Trabalho de Programação Funcional
Linguagem – Scheme

Fábio Davanzo F. de Oliveira, Daniel C. Santos

Bacharelado em Ciências da Computação – quarta fase, 2002
Departamento de Informatica e Estatística (INE)
Universidade Federal de Santa Catarina (UFSC), Brasil, 88040-900
Fone (48) 234-5441, (48) 334-2968
fabiodfo@zipmail.com.br, dcsantos@inf.ufsc.br

Resumo

Pretendemos com esse artigo demonstrar os principais tópicos da Programação Funcional e da linguagem Scheme descrevendo seus conceitos básicos, Estruturas dos programas, Expressões, Recursão, (Duplas (pares), listas e símbolos), Procedimentos Padrão e por fim demonstraremos alguns exemplos.

Palavras-chave: Scheme, LISP, Expressões, Recursão e Linguagem Funcional.

Abstract

We pretended with this article demonstrate the main topics about Funcional Program and Scheme linguage describing its basics concepts, programm estrutures, expressions, (double(pair),lists and symbols), standard procedure, and finally some examples.

Key-words: Scheme, LISP, Expressions, Recursion and Functional Linguage

Introdução

Scheme[1,2,3] 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. 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. A linguagem foi inicialmente criada para fins industriais, uma vez que é fácil de aprender e extremamente poderosa.
Neste artigo abordaremos os seguintes tópicos[1]: Historico, Conceitos básicos, Estruturas dos programas, Expressões, Recursão, (Duplas (pares), listas e símbolos), Procedimentos Padrão e Exemplos.

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. 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.

Conceitos Básicos

Scheme como o Lisp utiliza os conceitos de notações pré-fixada e calculo lambda.
Qualquer identificador que não é uma palavra chave sintática, pode ser usado como uma variável. Uma variável pode dar nome a uma locação onde um valor pode ser armazenado. Diferente da maioria dos outros dialetos de Lisp, exceto Common Lisp, Scheme é uma linguagem estendida estaticamente com blocos de estruturas. Qualquer valor em Scheme pode ser usado como um valor boolean para o propósito de um teste condicional.
O armazenamento de variáveis e objetos tais como pairs, vetores e strings implicitamente denotam locações e seqüências de locações. Uma string, por exemplo, denota tantas locações quanto há caracteres na string. Um novo valor pode ser armazenado dentro de uma destas locações mas mesmo assim a string continua a denotar a mesma locação de antes.

Estruturas dos Programas

Definições de topo:
No topo de um programa, uma definição (define <variável> <expressão>) tem essencialmente o mesmo efeito que a expressão de atribuição (set! <variável> <expressão>) se a variável esta associada. Se <variável> não esta associada, então a definição ira associa-la a uma nova localização antes de efetuar a atribuição.
Ex.:

==> (define add3(lambda (x) (+ x 3)))
==> (add3 3)
6

Definições Internas:
Algumas implementações Scheme permitem a ocorrência de definições no inicio de um <corpo> qualquer. Assim, todas as variáveis ai definidas são locais ao <corpo>. Dessa maneira, variáveis são definidas ao invés de serem associadas(com ambiente igual ao de todo o <corpo>).
Ex.:

==> (let ((x 5))
(define foo (lambda (y) (bar x y)))
(define bar (lambda (a b) (+ (a b) a)))*
(foo (+ x 3)))

Expressões

Uma expressão é uma construção que retorna um valor, sendo que o seu tipo pode ser categorizado como Primitivo ou Derivado. Expressões de tipo Primitivo incluem variáveis e chamadas de procedimentos, enquanto que tipos Derivados são construções de tipos Primitivos.
As expressões do tipo Primitivo são: Referencias a variáveis, Expressões literais, Chamadas de procedimentos, Expressões Lambda, Condicionais, Atribuições.
As expressões do tipo Derivada são: Condicionais, Construções de associação, Sequenciamento(de expressões), Iteração(de expressões), Execução Atrasada.
Exemplos das principais Expressões e Procedimentos.

Nomeando coisas com define:
(define a 10)
(define (dobro x) (2 x))*
Neste exemplo, primeiro associamos a a (variável) o valor 10, depois associamos a função dobro, que recebe x como parâmetro, o resultado que corresponde a 2*x.

Definindo procedimentos:
(define (nome arg1 arg2...) corpo)
Note que *define* é seguida do nome do procedimento e de seus argumentos (note também que o número de argumentos pode ser igual a zero!). O restante, o corpo do procedimento, especifica como o procedimento irá ser avaliado.

Definições temporárias: let
Observe a implementação da fórmula de Báskara:
(define (square x) (* x x))
(define (raizes a b c)

(display (/ (+ (- 0 b) (sqrt (- (square b) (4 a c)))) (* 2 a)))*
(newline)
(display (/ (- (- 0 b) (sqrt (- (square b) (4 a c)))) (* 2 a))))>*

Não é preciso ser especialista em programação para perceber que *(sqrt (- (square b) (* 4 a c)))) (* 2 a)))* está sendo calculado igualmente duas vezes. Para contornar esse problema, podemos

utilizar a seguinte forma:

```
(define (raizes a b c)
    (let ((raiz (sqrt (- (square b) (* 4 a c)))))
        (display (/ (+ (- 0 b) raiz)(* 2 a)))
        (newline)
        (display (/ (- (- 0 b) raiz) (* 2 a)))))
```

Expressões condicionais.

Expressão if:

estrutura de uma expressão *if* é como segue: *(if predicado conseqüência alternativa)* Scheme primeiramente avalia o predicado. Se o resultado for #t (true), então ele avalia a expressão conseqüência e retorna seu valor como sendo o valor de toda a expressão *if*. Porém, se o resultado for #f (false), então ele avalia a expressão alternativa e retorna o seu valor.

```
(define (duplica x)(* 2 x)) (if (= 1 1) 2 3)
(if (negative? 4) (* 3 6) (/ 10 2))
(if 1 2 3)
(if (= (+ 12 4)(duplica 8))(square 5)(square 7))
```

Estas expressões retornam 2, 5, 2 e 25, respectivamente. Lembre que square foi definido anteriormente.

Recursão

Um procedimento é dito recursivo se ele contém uma chamada a si mesmo. A recursão é um mecanismo poderoso que auxilia o programador de tal forma que procedimentos de difícil controle tornam-se praticamente automáticos. Em Scheme, além da recursão tradicional utilizada pelas linguagens imperativas, pode-se utilizar o conceito de *tail recursion* (recursão pela cauda). Não entraremos em grandes detalhes, mas, trocando em miúdos, *tail recursion* 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.

```
(define fatorialA x)
    (cond ((= x 0) 1)
        (else (* x (fatorialA (-1+ x))))))

(define fatorialB x) (aux x 1))
(define (aux cont result)
    (cond ((= cont 0) result)
        (else (aux (-1+ cont) (* cont
        result)))))
```

O primeiro exemplo, fatorialA, utiliza recursão tradicional, já o segundo, fatorialB, utiliza *tail recursion*.

Duplas (pares), listas e símbolos

Duplas

Informalmente, uma dupla (*pair*) é um objeto composto por duas partes, nas quais dois outros objetos são *colados* juntos. Duplas são criadas com a primitiva *cons*.

```
(define dupla (cons 3 4))
```

O exemplo acima associa a *dupla* a dupla (3 . 4). Observe os espaços antes e depois do ponto, pois essa é a diferença entre a notação de uma dupla e um número real.
Para acessar o primeiro e o último elementos de uma dupla, usamos as primitivas *car* e *cdr*,respectivamente.

```
(car dupla)
3
(cdr dupla)
4
```

Listas

Essencialmente, uma lista é uma série de duplas ligadas juntas, terminada com o objeto especial *nil*. Isto significa que podemos (mas não devemos!) criar listas usando a primitiva *cons* sucessivamente:

```
(defina lista (cons 1 (cons 2 (cons 3 nil))))
```

Note que não se trata de um método amigável. A melhor maneira de se definir uma lista é através da primitiva *list*.

```
(define lista (list 1 2 3))
```

Veja como é possível inserir elementos no início de uma lista:

(cons x dupla)

Insere o objeto **x** antes da lista **dupla**, criando uma nova lista. Para acessar o objeto x:

(car dupla)

Para acessar o primeiro elemento da lista original:

(cdr dupla)

Símbolos

Uma forma de representar as *strings* é colocando a string desejada entre aspas duplas ("isso é uma string"). As strings geralmente são utilizadas em Scheme quando se deseja imprimir alguma mensagem na tela. Porém, quando se deseja comparar nomes, como por exemplo, "João" é igual a "Maria"?, utilizamos símbolos em vez de strings. Quando utilizamos símbolos (precedendo o símbolo com um apóstrofe '), estamos dizendo ao interpretador para não avaliar os objetos, por exemplo:

(define a 10)

a
10
'a
A

Neste exemplo, quando *a* foi digitado, dissemos ao interpretador que ele devia avaliar *a*, e obtemos como resposta *10*. Porém, quando digitamos *'a*, dissemos ao interpretador Scheme que ele não devia avaliar o seu valor, mas sim, simplesmente "exibir" o objeto.

(car '(a b c))

O exemplo acima retorna o símbolo a.

Procedimentos Padrão

São procedimentos que já vem pronto quando se inializa Scheme e normalmente estão associados a procedimentos primitivos como abs, operador + e outros.
Assim, vemos os seguintes grupos:

Boolean
Objeto padrão que assume valor #t ou #f que é usado em expressões condicionais.
Predicados de Equivalência:O procedimento eqv? verifica se dois objetos são geralmente tratados como se fossem o mesmo objeto, retornando #t ou #f dependendo disso.

Tuplas e Listas
Uma lista é definida através de uma tupla onde se tem a cabeça e o rabo que podem ser adquiridos pelos procedimentos car e cdr e concatenados pelo procedimento cons. Existem demais operações com listas que não serão descritas.

Símbolos
Muito úteis por serem utilizados como identificadores, uma vez que dois símbolos são iguais apenas quando escritos da mesma maneira. O Scheme acaba utilizando símbolos para propósito interno também.

Números
Foram adotados os tipos number, complex, real, rational, integer. Existe um distinção entre um número exato e inexato.

E demais grupos que não serão descritos, mas mencionados:
Caracteres, Strings, Vectors, Control Features, Input and OutPut.

Exemplos

Fatorial
```
(define fatorial (lambda (n) (if (zero? n) 1 (* n (fatorial (- n 1))))))
```

Calcula comprimento de uma lista
```
(define comprimento (lambda (l) (if (null? l) 0 (+ 1 (comprimento (cdr l))))))
```

Retorna o n-ésimo termo de uma lista
```
(define n-esimo (lambda (n l) (if (= n 1) (car l) (n-esimo (- n 1) (cdr l)))))
```

Soma todos os elementos de uma lista
(define soma (lambda (l a) (if (null? l) a (soma (cdr l) (+ a (car l))))))

MDC (Máximo Divisor Comum)
(define mdc (lambda (a b) (cond ((zero? a) b) ((zero? b) a) ((>= a b) (mdc (- a b) b)) (else (mdc a (- b a))))))

Conclusão

Com este artigo podemos nos familiarizar com scheme que demonstrou ser uma poderosa linguagem de programação funcional e não apresenta grande complexidade o que facilita sua aprendizagem.

Referências

[1] http://www.inf.puc-rio.br/~roberto/icc/texto/indice.html

[2] http://www.schemers.com

[3] http://www.scheme.org/