Лабораторная работа №6

Table of Contents

1. Цель работы

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

2. Задания

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. Запрещается применять встроенную процедуру для преобразования строки к числу.

```
<дробь>           ::= <число-со-знаком> / <число-без-знака>
<число-со-знаком> ::= + <число-без-знака> | - <число-без-знака> | <число-без-знака>
<число-без-знака> ::= ЦИФРА <хвост-числа>
<хвост-числа>     ::= ЦИФРА <хвост-числа> | <пусто>
<пусто>           ::=
```

```
(define (check-frac str)
  (load "appendix/parser/stream.scm")
  (define (expect stream term error)
    (if (equal? (peek stream) term)
        (next stream)
        (error #f)))
  (define (frac stream error)
    (signed-num stream error)
    (expect stream #\/ error)
    (unsigned-num stream error))
  (define (signed-num stream error)
```

```
    (cond ((equal? #\+ (peek stream))
           (next stream)
           (unsigned-num stream error))
          ((equal? #\- (peek stream))
           (next stream)
           (unsigned-num stream error))
          (else (unsigned-num stream error))))
  (define (unsigned-num stream error)
    (cond ((and (char? (peek stream))
                (char-numeric? (peek stream)))
           (next stream)
           (num-tail stream error))
          (else (error #f))))
  (define (num-tail stream error)
    (cond ((and (char? (peek stream))
                (char-numeric? (peek stream)))
           (next stream)
           (num-tail stream error))
          (else #t)))
  (define stream (make-stream (string->list str) 'EOF))
  (call-with-current-continuation
   (lambda (error)
     (frac stream error)
     (eqv? (peek stream) 'EOF))))
```

```
(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)))

(run-tests check-frac-tests)
```

```
Test 1: (check-frac "110/111") ok
Test 2: (check-frac "-4/3") ok
Test 3: (check-frac "+5/10") ok
Test 4: (check-frac "5.0/10") ok
Test 5: (check-frac "FF/10") ok
```

- 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. Запрещается применять встроенные процедуры преобразования строки к числу к исходной строке до её посимвольной обработки сканером.

```
<дробь>           ::= <число-со-знаком> / <число-без-знака>
<число-со-знаком> ::= + <число-без-знака> | - <число-без-знака> | <число-без-знака>
<число-без-знака> ::= ЦИФРА <хвост-числа>
<хвост-числа>     ::= ЦИФРА <хвост-числа> | <пусто>
<пусто>           ::=
```

```
(define (scan-frac str)
  (load "appendix/parser/stream.scm")
  (define (expect stream term error)
    (if (equal? (peek stream) term)
        (next stream)
        (error #f)))
  (define (frac stream error)
    (define numerator (signed-num stream error))
    (expect stream #\/ error)
    (/ numerator
       (unsigned-num stream error)))
  (define (signed-num stream error)
    (cond ((equal? #\+ (peek stream))
           (next stream)
           (unsigned-num stream error))
          ((equal? #\- (peek stream))
           (next stream)
           (- (unsigned-num stream error)))
          (else (unsigned-num stream error))))
  (define (unsigned-num stream error)
    (cond ((and (char? (peek stream))
                (char-numeric? (peek stream)))
           (string->number (list->string
                            (cons (next stream)
                                  (num-tail stream error)))))
          (else (error #f))))
  (define (num-tail stream error)
    (cond ((and (char? (peek stream))
                (char-numeric? (peek stream)))
           (cons (next stream)
                 (num-tail stream error)))
          (else '())))
  (define stream (make-stream (string->list str) 'EOF))
  (call-with-current-continuation
```

```
   (lambda (error)
     (define res (frac stream error))
     (and (eqv? (peek stream) 'EOF)
          res))))
```

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

(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)))

(run-tests scan-frac-tests)
```

```
Test 1: (scan-frac "110/111") ok
Test 2: (scan-frac "-4/3") ok
Test 3: (scan-frac "+5/10") ok
Test 4: (scan-frac "5.0/10") ok
Test 5: (scan-frac "FF/10") ok
```

- 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 не возможен, процедура должна возвращать `#f`. Запрещается применять встроенные процедуры преобразования строки к числу к исходной строке до её посимвольной обработки сканером.

```
<список-дробей>   ::= <пробелы> <дробь> <пробелы> <список-дробей> | <пусто>
<пробелы>         ::= ПРОБЕЛЬНЫЙ-СИМВОЛ <пробелы> | <пусто>
<дробь>           ::= <число-со-знаком> / <число-без-знака>
<число-со-знаком> ::= + <число-без-знака> | - <число-без-знака> | <число-без-знака>
<число-без-знака> ::= ЦИФРА <хвост-числа>
<хвост-числа>     ::= ЦИФРА <хвост-числа> | <пусто>
<пусто>           ::=
```

```
(define (scan-many-fracs str)
  (load "appendix/parser/stream.scm")
```

```
(define (expect stream term error)
  (if (equal? (peek stream) term)
      (next stream)
      (error #f)))
(define (frac-list stream error)
  (cond ((and (char? (peek stream))
              (or (char-whitespace? (peek stream))
                  (equal? (peek stream) #\+)
                  (equal? (peek stream) #\-)
                  (char-numeric? (peek stream))))
         (spaces stream error)
         (let ((new-frac (frac stream error)))
           (spaces stream error)
           (cons new-frac (frac-list stream error))))
        (else '())))
(define (spaces stream error)
  (cond ((and (char? (peek stream))
              (char-whitespace? (peek stream)))
         (next stream)
         (spaces stream error))
        (else #t)))
(define (frac stream error)
  (define numerator (signed-num stream error))
  (expect stream #\/ error)
  (/ numerator
     (unsigned-num stream error)))
(define (signed-num stream error)
  (cond ((equal? #\+ (peek stream))
         (next stream)
         (unsigned-num stream error))
        ((equal? #\- (peek stream))
         (next stream)
         (- (unsigned-num stream error)))
        (else (unsigned-num stream error))))
(define (unsigned-num stream error)
  (cond ((and (char? (peek stream))
              (char-numeric? (peek stream)))
         (string->number (list->string
                          (cons (next stream)
                                (num-tail stream error)))))
        (else (error #f))))
(define (num-tail stream error)
  (cond ((and (char? (peek stream))
              (char-numeric? (peek stream)))
         (cons (next stream)
               (num-tail stream error)))
        (else '())))
(define stream (make-stream (string->list str) 'EOF))
(call-with-current-continuation
```

```
   (lambda (error)
     (define res (frac-list stream error))
     (and (eqv? (peek stream) 'EOF)
          res))))
```

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

(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 1: (scan-many-fracs "\t1/2 1/3\n\n10/8") ok
Test 2: (scan-many-fracs "\t1/2 1/3\n\n2/-5") ok
Test 3: (scan-many-fracs "\t1/2 1/32/-5") ok
```

Примеры вызова процедур:

```
(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
```

В начале текста программы, в комментариях, обязательно запишите грамматику в БНФ или РБНФ, которую реализуют ваши сканеры.

Рекомендация. Символ, маркирующий конец последовательности, выберете исходя из того, что на вход вашего лексера может поступить любая последовательность символов из таблицы ASCII, встречающаяся в текстовых файлах.

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> | .
<Article>  ::= define word <Body> end .
<Body>     ::= if <Body> endif <Body> | integer <Body> | word <Body> | .
```

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

```
(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 exit)) dup -- factorial *)))
```

```
      (0 factorial 1 factorial 2 factorial 3 factorial 4 factorial))

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

```
(define (parse tokens)
  (load "appendix/parser/stream.scm")
  (define (expect stream term error)
    (if (equal? (peek stream) term)
        (next stream)
        (error #f)))
  (define (program stream error)
    (let* ((t-articles (articles stream error))
           (t-body (body stream error)))
      (list t-articles t-body)))
  (define (articles stream error)
    (cond ((eqv? 'define (peek stream))
           (let* ((t-article (article stream error))
                  (t-articles (articles stream error)))
             (cons t-article t-articles)))
          (else '())))
  (define (article stream error)
    (let* ((t-define (expect stream 'define error))
           (t-word (next stream))
           (t-body (body stream error))
           (t-end (expect stream 'end error)))
      (list t-word t-body)))
  (define (body stream error)
    (cond ((eqv? 'if (peek stream))
           (let* ((t-if (next stream))
                  (t-body (body stream error))
                  (t-endif (expect stream 'endif error))
                  (t-body-tail (body stream error)))
             (cons (list 'if t-body) t-body-tail)))
          ((integer? (peek stream))
           (let* ((t-integer (next stream))
                  (t-body-tail (body stream error)))
             (cons t-integer t-body-tail)))
          ((and (symbol? (peek stream))
                (not (eqv? (peek stream) 'endif))
                (not (eqv? (peek stream) 'end)))
           (let* ((t-word (next stream))
                  (t-body-tail (body stream error)))
             (cons t-word t-body-tail)))
          (else '())))
  (define stream (make-stream (vector->list tokens) "EOF"))
  (call-with-current-continuation
   (lambda (error)
```

```
    (define res (program stream error))
    (and (eqv? (peek stream) "EOF")
         res))))
```

Подготовьте еще 2-3 примера для демонстрации. Обратите внимание, что грамматика позволяет записывать на исходном языке вложенные конструкции `if .. endif`. Учтите эту особенность при реализации парсера и продемонстрируйте её на примерах.

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

(define parse-tests
  (list (test (parse #(1 2 +))
              (() (1 2 +)))
        (test (parse #(x dup 0 swap if drop -1 endif))
              (() (x dup 0 swap (if (drop -1)))))
        (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)))))

(run-tests parse-tests)
```

```
Test 1: (parse #(1 2 +)) ok
Test 2: (parse #(x dup 0 swap if drop -1 endif)) ok
Test 3: (parse #(define -- 1 - end define =0? dup 0 = end define =1? dup 1 = end define factorial =0? if drop 1 (
Test 4: (parse #(define word w1 w2 w3)) ok
Test 5: (parse #(0 if 1 if 2 endif 3 endif 4)) ok
Test 6: (parse #(define =0? dup 0 = end define gcd =0? if drop exit endif swap over mod gcd end 90 99 gcd 234 810
```

Как изменится грамматика, если допустить вложенные статьи?

- <Body> в определении <Arcticle> заменится на <Program>:

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

Author: Starovoytov Alexandr
Created: 2021-12-12 Sun 15:56