Лабораторная работа № 6. Основы синтаксического и лексического анализа

19 декабря 2023 г.

Бойко Роман, ИУ9-12Б

Цель работы

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

Реализация

```
(load "../lab3/lab3.scm")
(load "stream.scm")

(define call/cc call-with-current-continuation)

;; Number 1

;; <Fraction> ::= <Signed-num> / <Unsigned-num>
;; <Signed-num> ::= + <Unsigned-num> | - <Unsigned-num> | <Unsigned-num>
;; <Unsigned-num> ::= Digit <Digit-tail>
;; <Digit-tail> ::= Digit <Digit-tail> | <Empty>
;; <Empty> ::=


;; 1.1
(define (check-frac str)
  ;; <Fraction> ::= <Signed-num> / <Unsigned-num>
  (define (check stream error)
    (signed-num stream error)
    (sign stream #\/ error)
    (unsigned-num stream error))

  (define (sign stream term error)
    (if (equal? (peek stream) term)
```

```
      (next stream)
      (error #f)))

  ;; <Signed-num> ::= + <Unsigned-num> | - <Unsigned-num> | <Unsigned-num>
  (define (signed-num stream error)
    (cond ((equal? (peek stream) #\+)
           (next stream)
           (if (unsigned-num stream error)
               #t
               (error #f)))
          ((equal? (peek stream) #\-)
           (next stream)
           (if (unsigned-num stream error)
               #t
               (error #f)))
          ((unsigned-num stream error) #t)
          (else (error #f))))

  ;; <Unsigned-num> ::= Digit <Digit-tail>
  (define (unsigned-num stream error)
    (cond ((and (char? (peek stream)) (char-numeric? (peek stream)))
           (next stream)
           (tail-num stream error))
          (else (error #f))))

  ;; <Digit-tail> ::= Digit <Digit-tail> | <Empty>
  (define (tail-num stream error)
    (cond ((and (char? (peek stream)) (char-numeric? (peek stream)))
           (next stream)
           (tail-num stream error))
          (else #t)))

  (define stream (make-stream (string->list str) 'EOF))

  (call/cc
   (lambda (error)
     (check stream error)
     (equal? (peek stream) 'EOF))))


;; <Fraction> ::= <Signed-num> / <Unsigned-num>
;; <Signed-num> ::= + <Unsigned-num> | - <Unsigned-num> | <Unsigned-num>
;; <Unsigned-num> ::= Digit <Digit-tail>
;; <Digit-tail> ::= Digit <Digit-tail> | <Empty>
;; <Empty> ::=
```

```
;; 1.2
(define (scan-frac str)
  (let ((res '()))

    ;; <Fraction> ::= <Signed-num> / <Unsigned-num>
    (define (scan stream error)
      (signed-num stream error)
      (sign stream #\/ error)
      (unsigned-num stream error))

    (define (sign stream term error)
      (if (equal? (peek stream) term)
          (begin
            (set! res (append res (cons (peek stream) '())))
            (next stream))
          (error #f)))

    ;; <Signed-num> ::= + <Unsigned-num> | - <Unsigned-num> | <Unsigned-num>
    (define (signed-num stream error)
      (cond ((equal? (peek stream) #\+)
             (begin
               (set! res (append res (cons (peek stream) '())))
               (next stream))
             (if (unsigned-num stream error)
                 #t
                 (error #f)))
            ((equal? (peek stream) #\-)
             (begin
               (set! res (append res (cons (peek stream) '())))
               (next stream))
             (if (unsigned-num stream error)
                 #t
                 (error #f)))
            ((unsigned-num stream error) #t)
            (else (error #f))))

    ;; <Unsigned-num> ::= Digit <Digit-tail>
    (define (unsigned-num stream error)
      (cond ((and (char? (peek stream)) (char-numeric? (peek stream)))
             (begin
               (set! res (append res (cons (peek stream) '())))
               (next stream))
             (tail-num stream error))
            (else (error #f))))
```

```
    ;; <Digit-tail> ::= Digit <Digit-tail> | <Empty>
    (define (tail-num stream error)
      (cond ((and (char? (peek stream)) (char-numeric? (peek stream)))
             (begin
               (set! res (append res (cons (peek stream) '())))
               (next stream))
             (tail-num stream error))
            (else #t)))

    (define (print-frac lst)
      (string->number (list->string lst)))

    (define stream (make-stream (string->list str) 'EOF))

    (call/cc
     (lambda (error)
       (scan stream error)
       (if (equal? (peek stream) 'EOF)
           (print-frac res)
           #f)))))


;; <List-of-fractions> ::= <Spaces> <Fraction> <Spaces> <List-of-fractions> | <Empty>
;; <Spaces> ::= SPACE <Spaces> | <Empty>
;; <Fraction> ::= <Signed-num> / <Unsigned-num>
;; <Signed-num> ::= + <Unsigned-num> | - <Unsigned-num> | <Unsigned-num>
;; <Unsigned-num> ::= Digit <Digit-tail>
;; <Digit-tail> ::= Digit <Digit-tail> | <Empty>
;; <Empty> ::=


;; 1.3
(define (scan-many-fracs str)
  (define (clean-string str)
    (define (concat str1 str2)
      (list->string
      (append (string->list str1) (string->list str2))))

    (let loop ((res '())
               (current-string "")
               (data (string->list (concat str " "))))
      (cond ((null? data) res)
            ((and
              (char? (car data))
              (char-whitespace? (car data))
              (not (equal? current-string "")))
```

```
            (loop (append res (cons current-string '()))
                  ""
                  (cdr data)))
           ((and
             (char? (car data))
             (char-whitespace? (car data))
             (equal? current-string ""))
            (loop res
                  current-string
                  (cdr data)))
            ((and
              (char? (car data))
              (not (char-whitespace? (car data))))
             (loop res (concat current-string
                               (make-string 1 (car data)))
                               (cdr data))))))

  (let inner ((data (clean-string str))
        (res '()))
    (cond ((null? data) res)
          ((equal? (scan-frac (car data)) #f) #f)
          (else (inner (cdr data) (append res (cons (scan-frac (car data)) '()))))))))


;; Number 2

;; <Program>  ::= <Articles> <Body> .
;; <Articles> ::= <Article> <Articles> | <Empty> .
;; <Article>  ::= define word <Body> end .
;; <Body>     ::= if <Body> endif <Body> | integer <Body> | word <Body> | <Empty> .


(define (parse tokens)
  (let ((tree '())
        (env '(+ - * / exit
               mod neg drop swap
               dup over rot and
               = > < not or depth)))

    (define (my-element? x xs)
      (cond
        ((null? xs) #f)
        ((equal? x (car xs)) #t)
        (else (my-element? x (cdr xs)))))

    ;; <Program> ::= <Articles> <Body> .
```

```
(define (program stream error)
  (cond ((start-articles? (peek stream))
         (let* ((term-atricles (articles stream error))
                (term-body (parse-body stream error)))
           (list term-atricles term-body)))
        (else (error #f))))

(define (start-articles? token)
  (or (equal? token 'define) #t))

(define (not-forbidden-symb? token)
  (and (not (equal? token 'define))
       (not (equal? token 'if))
       (not (equal? token 'endif))
       (not (equal? token 'end))
       (not (equal? token 'EOF))
       (symbol? token)))

 (define (word? token)
  (and (my-element? token env) (not-forbidden-symb? token)))

;; <Articles> ::= <Article> <Articles> | <Empty> .
(define (articles stream error)
  (cond ((start-article? (peek stream))
         (let* ((term-article (parse-article stream error))
                (term-articles (articles stream error)))
           (cons term-article term-articles)))
        (else '())))

(define (start-article? token)
  (equal? token 'define))

;; <Article> ::= define word <Body> end .
(define (parse-article stream error)
  (cond ((equal? (peek stream) 'define)
         (let* ((term-define (next stream))
                (term-word (if (not-forbidden-symb? (peek stream))
                               (begin
                                 (set! env (cons (peek stream) env))
                                 (next stream))
                               (error #f)))
                (term-body (parse-body stream error))
                (term-end (if (equal? (peek stream) 'end) (next stream) (error #f))))
           (list term-word term-body)))
        (else (error #f))))
```

```
    ;; <Body> ::= if <Body> endif <Body> | integer <Body> | word <Body> | <Empty> .
    (define (parse-body stream error)
      (cond ((equal? (peek stream) 'if)
             (let* ((term-if (next stream))
                    (term-body-head (parse-body stream error))
                    (term-endif (if (equal? (peek stream) 'endif)
                                    (next stream)
                                    (error #f)))
                    (term-body-tail (parse-body stream error)))
               (cons (list term-if term-body-head) term-body-tail)))

            ((number? (peek stream))
             (let* ((term-integer (next stream))
                    (term-body (parse-body stream error)))
               (cons term-integer term-body)))

            ((word? (peek stream))
             (let* ((term-word (next stream))
                    (term-body (parse-body stream error)))
               (cons term-word term-body)))
            (else '())))

    (define stream (make-stream (vector->list tokens) 'EOF))

    (call-with-current-continuation
     (lambda (error)
       (set! tree (program stream error))
       (if (equal? (peek stream) 'EOF) tree #f)))))
```

Тестирование

```
;; Test 1.1
(define check-frac-tests
  (list (test (check-frac "110/111")
              #t)
        (test (check-frac "-4/3")
              #t)
        (test (check-frac "+5/10")
              #t)
        (test (check-frac "5.0/10")
              #f)
        (test (check-frac "FF/10")
              #f)
        (test (check-frac "/")
              #f)
```

```
        (test (check-frac "1/")
              #f)
        (test (check-frac "/1")
              #f)
        (test (check-frac "")
              #f)
        (test (check-frac "+/1")
              #f)
        (test (check-frac "+1 1/1")
              #f)
        (test (check-frac "+56+4/10")
              #f)
        (test (check-frac "-2/1")
              #t)
        (test (check-frac "+1/")
              #f)
        (test (check-frac "-/1")
              #f)))
;; (run-tests check-frac-tests)

;; Test 1.2
(define scan-frac-tests
  (list (test (scan-frac "110/111")
              110/111)
        (test (scan-frac "-4/3")
              -4/3)
        (test (scan-frac "+5/10")
              1/2)
        (test (scan-frac "5.0/10")
              #f)
        (test (scan-frac "FF/10")
              #f)
        (test (scan-frac "/")
              #f)
        (test (scan-frac "1/")
              #f)
        (test (scan-frac "/1")
              #f)
        (test (scan-frac "")
              #f)
        (test (scan-frac "+/1")
              #f)
        (test (scan-frac "+1 1/1")
              #f)
        (test (scan-frac "+56+4/10")
              #f)
```

```
        (test (scan-frac "-2/1")
               -2/1)
        (test (scan-frac "+1/")
              #f)
        (test (scan-frac "-/1")
              #f)))
;; (run-tests scan-frac-tests)

;; Test 1.3
(define scan-many-fracs-tests
  (list (test (scan-many-fracs
               "\t1/2 1/3\n\n10/8")
              '(1/2 1/3 5/4))
        (test (scan-many-fracs
               "\t1/2 1/3\n\n2/-5")
              #f)
        (test (scan-many-fracs
               "\t1/2 1/32/-5")
              #f)))
;; (run-tests scan-many-fracs-tests)

;; Test 2
(define parse-tests
  (list (test (parse #(1 2 +))
              '(() (1 2 +)))
        (test (parse #(x dup 0 swap if drop -1 endif))
              #f)
        (test (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 exit)) dup -
- factorial *)))
          (0 factorial 1 factorial 2 factorial 3 factorial 4 factorial)))
        (test (parse #(define word w1 w2 w3))
              #f)
        (test (parse #(0 if 1 if 2 endif 3 endif 4))
              '(() (0 (if (1 (if (2)) 3)) 4)))
        (test (parse #(define =0? dup 0 = end
                             define gcd
                             =0? if drop exit endif
                             swap over mod
                             gcd
                             end
                             90 99 gcd
                             234 8100 gcd))
              '(((=0? (dup 0 =))
                (gcd (=0?
                      (if (drop exit))
                      swap over mod
                      gcd)))
               (90 99 gcd
                234 8100 gcd)))
        (test (parse #(if define end))
              #f)
        (test (parse #(if end))
              #f)
        (test (parse #(if endif))
              '(() ((if ()))))
        (test (parse #(if define endif end))
              #f)
        (test (parse #(define end end))
              #f)
        (test (parse #(define if end))
              #f)
        (test (parse #(define if  end endif))
              #f)
        (test (parse #())
              '(()()))
        (test (parse #(+ + +))
              '(()(+ + +)))))
;; (run-tests parse-tests)
```

Вывод

Научился реализовывать лексические анализаторы и нисходящие синтаксических анализаторы, использующие метод рекурсивного спуска. Научился работать с БНФ.