Лабораторная работа № 6. Основы синтаксического и лексического анализа

23 декабря 2024 г.

Булат Валиуллин, ИУ9-11Б

Цель работы

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

Индивидуальный вариант

Задание 1. Сканер, входной язык: строка, содержащая последовательность обыкновенных дробей, имеющих вид десятичное-целое-со-знаком/десятичное-целое-без-знака, разделённых нулём и более пробельных символов (целое число может начинаться на ноль):

Задание 2. Грамматика: ::=

. ::=

| .

::= define word

end .

::= if

endif

| while

do

wend

| integer

| word

.

Реализация

```scheme
;; Грамматика для дробей
;; <fraction> ::= <signed-integer> '/' <integer>
;; <signed-integer> ::= '+' <integer> | '-' <integer> | <integer>
;; <integer> ::= <digit> | <digit> <integer>
;; <digit> ::= '0' | '1' | '2' | ... | '9'
;; <fractions> ::= <fraction> <whitespaces> <fractions> | <fraction> <fractions>
;; <whitespaces> ::= ' ' | ' ' <whitespaces>

;; <digit> ::= '0' | '1' | '2' | ... | '9'
(define (digit? char)
  (and (>= (char->integer char) (char->integer #\0))
       (<= (char->integer char) (char->integer #\9))
       )
  )

;; <whitespaces> ::= ' ' | ' ' <whitespaces>
(define (delete-whitespaces chars)
  (if (and (not (null? chars))
           (or (= (char->integer (car chars)) (char->integer #\space))
               (= (char->integer (car chars)) (char->integer #\tab))
               (= (char->integer (car chars)) (char->integer #\newline))))
      (delete-whitespaces (cdr chars))
      chars))

;; <integer> ::= <digit> | <digit> <integer>
(define (parse-integer chars)
  (if (null? chars)
      (list #f chars)
      (let ((ch (car chars)))
        (if (digit? ch)
            (let loop ((chars chars) (acc 0))
              (if (and (not (null? chars)) (digit? (car chars)))
                  (loop (cdr chars)
                        (+ (* acc 10)
                           (- (char->integer (car chars)) (char->integer #\0))))
                  (list acc chars)))
            (list #f chars)))))

;; <signed-integer> ::= '+' <integer> | '-' <integer> | <integer>
(define (parse-signed-integer chars)
  (if (null? chars)
      (list #f chars)
      (let* ((first (car chars))
             (sign (cond
```

```scheme
                    ((= (char->integer first) (char->integer #\-)) -1)
                    ((= (char->integer first) (char->integer #\+)) 1)
                    (else 1)))
             (chars (if (or (= (char->integer first) (char->integer #\+))
                            (= (char->integer first) (char->integer #\-)))
                       (cdr chars)
                       chars)))
        (let ((num (car (parse-integer chars)))
              (rest (cadr (parse-integer chars))))
          (if (number? num)
              (list (* sign num) rest)
              (list #f chars))))))

;; <fraction> ::= <signed-integer> '/' <integer>
(define (parse-fraction chars)
  (let ((num (car (parse-signed-integer chars)))
        (rest1 (cadr (parse-signed-integer chars))))
    (if (and (number? num)
             (not (null? rest1))
             (= (char->integer (car rest1)) (char->integer #\/)))
        (let ((den (car (parse-integer (cdr rest1))))
              (rest2 (cadr (parse-integer (cdr rest1)))))
          (if (and (number? den) (not (= 0 den)))
              (list (cons num den) rest2)
              (list #f chars)))
        (list #f chars))))

;; <fractions> ::= <fraction> <whitespaces> <fractions> | <fraction> | ε
(define (parse-fractions chars)
  (let ((chars (delete-whitespaces chars)))
    (if (null? chars)
        (list '() chars) ; выход из рекурсии
        (let ((frac (car (parse-fraction chars)))
              (rest (cadr (parse-fraction chars))))
          (if (not frac)
              (list #f chars)
              (let* ((rest2 (delete-whitespaces rest))
                     (fractions (car (parse-fractions rest2)))
                     (tail (cadr (parse-fractions rest2))))
                (if (eq? fractions #f)
                    (list #f chars)
                    (list (cons frac fractions) tail))))))))

(define (valid-frac? str)
  (pair? (car (parse-fraction (string->list str)))))
```

```scheme
(define (valid-many-fracs? str)
  (pair? (car (parse-fractions (string->list str)))))

(define (simplify-fraction num den)
  (let ((div (gcd (abs num) (abs den))))
    (cons (/ num div) (/ den div))))


(define (scan str)
  (let* ((chars (delete-whitespaces (string->list str)))
         (num (car (parse-signed-integer chars)))
         (rest (cadr (parse-signed-integer chars))))
    (if (and (number? num) rest (not (null? rest))
             (= (char->integer (car rest)) (char->integer #\/)))
        (let ((denom (car (parse-integer (cdr rest))))
              (rem (cadr (parse-integer (cdr rest)))))
          (if (and (number? denom) (not (= denom 0)))
              (list (simplify-fraction num denom) rem)
              (list #f chars)))
        (list #f chars))))

(define (scan-frac str)
  (let ((res (scan str)))
    (and (list? res) (not (null? res)) (pair? (car res))
         (let ((num-den (car res)))
           (and (number? (car num-den))
                (/ (car num-den) (cdr num-den))
                ))
         )))

(define (scan-many-fracs str)
  (let loop ((chars (delete-whitespaces (string->list str))) (result '()))
    (if (null? chars)
        (reverse result)
        (let ((parsed (scan (list->string chars))))
          (and
           (car parsed)
           (loop (delete-whitespaces (cadr parsed))
                 (cons (/ (car (car parsed)) (cdr (car parsed))) result)))))))

;;-----------------------------------------------------------------------
;; <Program>  ::= <Articles> <Body> .
;; <Articles> ::= <Article> <Articles> | .
;; <Article>  ::= define word <Body> end .
;; <Body>     ::= if <Body> endif <Body>
;;              | while <Body> do <Body> wend <Body>
```

```
;;               | integer <Body>
;;               | word <Body>
;;               | .

;; <Body>     ::= if <Body> endif <Body>
;;               | while <Body> do <Body> wend <Body>
;;               | integer <Body>
;;               | word <Body>
;;               | .
;; <Body>     ::= if <Body> endif <Body>
;;               | while <Body> do <Body> wend <Body>
;;               | integer <Body>
;;               | word <Body>
;;               | .
(define (parse-body tokens dict ind)

  (define (parse-next-el tokens dict ind)
    (if (>= ind (vector-length tokens))
        (list '() ind)
        (let ((token (vector-ref tokens ind)))
          (cond
            ;; if <Body> endif <Body>
            ((equal? token 'if)
             (let ((if-body-res (parse-body tokens dict (+ ind 1))))
               (and if-body-res
                    (let ((if-body (car if-body-res))
                          (ind2 (cadr if-body-res)))
                      (and (< ind2 (vector-length tokens))
                           (equal? (vector-ref tokens ind2) 'endif)
                           (let ((after-if-res
                                  (parse-body tokens dict (+ ind2 1))))
                             (and after-if-res
                                  (let ((after-if-body (car after-if-res))
                                        (ind3 (cadr after-if-res)))
                                    (list (cons (list 'if if-body)
                                                after-if-body) ind3))))
                           )
                      )
                    )
               )
             )

            ;; while <Body> do <Body> wend <Body>
            ((equal? token 'while)
             (let ((cond-body-res (parse-body tokens dict (+ ind 1))))
               (and cond-body-res
```

```
        (let ((cond-body (car cond-body-res))
              (ind2 (cadr cond-body-res)))
          (and (and (< ind2 (vector-length tokens))
                    (equal? (vector-ref tokens ind2) 'do))
               (let ((do-body-res (parse-body tokens dict
                                              (+ ind2 1))))
                 (and do-body-res
                      (let ((do-body (car do-body-res))
                            (ind3 (cadr do-body-res)))
                        (and (and (< ind3 (vector-length tokens))
                                  (equal? (vector-ref tokens ind3)
                                          'wend))
                             (let ((after-wend-res
                                    (parse-body tokens
                                                dict (+ ind3 1))))
                               (and after-wend-res
                                    (let ((after-wend-body
                                           (car after-wend-res))
                                          (ind4
                                           (cadr after-wend-res)))
                                      (list (cons
                                             (list 'while
                                                   cond-body
                                                   do-body)
                                             after-wend-body) ind4))
                                    )
                               )
                             )
                        )
                      )
                 )
               )
          )
        )
   )
 )


((or (equal? token 'endif)
     (equal? token 'end)
     (equal? token 'wend)
     (equal? token 'do))
 (list '() ind))

;; integer <Body>
((integer? token)
```

```
                  (let ((after-integer-res (parse-body tokens dict (+ ind 1))))
                    (and after-integer-res
                         (list (cons token (car after-integer-res))
                               (cadr after-integer-res))
                         )
                    )
                  )

                 ;; word <Body>
                 ((symbol? token)
                  (and (not (member token '(define end if endif while do wend)))
                       (let ((after-word-res (parse-body tokens dict (+ ind 1))))
                         (and after-word-res
                              (list (cons token (car after-word-res))
                                    (cadr after-word-res))
                              )
                         )
                       )
                  )
                 )
               )
             )
         )

  (if (>= ind (vector-length tokens))
      (list '() ind)
      (parse-next-el tokens dict ind)))



;; <Article>  ::= define word <Body> end .
(define (parse-article tokens dict ind)
  (and (and (< ind (vector-length tokens))
            (equal? (vector-ref tokens ind) 'define))
       (let ((ind1 (+ ind 1)))
         (and (and (< ind1 (vector-length tokens)) (symbol?
                                                     (vector-ref tokens ind1)))
              (let ((new-word (vector-ref tokens ind1)))
                (let ((body-res (parse-body tokens
                                            (cons new-word dict) (+ ind1 1))))
                  (and body-res
                       (let ((body-parsed (car body-res))
                             (ind2 (cadr body-res)))
                         (and (< ind2 (vector-length tokens))
                              (equal? (vector-ref tokens ind2) 'end)
                              (let ((res (list (list new-word body-parsed)
```

```
                                                    (cons new-word dict)
                                                    (+ ind2 1))))
                                 res)
                               )
                          )
                        )
                   )
                 )
               )
          )
        )
   )


;; <Articles> ::= <Article> <Articles> | .
(define (parse-articles tokens dict ind)
  (let ((article-res (parse-article tokens dict ind)))
    (if article-res
        (let* ((article-parsed (car article-res))
               (dict2 (cadr article-res))
               (ind2 (caddr article-res)))
          (let ((articles-res (parse-articles tokens dict2 ind2)))
            (if articles-res
                (let ((res (list (cons article-parsed (car articles-res))
                                 (cadr articles-res)
                                 (caddr articles-res))))
                  res)
                (let ((res (list (list article-parsed) dict2 ind2)))
                  res))))
        (let ((res (list '() dict ind)))
          res))))


;; <Program>  ::= <Articles> <Body> .
(define (parse-program tokens dict ind)
  (let ((articles-res (parse-articles tokens dict ind)))
    (and articles-res
         (let* ((articles-ast (car articles-res))
                (dict2 (cadr articles-res))
                (ind2 (caddr articles-res))
                (body-res (parse-body tokens dict2 ind2)))
           (and body-res
                (let ((body-ast (car body-res))
                      (ind3 (cadr body-res)))
                  (if (null? articles-ast)
                      (list (list articles-ast body-ast) ind3)
```

```
                      (list (list (append articles-ast (list body-ast))) ind3)))
                )
           )
         )
    )
  )


(define (parse tokens)
  (let ((res (parse-program tokens '() 0)))
    (and (and res (pair? res))
         (let ((parsed (car res))
               (end-ind (cadr res)))
           (and (= end-ind (vector-length tokens))
                parsed
                )
           )
         )
    )
  )



(define (valid? tokens)
  (not (not (parse tokens))))
```

Тестирование

```
Welcome to DrRacket, version 8.15 [cs].
Language: R5RS; memory limit: 128 MB.
(valid-frac? 110/111) ok
(valid-frac? -4/3) ok
(valid-frac? +5/10) ok
(valid-frac? 5.0/10) ok
(valid-frac? FF/10) ok
(valid-many-fracs?  1/2 1/3

10/8) ok
(valid-many-fracs?  1/2 1/3

2/-5) ok
(valid-many-fracs? +1/2-3/4) 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-many-fracs    1/2 1/3

10/8) ok
(scan-many-fracs    1/2 1/3

2/-5) ok
(scan-many-fracs +1/2-3/4) ok
(valid? #(1 2 +)) ok
(valid? #(define 1 2 end)) ok
(valid? #(define x if end endif)) ok
(parse #(1 2 +)) ok
(parse #(x dup 0 swap if drop -1 endif)) ok
(parse #(x dup while dup 0 > do 1 - swap over * swap wend)) ok
(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 1 swap while dup 0 > do
          1 - swap over * swap wend drop end 0 factorial 1
          factorial 2 factorial 3 factorial 4 factorial))
(parse #(define word w1 w2 w3)) ok
#t
```

Вывод

Как там предыдущая лабораторная работа, сложной была, да? Это я просто эту в тот момент не видел. Теперь вижу вот в выводе плейсхолдер “пишите, чему научились” и первое, что в голову приходит: научился понимать, что даже что-то страшное и непонятное сделать можно. Ну а если касательно именно этой лабораторной работы, то тут я понял, что такое метод рекурсивного спуска в нисходящих синтаксических анализаторах, а также как примерно реализуются лексические анализаторы.