PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

Trabalho de Linguagem de Programação - Linguagem Scheme*

Gabriel Luciano Gomes[1]
Geovane Fonseca de Sousa Santos[2]
Isabelle Hirle Alves Langkammer[3]
Luigi Domenico Cecchini Soares[4]

*Artigo apresentado ao Instituto de Ciências Exatas e Informática da Pontifícia Universidade Católica de Minas Gerais como pré-requisito para obtenção do título de Bacharel em Ciência da Computação.

[1]Aluno, Ciência da Computação, Brasil, glgomes@sga.pucminas.br.
[2]Aluno, , Ciência da Computação, Brasil, , geovane.fonseca@sga.pucminas.br.
[3]Aluno, , Ciência da Computação, Brasil, , isabelle.langkammer@sga.pucminas.br.
[4]Aluno, , Ciência da Computação, Brasil, , luigi.domenico@sga.pucminas.br.

Sumário

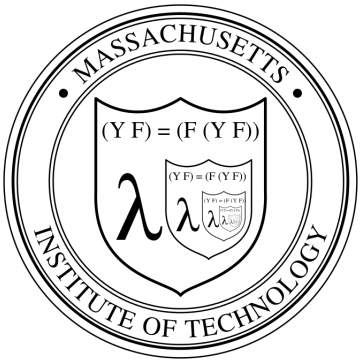
Lista de Figuras

1 INTRODUÇÃO

Scheme é uma linguagem de programação desenvolvida por Guy Steele e Gerald Sussman em meados dos anos 1970, no laboratório de Inteligência artificial do MIT. É uma versão estática e recursiva de LISP, sendo definida por alguns autores como um dialeto de tal linguagem. Além disso, apresenta como característica o fato de ser multiparadigma, tendo como base o cálculo lambda. Apesar de ser uma linguagem poderosa, possui sintaxe simples e com poucas regras, sendo, portanto, bem adequada para aplicações educacionais, como cursos sobre programação funcional, e para introduções a programação.

É a primeira linguagem a suportar macros higiênicas, que permitem que sua sintaxe estruturada em blocos possa ser estendida. O ultimo padrão oficializado é o R6RS, revisado em 2007. Existem diversas implementações da linguagem, tais como: Gambit, JScheme, RScheme. Scheme foi uma das primeiras linguagens de programação a incorporar o cálculo lambda na sua classe principal e o símbolo lambda foi adotado pela linguagem como sua marca visual, como por exemplo na logo do MIT Scheme (figura 1).

Figura 1 – Logo Scheme MIT GNU



Fonte: Wikipedia - MIT/GNU Scheme
Disponível em: <https://en.wikipedia.org/wiki/MIT/GNU_Scheme>;. Acesso em abr. 2018.

Outro ponto a ser considerado diz respeito as influências no desevolvimento de Scheme, que tem LISP e ALGOL como sendo suas principais influenciadoras. LISP forneceu sua sintaxe e semântica, enquanto ALGOL forneceu o uso de estrutura de blocos e escopo léxico. Cabe dizer que a influência de LISP nas características presentes na linguagem é tão marcante que boa parte das funções desenvolvidas em Scheme requerem apenas pequenas modificações para serem reescritas como funções LISP.

Por fim, vale ressaltar que ela teve influência no desenvolvimento das mais variadas linguagens. Como exemplo, podem ser citadas: Ruby, Javascript, GO, Haskell, R, Racket,

Common LISP, Lua, Dylan, MOO, Joy, K, Oaklisp, Ferite, Q, T, Sather, Scala, Cadence SKILL, ECMAScript, AmbientTalk, Mudlle, EuLISP, Clojure, Objeck e Kernel.

2 HISTÓRICO

A linguagem Scheme começou a ser desenvolvido entre os anos 1975 e 1980 por Guy Steele e Gerald Sussman no laboratório de Inteligência Artificial e Ciência da Computação do Massachusetts Institute of Technology (MIT). Foi criada a partir da linguagem LISP com o intuito de estudar a teoria dos atores de Carl Hewitt.

O modelo de ator foi proposto em 1973 por Carl Hewitt, Peter Bishop e Richard Steiger. Eles propuseram uma nova classe especial de objetos, chamado de Atores que são unidades autônomas dentro de uma rede que são ativados pelo recebimento de mensagens. A resposta do ator a essa mensagem pode enviar um número finito de mensagens, criar um número finito de atores ou decidir o comportamento para a próxima mensagem recebida. Esse padrão pode ser visto atualmente em Redes neurais, no qual um nó tem uma ligação, uma ou mais entradas e uma ou mais saídas.

Guy Steele e Gerald Sussman tiveram dificuldades para entender o modelo de Hewitt então resolveram implementar um interpretador de LISP para entender melhor sobre o problema. Adicionando alguns mecanismos de criação de atores e envio de mensagens ao programa, criaram assim uma nova linguagem inicialmente chamada de Schemer. Como o sistema operacional Incompatible Timesharing System (ITS) do MIT tinha limitação de caracteres, o nome passou a ser Scheme.

O primeiro relatório de Scheme, divulgado em 1978, descreveu a evolução da linguagem e sua implementação no compilador Rabbit do MIT, ficando conhecido como Scheme 78. A linguagem ficou conhecida e se espalhou, apresentando novos dialetos e assim começaram a ter diversas divergências de implementação entre diferentes locais. Com intuito de padronizar a implementação, 50 representantes das maiores implementações feitas em Scheme se reuniram em uma conferência em outubro de 1984, e no ano seguinte publicaram a Revised n Report on the Algorithmic Language Scheme (RnRS).

Scheme possui dois padrões, o oficializado em 1998 pelo Institute of Electrical and Electronics Engineers (IEEE) e o Revised n Report on the Algorithmic Language Scheme (RnRS). O n na sigla RnRS define o número de revisões realizadas, padrão altamente implementado é R5RS, realizado em 1998 e o último relatório conhecido é o R6RS feito em 2007.

2.1 Autores

Guy Lewis Steele Jr. nasceu em 2 de outubro de 1954 em Missouri, Estados Unidos. Em 1975 se tornou bacharel em matemática aplicada em Harvard, depois se tornou Mestre em 1977 e Ph.D do MIT em Ciência da Computação em 1980.

Gerald Jay Sussman nasceu em 8 de fevereiro de 1947 nos Estados Unidos. Em 1968 se tornou Bacharel em matemática pelo MIT, depois em 1973 continuou seus estudos e obteve seu Ph.D..

2.2 Influência

O desenvolvimento da linguagem foi influenciado por LISP, ALGOL e pelo cálculo lambda. LISP foi criada por John McCarthy em 1958 no MIT e estava intimamente ligado à comunidade de pesquisa em Inteligência Artificial. Influenciou Scheme, fornecendo sua sintaxe e semântica.

Algol 58 foi desenvolvido por um comitê de cientistas da computação na reunião ETH Zurich em 1958. Em 1960 teve uma revisão da linguagem e ficou comumente conhecida como ALGOL. Introduziu o uso de estrutura de blocos e escopo léxico, sendo considerada uma linguagem à frente do seu tempo.

Durante o desenvolvimento da linguagem, Sussman e Steele trabalharam em ideias sobre o cálculo lambda e outros conceitos avançados de programação, publicando uma série de memorandos influentes e relatórios técnicos publicados pelo MIT AI Lab que ficaram conhecidos como "Lambda Papers". Algumas publicações notáveis são:

- AI Memo 349 (1975) - Scheme: An Interpreter for Extended Lambda Calculus
- AI Memo 349 (1975) - Scheme: An Interpreter for Extended Lambda Calculus
- AI Memo 353 (1976) - Lambda: The Ultimate Imperative
- AI Memo 379 (1976) - Lambda: The Ultimate Declarative
- AI Memo 443 (1977) - Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO
- AI Memo 453 (1978) - The Art of the Interpreter of, the Modularity Complex (Parts Zero, One, and Two)
- AI Technical Report 474 (1978) - RABBIT: A Compiler for SCHEME
- AI Memo 514 (1979) - Design of LISP-based Processors, or SCHEME: A Dielectric LISP, or Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode

3 PARADIGMAS

Scheme é uma linguagem de programação multiparadigma, ou seja, é baseada em mais de um paradigma ao mesmo tempo. Como exemplo, podemos observar o suporte a programação imperativa, funcional e orientação à objetos.

3.1 Imperativo:

Quanto ao seu paradigma imperativo, podemos pensar que há contradição em relação a também ser funcional. Porém, a linguagem implementa as duas abordagens em seu conceito, de forma a coexistirem em seu modelo computacional. Como Scheme é usado em aplicações reais, como o ambiente WEB, não há como fugir de recursos imperativos por causa da demanda da funcionalidade no domínio.

Os principais comandos que definem o paradigma, são:

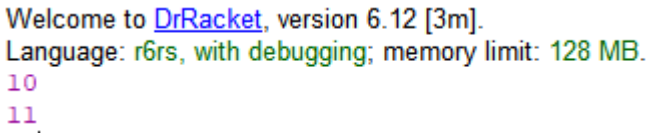
- Construtores Imperativos:
  - O mais importante construtor imperativo em Scheme é a atribuição **set!** (Figura 2);
  - construtor **(begin e1 ... en)**;
  - A estrutura de controle **do**;
  - Procedimentos de Entrada e Saída;
  - Procedimentos de mutadores de lista, string e vetor.
- Procedimentos de mutadores de Lista:
  - O comando **(set-car! x y)** modifica o valor da posição do primeiro elemento de um par ou de uma lista referenciada por x;
  - O comando **(set-cdr! x y)** modifica o valor da cauda da lista ou do segundo elemento de um par referenciados por x;
  - Usar mutadores de lista é possível fazer estruturas circulares.

3.1.1 Exemplo relacionado ao paradigma imperativo:

```
1 (define x 10)
2 (display x)
3 (newline)
4
```

```
5 (set! x (+ x 1))
6 (display x)
7 (newline)
```

Figura 2 – Efeito colateral - Scheme



Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

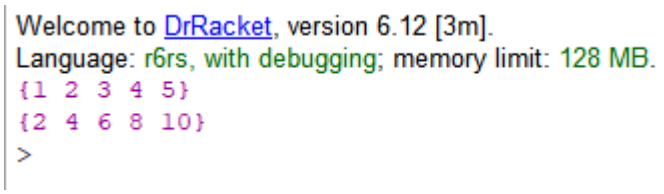
3.2 Funcional:

Em relação ao paradigma funcional, a principal característica de Scheme a implementar esse padrão é a técnica de Currying, onde uma função com n parâmetros é transformada em n funções com um único parâmetro que são equivalentes a original. Outras características são as funções que são consideradas como variáveis de primeira classe e dessa forma podem ser passadas como parâmetros para outras funções.

3.2.1 Exemplo relacionado ao paradigma funcional:

```
1  (define (map function lst)
2    (cond ((null? lst) '())
3          (else (cons (function (car lst))
4                      (map function (cdr lst))))))
5
6  (define (double x) (* 2 x))
7  (define lst (list 1 2 3 4 5))
8
9  (display lst)
10 (newline)
11
12 (display (map double lst))
13 (newline)
```

Figura 3 – Função de mapeamento de uma lista - Scheme



Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

3.3 Orientado à Objetos:

Scheme, diferente de outras linguagens como Java e C++, não dá ao programador a liberdade de se definir novos tipos ou classes. Dessa forma, a orientação à objetos aparece de forma indireta, levando em consideração a possiblidade de se interpretar uma *closure* como um objeto. Tal interpretação se mostra válida devido a duas caractéristicas presentes na linguagem: o status de primeira classe das funções e o uso de amarração estática das chamadas variáveis livres.

Ainda que seja possível se definir uma classe através de certas funções, ter somente isso em mãos torna a manipulação dessa abstração complicada. Para facilitar, pode-se construir algumas funções auxiliares externas à classe, tanto para instanciação da mesma quanto para a troca de mensagens (característica intrínseca ao paradigma orientado à objetos). Assim, é possível determinar um template a ser usado para tais abstrações.

3.3.1 Template de uma classe - Scheme:

```
1  (define (new-instance class . parameters)
2    (apply class parameters))
3
4  (define (send message object . args)
5    (let ((method (object message)))
6      (cond ((procedure? method) (apply method args))
7            (else (error "Error in method lookup " method)))))
8
9  (define (class-name construction-parameters)
10  (let ((instance-var init-value)
11        ...)
12
13    (define (method parameter-list)
14      method-body)
```

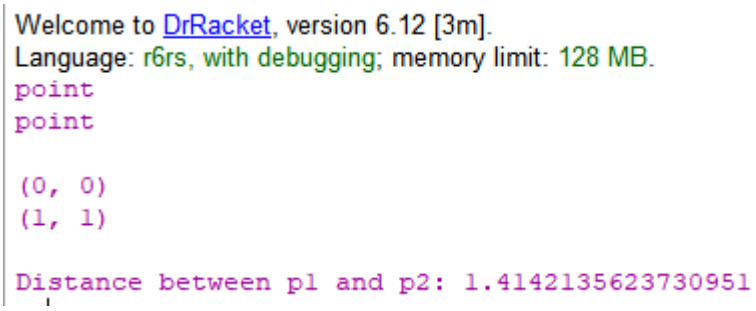
```
15
16    ...
17
18    (define (self message)
19      (cond ((eqv? message selector) method)
20        ...
21
22        (else (error "Undefined message " message))))
23
24    self))
```

3.3.2 *Exemplo de aplicação do template de uma classe:*

```
1  (define (new-instance class . parameters)
2    (apply class parameters))
3
4  (define (send message object . args)
5    (let ((method (object message)))
6       (cond ((procedure? method) (apply method args))
7             (else (error "Error in method lookup " method)))))
8
9  (define (point x y)
10   (let ((x x)
11         (y y))
12
13       (define (getx) x)
14
15       (define (gety) y)
16
17       (define (distance-from p2)
18         (let ((p2-x (send 'getx p2))
19               (p2-y (send 'gety p2)))
20             (sqrt (+ (expt (- x p2-x) 2) (expt (- y p2-y) 2)))))
21
22       (define (type-of) 'point)
23
24       (define (point->string)
25         (string-append "(" (number->string x) ", "
              (number->string y) ")\n"))
26
27       (define (self message)
```

```
28          (cond ((eqv? message 'getx) getx)
29                ((eqv? message 'gety) gety)
30                ((eqv? message 'distance-from) distance-from)
31                ((eqv? message 'type-of) type-of)
32                ((eqv? message 'point->string) point->string)
33                (else (error "Undefined message " message))))
34
35        self))
36
37  (define p1 (new-instance point 0 0))
38  (define p2 (new-instance point 1 1 ))
39  (display (send 'type-of p1))
40  (newline)
41  (display (send 'type-of p2))
42  (newline)
43  (newline)
44  (display (send 'point->string p1))
45  (display (send 'point->string p2))
46  (newline)
47  (display "Distance between p1 and p2: ")
48  (display (send 'distance-from p1 p2))
49  (newline)
```

Figura 4 – Demonstração da construção de uma classe - Scheme



Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

4 CARACTERÍSTICAS

4.1 Características de Família LISP:

1. Homoicônico:

    - Quando você escreve um programa LISP, a gramática utilizada é semelhante ao que

o interpretador visualiza.

- Aninhamento de estruturas de listas, símbolos e literais, podem ser gerados e modificados na própria linguagem.
- Permite ao código LISP gerar listas de estruturas que podem ser executadas como LISP.
  - Executada em tempo de compilação, por meio de macros
  - Executada em tempo de execução, por meio de verificações.
- Estrutura de código pode interferir no compreendimento do mesmo.

2. Macros:

   - Uma função que transforma Uma Árvore Sintática Abstrata em um novo código em tempo de compilação.
   - Podem ser utilizados para criar novos comandos e estruturas de controle.
   - Utilizados para otimizar códigos em tempo de compilação, semelhante ao esquema de *"Template metaprogramming"* em C++

4.2 Exemplo de uma macro - Scheme

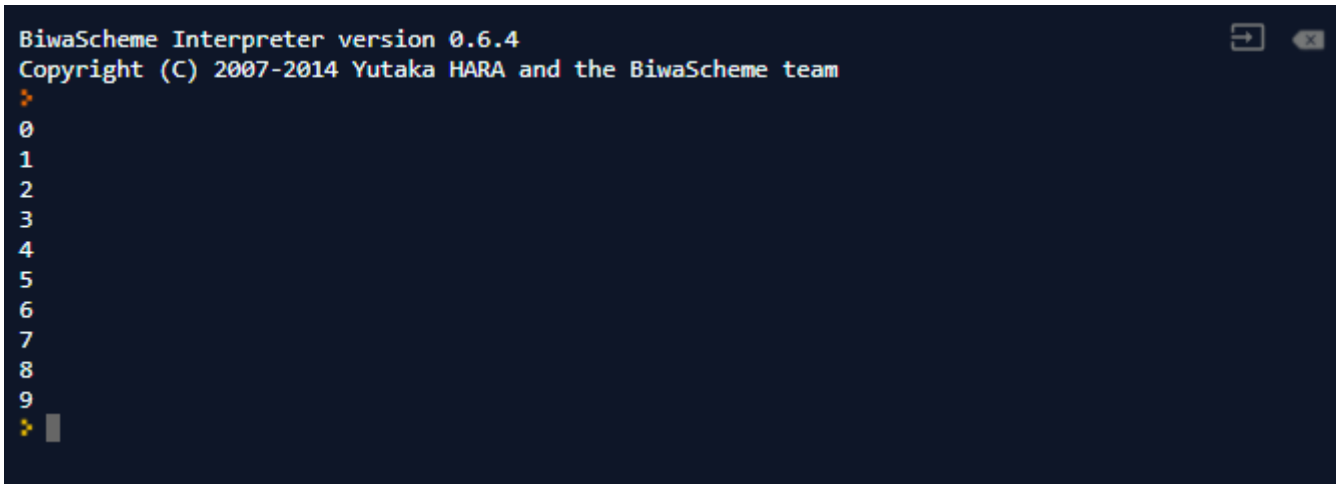
```
1  (define-macro (while condition . body)
2    `(let loop ()
3       (cond (,condition
4              (begin . ,body)
5              (loop)))))
6
7  (let ((i 0))
8    (while (< i 10)
9           (display i)
10          (newline)
11          (set! i (+ i 1))))
```

3. Condições LISP e Restart

   - Permite comunicação bidirecional entre diferentes partes de chamada a pilha. Mais poderoso que as exceções, permite desalocar a pilha, como também aloca-la novamente, através do comando *"Restart"*.

4. *Call with current continuation*

Figura 5 – Comando while construído a partir de uma macro - Scheme



Fonte: *print screen* do terminal referente ao interpretador *BiwaScheme*, no site <http://repl.it>

- Permite salvar o estado atual do programa em uma variável e restaurá-lo mais tarde, diversas vezes. Podendo implementar operadores complexos de controle de fluxo com essa função.

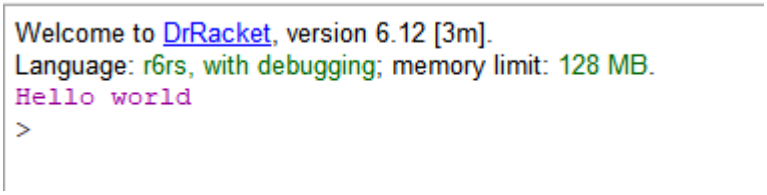
4.3 Características de Scheme

1. *Quote, quasiquote, unquote* e *unquote-splicing*

   Scheme possui uma *syntax* conveniente para a representação de literais. Para tal representação, basta prefixas uma expressão com uma aspas simples. Dessa forma, esta expressão não será avaliada mas sim retornada como um dado.

```
1 (define str1 (quote Hello))
2 (define str2 'world)
3 (display str1)
4 (display " ")
5 (display str2)
```

Figura 6 – *Quote* - Scheme



Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

Entretanto, em alguns casos, o programador não deseja que uma lista inteira de ex-

pressões deixe de ser avaliada. Pensando nisso, a linguagem fornece uma outra syntax, que utiliza do operador ‘ (*backquote*), denominada *quasiquote*. Essa forma, por si, só não apresenta diferenças em relação ao *quote*. Porém, se utilizada em conjunto com vírgulas (chamadas de *unquote*) ou em conjunto com os operadores @ (*unquote-splicing*), as expressões prefixadas por esses operados passam a ser avaliadas. A diferença entre os operadores de *unquote* e *unquote-splicing* está no fato de que, ao utilizar o operador @, a expressão avaliada produz uma lista que, por sua vez, tem seus elementos unidos como múltiplos valores.

```
1 (define str1 (quote Hello))
2 (define str2 '( world))
3 (define str3 `(,str1,str2))
4 (define str4 `(,str1,@str2))
5 (display str3)
6 (newline)
7 (display str4)
```

Figura 7 – *Quasiquote, unquote* e *unquote-splicing* - Scheme

```
Welcome to DrRacket, version 6.12 [3m].
Language: r6rs, with debugging; memory limit: 128 MB.
{Hello {world}}
{Hello world}
>
```

Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

2. Tipagem Forte

Scheme apresenta como uma de suas características o fato de ser uma linguagem **fortemente tipada**. De acordo com Sebesta (2009, p. 328):

> Uma linguagem de programação é **fortemente tipada** se erros de tipo são sempre detectados. Isso requer que os tipos de todos os operandos possam ser determinados, em tempo de compilação ou em tempo de execução. A importância da tipagem forte está na habilidade de detectar usos incorretos de variáveis que resultam em erros de tipo. Uma linguagem fortemente tipada também permite a detecção, em tempo de execução, de usos de valores de tipos incorretos em variáveis que podem armazenar valores de mais de um tipo.

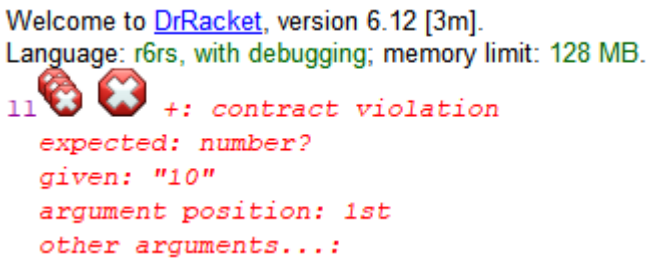
Uma função que receba um parâmetro e realize uma operação matemática em cima do argumento recebido, em Scheme, resultará em erro (figura 8) caso o valor passado seja de um tipo diferente do esperado. Em contrapartida, uma linguagem que não se comporta como fortemente tipada irá conseguir realizar a operação, mesmo com um tipo diferente do esperado. Um exemplo de linguagem que se comporta dessa forma é o C. Nesse caso,

ao realizar uma operação de soma com um dos operandos sendo, por exemplo, uma string (array de caracteres), o compilador gera um *warning*, mas a instrução é executada e um valor é retornado (figura 9).

4.3.1 Exemplo de tipagem forte - Scheme:

```
1 (define (addOne x)
2   (+ x 1))
3 (display (addOne 10))
4 (display (addOne "10"))
```

Figura 8 – Tipagem forte - Scheme

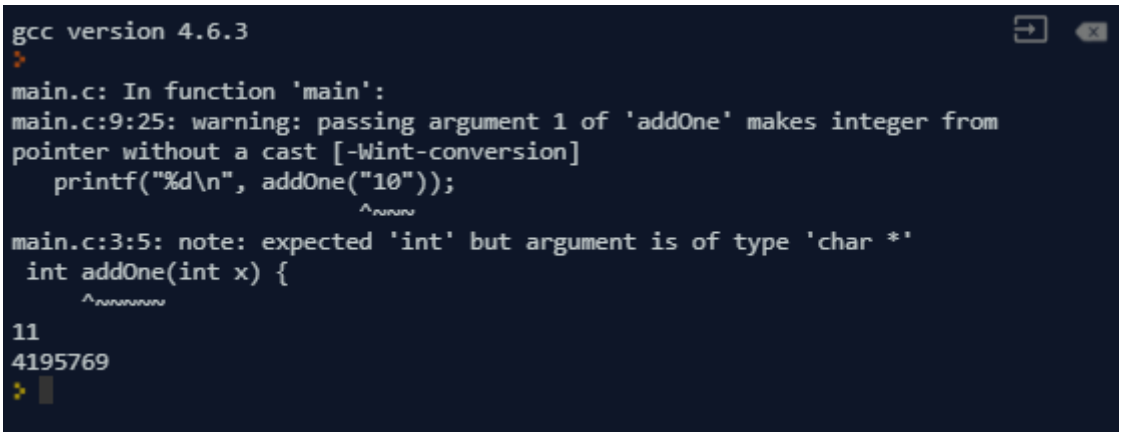


Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

4.3.2 Exemplo de tipagem fraca - C

```
1  #include "stdio.h"
2
3  int addOne(int x) {
4      return x + 1;
5  }
6
7  int main(void) {
8    printf("%d\n", addOne(10));
9    printf("%d\n", addOne("10"));
10
11   return 0;
12 }
```

Figura 9 – Tipagem forte - Scheme



Fonte: *print screen* do terminal referente ao compilador gcc 4.6.3, no site <http://repl.it>

3. Tipagem Dinâmica

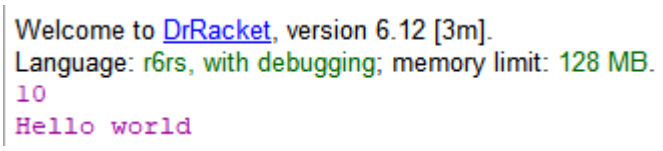
Além de ser uma linguagem fortemente tipada, Scheme realiza a verificação de tipos das variáveis em tempo de execução. Essa forma de verificação é conhecida como **tipagem dinâmica** ou **verificação de tipos dinâmica**. Sebesta (2009, p. 328) afirma que é melhor detectar erros em tempo de compilação do que em tempo de execução, uma vez que a correção feita mais cedo geralmente é menos custosa. Em contrapartida, a verificação de tipos estática dá ao programador uma flexibilidade reduzida.

Como forma de demonstrar tal característica, presente na linguagem tratada nesse artigo, seguem dois exemplos referentes as duas diferentes formas de verificação de tipos. O primeiro (figura 10) refere-se a linguagem Scheme e consiste em uma alteração de um tipo de uma variável por outro. Já o segundo (figura 11), refere-se a linguagem Go e apresenta um erro gerado em tempo de compilação, devido a atribuição de um valor de tipo diferente a uma variável declarada anteriormente.

4.3.3 Exemplo de tipagem dinâmica - Scheme

```
1 (define value 10)
2 (display value)
3 (newline)
4
5 (set! value "Hello world")
6 (display value)
7 (newline)
```

Figura 10 – Tipagem dinâmica - Scheme

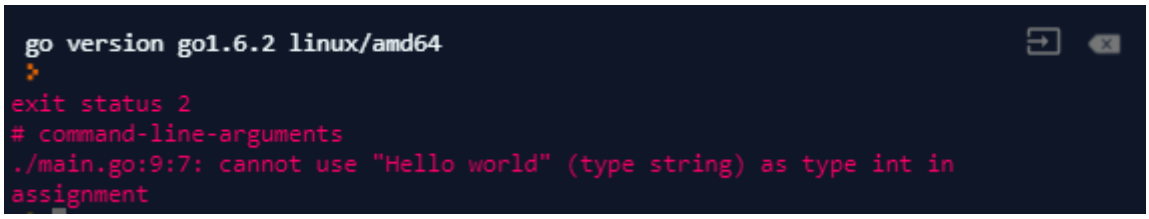


Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

4.4 Exemplo de tipagem estática - Go

```
1  package main
2  import "fmt"
3
4  func main() {
5    i := 10
6    fmt.Println(i)
7
8    i = "Hello world"
9  }
```

Figura 11 – Tipagem estática - Go



Fonte: *print screen* do terminal referente ao *go1.9.4 linux/amd64*, no site <http://repl.it>

4. *Closure:*

   Closure é uma ténica de implementação de amarração de entidades com escopo estático. Em termos práticos, é um registro que armazena uma função e o ambiente em que ela foi definida. Assim, as variáveis livres, utilizadas pela função, são persistidas seguindo o tempo de vida da mesma.

   Ambos os exemplos referem-se a implementação de uma função fibonacci, que faz uso da ténica de programação dinâmica, armazenando os valores já calculados em um tabela hash. O primeiro exemplo (figura 12) foi implementado em Scheme, enquanto o segundo (figura 13) foi desenvolvido em Python. No segundo exemplo, além da saída referente a um valor da sequência fibonacci, é mostrado os detalhes do registro da *closure*.

4.4.1 Exemplo de closure - Scheme:

```
1  (define fibonacci
2    (let ((memo (make-eq-hashtable)))
3      (lambda (n)
4        (cond
5          ((<= n 0) 0)
6          ((= n 1) 1)
7          ((hashtable-contains? memo n) (hashtable-ref memo n -1))
8          (else
9           (let ((value (+ (fibonacci (- n 1)) (fibonacci (- n
               2)))))
10            (hashtable-set! memo n value)
11            value))))))
12
13 (display (fibonacci 100))
14 (newline)
```

Figura 12 – Implementação de uma *Closure* - Scheme

```
Welcome to DrRacket, version 6.12 [3m].
Language: r6rs, with debugging; memory limit: 128 MB.
354224848179261915075
```

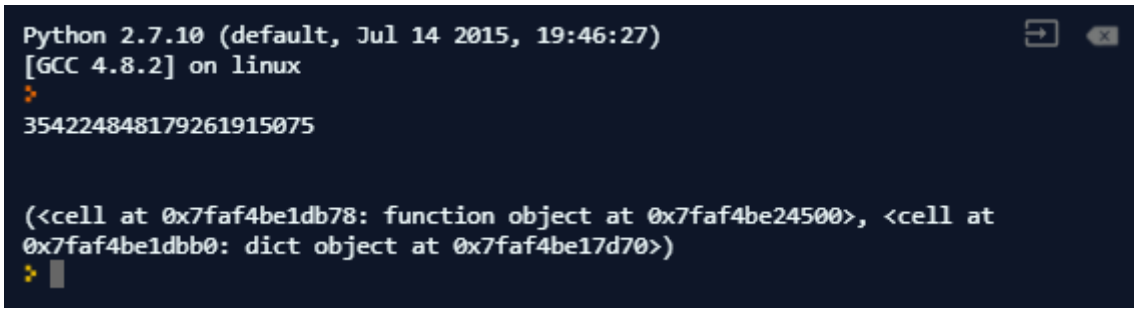
Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

4.4.2 Exemplo de closure - Python:

```
1  def fibonacci():
2    memo = {}
3
4    def calcFib (n):
5      value = -1
6      if n <= 0:
7        value = 0
8      elif n == 1:
9        value = 1
10     elif n in memo:
11       value = memo.get(n)
12     else:
13       value = calcFib(n - 1) + calcFib(n - 2)
```

```
14          memo[n] = value
15
16      return value
17    return calcFib
18
19  fib = fibonacci()
20  print(fib.__closure__)
```

Figura 13 – Implementação de uma *Closure* - Python



Fonte: *print screen* do terminal referente ao interpretador da versão 2.7.10 de Python, no site <http://repl.it>

5. *Tail Recursion:*

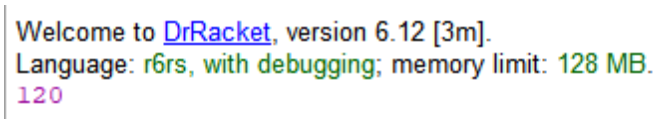
   É um tipo especial de recursão, onde a chamada recursiva é sempre a última coisa a ser realizada. É importante, porque significa que você pode passar o resultado da chamada recursiva diretamente, ao invés de espera-lo. Isto é uma otimização para o programa, que evita consumo excessivo da pilha.

   Nos exemplos a seguir, são apresentados dois tipos de recursão. Na primeira (figura 14), é abordado uma recursão básica, que não faz uso de recursos auxiliares. Já na segunda (figura 15), é apresentado o conceito de um *Tail Recursion*, que auxilia na gerência de memória.

4.4.3 Exemplo de recursividade - Scheme:

```
1  (define (factorial n)
2    (if (<= n 1)
3        1
4        (* n (factorial (- n 1)))))
5  (display (factorial 5))
6  (newline)
```

Figura 14 – Fibonacci calculado através de uma função recursiva comum - Scheme

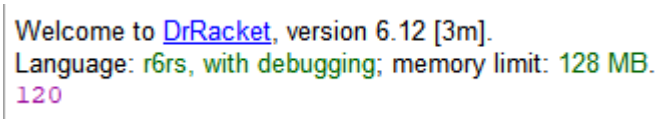


Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

4.4.4 Exemplo de tail recursion - Scheme:

```
1 (define (factorial n)
2     (define (factorial-helper n acc)
3         (if (<= n 1)
4             acc
5             (factorial-helper (- n 1) (* n acc))))
6   (factorial-helper n 1))
7
8 (display (factorial 5))
9 (newline)
```

Figura 15 – Fibonacci calculado através de uma *tail recursion - Scheme*



Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

6. *High-order-function*
   High-order functions são funções que fazem pelo menos um dos seguintes itens:

   - Aceitam uma ou mais funções como parâmetro
   - Retornam uma função como seu resultado

   No exemplo a seguir (figura 16), as funções *merge* e *mergesort* são *high-order functions*. Estas funções são assim consideradas, pois apresentam como característica a passagem de uma função de comparação como parâmetro.

4.4.5 Exemplo de high-order function - Scheme:

```
1 (define (split lst at)
2   (define (iter left right i)
```

```
3       (cond
4         ((or (null? right) (= i at)) (cons (reverse left) right))
5         (else (iter (cons (car right) left) (cdr right) (+ i
            1)))))
6     (iter '() lst 0))
7
8   (define (merge list1 list2 func)
9     (cond
10      ((null? list1) list2)
11      ((null? list2) list1)
12      ((func (car list1) (car list2)) (cons (car list1) (merge
           (cdr list1) list2 func)))
13      (else (cons (car list2) (merge (cdr list2) list1 func)))))
14
15  (define (mergesort lst func)
16    (define (sort currentLst)
17      (cond
18        ((<= (length currentLst) 1) currentLst)
19        (else
20        (let* ((splitted (split currentLst (div (length
             currentLst) 2))))
21          (merge (sort (car splitted)) (sort (cdr splitted))
               func)))))
22    (sort lst))
23
24  (define lst (list 312 12 2 302 238312 40 428 -1 2931 -53231 48
      24 5715 248248 4 525 248))
25  (display (mergesort lst <))
26  (newline)
27  (display (mergesort lst >))
28  (newline)
```

Figura 16 – *High-order function - Scheme*

```
Welcome to DrRacket, version 6.12 [3m].
Language: r6rs, with debugging; memory limit: 128 MB.
{-53231 -1 2 4 12 24 40 48 248 302 312 428 525 2931 5715 238312 248248}
{248248 238312 5715 2931 525 428 312 302 248 48 40 24 12 4 2 -1 -53231}
>
```

Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

7. *Hygienic Macro*

Scheme suporta diferentes tipos de macros. O tipo mais simples é o herdado de LISP (figura 17), conhecido como *defmacro* (em linguagens como *Common LISP* e *Emacs*

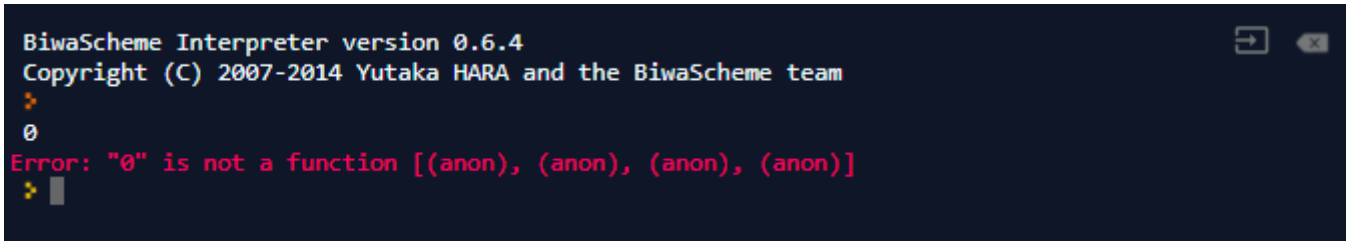
LISP) ou como *define-macro* (em algumas implementações de Scheme). Existem, também, outras implementações (como *Racket*) que rejeitam tal tipo de macro, por ser primitivo.

Essas macros apresentam um problema. Variáveis definidas no escopo referente a macro são visíveis exteriormente à ela. Dessa forma, é possível alterá-la em tempo de execução, resultando em erros ou comportamentos indefinidos do programa.

4.5 Exemplo de erro gerado em uma macro - Scheme:

```
1  (define-macro (while condition . body)
2    `(let loop ()
3       (cond (,condition
4              (begin . ,body)
5              (loop)))))
6
7  (let ((i 0) (loop  "0"))
8    (while (< i 10)
9           (set! loop (number->string i))
10          (display loop)
11          (newline)
12          (set! i (+ i 1))))
```

Figura 17 – Erro em uma macro - Scheme



Fonte: *print screen* do terminal referente ao interpretador *BiwaScheme*, no site <http://repl.it>

Para evitar este problema, Scheme introduziu o conceito de *Hygienic Macro* (figura 18). Este novo tipo de macro tem como característica impedir o acesso de uma variável que esteja dentro do corpo da mesma. Com isso, evita-se os possíveis erros, pois uma variável nunca poderá ser referenciada fora do escopo de definição.

4.6 Exemplo de uma *hygienic* macro - Scheme:

```
(define-syntax while
  (syntax-rules ()
    ((_ condition body ...)
     (let loop ()
       (cond (condition
              (begin body ...)
              (loop)))))))

(let ((i 0) (loop  "0"))
  (while (< i 10)
         (set! loop (number->string i))
         (display loop)
         (newline)
         (set! i (+ i 1))))
```

Figura 18 – *Hygienic macro - Scheme*

```
Welcome to DrRacket, version 6.12 [3m].
Language: r6rs, with debugging; memory limit: 128 MB.
0
1
2
3
4
5
6
7
8
9
```

Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

5 LINGUAGENS SEMELHANTES

A linguagens semelhantes são, em sua maioria, as que foram influenciadas em sua criação por Scheme e as que o influenciaram. Sendo assim, podemos analisar a lista de influências e de influenciadores da linguagem para traçar um modelo de linguagens que são consideradas equivalentes.

- LISP
  - Há mais facilidade para criar novos tipos;
  - Tem um sistema padrão de pacotes;
  - Tem um sistema padrão de objetos (CLOS, o CommonLISPObjectSystem);

  - Tem um sistema padrão de tratamento de exceções;
  - Tem o escopo de variáveis dinâmico;
  - Tem símbolos MACRO.

- Common LISP
  Apresenta as mesmas diferenças entre LISP e Scheme.

- Lua
  - Linguagem multiparadigma: imperativa, funcional e orientada à objetos;
  - Trata funções como variáveis de primeira classe;
  - Suporta funções de fechamento (closure);
  - Tem o escopo de variáveis dinâmico.

- Ruby
  - É fortemente tipada;
  - Tem o escopo de variáveis dinâmico.

- Racket

  Racket era originalmente Scheme, uma variante chamada PLT Scheme. Como houve a necessidade dos criadores da linguagem de uma implementação open source com bibliotecas gráficas, Racket foi desenvolvida.

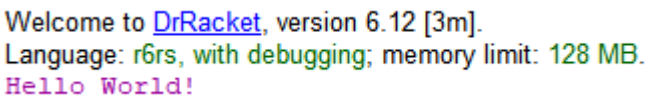
  As semelhanças são inúmeras com algumas diferenças como os dados de lista serem mutáveis e a linguagem ser funcional por padrão.

6 EXEMPLOS DE PROGRAMAS

6.1 Hello World:

```
1 (display "Hello World!\n")
```

Figura 19 – Hello World - Scheme

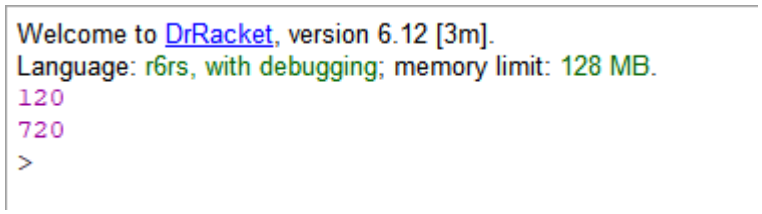


Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

6.2 Fatorial:

```
1  (define (factorial n)
2    (define (factorial-helper n result)
3      (if (<= n 1)
4          result
5          (factorial-helper (- n 1) (* result n))))
6    (factorial-helper n 1))
7
8  (display (factorial 5))
9  (newline)
10 (display (factorial 6))
11 (newline)
```

Figura 20 – Fatorial - Scheme



Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

6.3 Arvóre binária de busca:

```
1  (define (make-btree left value right)
2    (list left value right))
3
4  (define (btree-left btree)
5    (car btree))
6
7  (define (btree-value btree)
8    (cadr btree))
9
10 (define (btree-right btree)
11   (caddr btree))
12
13 (define (btree-search btree value)
14   (cond
15     ; If the current subtree is null then the searched element
```

```
16      ; does not exist and we must return false
17      ((null? btree) #f)
18      ; Else if we find the value, return true
19      ((= value (btree-value btree)) #t)
20      ; Else if the searched value is less than the value of the
21      ; current subtree, we need to look in the subtree
22      ; at the left
23      ((< value (btree-value btree)) (btree-search (btree-left btree)
            value))
24      ; Else, value is greater than the value of
25      ; the current subtree and we need to look
26      ; in the subtree at the right
27      (else (btree-search (btree-right btree) value))))
28
29  ; Returns the new tree
30  ; Use with set! function
31  (define (btree-insert btree value)
32    ; If it's a valid node, we can traverse the tree
33    ; to find the correct place to insert the new subtree (node)
34    (cond
35      ; Create a new subtree
36      ((null? btree) (make-btree '() value '()))
37      ; Else if value is already in the tree
38      ; just return the same tree
39      ((= value (btree-value btree)) btree)
40      ; Else if value is less than the current subtree value
41      ; go to the subtree at the left
42      ((< value (btree-value btree))
43          (make-btree (btree-insert (btree-left btree) value)
44                      (btree-value btree)
45                      (btree-right btree)))
46      ; Else, value is greater than the value
47      ; of the current subtree and
48      ; we need to look at the subtree at the right
49      (else (make-btree (btree-left btree)
50                        (btree-value btree)
51                        (btree-insert (btree-right btree) value)))))
52
53  (define btree (btree-insert '() 5))
54  (set! btree (btree-insert btree 10))
55  (set! btree (btree-insert btree 15))
56  (set! btree (btree-insert btree 3))
57
```

```
58 (display (btree-search btree 15))
59 (newline)
60 (display (btree-search btree 1))
61 (newline)
```

Figura 21 – Árvore binária de busca - Scheme

```
Welcome to DrRacket, version 6.12 [3m].
Language: r6rs, with debugging; memory limit: 128 MB.
#t
#f
```

Fonte: *print screen* da IDE DrRacket, no sistema operacional Windows 10

7 CONCLUSÃO

Scheme é uma linguagem de programação antiga, multiparadigma, destacada por abranger várias caraterísticas, como tipagem forte, dinâmica. Além disso, possui o recurso de macros, que possibilitam o programador a desenvolver novas implementações relacionadas a sintaxe da linguagem. É por meio disso que se pode prototipar uma gramática, sem que seja necessário desenvolver um novo compilador ou interpretador.

Por outro lado, um projeto muito extenso, que utilize de tal linguagem, tende a se torna inviável: sua sintaxe, muitas vezes extensa e minimalista, pode acabar por prejudicar o entendimento do mesmo. Por motivos como esse, novas linguagens acabaram sendo desenvolvidas, facilitando o compreendimento e encurtando as linhas de código, ao exemplo da linguagem Haskell.

Conclui-se, portanto, que a linguagem foi fundamental para desenvolvimento de outras novas, bem como exploração de novos recursos por ela possibilitada. Concomitantemente a isso, pode-se dizer que Scheme é uma linguagem didática em relação a avaliação de expressões. Isso se dá, pois, além de auxiliar no compreendimento e identificação de expressões, a forma de escrita do programa se assemelha ao que o interpretador visualiza, proporcionando a compreensão da forma de trabalho este. Não atoa, durante muitos anos tal linguagem foi utilizada no ensino em universidades de alto nível, como o MIT.

Referências

ABELSON, Harold; SUSSMAN, Jay Gerald; SUSSMAN, Julie. **Struct and Interpretation of Computer Programs**. 2nd. ed. [S.l.]: MIT Press, 1979. ISBN 0262510871.

DWARAMPUDI, Venkatreddy et al. Comparative study of the pros and cons of programming languages java, scala, c++, haskell, VB .net, aspectj, perl, ruby, PHP & scheme - a team 11 COMP6411-S10 term report. **CoRR**, abs/1008.3431, 2010. Disponível em: <http://arxiv.org/abs/1008.3431>.

DYBVIG, R. Kent. **The Scheme Programming Language: ANSI Scheme**. 2nd. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1996. ISBN 0134546466.

NAIM, Rana et al. Comparative studies of 10 programming languages within 10 diverse criteria. 08 2010.

NøRMARK, Kurt. **Functional Programming in Scheme**. 2013. Disponível em: <http://people.cs.aau.dk/~normark/prog3-03/html/notes/theme-index.html>.

REVOLVY. **History of the Scheme programming language**. 2017. Disponível em: <https://www.revolvy.com/main/index.php?s=History\%20of\%20the\%20Scheme\%20programming\%20language>.

SEBESTA, R.W. **Conceitos de Linguagens de Programação - 9.ed.:**. Grupo A - Bookman, 2009. ISBN 9788577808625. Disponível em: <https://books.google.com.br/books?id=vPldwBmt-9wC>.

WIKI, C2. **Functional Programming in Scheme**. 2014. Disponível em: <http://wiki.c2.com/?TailRecursion>.