Nix

Чистый функциональный пакетный менеджер и система развёртывания для POSIX-совместимых ОС.

??????????

lib.mapAttrsToList

???????????, ???????? ? ????????????

`lib.mapAttrsToList` — это функция, которая **преобразует набор атрибутов в список, применяя функцию к каждой паре ключ-значение.**

```
lib.mapAttrsToList f attrs
```

Описание:

- `f` — функция, принимающая два аргумента: `name` (ключ) и `value` (значение);
- `attrs` — набор атрибутов.

Возвращает **список** результатов применения `f` к каждому элементу.

Использование:

- Генерация конфигурационных файлов.
- Создания списков для динамической конфигурации.
- Преобразования данных между форматами.

???????

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

Исходный код:

```
{ lib }:
let
  users = {
    alice = 25;
    bob = 30;
    charlie = 35;
  };

  result = lib.mapAttrsToList (name: age: "${name} is ${toString age} years old") users;
in
```

```
result
```

Результат:

```
[ "alice is 25 years old" "bob is 30 years old" "charlie is 35 years old" ]
```

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

```
{ lib, ... }:
let
  userConfigs = {
    alice = {
      uid = 1001;
      group = "users";
      shell = "/bin/bash";
    };
    bob = {
      uid = 1002;
      group = "developers";
      shell = "/bin/zsh";
    };
  };

  # Преобразуем в формат, понятный для users.users
  usersList = lib.mapAttrsToList (username: config: {
    name = username;
    inherit (config) uid group shell;
    isNormalUser = true;
  }) userConfigs;
in
{
  users.users = builtins.listToAttrs (map (u: { name = u.name; value = u; }) usersList);
}
```

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

```
{ lib, ... }:
let
  services = {
    nginx = {
      port = 80;
```

```
      enable = true;
    };
    postgres = {
      port = 5432;
      enable = true;
    };
    redis = {
      port = 6379;
      enable = false;
    };
  };

  # Фильтруем и создаем конфиги только для включенных сервисов
  enabledServices = lib.mapAttrsToList (name: config:
    lib.optional config.enable {
      serviceName = name;
      inherit (config) port;
      configFile = ./${name}-config.conf;
    }
  ) services;
in
{
  # enabledServices будет списком списков, flatten его
  services = lib.flatten enabledServices;
}
```

????????? firewall

```
{ lib, ... }:
let
  openPorts = {
    http = 80;
    https = 443;
    ssh = 22;
    smtp = 25;
  };

  firewallRules = lib.mapAttrsToList (name: port: {
    # Создаем правило для каждого порта
    rule = "-p tcp --dport ${toString port} -j ACCEPT";
```

```
    description = "Allow ${name} (port ${toString port})";
  }) openPorts;
in
{
  networking.firewall.extraCommands = lib.concatMapStringsSep "\n" (rule:
    "iptables -A INPUT ${rule.rule} # ${rule.description}"
  ) firewallRules;
}
```

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

```
{ lib }:
let
  data = { a = 1; b = 2; c = 3; };
in
{
  # mapAttrsToList: возвращает список
  mapAttrsToList = lib.mapAttrsToList (n: v: "${n}=${toString v}") data;
  # Результат: [ "a=1" "b=2" "c=3" ]

  # mapAttrs: возвращает набор
  mapAttrs = lib.mapAttrs (n: v: v * 2) data;
  # Результат: { a = 2; b = 4; c = 6; }

  # attrValues: только значения
  attrValues = lib.attrValues data;
  # Результат: [ 1 2 3 ]
}
```

????????? systemd ?????

```
{ lib, ... }:
let
  backupJobs = {
    "backup-home" = {
      source = "/home";
      destination = "/backup/home";
      schedule = "daily";
    };
    "backup-etc" = {
```

```
      source = "/etc";
      destination = "/backup/etc";
      schedule = "weekly";
    };
  };

  systemdServices = lib.mapAttrsToList (jobName: config:
    {
      name = "backup-${jobName}";
      value = {
        description = "Backup ${config.source}";
        script = ''
          rsync -av ${config.source} ${config.destination}
        '';
        startAt = config.schedule;
      };
    }
  ) backupJobs;
in
{
  systemd.services = builtins.listToAttrs systemdServices;
}
```

Библиотеки

lib.mkMerge

???????????, ???????? ? ????????????

`lib.mkMerge` — это функция, которая позволяет **объединять несколько наборов атрибутов в один**, решая конфликты через **приоритеты**.
Это особенно полезно в модульной конфигурации NixOS, когда вы хотите разделить конфигурацию на части и объединить их без ручного разрешения конфликтов.
Когда вы объединяете наборы атрибутов в Nix, могут возникать конфликты (одинаковые ключи с разными значениями). `lib.mkMerge` решает эти конфликты, используя систему приоритетов: каждый элемент получает приоритет (по умолчанию 0), и выигрывает значение с наивысшим приоритетом.

???????

????????

Исходный код

```
{ lib, ... }:

let
  # Три конфигурации, которые мы хотим объединить
  config1 = { boot.loader.grub.enable = true; };
  config2 = { boot.loader.systemd-boot.enable = true; };
  config3 = { networking.hostName = "myhost"; };
in
lib.mkMerge [
  config1
  config2  # Конфликт с config1 по boot.loader.*
  config3
]
```

Описание

В этом примере `config1` и `config2` конфликтуют (оба определяют загрузчик). По умолчанию `lib.mkMerge` просто выберет последнее значение, но с приоритетами можно контролировать результат.

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

```
{ lib, ... }:

lib.mkMerge [
  # Приоритет 1000 (высокий)
  (lib.mkIf false {
    services.nginx.enable = true;
  })

  # Приоритет 100 (средний)
  {
    services.nginx.enable = lib.mkDefault false;
    services.nginx.virtualHosts."example.com".root = "/var/www";
  }

  # Приоритет 150 (выше среднего)
  (lib.mkForce {
    services.nginx.enable = true;  # Переопределит mkDefault
  })
]
```

Описание:

- `mkIf` имеет приоритет 1000
- `mkDefault` имеет приоритет 100
- `mkForce` имеет приоритет 150
- Результат: `services.nginx.enable = true` (побеждает `mkForce`)

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

```
{ lib, config, ... }:

let
  commonNetwork = {
    networking.networkmanager.enable = true;
    networking.firewall.enable = true;
  };

  homeConfig = {
    networking.hostName = "home-pc";
```

```
    networking.firewall.allowedTCPPorts = [ 80 443 ];
  };

  workConfig = {
    networking.hostName = "work-laptop";
    networking.firewall.allowedTCPPorts = [ 22 3389 ];
    networking.proxy.default = "http://proxy.company.com:8080";
  };

  # Динамически выбираем конфигурацию
  environmentConfig = if config.isWorkEnvironment then workConfig else homeConfig;
in
{
  imports = [ ./hardware-configuration.nix ];

  options.isWorkEnvironment = lib.mkOption {
    type = lib.types.bool;
    default = false;
  };

  config = lib.mkMerge [
    commonNetwork
    environmentConfig
    {
      # Этот блок имеет самый высокий приоритет
      networking.nameservers = lib.mkForce [ "1.1.1.1" "8.8.8.8" ];
    }
  ];
}
```

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

```
# hardware/base.nix
{ lib, ... }:

{
  options.hardware = {
    profile = lib.mkOption {
      type = lib.types.enum [ "desktop" "laptop" "server" ];
      default = "desktop";
```

```
    };

    hasBluetooth = lib.mkOption {
      type = lib.types.bool;
      default = false;
    };
  };

  config = lib.mkMerge [
    # Базовая конфигурация для всех систем
    {
      hardware.enableRedistributableFirmware = true;
      powerManagement.enable = true;
    }

    # Конфигурация для ноутбуков
    (lib.mkIf (config.hardware.profile == "laptop") {
      services.tlp.enable = true;
      services.auto-cpufreq.enable = true;
      hardware.hasBluetooth = lib.mkDefault true;
    })

    # Конфигурация для серверов
    (lib.mkIf (config.hardware.profile == "server") {
      powerManagement.enable = lib.mkForce false;  # Отключаем на серверах
      services.openssh.enable = true;
    })

    # Конфигурация Bluetooth
    (lib.mkIf config.hardware.hasBluetooth {
      hardware.bluetooth.enable = true;
      services.blueman.enable = true;
    })
  ];
}
```

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

```
{ lib, ... }:

lib.mkMerge [
  {
    services.postgresql = lib.mkMerge [
      {
        enable = true;
        package = pkgs.postgresql_15;
      }
      (lib.mkIf config.services.grafana.enable {
        authentication = ''
          host grafana all ::1/128 md5
        '';
      })
    ];
  }

  {
    environment.systemPackages = with pkgs; [
      vim
      htop
    ];
  }
]
```

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

Приоритеты по умолчанию (от высокого к низкому):

1. `lib.mkIf` (1000) — условное включение
2. `lib.mkOverride` — явное указание приоритета
3. `lib.mkForce` (50) — принудительное значение
4. `lib.mkDefault` (1000 для false, 100 для true) — значения по умолчанию
5. Обычные значения (0) — стандартный приоритет

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

1. **Порядок важен только при равных приоритетах** — при одинаковых приоритетах побеждает последнее значение

2. **Глубокое слияние** — `mkMerge` рекурсивно объединяет вложенные атрибуты
3. **Не путать с `mkOverride`** — `mkMerge` объединяет списки наборов, а `mkOverride` изменяет приоритет конкретного атрибута

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

```
# Объединение конфигураций
config = lib.mkMerge [
  commonConfig
  (lib.mkIf condition conditionalConfig)
  (lib.mkForce forcedConfig)
  userConfig
];

# Эквивалентно (но более читаемо и модульно):
# {
#   commonConfig
#   conditionalConfig (если condition == true)
#   forcedConfig (с приоритетом)
#   userConfig
# }
```

lib.mkIf

???????????, ???????? ? ????????????

`lib.mkIf` — это функция, используемая для **условного определения** конфигурационных опций в модулях.

```
lib.mapAttrsToList condition definition
```

Описание:

- `condition` — логическое выражение (true/false);
- `deffinition` — значение или набор атрибутов, которые будут применены, если условие истинно.

Возвращает `definition`, если `condition = true`, иначе возвращает особое значение "пустоты".

Использование:

- Позволяет включать и выключать части конфигурации на основе условий.

???????

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

Исходный код:

```
{ config, lib, ... }:
{
  config = lib.mkIf (config.networking.hostName == "webserver") {
    services.nginx.enable = true;
    services.nginx.virtualHosts."example.com".root = "/var/www";
  };
}
```

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

```
{ config, lib, ... }:
{
  config = lib.mkIf config.services.xserver.enable {
    sound.enable = true;

    hardware.pulseaudio = lib.mkIf config.services.pipewire.enable {
      enable = false;
    };
  };
}
```

?????????????? ? `mkMerge`

```
{ config, lib, ... }:
{
  config = lib.mkMerge [
    # Общие настройки
    {
      environment.systemPackages = with pkgs; [ vim wget ];
    }

    # Условные настройки для сервера
    (lib.mkIf (config.networking.hostName == "server") {
      services.openssh.enable = true;
      services.nginx.enable = true;
    })

    # Условные настройки для рабочей станции
    (lib.mkIf config.services.xserver.enable {
      services.xserver.desktopManager.gnome.enable = true;
      hardware.bluetooth.enable = true;
    })
  ];
}
```

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

```
{ config, lib, ... }:
{
  imports = [
```

```
    (lib.mkIf config.virtualisation.docker.enable ./docker-extra.nix)
    (lib.mkIf config.services.mysql.enable ./mysql-backup.nix)
  ];
}
```

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

```
{ config, lib, ... }:
let
  isProduction = config.networking.hostName == "prod-server";
  hasGPU = config.hardware.opengl.enable;
in
{
  config = lib.mkIf (isProduction && hasGPU) {
    services.tensorflow-serving.enable = true;
    services.cuda.enable = true;
  };
}
```

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

```
{ config, lib, ... }:
{
  options.myService = {
    enable = lib.mkEnableOption "My custom service";
    extraConfig = lib.mkOption {
      type = lib.types.str;
      default = "";
    };
  };

  config = lib.mkIf config.myService.enable {
    systemd.services.my-service = {
      description = "My Service";
      wantedBy = [ "multi-user.target" ];
      script = ''
        echo "Starting my service"
        ${lib.optionalString (config.myService.extraConfig != "") ''
          echo "Extra config: ${config.myService.extraConfig}"
        ''}
```

```
      '';
    };
  };
}
```

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

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

```
# Правильно:
config = lib.mkIf condition1 {
  services.xyz = lib.mkIf condition2 {
    enable = true;
  };
};

# Неправильно (может вызвать ошибки):
config = {
  services.xyz = lib.mkIf condition1 {
    enable = lib.mkIf condition2 true;
  };
};
```

?????? ? `mkOverride`

```
{ config, lib, ... }:
{
  config = lib.mkIf config.services.foo.enable {
    services.foo.configFile = lib.mkOverride 90 "/etc/foo/custom.conf";
    # Приоритет 90 (меньше = выше приоритет)
  };
}
```

????????????? ? `mkDefault` ? `mkForce`

```
{ config, lib, ... }:
{
  config = lib.mkIf config.networking.wireless.enable {
    networking.networkmanager.wifi.backend = lib.mkDefault "iwd";
    # Установит значение только если оно не было задано явно
```

```
  };
}
```

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

```
{ config, lib, pkgs, ... }:
let
  roles = {
    web = config.node.role.web or false;
    db = config.node.role.db or false;
    cache = config.node.role.cache or false;
  };
in
{
  options.node.role = {
    web = lib.mkEnableOption "Web server role";
    db = lib.mkEnableOption "Database server role";
    cache = lib.mkEnableOption "Cache server role";
  };

  config = lib.mkMerge [
    # Базовая конфигурация для всех узлов
    {
      environment.systemPackages = with pkgs; [ htop tmux ];
      services.openssh.enable = true;
    }

    # Конфигурация для веб-сервера
    (lib.mkIf roles.web {
      services.nginx.enable = true;
      services.phpfpm.enable = true;
      networking.firewall.allowedTCPPorts = [ 80 443 ];
    })

    # Конфигурация для БД сервера
    (lib.mkIf roles.db {
      services.postgresql.enable = true;
      services.postgresql.ensureDatabases = [ "appdb" ];
      networking.firewall.allowedTCPPorts = [ 5432 ];
    })
```

```
    # Конфигурация для кэш-сервера
    (lib.mkIf roles.cache {
      services.redis.enable = true;
      networking.firewall.allowedTCPPorts = [ 6379 ];
    })
  ];
}
```

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

1. **Ленивые вычисления**: `lib.mkIf` использует ленивые вычисления, поэтому определение вычисляется только если условие истинно.
2. **Обработка ошибок**: Условие должно быть полностью определено. Нельзя использовать значения, которые могут быть `undefined`.
3. **Композиция**: `mkIf` хорошо сочетается с другими функциями библиотеки: `mkMerge`, `mkOption`, `mkDefault`.
4. **Читаемость**: Для сложных условий рекомендуется использовать `let`-блоки для присвоения имен условиям.
5. **Отладка**: Если нужно увидеть, какие условия срабатывают, можно использовать `lib.traceIf` для отладки.

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

```
{ config, lib, ... }:
{
  config = lib.mkIf (lib.traceValFn (v: "Condition value: ${toString v}")
    (config.services.nginx.enable)) {
    # конфигурация
  };
}
```

Библиотеки

lib.genAttrs

`lib.genAttrs` — функция, которая принимает список имён и функцию-генератор, создавая набор атрибутов, где каждый элемент списка становится ключом, а значение вычисляется функцией-генератором на основе этого ключа.
Инструмент для избежания повторения в конфигурациях NixOS, особенно когда нужно применить похожую конфигурацию к множеству сущностей(службы, пользователи, способы взаимодействия(interfaces))

?????????

```
genAttrs :: [String] -> (String -> Any) -> AttrSet
```

- первый аргумент — список строк (имена будущих атрибутов);
- второй аргумент — функция, принимающая имя и возвращающая значение для этого атрибута;

Возвращает набор атрибутов `{ name1 = value1; name2 = value2; ... }`

???????

????????

```
{ lib }:
let
  names = [ "foo" "bar" "baz" ];
  # Функция-генератор: добавляет "-suffix" к имени
  addSuffix = name: "${name}-suffix";

  result = lib.genAttrs names addSuffix;
in
result
```

Результат:

```
{
  foo = "foo-suffix";
  bar = "bar-suffix";
  baz = "baz-suffix";
```

```
}
```

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

```
{ config, lib, pkgs, ... }:
let
  myServices = [ "nginx" "postgresql" "redis" ];

  # Создаём атрибутный набор включённых сервисов
  enabledServices = lib.genAttrs myServices (name: {
    enable = true;
  });
in
{
  # Включаем все сервисы одним выражением
  services = enabledServices;

  # Эквивалентно:
  # services.nginx.enable = true;
  # services.postgresql.enable = true;
  # services.redis.enable = true;
}
```

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

```
{ config, lib, ... }:
let
  users = [ "alice" "bob" "charlie" ];

  # Создаём базовую конфигурацию для каждого пользователя
  userConfigs = lib.genAttrs users (name: {
    isNormalUser = true;
    extraGroups = [ "wheel" "networkmanager" ];
    createHome = true;
    home = "/home/${name}";
  });
in
{
  users.users = userConfigs;
}
```

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

```
{ pkgs, lib, ... }:
let
  languages = [ "python3" "go" "nodejs" "rustc" ];

  # Устанавливаем последние версии языков программирования
  devPackages = lib.genAttrs languages (name: pkgs.${name});
in
{
  environment.systemPackages = builtins.attrValues devPackages;

  # Или с дополнительной конфигурацией:
  languages = lib.genAttrs languages (name: {
    enable = true;
    package = pkgs.${name};
  });
}
```

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

```
{ config, lib, ... }:
let
  domains = [ "example.com" "test.com" "admin.example.com" ];

  nginxVhosts = lib.genAttrs domains (domain: {
    serverName = domain;
    root = "/var/www/${domain}";
    locations."/".proxyPass = "http://localhost:8080";
    enableSSL = true;
    sslCertificate = "/var/ssl/${domain}.crt";
    sslCertificateKey = "/var/ssl/${domain}.key";
  });
in
{
  services.nginx.virtualHosts = nginxVhosts;
}
```

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

```
{ lib, ... }:
let
  interfaces = [ "eth0" "eth1" "wlan0" ];

  networkConfig = lib.genAttrs interfaces (name:
    if lib.hasPrefix "eth" name then {
      useDHCP = false;
      ipv4.addresses = [{
        address = "192.168.1.${toString (10 + lib.elemIndex name interfaces)}";
        prefixLength = 24;
      }];
    } else {
      useDHCP = true;
      wireless.enable = true;
    }
  );
in
{
  networking.interfaces = networkConfig;
}
```

??????? ? `listToAttrs`

Часто путают `genAttrs` с `listToAttrs`. Вот ключевое отличие:

```
# genAttrs - проще, когда нужно создать значения на основе имён
lib.genAttrs [ "a" "b" ] (name: "value-${name}")
# => { a = "value-a"; b = "value-b"; }

# listToAttrs - когда у вас уже есть пары {name, value}
lib.listToAttrs [
  { name = "a"; value = 1; }
  { name = "b"; value = 2; }
]
# => { a = 1; b = 2; }
```

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

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

```
lib.genAttrs (lib.filter (s: lib.hasPrefix "dev" s) allNames) (name: ...)
```

????????????? ? ??????? NixOS

```
options = lib.genAttrs [ "foo" "bar" ] (name: lib.mkOption {
  type = lib.types.str;
  default = name;
});
```

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

```
let
  keys = builtins.attrNames someInputSet;
  processed = lib.genAttrs keys (key: transformFunction someInputSet.${key});
in
processed
```

???? 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"; }
```

Nix. ??? ??? ? ? ??? ??? ???????????

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

Помимо NixOS, где ничего делать не нужно, Nix можно установить на любой (или почти любой) дистрибутив Linux. Для этого достаточно запустить следующую команду:

```
$ sh <(curl -L https://nixos.org/nix/install)
```

Дальше установочный скрипт сам всё сделает. После недавних изменений в MacOS, установка в ней немного осложнилась, раньше было достаточно команды выше. Про установку в последних версиях MacOS можно прочесть здесь.

???? Nix

Когда речь идёт о Nix, часто имеют в виду две разные сущности: Nix как язык и nixpkgs как репозиторий пакетов, в том числе составляющий основу NixOS. Начнём с первого.

Nix — функциональный ленивый язык с динамической типизацией. Синтаксис во многом похож на языки семейства ML (SML, OCaml, Haskell), поэтому у тех, кто с ними знаком, особых проблем возникнуть не должно.

Начать знакомство с языком можно просто запустив интерпретатор.

```
$ nix repl
Welcome to Nix version 2.3.10. Type :? for help.


nix-repl>
```

Отдельного синтаксиса для объявления функций в Nix нет. Функции задаются через присваивание, так же как и другие значения.

```
nix-repl> "Hello " + "World!"
"Hello World!"
```

```
nix-repl> add = a: b: a + b

nix-repl> add 1 2
3
```

Как и в языках, повлиявших на Nix, все функции каррированы.

```
nix-repl> addOne = add 1

nix-repl> addOne 3
4
```

Помимо примитивных типов, таких как числа и строки, Nix поддерживает списки и словари (attribute sets в терминологии Nix).

```
nix-repl> list = [ 1 2 3 ]

nix-repl> set = { a = 1; b = list; }

nix-repl> set
{ a = 1; b = [ ... ]; }

nix-repl> set.b
[ 1 2 3 ]
```

Значения в локальной области видимости можно задать через выражение `let...in`. Для примера, простая функция, реализующая факториал, как это принято делать в других статьях по функциональному программированию.

`fac.nix`:

```
let
  fac = n:
    if n == 0
    then 1
    else n * fac (n - 1);
in { inherit fac; }
```

Директива `inherit` вносит или "наследует" термин из текущей области видимости и даёт ему такое же имя. Пример выше эквивалентен записи `let fac = ... in { fac = fac; }`.

```
$ nix repl fac.nix
Welcome to Nix version 2.3.10. Type :? for help.

Loading 'fac.nix'...
Added 1 variables.

nix-repl> fac 3
6
```

При загрузке файлов или модулей в REPL, Nix ожидает, что результатом вычисления модуля будет множество, элементы которого будут импортированы в текущую область видимости.

Для загрузки кода из других файлов в Nix есть функция `import`, принимающая путь к файлу с кодом и возвращающая результат выполнения этого кода.

`mul.nix`:

```
let
  mul = a: b: a * b;
in { inherit mul; }
```

Новый `fac.nix`:

```
let
  multMod = import ./mul.nix;
  fac = n:
    if n == 0
    then 1
    else multMod.mul n (fac (n - 1));
in { inherit fac; }
```

Хотя присваивание модуля в отдельную переменную — довольно частая практика, в данном случае это выглядит несколько нелепо, правда? В Nix есть директива `with`, добавляющая в текущую область видимости все имена из множества, переданного в качестве параметра.

`fac.nix` с использованием `with`:

```
with import ./mul.nix;
let
  fac = n:
    if n == 0
```

```
    then 1
    else mul n (fac (n - 1));
in { inherit fac; }
```

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

Сборка программ и отдельных компонентов — это основная функция языка Nix.

В случае работы с пакетами, основным инструментом, про который нужно знать, является `Derivation`. Сам по себе `Derivation` — это специальный файл, содержащий рецепт для сборки в машинно-читаемом виде. Для компиляции программы на C, выводящей "Hello World!", derivation выглядит примерно следующим образом:

```
Derive([("out","/nix/store/1nq46fyv3629slgxnagqn2c01skp7xrq-hello-
world","","")],[("/nix/store/60xqp516mkfhf31n6ycyvxppcknb2dwr-build-
hello.drv",["out"])],["/nix/store/wiviq2xyz0ylhl0qcgfgl9221nkvvxfj-hello.c"],"x86_64-
linux","/nix/store/r5lh8zg768swlm9hxxfrf9j8gwyadi72-build-
hello",[],[("builder","/nix/store/r5lh8zg768swlm9hxxfrf9j8gwyadi72-build-
hello"),("name","hello-world"),("out","/nix/store/1nq46fyv3629slgxnagqn2c01skp7xrq-hello-
world"),("src","/nix/store/wiviq2xyz0ylhl0qcgfgl9221nkvvxfj-hello.c"),("system","x86_64-
linux")])
```

Как видно, в этом выражении содержится путь к результату сборки, который получится в итоге, а также пути к исходным файлам, скрипту сборки, и метаданные: имя проекта и платформа. Стоит так же заметить, что пути к исходникам начинаются с `/nix/store`. При сборке, Nix копирует всё нужное в эту директорию, после чего сборка происходит в изолированном окружении (sandbox). Таким образом достигается воспроизводимость сборки всех пакетов.

Разумеется, никто в здравом уме руками писать такое не станет! Для простых случаев, в Nix есть встроенная функция `derivation`, принимающая описание сборки.

`simple-derivation/default.nix`:

```
{ pkgs ? import <nixpkgs> {} }:

derivation {
  name = "hello-world";
  builder = pkgs.writeShellScript "build-hello" ''
    ${pkgs.coreutils}/bin/mkdir -p $out/bin
    ${pkgs.gcc}/bin/gcc $src -o $out/bin/hello -O2
```

```
  '';
  src = ./hello.c;
  system = builtins.currentSystem;
}
```

Давайте попробуем разобрать этот пример. Весь файл представляет собой определение функции, которая берёт один параметр — словарь, содержащий поле `pkgs`. Если оно не было передано при вызове этой функции, используется значение по умолчанию: `import <nixpkgs> {}`.

`derivation` — функция, так же принимающая словарь с параметрами сборки: `name` — имя пакета, `builder` — сборочный скрипт, `src` — исходный код, `system` — система или список систем, под который возможна сборка данного пакета.

`writeShellScript` — функция из `nixpkgs`, принимающая имя для скрипта и код и возвращающая путь к исполняемому файлу. Для многострочного текста в Nix есть альтернативный синтаксис с двумя парами одинарных кавычек.

С помощью команды `nix build`, этот рецепт для сборки можно запустить и получить работающий бинарник.

```
$ nix build -f ./simple-derivation/default.nix
[1 built]

$ ./result/bin/hello
Hello World!
```

При запуске `nix build`, в текущей директории создаётся символическая ссылка `result`, указывающая на созданный в `/nix/store` пакет.

```
$ ls -l result
lrwxrwxrwx 1 user users 50 Mar 29 17:53 result -> /nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple

$ find /nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple
/nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple
/nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple/bin
/nix/store/vpcddray35g2jrv40dg1809xrmz73awi-simple/bin/hello
```

?????? ????????, ??????????? ??????

`derivation` — достаточно низкоуровневая функция, на базе которой в Nix построены куда более мощные примитивы. Для примера, можно рассмотреть сборку широко известной утилиты `cowsay`.

```
{ lib, stdenv, fetchurl, perl }:

stdenv.mkDerivation rec {
  version = "3.03+dfsg2";
  pname = "cowsay";

  src = fetchurl {
    url = "http://http.debian.net/debian/pool/main/c/cowsay/cowsay_${version}.orig.tar.gz";
    sha256 = "0ghqnkp8njc3wyqx4mlg0qv0v0pc996x2nbyhqhz66bbgmf9d29v";
  };

  buildInputs = [ perl ];

  postBuild = ''
    substituteInPlace cowsay --replace "%BANGPERL%" "!${perl}/bin/perl" \
      --replace "%PREFIX%" "$out"
  '';

  installPhase = ''
    mkdir -p $out/{bin,man/man1,share/cows}
    install -m755 cowsay $out/bin/cowsay
    ln -s cowsay $out/bin/cowthink
    install -m644 cowsay.1 $out/man/man1/cowsay.1
    ln -s cowsay.1 $out/man/man1/cowthink.1
    install -m644 cows/* -t $out/share/cows/
  '';

  meta = with lib; {
    description = "A program which generates ASCII pictures of a cow with a message";
    homepage = "https://en.wikipedia.org/wiki/Cowsay";
    license = licenses.gpl1;
    platforms = platforms.all;
```

```
    maintainers = [ maintainers.rob ];
  };
}
```

Оригинал скрипта находится здесь.

`stdenv` — специальный `derivation`, содержащий правила сборки для текущей системы: нужный компилятор, флаги и прочие параметры. Основное содержимое — гигантских размеров скрипт на баше под названием `setup`, который и выступает в роле скрипта `builder` из нашего простого примера выше.

```
 $ nix build nixpkgs.stdenv

 $ find result/
result/
result/setup
result/nix-support

$ wc -l result/setup
1330 result/setup
```

`mkDerivation` — функция, создающая `derivation` с этим скриптом и заодно заполняющая другие поля.

Те читатели, кто раньше писал скрипты для сборки пакетов в Arch Linux или Gentoo, могут увидеть здесь крайне знакомую структуру. Как и в других дистрибутивах, сборка разбита на фазы, присутствует перечисление зависимостей (`buildInputs`) и так далее.

??????????

Каррирование (от англ. currying, иногда — карринг) — преобразование функции от многих аргументов в набор вложенных функций, каждая из которых является функцией от одного аргумента.

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

??????

> **Модуль** — это блок, описывающий настройку одной из частей системы, используя опции как способ взаимодействия с другими блоками, настройкой системы в целом или отдельными её частями.

Объявление этих опций является его основным отличием от других блоков с настройками. Модули позволяют разбивать настройку системы на части, делая ее более структурированной, повторно используемой и масштабируемой.

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

1. **Абстракция.** Модули скрывают детали реализации, предоставляя стандартизированный способ взаимодействия для настройки компонентов системы. В этом смысле он больше похож на описание класса в языке программирования.
2. **Повторное использование.** Один и тот же модуль может использоваться в разных частях системы или в разных конфигурациях для однотипных настроек.
3. **Использование других модулей**. Модули могут импортировать другие модули, позволяя создавать иерархическую структуру конфигурации.

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

1. Импорт других модулей(**imports**).
2. Объявление опций(**options**).
3. Определение опций и описание настроек модуля(**config**).

```
{
  imports = [
    # Пути к другим модулям.
    # Модули могут импортировать другие модули, позволяя
    # создавать иерархическую структуру настроек.
  ];

  options = {
    # Объявление опций.
```

```
    # Объявляет какие настройки пользователь этого модуля может устанавливать.
    # Обычно это включает общий пункт "enable" изначально установленный в ложное значение.
  };

  config = {
    # Определение опций.
    # Определяет какие другие настройки, службы и ресурсы должны быть задействованы.
    # Обычно это зависит от того выбрал ли пользователь этого модуля
    # пункт "enable" используя объявление выше.
    # Опции для модулей импортированных в секции "imports" могут быть установлены здесь.
  };
}
```

Каждый модуль может объявлять новые опции, которые являются настройками для других частей системы. Например, модуль для настройки веб-сервера может иметь опции порта и директории сайта.

Все модули получают доступ к переменной `config`, которая содержит текущие определения опций из других модулей, что позволяет модулям взаимодействовать друг с другом и использовать ранее определенные настройки.

Поскольку модули описывают желаемое состояние системы, а не шаги по его достижению, NixOS является декларативной системой, где настройка системы осуществляется средствами модулей.

Пример:
Вот как может выглядеть упрощенный модуль для гипотетической службы `my-service`:

```
# ./my-module.nix
{ config, lib, pkgs, ... }: # Это функция, принимающая аргументы

{
  # 1. Объявляем опции, которые можно использовать в configuration.nix
  options = {
    services.my-service = {
      enable = lib.mkEnableOption "My cool service"; # Создает опцию enable с описанием
      port = lib.mkOption {
        type = lib.types.port;
        default = 8080;
        description = "Port for my-service";
      };
    };
```

```
  };

  # 2. Определяем, что делать, если служба включена
  config = lib.mkIf config.services.my-service.enable {
    # Это "вклад" модуля в общую конфигурацию системы
    environment.systemPackages = [ pkgs.my-service-pkg ]; # Добавляем пакет в окружение
    systemd.services.my-service = { # Создаем systemd юнит
      description = "My Service";
      after = [ "network.target" ];
      wantedBy = [ "multi-user.target" ];
      serviceConfig = {
        ExecStart = "${pkgs.my-service-pkg}/bin/my-service --port ${toString
config.services.my-service.port}";
      };
    };
  };
}
```

Теперь в `configuration.nix` мы можем **импортировать** этот модуль и использовать его опции:

```
# /etc/nixos/configuration.nix
{ config, pkgs, ... }:

{
  imports = [ ./my-module.nix ]; # Импортируем модуль

  # Используем опции, объявленные в модуле
  services.my-service = {
    enable = true;
    port = 9000;
  };

  # ... остальная конфигурация системы
}
```

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

Модули могут использоваться для:

1. **Управления службами**. Модуль может описывать настройку и запуск службы, например, веб-сервера nginx, включая его зависимости.
2. **Настройки пользователя.** NixOS home-manager использует модули для управления настройками пользователя и содержимым его домашнего каталога, например, настройками оболочки или приложения.
3. **Описание пакетов.** Модули могут описывать сборку и установку пакетов, а также управлять их зависимостями.

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

1. **Воспроизводимость.** Модули упрощают создание воспроизводимых настроек, которые можно легко применить на разных машинах.
2. **Управление.** Разбиение настроек на модули делает ее более управляемой и понятной, даже для сложных систем.
3. **Повторное использование.** Модули могут быть повторно использованы в разных настройках, что сокращает дублирование кода.

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

> **Вложенные модули** — это тип, позволяющий определять вложенные модули с собственным набором опций. Он используется для структурирования сложных настроек, группируя связанные опции в отдельный блок.

При использовании `lib.types.submodule` вы создаёте новую опцию, значением которого будет набор атрибутов, соответствующих другому, вложенному модулю. Этот вложенный модуль имеет свой собственный набор опций, как и обычный модуль NixOS.

Пример:
Допустим, вы создаёте модуль для веб-сервиса, и хотите, чтобы опции для базы данных были сгруппированы.

1. **Объявление типа `submodule`:**
   Вы определяете опцию `database` с типом `submodule`. Внутри `submodule` вы задаёте собственные опции, например, `host`, `port` и `user`.

```
# module.nix
{ config, lib, ... }:

let
  types = lib.types;
in
```

```
{
  options.database = lib.mkOption {
    type = types.submodule {
      options = {
        host = lib.mkOption {
          type = types.str;
          default = "localhost";
        };
        port = lib.mkOption {
          type = types.int;
          default = 5432;
        };
        user = lib.mkOption {
          type = types.str;
          default = "admin";
        };
      };
    };
    description = "Настройки для базы данных.";
  };
}
```

2. **Использование в настройке:**
   Затем в вашей настройке вы можете определить значения для этих вложенных опций.

```
# configuration.nix
{ ... }:

{
  imports = [
    ./module.nix
  ];

  database = {
    host = "db.example.com";
    port = 5433;
    user = "web_user";
  };
}
```

В результате вы получите настройки, где `config.database.host` будет `"db.example.com"`, `config.database.port` будет `5433` и `config.database.user` будет `"web_user"`.

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

- **Модульность:** Позволяет разбивать сложные настройки на логические, повторно используемые блоки.
- **Структурирование:** Повышает читаемость и удобство управления настройками.
- **Типобезопасность:** Каждая опция внутри `submodule` имеет свой тип и проверяется, что предотвращает ошибки настройки.
- **Совместимость с другими типами:** Часто используется в сочетании с `lib.types.attrsOf` или `lib.types.listOf` для создания списков или наборов подмодулей. Например, можно создать `listOf submodules` для определения нескольких веб-сервисов с похожими настройками.

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

> **Конфигурация** – это блок, описывающий настройку системы в целом или отдельной её части.

В NixOS файл конфигурации `/etc/nixos/configuration.nix` предназначен для описания всех настроек системы. Он может включать другие файлы конфигурации, описывающие только часть настроек общей системы, а также модули, изменять настройки которых можно через их опции. **Конфигурация** описывает **заданное**, **статическое состояние** системы, в то время как **модули могут изменять свое поведение в зависимости от значения, передаваемое их опциям**. Например, вы можете указать, какие сайты разместить на nginx, используя для этого модуль.

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

Разница между модулем и конфигурацией в NixOS заключается в их функциях и способе использования. Модуль представляет собой часть кода Nix, которую можно настроить для создания конфигурации . Конфигурация, в свою очередь, является результатом работы модулей и определяет параметры всей системы .

Основные моменты, поясняющие разницу:

1. **Модули**:

- **Функция**: Модуль — это функция, которая принимает набор атрибутов и возвращает другой набор атрибутов .
- **Назначение**: Модули позволяют организовывать конфигурацию системы, абстрагируя сложные настройки и предоставляя возможность повторного использования кода .
- **Пример**: Модули могут использоваться для настройки служб, параметров загрузки и других системных компонентов .

2. **Конфигурация**:

- **Функция**: Конфигурация — это набор настроек, определяющих поведение системы.
- **Назначение**: Конфигурация создается на основе модулей и содержит все параметры, необходимые для работы системы.
- **Пример**: Конфигурационный файл `configuration.nix` в NixOS определяет всю конфигурацию системы, используя различные модули .

В заключение, модули в NixOS используются для организации и модульности конфигурации, в то время как конфигурация является конечным результатом работы этих модулей, определяющим настройки системы.

??????????

- **Опция** — необязательная возможность, меняющая поведение функции. Опция может иметь один или набор параметров.

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

1. NixOS modules
2. Understanding NixOS Modules and Declaring Options

?????????

> **Наложения (overlays)** – это функции на языке Nix, которые позволяют модифицировать существующие пакеты и добавлять новые, изменяя пакеты в Nixpkgs (библиотеке пакетов Nix).

Они принимают в качестве аргументов текущее состояние пакетов (final) и предыдущее (prev), а на выходе возвращают обновленный набор пакетов, что позволяет тонко настраивать систему без необходимости форкать или изменять оригинальный Nixpkgs.

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

???????????

Профиль - механизм отката изменений или смены поколений.
Поколение - версии профилей.
Деривация - источник со своим путём хранения(ответвление) в хранилище nix. Аналог **пакета** в других дистрибутивах linux.
Канал - аналог репозитория в других дистрибутивах linux.

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

Команда	Описание
nix-env -i <имя_деривации>	Установить деривацию
nix-env -q	Вывести установленные деривации
nix-env -e <имя_деривации>	Удалить деривацию
nix-env -u	Обновить все деривации в окружении

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

Команда	Описание
nix-env --list-generation	Вывести список поколений
nix-env --rollback	Откатиться к предыдущему поколению
nix-env -G <номер_поколения>	Перейти к поколению
nix-collect-garbage	Запустить сборщик мусора
nix-collect-garbage --delete-old	Очистить от всех поколений кроме последнего

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

Команда	Описание
nix-store -q --references `which <имя_программы>`	Вывести зависимости программы

nix-store -q --referrers `which <имя_программы>`	Вывести зависимых от программы
nix-store -qR `which <имя_программы>`	Вывести список всех зависимостей программы
nix-store -q --tree `which <имя_программы>`	Вывести список всех зависимостей программы в виде дерева
nix-store --read-log /nix/store/<имя деривации>	Вывести журнал сборки деривации
nix-store -q --root	Запросить корни сборщика мусора

??????

Команда	Описание
nix-channel --list	Вывести список каналов
nix-channel --update	Скачать новые описания дериваций, новое поколение профиля каналов и распаковать его в `~/.nix-defexpr/channels`

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

- Nix pillls.
- Nix pills(официальный перевод)
- Nix: Что это и с чем это употреблять?

????? ??????

Nix ? ???????

1. Nix в пилюлях: Почему вам стоит попробовать Nix.
2. Nix в пилюлях: Установка в вашей системе.
3. Nix в пилюлях: Погружаемся в среду.
4. Nix в пилюлях: Основы языка.
5. Nix в пилюлях: Функции и импорт.
6. Nix в пилюлях: Наша первая деривация.
7. Nix в пилюлях: Работающая деривация.
8. Nix в пилюлях: Универсальные скрипты сборки.
9. Nix в пилюлях: Автоматические зависимости времени выполнения.
10. Nix в пилюлях: Разработка с помощью nix-shell.