Домашнее задание №6

Table of Contents

Реализуйте разбор арифметического выражения, записанного в традиционной инфиксной нотации, и его преобразование к выражению, записанному на языке Scheme. Выражения могут включать в себя числа (целые и с плавающей запятой, в том числе — в экспоненциальной форме), переменные (имена переменных могут состоять из одной или нескольких латинских букв), арифметические операции (`+`, `-`, `*`, `/` и `^`), унарный минус и круглые скобки.

Задание выполните в три этапа:

1. 1. Лексический анализатор

Разработайте лексическую структуру языка (грамматику разбора арифметического выражения на токены) и запишите ее в БНФ перед главной процедурой лексера. Процедура должна называться `tokenize`, принимать выражение в виде строки и возвращать последовательность токенов в виде списка. Лексемы исходной последовательности должны быть преобразованы:

имена переменных и знаки операций — в символические константы Scheme, числа — в числовые константы Scheme соответствующего типа, скобки — в строки `"("` и `")"`. Пробельные символы должны игнорироваться лексером. Если исходная последовательность включает в себя недопустимые символы, лексер должен возвращать `#f`.

Примеры вызова лексера:

```
(tokenize "1")
  ⇒ (1)

(tokenize "-a")
```

```
    ⇒ (- a)

  (tokenize "-a + b * x^2 + dy")
    ⇒ (- a + b * x ^ 2 + dy)

  (tokenize "(a - 1)/(b + 1)")
    ⇒ ("(" a - 1 ")" / "(" b + 1 ")")
```

1.1. БНФ

```
<выражение>         ::= <пробелы> <объект> <пробелы> <выражение> | <пусто>
<пробелы>           ::= ПРОБЕЛ <пробелы> | <пусто>
<объект>            ::= (
                      | )
                      | +
                      | -
                      | *
                      | /
                      | ^
                      | <переменная>
                      | <число>
<число>             ::= ЦИФРА <хвост-числа>
<хвост-числа>       ::= ЦИФРА <хвост-числа> | e <хвост-числа> | . <хвост-числа> | <пусто>
<переменная>        ::= БУКВА <хвост-переменной>
<хвост-переменной> ::= БУКВА <хвост-переменной> | <пусто>
<пусто>             ::=
```

1.2. Реализация

```
(define (tokenize str)
  (load "appendix/parser/stream.scm")
  (define (expression stream error)
    (cond ((and (char? (peek stream))
                (or (char-whitespace? (peek stream))
                    (member (peek stream)
                            '(#\( #\) #\+ #\- #\* #\/ #\^))
                    (char-alphabetic? (peek stream))
                    (char-numeric? (peek stream))))
           (spaces stream error)
           (let* ((t-object (object stream error))
                  (t-spaces (spaces stream error))
                  (t-expr (expression stream error)))
             (cons t-object t-expr)))
          (else '())))
```

```
(define (spaces stream error)
  (cond ((and (char? (peek stream))
              (char-whitespace? (peek stream)))
         (next stream)
         (spaces stream error))
        (else '())))
(define (object stream error)
  (cond ((assoc (peek stream) '((#\( "(")
                                (#\) ")")
                                (#\+ +)
                                (#\- -)
                                (#\* *)
                                (#\/ /)
                                (#\^ ^))) => (lambda (ret)
                                               (next stream)
                                               (cadr ret)))
        ((and (char? (peek stream))
              (char-alphabetic? (peek stream)))
         (variable stream error))
        ((and (char? (peek stream))
              (char-numeric? (peek stream)))
         (number stream error))
        (else (error #f))))
(define (number stream error)
  (cond ((and (char? (peek stream))
              (char-numeric? (peek stream)))
         (let* ((n (next stream))
                (n-tail (number-tail stream error)))
           (string->number (list->string (cons n n-tail)))))
        (else (error #f))))
(define (number-tail stream error)
  (cond ((and (char? (peek stream))
              (or (char-numeric? (peek stream))
                  (char=? #\e (peek stream))
                  (char=? #\. (peek stream))))
         (let* ((n (next stream))
                (n-tail (number-tail stream error)))
           (cons n n-tail)))
        (else '())))
(define (variable stream error)
  (cond ((and (char? (peek stream))
              (char-alphabetic? (peek stream)))
         (let* ((letter (next stream))
                (var-tail (variable-tail stream error)))
           (string->symbol (list->string (cons letter var-tail)))))
        (else (error #f))))
(define (variable-tail stream error)
  (cond ((and (char? (peek stream))
              (char-alphabetic? (peek stream)))
```

```
          (let* ((letter (next stream))
                 (var-tail (variable-tail stream error)))
            (cons letter var-tail)))
         (else '())))
  (define stream (make-stream (string->list str) 'EOF))
  (call-with-current-continuation
   (lambda (error)
     (define res (expression stream error))
     (and (eqv? (peek stream) 'EOF)
          res))))
```

1.3. Тесты

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

(define tokenize-tests
  (list (test (tokenize "1")
              (1))
        (test (tokenize "-a")
              (- a))
        (test (tokenize "-a + b * x^2 + dy")
              (- a + b * x ^ 2 + dy))
        (test (tokenize "(a - 1)/(b + 1)")
              ("(" a - 1 ")" / "(" b + 1 ")"))))

(run-tests tokenize-tests)
```

```
Test 1: (tokenize "1") ok
Test 2: (tokenize "-a") ok
Test 3: (tokenize "-a + b * x^2 + dy") ok
Test 4: (tokenize "(a - 1)/(b + 1)") ok
```

2. 2. Синтаксический анализатор

Синтаксический анализатор должен строить дерево разбора согласно следующей грамматике, учитывающей приоритет операторов:

```
  Expr    ::= Term Expr' .
  Expr'   ::= AddOp Term Expr' | .
  Term    ::= Factor Term' .
  Term'   ::= MulOp Factor Term' | .
  Factor  ::= Power Factor' .
```

```
Factor' ::= PowOp Power Factor' | .
Power   ::= value | "(" Expr ")" | unaryMinus Power .
```

- *Кажется, в предложенной грамматике неверный приоритет у унарного минуса:*

  ```
  -10^2 -> (-10)^2
  ```

где терминалами являются `value` (число или переменная), круглые скобки и знаки операций.

Синтаксический анализатор реализуйте в виде процедуры `parse`, принимающую последовательность токенов в виде списка (результат работы `tokenize`) и возвращающую дерево разбора, представленное в виде вложенных списков вида `(операнд-1 знак-операции операнд-2)` для бинарных операций и `(- операнд)` для унарного минуса. Числа и переменные в списки не упаковывайте. Многократно вложенные друг в друга списки из одного элемента вида `(((имя-или-число)))` не допускаются.

Разбор осуществляйте методом рекурсивного спуска. Если исходная последовательность не соответствует грамматике, парсер должен возвращать `#f`.

При построении дерева разбора соблюдайте общепринятую ассоциативность бинарых операторов: левую для сложения, вычитания, умножения и деления и правую для возведения в степень. Вложенность списков должна однозначно определять порядок вычисления значения выражения.

Примеры вызова парсера:

```
; Ассоциативность левая
;
(parse (tokenize "a/b/c/d"))
  ⇒ (((a / b) / c) / d)

; Ассоциативность правая
;
(parse (tokenize "a^b^c^d"))
  ⇒ (a ^ (b ^ (c ^ d)))

; Порядок вычислений задан скобками
;
(parse (tokenize "a/(b/c)"))
  ⇒ (a / (b / c))

; Порядок вычислений определен только
; приоритетом операций
;
(parse (tokenize "a + b/c^2 - d"))
  ⇒ ((a + (b / (c ^ 2))) - d)
```

2.1. Реализация

```
(define (parse tokens)
  (load "appendix/parser/stream.scm")
  (define (expr stream error)
    (let loop ((res (term stream error)))
      (cond ((or (eqv? '+ (peek stream))
                 (eqv? '- (peek stream)))
             (let* ((op (next stream))
                    (t-term (term stream error)))
               (loop (list res op t-term))))
            (else res))))
  (define (term stream error)
    (let loop ((res (factor stream error)))
      (cond ((or (eqv? '* (peek stream))
                 (eqv? '/ (peek stream)))
             (let* ((op (next stream))
                    (t-factor (factor stream error)))
               (loop (list res op t-factor))))
            (else res))))
  (define (factor stream error)
    (let* ((t-power (power stream error))
           (t-factor1 (factor1 stream error)))
      (if (null? t-factor1)
          t-power
          (cons t-power t-factor1))))
  (define (factor1 stream error)
    (cond ((eqv? '^ (peek stream))
           (next stream)
           (let* ((t-power (power stream error))
                  (t-factor1 (factor1 stream error)))
             (if (null? t-factor1)
                 (cons '^ (list t-power))
                 (list '^ (cons t-power t-factor1)))))
          (else '())))
  (define (power stream error)
    (cond ((number? (peek stream))
           (next stream))
          ((equal? "(" (peek stream))
           (next stream)
           (let ((t-expr (expr stream error)))
             (next stream)
             t-expr))
          ((eqv? '- (peek stream))
           (next stream)
           (list '- (power stream error)))
```

```
          ((symbol? (peek stream))
           (next stream))
          (else (error #f))))
  (define stream (make-stream tokens "EOF"))
  (call-with-current-continuation
   (lambda (error)
     (define res (expr stream error))
     (and (equal? (peek stream) "EOF")
          res))))
```

2.2. Тесты

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

(define parse-tests
  (list (test (parse (tokenize "a + b + c+d"))
              (((a + b) + c) + d))
        (test (parse (tokenize "a/b/c/d"))
              (((a / b) / c) / d))
        (test (parse (tokenize "a^b^c^d"))
              (a ^ (b ^ (c ^ d))))
        (test (parse (tokenize "a/(b/c)"))
              (a / (b / c)))
        (test (parse (tokenize "a + b/c^2 - d"))
              ((a + (b / (c ^ 2))) - d))
        (test (parse (tokenize "(-a)^1e10"))
              ((- a) ^ 1e10))))

(run-tests parse-tests)
```

```
Test 1: (parse (tokenize "a + b + c+d")) ok
Test 2: (parse (tokenize "a/b/c/d")) ok
Test 3: (parse (tokenize "a^b^c^d")) ok
Test 4: (parse (tokenize "a/(b/c)")) ok
Test 5: (parse (tokenize "a + b/c^2 - d")) ok
Test 6: (parse (tokenize "(-a)^1e10")) ok
```

3. 3. Преобразователь дерева разбора в выражение на Scheme

Реализуйте процедуру `tree->scheme`, преобразующую дерево, возвращенное процедурой `parse`, в выражение на языке Scheme. Полученное выражение должно быть пригодно для вычисления его значения интерпретатором языка Scheme.

Для возведения в степень используйте встроенную процедуру Scheme `expt`. Не передавайте более двух аргументов встроенным процедурам для арифметических операций.

Примеры вызова конвертера:

```
  (tree->scheme (parse (tokenize "x^(a + 1)")))
    ⇒ (expt x (+ a 1))

  (eval (tree->scheme (parse (tokenize "2^2^2^2")))
        (interaction-environment))
    ⇒ 65536
```

3.1. Реализация

```
(define (tree->scheme tree)
  (cond ((not (list? tree))
         tree)
        ((eqv? '- (car tree))
         (list '- (tree->scheme (cadr tree))))
        ((eqv? '^ (cadr tree))
         (list 'expt
               (tree->scheme (car tree))
               (tree->scheme (caddr tree))))
        (else (list (cadr tree)
                    (tree->scheme (car tree))
                    (tree->scheme (caddr tree))))))
```

3.2. Тесты

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

(define tree->scheme-tests
  (list (test (tree->scheme (parse (tokenize "x^(a + 1)")))
              (expt x (+ a 1)))
        (test (eval (tree->scheme (parse (tokenize "2^2^2^2")))
                    (interaction-environment))
              65536)))

(run-tests tree->scheme-tests)
```

```
Test 1: (tree->scheme (parse (tokenize "x^(a + 1)"))) ok
Test 2: (eval (tree->scheme (parse (tokenize "2^2^2^2"))) (interaction-environment)) ok
```

Author: Starovoytov Alexandr
Created: 2021-12-13 Mon 16:48