УДК 004.42

Павлов Дмитрий Алексеевич

РАЗРАБОТКА ЯЗЫКА ПРОГРАММИРОВАНИЯ НА RACKET

Аннотация

Статья подробно описывает процесс создания простейшего предметно-ориентированного языка на платформе Racket. Присутствуют лексический анализатор, синтаксический анализатор, интерактивная среда, компиляция, отладка, бектрейсы, подсветка синтаксиса. Созданный язык интегрируется в среду разработки DrRacket. Весь проект занимает менее 300 строк кода.

Ключевые слова: предметно-ориентированный язык, парсинг, компиляция, Racket.

Статья в 6-м выпуске журнала за 2011 г. [1] была посвящена пользе предметно-ориентированных языков и основным подходам к их созданию. В этой статье будет детально показан процесс создания предметно-ориентированного языка на основе Racket – одного из современных языков программирования, ведущего свою родословную от языка LISP.

Пара слов о Лиспе: будучи созданным 50 лет назад, он живёт и развивается до сих пор, причём последние 25 лет практически без финансирования. Это означает, что язык действительно хорош. Надо отметить, что Джон Маккарти (автор термина «искусственный интеллект», AI) изобрёл Лисп как подручное средство для создания этого самого AI. Маккарти умер, AI как не было так и нет (см. «AI Winter»), но идеи, заложенные в Лисп, в течение полувека находят применение в совершенно не связанных с AI областях. Это означает, что идеи действительно хороши.

Цель авторов Racket можно сформулировать так: дать разработчикам возможность создавать свои языки, не лишая их при этом прелестей Лиспа. «Лисп как основа и любой ваш каприз вдобавок» – звучит как предложение, от которого невозможно отказаться. В отличие от Common Lisp, где новые языки, как правило, создаются на базе грамматики S-выражений, в Racket для них допустима любая грамматика. Добавим к этому, что отладчик и редактор с подсветкой синтаксиса – достаются практически даром.

В качестве «подопытного кролика» для демонстрации возможностей Racket мы возьмём простейший язык, который назовём «калькулятор» (`calc`). Вот как выглядит программа на этом языке:

```
X = 2.8
Y = (X + 1) * 5
print X / Y
```

Кроме простого запуска программ, у нас будет пошаговое выполнение, бектрейсы и средство интерактивной разработки, так называемый read-eval-print loop, или REPL.

То, что относится к языкам, Racket желает видеть в одной из своих «системных» директорий, поэтому директорию, в которой мы ведём разработку (назовём её `work`), лучше с самого начала внести в соответствующий список. В среде DrRacket это делается следующим образом: меню Language → Choose Language → кнопка «Show Details» → раздел «Collection Paths» → Add. В директории `work` у нас будет директория `calc`, куда мы поместим файлы, относящиеся к новому языку.

Этап 1: базовый комплект

Начнём с лексического анализатора, который будет в файле `calc/lexer.rkt`:

```
#lang racket

(require
  ; lex — стандартное средство для построения лексических анализаторов
  parser-tools/lex
  ; Библиотека регулярных выражений для lex. Символы, импортированные из неё,
  ; будут предваряться знаком двоеточия, в целях избежания конфликтов имён
  (prefix-in : parser-tools/lex-sre))

; Указанные вещи из этого модуля идут на экспорт
(provide value-tokens op-tokens
         position-line position-col position-offset
         calc-lexer)

; Регулярные выражения для букв, цифр и всяких пробелов
(define-lex-abbrevs
  (lex:letter (:or (:/ #\a #\z) (:/ #\A #\Z)))
  (lex:digit (:/ #\0 #\9))
  (lex:whitespace (:or #\newline #\return #\tab #\space #\vtab)))

; Токены, имеющие значения — идентификатор (например X) и число (например 2.8)
(define-tokens value-tokens (IDENTIFIER NUMBER))

; Токены, не имеющие значений — арифметические операции, скобки, присваивание,
; операция печати (PRINT) и специальный токен EOF
(define-empty-tokens op-tokens
  (EOF ASSIGN PLUS MINUS MULTIPLY DIVIDE LEFT-PAREN RIGHT-PAREN PRINT))

; calc-lexer — это функция, полученная в результате вызова lexer-src-pos
(define calc-lexer
  ; lexer-src-pos отличается от lexer тем, что выдаёт не просто токены,
  ; а position-token — токены с информацией об их положении в исходном файле
  (lexer-src-pos
   ; пробелы и прочее пропускаем, рекурсивно запуская lexer дальше
   ((:+ lex:whitespace) (return-without-pos (calc-lexer input-port)))
   ; токены операций
   ("=" (token-ASSIGN))
   ("+" (token-PLUS))
   ("-" (token-MINUS))
   ("*" (token-MULTIPLY))
   ("/" (token-DIVIDE))
   ("(" (token-LEFT-PAREN))
   (")" (token-RIGHT-PAREN))
   ("print" (token-PRINT))
   ; идентификатор = буква + [произвольный набор букв и цифр]
```

```
   ((:: lex:letter (:* (:or lex:letter lex:digit)))
    (token-IDENTIFIER (string->symbol lexeme)))
   ; число = [знак] + не менее одной цифры + [точка [с цифрами]]
   ((:: (:? #\-) (:+ lex:digit) (:? (:: #\. (:* lex:digit))))
    (token-NUMBER (string->number lexeme)))
   ; специальный токен EOF необходимо вернуть при достижении конца файла
   ((eof) 'EOF)))
```

Теперь парсер, `calc/parser.rkt`:

```
#lang racket

(require
  ; модуль с генератором парсеров
  parser-tools/yacc
  ; модуль, содержащий нужную нам функцию raise-read-error
  syntax/readerr
  ; наш лексический анализатор
  calc/lexer)

(provide calc-read-syntax
         calc-read)

; Функция, которую мы будем вызывать при ошибке парсинга
(define (on-error source-name)
  (lambda (tok-ok? tok-name tok-value start-pos end-pos)
    ; генерирует исключение
    (raise-read-error
             "Parser error" ; текст исключения
             source-name                  ; имя файла
             (position-line start-pos)    ; номер строки с ошибкой
             (position-col start-pos)     ; номер колонки с ошибкой
             (position-offset start-pos)  ; смещение от начала файла
             (- (position-offset end-pos) ; длина фрагмента с ошибкой
                (position-offset start-pos)))))

; calc-parser — это функция, возвращающая функцию, полученную
; в результате вызова функции parser из модуля parser-tools/yacc
(define (calc-parser source-name)
  (parser
   (src-po  s)    ; это нужно для того, чтобы в функцию обработки ошибки
                  ; передавалась позиция проблемного куска
   (start start) ; стартовый нетерминал у нас называется start
   (end EOF)     ; конец файла у нас называется EOF
   (tokens value-tokens op-tokens) ; две группы токенов, определённые в calc-lexer
   (error (on-error source-name)) ; функция обработки ошибок (см. выше)

  ; Грамматика

   (grammar
    ; программа = команды
    (start
     ((statements) $1))

    (statements
     ; команды = ничто
     (() '())
     ; или команда + команды
     ; $1 и $2 обозначают соответственно первый и второй элементы
```

```
     ; сопоставления, то есть statement и statements
     ((statement statements) (list* $1 $2)))

    (statement
     ((assignment) $1)
     ((printing) $1))

    (constant
     ((NUMBER) $1))

    (expression
     ; выражение = терм
     ((term) $1)
     ; или выражение плюс/минус токен.
     ; PLUS и MINUS здесь — терминалы, взятые из op-tokens из calc/lexer.
     ((expression PLUS term) (list 'plus $1 $3))
     ((expression MINUS term) (list 'minus $1 $3)))

    (term
     ((factor) $1)
     ((term MULTIPLY factor) (list 'multiply $1 $3))
     ((term DIVIDE factor) (list 'divide $1 $3)))

    (factor
     ((primary-expression) $1)
     ((MINUS primary-expression) (list 'negate $2))
     ((PLUS primary-expression) $2))

    (primary-expression
     ((constant) $1)
     ((IDENTIFIER) (list 'value-of $1))
     ((LEFT-PAREN expression RIGHT-PAREN) $2))

   (assignment
     ((IDENTIFIER ASSIGN expression) (list 'assign $1 $3)))

    (printing
     ((PRINT expression) (list 'print $2))))))
```

Теперь свяжем лексический и синтаксический анализаторы в одно целое с помощью функции, которая принимает некоторый входной поток (в Racket это называется «порт») и соответствующее ему имя файла и возвращает результат грамматического разбора. Правим дальше `calc/parser.rkt`:

```
(define (parse-calc-port port file)
  ; посчитать номера строк заранее
  (port-count-lines! port)
  ; создать и сразу вызвать парсер
  ((calc-parser file)
   ; и передать в него функцию, которая должна выдавать токены один за другим
   (lambda ()
     ; calc-lexer как раз это и делает
     (calc-lexer port))))
```

Поскольку наш анализатор должен будет работать в среде Racket на тех же правах, на-которых работает «родной» анализатор, мы должны реализовать функции `calc-read` и `calc-read-syntax`, которые придут на замену стандартным `read` и `read-syntax` [2]. Окончание `calc/parser.rkt`:

```
(define (calc-read in)
  (syntax->datum
   (calc-read-syntax #f in)))

(define (calc-read-syntax source-name input-port)
  (parse-calc-port input-port source-name))
```

Пояснение: в то время как `calc-read` возвращает прочитанные данные (`datum`) вида `((assign X (+ 1 2))`, `calc-read-syntax` возвращает синтаксические объекты [3], которые суть те же самые данные, но с привязкой к участкам исходного файла и прочей информацией. В зависимости от режима работы, Racket пользуется либо `read`, либо `read-syntax`, и, во-избежание неожиданностей, лучше сделать их одинаковыми. Что и сделано: `calc-read` просто берёт синтаксические объекты из `calc-read-syntax` и превращает их в данные с помощью `syntax->datum`.

Настало время встроить наш анализатор в среду Racket. Это очень просто: нужно создать файл `calc/lang/reader.rkt` следующего содержания:

```
(module reader syntax/module-reader
  #:language 'racket              ; об этом позже
  #:read calc-read                ; переопределение стандартного read
  #:read-syntax calc-read-syntax ; переопределение стандартного read-syntax
  #:whole-body-readers? #t ; этот флаг устанавливается в том случае, когда наши
                           ; read и read-syntax всегда читают входной поток до
                           ; конца (calc-read и calc-read-syntax так и делают)
  (require calc/parser))  ; модуль, в котором лежат calc-read и calc-read-syntax
```

Racket, увидев `#lang calc` в начале файла, будет искать `calc/lang/reader.rkt` в системных директориях, включая `work`, где мы всё и делаем. Поэтому мы уже можем начинать писать код на `calc` в среде Racket. Создадим файл `test.calc`:

```
#lang calc
X = 3
Y = X + 1
Print X * X + Y * Y

pi = 3.1415926535
r = 12.2
l = 40
Print pi * r * (2 + l)
```

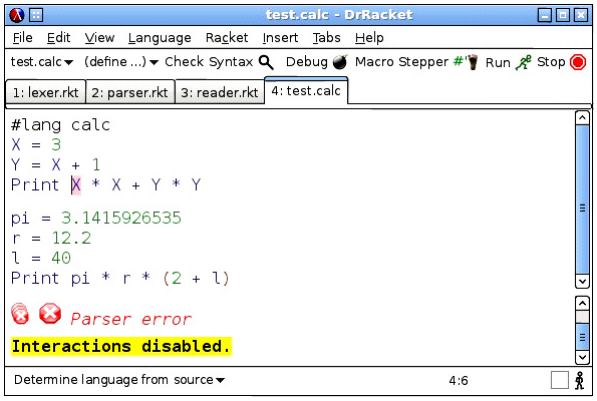

Рис. 1

Запускать файл на выполнение ещё рано, так как реализации языка пока ещё нет. Но уже можно посмотреть на результат синтаксического анализа кода, для чего в Racket есть удобный инструмент под названием «Macro stepper». Нажимаем... и видим сообщение «Parser error» из нашего `calc/parser.rkt` (рис. 1). Ошибка состоит в том, что слово Print написано с большой буквы, тогда как лексический анализатор допускает только «print» без вариантов. Сделаем его нечувствительным к регистру (`calc/lexer.rkt`):

```
; вспомогательная функция, преобразующая строку в выражение, нечувствительное
; к регистру, например ''foo'' -> (:: (:or #\f #\F) (:or #\o #\O) (:or #\o #\O))
(define-for-syntax (string->ci-pattern s)
  (cons ':: (map (lambda (c)
                   (list ':or (char-downcase c) (char-upcase c)))
                 (string->list s))))
```

```
; специализированный макрос для применения в лексическом анализаторе
(define-lex-trans lex-ci
  (lambda (stx)
    (syntax-case stx ()
      ((_ id) ; здесь id — это строка, которую мы преобразуем, т.е. "print"
       (with-syntax ((result (string->ci-pattern
                              (syntax->datum #'id))))
         #'result)))))

(define calc-lexer
  (lexer-src-pos
   ((:+ lex:whitespace) (return-without-pos (calc-lexer input-port)))
   ("=" (token-ASSIGN))
   ("+" (token-PLUS))
   ("-" (token-MINUS))
   ("*" (token-MULTIPLY))
   ("/" (token-DIVIDE))
   ("(" (token-LEFT-PAREN))
   (")" (token-RIGHT-PAREN))
   ((lex-ci "print") (token-PRINT))
   ((:: lex:letter (:* (:or lex:letter lex:digit)))
    (token-IDENTIFIER (string->symbol lexeme)))
   ((:: (:? #\-) (:+ lex:digit) (:? (:: #\. (:* lex:digit))))
    (token-NUMBER (string->number lexeme)))
   ((eof) 'EOF)))
```

Запускаем Macro stepper вторично на `test.calc` и видим (рис. 2).

Парсер успешно отработал и выдал в качестве результата набор конструкций вида

```
(assign X 3)
(assign Y
      (plus (value-of X) 1))
```

Это не является валидным кодом на Racket, что подтверждается сообщением «unbound identifier in module». Но по замыслу – полученный код и не олжен являться кодом на Racket! Для исполнения этого кода мы должны создать набор макросов, преобразующих его в код на Racket, а точнее – в синтаксические объекты. Декларация `#:language` в `calc/lang/reader.rkt` – именно об этом. Мы написали `#:language 'racket` исключительно потому, что совсем ничего не писать система не позволяет. На самом деле, в этом месте нужно указать путь к модулю с макросами:

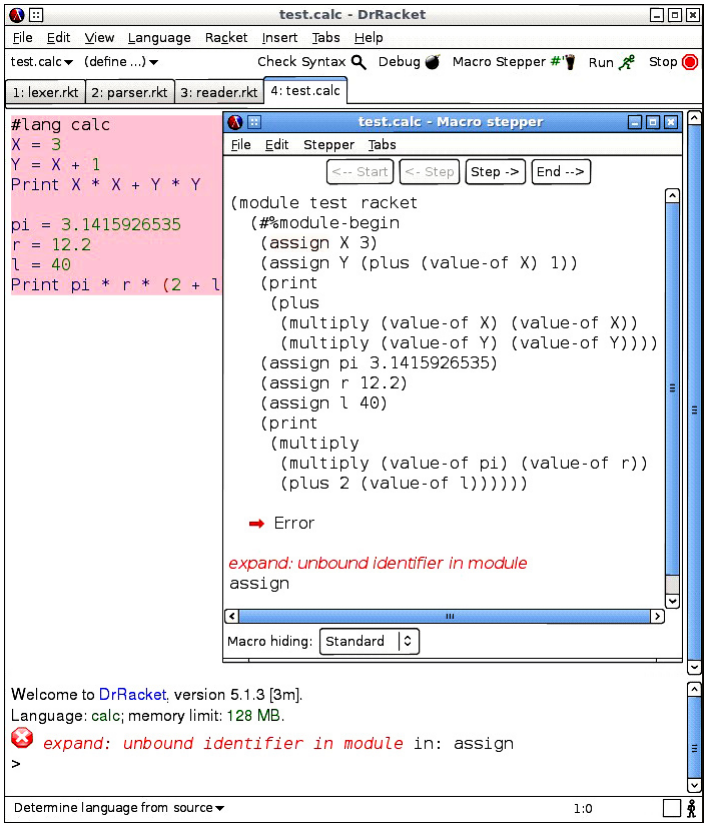

Рис. 2

```
(module reader syntax/module-reader
  #:language 'calc/language
  #:read calc-read
  #:read-syntax calc-read-syntax
  #:whole-body-readers? #t
  #:language-info '#(calc/lang/lang-info get-info #f)
  (require calc/parser))
```

Этот модуль мы сейчас и создадим (`calc/language.rkt`):

```
#lang racket

(provide
 ; Системные вещи, на которых мы не будем заострять внимание.
 ; Достаточно знать, что #%module-begin и #%datum из racket
 ; годятся и для calc, поэтому мы переэкспортируем их без изменений
 #%module-begin #%datum
 ; Макросы для выполнения операций языка calc
 assign plus minus divide multiply negate value-of print)

; Окружение = хеш-таблица со значениями переменных
(define current-env (make-hash))

; Присваивание переменной = запись в хеш-таблицу
(define-syntax-rule (assign name value)
  ; Обратите внимание на апостроф перед name: он делает из имени
  ; переменной, переданной в макрос, символ. Таким образом, из (assign X 3)
  ; получается (hash-set! current-env 'X 3),
  ; в то время как (hash-set! current-env X 3)
  ; выдало бы ошибку (hash-set! current-env 'name value))

; Арифметические операции

(define-syntax-rule (plus a b)
  (+ a b))

(define-syntax-rule (minus a b)
  (- a b))

(define-syntax-rule (divide a b)
  (/ a b))

(define-syntax-rule (multiply a b)
  (* a b))

(define-syntax-rule (negate a)
  (- a))

; Получение значения переменной по имени
(define-syntax-rule (value-of name)
  ; То же самое с апострофом,
  ; см. макрос assign
  (hash-ref current-env 'name))

; Печать
(define-syntax-rule (print value)
  (printf "~v\n" value))
```

Теперь можно наконец выполнить (Run) наш `test.calc` (рис. 3).

Запустив Macro stepper снова и выключив в нём опцию «macro hiding», мы можем наблюдать, во что в конечном итоге раскрылся наш код (рис. 4).

На этом этапе можно считать, что у нас есть парсер и компилятор. Всё это занимает 110 строк, не считая пустых строк и комментариев.

Этап 2: необходимые «примочки»

Если мы сейчас попробуем запустить отладчик (Debug) и поставить точки останова в код программы, нас ждёт неудача. Объясняется это просто: парсер в `calc/parser.rkt` возвращает не синтаксические объекты, а простые списки, то есть `datum`. Они преобразуются в синтаксические объекты позже, автоматически, но дела это в принципе не меняет, так как информации о местоположении синтаксических единиц в исходном коде не появляется. Эта информация известна только парсеру. Он её «забывает» везде, кроме функции on-error. И поставить точки останова невозможно, так как система просто не знает, какой синтаксический объект к какому месту исходного кода относится. Сейчас мы это исправим (`calc/parser.rkt`):

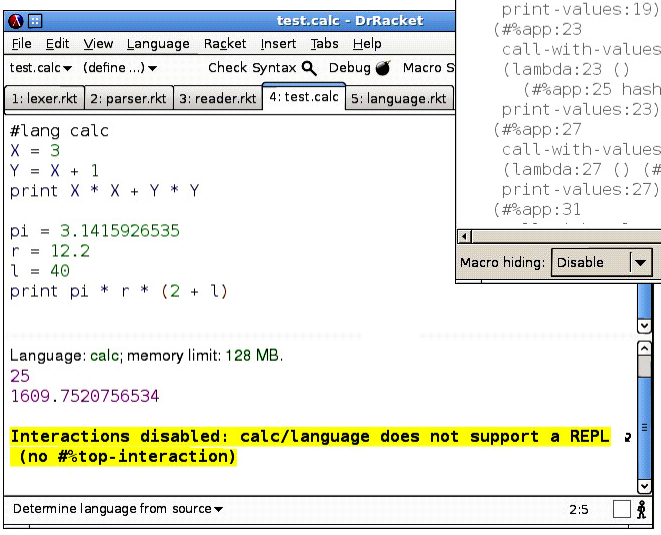

Рис. 3

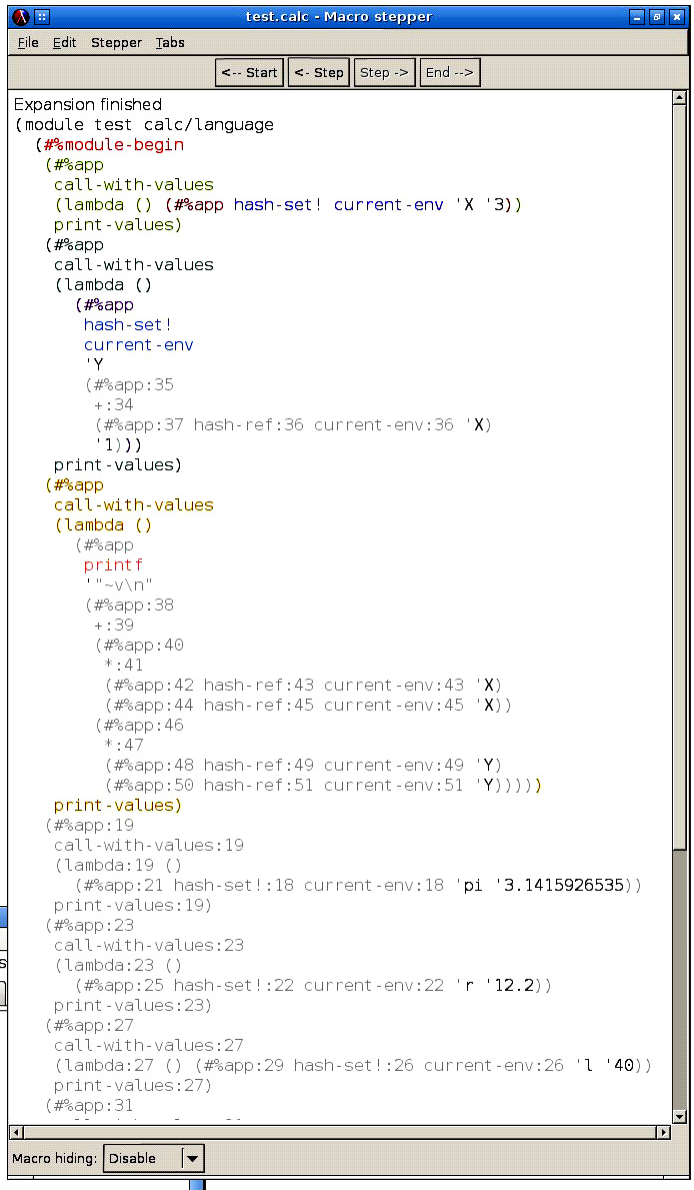

Рис. 4

```
(define-syntax (build-so stx)
  (syntax-case stx ()
    ((_ value start end)
     ; вытаскиваем из контекста (stx) $i-start-pos и $j-end-pos, где i и j —
     ; числа, переданные в макрос как start и end; а также source-name
     (with-syntax ((start-pos (datum->syntax
                               stx
                               (string->symbol
                                (format "$~a-start-pos"
                                        (syntax->datum #'start)))))
                   (end-pos (datum->syntax
                             stx
                             (string->symbol
                              (format "$~a-end-pos"
                                      (syntax->datum #'end)))))
                   (source (datum->syntax
                            stx
                            'source-name)))
       (syntax
        (datum->syntax
         #f
         value
         ; конструируем уже знакомую по on-error пятёрку значений
         (list source
               (position-line start-pos)
               (position-col start-pos)
               (position-offset start-pos)
               (- (position-offset end-pos)
                  (position-offset start-pos)))))))))

(define (calc-parser source-name)
  (parser
   (src-pos)
   (start start)
   (end EOF)
   (tokens value-tokens op-tokens)
   (error (on-error source-name))

  (grammar
    (start
     ; Числа 1 и 1, переданные в макрос build-so, раскрываются соответственно в
     ; $1-start-pos и $1-end-pos, каковые переменные доступны благодаря указанию
     ; (src-pos) выше и содержат начальную и конечную позиции первого нетерминала
     ; в списке (в данном случае он всего один и есть).
     ((statements) (build-so $1 1 1)))

    (statements
     (() '())
     ((statement statements) (list* $1 $2)))

    (statement
     ((assignment) $1)
     ((printing) $1))

    (constant
     ((NUMBER) $1))
```

```
(expression
 ((term) $1)
 ; Числа 1 и 3 раскрываются макросом build-so в $1-start-pos и $3-end-pos
 ; соответственно, т.е. от начала первого нетерминала до конца третьего.
 ((expression PLUS term) (build-so (list 'plus  $1 $3) 1 3))
 ((expression MINUS term) (build-so (list 'minus $1 $3) 1 3)))

(term
 ((factor) $1)
 ((term MULTIPLY factor) (build-so (list 'multiply $1 $3) 1 3))
 ((term DIVIDE factor) (build-so (list 'divide $1 $3) 1 3)))

(factor
 ((primary-expression) $1)
 ((MINUS primary-expression) (build-so (list 'negate $2) 1 2))
 ((PLUS primary-expression) $2))

(primary-expression
 ((constant) $1)
 ((IDENTIFIER) (build-so (list 'value-of $1) 1 1))
 ((LEFT-PAREN expression RIGHT-PAREN) (build-so $2 1 3)))

(assignment
 ((IDENTIFIER ASSIGN expression) (build-so (list 'assign $1 $3) 1 3)))

(printing
 ((PRINT expression) (build-so (list 'print $2) 1 2))))))
```

И – о чудо!– теперь работает отладчик, можно ставить точки останова, ходить по шагам, даже стек вызовов имеется (рис. 5).

Racket позволяет отладчику заходить в реализацию вашего языка в лисповых функциях, если в момент отладки модули с этими функциями открыты. Но нашего `calc/language.rkt` это не касается, поскольку ни одной функции у нас нет – мы обошлись макросами. То есть ближайший уровень, в который может попасть отладчик после кода на calc – это реализация `hash-set`, + и прочих примитивных вызовов. На такой уровень нам нет необходимости опускаться.

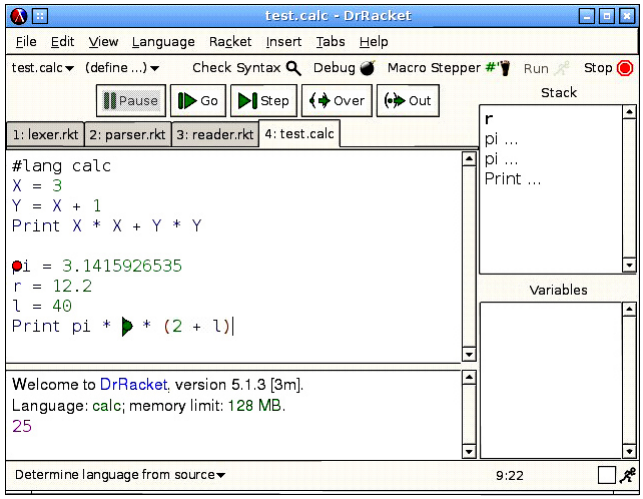

Рис. 5

Обратим теперь внимание на то, что для языка calc не работает REPL! Об этом честно сообщается после выполнения программы (Run) чёрным текстом на жёлтом фоне. REPL не работает, потому что в реализации (`calc/language.rkt`) не определён макрос `#%top-interaction`, который, собственно, и должен раскрывать полученные в REPL синтаксические объекты. Стандартный макрос из Racket не годится, так что мы напишем свой, очень простой:

```
#lang racket

(provide
 #%module-begin #%datum
 ; Переопределять #%top-interaction внутри модуля нельзя, поэтому мы
 ; сделаем макрос top-interaction и переименуем его при экспорте.
 (rename-out (top-interaction #%top-interaction))
 assign plus minus divide multiply negate value-of print)

; Макрос действительно такой — с троеточиями. Это часть синтаксиса.
(define-syntax-rule (top-interaction body ...)
    (begin body ...))
```

Но этого недостаточно. Проблема в том, что декларации `#:read` и `#:read-syntax` в `calc/reader.rkt` по какой-то причине не распространяются на REPL. REPL в DrRacket принадлежит самой DrRacket и подлежит настройке отдельно, по строго определённым правилам. Нужно добавить декларацию `#:language-info` в `calc/reader.rkt`:

```
(module reader syntax/module-reader
  #:language 'calc/language
  #:read calc-read
  #:read-syntax calc-read-syntax
  #:whole-body-readers? #t
  #:language-info '#(calc/lang/lang-info get-info #f)
  (require calc/parser))
```

Затем нужно создать модуль `calc/lang/lang-info.rkt` такого содержания:

```
#lang racket/base

(provide get-info)

(define (get-info data)
  (lambda (key default)
    (case key
      ((configure-runtime)
       '(#(calc/lang/configure-runtime configure #f)))
      (else
       default))))
```

Наконец, нужно создать модуль `calc/lang/configure-runtime.rkt`, содержащий функцию конфигурации рантайма:

```
#lang racket/base

(require calc/parser)
(provide configure)

(define (configure data)
  ; Конфигурация заключается в установке параметра current-read-interaction
  ; (Параметры в Racket — что-то вроде динамических переменных в Common Lisp.)
  ; Мы меняем этот параметр на нашу функцию, которая вызывает наш парсер.
  (current-read-interaction even-read))

(define (even-read source-name input-port)
  (begin0
    (parse-calc-port input-port source-name)
    ; Почему-то нужно делать так, чтобы последующий вызов
    ; current-read-interaction вернул EOF. С этой целью производится
```

```
    ; нижеследующий трюк с заменой параметра на odd-read.
    (current-read-interaction odd-read)))

; Вторая часть трюка заключается в замене параметра обратно на even-read.
; Среди разработчиков нет согласия по поводу того, правильно ли так делать;
; оставим этот вопрос на их совести, в любом случае все имеющиеся примеры
; работают именно так.
(define (odd-read src ip)
  (current-read-interaction even-read)
  eof)
```

И вот у нас работает REPL (рис. 6).

Благодаря умной работе механизма модулей, два и более открытых одновременно окна с программами на `calc` не конфликтуют и не засоряют окружение друг друга (рис. 7).

Самое время заметить, что ошибки с необъявленными переменными возникают на этапе выполнения программы, тогда как во всех нормальных средах они должны отлавливаться на этапе компиляции. Полноценный компилятор мы писать не будем, ограничимся проверкой того, что все используемые переменные встретились перед использованием в левой части. Файл `calc/compiler.rkt`:

```
#lang racket

(provide compile-program
         compile-statement)

; Хеш-таблица имён переменных
; (не путать с current-env в language.rkt,
; это разные этапы)
(define variables (make-hash))

(define (compile-program p)
  ; проверить все подвыражения
  ;(syntax-e раскрывает синтаксический
  ; объект-список в список
  ; синтаксических объектов)
```

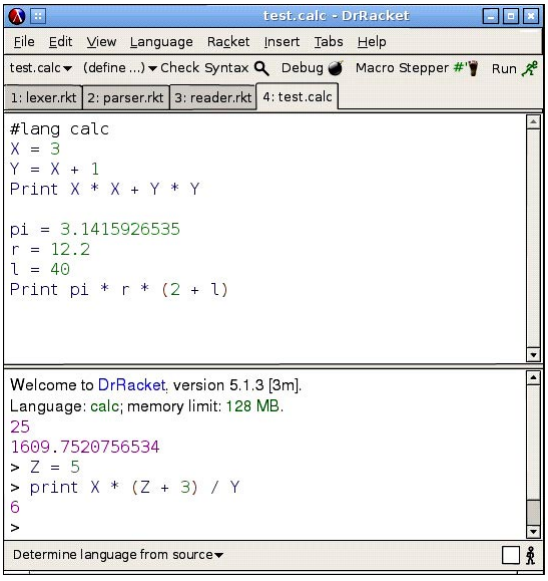

Рис. 6

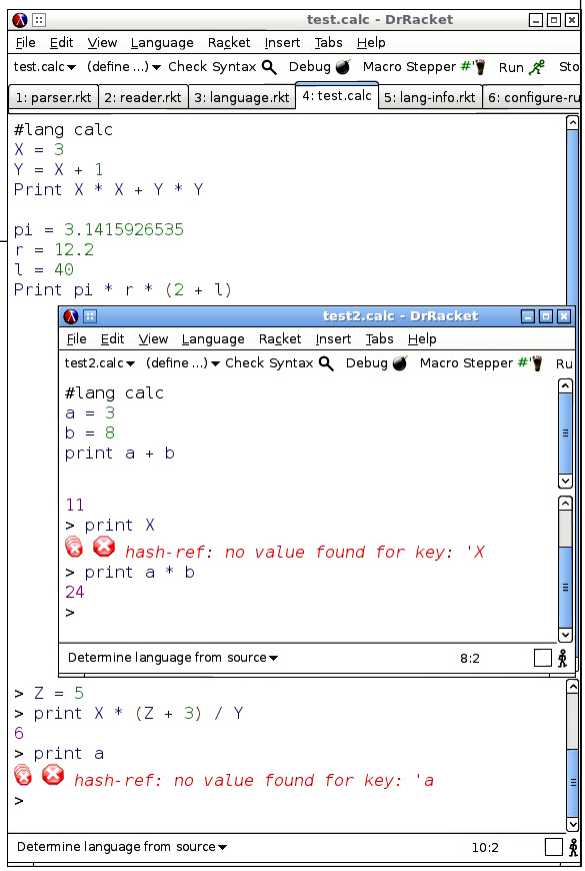

Рис. 7

```
  (for-each check-syntax (syntax-e p))
  ; вернуть исходную синтаксическую
  ; конструкцию как результат
 ; (приличный компилятор вернул бы новую,
  ; оптимизированную конструкцию)
  p)

(define (compile-statement s)
  (check-syntax s)
  s)

(define (check-syntax s)
  ; Что за объект?
  ; Преобразуем в данные и посмотрим.
  (match (syntax->datum s)
    ; Команда присваивания
    ((list 'assign a b)
     ; Сначала проверим правую часть,
     ; чтобы поймать ошибки вида X = X
    (check-syntax (third (syntax-e s)))
     ; Теперь занесём левую часть в словарь
     (hash-set! variables a a))
    ; Доступ к переменной
    ((list 'value-of a)
     ; Есть ли она в словаре?
     (unless (hash-has-key? variables a)
       ; Если нет — ошибка со ссылкой на соответствующий синтаксический объект
       (raise-syntax-error
        #f "access to unassigned variable" (second (syntax-e s)))))
    ; Все прочие команды
    ((list x ...)
     ; проверяются рекурсивно
     (for-each check-syntax (syntax-e s)))
    ; Константы не подлежат проверке
    (_ #f)))
```

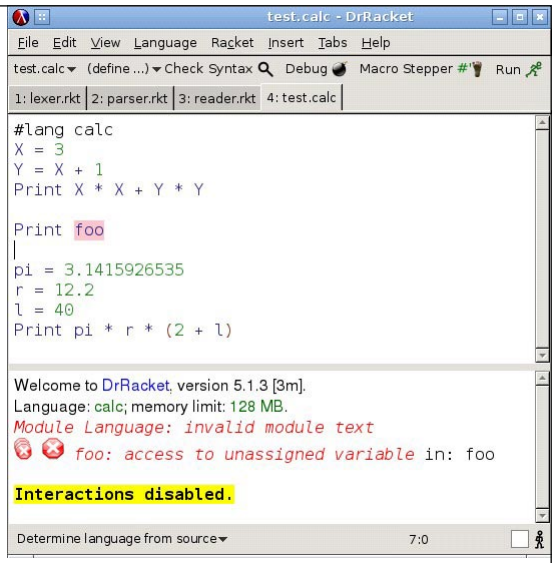

Рис. 8

Попросим парсер вызывать наш «компилятор» перед возвращением результата (`calc/parser.lisp`):

```
(require parser-tools/yacc
         syntax/readerr
         calc/lexer
         calc/compiler)

(define (calc-read-syntax source-name input-port)
  (compile-program
   (parse-calc-port input-port source-name)))
```

Насладимся результатом (рис. 8).

Попросим REPL делать то же самое (`calc/lang/configure-runtime.rkt`):

```
(require calc/parser
         calc/compiler)

(define (even-read source-name input-port)
  (begin0
    (compile-statement (parse-calc-port input-port source-name))
    (current-read-interaction odd-read)))
```

Насладимся результатом и здесь (рис. 9).

Отметим, что ошибки времени выполнения всё же случаются и снабжены бектрейсом (рис. 10).

На данный момент у нас есть всё необходимое, а написали мы всего 208 строк кода.

Этап 3: элементы «роскоши»

Заметим небольшое неудобство: REPL не позволяет переводить строку. Нажатие Enter ведёт к немедленному отправлению строки на синтаксический анализ. Мы собирались дописать команду на следующей строке, но REPL безжалостен (рис. 11).

Хорошая новость: поведение REPL можно настроить. Открываем снова `calc/lang/lang-info.rkt`:

```
(define (get-info data)
  (lambda (key default)
    (case key
      ((configure-runtime)
       '(#(calc/lang/configure-runtime configure #f)))
      ((drracket:submit-predicate)
       (dynamic-require 'calc/tool/submit 'repl-submit?))
      (else
       default))))
```

и создаём модуль `calc/tool/submit.rkt` с функцией `repl-submit?`, которая возвращает `#t`, если перевод строки завершающий, и `#f`, если промежуточный. Определить это не так просто, учитывая, что пользователь может вводить в REPL всё, что ему заблагорассудится. Будем считать, что строка готова к синтаксическому анализу, если она не пустая, не завершается оператором (`+, -, /, *, =, print`) и если скобки в ней сбалансированы. Весьма уместно использовать уже готовый лексический анализатор для выполнения этой задачи:

```
#lang racket

(require calc/lexer
         parser-tools/lex)
```

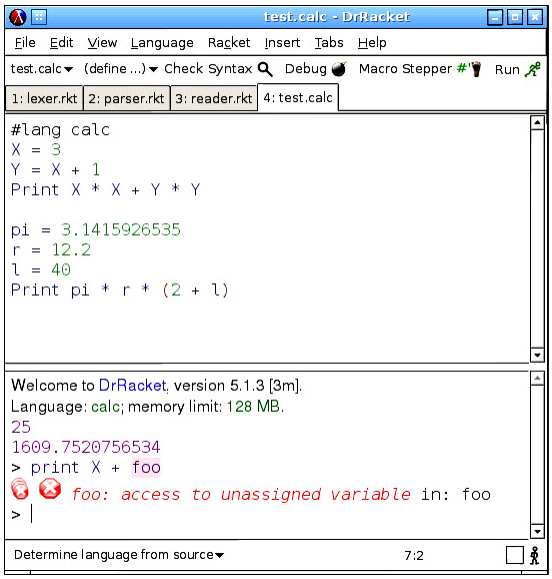

Рис. 9

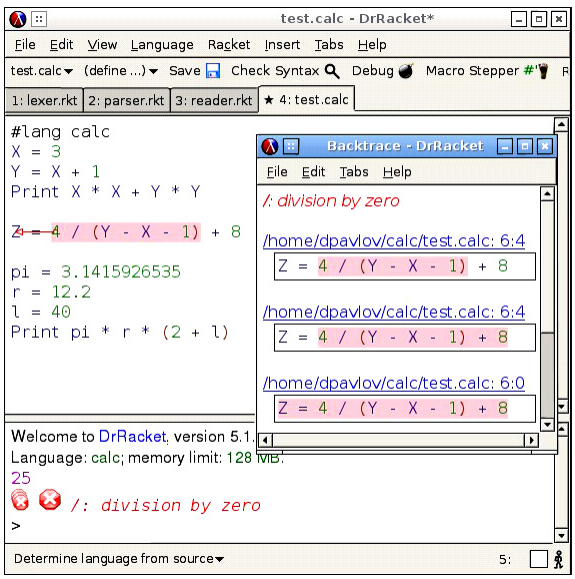

Рис. 10

```
(provide repl-submit?)

(define (repl-submit? ip has-white-space?)
  (let loop ((blank? #t)       ; строка пустая?
             (pending-op? #f) ; строка завершается оператором?
             (paren-count 0)) ; баланс скобок
    ; Все ошибки лексического анализатора мы обязаны пропускать
    (with-handlers ((exn:fail:read?
                     (lambda (e)
                       #t)))
      (let ((token (position-token-token (calc-lexer ip))))
        (case token
          ((EOF)
           (and (zero? paren-count)
                (not blank?)
                (not pending-op?)))
          ((PLUS MINUS MULTIPLY DIVIDE ASSIGN PRINT)
           (loop #f #t paren-count))
          ((LEFT-PAREN)
           (loop #f #f (+ paren-count 1)))
          ((RIGHT-PAREN)
           (loop #f #f (- paren-count 1)))
          (else
           (loop #f #f paren-count)))))))
```

Вроде работает (рис. 12).

Почти всё. Осталось только добавить подсветку синтаксиса. Ну и, например, комментарии, чтобы было что подсвечивать. Пусть комментарии будут в стиле bash – от символа «#» до конца строки. Обновляем `calc/lexer.rkt`, попутно экспортируя из него некоторые вещи, которые скоро понадобятся:

```
(provide value-tokens op-tokens
         position-line position-col position-offset
         calc-lexer
         lex:comment lex:identifier lex:number lex-ci)
```

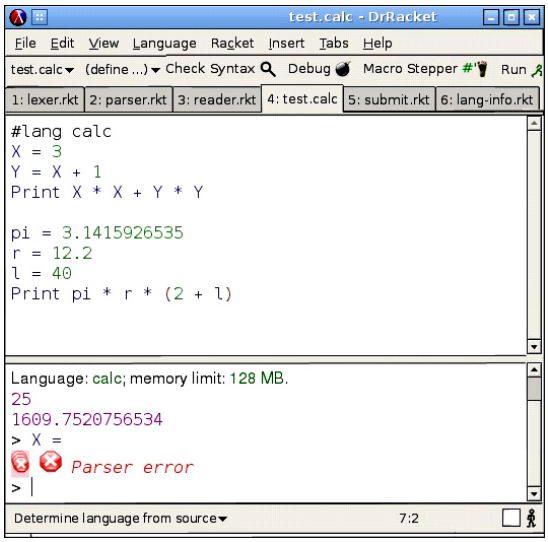

Рис. 11

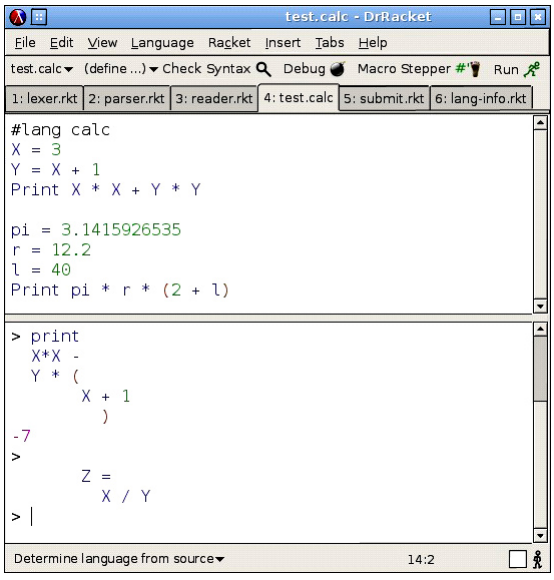

Рис. 12

```
(define-lex-abbrevs
  (lex:letter (:or (:/ #\a #\z) (:/ #\A #\Z)))
  (lex:digit (:/ #\0 #\9))
  (lex:comment (:: "#" (:* (:: (char-complement #\newline))) (:? #\newline)))
  (lex:whitespace (:or #\newline #\return #\tab #\space #\vtab))
  (lex:identifier (:: lex:letter (:* (:or lex:letter lex:digit))))
  (lex:number (:: (:? #\-) (:+ lex:digit) (:? (:: #\. (:* lex:digit))))))

(define calc-lexer
  (lexer-src-pos
   ((:+ lex:whitespace) (return-without-pos (calc-lexer input-port)))
   ; комментарии пропускаем так же, как и пробелы
   ((:+ lex:comment) (return-without-pos (calc-lexer input-port)))
   ("=" (token-ASSIGN))
   ("+" (token-PLUS))
   ("-" (token-MINUS))
   ("*" (token-MULTIPLY))
   ("/" (token-DIVIDE))
   ("(" (token-LEFT-PAREN))
   (")" (token-RIGHT-PAREN))
   ((lex-ci "print") (token-PRINT))
   (lex:identifier (token-IDENTIFIER (string->symbol lexeme)))
   (lex:number (token-NUMBER (string->number lexeme)))
   ((eof) 'EOF)))
```

Информация о подсветке синтаксиса задаётся определённым образом в директиве `#:info` файла `calc/lang/reader.rkt`:

```
(module reader syntax/module-reader
  #:language 'calc/language
  #:read calc-read
  #:read-syntax calc-read-syntax
  #:whole-body-readers? #t
  #:language-info '#(calc/lang/lang-info get-info #f)
  #:info (lambda (key defval default)
           (case key
             ((color-lexer)
              (dynamic-require 'calc/tool/syntax-color 'get-syntax-token))
             (else (default key defval))))
  (require calc/parser))
```

Собственно подсветку осуществляет отдельный модуль `calc/tool/syntax-color.rkt`. В нём находится, по сути, ещё один лексический анализатор, но такой, который выдаёт не-токены, а особые «пятёрки» значений: лексема, её тип (комментарий/константа/ключевое слово/символ/etc), «скобочность», а также начало и конец лексемы в исходном файле:

```
#lang racket

(require parser-tools/lex
         (prefix-in : parser-tools/lex-sre)
         calc/lexer)

(provide get-syntax-token)

(define (syn-val lexeme type paren start end)
  (values lexeme type paren (position-offset start) (position-offset end)))
```

```
(define get-syntax-token
  (lexer
   ((:+ whitespace)
    (syn-val lexeme 'whitespace #f start-pos end-pos))
   (lex:comment
    (syn-val lexeme 'comment #f start-pos end-pos))
   (lex:number
    (syn-val lexeme 'constant #f start-pos end-pos))
   ; "Print" у нас будет единственным ключевым словом
   ((lex-ci "print")
    (syn-val lexeme 'keyword #f start-pos end-pos))
   ; Имена переменных у нас будут идентфикаторами
   (lex:identifier
    (syn-val lexeme 'symbol #f start-pos end-pos))
   ; Арифметические операции и "=" будут считаться за скобки
   ; (Операций кроме скобок Racket, похоже, не знает)
   ((:or #\+ #\- #\/ #\* #\=)
    (syn-val lexeme 'parenthesis #f start-pos end-pos))
   ; Сами скобки тоже считаются за скобки, и вдобавок к тому обладают
   ;  свойством "скобочности", чтобы редактор Racket подсвечивал
   ;  открывающие и закрывающие, опять же, скобки.
   (#\( (syn-val lexeme 'parenthesis '|(| start-pos end-pos))
   (#\) (syn-val lexeme 'parenthesis '|)| start-pos end-pos))
   ((eof) (syn-val lexeme 'eof #f start-pos end-pos))
   (any-char (syn-val lexeme 'error #f start-pos end-pos))))
```

Внимание: чтобы подсветка синтаксиса заработала, надо перезапустить Racket. Если вдуматься, это значит, что при выполнении всех предыдущих операций Racket не надо было перезапускать! И даже `test.calc` можно было не закрывать. При том что мы там «резали по живому» во многих местах. Racket истинно динамическая платформа. Вот что получится после перезапуска (рис. 13).

Подсветка синтаксиса в действии. В REPL, правда, она не работает (то есть работает, но не наша, а стандартная), но не будем придираться к мелочам. Racket – прекрасная вещь. Всё вышеописанное мы сделали, написав 268 строк кода. То, что получилось, можно скачать по ссылке [4]. Вообще-то можно ещё сделать так, чтобы язык `calc` включался без `#lang calc`, и ещё можно научить DrRacket смотреть переменные в отладчике, и ещё можно собрать DrRacket, кастомизированный под calc и готовый к распространению, но обо всём этом как-нибудь в другой раз.

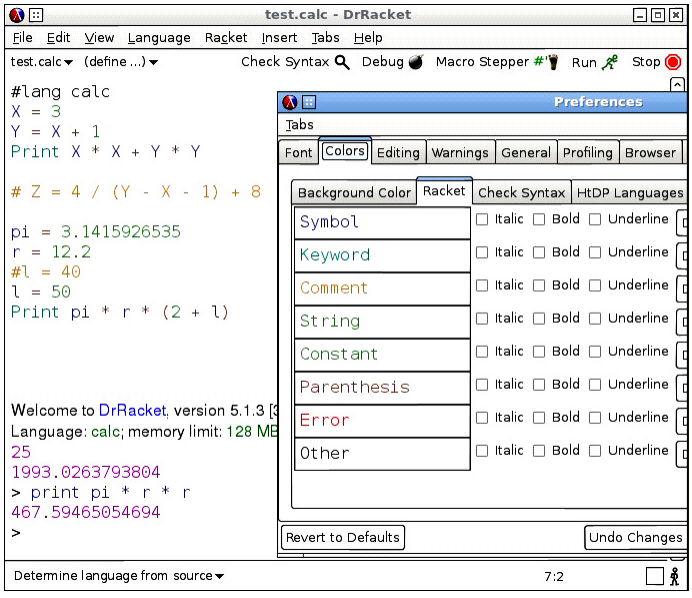

Рис. 13

Литература

1. *Павлов Д.А.* Создание предметно-ориентированных языков // Компьютерные инструменты в образовании, 2011. № 6. С. 57–60.
2. http://docs.racket-lang.org/reference/Reading.html (дата обращения: 30.10.2012).
3. http://docs.racket-lang.org/reference/syntax-model.html (дата обращения: 30.10.2012).
4. https://github.com/kugelblitz/calc (дата обращения: 30.10.2012).

Abstract

The article gives a detailed explanation of how to create a simple domain-specific language (DSL) on the Racket development platform. There is a lexical analyzer, parser, REPL, compiler, debugger, backtraces, syntax highlighting. The new language is integrated into the DrRacket IDE. The whole project is less than 300 lines of code.

Keywords: domain-specific language, parsing, compilation, Racket.

Павлов Дмитрий Алексеевич,
научный сотрудник Института прикладной астрономии РАН,
dpavlov@ipa.nw.ru