???? Nix

????????????

Идентификаторы в Nix такие же, как в других языках, за исключением того, что позволяют писать дефис (-). Удобно, имея дело с пакетами, писать дефис в имени. Пример:

```
nix-repl> a-b
error: undefined variable `a-b' at (string):1:1
nix-repl> a - b
error: undefined variable `a' at (string):1:1
```

Как видите, `a-b` распознаётся как идентификатор, а не как вычитание.

??????

Строки заключаются в двойные кавычки (") или в пару одинарных кавычек ('').

```
nix-repl> "foo"
"foo"


nix-repl> ''foo''
"foo"
```

В других языках, например, в Python, можно заключать строки в одиночные кавычки `('foo')` , но не в Nix.
Можно интерполировать выражения Nix внутри строк с помощью синтаксиса `${...}`. Если вы писали на других языках, то можете по привычке написать `$foo` или `{$foo}`, но этот синтаксис работать не будет.

```
nix-repl> foo = "strval"
nix-repl> "$foo"
"$foo"
nix-repl> "${foo}"
"strval"
nix-repl> "${2+3}"
error: cannot coerce an integer to a string, at (string):1:2
```

Помните, что присваивание `foo = "strval"` — это специальный синтаксис, доступный только в nix repl и недоступный в обычном языке.
Как я уже говорил, нельзя смешивать целые числа и строки, нужно в явном виде приводить тип. Мы вернёмся к обсуждению этого вопроса позже, как и к вызову функций.
Заключая строку в пару одинарных кавычек, можно писать двойные кавычки внутри без необходимости их экранировать.

```
nix-repl> ''test " test''
"test \" test"
nix-repl> ''${foo}''
"strval"
```

Экранирование `${...}` в строках с двойными кавычками делается с помощью обратной косой линии (бекслеша), а в строках с парой одиночных кавычек — с помощью '':

```
nix-repl> "\${foo}"
"${foo}"
nix-repl> ''test ''${foo} test''
"test ${foo} test"
```

??????

Списки — это последовательность выражений, разделённая пробелами (не запятыми):

```
nix-repl> [ 2 "foo" true (2+3) ]
[ 2 "foo" true 5 ]
```

Списки, как и всё в Nix, неизменяемы (иммутабельны). Добавление или удаление элементов в списке возможно, но возвращает новый список.

??????

Набор атрибутов — это ассоциативный массив со строковыми ключами и значениями Nix. Ключи могут быть только строками. Если ключи являются правильными идентификаторами, их можно записывать без кавычек.

```
nix-repl> s = { foo = "bar"; a-b = "baz"; "123" = "num"; }
nix-repl> s
{ "123" = "num"; a-b = "baz"; foo = "bar"; }
```

Набор атрибутов можно перепутать с набором аргументов при вызове функций, но это разные вещи.

Чтобы обратиться к элементу в наборе атрибутов:

```
nix-repl> s.a-b
"baz"
nix-repl> s."123"
"num"
```

Чтобы обратиться к ключу, который не является правильным идентификатором, используйте кавычки.
Внутри набора нельзя ссылаться на другие элементы или на сам набор:

```
nix-repl> { a = 3; b = a+4; }
error: undefined variable `a' at (string):1:10
```

Это можно делать с помощью рекурсивных наборов:

```
nix-repl> rec { a = 3; b = a+4; }
{ a = 3; b = 7; }
```

Такая возможность полезна при описании пакетов, которые часто имеют рекурсивную природу.

Набор аргументов - набор ключей атрибутов использующийся в качестве аргументов вызова функции.

Одна из самых мощных возможностей Nix — сопоставление с образцом параметра, который имеет тип набор атрибутов. Напишем альтернативную версию `mul = a: b: a*b` сначала используя набор аргументов, а затем — сопоставление с образцом.

```
nix-repl> mul = s: s.a*s.b
nix-repl> mul { a = 3; b = 4; }
12
nix-repl> mul = { a, b }: a*b
nix-repl> mul { a = 3; b = 4; }
12
```

В первом случае мы определили функцию, которая принимает один параметр-набор. Затем мы взяли атрибуты `a` и `b` из этого набора. Заметьте, как элегантно выглядит запись вызова без скобок. В других языках нам пришлось бы написать `mul({ a=3; b=4; })`.

Во втором случае мы определили набор аргументов. Это похоже на определение набора атрибутов, только без значений. Мы требуем, чтобы переданный набор содержал ключи `a` и `b` . Затем мы можем использовать эти `a` и `b` непосредственно в теле функции.

```
nix-repl> mul = { a, b }: a*b
nix-repl> mul { a = 3; b = 4; c = 6; }
error: anonymous function at (string):1:2 called with unexpected argument `c', at (string):1:1
nix-repl> mul { a = 3; }
error: anonymous function at (string):1:2 called without required argument `b', at
(string):1:1
```

Функция принимает набор ровно с теми атрибутами, которые были указаны при её определении.

???????? ?? ????????? ? ??????????? ????????

В наборе аргументов можно указывать **значения атрибутов умолчанию**:

```
nix-repl> mul = { a, b ? 2 }: a*b
nix-repl> mul { a = 3; }
6
nix-repl> mul { a = 3; b = 4; }
12
```

Функция может принимать больше атрибутов, чем ей нужно. Такие атрибуты называются **вариативными**:

```
nix-repl> mul = { a, b, ... }: a*b
nix-repl> mul { a = 3; b = 4; c = 2; }
```

Здесь вы не можете получить доступ к атрибуту `c`. Но вы сможете обратиться к любым атрибутам, дав имя всему набору с помощью **@-образца**:

```
nix-repl> mul = s@{ a, b, ... }: a*b*s.c
nix-repl> mul { a = 3; b = 4; c = 2; }
24
```

Написав `name`@ перед образцом, вы даёте имя `name` всему набору атрибутов.

Преимущества использования наборов аргументов:

- Из-за того, что аргументы именованы, вы не должны запоминать их порядок. В качестве аргументов можно передать набор, что создаёт совершенно новый уровень гибкости и удобства.

Недостатки:

- Частичное применение не работает с набором аргументов. Вы должны определить набор атрибутов целиком, нельзя определить только его часть.

Наборы атрибутов похожи на `**kwargs` из языка Python.

????????? `if`

Это всё ещё выражения, не операторы.

```
nix-repl> a = 3
nix-repl> b = 4
nix-repl> if a > b then "yes" else "no"
"no"
```

> Нельзя записывать только ветку `then` без ветки `else`, потому что у выражения при любом раскладе должен быть результат.

????????? `let`

Выражения `let` используются, чтобы определить локальные переменные для других (внутренних) выражений.

```
nix-repl> let a = "foo"; in a
"foo"
```

Синтаксис такой: сначала определяем переменные, затем пишем ключевое слово `in`, затем выражение, в котором можно ссылаться на определённые переменные. Значением всего выражения `let` будет значение выражения после `in`.

```
nix-repl> let a = 3; b = 4; in a + b
7
```

Попробуем записать два выражения `let`, одно внутри другого:

```
nix-repl> let a = 3; in let b = 4; in a + b
7
```

Помните, что с помощью `let` нельзя присвоить переменной другое значение. Однако, можно перекрывать внешние переменные:

```
nix-repl> let a = 3; a = 8; in a
error: attribute `a' at (string):1:12 already defined at (string):1:5
```

```
nix-repl> let a = 3; in let a = 8; in a
8
```

Нельзя ссылаться на переменные в выражении `let` снаружи:

```
nix-repl> let a = (let c = 3; in c); in c
error: undefined variable `c' at (string):1:31
```

Можно ссылаться на переменные в выражении `let`, определяя другие переменные, как в рекурсивных наборах.

```
nix-repl> let a = 4; b = a + 5; in b
9
```

Общее правило: избегайте ситуаций, когда вам надо сослаться на внешнюю переменную, но переменная с таким же именем есть в текущем выражении `let`. Это же правило действует и в отношении рекурсивных наборов.

????????? `with`

Это непривычный тип выражений — его нечасто можно встретить в других языках. Можно считать его расширенной версией оператора `using` из C++, или `from module import*` из Python. Конструкция `with` включает атрибуты набора в область видимости.

```
nix-repl> longName = { a = 3; b = 4; }
nix-repl> longName.a + longName.b
7
nix-repl> with longName; a + b
7
```

Оператор получает набор атрибутов и включает их в область видимости вложенного выражения. Естественно, в область видимости попадают только корректные идентификаторы. Переменные из внешней области видимости с совпадающими именами не перекрываются. В случае необходимости вы всегда можете обратиться к атрибуту через набор:

```
nix-repl> let a = 10; in with longName; a + b
14
nix-repl> let a = 10; in with longName; longName.a + b
7
```

??????? ???????????

Nix вычисляет выражения только тогда, когда ему нужен результат. Эта особенность языка активно используется при описании пакетов.

```
nix-repl> let a = builtins.div 4 0; b = 6; in b
6
```

Здесь значение a не требуется, поэтому ошибка деления на ноль не возникает — выражение просто не вычисляется. Из-за этой особенности языка, пакеты можно определять по мере необходимости, при этом доступ к ним осуществляется очень быстро.

????????? `inherit`

Выражение `inherit` используется чтобы сопоставить названия атрибутов с их значениями в наборе атрибутов.

Выражение `inherit gcc coretutils;` соответствует набору выражений `gcc = gcc; coreutils = coreutils`, выражение `inherit (pkgs) gcc coreutils;` - `gcc = pkgs.gcc; coreutils = pkgs.coreutils;`

Этот синтаксис имеет смысл только внутри наборов. Это удобный способ избежать повторения одного и того же имени и для атрибута, и для значения в области видимости.

???????? `//`

Оператор `//` принимает на вход два набора. Результатом является их **объединение**. В случае конфликта имён атрибутов, используется значение из правого набора.

```
nix-repl> { a = "b"; } // { c = "d"; }
{ a = "b"; c = "d"; }

nix-repl> { a = "b"; } // { a = "c"; }
{ a = "c"; }
```

Revision #4
Created 2025-11-14 15:29:20 UTC by Антон Сергеевич Абраменко
Updated 2025-11-15 09:31:24 UTC by Антон Сергеевич Абраменко