Parser Combinatorial em Scheme

Vinícius Valente Maciel
Infax Tecnologia e Sistemas,
Rio de Janeiro

9 de abril de 2002

1 Motivação

Apos comprar o livro *Fundamentos de linguagem de programação* de Daniel P. Friedman, Mitchell Wand e Christopher T. Haynes e começar a lê-lo fiquei interessado em estudar interpretadores. Contudo o tempo disponível dificultava a tarefa. Então resolvi usar o tempo que tinha no percurso de metrô até o trabalho, da Tijuca para o centro, e uma poderosa e elegante ferramenta disponível, um interpretador Scheme chamado LispMe instalado em um Palm M100.

O primeiro grande problema encontrado foi não conseguir, depois de muitas tentativas rodar o gerador de analisador sintático SLLGEN utilizado pelos autores do livro. Infelizmente o LispMe não implementa na integra o padrão R5RS, necessario para o SLLGEN. Esta deficiência me deixava sem *datatypes*, *cases* e principalmente sem um analisador de sintaxe.

Tinha então tres opções:

- desistir;
- escrever um analisador no munhecão;
- escrever um gerador de analizadores;

Já havia lido artigos sobre monada e parser combinatorial, resolvi então tentar escrever meu parser com funções de alta ordem. O que me daria um analisador com a flexibilidade e facilidade de alteração de um gerador de analisadores.

Este trabalho é o resultado de 2 meses de idas e voltas ao trabalho. Nunca acho que alguma coisa que escrevo esteja realmente pronta, portanto o código ainda sofrerá muitas alterações, até que, de repente, eu canse de mexer com isso e procure outra brincadeira.

Este artigo não é somente a explicação dos conceitos e do código fonte, ele é o código fonte. Utilizo um programa chamado *noweb* para fazer *literate programming*. Com o *noweb* posso de um único arquivo extrair o codigo fonte do sistema, uma versão deste documento para impressão (PostScript ou pdf) ou até uma página html.

2 Introdução

Linguagens de Alta Ordem suportam funções e continuações como cidadões de primeira classe, podendo serem utilizados como parâmetros, retorno de funções, armazenados em estruturas de dados e seram tratados como valores ordinários.

Scheme é uma linguagem de alta ordem, é utilizando esta característica um conjunto de combinatoriais primitivos será criado é usado para escrever um parser. Funções combinatoriais, quando bem projetadas, tem uma flexibilidade muito grande, as que escreveremos podem ser utilizadas em parseres ou em algoritmos de busca com facilidades para a introdução de heurísticas.

2.1 Lendo o artigo

Como mencionado, este artigo além de explicar o funcionamento de um parser combinatorial, é o parser que esta explicando. O leitor irá encontrar todo o código fonte listado em fragmentos rotulados intercalados com texto em prosa explicando seu funcionamento.

Os fragmentos de código são precedidos de rótulos entre parênteses angulares, e podem comter referências para outros fragmentos. Por exemplo:

3 ⟨*mesg* 3⟩≡

```
(define (mesg m) (display m))
```

Define:
`mesg`, usado em chunk 11a.

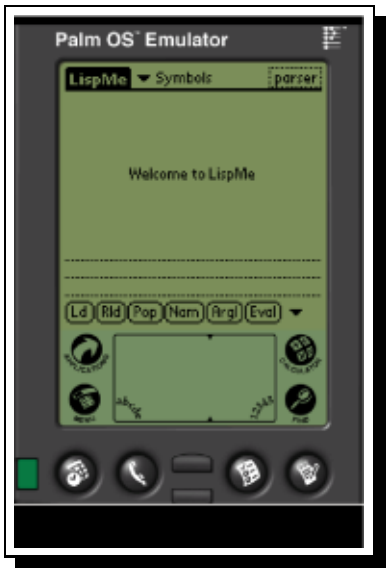

Figura 1: Palm rodando LispMe

é um fragmento de código válido do nosso programa, e é utilizado para encapsular uma função de saida de mensagens. Este encapsulamento faz com que seja fácil modificar o programa para que exiba mensagens em um console ou em um Alert do PalmOS.

O rótulo do fragmento é *mesg*, e em baixo da sua listagem temos informações adicionais como:

- Quais funções este fragmento de código declara;
- Em quais chunks(fragmentos) estas funções declaradas são utilizadas. 10,a (por exemplo) é uma referencia ao chunk *a* da página *10*.

Estas referências fazem com que seja fácil localizar no artigo trechos que nos ajudam a entender o chunk que está sendo estudado.

Neste exemplo, não temos referências a outros chunks dentro de *mesg*, mas o leitor não deve se preocupar, pois eles aparecerão no artigo, e serão fáceis de identificar.

Outra convênção adotada, é a de enfatizar os trechos do texto em prosa que se referem a nomes de funções Scheme. Por exemplo a função car deve aparecer no texto assim: *car*.

3 Combinatoriais básicos

Nesta seção será mostradas as funções primitivas que serão utilizadas para escrever nossos parsers.

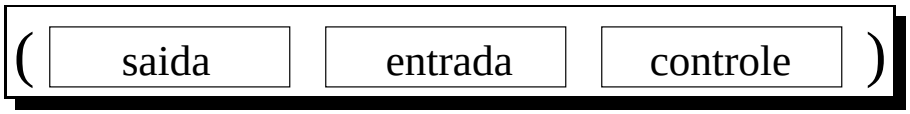

Figura 2: Estrutura do tipo *dado*.

Como dito anteriormente, nossas primitivas podem ser utilizadas com outros objetivos além de implementação de um parser. Por isso definiremos um tipo com nome *dado*, ao invés de chamá-lo de *parser*. Este tipo, como ilustrado na figura 2 na página 5, é uma lista com três elementos:

1. saida - armazena os elementos que já foram processados.
2. entrada - armazena os elementos que ainda devem ser processados.
3. controle - carrega informações de controle utilizadas no processamento do tipo *dado*.

A função abaixo chamada *algum* recebe como parâmetro um predicado, e tem como retorno uma outra função que tem como único parâmetro, um valor do tipo *dado*. Essa função retornada por *algum* transforma o seu parâmetro *input*, do tipo *dado*, passando o primeiro elemento de *(cadr input)*, para a saída, se o predicado *p* retornar verdade quando aplicado a *(car (cadr input))*.

5 ⟨*algum* 5⟩≡
```
(define algum
  (lambda (p)
    (lambda (input)
       (if (null? (cadr input))
           (list (car input) (cadr input) 'falha)
           (if (p (car (cadr input)))
               (list (append (list (car (cadr input)))
                     (list (car input)))
                     (cdr (cadr input)) '())
               (list (car input)
                     (cadr input)
                     'falha))))))
```

Define:
algum, usado em chunks 6, 17–20, e 24a.

Este exemplos abaixo ilustram o funcionamento de *algum*(utilizaremos alguns predicados que ainda serão declarados na próxima seção):

```
>((algum isletra) '(()(#\a)()))
((#\a ())()())

>((algum isletra) '(()(#\1)()))
(()(#\1) falha)

>((algum isletra) '(()(#\a #\b)()))
((#\a ())(#\b)())
```

O fechamento *um* é uma utilização de *algum* que utiliza um valor *tok* no lugar de um predicado. A *um* chama *algum* é utiliza como predicado a comparação de *tok* com *i* que é o primeiro elemento da saida do tipo *dado*.

6 ⟨*um* 6⟩≡

```
(define um
  (lambda (tok)
    (let ((aux (lambda (i) (equal? i tok))))
      (algum aux))))
```

Define:
`um`, usado em chunks 20, 23, e 24a.
Usa `algum` 5.

Exemplos de utilização de *um*.

```
>((um #\a) '(()(#\a())))
((#\a ())()())

>((um 2) '(()(1())))
(()(1) falha)

>((um 2) '(()(2())))
((2 ())()())
```

A próxima função chamada *ou* recebe duas funções como parâmetro e retorna uma função que processa um valor do tipo *dado*. A função retornada tenta processar a variável *input* com a função *fun1*, se essa função não falhar, seu resultado será retornado, se falhar a função *fun2* será aplicada sobre *input*, novamente, se *fun2* não falhar, seu resultado será retornado, senão *input* será retornado.

7 ⟨*ou* 7⟩≡

```
(define ou
  (lambda (fun1 fun2)
    (lambda (input)
      (let ((p1 (fun1 input)))
         (begin
            (if (equal? (caddr p1) 'falha)
                (fun2 input)
                p1))))))
```

Define:
ou, usado em chunks 17b, 21a, 23, e 24a.

Abaixo temos exemplos de utilização da função *ou*.

```
>((ou (algum isletra)(algum isnumero)) '(()(#\a)()))
((#\a ())()())

>((ou (algum isletra)(algum isnumero)) '(()(#\1)()))
((#\1 ())()())

>((ou (algum isletra)(algum isnumero)) '(()(#\1 #\a)()))
((#\1 ())(#\a)())

>((ou (algum isletra)(algum isnumero)) '(()(#\+)()))
(()(#\+)falha)
```

A função *entao* e muito semelhante a função *ou*, enquanto essa tenta aplicar *fun2* como alternativa para *fun1*, *entao* tenta fazer estas ações em sequência. Se *fun1* ou *fun2* falham *entao* também falha.

8 ⟨*entao* 8⟩≡

```
(define entao
  (lambda (fun1 fun2)
    (lambda (input)
      (let ((p1 (fun1 input)))
        (if (equal? (caddr p1) 'falha)
            p1
            (let ((p2 (fun2 (list '() (cadr p1) (caddr p1)))))
              (if (equal? (caddr p2) 'falha)
                  p2
                  (list (list (car p1) (car p2))
                        (cadr p2)
                        (caddr p2)))))))))
```

Define:
entao, usado em chunks 17–20, 23, e 24a.

A *entao* pode ser usada como nos exemplos abaixo:

```
>((entao (algum isletra)(algum isnumero)) '(()(#\a #\2)()))
(((#\a ())(#\2 ()))()())

>((entao (algum isletra)(algum isnumero)) '(()(#\1 #\b)()))
(()(#\1 #\b) falha)

>((entao (algum isletra)(algum isnumero)) '(()(#\1)()))
(()(#\1) falha)
```

A função retornada pela função *varios* tenta repetir a aplicação de *fun* em *input* até que essa aplicação falhe. O retornado é o que esta função consegue transformar em *input* até que *fun* falhe.

9 ⟨*varios* 9⟩≡

```
(define varios
  (lambda (fun)
    (lambda (input)
       (let ((p1 (fun  input)))
          (if (equal? (caddr p1) 'falha)
              (list (car p1) (cadr p1) '())
              (let ((p2 ((varios fun) p1)))
                 (list (list (car p1) (car p2))
                       (cadr p2)
                       (caddr p2))))))))
```

Define:
varios, usado em chunks 17–21.

Veja os exemplos de utilização de *varios*:

```
>((varios (algum isletra)) (list '()(string->list "ab")'()))
(((#\a ())((#\b(#\a()))(#\b(#\a()))))) () ())

>((varios (algum isletra)) (list '()(string->list "a1")'()))
(((#\a())(#\a())) (#\1) ())
```

A *aplica* é uma função que altera o primeiro elemento de saida de uma variável do tipo *dado*. Ela aplica a função *tratamento* sobre o valor retornado da aplicação de *fun* em *input*.

10 ⟨*aplica* 10⟩≡

```
(define aplica
  (lambda (fun tratamento)
    (lambda (input)
       (let ((p (fun  input)))
          (if (equal? (caddr p) 'falha)
              p
              (list (tratamento (car p))
                    (cadr p)
                    (caddr p)))))))
```

Define:
aplica, usado em chunks 18–21, 23, e 24a.

Esta função é utilizada para ajustar os valores de saida de um *dado*. Nos exemplos abaixo vemos como isso pode ser feito:

```
>((aplica (algum isletra) (lambda(x) (list x))) '(()(#\a)()))
(((#\a ())) () ())

>((aplica (algum isnumero)(lambda(x)'())) '(()(#\1)'()))
(() () ())
```

Nosso tratamento de erros será realizado através da função *fix*, que retorna uma função que tenta aplicar *fun* a *input*. Se esta aplicação falhar a mensagem de erro *err* será exibida. Das funções combinatorias primitivas utilizadas, esta é a única que não é inteiramente funcional, devido a utilização do fechamento *mesg*.

11a ⟨*fix* 11a⟩≡

```
(define fix
  (lambda (err fun)
    (lambda (input)
       (let ((p (fun  input)))
          (if (equal? (caddr p) 'falha)
              (mesg err)
              p)))))
```

Define:
`fix`, nunca usado.
Usa `mesg` 3.

O fechamento *fix* pode ser utilizado como nos exemplos abaixo:

```
>((fix "Não é uma letra" (algum isletra)) '(()(#\1)()))
Não é uma letra
"Não é uma letra"

>((fix "Não é um número" (algum isnumero)) '(()(#\1)'()))
((#\1 ())()())
```

As funções *nada* e *limpa* são utilizadas em parceria com a função *aplica* e, respectivamente, desaparece com um elemente que vai entrar na saída e limpa todo o conteúdo da saída.

11b ⟨*nada e limpa* 11b⟩≡

```
(define nada (lambda (input) '()))

(define limpa
   (lambda (id) '()))
```

Define:
`limpa`, nunca usado.

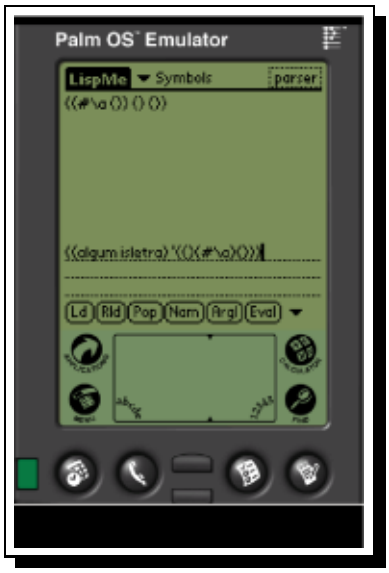

Figura 3: LispMe rodando função primitiva.

O módulo com as funções primitivas agrupas todas as funções declaradas nesta seção como mostrado no chunck abaixo.

12 ⟨*combinatoriais basicos* 12⟩≡

```
;combinatoriais basicos

⟨mesg 3⟩
⟨ou 7⟩
⟨entao 8⟩
⟨varios 9⟩
⟨aplica 10⟩
⟨fix 11a⟩
⟨nada e limpa 11b⟩
⟨algum 5⟩
⟨um 6⟩
```

4 Exemplo de utilização - interpretador

O nosso objetivo nesta seção é demonstrar a utilização dos combinatoriais primitivos que declaramos na implementação de um interpretador simples. A linguagem implementada é muito simples, por isso, principalmente o analisador léxico, vai parecer muito completa para nosso exemplo. Mas a intenção é termos um analisador léxico que possa ser reaproveitado.

4.1 A linguagem

O interpretador de exemplo será um avaliador de expressões aritméticas simples, com soma(+) e multiplicação (*). A gramática abaixo mostra a estrutura das expressões.

$$
\begin{array}{lcl}
expressao & \rightarrow & termo \\
 & & \mid termo + expressao \\
termo & \rightarrow & fator \\
 & & \mid fator * termo \\
fator & \rightarrow & numero \\
 & & \mid (expressao)
\end{array}
$$

Exemplos de expressões válidas: "1", "(2)", "9+7", "9+7*9", "(9+7)*9", "8*9*3+8*(8*(7+4))".

4.2 Funções Auxiliares

Declaramos nesta seção funções que auxiliarão no implementação do analisador léxico e no analisador sintático. Temos inicialmente um grupo de funções que são predicados utilizador por analisadores léxicos, normalmente estes fechamentos são utilizados com o combinatorial *algum*.

13 ⟨*caracteres* 13⟩≡

```
(define alfa
  (string->list
    "abcdefghijklmnopqrstuvwxyz_ABCDEFGHIJKLMNOPQRSTUVWXYZ"))

(define pontuacao (string->list
  "()=+-/*><[]{}~&$.,?"))

(define nums (string->list "0123456789"))

(define brackets (string->list "()[]{};,"))

(define espacos (string->list " \t\n"))
(define (isespaco a) (list? (member a espacos)))
(define (isbra a) (list? (member a brackets)))
(define (isletra a) (list? (member a alfa)))
(define (isnumero a) (list? (member a nums)))
```

```
(define ispontuacao (lambda(a)
   (list? (member a pontuacao))))
```

Define:
`isbra`, nunca usado.
`isespaco`, usado em chunks 18–20.
`isletra`, usado em chunk 17b.
`isnumero`, usado em chunks 17b e 19.
`ispontuacao`, usado em chunk 18b.

Exemplos de utilização destas funções já são mostrados quando demonstramos o uso de *algum*.

O próximo grupo de funções é mais complexo é são utilizadas, normalmente, com o combinatorial *aplica* para ajustar as saídas. A primeira função, chamada *ultimo*, retorna o ultimo elemento de uma lista.

14a ⟨*ultimo* 14a⟩≡
```
(define ult (lambda (l u)
   (if (null? l)
      u
      (ult (cdr l) (car l)))))

(define (ultimo l) (ult l '()))
```

Abaixo vemos exemplos de utilização de *ultimo*:

```
>(ultimo '(1 2 3 4 5))
5

>(ultimo '( 1 2 3 (4 5) (6)))
(6)
```

Em alguns momentos na utilização do combinatorial *varios* surgem listas como *(1 (2 (3 (5 ()))))* e desejamos obter a lista *(1 2 3 4 5)*. Neste ponto utilizamos a funçao *coleta* declarada abaixo.

14b ⟨*coleta* 14b⟩≡
```
(define (coleta l)
   (if (null? l)
      '()
       (append (list (car l))
               (coleta (cadr l)))))
```

Define:
`coleta`, usado em chunks 15, 16, e 18–21.

Para ilustrar o uso de *coleta*, temos o exemplo abaixo.

```
>(coleta '( 1 (#\a ( 3 ( #\c))))))
(1 #\a 3 #\c)
```

Outra função que é utilizada para coletar dados das listas geradas por *varios* e *algum* chama-se *coleta-ultimo-interno*. Abaixo temos sua listagem.

15a ⟨*coleta-ultimo-interno* 15a⟩≡

```
(define (col-ult l u)
   (if (null? l) u
       ( if (list? (car l))
            (col-ult (cadr l)
                     (car l)) u)))

(define (coleta-ultimo-interno l)
   (coleta (col-ult l '())))
```

Define:
  `coleta-ultimo-interno`, usado em chunks 15 e 16.
Usa `coleta` 14b.

Para criar uma forma de ter os tipos dos tokens analisados, construtores para listas onde o primeiro elemento é um rótulo e o segundo e o valor do token são utilizados. Além disso, foi criado também funções reconhecedoras para os *pseudo*-tipos criados.

A função *make-ident* é um construtor para um valor do tipo *identificador*, ela é utilizada em conjunto com o combinatorial *aplica*. Também um reconhecedor para identificadores é declarado.

15b ⟨*aux identificador* 15b⟩≡

```
(define make-ident
  (lambda (id)
   (list 'ident
     (list->string (append (list (caaar id))
       (reverse (coleta-ultimo-interno
         (cadar id))))))))

(define (identificador? a)
     (equal? (car a)  'ident ))
```

Define:
  `identificador?`, nunca usado.
  `make-ident`, usado em chunk 18a.
Usa `coleta` 14b, `coleta-ultimo-interno` 15a, e `ident` 17b.

De forma semelhante os tipos *inteiro*,*real* e *simbolos* têm construtores, utilizados com *aplica*, e reconhecedores declarados.

Abaixo temos a *construção* desses 3 *pseudo*-tipos.

16a ⟨*aux inteiro* 16a⟩≡

```
(define make-int
  (lambda (id)
   (list 'int
     (list->string (append (list (caaar id))
       (reverse (coleta-ultimo-interno
          (cadar id))))))))


(define (inteiro? a)
     (equal? (car a)  'int))
```

Define:
`inteiro?`, usado em chunks 24a e 25c.
`make-int`, usado em chunk 19.
Usa `coleta` 14b, `coleta-ultimo-interno` 15a, e `inteiro` 19.

16b ⟨*aux real* 16b⟩≡

```
(define make-real
  (lambda (id)
   (list 'real
      (list->string (append (list (caaar id))
         (reverse (coleta-ultimo-interno
            (cadar id))))))))


(define (realp? a)
     (equal? (car a)  'int))
```

Define:
`make-real`, usado em chunk 20.
`realp?`, nunca usado.
Usa `coleta` 14b, `coleta-ultimo-interno` 15a, e `real` 20.

16c ⟨*aux simb* 16c⟩≡

```
(define make-simb
  (lambda (id)
   (list 'simb id)))

(define (simb? a)
     (equal? (car a)   'simb ))
```

Define:
`make-simb`, nunca usado.
`simb?`, usado em chunk 22b.
Usa `simb` 18b.

As funções auxiliares de nosso sistema são as declaradas nessa seção e agrupadas pelo chunk *auxiliares*.

17a ⟨*auxiliares* 17a⟩≡
⟨*caracteres* 13⟩
⟨*ultimo* 14a⟩
⟨*coleta* 14b⟩
⟨*coleta-ultimo-interno* 15a⟩
⟨*aux identificador* 15b⟩
⟨*aux inteiro* 16a⟩
⟨*aux real* 16b⟩
⟨*aux simb* 16c⟩

4.3 Analisador léxico

Nesta seção começaremos a utilizar os combinatoriais da forma que objetivamos neste artigo. Inicialmente declaremos a função *ident* que têm como parâmetro de entrada um valor do tipo *dado*. Podemos ler esta declaração da seguinte forma:

> Um identificador é formado de uma letra seguida de nenhum ou várias letras ou números.

17b ⟨*ident* 17b⟩≡

```
(define ident (lambda(input)
   ((entao (algum isletra)
      (varios (ou (algum isletra)
        (algum isnumero))))
   input)))
```

Define:
`ident`, usado em chunks 15b e 18a.
Usa `algum` 5, `entao` 8, `isletra` 13, `isnumero` 13, `ou` 7, e `varios` 9.

Exemplos de utilização de *ident*:

```
>(ident '(()(#\i #\d  #\1)()))
(((#\i())((#\d())((#\1 (#\d()))(#\1 (#\d()))))))()())
>(ident '(()(#\1 #\i #\d )()))
(() (#\1 #\i #\d) falha)
```

Podemos agora declarar nosso *token* que pode ser lido da seguinte forma:

> Um token e formado por nenhum ou vários espaços seguido de um *ident* seguido por nenhum ou vários espaços.

Na *token* o combinatorial *aplica* é utilizado com a função f para ajustar a saida.

18a ⟨*token* 18a⟩≡

```
(define token (lambda(input)
 (let ((f (lambda (a)
            (make-ident (coleta
              ( ultimo a))))))
 ((aplica
    (entao
         (aplica (varios (algum isespaco))
             nada)
             (entao  ident
            (aplica (varios (algum isespaco))
             nada)))
  f) input))))
```

Define:
  `token`, usado em chunk 21a.
Usa `algum` 5, `aplica` 10, `coleta` 14b, `entao` 8, `ident` 17b, `isespaco` 13, `make-ident` 15b, e `varios` 9.

Exemplos de utilização de *token*:

```
>(token (list '() (string->list " id1 ") '()))
((ident "id1")()())
```

18b ⟨*simb* 18b⟩≡

```
(define simb  (lambda(input)
    (let ((f (lambda (a)
     (list 'simb (caaadr a)))))
   ( (aplica (entao
     (aplica
         (varios (algum isespaco)) nada)
     (entao (algum ispontuacao)
       (aplica
         (varios (algum isespaco)) nada)
      )) f) input))))
```

Define:
  `simb`, usado em chunks 16c e 21–24.
Usa `algum` 5, `aplica` 10, `entao` 8, `isespaco` 13, `ispontuacao` 13, e `varios` 9.

Análogo a *ident* e *token*, abaixo declaramos *inteiroaux* e *inteiro*.

19 ⟨*inteiro* 19⟩≡

```
(define inteiroaux (lambda(input)
     ((entao (algum isnumero)
             (varios  (algum isnumero)))
input)))

(define inteiro (lambda(input)
 (let ((f (lambda (a)
           (make-int (coleta
             ( ultimo a))))))
 ((aplica
    (entao
         (aplica (varios (algum isespaco))
             nada)
             (entao inteiroaux
        (aplica (varios (algum isespaco))
             nada)))
     f) input))))
```

Define:
`inteiro`, usado em chunks 16a, 20, 21a, 24a, e 25c.
`inteiroaux`, nunca usado.
Usa `algum` 5, `aplica` 10, `coleta` 14b, `entao` 8, `isespaco` 13, `isnumero` 13, `make-int` 16a, e `varios` 9.

Exemplo de utilização de *inteiro*:

```
>(inteiro (list '() (string->list " 1234 ") '()))
((int "1234")()())
```

Também análogo a *ident* e *token*, abaixo declaramos *realaux* e *real*. A função *real* reconhece somente números com o separador (.) ponto.

20 ⟨*real* 20⟩≡

```
(define realaux (lambda(input)
     ((entao inteiro
          (entao (um #\.) inteiro ))
        input)))

(define real (lambda(input)
 (let ((f (lambda (a)
           (make-real (coleta
             ( ultimo a))))))
 ((aplica
    (entao
         (aplica (varios (algum isespaco))
             nada)
             (entao realaux
         (aplica (varios (algum isespaco))
             nada)))
     f) input))))
```

Define:
  `real`, usado em chunk 16b.
  `realaux`, nunca usado.
Usa `algum` 5, `aplica` 10, `coleta` 14b, `entao` 8, `inteiro` 19, `isespaco` 13, `make-real` 16b, `um` 6, e `varios` 9.

Exemplo de utilização de *inteiro*:

```
>(real (list '() (string->list " 123.456 ") '()))
((real "123.456")()())
>(real (list '() (string->list " 123. ") '()))
(()() falha)
>(real (list '() (string->list " 123 ") '()))
(()() falha)
```

No momento utilizaremos um analizador léxico que reconhece todos os tokens, símbolos e inteiros. Abaixo declaramos a função *lexaux* que processa todos os elementos da entrada utilizando o combinatorial *varios*.

A *lexaux*, antes de iniciar sua análise, transforma sua string de entrada em um valor do tipo *dado*.

21a ⟨*lex* 21a⟩≡

```
(define lexaux(lambda(input)
   (let ((f (lambda (a)(coleta a))))
     (
       (aplica
        (varios (ou
               (ou inteiro
                        token)
               simb)
        )
       f)
      (list '() (string->list input) '())
     ))))
```

Define:
  `lexaux`, usado em chunk 21b.
Usa `aplica` 10, `coleta` 14b, `inteiro` 19, `ou` 7, `simb` 18b, `token` 18a, e `varios` 9.

O fechamento *lex* é a interface do nosso analizador léxico. Ele utiliza a função *lexaux* para processar nossos tokens e retorna, se obtiver sucesso, um novo valor do tipo *dado* em que os tokens da saida do processamento passam para a entrada.

21b ⟨*lex* 21a⟩+≡

```
(define lex(lambda(input)
   (let ((l (lexaux input)))
       (if (equal? (caddr l) 'noparse)
           l
          (list '() (car l) '())))))
```

Define:
  `lex`, usado em chunk 26a.
Usa `lexaux` 21a.

Exemplos de utilização de *dado*:

```
>(lex " x = 12 ")
((()((ident "x")(simb #\=)(int "12"))())

>(lex " um dois 3 ")
((()((ident "um")(ident "dois") (int "3"))())
```

22a ⟨*analizador lexico* 22a⟩≡
⟨*ident* 17b⟩
⟨*token* 18a⟩
⟨*simb* 18b⟩
⟨*inteiro* 19⟩
⟨*real* 20⟩
⟨*lex* 21a⟩

4.4 Analisador sintático

Também para o analisador sintático precisamos criar tipos, no caso da nossa linguagem serão declarado contrutores e reconhecedores para os novos tipos *expressao*, *termo* e *fator*.

22b ⟨*construtores gramatica* 22b⟩≡

```
(define make-fator (lambda (i )
    (if (list? (caar i))
     (if (simb? (caar i))
       (list 'fator (caadr  i)))
      (list 'fator (car i)))))

(define  make-termo (lambda (i )
  (if (fator? i)
      (list 'termo  i)
      (list 'termo(caar i)(cadr i)))))

(define make-expressao (lambda (i)
(if (termo? i)
      (list 'expressao  i)
      (list 'expressao (caar i)(cadr i)))))
```

Define:
`make-expressao`, usado em chunk 23b.
`make-fator`, usado em chunk 24a.
`make-termo`, usado em chunk 23c.
Usa `expressao` 23b, `fator` 24a, `fator?` 23a, `simb` 18b, `simb?` 16c, `termo` 23c, e `termo?` 23a.

23a ⟨*reconhecedores gramatica* 23a⟩≡
```
(define fator? (lambda (i )
    (equal?  'fator (car i))))
(define  termo? (lambda (i )
    (equal? 'termo (car i))))
(define expressao? (lambda (i)
    (equal? 'expressao (car i))))
```
Define:
`expressao?`, usado em chunk 25a.
`fator?`, usado em chunks 22b e 25a.
`termo?`, usado em chunk 22b.
Usa `expressao` 23b, `fator` 24a, e `termo` 23c.

As regras da gramática serão mapeadas diretamente em Scheme utilizando nossos combinatorias. Abaixo, na função *expressao*, fazemos o mapeamento da regras expressao da nossa linguagem utilizando uma notação prefixada, caracteristicas do Scheme. Em seguida utilizamos o combinatorial *aplica* para criarmos um valor do tipo *expressao* utilizando o construtor *make-expressao*.

23b ⟨*nao-terminal expressao* 23b⟩≡
```
(define expressao (lambda (input )
   (
      (aplica
          (ou (entao
                  (entao termo
                              (um '(simb #\+)))
                     expressao) termo)
               make-expressao)
      input)))
```
Define:
`expressao`, usado em chunks 22–26.
Usa `aplica` 10, `entao` 8, `make-expressao` 22b, `ou` 7, `simb` 18b, `termo` 23c, e `um` 6.

Tarefa análoga e feita com as regras termo e fator, utilizando *make-termo* e *make-fator*, respectivamente, com *aplica*.

23c ⟨*nao-terminal termo* 23c⟩≡
```
(define termo (lambda (input )
   (
     (aplica
          (ou  (entao (entao fator
                              (um '(simb #\*)))
                   termo) fator )
         make-termo)
      input)))
```
Define:
`termo`, usado em chunks 22, 23, e 25.
Usa `aplica` 10, `entao` 8, `fator` 24a, `make-termo` 22b, `ou` 7, `simb` 18b, e `um` 6.

24a ⟨*nao-terminal fator* 24a⟩≡

```
(define fator (lambda (input )
    (
      (aplica
           (ou   (algum inteiro?)
            (entao
              (um '(simb #\())
                              (entao expressao
                               (um '(simb #\))))))
     make-fator)
  input)))
```

Define:
`fator`, usado em chunks 22, 23, e 25.
Usa `algum` 5, `aplica` 10, `entao` 8, `expressao` 23b, `inteiro` 19, `inteiro?` 16a, `make-fator` 22b, `ou` 7, `simb` 18b, e `um` 6.

Exemplo de análise sintática:

```
>(expressao (lex "12"))
((expressao (termo (fator (int "12")))) ()())

>(expressao (lex "  3 + 1"))
((expressao (termo(fator(int "3")))
            (expressao(termo(fator(int "1"))))))()())
```

Note que como em nossa linguagem podemos resolver qual operaçao é aplicada sobre nossos números utilizando somente a produção, não é necessário colocar na árvore sintática os operadores "+" e "*".

24b ⟨*analizador sintatico* 24b⟩≡
⟨*construtores gramatica* 22b⟩
⟨*reconhecedores gramatica* 23a⟩
⟨*nao-terminal expressao* 23b⟩
⟨*nao-terminal termo* 23c⟩
⟨*nao-terminal fator* 24a⟩

4.5 Interpretador

Nosso interpretador é implementado da forma proposta no livro *Fundamentos de linguagem de programação* de Daniel P. Friedman, Mitchell Wand e Christopher T. Haynes. Declaramos uma função de avaliação para cada não terminal da gramática, e estas funções percorrem a árvore sintática recursivamente processando o valor da expressao.

25a ⟨*avalia expressao* 25a⟩≡

```
(define avalia-expressao
   (lambda(e)
    ((and (expressao? e)
                (= (length e) 3) )
     (+ (avalia-termo (cadr e))
          (avalia-expressao (caddr e))))
      ((fator? e)
        (avalia-fator  (cadr e)))
      (else (avalia-termo(cadr e))))))
```

Define:
`avalia-expressao`, usado em chunks 25c e 26a.
Usa `avalia-fator` 25c, `avalia-termo` 25b, `expressao` 23b, `expressao?` 23a, `fator` 24a, `fator?` 23a, e `termo` 23c.

25b ⟨*avalia termo* 25b⟩≡

```
(define avalia-termo
   (lambda(e)
    (cond
     ((= (length e) 3)
     (* (avalia-fator (cadr e))
          (avalia-termo (caddr e))))
      (else (avalia-fator(cadr e))))))
```

Define:
`avalia-termo`, usado em chunk 25a.
Usa `avalia-fator` 25c, `fator` 24a, e `termo` 23c.

25c ⟨*avalia fator* 25c⟩≡

```
(define avalia-fator
   (lambda(e)
    (cond
     ((inteiro? (cadr e) )
             (string->object  (cadadr  e)))
     (else  (avalia-expressao e)))))
```

Define:
`avalia-fator`, usado em chunk 25.
Usa `avalia-expressao` 25a, `expressao` 23b, `fator` 24a, `inteiro` 19, e `inteiro?` 16a.

A interface de nosso interpretador é a funçao *interprete* que chama o analisador léxico, em seguida o analisador sintático, e passa o resultado para a função *avalia-expressao*.

26a ⟨*interprete* 26a⟩≡
```
(define interprete
   (lambda(e)
   (avalia-expressao(car
       (expressao (lex e))))))
```
Define:
`interprete`, nunca usado.
Usa `avalia-expressao` 25a, `expressao` 23b, e `lex` 21b.

Exemplo de utilização de *interprete*:

```
>(interprete "  3")
3

>(interprete "  3 + 1")
4

>(interprete "  4 *3 + 1")
13

>(interprete "  4*(3 + 1)")
16

>(interprete "  (3) + (1)")
4
```

26b ⟨*interpretador* 26b⟩≡
⟨*avalia expresao* (nunca definido)⟩
⟨*avalia termo* 25b⟩
⟨*avalia fator* 25c⟩
⟨*interprete* 26a⟩

5 Melhorias

Nesta seção algumas melhorias no nosso sistema são propostas:

1. Os valores colocados na saida por *algum* e *varios* são confusos e dificultam o trabalho de escrever funções para ser utilizadas com o combinatorial *aplica*. Isso pode ser melhorado.

2. Os combinatoriais *ou* e *entao* pode ser redefinidos para poder aceitar numero de paramentros variáveis, tornado as expressões de suas utilizações mais claras, como nos exemplos:

```
>(ou ident numero simb)

>(entao ident (um #\=) numero)
```

6 Índices

6.1 Chunks

⟨*mesg* 3⟩
⟨*nada e limpa* 11b⟩
⟨*nao-terminal expressao* 23b⟩
⟨*nao-terminal fator* 24a⟩
⟨*nao-terminal termo* 23c⟩
⟨*ou* 7⟩
⟨*real* 20⟩
⟨*reconhecedores gramatica* 23a⟩
⟨*simb* 18b⟩
⟨*token* 18a⟩
⟨*ultimo* 14a⟩
⟨*um* 6⟩
⟨*varios* 9⟩

6.2 Identificadores