Código para bootstrapear sweb

Eduardo Acuña Yeomans

6 de Octubre del 2016

1 (tangle.scm)

Esta es la implementación del procedimiento `tangle`, este tiene el objetivo de tomar un archivo de entrada y un nombre de fragmento de código y extraer todos los fragmentos de código del archivo con el nombre dado expandiendo las referencias a otros fragmentos.

Esta extracción de código es vaciada en un archivo nombrado como el nombre del fragmento dado.

```
(load "lib/utils.scm")
(load "lib/input.scm")
(load "lib/parcomb.scm")
```

```
(define (tangle ifilename cname)
  (pipe #f
        (markup ifilename)
        (flatten-code-content cname)
        (dump-flat-content cname)))
```

1.1 *Etapa `flatten-code-content`*

En esta etapa de toma una lista de fragmentos y a partir del nombre de un fragmento de código `cname` se concatena y aplana de manera recursiva el contenido de los códigos relevantes a `cname`.

```
(define ((flatten-code-content cname) chunks)
  (let ((code-chunks (filter code? chunks)))
    (expand-code-content-refs (append-code-content cname code-chunks)
                              code-chunks 0)))
```

El siguiente procedimiento se encarga únicamente de concatenar el contenido de los códigos relevantes al nombre `cname`.

```
(define (append-code-content cname code-chunks)
  (define content
    (apply append (map code-content
                       (filter (lambda (code) (string=? cname (code-name code)))
                               code-chunks))))
  (define len (length content))
  (if (zero? len)
```

```
      (error "No chunk is named" cname)
      (drop-right content 1)))
```

El siguiente procedimiento expande las referencias de un contenido concatenado tomando en cuenta las columnas en las que inician las referencias.

```
(define (expand-code-content-refs content code-chunks indent-spaces)
  (define indent-str (make-string indent-spaces #\space))
  (let recur ((content content)
              (col  0))
    (if (null? content)
        '()
        (let ((x  (car content))
              (xs (cdr content)))
          (cond ((eqv? #\newline x)
                 (cons x (cons indent-str (recur xs 0))))
                ((string? x)
                 (cons x (recur xs (+ col (string-length x)))))
                ((refs? x)
                 (let ((fill (append-code-content (refs-name x) code-chunks)))
                   (append (expand-code-content-refs fill code-chunks col)
                           (recur xs col))))
                (else
                 (error "Malformed code content" content)))))))
```

1.2 *Etapa `dump-flat-content`*

En esta etapa se escribe a un archivo un contenido sin referencias, es decir, una lista compuesta únicamente de cadenas de caracteres o del caracter `#\newline`.

```
(define ((dump-flat-content ofilename) content)
  (with-output-to-file ofilename
    (lambda ()
      (for-each display content))))
```

1.3 *Etapa `markup`*

Esta etapa es la más complicada de todo el programa, se encarga de leer un archivo con la sintaxis de `sweb` y regresar la representación interna del programa como una lista de fragmentos.

```
(define ((markup ifilename) anything)
  (parse (tokenize (open-input-from-file ifilename))))
```

1.3.1 *Análisis léxico*

Primero se tokeniza la entrada, esto se logra identificando que caracteres de la entrada conforman tokens, nuevas líneas o texto.

```
(define token-newline ':token-newline)
(define token-lbracks ':token-lbracks)
(define token-rbreqs  ':token-rbreqs)
(define token-rbracks ':token-rbracks)
(define token-at      ':token-at)
(define token-def     ':token-def)

(define (tokenize in)
  (if (input-null? in)
      (input-cons token-newline input-null)
      (let ((ch  (input-car in)))
        (cond ((and (special? ch)
                    (match-input/tokens in))
               => (lambda (entry)
                    (input-cons (cdr entry)
                                (tokenize (input-drop in (string-length (car entry)))))))
              (else
               (receive (lis in) (input-break special? (input-cdr in))
                 (input-cons (list->string (cons ch lis))
                             (tokenize in))))))))

(define *string-token-map*
  `(("\n"     . ,token-newline)
    ("<<"     . ,token-lbracks)
    (">>="    . ,token-rbreqs)
    (">>"     . ,token-rbracks)
    ("@<<"    . "<<")
    ("@>>"    . ">>")
    ("@@"     . "@")
    ("@%def " . "%def ")
    ("@"      . ,token-at)
    ("%def "  . ,token-def)))

(define special?
  (let ((chs (delete-duplicates (map (lambda (entry)
                                       (string-ref (car entry) 0))
                                     *string-token-map*)
                                char=?)))
    (lambda (ch)
      (member ch chs char=?))))

(define (match-input/tokens in)
  (let loop ((entries *string-token-map*))
    (if (null? entries)
        #f
        (let ((str (caar entries)))
          (if (equal? (string->list str)
                      (input-take in (string-length str)))
              (car entries)
```

```
              (loop (cdr entries)))))))
```

Los siguientes procedimientos son algoritmos para entradas análogos a los procedimientos `take`, `drop` y `break` especificados para listas en SRFI-1.

```
(define (input-take in n)
  (if (or (zero? n) (input-null? in))
      '()
      (cons (input-car in)
            (input-take (input-cdr in) (- n 1)))))

(define (input-drop in n)
  (if (or (zero? n) (input-null? in))
      in
      (input-drop (input-cdr in) (- n 1))))

(define (input-break pred in)
  (let loop ((nots '())
             (rest in))
    (if (or (input-null? rest)
            (pred (input-car rest)))
        (values (reverse nots) rest)
        (loop (cons (input-car rest) nots)
              (input-cdr rest)))))
```

1.3.2 *Análisis sintáctico*

El parseo de los tokens se logra programando la gramática de los programas con sintaxis de `sweb`, la especificación de la gramática con la sintaxis de `lib/parcomb.scm` es:

```
<program> => (:+: (:alt: <docs> <code>))

<docs>    => (:alt: (:seq: (:eq: token-at)
                           (:*: (:pred: string-spaces?)))
                           (:eq: token-newline))
                           <docs-lines>)
                    <docs-lines>)

<code>    => (:seq: (:eq: token-lbracks)
                    (:*: (:pred: string?))
                    (:eq: token-rbreqs)
                    (:*: (:pred: string-spaces?))
                    (:eq: token-newline)
                    <code-lines>)

<docs-lines> => (:+: (:seq: (:*: (:alt: (:pred: string?) <refs>))
                            (:eq: token-newline)))
```

```
<code-lines> => (:alt: (:seq: (:*: (:alt: (:pred: string?) <refs>))
                               (:eq: token-newline)
                               <code-lines>)
                         (:seq: (:eq: token-at)
                               (:+: (:pred: string-spaces?))
                               (:eq: token-def)
                               (:+: (:pred: string?))
                               (:eq: token-newline))
                         :succeed:)

<refs>       => (:seq: (:eq: token-lbracks)
                         (:*: (:pred: string?))
                         (:eq: token-rbracks))
```

La implementación concreta de la gramática implementa la construcción de la representación interna usando envolturas de los éxitos de parseo y la construcción de un árbol de razones por las que el parseo pudo fallar usando envolturas de las razones de fallo de parseo.

```
(define (parse tks)
  (<program> tks
             (lambda (s)
               (if (input-null? (pending-input s))
                   (success-match s)
                   (parse (pending-input s))))
             (lambda (f)
               (display "THE PARSING PROCESS FAILED\n")
               (display "==========================\n\n")
               (display "Here is the reason tree of why it may have failed:\n")
               ;; (pp (why-failed f))
               (dump-error-tree (why-failed f))
               (display "\nHere is a textual representation of the pending input:\n")
               (display "---------- input starts here ----------\n")
               (dump-detokenized (pending-input f))
               (display "---------- input ends here ----------\n")
               (error "Parsing failed"))))
```

El siguiente procedimiento revierte la tokenización para facilitar la lectura del archivo que falla parsear.

```
(define (dump-detokenized tks)
  (unless (input-null? tks)
    (let ((tk  (input-car tks)))
      (cond ((eq? token-lbracks tk) (display "<<"))
            ((eq? token-rbreqs tk)  (display ">>="))
            ((eq? token-newline tk) (newline))
            ((eq? token-def tk)     (display "%def "))
            ((eq? token-at tk)      (display "@"))
            ((eq? token-rbracks tk) (display ">>"))
            ((string=? "<<" tk)     (display "@<<"))
            ((string=? ">>" tk)     (display "@>>"))
            ((string=? "@" tk)      (display "@@"))
```

```
          ((string=? "%def " tk)  (display "@%def "))
          (else                   (display tk)))
      (dump-detokenized (input-cdr tks)))))
```

El siguiente procedimiento imprime de manera "*bonita*" un árbol de razones de fallo.

```
(define (dump-error-tree err)
  (define indent-factor 2)
  (define (display-error-lvl err lvl)
    (define spaces (make-string (* lvl indent-factor) #\space))
    (cond ((list? err)
           (for-each (lambda (x)
                       (display-error-lvl x (+ lvl 1)))
                     err))
          ((pair? err)
           (display spaces)
           (display (car err))
           (newline)
           (display-error-lvl (cdr err) (+ lvl 1)))
          ((string? err)
           (display spaces)
           (display err)
           (newline))
          ((procedure? err)
           (display spaces)
           (let ((op (open-output-string)))
             (pp (unsyntax err) op)
             (let ((ip (open-input-string (get-output-string op))))
               (let ((lines (let recur ((line (read-line ip)))
                              (if (eof-object? line)
                                  '()
                                  (cons line (recur (read-line ip)))))))
                 (for-each (lambda (x)
                             (display x)
                             (newline)
                             (display spaces))
                           (drop-right lines 1))
                 (unless (null? lines)
                   (display (last lines))
                   (newline)))
               (close-input-port ip))
             (close-output-port op)))
          (else
           (display spaces)
           (display err)
           (newline))))
  (for-each (lambda (x)
              (display-error-lvl x 0))
            err))
```

```
(define-parser <program>
  (:+: (:alt: <docs> <code>)))

(define-parser <code>
  (wrap-code (:seq: (wrap-code-lbracks (:eq: token-lbracks))
                    (wrap-code-name    (:*: (:pred: string?)))
                    (wrap-code-rbreqs  (:eq: token-rbreqs))
                    (wrap-code-spaces  (:*: (:pred: string-spaces?)))
                    (wrap-code-newline (:eq: token-newline))
                    (wrap-code-lines   <code-lines>))))

(define wrap-code
  (:><: (lambda (match)
          ;; (list (cons 'code match))
          (assert (string? (car match)))
          (assert (list?   (cadr match)))
          (assert (null?   (cddr match)))
          (let ((defs (and-let* ((maybe-defs (last (cadr match)))
                                 ((pair? maybe-defs))
                                 ((eq? 'defs (car maybe-defs)))
                                 ((not (null? (cdr maybe-defs)))))
                        (cdr maybe-defs))))
            (list (make-code (list-ref match 0)
                             defs
                             (if defs
                                 (drop-right (cadr match) 1)
                                 (cadr match))))))
        (lambda (why)
          (list (cons "malformed code chunk"
                      why)))))

(define wrap-code-lbracks
  (:><: (constant '())
        (lambda (why)
          (list (cons "missing << in a starting code line"
                      why)))))

(define wrap-code-name
  (:><: (lambda (match)
          (list (apply string-append match)))
        (lambda (why)
          (list (cons "missing code name in a starting code line"
                      why)))))

(define wrap-code-rbreqs
  (:><: (constant '())
        (lambda (why)
          (list (cons "missing >>= in a starting code line"
```

```
                    why)))))

(define wrap-code-spaces
  (:><: (constant '()) id))

(define wrap-code-newline
  (:><: (constant '())
        (lambda (why)
          (list (cons "missing a new line at the end of a starting code line"
                      why)))))

(define wrap-code-lines
  (:><: (lambda (match) (list match)) id))

(define (string-spaces? x)
  (and (string? x)
       (not (string=? "" x))
       (string=? "" (string-trim x))))

(define-parser <code-lines>
  (:alt: (:seq: (:*: (:alt: (:pred: string?) <refs>))
                (wrap-content-newline (:eq: token-newline))
                <code-lines>)
         (wrap-code-line-end
          (:seq: (wrap-code-line-at      (:eq: token-at))
                 (wrap-code-line-spaces  (:+: (:pred: string-spaces?)))
                 (wrap-code-line-def     (:eq: token-def))
                 (wrap-code-line-names   (:+: (:pred: string?)))
                 (wrap-code-line-newline (:eq: token-newline))))
         :succeed:))

(define wrap-code-line-end
  (:><: (lambda (match)
          (list (cons 'defs match)))
        (lambda (why)
          (cons "malformed code @ %def line"
                why))))

(define wrap-code-line-at
  (:><: (constant '()) id))

(define wrap-code-line-spaces
  (:><: (constant '()) id))

(define wrap-code-line-def
  (:><: (constant '()) id))

(define wrap-code-line-names
  (:><: (lambda (match)
```

```
          (let recur ((chs (string->list (apply string-append match)))
                      (sub '()))
            (cond ((null? chs)
                   (if (null? sub)
                       sub
                       (cons (list->string (reverse sub))
                             '())))
                  ((char=? #\space (car chs))
                   (if (null? sub)
                       (recur (cdr chs) sub)
                       (cons (list->string (reverse sub))
                             (recur (cdr chs) '()))))
                  (else
                   (recur (cdr chs)
                          (cons (car chs) sub))))))
        (lambda (why)
          (cons "missing definitions names in @ %def line"
                why))))

(define wrap-code-line-newline
  (:><: (constant '())
        (lambda (why)
          (list (cons "missing a new line at the end of a @ %def line"
                      why)))))

(define-parser <refs>
  (wrap-refs (:seq: (wrap-refs-lbrack (:eq: token-lbracks))
                    (wrap-refs-name   (:*: (:pred: string?)))
                    (wrap-refs-rbrack (:eq: token-rbracks)))))

(define wrap-refs
  (:><: (lambda (match)
          ;; (list (cons 'refs match))
          (assert (string? (car match)))
          (assert (null?   (cdr match)))
          (list (make-refs (car match))))
        (lambda (why)
          (list (cons "malformed refs"
                      why)))))

(define wrap-refs-lbrack
  (:><: (constant '()) id))

(define wrap-refs-name
  (:><: (lambda (match)
          (list (apply string-append match)))
        id))
```

```
(define wrap-refs-rbrack
  (:><: (constant '())
        (lambda (why)
          (list (cons "bad refs ending, there must be just text and then >>"
                      why)))))

(define-parser <docs>
  (wrap-docs (:alt: (:seq: (wrap-docs-at (:eq: token-at))
                           (wrap-docs-spaces (:*: (:pred: string-spaces?)))
                           (wrap-docs-newline (:eq: token-newline))
                           <docs-lines>)
                    <docs-lines>)))

(define wrap-docs
  (:><: (lambda (match)
          ;; (list (cons 'docs match))
          (assert (list? (car match)))
          (assert (null? (cdr match)))
          (list (make-docs (car match))))
        (lambda (why)
          (list (cons "malformed documentation chunk"
                      why)))))

(define wrap-docs-at
  (:><: (constant '()) id))

(define wrap-docs-spaces
  (:><: (constant '()) id))

(define wrap-docs-newline
  (:><: (constant '()) id))

(define-parser <docs-lines>
  (wrap-docs-lines (:+: (wrap-docs-line-content
                         (:seq: (:*: (:alt: (:pred: string?) <refs>))
                                (wrap-content-newline (:eq: token-newline)))))))

(define wrap-docs-lines
  (:><: (lambda (match)
          (list match))
        (lambda (why)
          (list (cons "there must be at least one valid documentation line"
                      why)))))

(define wrap-docs-line-content
  (:><: id
        (lambda (why)
          (list (cons "malformed documentation line"
```

```
                    why)))))

(define wrap-content-newline
  (:><: (constant '(#\newline))
        (lambda (why)
          (list (cons "missing a new line in chunk content"
                      why)))))
```

1.3.3 *Representación de fragmentos*

```
(define (make-code name defs content)
  (list ':code name defs content))

(define (code? x)
  (and (list? x)
       (eq? ':code (car x))
       (= 4 (length x))
       (string? (list-ref x 1))
       (or (boolean? (list-ref x 2)) (list? (list-ref x 2)))
       (list? (list-ref x 3))))

(define (code-name code)
  (assert (code? code))
  (list-ref code 1))

(define (code-defs code)
  (assert (code? code))
  (list-ref code 2))

(define (code-content code)
  (assert (code? code))
  (list-ref code 3))

(define (make-docs content)
  (list ':docs content))

(define (docs? x)
  (and (list? x)
       (eq? ':docs (car x))
       (= 2 (length x))
       (list? (list-ref x 1))))

(define (docs-content docs)
  (assert (docs? docs))
  (list-ref docs 1))

(define (make-refs name)
  (list ':refs name))
```

```
(define (refs? x)
  (and (list? x)
       (eq? ':refs (car x))
       (= 2 (length x))
       (string? (list-ref x 1))))

(define (refs-name refs)
  (assert (refs? refs))
  (list-ref refs 1))
```

2 (lib/utils.scm)

```
(define (id x)
  x)

(define ((constant x) y)
  x)

(define (negate pred)
  (lambda (x)
    (not (pred x))))

(define (compose f . fs)
  (lambda args
    (let loop ((f  f)
               (fs fs))
      (if (null? fs)
          (apply f args)
          (f (loop (car fs) (cdr fs)))))))

(define (reverse-compose f . fs)
  (lambda args
    (let loop ((res (apply f args))
               (fs  fs))
      (if (null? fs)
          res
          (loop ((car fs) res) (cdr fs))))))

(define (pipe x . fs)
  ((apply reverse-compose fs) x))
```

3 (lib/input.scm)

3.1 *Documentación*

Una entrada (*input*) se define como:

```
input := input-null
       | (input-cons x input)
```

Internamente una entrada se implementa como un flujo (*stream*) pero todo código que haga uso de esta implementación deberá restringirse a usar los siguientes procedimientos:

`input-null?` Predicado para la entrada vacía `input-null`.
`input?` Predicado para las entradas.
`input-car` Selector del primer elemento de una entrada.
`input-cdr` Selector de una entrada sin el primer elemento.
`open-input-from-file` Procedimiento para crear una entrada a partir del nombre de un archivo.
`open-input-from-string` Procedimiento para crear una entrada a partir de una cadena de caracteres.

3.2 *Implementación*

3.2.1 *Macros*

```
(define-syntax input-cons
  (syntax-rules ()
    ((input-cons x in)
     (cons-stream x in))))
```

3.2.2 *Procedimientos*

```
(define input-null (stream))

(define (input-null? x)
  (stream-null? x))

(define (input? x)
  (or (input-null? x)
      (and (pair? x) (promise? (cdr x)))))

(define (input-car in)
  (stream-car in))

(define (input-cdr in)
  (stream-cdr in))

(define (open-input-from-file filename)
  (define ip (open-input-file filename))
  (%input-port->input% ip))

(define (open-input-from-string str)
  (define ip (open-input-string str))
  (%input-port->input% ip))
```

3.2.3 *Procedimientos internos*

(no utilizar fuera de este archivo)

```
(define (%input-port->input% ip)
  (let recur ((ch (read-char ip)))
    (cond ((eof-object? ch)
           (close-input-port ip)
           input-null)
          (else
           (input-cons ch (recur (read-char ip)))))))
```

4 (lib/parcomb.scm)

4.1 *Documentación*

Un parser es un procedimiento que toma tres argumentos:

1. Una entrada `in`
2. Una continuación `ok`
3. Una continuación `bad`

Las *continuaciones* `ok` y `bad` son procedimientos que reciben un resultado de parseo y realizan algún cómputo con él.

Hay dos tipos de resultado de parseo:

- Éxitos
- Fallos

Las continuaciones `ok` reciben un resultado de éxito `s` mientras que las continuaciones `bad` reciben un resultado de fallo `f`.

Un resultado de éxito se puede construír con el procedimiento `success` el cuál toma dos argumentos:

1. Un objeto cualquiera `match` que corresponde la información que fue parseada
2. Una entrada `pending` que corresponde al resto de la entrada que no fue consumida

Un resultado de fallo se puede construír con el procedimiento `failure` el cuál toma dos argumentos:

1. Un objeto cualquiera `why` que corresponde a la información de fallo
2. Una entrada `pending-input` que corresponde a la entrada que no pudo ser parseada

Los procedimientos `success-match`, `why-failed` y `pending-input` obtienen el `match` el `why` y el `pending` de los resultados.

Los combinadores de parsers son procedimientos que toman como argumentos a parsers y regresan un parser.

4.2 *Implementación*

4.2.1 *Parsers*

`:succeed:` siempre parsea la entrada correctamente sin consumirla y el *match* satisface `null?`
`:fail:` siempre parsea la entrada incorrectamente sin consumirla y el *why* es la cadena `"no reason"`

`:empty:` parsea correctamente solo las entradas vacías
`:nempty` parsea correctamente solo las entradas no vacías
`(:pred: pred)` parsea correctamente si el primer elemento de una entrada satisface `pred`
`(:npred: pred)` similar a `:pred:` pero con el predicado negado
`((:cmp: =) x)` produce creadores de parsers que comparan el primer elemento de una entrada con `x` bajo la relación `=`
`((:ncmp: =) x)` similar a `:cmp:` pero negando la relación `=`
`(:equal: x)` equivalente a `(:cmp: equal?)`
`(:eqv: x)` equivalente a `(:cmp: eqv?)`
`(:eq: x)` equivalente a `(:cmp: eq?)`
`(:nequal: x)` equivalente a `(:ncmp: equal?)`
`(:neqv: x)` equivalente a `(:ncmp: eqv?)`
`(:neq: x)` equivalente a `(:ncmp: eq?)`

```
(define (:succeed: in ok bad)
  (ok (success '() in)))

(define (:fail: in ok bad)
  (bad (failure '() in)))

(define (:empty: in ok bad)
  (if (input-null? in)
      (:succeed: in ok bad)
      (bad (failure (list (list "not an empty input"
                                (list (cons "found:" (input-car in))))) in))))

(define (:nempty: in ok bad)
  (if (input-null? in)
      (bad (failure '(("not expecting empty input")) in))
      (:succeed: in ok bad)))

(define ((:pred: pred) in ok bad)
  (cond ((input-null? in)
         (:nempty: in ok bad))
        ((pred (input-car in))
         (ok (success (list (input-car in))
                      (input-cdr in))))
        (else
         (bad (failure (list (list "(pred x) => #f"
                                   (list (cons "x is:" (input-car in))
                                         (cons "pred is:" pred))))
                       in)))))

(define ((:npred: pred) in ok bad)
  (cond ((input-null? in)
         (:nempty: in ok bad))
        ((pred (input-car in))
         (bad (failure (list (list "(pred x) => #t"
                                   (list (cons "x is:" (input-car in))
                                         (cons "pred is:" pred)))))))
        (else
```

```
         (ok (success (list (input-car in))
                      (input-cdr in))))))

(define (((:cmp: =) x) in ok bad)
  (cond ((input-null? in)
         (:nempty: in ok bad))
        ((= x (input-car in))
         (ok (success (list (input-car in))
                      (input-cdr in))))
        (else
         (bad (failure (list (list "x ≠ y"
                                   (list (cons "x is:" x)
                                         (cons "y is:" (input-car in))
                                         (cons "= is:" =))))
                       in)))))

(define :equal: (:cmp: equal?))
(define :eqv: (:cmp: eqv?))
(define :eq: (:cmp: eq?))

(define (((:ncmp: =) x) in ok bad)
  (cond ((input-null? in)
         (:nempty: in ok bad))
        ((= x (input-car in))
         (bad (failure (list (list "x = y"
                                   (list (cons "x is:" x)
                                         (cons "y is:" (input-car in))
                                         (cons "= is:" =))))
                       in)))
        (else
         (ok (success (list (input-car in))
                      (input-cdr in))))))

(define :nequal: (:ncmp: equal?))
(define :neqv: (:ncmp: eqv?))
(define :neq: (:ncmp: eq?))
```

4.2.2 *Combinadores básicos*

(:seq: pars ...) Parsea de manera secuencial la entrada utilizando sus argumentos en orden, corresponde a una concatenación o secuenciación de parsers

(:alt: pars ...) Parsea de manera condicional la entrada utilizando sus argumentos en orden, corresponde a una disyunción o alternativa de parsers y deja de parsear cuando uno de sus parsers es un éxito

(:*: par) Parsea cero o más veces la entrada utilizando el parser par

(:+: par) Parsea una o más veces la entrada utilizando el parser par

```
(define (:seq: . pars)
  (define ((:seq2: par1 par2) in ok bad)
```

```
    (par1 in
          (lambda (s1)
            (par2 (pending-input s1)
                  (lambda (s2)
                    (ok (success (append (success-match s1)
                                         (success-match s2))
                                 (pending-input s2))))
                  (lambda (f2)
                    (bad f2))))
          bad))
  (fold-right :seq2: :succeed: pars))

(define (:alt: . pars)
  (define ((:alt2: par1 par2) in ok bad)
    (par1 in
          ok
          (lambda (f1)
            (par2 in
                  ok
                  (lambda (f2)
                    (bad (failure (append (why-failed f1)
                                          (why-failed f2))
                                  in)))))))
  (fold-right :alt2: :fail: pars))

(define (:*: par)
  (define (par* in ok bad)
    (par in
         (lambda (s1)
           (let ((s2 (par* (pending-input s1) id id)))
             (ok (success (append (success-match s1)
                                  (success-match s2))
                          (pending-input s2)))))
         (lambda (f1)
           (:succeed: in ok bad))))
  par*)

(define (:+: par)
  (:seq: par (:*: par)))
```

4.2.3 *Metacombinadores*

`((:><: fmatch fwhy) par)` Regresa un combinador que utiliza `par` sobre la entrada y aplica `fmatch` al `match` de un éxito y `fwhy` al `why` de un fallo.

```
(define (((:><: fmatch fwhy) par) in ok bad)
  (par in
       (lambda (s)
         (ok (success (fmatch (success-match s))
```

```
                    (pending-input s))))
      (lambda (f)
        (bad (failure (fwhy (why-failed f))
                      (pending-input f))))))
```

4.2.4 *Resultados de parseo*

```
(define (success match pending)
  `(success ,match ,pending))

(define (success? x)
  (and (list? x)
       (= 3 (length x))
       (eq? 'success (car x))))

(define (failure why pending)
  `(failure ,why ,pending))

(define (failure? x)
  (and (list? x)
       (= 3 (length x))
       (eq? 'failure (car x))))

(define (success-match s)
  (assert (success? s))
  (cadr s))

(define (why-failed f)
  (assert (failure? f))
  (cadr f))

(define (pending-input r)
  (assert (or (success? r) (failure? r)))
  (caddr r))
```

4.2.5 *Macros*

Se utilizan los macros `parser` y `define-parser` para admitir la definición recursiva de parsers. Si estas formas no son utilizadas se pueden encontrar errores, considera:

```
> (define foo (:alt: :empty: (:seq: (:eqv: 1) foo)))
```

```
(define-syntax parser
  (syntax-rules ()
    ((parser par)
     (lambda (in ok bad)
       (par in ok bad)))))
```

```
(define-syntax define-parser
  (syntax-rules ()
    ((define-parser name par)
     (define name (parser par)))))
```

5 (doc/gendoc.scm)

Simón

```
(define (gendoc ifilenames ofilename)
  (with-output-to-file ofilename
    (lambda ()
      (for-each (lambda (ifilename)
                  (with-input-from-file ifilename
                    display-reversed-markdown))
                ifilenames))))
```

```
(define (display-reversed-markdown)
  (let loop ((line     (read-line))
             (in-code? #f))
    (unless (eof-object? line)
      (cond ((string=? "#|" line)
             (let write-markdown ((line (read-line)))
               (if (or (eof-object? line)
                       (string=? "|#" line))
                   (loop (read-line) #f)
                   (begin
                     (display line)
                     (newline)
                     (write-markdown (read-line))))))
            ((string=? "#||#" line)
             (when in-code?
               (display "```\n"))
             (loop (read-line) #f))
            ((string=? "" (string-trim line))
             (when in-code?
               (newline))
             (loop (read-line) in-code?))
            (else
             (unless in-code?
               (display "```scheme\n"))
             (display line)
             (newline)
             (loop (read-line) #t))))))
```

```
(define *doc-filenames*
  (list "preamble.scm"
        "../tangle.scm"
```

```
        "../lib/utils.scm"
        "../lib/input.scm"
        "../lib/parcomb.scm"
        "gendoc.scm"))

(define *doc-output* "README.md")

(gendoc *doc-filenames* *doc-output*)
(run-shell-command
 (string-append
  "pandoc"
  " -V lang=es"
  " -V papersize=letter"
  " -V geometry=margin=1in"
  " -V fontsize=11pt"
  " -V mainfont=\"Linux Libertine O\""
  " -V sansfont=\"Linux Biolinum O\""
  " -V monofont=\"Inconsolata LGC\""
  " " *doc-output*
  " --number-sections"
  " --latex-engine=xelatex"
  " -o " *doc-output* ".pdf"))
```