Лабораторная работа № 6. Основы синтаксического и лексического анализа

17 декабря 2023 г.

Дмитрий Лимонов, ИУ9-12Б

Цели работы

Получение навыков реализации лексических анализаторов и нисходящих синтаксических анализаторов, использующих метод рекурсивного спуска. # Реализация

```
;; Конструктор потока
(define (make-stream items . eos)
  (if (null? eos)
      (make-stream items #f)
      (list items (car eos))))

;; Запрос текущего символа
(define (peek stream)
  (if (null? (car stream))
      (cadr stream)
      (caar stream)))

;; Запрос первых двух символов
(define (peek2 stream)
  (if (null? (car stream))
      (cadr stream)
      (if (null? (cdar stream))
          (list (caar stream))
          (list (caar stream) (cadar stream)))))

;; Продвижение вперёд
(define (next stream)
  (let ((n (peek stream)))
    (if (not (null? (car stream)))
        (set-car! stream (cdr (car stream))))
    n))
```

```scheme
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (check-frac str)
  (define stream
    (make-stream (string->list str) (integer->char 0)))

  (call-with-current-continuation
   (lambda (error)
     (ans stream error)
     (equal? (peek stream) (integer->char 0)))))

(define (char-sign? char)
  (or (equal? char #\+) (equal? char #\-)))

; <ans> ::= <sign> <rest> |
;           <rest>
(define (ans stream error)
  (let ((current (peek stream)))
    (and (char-sign? current) (next stream))
    (rest stream error)))


; <rest> ::= <number> '/' <number>
(define (rest stream error)
  (number stream error)
  (and (not (equal? (peek stream) #\/)) (error #f))
  (next stream)
  (number stream error))


; <number> ::= DIGIT <tail>
(define (number stream error)
  (if (char-numeric? (peek stream))
      (begin
        (next stream)
        (tail stream error))
      (error #f)))

; <tail> ::= <empty> | DIGIT <tail>
(define (tail stream error)
  (if (char-numeric? (peek stream))
      (begin
        (next stream)
        (tail stream error))
      (and (not (or (equal? (peek stream) #\/) (equal? (peek stream) (integer->char 0)))) (e
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (list->integer sp)
  (if (null? sp)
      0
      (+ (* (expt 10 (- (length sp) 1)) (- (char->integer (car sp)) 48)) (list->integer (cdr


(define (scan-frac str)
  (define stream
    (make-stream (string->list str) (integer->char 0)))

  (let* ((result
          (call-with-current-continuation
           (lambda (error)
             (ans2 stream error)))))
    (and (equal? (peek stream) (integer->char 0)) result)))

(define (char-sign? char)
  (or (equal? char #\+) (equal? char #\-)))

; <ans> ::= <sign> <rest> |
;           <rest>
; (ans2 stream error) -> (- rest stream error)) | (rest stream error)
(define (ans2 stream error)
  (let ((current (peek stream)))
    (and (char-sign? current) (next stream))
    (define ans (rest2 stream error))
    (if (equal? current #\-)
        (- ans)
        ans)))


; <rest> ::= <number> '/' <number>
;
; (rest2 stream error) -> (number stream error) / (number stream error)
(define (rest2 stream error)
  (define num1 (number2 stream error))
  (and (not (equal? (peek stream) #\/)) (error #f))
  (next stream)
  (define num2 (number2 stream error))
  (/ num1 num2))
```

```scheme
; <number> ::= DIGIT <tail>
(define (number2 stream error)
  (if (char-numeric? (peek stream))
      (list->integer (cons (next stream) (tail2 stream error)))
      (error #f)))

; <tail> ::= <empty> | DIGIT <tail>
(define (tail2 stream error)
  (if (char-numeric? (peek stream))
      (cons (next stream) (tail2 stream error))
      (if (not (or (equal? (peek stream) #\/) (equal? (peek stream) (integer->char 0))))
          (error #f)
          '())))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (skip? char)
  (or
   (equal? char #\newline)
   (equal? char #\tab)
   (equal? char #\space)))

(define (scan-many-fracs str)
  (define stream
    (make-stream (string->list str) (integer->char 0)))

  (let* ((answer (call-with-current-continuation
                  (lambda (error)
                    (filter stream error)))))
    (and (not (null? answer)) answer)))

; <filter> ::= <empty> |
;              <skip> <filter> |
;              <ans> <filter>
(define (filter stream error)
  (if (equal? (peek stream) (integer->char 0))
      '()
      (begin
        (if (skip? (peek stream))
            (begin
              (next stream)
              (filter stream error))
            (cons (ans3 stream error) (filter stream error))))))

(define (char-sign? char)
  (or (equal? char #\+) (equal? char #\-)))
```

```
; <ans> ::= <sign> <rest> |
;           <rest>
; (ans stream error) -> (- rest stream error)) | (rest stream error)
(define (ans3 stream error)
  (let ((current (peek stream)))
    (and (char-sign? current) (next stream))
    (define ans (rest3 stream error))
    (if (equal? current #\-)
        (- ans)
        ans)))


; <rest> ::= <number> '/' <number>
;
; (rest stream error) -> (number stream error) / (number stream error)
(define (rest3 stream error)
  (define num1 (number3 stream error))
  (and (not (equal? (peek stream) #\/)) (error #f))
  (next stream)
  (define num2 (number3 stream error))
  (/ num1 num2))



; <number> ::= DIGIT <tail>
(define (number3 stream error)
  (if (char-numeric? (peek stream))
      (list->integer (cons (next stream) (tail3 stream error)))
      (error #f)))

; <tail> ::= <empty> | DIGIT <tail>
(define (tail3 stream error)
  (if (char-numeric? (peek stream))
      (cons (next stream) (tail3 stream error))
      '()))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;<Program>  ::= <Articles> <Body> .
;<Articles> ::= <Article> <Articles> | .
;<Article>  ::= define word <Body> end .
;<Body>     ::= if <Body> endif <Body> | integer <Body> | word <Body> | .
(define (word? symb)
  (and symb (not (or (equal? symb 'define) (equal? symb 'if) (equal? symb 'endif) (equal? sy
(define (parse vector)
  (define stream (make-stream (vector->list vector)))
  (call-with-current-continuation
   (lambda (error)
     (program stream error))))

(define (program stream error)
  (list (articles stream error) (body stream error)))

(define (articles stream error)
  (let ((art (article stream error)))
    (if art
        (cons art (articles stream error))
        '())))

(define (article stream error)
  (and (equal? (peek stream) 'define)
       (begin
         (next stream)
         (and (not (word? (peek stream))) (error #f))
         (let ((bod (list (next stream) (body stream error))))
           (if (equal? (next stream) 'end)
               bod
               (error #f))))))

(define (body stream error)
  (let ((symb (peek stream)))
    (cond
      ((number? symb) (begin
                        (next stream)
                        (cons symb (body stream error))))
      ((word? symb) (begin
                      (next stream)
                      (cons symb (body stream error))))
      ((equal? symb 'if) (begin
                           (next stream)
                           (let ((bod (list 'if (body stream error))))
                             (if (equal? 'endif (peek stream))
                                 (begin
                                   (next stream)
                                   (cons bod (body stream error)))
                                 (error #f)))))
      (else '()))))
```

Тестирование

```
Welcome to DrRacket, version 8.11 [cs].
Language: R5RS; memory limit: 128 MB.
> (check-frac "110/111")
#t
> (check-frac "-4/3")
#t
> (check-frac "+5/10")
#t
> (check-frac "5.0/10")
#f
> (check-frac "FF/10")
#f
> (scan-frac "110/111")
110/111
> (scan-frac "-4/3")
-4/3
> (scan-frac "+5/10")
1/2
> (scan-frac "5.0/10")
#f
> (scan-frac "FF/10")
#f
> (scan-many-fracs
 "\t1/2 1/3\n\n10/8")
(1/2 1/3 5/4)
> (scan-many-fracs
 "\t1/2 1/3\n\n2/-5")
#f
> (parse #(1 2 +))
(() (1 2 +))
> (parse #(x dup 0 swap if drop -1 endif))
(() (x dup 0 swap (if (drop -1))))
> (parse #( define -- 1 - end
          define =0? dup 0 = end
          define =1? dup 1 = end
          define factorial
              =0? if drop 1 exit endif
              =1? if drop 1 exit endif
              dup --
              factorial
              *
          end
          0 factorial
          1 factorial
          2 factorial
          3 factorial
          4 factorial ))
(((-- (1 -)) (=0? (dup 0 =)) (=1? (dup 1 =)) (factorial (=0? (if (drop 1 exit)) =1? (if (drop 1 exi
 dup -- factorial *))) (0 factorial 1 factorial 2 factorial 3 factorial 4 factorial))
```

Вывод

Лексический и синтаксический анализ - фундаментальный процесс написания собственного языка. Эта работа важна для понимания того, как именно работает язык программирования, каким образом готовый код понимается компьютером. Написание собственного парсера знакомит с нисходящим рекурсивным парсером, широко используемому на практике методу анализа