## **Оригинальный текст**

## 2 . 1 The Parts of a Language

Engineers have been building programming languages since the Dark Ages of computing. As soon as we could talk to computers, we discovered doing so was too hard, and we enlisted their help. I find it fascinating that even though today’s machines are literally a million times faster and have orders of magnitude more storage, the way we build programming languages is virtually unchanged.

Though the area explored by language designers is vast, the trails they’ve carved through it are few. Not every language takes the exact same path—some take a shortcut or two—but otherwise they are reassuringly similar, from Rear Admiral Grace Hopper’s first COBOL compiler all the way to some hot, new, transpile-to-JavaScript language whose “documentation” consists entirely of a single, poorly edited README in a Git repository somewhere.

There are certainly dead ends, sad little cul-de-sacs of CS papers with zero citations and now-forgotten optimizations that only made sense when memory was measured in individual bytes.

I visualize the network of paths an implementation may choose as climbing a mountain. You start off at the bottom with the program as raw source text, literally just a string of characters. Each phase analyzes the program and transforms it to some higher-level representation where the semantics—what the author wants the computer to do—become more apparent.

Eventually we reach the peak. We have a bird’s-eye view of the user’s program and can see what their code *means*. We begin our descent down the other side of the mountain. We transform this highest-level representation down to successively lower-level forms to get closer and closer to something we know how to make the CPU actually execute.

Let’s trace through each of those trails and points of interest. Our journey begins on the left with the bare text of the user’s source code:

var average = (min + max) / 2;

### **2 . 1 . 1 Scanning**

The first step is **scanning**, also known as **lexing**, or (if you’re trying to impress someone) **lexical analysis**. They all mean pretty much the same thing. I like “lexing” because it sounds like something an evil supervillain would do, but I’ll use “scanning” because it seems to be marginally more commonplace.

A **scanner** (or **lexer**) takes in the linear stream of characters and chunks them together into a series of something more akin to “words”. In programming languages, each of these words is called a **token**. Some tokens are single characters, like ( and ,. Others may be several characters long, like numbers (123), string literals ("hi!"), and identifiers (min).

“Lexical” comes from the Greek root “lex”, meaning “word”.

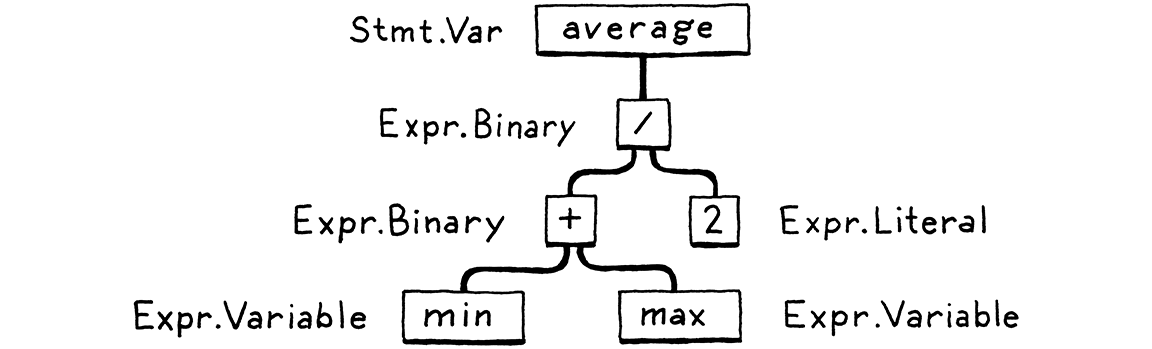
Some characters in a source file don’t actually mean anything. Whitespace is often insignificant, and comments, by definition, are ignored by the language. The scanner usually discards these, leaving a clean sequence of meaningful tokens.



### 2 . 1 . 2 Parsing

The next step is **parsing**. This is where our syntax gets a **grammar**—the ability to compose larger expressions and statements out of smaller parts. Did you ever diagram sentences in English class? If so, you’ve done what a parser does, except that English has thousands and thousands of “keywords” and an overflowing cornucopia of ambiguity. Programming languages are much simpler.

A **parser** takes the flat sequence of tokens and builds a tree structure that mirrors the nested nature of the grammar. These trees have a couple of different names—**parse tree** or **abstract syntax tree**—depending on how close to the bare syntactic structure of the source language they are. In practice, language hackers usually call them **syntax trees**, **ASTs**, or often just **trees**.



Parsing has a long, rich history in computer science that is closely tied to the artificial intelligence community. Many of the techniques used today to parse programming languages were originally conceived to parse *human* languages by AI researchers who were trying to get computers to talk to us.

It turns out human languages were too messy for the rigid grammars those parsers could handle, but they were a perfect fit for the simpler artificial grammars of programming languages. Alas, we flawed humans still manage to use those simple grammars incorrectly, so the parser’s job also includes letting us know when we do by reporting **syntax errors**.

Источник: <https://craftinginterpreters.com/a-map-of-the-territory.html>

# Превод Google Translate

## 2 .1 Части языка

Инженеры создавали языки программирования со времен Темных веков вычислительной техники. Как только мы научились разговаривать с компьютерами, мы обнаружили, что это слишком сложно, и заручились их помощью. Меня поражает тот факт, что хотя современные машины буквально в миллион раз быстрее и имеют на порядки больше памяти, способы создания языков программирования практически не изменились.

Хотя область, исследованная языковыми дизайнерами, обширна, тропы, которые они проложили через нее, невелики. Не все языки идут по одному и тому же пути — некоторые идут коротким путем или двумя, — но в остальном они обнадеживающе похожи: от первого компилятора COBOL контр-адмирала Грейс Хоппер до какого-то модного нового языка, транспилируемого в JavaScript, чья «документация» полностью состоит из одного плохо отредактированного файла README где-то в репозитории Git.

Конечно, существуют тупики, печальные тупики статей по компьютерной науке с нулевым цитированием и ныне забытыми оптимизациями, которые имели смысл только тогда, когда память измерялась в отдельных байтах.

Я представляю себе сеть путей, которые реализация может выбрать, как восхождение на гору. Вы начинаете с самого низа программы в виде необработанного исходного текста, буквально строки символов. На каждом этапе программа анализируется и преобразует ее в некое представление более высокого уровня, где семантика — то, что автор хочет, чтобы компьютер делал — становится более очевидной.

В конце концов мы достигаем вершины. Мы видим программу пользователя с высоты птичьего полета и можем видеть, что означает его код. Начинаем спуск с другой стороны горы. Мы преобразуем это представление самого высокого уровня в последовательные формы более низкого уровня, чтобы все ближе и ближе приближаться к тому, что мы знаем, как заставить ЦП действительно выполняться.

Давайте проследим каждую из этих троп и достопримечательностей. Наше путешествие начинается слева с чистого текста исходного кода пользователя:

### 2 . 1 . 1 сканирование

Первый шаг — сканирование, также известное как лексинг или (если вы пытаетесь произвести на кого-то впечатление) лексический анализ. Все они означают примерно одно и то же. Мне нравится «лексинг», потому что это похоже на то, что сделал бы злой суперзлодей, но я буду использовать «сканирование», потому что это кажется немного более обычным явлением.

Сканер (или лексер) воспринимает линейный поток символов и объединяет их в серию, более похожую на «слова». В языках программирования каждое из этих слов называется токеном. Некоторые токены представляют собой одиночные символы, например ( и ,). Другие могут иметь длину в несколько символов, например числа (123), строковые литералы («привет!») и идентификаторы (мин).

Слово «лексический» происходит от греческого корня «lex», что означает «слово».

Некоторые символы в исходном файле на самом деле ничего не означают. Пробелы часто незначительны, а комментарии по определению игнорируются языком. Сканер обычно отбрасывает их, оставляя чистую последовательность значимых токенов.

### 2 . 1 . 2 Парсинг

Следующий шаг — разбор. Именно здесь наш синтаксис получает грамматику — возможность составлять более крупные выражения и инструкции из более мелких частей. Вы когда-нибудь рисовали предложения на уроках английского языка? Если да, то вы сделали то же, что и парсер, за исключением того, что в английском языке есть тысячи и тысячи «ключевых слов» и избыточный рог изобилия двусмысленностей. Языки программирования намного проще.

Анализатор берет плоскую последовательность токенов и строит древовидную структуру, отражающую вложенный характер грамматики. Эти деревья имеют несколько разных названий — дерево синтаксического анализа или абстрактное синтаксическое дерево — в зависимости от того, насколько они близки к простой синтаксической структуре исходного языка. На практике языковые хакеры обычно называют их синтаксическими деревьями, AST или просто деревьями.

Синтаксический анализ имеет долгую и богатую историю в информатике, тесно связанную с сообществом искусственного интеллекта. Многие методы, используемые сегодня для анализа языков программирования, изначально были задуманы для анализа человеческих языков исследователями искусственного интеллекта, которые пытались заставить компьютеры разговаривать с нами.

Оказывается, человеческие языки были слишком запутанными для жестких грамматик, с которыми могли справиться эти парсеры, но они идеально подходили для более простых искусственных грамматик языков программирования. Увы, нам, несовершенным людям, все еще удается неправильно использовать эти простые грамматики, поэтому работа парсера также включает в себя информирование нас, когда мы это делаем, сообщая о синтаксических ошибках.

# Перевод Yandex Translate

## 2.1 Части языка

Инженеры разрабатывали языки программирования со времен Темных веков вычислительной техники. Как только мы научились разговаривать с компьютерами, мы обнаружили, что делать это слишком сложно, и заручились их помощью. Я нахожу удивительным тот факт, что, несмотря на то, что современные машины буквально в миллион раз быстрее и имеют на порядки больше памяти, способ, которым мы создаем языки программирования, практически не изменился.

Хотя область, исследованная языковыми дизайнерами, обширна, тропинок, которые они проложили в ней, немного. Не все языки идут по одному и тому же пути — некоторые выбирают один или два коротких пути, — но в остальном они обнадеживающе похожи, начиная с первого компилятора COBOL контр-адмирала Грейс Хоппер и заканчивая каким-то популярным новым языком, преобразуемым в JavaScript, чья “документация” полностью состоит из одного плохо отредактированного README где-нибудь в репозитории Git. Безусловно, есть тупики, маленькие печальные тупички статей CS с нулевыми цитатами и ныне забытыми оптимизациями, которые имели смысл только тогда, когда объем памяти измерялся в отдельных байтах.

Я представляю сеть путей, которые реализация может выбрать, как восхождение на гору. Вы начинаете с самого низа, имея программу в виде необработанного исходного текста, буквально просто строки символов. На каждом этапе программа анализируется и преобразуется в представление более высокого уровня, где семантика - то, что автор хочет, чтобы компьютер делал, — становится более очевидной.

В конце концов мы достигаем вершины. Мы видим программу пользователя с высоты птичьего полета и можем видеть, что означает их код. Мы начинаем наш спуск с другой стороны горы. Мы преобразуем это представление самого высокого уровня в формы последовательно более низкого уровня, чтобы все ближе и ближе подходить к тому, что мы знаем, как заставить центральный процессор фактически выполнять.

Давайте проследим каждый из этих путей и точек интереса. Наше путешествие начинается слева с простого текста исходного кода пользователя:

### 2.1.1 Сканирование

Первым шагом является сканирование, также известное как лексинг, или (если вы пытаетесь произвести на кого-то впечатление) лексический анализ. Все они означают практически одно и то же. Мне нравится “лексинг”, потому что это звучит как что-то, что сделал бы злой суперзлодей, но я буду использовать “сканирование”, потому что это кажется немного более распространенным явлением.

Сканер (или лексер) принимает линейный поток символов и объединяет их в последовательность, более похожую на “слова”. В языках программирования каждое из этих слов называется токеном. Некоторые токены состоят из отдельных символов, например ( и ,. Другие могут состоять из нескольких символов, например чисел (123), строковых литералов ("привет!") и идентификаторов (min).

“Лексический” происходит от греческого корня “lex”, означающего “слово”.

Некоторые символы в исходном файле на самом деле ничего не значат. Пробелы часто несущественны, а комментарии, по определению, игнорируются языком. Сканер обычно отбрасывает их, оставляя чистую последовательность значимых токенов.

### 2.1.2 Синтаксический анализ

Следующий шаг - синтаксический анализ. Именно здесь наш синтаксис приобретает грамматику — способность составлять более крупные выражения и утверждения из более мелких частей. Вы когда-нибудь рисовали предложения на уроках английского языка? Если это так, то вы сделали то, что делает синтаксический анализатор, за исключением того, что в английском языке тысячи и тысячи “ключевых слов” и переполненный рог изобилия двусмысленностей. Языки программирования намного проще.

Синтаксический анализатор берет плоскую последовательность токенов и строит древовидную структуру, отражающую вложенный характер грамматики. У этих деревьев есть несколько разных названий — дерево синтаксического анализа или абстрактное синтаксическое дерево — в зависимости от того, насколько они близки к простой синтаксической структуре исходного языка. На практике языковые хакеры обычно называют их синтаксическими деревьями, AST или часто просто деревьями.

Синтаксический анализ имеет долгую и богатую историю в информатике, которая тесно связана с сообществом искусственного интеллекта. Многие из методов, используемых сегодня для анализа языков программирования, изначально были задуманы для анализа человеческих языков исследователями искусственного интеллекта, которые пытались заставить компьютеры разговаривать с нами.

Оказывается, человеческие языки были слишком запутанными для жестких грамматик, с которыми могли справиться эти анализаторы, но они идеально подходили для более простых искусственных грамматик языков программирования. Увы, мы, ущербные люди, все еще умудряемся неправильно использовать эти простые грамматики, поэтому в обязанности синтаксического анализатора также входит сообщать нам, когда мы это делаем, сообщая о синтаксических ошибках.

# Сравнение результатов

Были выписаны следующие незнакомые слова:

| **Незнакомое слово** | **Livigvo Live** | **Перевод Google Translate** | **Перевод Yandex Translate** |
| --- | --- | --- | --- |
| reassuringly | успокоительно | обнадеживающе | обнадеживающе |
| dead ends | тупики | тупики | тупики |
| semantics | Семантика | семантика | семантика |
| marginally | косвенно | немного | немного |
| commonplace | Общее место; банальность | обычное дело | распространенное явление |
| discards | сбрасывание | отбрасывает | отбрасывает |
| rigid | Жесткий, негнущийся | жестких | жестких |

# Облако слов



## Топ 5 слов:

1. Language
2. Tree
3. Programming languages
4. Grammar
5. Syntax