Лабораторная работа № 6. Основы синтаксического и лексического анализа

2 декабря 2024 г.

Александр Яннаев, ИУ9-11Б

Цель работы

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

Задания

1. Сканер: Входным языком является строка, содержащая последовательность десятичных дробей, имеющих вид `десятичное-целое-со-знаком.десятичные-цифры`, разделённых нулём и более пробельных символов (число может начинаться на ноль): `3.1415  10001.1  -7.0  +22.4-48.1` Пробелы обязательны лишьв случае, когда слитное написание приводит к противоречиям, т.е. когда вторая дробь не начинается на знак. Следует составить БНФ и написать следующие процедуры:
   - `valid-dec?` — предикат, принимающий строку, и возвращающий `#t` для корректной записи дроби, иначе `#f`.
   - `valid-many-decs?` — предикат, принимающий строку, и возвращающий `#t` для корректной записи последовательности дробей.
   - `scan-dec` — процедура, принимающая на вход строку, и возвращающая число или `#f` в противном случае.
   - `scan-many-decs` — процедура, принимающая на вход строку и возвращающая список чисел или `#f`.
2. Грамматика:

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

Реализация

```
;; Функции для проверки
(define (digit? c)
  (and (char>=? c #\0) (char<=? c #\9)))

(define (sign? c)
  (or (char=? c #\+) (char=? c #\-)))

;; Проверка корректности дробного числа
(define (valid-dec? str)
  (let loop ((chars (string->list str)) (state 'start) (has-dot? #f) (has-fraction? #f))
    (cond
      ((null? chars) (and has-dot? has-fraction?))
      ((eq? state 'start)
       (cond
         ((sign? (car chars))
          (if (or (null? (cdr chars)) (char=? (car (cdr chars)) #\.))
              #f
              (loop (cdr chars) 'int has-dot? has-fraction?)))
         ((digit? (car chars)) (loop (cdr chars) 'int has-dot? has-fraction?))
         (else #f)))
      ((eq? state 'int)
       (cond
         ((digit? (car chars)) (loop (cdr chars) 'int has-dot? has-fraction?))
         ((char=? (car chars) #\.)
          (if (null? (cdr chars)) #f
              (loop (cdr chars) 'frac #t has-fraction?)))
         (else #f)))
      ((eq? state 'frac)
       (cond
         ((digit? (car chars)) (loop (cdr chars) 'frac has-dot? #t))
         (else #f))))))

;; Проверяет строку на валидность последовательности дробей
(define (valid-many-decs? str)
  (let ((tokens (split-string str)))
    (if (null? tokens)
        #f
        (let loop ((tokens tokens))
          (if (null? tokens)
              #t
              (and (valid-dec? (car tokens))
                   (loop (cdr tokens)))))))))

;; Разделение строки на токены
```

```
(define (split-string str)
  (let loop ((chars (string->list str)) (tokens '()) (current-token '()))
    (cond
      ((null? chars)
       (if (null? current-token)
           (reverse tokens)
           (reverse (cons (list->string (reverse current-token)) tokens))))
      ((and (or (char=? (car chars) #\+) (char=? (car chars) #\-))
            (not (null? current-token)))
       (loop chars
             (cons (list->string (reverse current-token)) tokens)
             '()))
      ((or (char=? (car chars) #\space) (char=? (car chars) #\tab) (char=? (car chars) #\new
       (if (null? current-token)
           (loop (cdr chars) tokens '())
           (loop (cdr chars) (cons (list->string (reverse current-token)) tokens) '())))
      (else
       (loop (cdr chars) tokens (cons (car chars) current-token))))))

;; Преобразование строки в число
(define (string->number-custom str)
  (let loop ((chars (string->list str)) (state 'int) (result 0.0) (sign 1) (fraction-scale 1
    (cond
      ((null? chars) (exact->inexact (* sign result)))
      ((eq? state 'int)
       (cond
         ((char=? (car chars) #\+) (loop (cdr chars) 'int result sign fraction-scale))
         ((char=? (car chars) #\-) (loop (cdr chars) 'int result -1 fraction-scale))
         ((digit? (car chars))
          (loop (cdr chars) 'int
                (+ (* result 10) (- (char->integer (car chars))
                                    (char->integer #\0)))
                sign fraction-scale))
         ((char=? (car chars) #\.) (loop (cdr chars) 'frac result sign 10))
         (else #f)))
      ((eq? state 'frac)
       (cond
         ((digit? (car chars))
          (loop (cdr chars) 'frac
                (+ result (/ (- (char->integer (car chars))
                                (char->integer #\0))
                             fraction-scale))
                sign (* fraction-scale 10)))
         (else #f))))))

;; Разбор одной дроби
```

```
(define (scan-dec str)
  (if (valid-dec? str)
      (string->number-custom str)
      #f))

;; Разбор последовательности дробей
(define (scan-many-decs str)
  (let ((tokens (split-string str)))
    (let loop ((tokens tokens) (results '()))
      (if (null? tokens)
          (reverse results)
          (let ((value (scan-dec (car tokens))))
            (if value
                (loop (cdr tokens) (cons value results))
                #f))))))

;; Утилиты для работы с потоком токенов
(define (make-stream lst) (list lst '()))
(define (peek s) (if (null? (car s)) #f (car (car s))))
(define (pop s)
  (let ((next (peek s)))
    (if next (set-car! s (cdr (car s))))
    next))

;; Разбор <Body> ::= if <Body> <ElsePart> endif <Body> | integer <Body> | word <Body> | .
(define (parse-body s)
  (let* (
          (borders (list 'endif 'end))
          (parsed
              (cond
                  ((or (not (peek s)) (member (peek s) borders)) '())
                  ((equal? (peek s) 'if)
                   (let ((if-result (parse-conditional s)))
                     (if if-result
                         (append (list if-result) (parse-body s))
                         #f)))
                  ((or (integer? (peek s)) (symbol? (peek s)))
                   (cons (pop s) (parse-body s)))
                  (else #f))))
    (if (or (not parsed) (member #f parsed)) #f parsed)))

;; Разбор if <Body> <ElsePart> endif
(define (parse-conditional s)
  (let* (
      (opening (if (equal? (peek s) 'if) (pop s) #f))
      (condition (if opening (parse-body s) #f))
```

```
      (else-part (if (and condition (equal? (peek s) 'else)) (parse-elsepart s) '()))
      (endif (if (and condition (equal? (peek s) 'endif)) (pop s) #f)))
    (if endif (list 'if condition else-part) #f)))

;; Разбор <ElsePart> ::= else <Body> | .
(define (parse-elsepart s)
  (if (equal? (peek s) 'else)
      (let ((else-keyword (pop s))
            (else-body (parse-body s)))
        (if else-body
            (cons else-keyword else-body)
            #f))
      '()))

;; Разбор <Article> ::= define word <Body> end
(define (parse-article s)
  (let* (
      (define-keyword (if (equal? (peek s) 'define) (pop s) #f))
      (name (if (and define-keyword (symbol? (peek s))) (pop s) #f))
      (body (if name (parse-body s) #f))
      (end-keyword (if (and body (equal? (peek s) 'end)) (pop s) #f)))
    (if end-keyword (list 'define name body) #f)))

;; Разбор <Articles> ::= <Article> <Articles> | .
(define (parse-articles s)
  (if (equal? (peek s) 'define)
      (let ((curr (parse-article s))
            (next (parse-articles s)))
        (if (or (not curr) (not next)) #f (cons curr next)))
      '()))

;; Разбор <Program> ::= <Articles> <Body>
(define (parse-program s)
  (let* (
      (articles (parse-articles s))
      (body (if articles (parse-body s) #f)))
    (if (and body (not (member #f body)) (not (member #f articles)))
        (list articles body)
        #f)))

;; Основные функции
(define (parse s)
  (parse-program (make-stream (vector->list s))))

(define (valid? s)
  (if (parse s) #t #f))
```

```
;; testing...
(valid-dec? "7234.4")  ;⇒ #t
(valid-dec? "+07.70")  ;⇒ #t
(valid-dec? "-4.111")  ;⇒ #t
(valid-dec? "+.7777")  ;⇒ #f
(valid-dec? "5555.")   ;⇒ #f
(valid-dec? "10")      ;⇒ #f
(newline)

(valid-many-decs?
 "\t1.8 -2.4\n\n5.0")  ;⇒ #t
(valid-many-decs?
 "\t1.8 -2.4\n\n5.-")  ;⇒ #f
(valid-many-decs?
 "-568.3+77.1")        ;⇒ #t
(newline)

(scan-dec "7234.4")    ;⇒ 7234.4
(scan-dec "+07.70")    ;⇒ 7.7
(scan-dec "-4.111")    ;⇒ -4.111
(scan-dec "+.7777")    ;⇒ #f
(scan-dec "5555.")     ;⇒ #f
(scan-dec "10")        ;⇒ #f
(newline)


(scan-many-decs
 "\t1.8 -2.4\n\n5.0")  ;⇒ (1.8 -2.4 5.0)
(scan-many-decs
 "\t1.8 -2.4\n\n5.-")  ;⇒ #f
(scan-many-decs
 "-568.3+77.1")        ;⇒ (-568.3 77.1)
(newline)


(valid? #(1 2 +))                ;⇒ #t
(valid? #(define 1 2 end))       ;⇒ #f
(valid? #(define x if end endif)) ;⇒ #f
(newline)

(parse #(1 2 +)) ;⇒ (() (1 2 +))
(parse #(x dup 0 swap if drop -1 endif))
    ;⇒ (() (x dup 0 swap (if (drop -1))))
```

```
(parse #(x dup 0 swap if drop -1 else swap 1 + endif))
    ;□ (() (x dup 0 swap (if (drop -1) (swap 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 exit)) dup -- factorial *)))
  (0 factorial 1 factorial 2 factorial 3 factorial 4 factorial))|#

(parse #( define -- 1 - end
          define =0? dup 0 = end
          define =1? dup 1 = end
          define factorial
              =0? if
                  drop 1
              else =1? if
                  drop 1
              else
                  dup --
                  factorial
                  *
              endif
              endif
          end
          0 factorial
          1 factorial
          2 factorial
          3 factorial
          4 factorial ))
```

```
 #|
 (((-- (1 -))
   (=0? (dup 0 =))
   (=1? (dup 1 =))
   (factorial
    (=0? (if (drop 1)(=1? (if (drop 1) (dup -- factorial *)))))))
  (0 factorial 1 factorial 2 factorial 3 factorial 4 factorial))|#

(parse #(define word w1 w2 w3)) ;  #f
```

Тестирование

```
Welcome to DrRacket, version 8.15 [cs].
Language: R5RS; memory limit: 128 MB.
#t
#t
#t
#f
#f
#f

#t
#f
#t

7234.4
7.7
-4.111
#f
#f
#f

(1.8 -2.4 5.0)
#f
(-568.3 77.1)

#t
#f
#f

(() (1 2 +))
(() (x dup 0 swap (if (drop -1) ())))
(() (x dup 0 swap (if (drop -1 else swap 1 +) ())))
(((define -- (1 -)) (define =0? (dup 0 =)) (define =1? (dup 1 =)) (define factorial (=0?
(if (drop 1 exit) ()) =1? (if (drop 1 exit) ()) dup -- factorial *)))
```

```
 (0 factorial 1 factorial 2 factorial 3 factorial 4 factorial))
(((define -- (1 -)) (define =0? (dup 0 =)) (define =1? (dup 1 =)) (define factorial (=0? (if
(drop 1 else =1? (if (drop 1 else dup -- factorial *) ())) ())))))
 (0 factorial 1 factorial 2 factorial 3 factorial 4 factorial))
#f
>
```

Вывод

В ходе выполнения лабораторной работы я научился разрабатывать программы для анализа и обработки текстовых данных, используя метод рекурсивного спуска. Мне было интересно разбираться с грамматикой языка, определять её правила и реализовывать функции для разбора конструкций. Я освоил работу с потоками данных, такими как токены, и использовал утилиты для их последовательной обработки, включая функции `peek` и `pop`.

Особенно увлекательно было реализовывать разбор конструкций `if ... else ... endif`, а также обрабатывать вложенные структуры. Я научился проверять корректность чисел и последовательностей дробей, что углубило моё понимание работы с данными. Разделение логики на модули помогло сделать код более структурированным и понятным.

Эта работа позволила мне не только изучить базовые алгоритмы, но и почувствовать себя создателем мини-интерпретатора. Это сделало процесс изучения программирования ещё более интересным и вдохновляющим.