Scheme: Simplicidade e Eficiência

Geraldo Volpato Calegari Júnior[1], Maria Tereza Nagel[2]
[1,2]Curso de Ciências da Computação – 4ª fase, 2002
Departamento de Informática e Estatística,
Universidade Federal de Santa Catarina (UFSC),
Cx Postal 476, CEP 88040-900, Florianópolis – SC – Brasil.
geraldo@inf.ufsc.br, tereza@inf.ufsc.br

Resumo

Este trabalho é uma apresentação da linguagem funcional Scheme, seus aspectos gerais e pontos fortes. Scheme é uma linguagem que evidencia um estilo de programação diferente das linguagens procedurais, chamado de programação funcional. A programação funcional enfatiza a avaliação de expressões, ao invés da execução de comandos. As expressões nessas linguagens são formadas utilizando-se funções para combinar valores básicos.

Palavras-chave: Programação Funcional, Linguagem, Recursão.

Introdução

A linguagem Scheme é um dialeto do LISP que enfatiza a elegância conceitual e a simplicidade. É definida pelo padrão IEEE P1178, sendo substancialmente menor. Sua especificação tem aproximadamente 50 páginas, quando comparadas às 1300 páginas do Common Lisp. O Scheme é freqüentemente utilizado no estudo de linguagens de programação, devido à sua habilidade em representar várias abstrações de programação com um conjunto bem definido de primitivas.

A Linguagem Scheme

Scheme é um linguagem funcional que foi criada a partir de LISP. Diferentemente de LISP, Scheme é mais simples e fácil de aprender por conter um pequeno grupo de regras e ter a possibilidade de fazer composições dessas regras. A flexibilidade é garantida devido à ausência de restrições, não limitando o poder da linguagem. Logo, não existe a necessidade do Scheme incorporar novas funções para contornar inconvenientes causados pela existência de alguma restrição na linguagem.

Diferentemente das outras linguagens, Scheme utiliza um inovado sistema de recursão denominado "cauda-recursão" o qual considera a recursão como apenas uma chamada de procedimentos sem retorno, passando apenas parâmetros; podendo utilizar uma única área de memória para isso.

O programa Scheme normalmente é um programa interativo o qual pode ser executado em partes. Depois da execução, os dados do programa não desaparecerão, porém aguarda o usuário descrever o que deve ser feito. Tal fato torna a linguagem extremamente segura, uma vez que sistemas interativos normalmente não travam. Além disso, o Scheme detecta possibilidades do sistema trancar, alertando o usuário e aguardando uma ordem para contornar o problema.

A linguagem foi inicialmente criada para fins industriais, uma vez que é fácil de aprender e extremamente poderosa. Atualmente, está sendo usada por algumas universidades, por indústrias em companhias tais como a DEC, TI, Tektronik e Sun.

Histórico

- 1975: A primeira descrição do Scheme, [Scheme 75];
- 1978: Relatório revisado [Scheme 78], descrevendo a evolução da linguagem e posterior implementação a partir do inovado compilador [Rabbit] na MIT;
- 1981 a 1982: Três projetos distintos utilizando variações do Scheme para cursos na MIT [Rees82], Yale [MITScheme], Indiana University [Scheme311];
- 1984 : Um livro texto de introdução a ciência da computação foi publicado [SICP].

Como o Scheme começou a se espalhar, dialetos locais começaram a surgir, havendo divergências de implementação entre um lugar e outro. Com

isso, estudantes e pesquisadores achavam difícil o entendimento de código escritos em outras localidades. Para sanar os problemas e padronizar a implementação do Scheme, os 50 representantes dos maiores implementações do Scheme fizeram uma conferência em outubro de 1984.

- 1985: Relatório [RRRS] da conferência descrita acima foi publica na MIT e Indiana University;
- 1986: Outra revisão da linguagem [R3RS];

1988: O relatório atual reflete revisões aceitas nos encontros que precederam conferencia de Lisp e programação funcional.

Aspectos Gerais

Uma primeira visão que podemos ter de Scheme é como uma calculadora: escrevemos uma expressão e o interpretador Scheme nos dá o resultado. Portanto, nosso primeiro passo é ver como expressões são escritas em Scheme.

Notação Pré-fixada: Este tipo de notação tem uma grande vantagem que é a homogeneidade. Existe basicamente uma única regra que rege a escrita de qualquer expressão. Em Scheme, quando queremos calcular o valor de uma função, escrevemos:

```
(f x)
```

ao invés de f(x), como é usual em matemática. Se a função tiver vários parâmetros, como g(x,y), escrevemos:

```
(g x y)
```

Sintaxe: Tudo que escrevemos em Scheme é chamado genericamente de *expressão S*, ou mais simplesmente *expressão*. As expressões que escrevemos em Scheme podem ser classificadas em dois tipos: *átomos* e *listas*. Os átomos podem ser de três tipos: numerais, textos e *símbolos*. Uma lista, por sua vez, é uma sequência de expressões entre parênteses.

Definições: Uma primeira forma de *abstração* é darmos nomes às coisas. Em Scheme, usamos o operador `define` para isso. Por exemplo,

```
(define x 3)
```

associa x ao valor 3. Também podemos dizer que a *variável* x tem valor 3. A partir daí, quando calculamos o valor da expressão x, o valor resultante será 3. Como o valor definido pode ser usado em qualquer expressão após a execução do define, essas definições são chamadas de *definições globais*.

Uma outra forma de darmos nomes a valores é com a construção `let`. Essa construção permite darmos nomes *locais* a valores. Ao contrário de um define, que define nomes globais, esses nomes só são válidos dentro da expressão onde eles são definidos, não interferindo com outras partes de um programa.

Listas: Em Scheme, o mecanismo básico de estruturação de dados é a *lista*. Listas representam sequências de valores. Como listas também são valores, podemos ter listas como elementos de outras listas; essas listas são chamadas *aninhadas*. Podemos criar listas usando a primitiva `cons` sucessivamente, ou através da primitiva `list` que é a maneira mais indicada.

Definição de Funções: Uma segunda forma de abstração, bem mais poderosa que o uso de nomes, é a definição de *funções*. Em matemática, escrevemos $f(x) = 2x$ para definir uma função *f* que leva um número ao seu dobro. Em Scheme, essa função seria definida como:

```
(define f (lambda (x) (* 2 x)))
```

A construção lambda usada acima serve exatamente para criarmos funções. O construtor lambda atua sobre uma lista com os *parâmetros* da função e um *corpo* que calcula o valor da função. Funções podem ter zero, um, ou mais parâmetros.

Predicados: Existe uma classe de funções em matemática que retornam como valor simplesmente *verdadeiro* ou *falso*, isto é, tais funções apenas testam uma determinada propriedade. Predicados em Scheme são escritos como qualquer outra função, mas retornam valores especiais que representam *verdadeiro* ou *falso*. *Em Scheme, o valor verdadeiro é denotado pelo símbolo #t, e o valor falso pelo símbolo #f.*

Abaixo, exemplos de predicados úteis que o Scheme possui:

- `number?` Testa se um dado valor é um número.
- `symbol?` Testa se um dado valor é um símbolo.
- `list`? Testa se um dado valor é uma lista.
- `even?` Testa se um dado número inteiro é par.
- `odd?` Testa se um dado número inteiro é ímpar.
- `positive?` Testa se um dado número é maior que 0.
- `zero?` Testa se um dado número é igual a 0.

Como predicados são funções, podemos definir novos predicados da mesma forma que definimos funções.

Recursividade

A recursão é um mecanismo poderoso que auxilia o programador de tal forma que procedimentos de difícil controle tornam-se praticamente automáticos. Um procedimento é dito recursivo se ele contém uma chamada a si mesmo. Em Scheme, além da recursão tradicional utilizada pelas linguagens imperativas, pode-se utilizar o conceito de tail recursion (recursão final) que é um tipo de recursão onde, uma vez que a recursão chegou ao seu fim, não é necessário nenhum trabalho adicional para finalizar o procedimento: o resultado já está calculado. Isso ocorre porque, a medida que a função vai se desenrolando, ela não cria operações pendentes, a chamada da função principal vai sendo simplesmente repetida, cada vez com novos parâmetros, até chegar no caso final. O traço de uma função recursiva é feito da mesma forma que o de uma função não recursiva.

Por causa dessas características, funções com recursão final, em geral, são ligeiramente mais eficientes que funções com recursão normal. Além disso, computadores têm um limite para o número de operações pendentes que eles podem manter. Apesar desse limite ser bastante alto (da ordem de 10^3~10^4 operações), podemos ter problemas quando manipulamos listas ou números muito grandes usando recursão normal. Funções que usam recursão final, como não acumulam nada, não sofrem esse limite.

Como funções com recursão final são mais eficientes, surge a questão de quais funções podemos definir desta forma. Algumas funções, são naturalmente definidas sob a forma de recursão final. Outras funções podem ser colocadas nessa forma, através de algumas modificações. Uma técnica bastante importante que podemos usar para colocar uma função no formato de recursão final é o uso de um *acumulador*. Ela consiste em colocarmos um parâmetro extra na função que estamos definindo, e a cada chamada recursiva "acumular" parte do resultado nesse parâmetro, de modo que ao chegarmos no caso base o resultado final é exatamente o valor do acumulador, não restando mais nada a ser feito.

Representação de Dados

Até agora, todas as funções que definimos trabalhavam sobre números e listas. Entretanto, um computador não manipula apenas esses tipos de dados; um computador é capaz de tratar conjuntos, imagens, matrizes, sons, enfim, uma variedade ilimitada de tipos de informação. Como um computador pode manipular todos esses tipos de informação, e como uma linguagem pode nos oferecer todos esses tipos?

Temos aqui uma questão similar a que já encontramos com funções. Lá, a questão era: como uma linguagem pode oferecer todas as funções que precisamos? E a solução era que, ao invés de fornecer todas as funções que precisamos, uma linguagem deve oferecer mecanismos para criarmos novas funções.

Da mesma forma, uma linguagem não pode nos oferecer todos os tipos de dados que precisamos. Ao contrário, mais importante que os tipos básicos que ela oferece são os mecanismos oferecidos para criarmos novos tipos. Uma das tarefas mais importantes de programação é decidirmos como cada informação que precisamos tratar deverá ser representada no computador, através da linguagem que utilizamos. Em Scheme, o principal mecanismo para a construção de novos tipos é a lista. Assim, as listas `(a b)` e `(b a)` podem representar um conjunto *{a,b}*.

Conjuntos: Uma maneira bastante direta para se representar conjuntos em Scheme é através de uma lista de seus elementos. Mais do que armazenar dados, um computador deve ser capaz de manipular esses dados. A operação mais básica sobre conjuntos é o predicado `pertence?`, para verificar se um dado elemento pertence a um conjunto. Sua definição não apresenta dificuldades:

```
(define pertence?
   (lambda (e c)
     (if (null? c) #f
     (or (equal? e (car c))
     (pertence? e (cdr c))))))
```

Tabelas Associativas: Uma outra estrutura de dados muito comum é a *tabela*, também chamada de *catálogo*, *diretório*, *lista associativa*, *tabela de símbolos*, etc, conforme seu uso específico. Esse tipo de estrutura associa um determinado valor, chamado de *chave*, a outros valores que tenham alguma relação específica com a chave. Um exemplo típico de tabela é um catálogo telefônico, que associa a cada nome de pessoa (a chave) um ou mais telefones.

Uma maneira de representarmos uma tabela contendo um "caderno de telefones" é com uma estrutura como mostrada abaixo:

```
(
  (Denise  2221111)
  (Paulo   2741555)
  (Jose    2221133)
  (Maria   2129876)
)
```

Isso é, uma lista, onde cada elemento da lista é outra lista, com um nome (a chave) e o telefone associado ao nome.

Árvores Binárias: Para manipular árvores binárias em Scheme, podemos representar essas árvores através de listas. Uma forma possível para essa representação é descrita a seguir: árvores vazias são representadas pela lista vazia. Árvores não vazias são representadas por uma lista com 3 elementos: o primeiro é a raiz da árvore, enquanto o segundo e o terceiro elementos são listas representando as sub-árvores esquerda e direita. Essa representação é chamada *pré-fixada*.

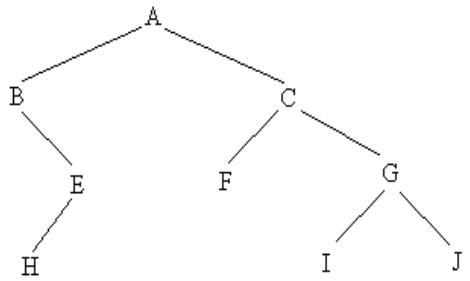

Figura 1 - Árvore binária da figura sem mostrar as sub-árvores vazias.

Usando a representação pré-fixada na árvore da figura 1, temos a lista:

```
(A (B () (E (H () ()) ())) (C (F () ())
       (G (I () ()) (J () ()))))
```

Resultados

Os conhecimentos obtidos durante a elaboração deste, nos permitiram criar alguns exemplos práticos. O interpretador utilizado foi o *Furg Scheme*:

Roleta

```
(define(roleta aposta)
  (sorteio (random 40) aposta))
(define(sorteio numero aposta)
  (if(= numero aposta)
    (display " VOCE GANHOU  ")
    (display " VOCE PERDEU  "))
     numero)
```

Potência

```
  (define(exp m n)
    (if(zero? n)1
      (* m (exp m (- n 1)))))
```

Fibonacci

```
  (define(fibonacci n)
    (if(or(= n 0) (= n 1)) 1
      (+ (fibonacci(- n 1))
      (fibonacci(- n 2)))))
```

Retorna o maior valor absoluto da lista

```
  (define(max_absoluto l)
    (cond(( null? (cdr l)) (car l))
      ((> (abs(car l))
      (abs(car(cdr l))))
      (max_absoluto(cons(abs(car l))
      (cdr(cdr l)))))
    (else(max_absoluto(cdr l)))))
```

Conclusões

A linguagem funcional Scheme é um dos dialetos de LISP, que tem por principal qualidade a simplicidade. Possui todos os aspectos das linguagens funcionais, como a sintaxe simples, flexibilidade, etc. Usa recursão com o conceito de tail recursion (recursão final) que é muito poderoso, pois não se preocupa com o limite de operações. A representação das estruturas de dados tais como: conjuntos, tabelas associativas e árvores binárias é feita através da manipulação de listas.

Referências

[1] Shermers.org, *an improper list of Scheme resources.* URL: http://www.schemers.org/

[2] Página Pessoal, *linguagem Scheme* URL: http://aton.inf.ufrgs.br/~cicero

[3] Grotta 's Home Page, *linguagens de programação*. http://www.geocities.com/researchtriangle/