What is a programming Language?
Why should students study programming languages?
How do people study programming languages?

1 The Pragmatics Question

Software developers create software systems with programming languages and their tool chains. They edit code in an IDE, which, in turn, provides feedback as the code evolves. They read code and they rely on properties of the language to help them with this task; for example, in a type-safe language, a method that consumes an object of type T is guaranteed to be created by a corresponding constructor. They add test suites and run them; the IDE indicates which tests fail. They use failing test cases to start the bug locating process, using the debugger to set breakpoints so that they can see where execution goes wrong. They change the code to fix bugs once found. They measure the performance of systems, and if it is found wanting, they try replace low-performing pieces with faster ones. During this entire process, software developers engages with the chosen programming language and its tools, and the question is how well the language and its tools serve their purpose in each of these situations.

Linguists have studied natural languages and their use in various contexts for even longer than programming language researchers have studied their subjects. They recognize this description immediately. They have a word for it: *pragmatics*—short for "use of language in context." Linguists appreciate the central role that pragmatics plays in improving people's understanding of the fundamental laws of language.

Our use of "pragmatics" slightly differs from the use in linguistics.

This text book tackles the question of programming-language pragmatics head on. Following the historic development, the emphasis is on how

certain elements of languages facilitate the creation of code and reasoning about it. With "reasoning" we mean reading, understanding relationships between pieces of code, validating those, and identifying problematic spots. Each element of a language represents a design decision, that is, a choice by language creators to pick one of several alternatives. Such design alternatives come with different costs and benefits, and understanding the rationale behind the choice is an essential aspect of the study of programming languages.

Our journey through the landscape of programming languages starts with the bare-bones language from the beginning of time. The path gets gradually steeper with the addition of classes, modules, and types. Near the end, this book presents the design rationale behind the essence of Type Script, a recently designed and already widely used language. Its construction combines many advanced language elements in an intriguing manner. At the same time, the language is a case study of how its creators strictly emphasized one cost-benefit aspect over all others—to the potential detriment of developers because this choice denies support in many important uses cases.

2 Programming Language

A *programming language* is an artificial language. Like a natural language, a programming language has three properties that a programmer must understand to use it properly: syntax, semantics, and pragmatics.

- *Syntax* consists of the vocabulary and the grammatical rules. These rules determine which sequences of characters form a "word" and which sequences of "words" are *well-formed* sentences. A syntax specification may also impose *validity* constraints on well-formed sentences before one of these is accepted as an actual program.

- *Semantics* assigns meaning to programs and, implicitly, to its pieces. This meaning is typically specified via a natural-language document; a few programming languages come with mathematical specifications, based on the research of "semanticists.". A software developer, however, has to accept whatever the chosen implementation of a language does when presented with a program. Language researchers tend to call the latter "behavior" to differentiate it from the specified and thus *desired* semantics of the language.

- *Pragmatics* is about the usage of a language in specific work situations. This wording covers the creation of programs, the reading of code, the search for bugs, and so on. Unlike syntax and semantics, pragmatics remains an imprecise term and the idea of pragmatics has received much less attention from researchers than syntax and semantics—even though it clearly matters most to the working software developer.

3 Studying Programming Languages: Why

When the weather app on your phone tells you in the morning that it is going to rain today with a 90% probability, you can't do much about the rain. Similarly, when the central bank announces an increase in the price of money—also known as interest rate—your parents can't change the bank's mind just because they need to get a mortgage now. And when a manager tells a newly hired junior developer that the team's chosen programming language is PLX, then this new person can't just start writing code in some different language YPL, because it is fashionable right now.

But, in all of these situations, you can act consistently with the new information. As they say, there is no bad weather, there's only bad clothing. So, if the app says it's going to rain and you have to go somewhere on foot, you will grab a rain coat or an umbrella. And if your parents are really into buying a home with a mortgage, they will pay attention to what the central bankers discuss prior to an interest-rate meeting in public, and they may accelerate their home purchase. Put differently, learning and following basic rules of thumb helps people navigate life situations, small and large.

As a newly hired junior software developer you can't change the language that the team has already chosen for its on-going project. Indeed, it may take several years before you can influence the choice of language for a "green field" project. But, if you enter professional life equipped with a solid understanding of the principles of syntax, semantics, and pragmatics, you will have a head start. To understand why, we need to consider how people study programming languages.

4 Studying Programming Languages: How

Programming language researchers use models to study programming languages. In this, they follow standard scientific procedure. Roughly speaking, a model reduces an object of study to those elements that a researcher

wishes to understand in depth—without eliminating elements that would invalidate any results for the actual object. Put differently, the goal is to answer questions about the object via the model so that these answers are of some value in the actual world.

One great example is physics. In many cases, a physicist can think about moving objects as if they were simple points, even though the object may experience friction due to its extent. Similarly, a system biologist may act as if DNA were just a sequence of "letters" (A, C, G, and T), but these letters are nucleotide bases with chemical and physical properties. Models are also of use in social sciences. Classical economists assume that human beings act rationally as participant in economic activities, and they do so with some success for macro-level predictions.

Over time, researchers have developed three kinds of programming-language models:

- informal descriptions, using natural languages;
- mathematical functions and relations; and
- executable programs.

All of these share the key property of models, namely, that they do not cover the entire language to be studied. They tend to restrict the syntax to essential features, and they tend to formulate a semantics that covers a wide range of cases, not all of them.

Each of these approaches has advantages and disadvantages. While a natural language description of a programming language can be used for an entire language, it necessarily comes with gaps. By their very nature, natural languages tend to allow ambiguous formulations. A mathematical model is usually extremely exact, but it tends to describe a small, even minuscule fraction of any given real language. To a working developer, an executable model of any aspect of a programming language has the advantage that it runs and that it is amenable to quick explorations for specific cases. If the source code is available, a programmer can even explore alternatives to the chosen language design. At the same time, writing a program to understand a fragment of a programming language means that an understanding of one language—at least partially—depends on an understanding of another one. For this reason, this text uses executable models while borrowing elements from the mathematical side—this allows for making precise models without tying them too closely to the chosen implementation language.

Implementing executable models is also a good way to reinforce good programming habits.

Given this background, let's take a close but quick look at each of the properties of a programming language—that is, how people about and what people get out of studying syntax, semantics, and pragmatics.

4.1 Syntax

People are told not to judge a book by its cover, but beginning programmers—and experienced ones—routinely do. The "cover" of a programming language is the program text that developers enter into an IDE, read, and comprehend. Different language creators make different assumptions about which "letters" make up the *vocabulary* of the words of programs. And, these assumptions have a direct impact on developers. Some opt to allow only sequences of alpha-numerical characters as the "letters" of the vocabulary; others accept that developers wish to use fahrenheit->celsius as the name of a temperature conversion function.

Similarly, language creators also make assumptions about how developers wish to compose words into sentences. It starts with simple expressions, that is, the *grammatical rules* that govern how individual words are composed into expressions. Some creators think that the infix notation of algebra, which only emerged over the past couple of centuries, elegantly generalizes to programming, where developers define new operators all the time. Thus, they allow developers to write

```
x -> f ++ k
```

and, implicitly, assume that all future readers of such expressions can easily decipher the order in which the two operations are evaluated. Others prefer to have developers spell out this order of operator precedence so that the same expression may look like this:

```
(++ (-> x f) k)
```

After all, all students in elementary schools know that parentheses clarify operator precedence. People also intensely disagree whether ; is a statement terminator or separator; others want TAB characters to signify something about programs; and the parentheses lovers don't care because parentheses and a bit of white space make it all obvious.

This approach is the definition of abstraction.

A student of programming languages must ignore the specific differences of syntax and focus on the commonalities. This focus is a step from concrete examples to general ideas, insights that should be useful across languages. Concerning syntax, language researchers figured out two such general insights by the 1960s:

1. The first one is a sufficiently general technique for describing vocabularies and grammars. In essence, this technique—called *Backus-Naur Form* (short BNF)—is a standard grammar for defining grammars.

   A logical-mathematical way of expressing this idea is that a grammar consists of a collection of n equations in n variables. Unlike the equations we know from high school and college courses, solutions to these grammar equations are sets of terms.

2. The second concerns the description of syntax with respect to questions concerning semantics. Whether a language uses prefix, infix, or postfix notation for an addition expression is simply irrelevant. What matters is that (1) the description brings across that it *is* about addition, (2) how many operands there are, and (3) how to refer to the operands. This idea is referee to as *abstract syntax*.

Chapter II expands on these insights and provides concrete examples. Critically, it explains how these two insights are the basis of a straightforward implementation of syntax checking—an executable model of syntax.

4.2 Semantics

English speakers are familiar with the phrase "it's just semantics," typically used to dismiss an argument because the differences are trivial, but people who know that "semantics" is a fancy word for "meaning" must be stumped by this usage. Of course, it's all about "meaning" and software developers are perfectly aware of this fact. They know that it matters whether

> an array reference enforces the container's boundaries.

If it doesn't, an array reference can cause a program to segfault when it uses a negative index or an index that exceeds the length of the array. Alternatively, the lookup succeeds and retrieves random bits that masquerade as an array element—and the code may appear to terminate normally and output proper results. This kind of semantics has posed major security problems for decades. By contrast, a boundary-enforcing array reference signals a run-time exception for such use cases, preventing catastrophic misinterpretations. In short, semantics very much matters.

Inspired by the success of investigations into syntax, programming language researchers struggled for quite some time to come up with a simple mathematical framework for describing the meaning of programs. Here are the most well-known, in historical order:

- An interpreter is a recursive function from syntax to values in some chosen programming language.

- An *abstract machine* consists of two pieces: (1) a (large) set of states and (2) a transition function that maps one state to another.

  Instead of the sequence of all bits in a hardware machine—where each feasible sequence of 0s and 1s describes a possible state—an abstract machine uses pieces of the modeled language. Some states are marked as initial, others as final.

  The function's cases correspond to high-level instructions, hence its name. For final machine states, this function is undefined.

  Given a program, it is loaded into the machine, which puts it into an initial state. Then the transition function is applied until the machine is in a final state. This final state is unloaded to yield an "answer."

- A *denotational semantics* consists of a *domain*—a set of mathematical elements that satisfies some properties—-plus a function from syntax to the domain. This meaning function tends to resemble an interpreter.

- An *operational semantics* combines ideas from the meaning function of denotational semantics with those of the world of abstract machines.

None can easily express all possible behaviors.

Of these approaches, the interpreter—by definition—and the abstract machine approach lend themselves most easily to implementation. Implementing an interpreter or abstract machine yields a first impression of how much work it will be build a performant implementation. Most importantly, with an implementation, people can check whether programs behave in the expected manner; they can explore corner cases of behavior; and they can begin to answer questions about how software developers can use language elements in various work situations.

4.3 The Covers of Books, or It's Just Semantics

Given some understanding of the notions of syntax and semantics, we can now explain in with a single word why the former is less meaningful than the latter. Consider the words

Ignore the use of the ugly capital letter.

US English	German
billion	Billion

At first glance, the two words from two different natural languages seem to correspond to each other. So if someone were to offer you a billion pennies, you would probably not worry about which language the person uses to make this promise. This is *syntax*.

It turns out, however, that the semantics of these words differs radically:

US English	German
billion	Billion
1,000,000,000	1,000,000,000,000

When someone tells you next time "it's just semantics," ask for a Billion dollars.

Imagine this. The *semantics* of the two words differ by a factor of 1,000, which at the level of pennies, makes a serious difference.

Which language should the promise use now?

This difference is what we refer to when we say "don't judge the language by its syntax." While the way you read and write the words of a program clearly affects your abilities as a programmer, an easy-to-grasp semantics of the syntax matters even more than the plain words. And how the combination relates to the productivity of developers in certain work situations is "pragmatics," the final topic of this first chapter.

4.4 Pragmatics

Models of syntax and semantics enable researchers to explore question of pragmatics. Executable models help most, because researchers can interact with them and explore ideas rapidly. Well-organized executable models also lend themselves to language experimentation.

Such a language experiment can help answer questions about language pragmatics. Consider the already-mentioned case of an array lookup. Employing the widely used syntax of `a[i]` a researcher can ponder what the impact of the two distinct meanings is in various work situations. Recall the two possible meanings, formulated in terms of a machine controlled by program expressions and a memory:

- `a[i]` checks whether `i` is between `0` and the upper boundary of `a`. If so, it retrieves the corresponding element; otherwise it stops the executing machine or interpreter and signals a problem.

- `a[i]` retrieves the `i`th element or, if `i` is outside the boundary, makes up an element that meets the type of `a`. The machine continues to execute the next instruction.

A researcher can now ask two different, but related sets of questions. The first set concerns the costs and benefits to the person who uses the language. The second one is of interest to the implementer of the language, that is, the person who builds a robust and performant implementation from the executable model.

Let's take a look at some work situations that developers encounter:

- During the editing of code, what matters is whether the sequence of characters entered into the IDE form a well-formed and valid sentence. Since the syntax is the same for both meanings, editing code isn't affected by the choice of meaning.

- The developer runs the program and notices that it doesn't run as fast as expected. It turns out that the most frequently executed code consists of a nest of loops with many array references in the body of the innermost loop. In this case, it is quite possible that a boundary-checking implementation of array references causes the bottleneck. That is, a developer might prefer the non-checking variant here.

- When the program is tested, a quality assurance engineer notices an unusual output. Although the program does not crash, its answers just seem off. Now the engineer may question whether a non-checking array reference is going wrong and whether by going wrong it injects some random element that causes the program to give a bad answer—at least on some occasions. This developer would clearly prefer the bounds-checking variant of array references.

  Alternatively, the developer should inspect all array references and manually wrap them with boundary checks. That is, a developer can take on the work of implementing the checked variant based on the unchecked one, if needed.

In sum, from the perspective of a language user, there might not be a clear answer as to which meaning of some syntax is preferable.

From the perspective of a language implementer, the questions are related those of a user but differ a bit for this specific case. Let's assume the implementer wishes to write a compiler, that is, a translation of source code to assembly. An array lookup with two different meanings is subject to the following considerations:

- A translation of a non-checking array reference is straightforward. Assuming `a` is represented as a memory address and `i` denotes an off-

set, the compiler should issue an instruction to load the bits at memory location `a + i`.

- Without any further information, the compilation for the alternative meaning differs sharply. The compiler cannot represent an array as just an address; it must store the size of the array somewhere. This number must be retrieved and compared to `i`, assuming the latter is even a non-negative integer. If the comparison succeeds, the retrievable of the bits can proceed in a manner similar to the first case; otherwise the sequence of assembly instructions must jump to a place that reports the error.

- The phrase "without any further information" is suggestive. What if the compiler could "think" about the program and determine up front that some particular array reference will always succeed? That is, the generated code does not need to perform the bound checking.

  Next the author of the compiler must consider (1) how difficult it is to implement this "thinking," (2) how safe this thinking is, and (3) how much it slows down the translation process.

Together these two sets of questions yield a cost-benefit analysis. For many decades, the decision almost always came down to what was easiest to implement. That is, until people discovered that unchecked array references are a major source of safety and security flaws. In turn, the work shifted from the many software developers using the language to the few compiler writers.

We ignore the case when two forms of syntax have the same meaning, because we consider this one mostly a question of taste.

Questions of pragmatics aren't always as clear-cut as the one for array references. Design alternatives may require different source syntax and different meaning. People encounter many different work situations, not just editing code, observing performance, and locating bugs. For example, a student in a beginning programming course *is* in a work situation, a situation that differs from those just mentioned. Try to imagine some others.

This rest of book focuses on those question of pragmatics that concern the working developer. The expectation is that the series of examples help (future) developers create an informal framework for approach programming languages and evaluating features of programming languages. Before we can get started though, we need to take a close look at the basic notions of syntax and semantics.

What is a (n executable) model of syntax?

1 Concrete Syntax is Mostly Irrelevant

Different programming languages use different kinds of syntax. Many present syntax as sequences of any characters, with words delineated via some white space, and sentences separated via semicolons or special forms of white space. Language creators write down rules that explain the organization of sequences of characters into words, the grouping of words into sentences, and the recognition of sentences as complete programs.

Borrowing terminology from linguistics, these rules are called *grammars*. The word *Parsing* denotes the process of recognizing sequences of characters as words, groups of words as sentences, and certain sentences as complete program. A *parser* implements the parsing process.

Lisp stands out. The members of the Lisp family of languages use parentheses, braces, and brackets (and arbitrary white space) to separate words and to organize words into groups. Indeed, due to nesting, it is the programmer who takes on some of the task of organizing code to help the parser. People refer to this kind of notation as a *concrete syntax tree*, because the nesting of parenthesize sentences suggests what computer scientists call trees. In Lisp languages, notations for instructions as well as data employ parenthetical notation since the late 1950s; the latter is called an *S-expression*.

The designers of contemporary data description notations, such as XML and JSON, have adapted Lisp's S-expression notation, while making a few concessions to people who like conventional syntax. They employ brackets, braces, and named parentheses to organize data. Developers tend to speak of *semi-structured data*.

XML's pairs of parentheses have the shape <letters> ... </letters>.

Due to the large variety of syntax, studying grammars and parsing techniques has essentially become its own research area. On the positive side, grammars for describing grammars date back to the 1960s, and their expressive power and limitations is well understood. Furthermore, by the 1970s, programming language researchers could offer programs that turn grammars into parsers, so-called *parser generators*. On the negative side, using these tools isn't straightforward. Worse, while a generated parser typically is good at determining whether some sequence of characters satisfies the rules of a grammar, it typically fails to provide an accurate diagnosis when the characters fail the rules.

This is also a reason why courses for beginners should avoid text and parsing.

Hence, if a course on programming language pragmatics were to pay close attention to grammars and parsing, students might never get to see any topics related to semantics or pragmatics. To avoid this problem, we compromise. Concretely, this book uses semi-structured data as concrete syntax for its sample programs; to make this precise, it uses S-expressions. As the history of Lisp and data notations validates, programmers can easily use such a notation to write down programs. At the same time, it enables us to explain the concepts of grammar and parsing in a concise manner and to include a discussion of the pragmatics of parsing.

2 Grammars for Describing Grammars

As the preceding chapter mentions, BNF (or one of its modern variants) is the most commonly used tool to describe the concrete syntax of a programming language. Instead of describing BNF formally, we present just enough examples so that readers can easily comprehend the syntax descriptions in the remaining chapters.

The primary element of a BNF grammar is a *production*, which have this shape:

```
Program ::= a
```

The ::= symbol separates the name that is being defined from its definition. Our convention is to use capitalized words for defined entities and lower case words plus other sequences of keyboard characters for pieces of code. People pronounce such a definition as "Program is a".

A typical grammar describes elements of a language's syntax as not just one possible shape but several possible shapes:

```
Program ::= a | b | c
```

The BNF symbol | is pronounced as "or" and, when a production has this shape, the ::= is pronounced as "one of."

For the syntax description of most languages, it is necessary to use several productions. Additionally, grammatical rules often need to specify that developers may repeat some element several times:

```
Program    ::= Statement* Expression
Statement  ::= print(Expression);
Expression ::= 43
```

This collection of three production rules informs a programmer of three definitions:

1. A Program is a possibly empty sequence of Statements followed by one Expression.

2. A Statement consists of a four pieces: the word print, the character `(`, an Expression, and the characters `)` and `;`.

3. An Expression is just the text 43.

Given a bunch of productions, a programmer can follow their implied instructions to write code. For example, the following is a Program according to the above rules:

```
43
```

It consists of no Statement and the required Expression. Here is another one:

Our definitions leave it implicit that white space is ignored.

```
print(43); print(43); 43
```

This program consists of two Statements, followed by the one required Expression.

Finally, a language creator may wish to require that there is at least one particular piece in a sequence, say, at least one Statement in a Program. The BNF convention is to use a plus superscript as follows:

```
Program    ::= Statement+ Expression
```

And that is all there is to describing grammars with BNF as far as this book is concerned.

3 The Parsing Process and the Parser Function

While the first role of a BNF is to inform the future developer about the shape of well-formed code, its second role is to tell a language implementer how to parse any given text according to some production. For example, given the text

```
44
```

it is impossible to parse it according to the production for Expression. Similarly,

```
print(44);
```

does not belong to the syntax specified by Statement, even though a lot of the characters of this text fit the pattern of the production. By contrast,

```
print(43);
```

is a legal Statement.

In general, parsing some text essentially answers a yes-no question relative to some production:

- yes, the given text belongs to all the collection of all texts that the production describes;
- no, the given text does not belong to the collection of texts that the production describes.

The task of a language implementer is to create a program that, given some inputs, answers this question.

At first glance, this suggests that a parser for some production has the following signature:

```
parseExpression : PlainText  -> Boolean
  // does the given text belong the collection of texts
  // described by the Expression production
```

In words, the parse for the Expression production from the preceding section consumes some text and produces true if the given text belongs to the syntax of the production; otherwise, it returns false to indicate that the text somehow does not fit the description.

For this particular production, it is trivial to define this function:

```
parseExpression(text) =
  true if the text consists of ``4'' followed by ``3''
  false otherwise
```