Лабораторная работа № 6. Основы синтаксического и лексического анализа

3 января 2024 г.

Семён Чайкин, ИУ9-11Б

Цель работы

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

Задание 1

Реализуйте простейшие сканеры:

- Процедуру `check-frac`, принимающую на вход строку и возвращающую `#t`, если в строке записана простая дробь в формате десятичное-целое-со-знаком/десятичное-целое-без-знака, и `#f` в противном случае.
- Процедуру `scan-frac`, принимающую на вход строку и возвращающую значение, если в строке записана простая дробь в формате десятичное-целое-со-знаком/десятичное-целое-без-знака, и `#f` в противном случае.
- Процедуру `scan-many-fracs`, принимающую на вход строку, содержащую простые дроби, разделенные пробельными символами (строка также может начинаться и заканчиваться произвольным числом пробелов, символов табуляции, перевода строки и др.), и возвращающую список этих дробей. Если разбор невозможен, процедура должна возвращать `#f`.

Грамматика

Грамматика лексера:

```
number ::= DIGIT number-tail
number-tail ::= number | ε
sign ::= + | -
separator ::= /
spaces ::= SPACE spaces | ε
token ::= number | sign | separator
tokens ::= token tokens | spaces tokens | ε
```

Грамматика парсера:

```
signed-number ::= sign signed-number | number
frac          ::= signed-number separator number
fracs         ::= frac fracs | ε
```

Реализация

```
(load "stream.scm")

;; ======================
;; Вспомогательные функции
;; ======================
```

```
(define (all? pred? xs)
  (or (null? xs)
      (and (pred? (car xs))
           (all? pred? (cdr xs)))))

(define char-digit? char-numeric?)

(define (char-sign? ch)
  (and (memq ch '(#\+ #\-)) #t))

(define (char-sep? ch)
  (equal? ch #\/))

(define (char->digit ch)
  (- (char->integer ch)
     (char->integer #\0)))

(define (list->integer xs)
  (define (list->integer-rec xs)
    (if (null? xs)
        0
        (+ (* 10 (list->integer-rec (cdr xs)))
           (char->digit (car xs)))))
  (list->integer-rec (reverse xs)))

;; ======================
;; Лексический анализатор
;; ======================

(define (tokenize str)
  (let* ((EOF (integer->char 0))
         (stream (make-stream (string->list str) EOF)))

    (call-with-current-continuation
     (lambda (error)
       (define result (tokens stream error))
       (and (equal? (peek stream) EOF) result)))))

;; tokens ::= token tokens | spaces tokens | ε
(define (tokens stream error)
  (define (start-token? char)
    (or (char-sign? char)
        (char-sep? char)
        (char-digit? char)))
  (cond
    ((char-whitespace? (peek stream))
     (spaces stream error)
     (tokens stream error))
    ((start-token? (peek stream))
     (cons (token stream error)
           (tokens stream error)))
    (else '())))

;; spaces ::= SPACE spaces | ε
(define (spaces stream error)
  (cond
```

```
    ((char-whitespace? (peek stream))
     (next stream))
    (else #t)))

;; token ::= number | sign | separator
(define (token stream error)
  (cond
    ((char-sign? (peek stream))
     (next stream))
    ((char-sep? (peek stream))
     (next stream))
    ((char-digit? (peek stream))
     (number stream error))
    (else (error #f))))

;; number ::= DIGIT number-tail
(define (number stream error)
  (cond
    ((char-digit? (peek stream))
     (list->integer
      (cons (next stream)
            (number-tail stream error))))
    (else (error #f))))

;; number-tail ::= number | ε
(define (number-tail stream error)
  (cond
    ((char-digit? (peek stream))
     (cons (next stream)
           (number-tail stream error)))
    (else '())))

;; =========================
;; Синтаксический анализатор
;; =========================

(define (parse tokens axiom)
  (define stream
    (make-stream tokens))

  (call-with-current-continuation
   (lambda (error)
     (let ((result (axiom stream error)))
       (and (equal? (peek stream) #f) result)))))

;; fracs ::= frac fracs | ε
(define (fracs stream error)
  (cond
    ((peek stream)
     (cons
      (frac stream error)
      (fracs stream error)))
    (else '())))

;; frac ::= signed-number separator number
(define (frac stream error)
```

```
  (cond
    ((or (char-sign? (peek stream))
         (number? (peek stream)))
     (list (signed-number stream error)
           (if (char-sep? (peek stream))
               (next stream)
               (error #f))
           (non-zero-number stream error)))
    (else (error #f))))

;; signed-number ::= sign signed-number | number
(define (signed-number stream error)
  (cond
    ((char-sign? (peek stream))
     (cons (next stream)
           (signed-number stream error)))
    ((number? (peek stream))
     (list (next stream)))
    (else (error #f))))

(define (non-zero-number stream error)
  (cond
    ((and
      (number? (peek stream))
      (> (peek stream) 0))
     (list (next stream)))
    (else (error #f))))

;; ====================
;; Упрощение выражения
;; ====================

(define (simplify-frac-tree tree)
  (define (list->number list)
    (cond
      ((null? (cdr list)) (car list))
      ((equal? (car list) #\-)
       (- (list->number (cdr list))))
      ((equal? (car list) #\+)
       (list->number (cdr list)))))
  (and tree (/ (list->number (car tree)) (list->number (caddr tree)))))

(define (simplify-fracs tree)
  (and tree (append (map simplify-frac-tree tree))))

;; ================
;; Основные функции
;; ================

;; S ::= frac
(define (check-frac expr)
  (define tkns (tokenize expr))
  (and tkns (parse tkns frac) #t))

;; S ::= frac
(define (scan-frac expr)
```

```
  (define tkns (tokenize expr))
  (and tkns (simplify-frac-tree (parse tkns frac))))

;; S ::= fracs
(define (scan-many-fracs expr)
  (define tkns (tokenize expr))
  (and tkns (simplify-fracs (parse tkns fracs))))
```

Тестирование

Пример тестирования:

```
(load "unit-test.scm")

(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 "2/") #f)
   (test (check-frac "/2") #f)
   (test (check-frac "+/2") #f)
   (test (check-frac "+1/") #f)
   (test (check-frac "-2/1") #t)
   (test (check-frac "-2/1/12") #f)
   (test (check-frac "-2//12") #f)
   (test (check-frac "5/0") #f)
   (test (check-frac "/") #f)
   (test (check-frac "-334//234/4/234/342///4//2342") #f)
   (test (check-frac "-10/0") #f)
   ))

(define scan-frac-tests
  (list
   (test (scan-frac "110/111") 110/111)
   (test (scan-frac "-4/3") -4/3)
   (test (scan-frac "+-----+---+-5/10") -5/10)
   (test (scan-frac "5.0/10") #f)
   (test (scan-frac "FF/10") #f)
   (test (scan-frac "-2/1") -2/1)
   (test (scan-frac "-20/123") -20/123)
   ))

(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 "") '())
   (test (scan-many-fracs " 1/2 ") '(1/2))
   (test (scan-many-fracs "1/2-1/2") '(1/2 -1/2))
```

```
   (test (scan-many-fracs "1/2 -1/2") '(1/2 -1/2))
   (test (scan-many-fracs "") '())
   (test (scan-many-fracs " ") '())
   (test (scan-many-fracs "\t\t\t\t\t\t    1/2     ") '(1/2))
   (test (scan-many-fracs " /2 ") #f)
   (test (scan-many-fracs " 2/ ") #f)
   (test (scan-many-fracs "1/2 -2/1") '(1/2 -2/1))
   (test (scan-many-fracs "   1/2  ") '(1/2))
   (test (scan-many-fracs "1234/234 2345/432 \t\t\n343/324") '(1234/234 2345/432 343/324))
   (test (scan-many-fracs "1/2 ** 2/3") #f)
   ))

(define tests
  (append check-frac-tests scan-frac-tests scan-many-fracs-tests))

(run-tests tests)
```

Результат в консоли:

```
(check-frac "110/111") ok
(check-frac "-4/3") ok
(check-frac "+5/10") ok
(check-frac "5.0/10") ok
(check-frac "FF/10") ok
(check-frac "/") ok
(check-frac "1/") ok
(check-frac "/1") ok
(check-frac "") ok
(check-frac "+/1") ok
(check-frac "+1 1/1") ok
(check-frac "2/") ok
(check-frac "/2") ok
(check-frac "+/2") ok
(check-frac "+1/") ok
(check-frac "-2/1") ok
(check-frac "-2/1/12") ok
(check-frac "-2//12") ok
(check-frac "5/0") ok
(check-frac "/") ok
(check-frac "-334//234/4/234/342///4//2342") ok
(check-frac "-10/0") ok
(scan-frac "110/111") ok
(scan-frac "-4/3") ok
(scan-frac "+-----+---+-5/10") ok
(scan-frac "5.0/10") ok
(scan-frac "FF/10") ok
(scan-frac "-2/1") ok
(scan-frac "-20/123") ok
(scan-many-fracs "\t1/2 1/3\n\n10/8") ok
(scan-many-fracs "") ok
(scan-many-fracs " 1/2 ") ok
(scan-many-fracs "1/2-1/2") ok
(scan-many-fracs "1/2 -1/2") ok
(scan-many-fracs "") ok
(scan-many-fracs " ") ok
(scan-many-fracs "\t\t\t\t\t\t    1/2     ") ok
(scan-many-fracs " /2 ") ok
```

```
(scan-many-fracs " 2/ ") ok
(scan-many-fracs "1/2 -2/1") ok
(scan-many-fracs "   1/2  ") ok
(scan-many-fracs "1234/234 2345/432 \t\t\n343/324") ok
(scan-many-fracs "1/2 ** 2/3") ok
#t
```

Задание 2

Реализуйте процедуру `parse`, осуществляющую разбор программы на модельном языке, представленной в виде последовательности (вектора) токенов (см. Лабораторную работу №4 «Интерпретатор стекового языка программирования»). Процедура `parse` должна включать в себя реализацию синтаксического анализа последовательности токенов методом рекурсивного спуска согласно следующей грамматике:

```
<Program> ::= <Articles> <Body>
<Articles> ::= <Article> <Articles> | ε
<Article> ::= define word <Body> end
<Body> ::= if <Body> endif <Body> | integer <Body> | word <Body> | ε
```

Процедура должна возвращать синтаксическое дерево в виде вложенных списков, соответствующих нетерминалам грамматики. В случае несоответствия входной последовательности грамматике процедура должна возвращать `#f`.

Грамматика

Грамматика парсера:

```
Program ::= Articles Body
Articles ::= Article Articles | ε
Article ::= define word Program end ε
Body ::= if Body ElsePart endif Body | integer Body | word Body | ε
ElsePart ::= else Body | ε
```

Реализация

```
(load "stream.scm")

;; =======================
;; Вспомогательные функции
;; =======================

(define keywords
  '(+ - * / mod neg
      = > < not and or
      drop swap dup over rot depth))

(define reserved
  '(define end if endif else))

(define (integer-token? token)
  (number? token))

(define (reserved-token? token)
  (and (memq token reserved) #t))

(define (keyword-token? token)
  (and (memq token keywords) #t))
```

```
(define (variable-token? token)
  (and (not (keyword-token? token))
       (not (reserved-token? token))
       (not (integer-token? token))))

(define (word-token? token)
  (and token
       (or (keyword-token? token)
           (variable-token? token))))

;; =========================
;; Синтаксический анализатор
;; =========================

(define (parse tokens)
  (define stream
    (make-stream (vector->list tokens)))
  (define axiom program)

  (call-with-current-continuation
   (lambda (error)
     (let ((result (axiom stream error)))
       (and (equal? (peek stream) #f) result)))))

;; Program ::= Articles Body ε
(define (program stream error)
  (list (articles stream error)
        (body stream error)))

;; Articles ::= Article Articles | ε
(define (articles stream error)
  (cond
    ((equal? (peek stream) 'define)
     (cons (article stream error)
           (articles stream error)))
    (else '())))

;; Article ::= define word Program end ε
(define (article stream error)
  (if (equal? (peek stream) 'define)
      (next stream)
      (error #f))

  (define result
    (append (if (word-token? (peek stream))
                (list (next stream))
                (error #f))
            (program stream error)))

  (if (equal? (peek stream) 'end)
      (next stream)
      (error #f))

  result)

;; Body ::= if Body ElsePart endif Body | integer Body | word Body | ε
```

```
(define (body stream error)
  (cond
    ((equal? (peek stream) 'if) (if-expr stream error))
    ((integer-token? (peek stream)) (integer-expr stream error))
    ((word-token? (peek stream)) (word-expr stream error))
    (else '())))

(define (if-expr stream error)
  (define result
    (append
     (if (equal? (peek stream) 'if)
         (list (next stream))
         (error #f))

     (list (body stream error))
     (else-expr stream error)))

  (if (equal? (peek stream) 'endif)
      (begin (next stream) '())
      (error #f))

  (cons result (body stream error)))

;; ElsePart ::= else Body | ε
(define (else-expr stream error)
  (cond
    ((equal? (peek stream) 'else)
     (next stream)
     (list (body stream error)))
    (else '())))

(define (word-expr stream error)
  (cons (if (word-token? (peek stream))
            (next stream)
            (error #f))
        (body stream error)))

(define (integer-expr stream error)
  (cons (if (integer-token? (peek stream))
            (next stream)
            (error #f))
        (body stream error)))
```

Тестирование

```
Welcome to DrRacket, version 8.10 [cs].
Language: R5RS; memory limit: 128 MB.
> (parse #(define a define b 1 end define c 2 end b c end a + 1 2 -))
(((a ((b () (1)) (c () (2))) (b c))) (a + 1 2 -))

> (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))

> (parse #(if a b else c d endif))
(() ((if (a b) (c d))))

> (parse #(if a else endif))
(() ((if (a) ())))

> (parse #(define mod end define mod1 end mod mod1))
(((mod () ()) (mod1 () ())) (mod mod1))

> (parse #(define a define b 1 end define c 2 end b c d end if a mod 123 endif))
(((a ((b () (1)) (c () (2))) (b c d))) ((if (a mod 123))))

> (parse #(if endif))
(() ((if ())))

> (parse #(define a define b body end end a))
(((a ((b () (body))) ())) (a))

> (parse #(define x if end))
#f

> (parse #(if -- exit else end endif))
#f
```

Вывод

Научился составлять простейшую грамматику для лексических и синтаксических анализаторов, ознакомился и разобрался с реализацией синтаксического анализатора, использующего метод рекурсивного спуска.