

Расширение транслятора
предметно-ориентированного
языка Landau арифметикой
сдвоенных чисел двойной
точности (double-double)

Языково-ориентированное программирование

- Языково-ориентированное программирование - решение проблемы программирования путем создания нового языка и затем написание программы на нём.
- Создаваемые языки называются **предметно-ориентированными языками** (DSL), они удобны для применения в задачах с узкой специализацией.
- Racket подходит для ЯОП из-за своей **системы макросов**. Они работают в стиле компилятора, упрощая преобразование кода.

Преимущества Racket

- Данный язык обладает проработанной системой макросов, что важно для определения синтаксиса языка.
- Racket позволяет компилировать текст в специальные *синтаксические объекты* при написании макросов.
- Синтаксические объекты хранят не только строку макроса, но так же связанные с ней метаданные (местоположение в коде, синтаксический контекст и дополнительные свойства), эти данные используются при обмене между макросами.
- Сохранение лексического контекста сильно упрощает написание предметно-ориентированных языков, например уменьшается возможность коллизий названий объектов.

Landau

Landau – язык для динамических систем с автоматическим дифференцированием (AD).

В языке есть функции, циклы, условия, вещественные числа, массивы.

Landau достаточно полон для того, чтобы выразить и продифференцировать уравнение любой сложности за минимальные усилия.

```
#lang landau
```

```
parameter[6] initial
```

```
real[6 + 36 + 6] xdot (  
  real[6 + 36 + 6] x,  
  real GM)
```

```
{  
  real[36] state_derivatives_initial  
  state_derivatives_initial[0 : 36] = x[6 : 6 + 36]  
  
  real[6] state_derivatives_gm  
  state_derivatives_gm[0 : 6] = x[6 + 36 : 6 + 36 + 6]
```

```
  real[6] state  
  state[ : ] = x[0 : 6]
```

```
  state[ : ] ' initial[ : ] = state_derivatives_initial[0 : 36]  
  state[ : ] ' GM = state_derivatives_gm[0 : 6]
```

```
  real[6] state_dot
```

```
  state_dot[ : 3] = state[3 : 6]  
  xdot[ : 3] = state_dot[0 : 3]
```

```
  real dist2  
  real dist3inv
```

```
  dist2 = sqr(state[0]) + sqr(state[1]) + sqr(state[2])  
  dist3inv = 1 / (dist2 * sqrt(dist2))
```

```
  state_dot[3 : 6] = GM * (-state[0 : 3]) * dist3inv  
  xdot[3 : 6] = state_dot[3 : 6]
```

```
  xdot[6 : 6 + 36] = state_dot[ : ] ' initial[ : ]  
  xdot[6 + 36 : 6 + 36 + 6] = state_dot[ : ] ' GM  
}
```

Преимущества языка

- Широкие возможности языков общего назначения затрудняют реализацию удобной системы AD, особенно при наличии векторных функций со многими независимыми переменными.
- Существующие инструменты для дифференцирования функций, такие как MATLAB (ADMAT) или Mathematica (TIDES) часто требуют больших усилий (по сравнению с языками общего назначения) для ввода практической динамической системы большого размера с множеством свободных переменных.

Компоненты реализации Landau

- Транслятор Landau написан на Racket и состоит из двух основных частей:
 - Синтаксический анализатор
 - Разворачивание циклов в последовательности присваиваний.
 - Отслеживание зависимости переменных и их производных.
 - Проверка программы на корректность.
 - Генератор кода
 - Генерация ANSI C и Racket кода.

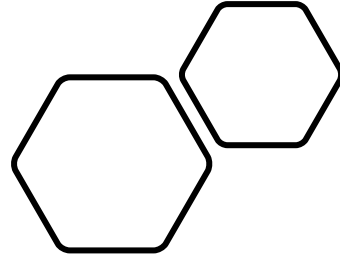
Синтаксический анализатор

- Получает на вход список токенов и генерирует из них абстрактное синтаксическое дерево (AST).
- На данном этапе обрабатывается большинство возможных ошибок в коде.

Генератор кода

- Получает на вход AST из синтаксических объектов
- Использует макросы Racket для трансформирования литералов исходного кода без его выполнения
- Узлы дерева преобразуются в соответствии с грамматикой целевого языка (завести необходимые переменные, выставить соответствующие идентификаторы и т.д.)
- В итоге код Landau транслируется в код на соответствующем языке (Racket/ANSI-C)

Double-double



- Число double-double выражается как сумма двух компонент double (большого и маленького числа) $x = x_l + x_h$.
- Арифметика double-double имеет программную реализацию.

Интеграция double-double.
Описание основных
модулей, в которых
происходили изменения.

- [#semantics.rkt](#) модуль с логикой генерации кода вычисления производных. Он использует информацию из синтаксического анализатора и генерирует код в промежуточном представлении.
- Были добавлены ветвления для операций с double-double и double ([сложение/вычитание](#), [умножение/деление](#)).

```

(cond
  ((or (equal? type 'real) (is-slice-of-type 'real type))
    (cond
      [(and expr1-atom expr2-atom) (is-type_ type (quasisyntax/loc stx #,((if (equal? op "+") rl+ rl-) expr1-atom expr2-atom)))]
      [(and expr1-atom) (is-type_ type (datum->syntax stx `((if (equal? op "+") #'_rldl+ #'_rldl-) ,expr1-atom ,expr2)))]
      [(and expr2-atom) (is-type_ type (datum->syntax stx `((if (equal? op "+") #'_rldr+ #'_rldr-) ,expr1 ,expr2-atom)))]
      [else (is-type_ type (datum->syntax stx `((if (equal? op "+") #'_rl+ #'_rl-) ,expr1 ,expr2)))] ) )

```

Landau код

```

#lang landau

real[3] test_function_1(real[3] a, real b)
{
  real c = b + 0.5 - a[0]
  test_function_1[:] = a[:] + b
}

```



ANSI C код

```

int test_function_1(dd *restrict test_function_111, dd *restrict a12, dd b13) {
  dd c14;
  c14 = dd_sub(dd_add_dd_d(b13 , 0.5e0) , a12[0]);
  {
    for (int slice_idx = 0; slice_idx < 3; slice_idx++) {
      test_function_111[(0 + slice_idx)] = dd_add(a12[(0 + slice_idx)] , b13);
    }
  }
  return 0;
}

```

#[metalang.rkt](#) модуль описывает макросы промежуточного представления для всех бэкендов. Каждый добавленный сюда макрос должен быть экспортирован из semantics.rkt.

Написаны новые [функции](#), которые используются в semantics для операций с операндами double и double-double.

```
(define-syntax (_rldr+ stx)
  (syntax-parse stx
    ((_ x y)
      (match (target-lang TARGET)
        ('racket
          (quasisyntax/loc stx (#,(match (target-real-implementation TARGET) ('double #'fl+) ('long-double #'extfl+)) x y)))
        ('ansi-c
          (syntax/loc stx (crdr+ (to-string x) (to-string y))))))
    ((_ x rest ...)
      (match (target-lang TARGET)
        ('racket
          (quasisyntax/loc
            stx
            (#,(match (target-real-implementation TARGET) ('double #'fl+) ('long-double #'extfl+)) x (_rldr+ #,@#'(rest ...))))))
        ('ansi-c
          (quasisyntax/loc stx (crdr+ (to-string x) (_rldr+ #,@#'(rest ...)))))))))
```

- [#combinators.rkt](#) модуль описывает функции кодогенерации для ansi-c бэкенда.
- Написаны новые функции и ветвления, которые генерируют ANSI-C код.
- К примеру [функция](#) для сложения в случае double-double бэкенда генерирует C-функцию dd_add, а в случае double и long double бэкендов генерирует "+".

Сгенерированный C код с double бэкендом

```
int test_function_1(double *restrict test_function_111, double *restrict a12, double b13) {
    double c14;
    c14 = ((b13 + 0.5e0) - a12[0]);
    {
        for (int slice_idx = 0; slice_idx < 3; slice_idx++) {
            test_function_111[(0 + slice_idx)] = (a12[(0 + slice_idx)] + b13);
        }
    }
    return 0;
}
```

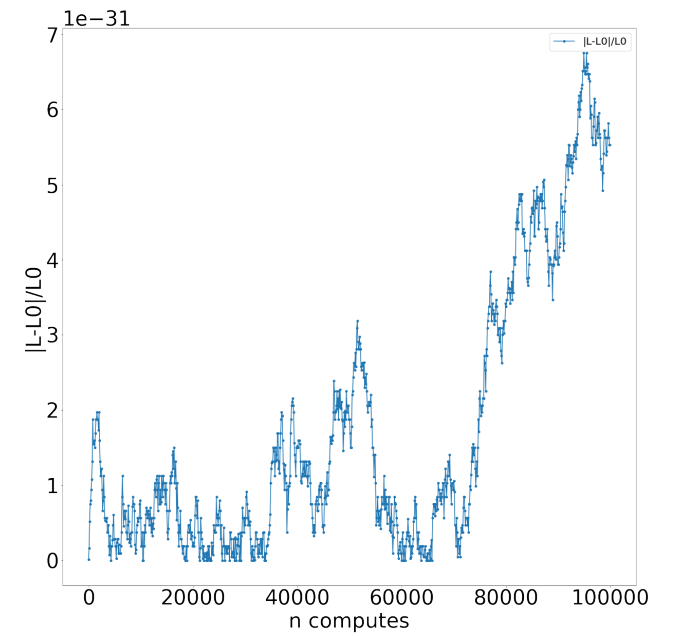
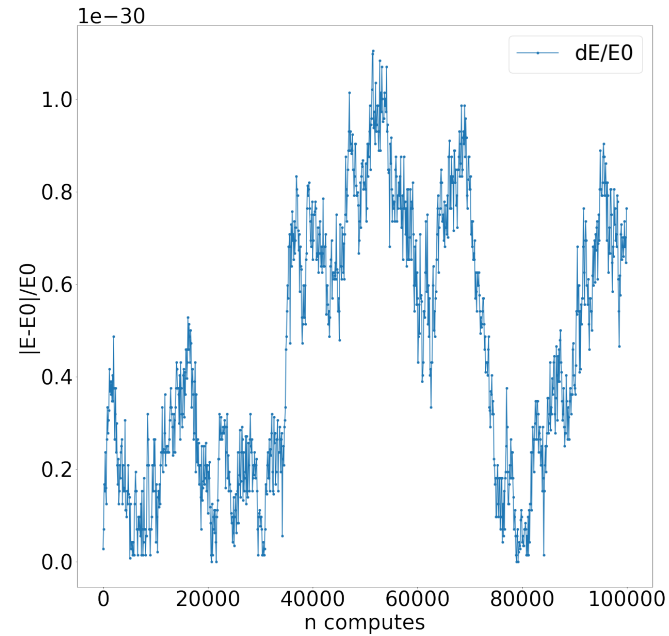
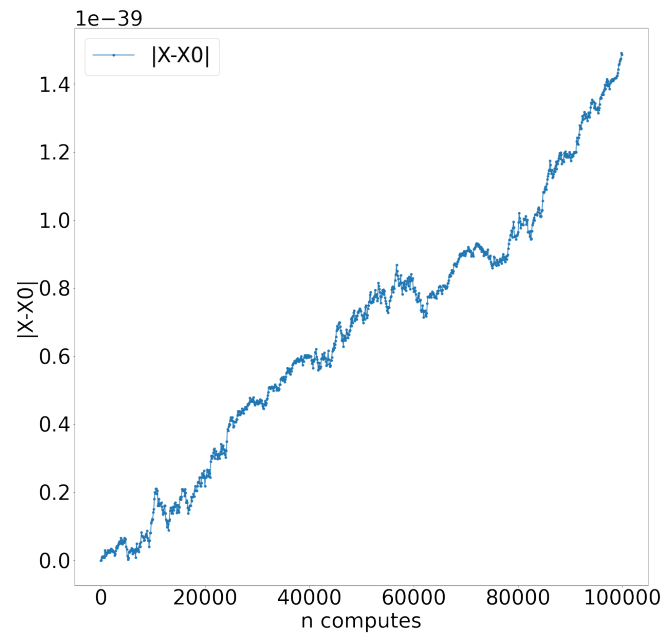
Сгенерированный C код с double-double бэкендом

```
int test_function_1(dd *restrict test_function_111, dd *restrict a12, dd b13) {
    dd c14;
    c14 = dd_sub(dd_add_dd_d(b13 , 0.5e0) , a12[0]);
    {
        for (int slice_idx = 0; slice_idx < 3; slice_idx++) {
            test_function_111[(0 + slice_idx)] = dd_add(a12[(0 + slice_idx)] , b13);
        }
    }
    return 0;
}
```


Внедрение сгенерированного ANSI-C кода с помощью Landau в проект с задачей N тел

- На языке Landau была написана функция для расчёта состояния системы N тел в следующий момент времени.
- Затем был сгенерирован C код.
- Из библиотеки QD были выбраны и переписаны на ANSI-C нужные функции для работы с double-double.
- Из этого кода была сгенерирована статическая библиотека и интегрирована в проект с задачей N тел.

Результаты работы программы интегрирования с функциями из Landau сходятся с исходной программой.



Вывод

- В Landaui был успешно интегрирован тип данных double-double.
- Была написана и применена в стороннем проекте функция.