Lisp: sintaxe e semântica

Abrantes Araújo Silva Filho

2020-03-15

Este texto traduz meu entendimento do Capítulo 4 do livro de **Peter Seibel**, *Practical Common Lisp*, e serve como um "resumo" para meu próprio consumo futuro. O livro está disponível gratuitamente na internet[1] para leitura online, ou você pode adquirir um exemplar na Amazon[2] ou outra livraria.

Obviamente não posso ser considerado o autor deste texto pois praticamente tudo aqui foi retirado do *Practical Common Lisp*. Minha contribuição foi apenas a de resumir o texto, torná-lo um pouco mais claro para iniciantes e acrescentar figuras e códigos que eu mesmo produzi para ilustrar e tornar mais claros os conceitos.

Nota: quando escrevo "Lisp" estou me referindo especificamente ao dialeto **Common Lisp**[3]. Os exemplos foram gerados com a implementação **Steel Bank Common Lisp (SBCL)**[4], usando o **Portacle**[5] como editor.

1 Como Lisp funciona?

Para entender corretamente a sintaxe e a semântica de Lisp temos que compreender como as linguagens de programação entendem e executam o código fonte (o texto do programa), e compreender como a Lisp é diferente das outras linguagens nesse aspecto.

De modo geral as linguagens de programação definem um **Processador de Linguagem**, uma "caixa-preta" responsável por ler o texto do código fonte e transformar esse texto em alguma ação. Esse processador de linguagem, essa caixa-preta abstrata, é **única** mas costuma ser constituída por em três partes:

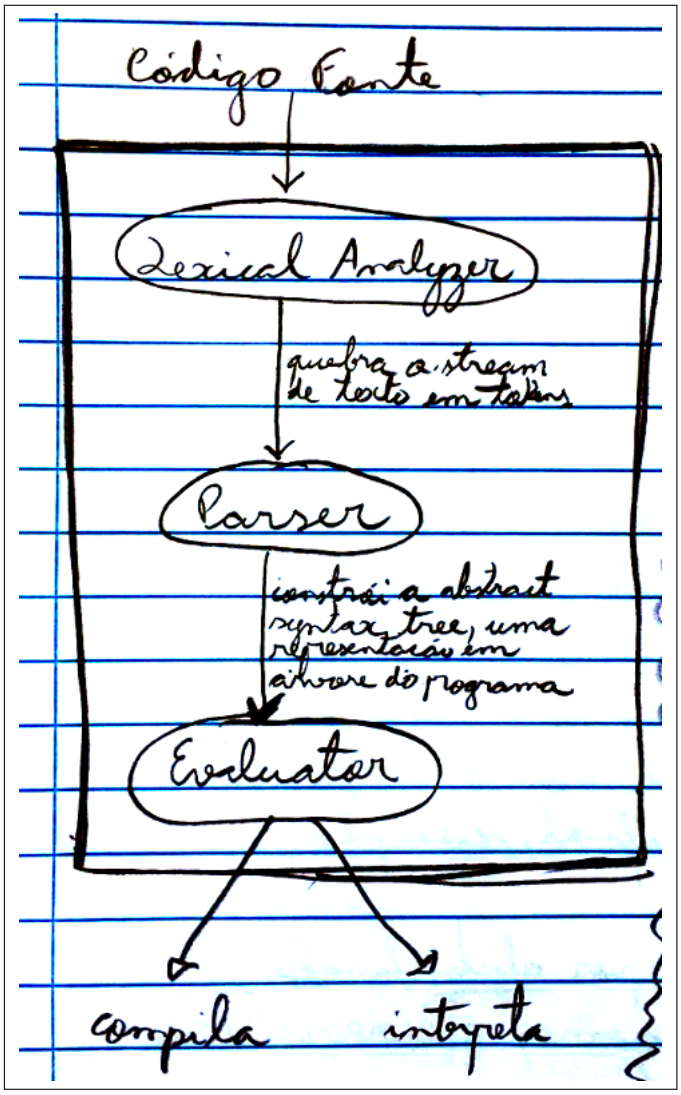
- **Lexical Analyzer**: quebra o strem de caracteres o texto do código fonte em tokens apropriados e encaminha esses tokens para o parser;

[1] http://www.gigamonkeys.com/book/
[2] https://www.amzn.com/1590592395
[3] https://common-lisp.net/
[4] http://www.sbcl.org/
[5] https://portacle.github.io/

- **Parser**: pega os tokens e, de acordo com a gramática da linguagem, constrói uma *estrutura em árvore* que representa as expressões do programa, a *Abstract Syntax Tree* (AST). Essa AST é então enviada ao evaluator; e
- **Evaluator**: recebe a AST e a interpreta diretamente ou realiza uma compilação (dependendo se a linguagem é interpretada ou compilada).

A figura 1 ilustra como podemos entender a caixa-preta única do processador de linguagem:

Figura 1: Processador de Linguagem: caixa-preta única



Importante: como o processador de linguagem é uma caixa-preta, as estruturas de dados utilizadas internamente (tokens, ASTs e outras) são completamente diferentes da estrutura do código fonte em si (o texto do programa) e são de interesse apenas para o desenvolvedor da linguagem.

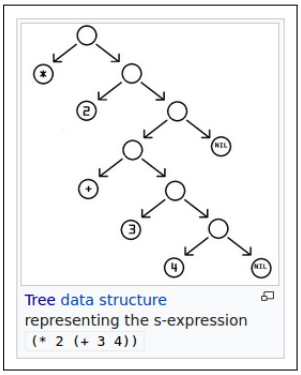
Em Lisp a situação é diferente: existem 2 caixas-pretas distintas: o **Reader** e o **Evaluator**. A primeira caixa-preta é o Reader:

- **READER**: lê o texto do código fonte e transforma esse texto em **Lisp Objects** na forma de *S-expressions* (symbolic expressions[6]). As s-expressions, grosso modo, são uma *notação simbólica para representação de dados em listas aninhadas*, formando assim uma estrutura em árvore para a representação do programa (parecido com a AST).

Note agora o seguinte: as s-expressions não são usadas apenas internamente na criação dos Lisp Objects: **o próprio código fonte em Lisp é escrito em s-expressions**! Aqui temos a primeira distinção importante entre Lisp e outras linguagens de programação: em Lisp a estrutura de dados utilizada internamente pelo processador de linguagem não fica "escondida", pois é a mesma estrutura utilizada pelo programador ao escrever o código fonte: as s-expressions!

Considere o seguinte exemplo: para calcular `2 x (3 + 4)` podemos escrever a seguinte s-expression: `(* 2 (+ 3 4))`. Essa s-expression, no texto do código fonte, é lida pelo READER e transformada em um Lisp Object também representado por uma s-expression (figura 2):

Figura 2: Uma s-expression é uma estrutura em árvore



Fonte: Nate Cull, na Wikipedia[7]

[6] https://en.wikipedia.org/wiki/S-expression

[7] https://en.wikipedia.org/wiki/File:Corrected_S-expression_tree_2.png

Note que uma s-expression pode se referir tanto ao código fonte do programa, quanto a um Lisp Object que foi criado pelo READER.

A segunda caixa-preta no processador de linguagem da Lisp é o Evaluator:

- **EVALUATOR**: recebe os Lisp Objects na forma de s-expressions e decide quais desses Lisp Objects (quais dessas s-expressions) serão interpretadas como **Lisp Forms**, que são *s-expressions que têm significado semântico*. Após decidir quais Lisp Objects são Lisp Forms, o evaluator interpreta os Lisp Forms.

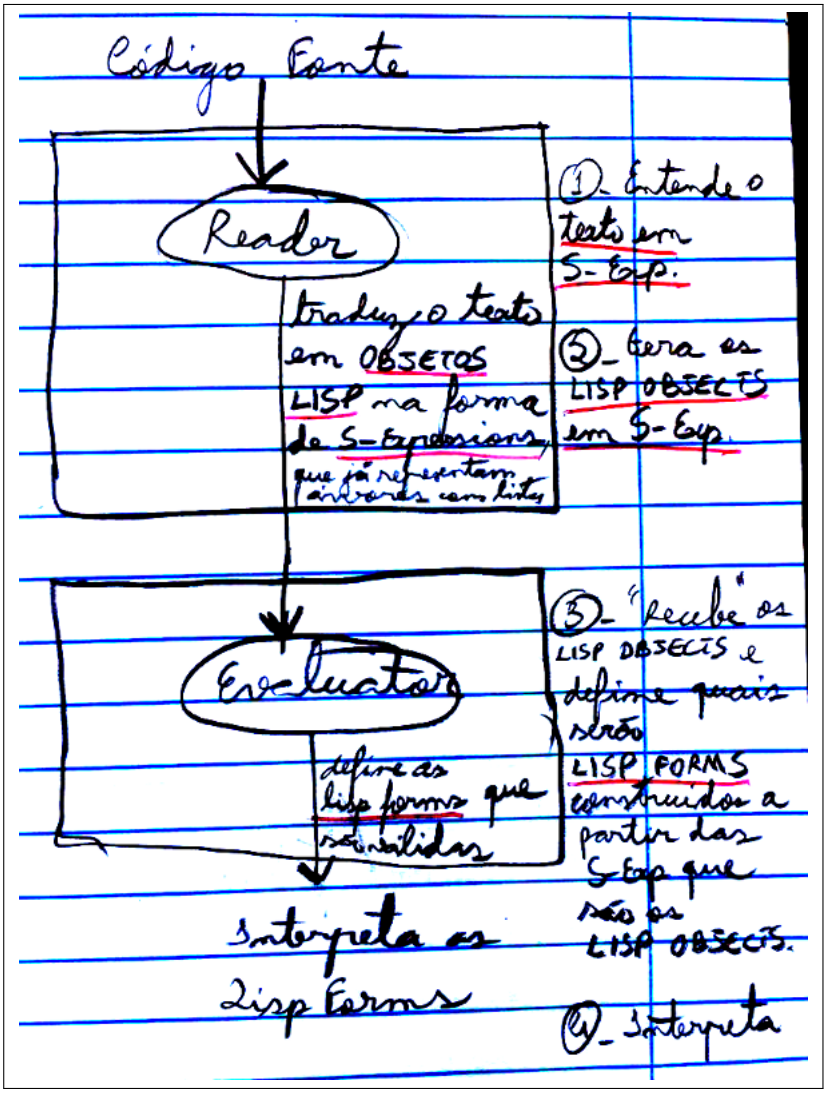
Para entender a diferença entre os **Lisp Objects** e os **Lisp Forms**, note o seguinte: `("somar" 1 2)` é uma s-expression válida que formará um Lisp Object válido, mas *não é* um Lisp Form válido pois uma lista que começa com uma string não tem significado semântico na linguagem. Já a expressão `(somar 1 2)` é tanto um Lisp Object válido como um Lisp Form válido (neste último caso a palavra `somar` é interpretada como um símbolo que representa alguma procedimento, provavelmente para realizar a adição de 1 com 2).

A figura 3, na página 5, ilustra como podemos esquematizar essas duas caixas-pretas do processador de linguagem em Lisp.

É importante notar que:

- Cada nível define sua própria sintaxe:
  - O READER define como as s-expressions que representam o código fonte serão transformadas em Lisp Objects;
  - O EVALUATOR define como os Lisp Forms serão construídos a partir dos Lisp Objects.
- A mesma estrutura é utilizada tanto para escrever o código fonte quanto para representar o programa no processador de linguagem: as s-expressions;
- Como tudo é baseado em s-expressions, você pode utilizá-las como um formato externalizável de dados;
- Como a semântica da linguagem é baseada em s-expressions, então a **semântica é baseada em árvores de objetos** (e não em strings de caracteres como em outras linguagens). Isso faz com que seja muito mais fácil gerar código dentro da própria Lisp, principalmente através da manipulação de dados e códigos já existentes (essa é a base para as Macros em Lisp, uma das características mais importantes dessa linguagem).

Figura 3: Processador de Linguagem em Lisp:
duas caixas-pretas (Reader e Evaluator)



2 A sintaxe do READER

O READER lê as s-expressions do código fonte e as transforma em Lisp Objects, também representados por s-expressions. Mas o que são, afinal, as s-expressions?

Uma s-expression é uma expressão formada por **lists** e/ou **atoms**:

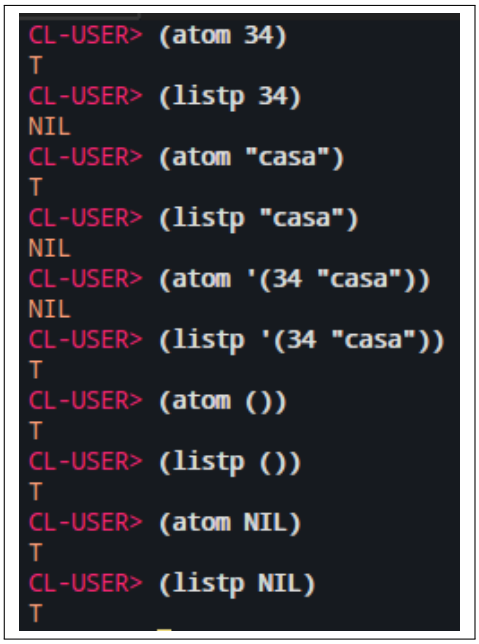
- **List**: é delimitada por parênteses e pode conter um número ilimitado de elementos separados por espaços. Esses elementos podem ser *atoms* ou outras *lists*;
- **Atom:** é qualquer outra coisa (uma string, um símbolo, um número, etc.).

ATENÇÃO: note que pela definição anterior:

- Uma lista *é* uma s-expression
- Um atom *é* uma s-expression
- A lista vazia `()` é tanto um atom quanto uma lista. Observação: a lista vazia pode ser escrita como `NIL`, e é a única representação do valor booleano "falso" em Lisp.

Para checar se alguma expressão é um atom ou uma list, podemos usar as funções `ATOM` e `LISTP`, que retornam verdadeiro (T) ou falso (NIL) se a expressão é, respectivamente, um atom ou uma lista. Alguns exemplos são ilustrados na figura 4.

Figura 4: Atoms e Lists são S-expressions



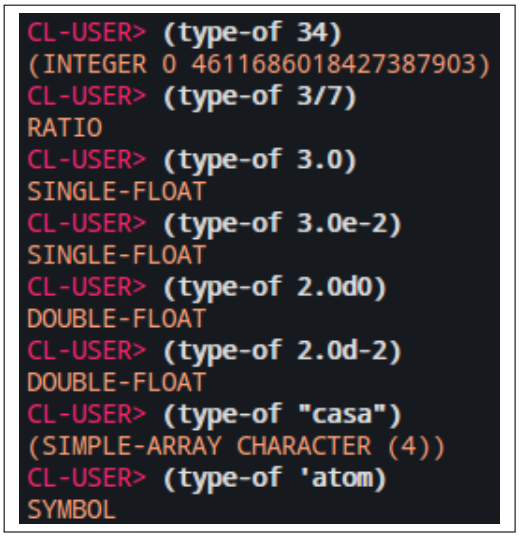
Os atoms mais comuns em Lisp são os **números**, **strings** e **nomes**:

- **Números**: podem ser de diferentes tipos:
  - inteiros: `34`
  - razões (exatos): `1/3`
  - single-floating: `1.0`, `1.0e0`
  - double-floating: `1.0d0`, `1.0d-2`
  - complex
- **Strings**: caracteres entre *aspas duplas*. O caractere \ escapa o próximo caractere.
- **Nomes**: são os identificadores que dão nomes à funções, variáveis, etc. Quase todos os caracteres podem ser utilizados, exceto: espaço, parênteses, aspas simples ou duplas, crase, dois pontos, vírgulas, ponto e vírgula, contra barra e pipe. ATENÇÃO:
  - O READER transforma todos os nomes em objetos chamados de **Symbols**, que representam os nomes/identificadores e dão nomes às coisas, mas os symbols são sempre convertidos em LETRAS MAIÚSCULAS (exceto se escapados com \ ou | |).
  - Ao encontrar um nome o READER primeiro converte em maiúsculas, e depois procura no ambiente do Lisp se já existe um symbol que represente esse nome: se existir ele simplesmente retorna o symbol que já existe; se não existir ele cria um novo symbol e internaliza essa definição no ambiente do Lisp (em uma "tabela" chamada de package).
  - Convenções importantes para nomes:
    - para separar palavras: `calcula-numero`
    ? para testes lógicos: `number?`
    -> para conversões: `de->para`
    * para para variáveis globais: `*idade*`
    + para para constantes globais: `+taxa+`
    % para função: `%funcao`
    %% para função de baixo nível: `%%funcao`
    : para keywords: `:chave`

Para verificar o tipo de uma s-expression podemos usar a função `TYPE-OF` conforme ilustrado na figura 5.

Seguindo as regras de sintaxe acima o READER lê as s-expressions do código fonte e as transforma em Lisp Objects que são passados ao EVALUATOR.

Figura 5: A função `TYPE-OF` retorna o tipo de uma s-expression



3 A sintaxe do EVALUATOR

Depois que o READER leu e transformou as s-expressions do código em Lisp Objects, o EVALUATOR determina quais Lisp Objects têm valor semântico e trata essas expressões como Lisp Forms. Essas Lisp Forms serão posteriormente avaliadas pelo próprio EVALUATOR para retornar algum valor.

A regra sintática que o EVALUATOR utiliza para determinar se um Lisp Object é um Lisp Form válido é a seguinte:

- Todo atom é um Lisp Form válido;
- A lista vazia `()`, ou `NIL`, é um Lisp Form válido;
- Toda lista não vazia é um Lisp Form válido desde que seu primeiro elemento seja:
  - Um *symbol*; ou
  - Uma *expressão lambda*.

Todo Lisp Form é então avaliado pelo EVALUATOR que retorna o valor da Lisp Form.

4 Como o EVALUATOR avalia os Lisp Forms?

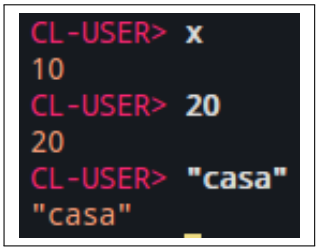
Já vimos que uma Lisp Form é um objeto sob a forma de uma s-expression (Lisp Object) que tem sentido semântico na linguagem. O EVALUATOR avalia cada Lisp Form e retorna algum valor para essa form, de acordo com **três regras principais de avaliação**.

Regra 1: Se a Lisp Form for um **atom**:

- Se o atom for um *symbol*, retorna o valor atual do symbol;
- Se o atom for um *keyword symbol*, define uma constante com esse nome, e o symbol como o valor; e
- Se o atom for outra coisa: ele é um *self-evaluating object*, retorna seu próprio valor.

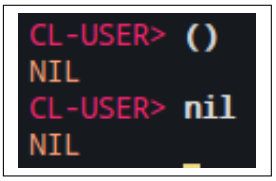
Na figura 6, o atom `X` é um symbol que foi avaliado pelo EVALUATOR: seu valor atual, 10, foi retornado. O inteiro `20` também é um atom, mas não é um symbol, então é *self-evaluating*: o próprio valor do atom é retornado. Por fim, a string `"casa"` também é um atom que não é symbol, então é *self-evaluating* e o próprio valor do atom é retornado.

Figura 6: Avaliação de atoms



Regra 2: Se a Lisp Form for a lista vazia `()`, retorna `NIL` (ver figura 7).

Figura 7: Lista vazia retorna `NIL`



Regra 3: Se a Lisp Form for uma lista não vazia e o primeiro elemento for um symbol, esse elemento será avaliado para determinar se o symbol representa uma **função**, um **operador especial** ou uma **macro**. É importante definir o que o symbol representa pois a avaliação será diferente dependendo se o Lisp Form for uma **function call**, uma **special form** ou uma **macro form**:

- **Function call**: se o symbol representar uma função, a regra de avaliação é a seguinte: avalie todos os elementos restantes na lista, recursivamente, como Lisp Forms, e passe os valores resultantes dessas avaliações como argumentos para a função representada pelo symbol. Note que os argumentos são avaliados recursivamente antes da função ser chamada.
- **Special form**: se o symbol representar um operador especial (existem 25 operadores especiais, como `IF`, `QUOTE` e `LET`) a regra de avaliação é a seguinte: avalie todos os elementos restantes na lista, de acordo com a regra específica de cada operador especial. Dessa forma as Special Form servem para implementar características da linguagem que precisam de algum processamento especial pelo EVALUATOR, diferente das Function Call.
- **Macro form**: se o symbol representar uma macro, a regra de avaliação é a seguinte: em primeiro lugar, os elementos da macro são passados, *não avaliados*, para a função macro; em segundo lugar, a Lisp Form retornada pela função macro (chamada de *expansion*) é então avaliada de acordo com as regras normais de avaliação.
  * Note que uma macro é uma função que pega s-expressions como argumentos não avaliados e retorna um *outro* Lisp Form que é então avaliado no lugar da macro form.
  * Como o EVALUATOR não avalia os elementos antes de passá-los à função macro, eles não precisam ser Lisp Forms válidos (essa é a única exceção na linguagem). Cada macro define sua própria sintaxe local para dar significado às s-expressions não avaliadas que são passadas à função macro.

5 Por fim

Talvez de forma mais exigente do que em outras linguagens, a Lisp realmente OBRIGA que os programadores:

- Entendam a *sintaxe* (tanto do READER quando do EVALUATOR); e
- Entendam de modo cristalino, as *regras de avaliação dos Lisp Forms* (essas regras são a parte mais importante deste texto).

Se as regras de avaliação dos Lisp Forms não estiverem absolutamente claras, programar em Lisp pode ser bem frustrante. Entretanto, uma vez que a sintaxe e as regras de avaliação estejam dominadas e internalizadas, a programação em Lisp é extremamente produtiva!