ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
Факультет компьютерных наук
Образовательная программа «Программная инженерия»

СОГЛАСОВАНО

Научный руководитель, старший преподаватель департамента больших данных и информационного поиска

_____ В. В. Куренков

«___» _____ 2025 г.

УТВЕРЖДЕНО

Академический руководитель образовательной программы «Программная инженерия», старший преподаватель департамента программной инженерии

_____ Н. А. Павлочев

«___» _____ 2025 г.

Подп. и дата	
Инв.№ дубл.	
Взам. инв.№	
Подп. и дата	
Инв.№ подл	

СИСТЕМА ПОСТРОЕНИЕ ГЕОМЕТРИЧЕСКИХ ЧЕРТЕЖЕЙ СО ВСТРОЕННЫМ ЯЗЫКОМ ПРОГРАММИРОВАНИЯ И ВОЗМОЖНОСТЬЮ УДАЛЕННОГО ПРОГРАММНОГО УПРАВЛЕНИЯ

Описание языка

ЛИСТ УТВЕРЖДЕНИЯ

RU.17701729.12.17-01 35 01-1-ЛУ

Исполнитель:

Студент группы БПИ233

_____ / С. А. Чубий /

«___» _____ 2025 г.

Москва 2025

УТВЕРЖДЕН

RU.17701729.12.17-01 35 01-1-ЛУ

СИСТЕМА ПОСТРОЕНИЕ ГЕОМЕТРИЧЕСКИХ ЧЕРТЕЖЕЙ СО ВСТРОЕННЫМ ЯЗЫКОМ ПРОГРАММИРОВАНИЯ И ВОЗМОЖНОСТЬЮ УДАЛЕННОГО ПРОГРАММНОГО УПРАВЛЕНИЯ

Описание языка

RU.17701729.12.17-01 35 01-1

Листов 15

Москва 2025

Инв.№ подл	Подп. и дата	Взам. инв.№	Инв.№ дубл.	Подп. и дата

СОДЕРЖАНИЕ

1. ОБЩИЕ СВЕДЕНИЯ

Встроенный язык программирования системы Geometrica (далее Язык) совмещает в себе элементы как императивного, так и функционального программирования. Имеет сильную статическую типизацию.

Верхнеуровневые (top-level) конструкции языка являются императивными: они выполняются последовательно, одна за другой, и могут удалять, добавлять и изменять элементы чертежа. Они дают пользователю возможность интерактивно работать с геометрическим рисунком.

Все вычисления внутри чертежа (например, арифметические операции с числами и векторами, построения прямых по точкам и т.д.) задаются в функциональном стиле. Это позволяет определить структуру чертежа и впоследствии легко пересчитывать все его элементы.

2. ЭЛЕМЕНТЫ ЯЗЫКА

Далее дано описание синтаксиса Языка на русском языке. Синтаксические элементы выделены *курсивом*.

Более формальное объявление можно найти в исходном коде, в файле `crates/parser/src/parser/mod.rs`. Парсер написан с использованием библиотеки `peg` и схож с PEG грамматикой.

О процессе исполнения кода смотри «Пояснительную записку», гл. 3.2.4 «Крейт Client».

2.1. Скрипт

Скрипт на Языке состоит из *выражений* (`Statement`; Раздел 2.2), разделенных пробельными символами[1].

2.2. Выражение (`Statement`)

Каждое *выражение* (`Statement`) является либо *объявлением функции* (`FunctionDefinition`; Раздел 2.6), либо *объявлением значение* (`ValueDefinition`; Раздел 2.5), либо *вызовом команды* (`Command`; Раздел 2.3).

2.3. Вызов команды (`Command`)

Вызов команды состоит из *названия команды*, и *аргументов команды*, разделенных пробельными символами.

Названием команды является *идентификатор* (`Ident`; Раздел 2.4), к которому последним символом дописан `!` (восклицательный знак).

Аргументом команды может быть либо *идентификатор*, либо *выражение* (`Expr`; Раздел 2.7).

2.3.1. Примеры

- `set! x 10.0`
- `set! x (1.0 + 2.0)`
- ```
  eval! (
      (x + y) / 2
  )
  ```

2.4. Идентификатор (`Ident`)

Идентификатором является непустая последовательность символов, где первый символ может быть маленькой или большой латинской буквой или нижним подчеркиванием (`[a-zA-Z_]`), а последующие могут быть всем вышеперечисленным или цифрой (`[a-zA-Z0-9_]`).

[1] Имеются в виду символы `'\n'` (конец строки), `'\t'` (таб) и `' '` (пробел).
  ```

Идентификаторы используются для имен переменных, команд и функций.

2.5. Объявление значение (`ValueDefinition`)

Объявление значения состоит из *имени значения*, *типа значения* (опционально) и *тела значения* (`Expr`; Раздел 2.7).

Имя значения является *идентификатором* (`Ident`; Раздел 2.4).

Тип значения отделяется от имени символом `:` (двоеточие). Если тип не указан явно, то он будет определен из *выражения*.

Тело значения является *выражением* (`Expr`; Раздел 2.7) и отделяется знаком = (равно). В *теле значения* могу быть использованы определенные раннее переменные.

Рекурсивные объявления запрещены: переменная не может определяться сама через себя. Переопределения запрещены: нельзя дважды определить одну и ту же переменную (при этом можно обновить значение переменной, используя команду `set!`).

2.5.1. Примеры

- Без явного указания типа:

```
x = 1
```

- С явным указанием типа:

```
y: int = 10
```

- Со сложным выражением в правой части:

```
z = 1 + 1
```

- С другими переменными в правой части (`x`, `y` и `z` должны быть определены):

```
w = (x + y) / 2 - z
```

2.6. Объявление функции (`FunctionDefinition`)

Объявление функции состоит из *имени функции*, *объявлений аргументов*, разделенных пробелами, *возвращаемого типа* и *тела функции*.

Имя функции является идентификатором (`Ident`; Раздел 2.4).

Объявление аргумента состоит из *имени аргумента* и его *типа*. *Тип* отделяется символом `:` (двоеточие).

Возвращаемый тип отделяется символами `->`.

Тело функции является *выражением* (`Expr`; Раздел 2.7) и отделяется знаком = (равно). В *теле функции* из переменных могут быть использованы только аргументы функции или переменные определенные внутри тела функции. Переменные, определенные, вне функции использовать **нельзя**.

Возможна перегрузка функций: можно создать несколько функций одинаковыми именами, но разными типами аргументов. Возможна рекурсия: функция может вызывать саму себя. Переопределение функций запрещено.

2.6.1. Примеры

- Обычное объявление:

```
avg a:real b:real -> real = (a + b) / 2
```

- Перегрузка:

```
sum a:int b:int -> int = a + b
sum a:real b:real -> real = a + b
```

2.7. Выражение (`Expr`)

Выражение (`Expr`) может быть одним из следующих:

- *Литерал* (`Value`; Раздел 2.7.1)
- *Переменная* (`Ident`; Раздел 2.4)
- *Выражение* `let` (`LetExpr`; Раздел 2.7.2)
- *Условное выражение* (`IfExpr`; Раздел 2.7.3)
- *Вызов функции* (`FuncCallExpr`; Раздел 2.7.4)
- *Dot-нотация* (Раздел 2.7.5)
- *Применение унарного оператора* (`UnaryExpr`; Раздел 2.7.6)
- *Применение инфиксного оператора* (`InfixExpr`; Раздел 2.7.7)
- *Преобразование типа* (`AsExpr`; Раздел 2.7.8)

2.7.1. Литерал (`Value`)

Значения бывают следующих типов (после названия типа приведены примеры значений этих типов):

- `bool`: `true`, `false`
- `int`: `-5`, `42`
- `real`: `10.0`, `-1.0`, `1e7`, `1.7e4`
- `str`: `"Hello, world!"`, `"Two\nlines"`, `"\"Quoted\""`
- `pt`: `pt 100.0 100.0`

- `line`: `line p1 p2`, где `p1` и `p2` — переменные типа `pt`
- `circ`: `circ o r`, где `o` — переменная типа `pt`, центр окружности. `r` — переменная типа `real`, радиус окружности

Типы `bool`, `int`, `real` и `str` можно создать при помощи *литерала*. Для `pt`, `line` и `circ` литералов не существует — значения этих типов можно получить, воспользовавшись соответствующими встроенными функциями (смотри примеры выше и Раздел 3.4).

Каждое значение также может быть пустым. Для создания пустого значения используйте конструкцию `none <type>`. Например, `none pt`.

2.7.2. Выражение `let` (`LetExpr`)

Выражение `let` состоит из *объявления переменных*, разделяемых запятыми , и *тела выражения* `let`. Запятая после последнего *объявления* также допустима.

Объявление переменной в *выражении* `let` аналогично обычному объявлению переменной.

Тело выражения `let` является *выражением* (`Expr`; Раздел 2.7). В нем можно использовать все переменные, которые уже были доступны, а также те, что были объявлены в данном `let`.

2.7.2.1. Примеры

- Без внешних переменных:

```
let
  x = 1,
  y = 2,
in
  x + y
```

- С внешними переменными (пусть переменная `x` уже объявлена):

```
let t = x^2 + x in t^2 - 10 * t + 7
```

2.7.3. Условное выражение (`IfExpr`)

Условное выражение состоит из набора *веток*, разделенных запятыми, и *ветки else* (опционально). Запятая после последней *ветки* также допустима.

Ветка состоит из *условия* – выражения (`Expr`; Раздел 2.7), типа `bool`, и *значения* – выражения любого типа.

Ветка else состоит из *значения* – выражения любого типа.

Значения всех *веток* (включая *ветку else*) должны иметь одинаковый тип.

Все ветки рассматриваются в порядке их следования. If возвращает значение первой ветки, *условие* которой равно `true`. Если все *условия* равны `false`, то возвращается *значение ветки else*. Если *ветка else* отсутствует, то возвращается ошибка.

2.7.3.1. Примеры

- Без ветки else:

```
if
    1 == 2 then "equals",
    1 < 2  then "less",
    2 > 1  then "greater",
```

- С веткой else:

```
if
    1 == 2 then "equals",
    1 < 2  then "less",
    else        "greater"
```

2.7.4. Вызов функции (`FuncCallExpr`)

Вызов функции состоит из *имени функции* и её *аргументов*, разделенных пробельными символами.

Аргументом является выражение (`Expr`).

2.7.4.1. Примеры

- `sum 1.0 2.0`
- `max (2 * x) (avg x 10)`

2.7.5. Dot-нотация

Dot-нотация является альтернативным синтаксисом для вызова функции **с ровно одним аргументом**. Она состоит из *аргумента* и *имени функции*, определенных также, как и при простом вызове функции.

Dot-нотация полезная для функций-`getter`-ов и для последовательного вызова нескольких функций.

2.7.5.1. Примеры

Пусть `p` — переменная типа `pt`, `l` — переменная типа `line`, `x` — функция, возвращающая `x` координату точки, `p1` — функция возвращающая первую из двух точек, по которым была построена прямая.

- `p.x`

  Возвращает `x` координату точки `p`. Равносильно обычному вызову функции:

```
x p
```

- Цепочка вызовов:

```
l.p1.x
```

- Со сложным выражением:

```
(pt 100.0 200.0).x
```

2.7.6. Применение унарного оператора (`UnaryExpr`)

Применение унарного оператора состоит из *унарного оператора* и *выражения*.

Раздел 3.1 содержит список всех *унарных операторов*.

2.7.6.1. Примеры

- `-x`

2.7.7. Применение инфиксного оператора (`InfixExpr`)

Применение унарного оператора состоит из *левого выражения*, *инфиксного оператора* и *правого выражения*.

Раздел 3.2 содержит список всех *бинарных операторов*.

2.7.7.1. Примеры

- `x + y`
- `(10 * x + 7 * y)^2`

2.7.8. Преобразование типа (`AsExpr`)

Преобразование типа состоит из *выражения* и *типа*.

Выражение будет преобразовано в указанный *тип*, если это возможно. Иначе, произойдет ошибка.

2.7.8.1. Примеры

- `(x + y) as real`
- `100 as str`

2.8. Комментарии

Комментарии могут быть помещены почти в любое место кода. Они бывают двух видов: многострочные и однострочные. Однострочный комментарий может содержать любые символы, кроме `'\n'` (конца строки); многострочный — любые символы, кроме последовательности `"*/"` (звездочка, слеш).

2.8.1. Примеры

- Однострочный:

```
// Comment here
```

- Многострочный:

```
/*
    There is a lot
    of text here!
    Isn't it?
*/
```

- В объявлении функции:

```
// Being very verbose here:
sum // <- this is a function name
    x /* first arg */: /* it's a type */ int
    y: int // second arg here!
-> int // <- return type
= x + y
```

3. ВСТРОЕННЫЕ ЭЛЕМЕНТЫ

Здесь указан **минимальный** список встроенных элементов. Конкретная реализация может расширять этот список. Для получения полного списка используйте функции `list_func!` и `list_cmd!` (смотри Раздел 3.3).

3.1. Унарные операторы

Оператор	Сигнатура	Операция
`!`	`bool -> bool`	Булево «НЕ»
`-`	`int -> int`	Отрицание
	`real -> real`	
	`pt -> pt`	

Таблица 1. Встроенные унарные операторы

3.2. Бинарные операторы

Оператор	Сигнатура	Операция
`+`	`int int -> int`	Сложение/ конкатенация
	`real real -> real`	
	`pt pt -> pt`	
	`str str -> str`	
`-`	`int int -> int`	Вычитание
	`real real -> real`	
	`pt pt -> pt`	
`*`	`int int -> int`	Умножение
	`real real -> real`	
	`real pt -> pt`	
	`pt real -> pt`	
`/`	`int int -> int`	Деление
	`real real -> real`	
	`pt real -> pt`	
`**` `^`	`int int -> int`	Возведение в степень
	`real real -> real`	
	`real int -> real`	
`%`	`int int -> int`	Взятие остатка
	`real real -> real`	
`<`	`int int -> bool`	Меньше
	`real real -> bool`	

Оператор	Сигнатура	Операция
	`str str -> bool`	
`>`	`int int -> bool`	Больше
	`real real -> bool`	
	`str str -> bool`	
`<=`	`int int -> bool`	Меньше или равно
	`real real -> bool`	
	`str str -> bool`	
`>=`	`int int -> bool`	Больше или равно
	`real real -> bool`	
	`str str -> bool`	
`==`	`int int -> bool`	Равно
	`real real -> bool`	
	`str str -> bool`	
`!=`	`int int -> bool`	Не равно
	`real real -> bool`	
	`str str -> bool`	
`|`	`bool bool -> bool`	Булево «ИЛИ»
`&`	`bool bool -> bool`	Булево «И»

Таблица 2. Встроенные бинарные операторы

3.3. Команды

Команда	Аргументы	Комментарий
`get!`	`ident+`	Получить значение переменных
`get_all!`	∅	Получить значение всех переменных
`eval!`	`expr+`	Вычислить значения выражений `expr+`
`set!`	`ident expr`	Установить значение переменной `ident`, равным значению выражения `expr`
`rm!`	`ident+`	Удалить переменные `ident+`
`list_cmd!`	∅	Вывести список всех команд

Команда	Аргументы	Комментарий
`list_func!`	∅	Вывести список всех функций

Таблица 3. Встроенные команды

3.4. Функции

Функция	Сигнатура	Комментарий
`dot`	`pt pt -> real`	Скалярное произведение
`cross`	`pt pt -> real`	Косое произведение
`pt`	`real real -> pt`	Точка по двум координатам
`x`	`pt -> real`	x-координата точки
`y`	`pt -> real`	y-координата точки
`line`	`pt pt -> line`	Прямая по двум точкам
`p1`	`line -> pt`	Первая точка, по которой была построена прямая
`p2`	`line -> pt`	Вторая точка, по которой была построена прямая
`a`	`line -> real`	Коэффициент `a` в общем уравнении прямой: $ax + by + c = 0$
`b`	`line -> real`	Коэффициент `b` в общем уравнении прямой: $ax + by + c = 0$
`c`	`line -> real`	Коэффициент `c` в общем уравнении прямой: $ax + by + c = 0$
`circ`	`pt real -> circ`	Окружность по центру и радиусу
`o`	`circ -> pt`	Центр окружности
`r`	`circ -> real`	Радиус окружности

Таблица 4. Встроенные функции

4. ПРИМЕР ЦЕЛЬНОЙ ПРОГРАММЫ

```
/*
   Эта программа показывает, как использовать встроенный язык программирования
   системы Geometrica и демонстрирует теорему о пересечении высот треугольника в
   одной точке.

   Некоторые функции, аналогичные встроенным, определены вручную, чтобы показать
   возможности Языка.
*/

// -------------------- Объявление вспомогательных функций --------------------

// Возвращает точку пересечения двух прямых, в том случае, если она ровно одна.
// Иначе возвращает `none pt`.
//
// Функция использует представления прямых в виде
// `ax + by + c = 0` и метод Крамера.
inter_point_ l1:line l2:line -> pt = let
        a1 = l1.a,
        b1 = l1.b,
        c1 = l1.c,
        a2 = l2.a,
        b2 = l2.b,
        c2 = l2.c,

        d  = a1 * b2 - b1 * a2,
        dx = -c1 * b2 + b1 * c2,
        dy = a1 * -c2 + c1 * a2,
    in if
        d == 0.0 then none pt,
        else (pt dx dy) / d

// Длина вектора
len_ v:pt -> real = (v.x^2.0 + v.y^2.0)^0.5

// Расстояние между двумя точками
dist_ a:pt b:pt -> real = len_ (a - b)
```

```
// Нормировка вектора
norm_ v:pt -> pt = let
        l = len_ v,
    in if
        l == 0.0 then v,
        else v / l

// Перпендикуляр к прямой через точку
//
// Перпендикуляр построен по двум точкам, расположенным на расстоянии 1.
perp_ l:line p:pt -> line = let
        l_dir = l.p2 - l.p1,
        x = l_dir.y,
        y = -l_dir.x,
        perp_dir = norm_ (pt x y),
    in
        line p (p + perp_dir)

// Перпендикуляр к прямой через точку.
//
// Перпендикуляр построен по двум точкам: данной и лежащей на прямой.
// Если точка p лежит на прямой, то возвращается `perp_ l p`.
// От altitude --- высота (треугольника).
alt_ l:line p:pt -> line = let
        unit_perp = perp_ l p,
        p2 = inter_point_ l unit_perp,
        eps_ = 1e-7,
    in if
        dist_ p2 p < eps_ then unit_perp,
        else line p p2

// -------------------- Демонстрация теоремы --------------------

// Построим треугольник ABC
A = pt 100.0 100.0
B = pt 200.0 400.0
C = pt 400.0 200.0
```

```
AB = line A B
AC = line A C
BC = line B C

// Построим высоты
A_alt = alt_ BC A
B_alt = alt_ AC B
C_alt = alt_ AB C

// Заметим, что все высоты пересеклись в одной точке (Рис. 1).

// Передвинем точки
set! A (pt 100.0 300.0)
set! B (pt 200.0 300.0)
set! C (pt 150.0 100.0)

// Заметим, что все высоты всё ещё пересекаются в одной точке (Рис. 2).
```

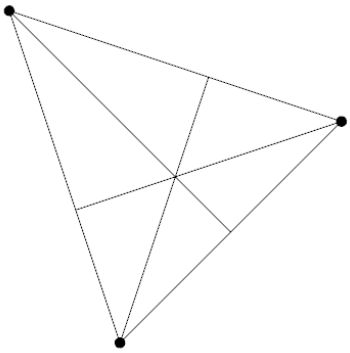

Рис. 1. Снимок экрана

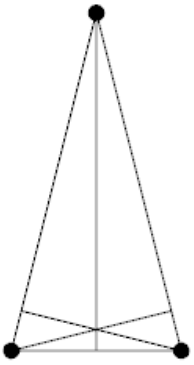

Рис. 2. Снимок экрана