Computer assisted mathematics, 2019
№ 1: 20–31
http://cte.eltech.ru

USING CAPABILITIES OF RACKET TO IMPLEMENT A DOMAIN-SPECIFIC LANGUAGE

Dolgakov I. A., research engineer, ✉ ia.dolgakov@iaaras.ru
Pavlov D. A., PhD, senior researcher, dpavlov@iaaras.ru

[1]Institute of Applied Astronomy RAS, 10 Kutuzova Embankment, 191187, Saint Petersburg, Russia

Abstract

An implementation of a domain-specific language, Landau, based on the Racket platform, is presented. Landau is a dynamical system specification language, used in an environment where parameters of the dynamical system must be determined from processing of observational data. That, in turn, requires efficient and accurate calculation of derivatives, which can be achieved with the automatic differentiation (AD) technique. Landau is a Turing incomplete statically typed language aimed to support code generation with AD. The Turing incompleteness provides the ability of analyzing the whole program workflow to figure out the chain of derivatives' calculation. Landau has compile-time ranged for loops, if/else branching constructions, mutable variables and arrays. Landau can be compiled to both Racket and ANSI C. Landau implementation takes advantage of features that are unique to the Racket platform and make creation of DSLs more convenient than on other platforms.

Keywords: *Automatic differentiation, dynamical systems, compilers, domain-specific languages, Racket.*

Citation: I. A. Dolgakov and D. A. Pavlov, "Using Capabilities of Racket to Implement a Domain-Specific Language," *Computer assisted mathematics*, no. 1, pp. 20–31, 2019.

1. INTRODUCTION

In dynamical system modeling, various systems from different application domains can be represented by an autonomous system of first-order ODEs:

$$\dot{\vec{x}}(t) = f(\vec{x}(t), \vec{p}). \tag{1}$$

where $\vec{p} = \{p_i\}_{i=1}^{m}$ is a vector of m fixed parameters. One instance of the model is based on the values of the parameters, and also on the initial conditions:

$$\vec{x}(t_0) = \vec{x}_0 \tag{2}$$

Although the precise values of the initial conditions and parameters are unknown, they can be determined from observations using regression methods. The cost function includes the solution of ODE (1) and is parametrised with respect to $\vec{P} = (x_0^{(0)}, \ldots, x_0^{(n)}, p_1, \ldots, p_m)$. To perform optimization one needs to find $\frac{\mathrm{d}\vec{x}}{\mathrm{d}\vec{P}}$.

One way to obtain the derivative is to include it into our system of ODEs, together with $\vec{x}$ itself. Accordingly, the initial conditions $\frac{\mathrm{d}\vec{x}_0}{\mathrm{d}\vec{P}}$ and the time derivative $\frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}\vec{x}}{\mathrm{d}\vec{P}}$ are needed to solve

the IVP for the new system. While the initial conditions are trivial, the time derivative must be obtained by substituting (1):

$$\frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}\vec{x}}{\mathrm{d}\vec{P}} = \frac{\mathrm{d}f(\vec{x},\vec{p})}{\mathrm{d}\vec{P}}. \tag{3}$$

Thus, in order to estimate the free variables, one needs to compute the derivative of the ODE's right-hand side w.r.t. $\vec{P}$.

The most convenient way to do that is to use the Automatic Differentiation (AD) technique of obtaining numerical values of derivatives. As opposed to the symbolic differentiation, AD not only reduces the computation time by using memoization techniques, but also provides more flexibility as it can deal with complicated structures from programming languages, such as conditions and loops.

Each variable of the original program is associated with its derivative counterpart(s), which is(are) computed along with the original variable value.

2. KEY DECISIONS

AD can be implemented in one of two ways: operator overloading and source code transformation. The first approach is based on describing the "dual number" data structure and overloading arithmetic operators and functions to operate on them. The second approach involves the analysis of function source and generation of the differentiation code. It was found [1] that the latter approach generally produces more efficient derivative code.

While there exist, and have existed for decades, tools to generate AD code from program code on general-purpose languages [2–4], it has been decided in [5] to create a domain-specific, Turing-incomplete language with AD and code generation in mind.

Racket [6] (http://racket-lang.org) is a general-purpose, multi-paradigm programming language and platform, used in education, science, and commerce. Historically, Racket is a descendant of implementation of Scheme made by PLT Inc. and used in university courses related to programming, math, and computer science.

Racket provides advanced tools for implementation of DSLs [7], not all of which are available on other platforms [8]. In particular, Racket has hygienic macros and generally very advanced macro transformation tools. A compiler written in Racket does not necessarily need to compile text to text, or text to machine code–Racket allows to parse text into *syntax objects*. Those objects are native to Racket itself and thus can be dealt with using the built-in Racket compiler, debugger, diagnostic tools etc.

To summarize, the choice of Racket was motivated by the following tools for rapid compiler development provided out of the box:

— parser generation from the specified grammar,
— framework for abstract syntax tree processing,
— automatical integrated development environment (IDE) support for the new language.

3. SYNTAX

Landau syntax offers mutable real and integer variables, mutable real arrays, constants, `if/else` statements and `for` loops. Special type `parameter` is used to express variables which are not used in expressions, but have derivatives w.r.t. them. In case of dynamical equations differentiation such parameters could express initial conditions vectors. Special derivative operator `'` is used to annotate or assign the value of the derivative. Even with branching constructions (`if/else` statements), the function is guaranteed to be continuously differentiable thanks to the

prohibition of the real arguments inside the condition body. Moreover, it is allowed to manually omit negligibly small derivatives using the `discard` keyword (e.g. if $x(a) = y(a) + z(a) + t(a)$ and command `discard y ' a` is typed, then $\frac{\partial x}{\partial a} = \frac{\partial x}{\partial a} + \frac{\partial t}{\partial a}$).

Listing 1 demonstrates a Landau program for a dynamical system describing the motion of a spacecraft. The state of the system, i.e. the 3-dimensional position and velocity of the spacecraft, obeys Newtonian laws. The derivatives of the state w.r.t. 6 initial conditions (position and velocity) and one parameter (the gravitational parameter of the central body) are calculated using AD. Listing 2 shows a snippet (reduced for clarity) of the C code that is generated from the Landau code of Listing 1.

```
#lang landau

# Annotated parameters. Function does not have them directly
# as arguments, but has derivatives w.r.t. them in the state vector.
parameter[6] initial

real[6 + 36 + 6] x_dot (
  real[6 + 36 + 6] x, # state + derivatives w.r.t. initial and GM
  real GM)
{
  real[36] state_derivatives_initial = x[6 : 6 + 36]
  real[6] state_derivatives_gm = x[6 + 36 : ]
  real[6] state = x[ : 6]
  real[6] state_dot

  # Set the state vector's Jacobian values.
  state[ : ] ' initial[ : ] = state_derivatives_initial[ : ]
  state[ : ] ' GM = state_derivatives_gm

  # Transfer the time derivatives from x to their xdot counterparts,
  # because ẋ = v_x.
  state_dot[ : 3] = state[3 : ]

  # Write the velocity part to the function output.
  x_dot[ : 3] = state_dot[ : 3]

  # Apply Newtonian laws.
  real dist2 = sqr(state[0]) + sqr(state[1]) + sqr(state[2])
  real dist3inv = 1 / (dist2 * sqrt(dist2))

  state_dot[3 : ] = GM * (-state[ : 3]) * dist3inv

  # Write the acceleration part to the function output.
  x_dot[3 : ] = state_dot[3 : ]

  # Write the state_dot derivatives to the function output.
  x_dot[6 : 6 + 36] = state_dot[ : ] ' initial[ : ]
  x_dot[6 + 36 : 6 + 36 + 6] = state_dot[ : ] ' GM
}
```

Listing 1. Landau program for modeling spacecraft movement around a planet. Spacecraft's initial position and velocity, as well as the gravitational parameter of the planet, are supposed to be determined by nonlinear least-squares method

```
1   for (int derivative_index = 0; derivative_index < 6; derivative_index++){
2     d_dist2_d_initial[derivative_index] =
3       2 * state[0] * d_state_d_initial[0 * 6 + derivative_index] +
4         2 * state[1] * d_state_d_initial[1 * 6 + derivative_index] +
5           2 * state[2] * d_state_d_initial[2 * 6 + derivative_index];
6   }
7   double dist2 = pow(state[0], 2) + pow(state[1], 2) + pow(state[2], 2);
8
9   for (int derivative_index = 0; derivative_index < 6; derivative_index++){
10    d_dist3inv_d_initial[derivative_index] =
11      dist2 * sqrt(dist2) /
12        pow(dist2 * sqrt(dist2), 2) *
13          (dist2 * 0.5 * pow(dist2, -0.5) *
14            d_dist2_d_initial[derivative_index] +
15              d_dist2_d_initial[derivative_index] * sqrt(dist2));
16  }
17  double dist3inv = 1. / (dist2 * sqrt(dist2));
18
19  for (int slice_index = 0; slice_index < 3; slice_index++){
20    for (int derivative_index = 0; derivative_index < 6; derivative_index++){
21      d_state_dot_d_initial[(slice_index + 3) * 6 + derivative_index] =
22        GM * (- state[slice_index + 3]) *
23          d_dist3inv_d_initial[derivative_index] +
24            (GM * (- d_state_d_initial[(slice_index + 3) * 6 + derivative_index])) *
25              dist3inv;
26    }
27    state_dot[slice_index + 3] = GM * (- state[slice_index]) * dist3inv;
28  }
```

Listing 2. The C code generated by Landau compiler from lines 28–31 of the listing 1

4. IMPLEMENTATION

4.1. Preliminaries

The main goal is to develop a source-to-source compiler which translates a Landau program into Racket syntax objects or ANSI C code. A typical program in Landau includes functions performing a computation on arrays of real numbers. The compiler has not only to translate a function to a target language but to augment its body with a code for computing derivatives with respect to the desired variables.

In further sections, we share some experience gained during the development of the compiler, but first we should explain some basics of Racket.

Every Racket compiler consists of two components: the reader and the expander. The reader consumes the source as a text and either produces its abstract syntax tree (AST) according to the language grammar or raises a syntax error. Expander transforms AST to a Racket code. Racket's reader routine can be separated into two processes: lexing and parsing. Lexer has to split a solid source into a list of tokens. After a successful lexing stage, the list of tokens is passed to a parser, which generates AST. Racket exempts a programmer from writing parsers manually and provides the ability of parser generation from specified grammar. After the parsed has finished,

an AST with Racket syntax objects is emitted, where a syntax object is a key-value map with the following keys:

— source literal,
— source location,
— source lexical bindings,
— any optional key-value pairs (syntax properties).

Literals are just strings of the source code. Location is the source file path, row, and column. Lexical bindings can be thought of as links to the other syntax objects.

To implement expander, one needs to define functions operating on Racket syntax objects (nodes of AST) called macros. Macros transform the source code without executing it. The process of a macro application called “macro expansion” because a macro output is often longer than the input.

To expand AST, one needs to implement macros for all types of nodes. To illustrate the expander implementation technique we will write one for the `expr` node from the grammar (see Listing 3).

```
1  program : constant* parameter* (func | expr)*
2
3  constant : 'const' type IDENTIFIER '=' (expr | "{" parlist "}")
4
5  expr : term
6       | expr '+' term
7       | expr '-' term
8
9  term : factor
10      | term '*' factor
11      | term '/' factor
12
13 factor : primary '^' factor
14        | primary
15
16 primary : unop primary
17         | element
18
19 element : number
20         | '(' expr ')'
21         | get-value
22         | func-call
23
24 number : INTEGER | FLOAT
25 unop : '-' | '+'
26
27 ...
```

Listing 3. A part of Landau grammar

In case when we do not take into account variable types and are not interested in derivatives the macro expander for the `expr` node can be defined as shown in Listing 4.

```
;; ‘define-syntax‘ means that defined function ought to be running in compile time
;; ‘expr‘ matches the name of the AST node syntax object.
(define-syntax (expr stx)
    ;; Parses the part of AST with ‘expr‘ in the root.
    (syntax-parse stx
    ;; Matches expression of sum, ‘_‘ matches any literal. Binds left and right
    ;; children (subtrees) of ‘expr‘ node to so called pattern variables.
    ((_ expr '+' term)
      ;; Output syntax object is an addition of the children nodes.
      ;; Racket will expand ‘expr‘ and ‘term‘ subtrees just before
      ;; substituting to the output syntax object.
      #'(+ expr term))
    ((_ expr '-' term)
      #'(- expr term))
    [(_ term-body)
      #'term-body]))
```

Listing 4. The simplified implementation of the `expr` macro.

4.2. Syntax parameters

In trivial example of Listing 4, the macro is not aware of the context it is invoked in, but in a real language we have to make local variables, function parameters, function name and type visible to each function body's subtree (but not to other function's child nodes). Passing all this parameters to each macro would be too cumbersome. In Racket, this problem is solved by *dynamic bindings* of parameters used in the expansion of syntax objects. See Listing 5 for an example of dynamic bindings with the `make-parameter` and `parameterize` constructions. The key idea is that binding created with the `make-parameter` can be overridden by `parameterize` macro. A function, parameterized in such a manner, can use the parameter like a global variable but it has a predictable value specified in the argument of `parameterize`. A parameter is automatically set to the default value when called outside the `parameterize` body.

```
(define x (make-parameter 0))

(print ( * 100 (x)))
;; prints: 0

(define (some-complicated-function a)
  ( * a (x)))

(parameterize ([x 2])
  (print
    (some-complicated-function 100)))
;; prints: 200

(print ( * 100 (x)))
;; prints: 0
```

Listing 5. Illustrating the dynamic bindings concept with the help of Racket's parameters.

Racket provides dynamical binding mechanism not only for ordinary code, but for macro expansion as well, which is suitable for providing context to the desired subtrees of our AST. It this case the parametrization is performed with `make-syntax-parameter` and `syntax-parameterize` constructions (Listing 6).

```
(define-syntax (func stx)
  (syntax-parse stx
    (({~literal func} type name "(" ({~literal arglist} arg*:arg-spec ...) ")"
                      "{" body ... "}")

     (let* (;; generate a symbol for function's output symbol
            (func-return-value (gensym))
            (func-return-type (parse-type (syntax->datum #'type)))
            ;; generate arguments container
            (args (make-hash))
            ;; generate function's local variables container
            (local-variables (initiate-local-variables))
            ;; generate arguments' symbols
            (argnames (args->argnames #'arg*)))
       (with-syntax ((ret (datum->syntax stx func-return-value)))
         (quasisyntax/loc stx
             (begin
               (define (name #,@argnames)
                 (syntax-parameterize
                     ;; parameterize function's body with
                     ;; respect to variables and arguments container,
                     ;; function's name, return type and output variable.
                      ((local-variables '#,local-variables)
                       (function-name (syntax->datum #'name))
                       (function-return-value '#,func-return-value)
                       (function-return-type '#,func-return-type)
                       (current-arguments '#,args))
                    (let ((ret #,(instantiate func-return-type)))
                      body ...
                      ret)))
               (provide name)))))))
```

Listing 6. Variables scope handling with the help of the Racket syntax parameters

4.3. Syntax properties

Another AST operation worth discussing is devoted to the ability for any tree node to be aware of the properties assigned to the leaves. Such a necessity arises for example in developing a type system. Our language is a statically typed one and we want to be able to cast types and throw an exception if for example `real` valued expression is assigned to an `int` variable. Racket allows to assign any key-value pair to the syntax object and to manually force macro expansion of children's expression subtrees so the parent nodes can be aware of the children's properties.

```
(define-syntax (number stx)
  (syntax-parse stx
    (((~literal number) number-stx)
     (syntax-property
      #'number-stx
```

```
      'landau-type
      (if (exact? (syntax->datum #'number-stx))
          'int
          'real)
      #t))))
```

Listing 7. Using the syntax properties to set type of leave node: `int` or `real`

```
(define-syntax (expr stx)
  (syntax-parse stx
  ((_ expr '+' term)
  (let ((expr (local-expand #'expr 'expression '()))
        (term (local-expand #'term 'expression '()))
        (type1 (syntax-property expr 'landau-type))
        (type2 (syntax-property term 'landau-type)))
        (match (list type1 type2)
          ((list 'real 'real)
            (is-real
              #'(+ expr term)))
          ((list 'real 'int)
            (is-real
              #'(+ expr (int->real term))))
              ...)))))
```

Listing 8. Using the syntax properties and `local-expand` to expand children's subtrees. In case when both subtrees have the type of 'real they are passed without changes. If the right subtree has the type of 'int it is explicitly converted to the real number.

In fact, Landau uses more types to describe variables, slices, arrays and their dual counterparts. Some of them are assigned by the compiler automatically and are not controlled by a user. For example, the `dual` type is assigned to the variables which are known to have derivatives. Then, the variables of the `dual` type are expanded to the syntax object carrying the value and the derivative part of the expression. `expr` and `term` macros perform expansion of children subtrees and if at least one of them has the type of `dual`, they generate syntax object of type `dual` filled with a pair of expressions for computing the value and derivative part, where the derivative part is constructed from the operand's value and derivative syntax parts according to the differentiation rules.

```
(define-syntax (term stx)
  (syntax-parse stx
    ((_ term '*' factor)
       (let* ((term-expanded (local-expand #'term 'expression '()))
              (factor-expanded (local-expand #'factor 'expression '()))
              (type1 (syntax-property term-expanded 'landau-type))
              (type2 (syntax-property factor-expanded 'landau-type)))

          (match (list type1 type2)
           ((list 'dual 'dual)
            (with-syntax*
             ((expanded-dual-expr-1 term-expanded)
              (dual-b-value-1
               (get-value-stx #'expanded-dual-expr-1))
```

```
15              (dual-b-derivative-1
16               (get-derivative-stx #'expanded-dual-expr-1))
17
18              (expanded-dual-expr-2 factor-expanded)
19              (dual-b-value-2
20               (get-value-stx #'expanded-dual-expr-2))
21              (dual-b-derivative-2
22               (get-derivative-stx #'expanded-dual-expr-2))
23
24              (result (is-type_ type
25                #'(list
26                    (fl*
27                      dual-b-value-1
28                      dual-b-value-2)
29                    (fl+ (fl* dual-b-value-1 dual-b-derivative-2)
30                       (fl* dual-b-value-2 dual-b-derivative-1))))))
31            #'result))))))))))
```

Listing 9. The part of the `term` macro illustrating the expression differentiation technique

4.4. Phase levels

Before macro expansion starts producing AD code, the set of derivatives for each `real` variable should be established. One way to do that is to trace the function evaluation from the bottom to the top and pass the collected information to macros. That can be achieved by transforming the function body to a flat sequence of actions which means unrolling all loops and precomputing branching conditions before code generation starts. Because Landau is a Turing incomplete language, it is guaranteed that every Landau's function can be processed that way.

In order to express the above-described idea in Racket, it is crucial to understand the concept of Racket's phase levels, which we will briefly describe. Like in various other languages, Racked bindings can be aggregated in modules. A module can export and import bindings with `(require module-name)` and `(provide module-binding)` commands. Racket allows swapping between namespaces of different modules which means we can redefine macros with different semantics in separate module and traverse our AST twice: backward — to perform static source code analysis, forward — to generate a target code relying on information from the previous stage.

Each definition (binding) in Racket has a so-called phase level. Code with different phase levels has separate namespaces and execution time. For example, identifiers bound with `define` keyword are runtime bindings, e.g. they are used during the program execution. In terms of racket phases, it has phase level 0. Keyword `begin-for-syntax` shifts its body's phase level one step forward. Bindings defined with `(begin-for-syntax (define ...))` are used in compile-time and have phase level 1.

The phase level of imported binding depends on the way it was imported. For example, `(require module-name)` leaves phase level of all `module-name`'s bindings unchanged, but `(require (for-syntax module-name))` shifts all `module-name`'s bindings one step forward.

```
1   codegeneration.rkt:
2
3   ;; match the root node of AST
4   (define-syntax (program stx)
```

```
5     (syntax-parse stx
6      ;; binds body to the children subtrees of ‘program‘ node
7      [({~literal program} body ...)
8
9       ;; bind info variable to the output of generated by backrun.rkt macros code
10      (let ((info
11              ;; expand AST using macros defined in backrun.rkt module
12              (parameterize
13                  ([current-namespace
14                    (module->namespace
15                      (collection-file-path "backrun.rkt" "landau"))])
16                ;; evaluate generated by backrun.rkt macros code
17                (eval stx))))
18
19          ;; expand AST to generate target code
20          ;; using the ‘info‘ variable.
21          #‘(syntax-parameterize ((info ’#,info))
22              (begin body ...)))]))
23
24    backrun.rkt:
25
26    (define-syntax (program stx)
27     (syntax-parse stx
28      [(_ body ...)
29       (quasisyntax/loc
30         stx
31         (process #,#’(list body ...)))]))
```

Listing 10. Two stage compilation in simplified case when the program includes only a single function. The `program` macro implementation code parts from `codegeneration.rkt` and `backrun.rkt` files. Before `program` from `codegeneration.rkt` emit it's syntax object it expands AST with macros defined in `backrun.rkt` and evaluates expanded racket code. The result of the program evaluation is bound to the `info` variable and used in expansion of the `program`'s body.

Macros defined in the `backrun.rkt` module expand AST in a natural forward way, meanwhile performing the code validation like variables declarations checks and type checking. The output of the macro expansion is the Racket code for generating the intermediate form of AST with only the following nodes preserved: derivative annotations, variable assignations and function's derivatives location markers. For example, macros transform array's cell access node to index range check routine, loop node is transformed into the routine emitting the unrolled list of body actions. This generated code initially has a phase level 0, but the phase is shifted during the import which allows us to run it in compile time inside the `program` macro of the code generation module.

4.5. IDE support

While the compiler itself can be used as a standalone console program, it would be great to have some integrated development environment (IDE) support at least for the code highlighting and nice compilation error reports with error location highlighting right in the source. The good point in not only Racket is shipped with IDE (called DrRacket), but that any language implemented in Racket automatically supported by DrRacket. That means no additional work needs to be done to make the language ready to use. The example of IDE usage is presented in Figure 1,

where we deliberately made an array access mistake which was duly reported by the compiler and nicely shown in DrRacket.

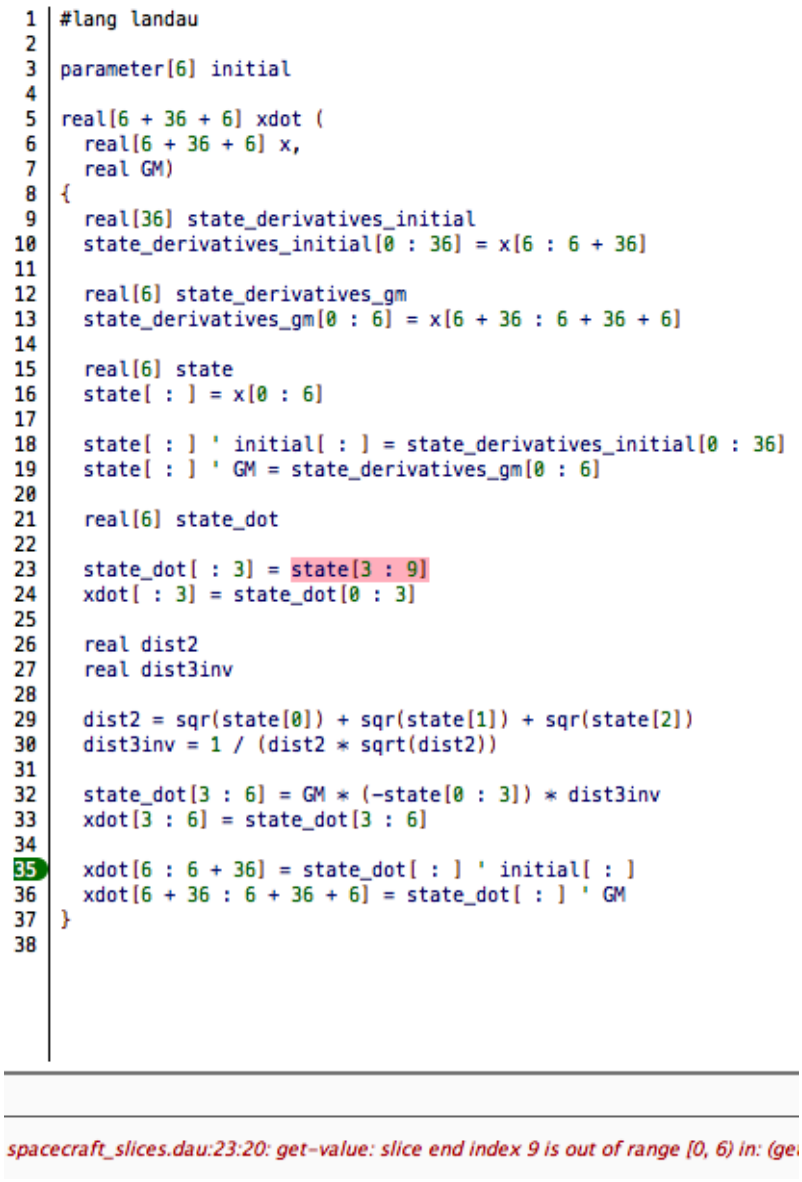

Figure 1. Demonstration of the DrRacket IDE. The compilation error tells user that the slice end index 9 is greater then `state` array's maximal available index, IDE highlights error location

5. CONCLUSION

A new language called Landau has been invented to fill the niche of a domain-specific language designed for practically usable forward-mode AD for estimating the values of free parameters of a complex dynamical system.

It has been shown that the Racket platform has tools that are useful for rapid compiler development. Some know-how of building a source-to-source compiler have been presented, that should be useful for further attempts to design and implement complex DSLs.

Acknowledgements. Authors are thankful to Matthew Flatt and Matthias Felleisen of the PLT Racket team for their help with the Racket programming platform.

References

1. M. Tadjouddine, S. A. Forth, and J. D. Pryce, “AD tools and prospects for optimal AD in CFD flux Jacobian calculations,” *Automatic differentiation of algorithms*, NY: Springer, pp. 255–261, 2002. doi: 10.1007/978-1-4613-0075-5_30
2. C. Bischof, A. Carle, G. Corliss, A. Griewank, and P. Hovland, “ADIFOR–generating derivative codes from Fortran programs,” *Scientific Programming*, vol. 1, no. 1, pp. 11—29, 1992; doi: 10.4236/ojas.2013.34040
3. C. H. Bischof, L. Roh, and A. J. Mauer-Oats, “ADIC: an extensible automatic differentiation tool for ANSI-C,” *Software: Practice and Experience*, vol. 27, no. 12, pp. 1427—1456, 1997.
4. A. Griewank, D. Juedes, and J. Utke, “Algorithm 755: ADOL-C: a package for the automatic differentiation of algorithms written in C/C++,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 22, no. 2, pp. 131–167, 1996.
5. I. Dolgakov and D. Pavlov, “Landau: language for dynamical systems with automatic differentiation,” *preprint arXiv:1905.10206*, 2019.
6. M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S.Tobin-Hochstadt, “A programmable programming language,” *Communications of the ACM*, vol. 61, no 3, pp. 62–71, 2018; doi: 10.1145/3127323
7. D. A. Pavlov, “Developing a programming language in Racket,” *Computer tools in Education*, no. 5, pp. 46–63, 2012 (in Russian).
8. D. A. Pavlov, “Creating domain-specific languages,” *Computer Tools in Education*, no. 6, pp. 57–60, 2011 (in Russian).

Received 10.07.2019, the final version — 21.08.2019.

Ivan A. Dolgakov, research engineer at the Laboratory of Ehemeris Astronomy, Institute of Applied Astronomy RAS, ✉ ia.dolgakov@iaaras.ru

Dmitry A. Pavlov, PhD, senior researcher at the Laboratory of Ehemeris Astronomy, Institute of Applied Astronomy RAS, dpavlov@iaaras.ru