3
Expressions and the linguistics of programming languages

Programming is an expressive activity: We express our intentions to a computer using a language – a programming language – that is in some ways similar to the natural languages with which we communicate with each other.

One of the deep truths of linguistics, known since the time of the great Sanskrit grammarian Pāṇini in the fourth century BCE, is that the expressive units of natural languages, or EXPRESSIONS as we will call them, have hierarchical structure. (The recovery of that structure used to be a typical subject matter taught to students in “grammar school” through the exercise of sentence diagramming.) Characterizing what are the well-formed and -structured phrases of a language constitutes the realm of SYNTAX.

3.1 Specifying syntactic structure with rules

The expressions of English (and other natural languages) are formed as sequences of words to form expressions of various types. By way of example, noun phrases can be formed in various ways: as a single noun (*party* or *drinker* or *tea*), or by putting together (in sequential order) a noun phrase and a noun (as in *tea party*), or by putting together (again in order) an adjective (*iced* or *mad*) and another noun phrase as in (*iced tea*). We can codify these rules by defining classes of expressions like ⟨*noun*⟩ or ⟨*nounphrase*⟩ or ⟨*adjective*⟩. We’ll write the rule that allows forming a noun phrase from a single noun as

⟨*nounphrase*⟩ ::= ⟨*noun*⟩

The rules that form a noun phrase from an adjective and a noun phrase or from a noun phrase and a noun are, respectively,

⟨*nounphrase*⟩ ::= ⟨*adjective*⟩⟨*nounphrase*⟩

⟨*nounphrase*⟩ ::= ⟨*nounphrase*⟩⟨*noun*⟩

In these rules, we write ⟨*noun*⟩ to indicate the class of noun expressions, ⟨*nounphrase*⟩ to indicate the class of noun phrases, and in general, put the names of classes of expressions in angle brackets to represent elements of that class. The notation ::= should be read as "can be composed from", so that expressions of the class on the left of the ::= can be composed by putting together expressions of the classes listed on the right of the ::=, in the order indicated.

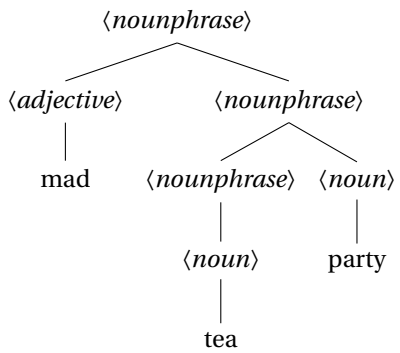
This rule notation for presenting the syntax of languages is called BACKUS-NAUR FORM (BNF), named after John Backus and Peter Naur, who proposed it for specifying the syntax of the ALGOL family of programming languages. But as noted above, the idea goes back much further, at least to Pāṇini.

Putting these rules together, the BNF specification for noun phrases is

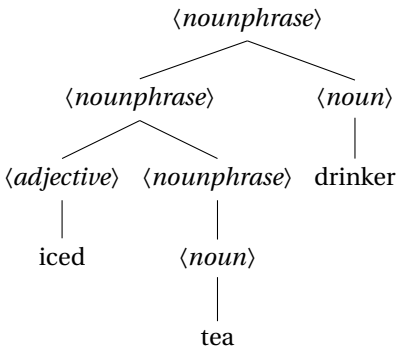
⟨*nounphrase*⟩ ::= ⟨*noun*⟩
| ⟨*adjective*⟩⟨*nounphrase*⟩
| ⟨*nounphrase*⟩⟨*noun*⟩

Here, we've rephrased the three rules as a single rule with three alternative right-hand sides. The BNF notation allows separating alternative right-hand sides with the vertical bar (|) as we have done here.

A specification of a language using rules of this sort is referred to as a GRAMMAR. According to this grammar, we can build noun phrases like *mad tea party*

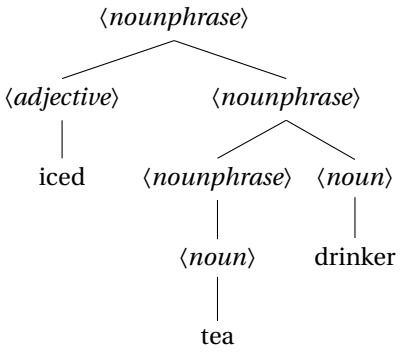


or *iced tea drinker*



Notice the difference in structure. In *mad tea party*, the adjective *mad* is combined with the phrase *tea party*, but in *iced tea drinker*, the adjective *iced* does not combine with *tea drinker*. The drinker isn't iced; the tea is!

But these same rules *can* also be used to build an alternative tree for "iced tea drinker":



The expression *iced tea drinker* is AMBIGUOUS (as is *mad tea party*); the trees make clear the two syntactic analyses.

Importantly, as shown by these examples, it is the syntactic tree structures that dictate what the expression means. The first tree seems to describe a drinker of cold beverages, the second a cold drinker of beverages. The syntactic structure of an utterance thus plays a crucial role in its meaning. The characterization of the meanings of expressions on the basis of their structure is the realm of SEMANTICS, pertinent to both natural and programming languages. We'll come back to the issue of semantics in detail in Chapters 13 and 19.

Exercise 3

Draw a second tree structure for the phrase *mad tea party*, thereby demonstrating that it is also ambiguous.

Exercise 4

How many trees can you draw for the noun phrase *flying purple people eater*? Keep in mind that *flying* and *purple* are adjectives and *people* and *eater* are nouns.

The English language, and all natural languages, are ambiguous that way. Fortunately, context, intonation, and other clues disambiguate

these ambiguous constructions so that we are mostly unaware of the ambiguities.[1] In the case of the mad tea party, we understand the phrase as having the syntactic structure as displayed above (as opposed to the one referred to in Exercise 3).

[1] The rare exceptions where ambiguities are brought to our attention account for the humor (of a sort) found in syntactically ambiguous sentences, as in the old joke that begins "I shot an elephant in my pajamas."

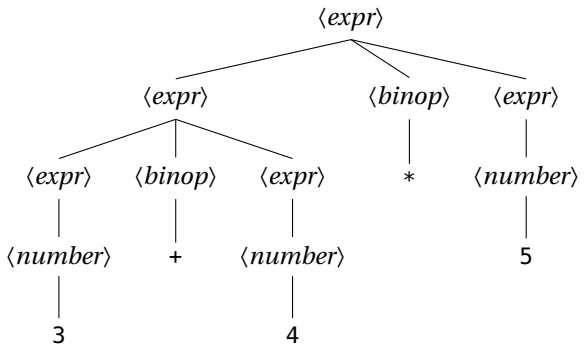
3.2 *Disambiguating ambiguous expressions*

Programming language expressions, like the utterances of natural language, have syntactic structure as well. Without some care, programming languages might be ambiguous too. Consider the following BNF rules for simple arithmetic expressions built out of numbers and BINARY OPERATORS (operators, like `+`, `-`, `*`, and `/`, that take two arguments).[2]

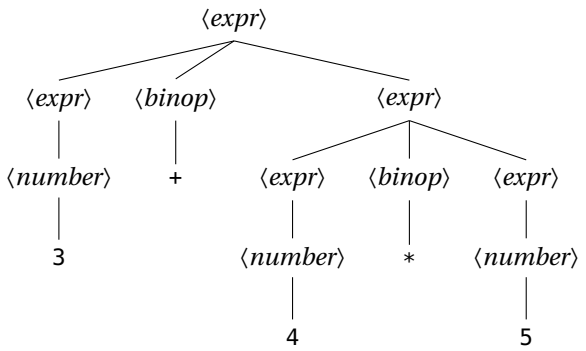
[2] In defining expression classes using this notation, we use subscripts to differentiate among different occurrences of the same expression class, such as the two ⟨*expr*⟩ instances ⟨$expr_{left}$⟩ and ⟨$expr_{right}$⟩ in the first BNF rule.

$$\begin{aligned}
\langle expr \rangle &::= \langle expr_{left} \rangle \, \langle binop \rangle \, \langle expr_{right} \rangle \\
&\quad \mid \langle number \rangle \\
\langle binop \rangle &::= \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\langle number \rangle &::= \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \cdots
\end{aligned}$$

Using these rules, we can build two trees for the expression `3 + 4 * 5`:



or



But in the case of programming languages, we don't have the luxury of access to intonation or shared context to disambiguate expressions. Instead, we rely on other tools – *conventions* and *annotations*.

In the way of conventions, we rely on a conventional ORDER OF OPERATIONS that dictates which operations we tend to do "first", that is, lower in the tree. We refer to this kind of priority of operators as their PRECEDENCE, with higher precedence operators appearing lower in the tree than lower precedence operators. By convention, we take the additive operators (`+` and `-`) to have lower precedence than the multiplicative operators (`*`, `/`). Thus, the expression `3 + 4 * 5` has the structure shown in the second tree, not the one shown in the first. For that reason, it expresses the value `23` and not `35`.

Precedence is not sufficient to disambiguate, for instance, expressions with two binary operators of the same precedence. Precedence alone doesn't disambiguate the structure of `5 - 4 - 1`: Is it `(5 - 4) - 1`, that is, `0`, or `5 - (4 - 1)`, that is, `2`. Here, we rely on the ASSOCIATIVITY of an operator. We say that subtraction, by convention, is LEFT ASSOCIATIVE, so that the operations are applied starting with the left one. The grouping is `(5 - 4) - 1`. Other operators, such as OCaml's exponentiation operator `**` are RIGHT ASSOCIATIVE, so that `2. ** 2. ** 3.` is disambiguated as `2. ** (2. ** 3.)`. Its value is `256.`, not `64.`.[3]

Associativity and precedence conventions go a long way in picking out the abstract structure of concrete expressions. But what if we want to override those conventions? What if, say, we want to express the left-branching tree for `3 + 4 * 5`? We can use annotations, as indeed, we already have, to enforce a particular structure. This is the role of PARENTHESES, to override conventional rules for disambiguating expressions. In the case at hand, we write `(3 + 4) * 5` to obtain the left-branching tree.

Exercise 5

What is the structure of the following OCaml expressions? Draw the corresponding tree so that it reflects the actual precedences and associativities of OCaml. Then, try typing the expressions into the REPL to verify that they are interpreted according to the structure you drew.

1. `10 / 5 / 2`
2. `5. +. 4. ** 3. /. 2.`
3. `(5. +. 4.) ** (3. /. 2.)`
4. `1 - 2 - 3 - 4`

You may have been taught this kind of rule under the mnemonic PEMDAS. But the ideas of precedence, associativity, and annotation are quite a bit broader than the particulars of the PEMDAS convention. They are useful in thinking more generally about the relationship between what we will call concrete syntax and abstract syntax.

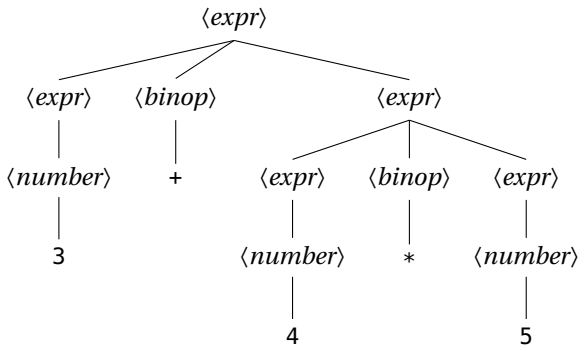
[3] The `**` operator applies to and returns floating point values, hence the decimal point dots in the arguments and return values.

For a more complete presentation of the precedences and associativities of all of the built-in operators of OCaml, see the documentation on OCaml's operators.

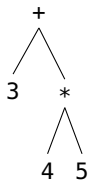
3.3 *Abstract and concrete syntax*

The right way to think of expressions, then, is as hierarchically structured objects, which we have been depicting with trees as specified by BNF grammar rules. From a practical perspective, however, when programming, we are forced to notate these expressions in an unstructured linear form as a sequence of characters, in order to enter them into a computer. We use the term ABSTRACT SYNTAX for expressions *qua* structured objects, and CONCRETE SYNTAX for their linear-notated manifestations.

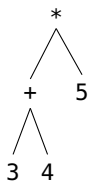
In order to more directly present the abstract syntax that corresponds to a concrete expression, we draw trees as above that depict the structure.[4] So, for instance, the concrete syntax expression `3 + 4 * 5` corresponds to the ABSTRACT SYNTAX TREE



We might abbreviate the tree structure to highlight the important aspects by eliding the expression classes as



Then the alternative abstract syntax tree



would correspond to the concrete syntax `(3 + 4) * 5`. Parentheses as used for grouping are therefore notions of concrete syntax, not abstract syntax. Similarly, conventions of precedence and associativity have to do with the interpretation of concrete syntax, as opposed to abstract syntax.

[4] The trees shown in Section 3.1, and those shown below, provide more detail than necessary for capturing the structure of the concrete expressions. For that reason, they are, strictly speaking, more like PARSE TREES, rather than abstract syntax trees. (The abbreviated versions introduced below get more to the point of true abstract syntax trees.) But since these parse trees capture structure that the concrete linear forms do not, they will serve our purposes. A good course in programming languages will more precisely distinguish parse trees that structure the concrete syntax from abstract syntax trees.

In fact, there are multiple concrete syntax expressions for this abstract syntax, such as `(3 + 4) * 5`, `((3 + 4) * 5)`, `(3 + ((4))) * 5`. But certain expressions that may seem related do not have this same abstract syntax: `5 * (3 + 4)` or `((4 + 3) * 5)` or `(3 + 4 + 0) * 5`. Although these expressions specify the same value, they do so in syntactically distinct ways. The fact that multiplication and addition are commutative, or that 0 is an additive identity – these are *semantic* properties, not syntactic.

Exercise 6

Draw the (abbreviated) abstract syntax tree for each of the following concrete syntax expressions. Assume the further BNF rule

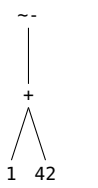
⟨*expr*⟩ ::= ⟨*unop*⟩⟨*expr*⟩

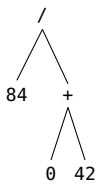
for unary operators like `~-`, the unary negation operator.

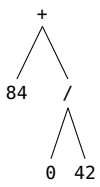
1. `(~- 4) + 6`
2. `~- (4 + 6)`
3. `20 / ~- 4 + 6`
4. `5 * (3 + 4)`
5. `((4 + 3) * 5)`
6. `(3 + 4 + 0) * 5`

Exercise 7

What concrete syntax corresponds to the following abstract syntax trees? Show as many as you'd like.

1.


2.


3.


3.4 Expressing your intentions

It is through the expressions of a programming language – structured as abstract syntax and notated through concrete syntax – that programmers express their intentions to a computer. The computer interprets the expressions in order to carry out those intentions.

Programming is an expressive activity with multiple audiences. Of course, the computer is one audience; a program allows for programmers to express their computational intentions to the computer. But there are human audiences as well. Programs can be used to communicate to other people – those who might be interested in an algorithm for its own sake, or those who are tasked with testing, deploying, or maintaining the programs. One of these latter programmers might even be the future self of the author of the original code. Weeks or even days after writing some code, you might well have already forgotten why you wrote the code a certain way. The following fundamental principle thus follows:

Edict of intention:
Make your intentions clear.

Programmers make mistakes. If their intentions are well expressed, other programmers reviewing the code can notice that those intentions are inconsistent with the code. Even the computer interpreting the program can itself take appropriate action, notifying the programmer with a useful error or warning before the code is executed and the unintended behavior can manifest itself.

Over the next chapters, we'll see many ways that the edict of intention is applied. One of the most fundamental is through documentation of code.

3.4.1 Commenting

One of the most valuable aspects of the concrete syntax of any programming language is the facility to provide elements in a concrete program that have *no correspondence whatsoever* in the abstract syntax, and therefore no effect on the computation expressed by the program. The audience for such COMMENTS is the population of human readers of the program. Comments serve the crucial expressive purpose of documenting the intended workings of a program for those human readers.

In OCaml, comments are marked by surrounding them with special delimiters: `(* ⟨⟩ *)`.[5] The primary purpose of comments is satisfying the edict of intention. Comments should therefore describe the *why* rather than the *how* of a program. Section C.2 presents some useful stylistic considerations in providing comments for documenting programs.

There are other aspects of concrete syntax that can be freely deployed because they have no affect on the computation that a program carries out. These too can be judiciously deployed to help express your

[5] We use the symbol ⟨⟩ here and throughout the later chapters as a convenient notation to indicate unspecified text of some sort, a textual anonymous variable of a sort. Here, it stands in for the text that forms the comment. In other contexts it stands in for the arguments of an operator, constructor, or subexpression, for instance, in ⟨⟩ `+` ⟨⟩ or ⟨⟩ `list` or `let` ⟨⟩ `in` ⟨⟩.

intentions. For instance, the particular spacing used in laying out the elements of a program doesn't affect the computation that the program expresses. Spaces, newlines, and indentations can therefore be used to make your intentions clearer to a reader of the code, by laying out the code in a way that emphasizes its structure or internal patterns. Similarly, the choice of variable names is completely up to the programmer. Names can be consistently renamed without affecting the computation. Programmers can take advantage of this fact by choosing names that make clear their intended use.

❧

Having clarified these aspects of the syntactic structure of programming languages (and OCaml in particular) – distinguishing concrete and abstract syntax; presenting precedence, associativity, and parenthesization for disambiguation – we turn now to begin the discussion of OCaml as a language of types and values.