.naughtydog.com

[6] www.gamasutra.com

[7] www.inf.ufsc.br/~awangenh