1 • Semantics via Syntax

The specification of a programming language starts with its syntax. As every programmer knows, the syntax of a language comes in the shape of a variant of a BNF (Backus-Naur Form) grammar, which enumerates the grammatically legal vocabulary and sentences. The difficult part is the specification of the meaning of programs, i.e., how a program computes.

In this first part of the book, we develop a syntax-based approach to the specification of semantics. We start with the observation that computation generalizes calculations and that a child's training in calculations starts with phrases such as "1 plus 1 is 2." The trick is to see that such forms of calculation also apply to programs.

Calculating with programs means looking at the syntax of an expression or a statement and relating it to some other—presumably simpler—expression or statement. For the expression $1 + 1$, this claim is easy to understand; it is equal to 2, i.e., $1 + 1$ is related to 2. Even the application of a function to argument values can be expressed in this manner:

$$f(4) = 2 * 4 + 55 \qquad \text{if} \quad \textbf{define}\ f(x) = 2 * x + 55.$$

Mathematically put, we are specifying *the semantics of a programming language with a relation on its syntax.* For students of functional programming languages, this claim does not come as a surprise. They know that functional programming is little more than seventh grade algebra turned into a programming language, and the laws of algebra are equations that relate algebraic expressions to each other. What may come as a surprise is the possibility of specifying the semantics of (almost) all programming languages in this manner, including languages with imperative effects.

Here we introduce the idea, the minimal mathematical meta-knowledge, starting with syntax definitions as sets.

1·1 **Defining Sets.**

A BNF grammar is used for several different purposes. One meaning is a set of strings. Another interpretation is a set of "trees," often called **abstract syntax** (trees). In this book, we always adopt the second perspective.

For this chapter and the next, we use the following BNF grammar as a running example:

$$\boxed{\begin{array}{rcl} B & = & \mathtt{t} \\ & | & \mathtt{f} \\ & | & (B \bullet B) \end{array}}$$

We take it as a short-hand for following constraints on the set B of abstract syntax trees:

$$\begin{array}{c} \mathtt{t} \in B \\ \mathtt{f} \in B \\ a \in B \text{ and } b \in B \;\Rightarrow\; (a \bullet b) \in B \end{array}$$

Technically, B is the *smallest* set that obeys the above constraints. To construct this set, we first include the basic elements t and f, and then inductively combine those into compound elements.

Notation: we sometimes use “B” to mean “the set B,” but sometimes “B” will mean “an arbitrary element of B.” The meaning is always clear from the context. Sometimes, we use a subscript or a prime on the name of the set to indicate an arbitrary element of the set, such as “B_1” or “B'.” Thus, the above constraints might be written as

$$\begin{array}{cl} \mathtt{t} \in B & \text{[a]} \\ \mathtt{f} \in B & \text{[b]} \\ (B_1 \bullet B_2) \in B & \text{[c]} \end{array}$$

Enumerating all of the elements of B in finite space is clearly impossible in this case:

$$B = \{\mathtt{t},\ \mathtt{f},\ (\mathtt{t} \bullet \mathtt{t}),\ (\mathtt{t} \bullet \mathtt{f}),\ \ldots\}$$

Given some tree, however, we can demonstrate that it belongs to B by showing that it satisfies the constraints. For example, $(\mathtt{t} \bullet (\mathtt{f} \bullet \mathtt{t}))$ is in B:

$$\begin{array}{lll} 1. & \mathtt{t} \in B & \text{by [a]} \\ & & \\ 2. & \mathtt{f} \in B & \text{by [b]} \\ 3. & \mathtt{t} \in B & \text{by [a]} \\ 4. & (\mathtt{f} \bullet \mathtt{t}) \in B & \text{by 2, 3, and [c]} \\ & & \\ 5. & (\mathtt{t} \bullet (\mathtt{f} \bullet \mathtt{t})) \in B & \text{by 1, 4, and [c]} \end{array}$$

Usually, such arguments are arranged in the shape of a so-called proof tree:

$$\dfrac{\texttt{t} \in B \text{ [a]} \qquad \dfrac{\texttt{f} \in B \text{ [b]} \quad \texttt{t} \in B \text{ [a]}}{(\texttt{f} \bullet \texttt{t}) \in B}\text{ [c]}}{(\texttt{t} \bullet (\texttt{f} \bullet \texttt{t})) \in B}\text{ [c]}$$

Most of the time, proof trees come without labels that justify each step, because the steps are typically obvious:

$$\dfrac{\texttt{t} \in B \qquad \dfrac{\texttt{f} \in B \quad \texttt{t} \in B}{(\texttt{f} \bullet \texttt{t}) \in B}}{(\texttt{t} \bullet (\texttt{f} \bullet \texttt{t})) \in B}$$

Exercise 1.1. Which of (1) $\texttt{t}$, (2) $\bullet$, (3)$((\texttt{f} \bullet \texttt{t}) \bullet (\texttt{f} \bullet \texttt{f}))$, (4) $((\texttt{f}) \bullet (\texttt{t}))$, or (5) “hello” are in B? For each member of B, provide a proof tree.

1·2 Relations.

A **relation** is a set whose elements consist of ordered pairs. For example, we can define the $\mathbf{R}$ relation to match each element of B with itself:

$$a \in B \quad \Rightarrow \quad \langle a, a \rangle \in \mathbf{R}$$

For binary relations such as $\mathbf{R}$, instead of $\langle a, a \rangle \in R$, we usually write $a\,\mathbf{R}\,a$:

$$a \in B \quad \Rightarrow \quad a\,\mathbf{R}\,a$$

or even simply

$$B_1\,\mathbf{R}\,B_1$$

as long as it is understood as a definition of $\mathbf{R}$. As it turns out, the relation $\mathbf{R}$ is reflexive, symmetric, and transitive; that is, it satisfies the following three constraints:

a relation $\mathbf{R}$ is **reflexive**	iff $a\,\mathbf{R}\,a$ (for any a)
a relation $\mathbf{R}$ is **symmetric**	iff $a\,\mathbf{R}\,b \Rightarrow b\,\mathbf{R}\,a$
a relation $\mathbf{R}$ is **transitive**	iff $a\,\mathbf{R}\,b$ and $b\,\mathbf{R}\,c \Rightarrow a\,\mathbf{R}\,c$

If a relation is reflexive, symmetric, and transitive, then it is an **equivalence**. Certain names for a relation, such as $=$, suggest that the relation is an equivalence.

The following defines a relation $\mathbf{r}$ that is neither reflexive, symmetric, nor transitive.

$$\boxed{\begin{array}{llll} (\texttt{f} \bullet B_1) & \mathbf{r} & B_1 & \text{[a]} \\ (\texttt{t} \bullet B_1) & \mathbf{r} & \texttt{t} & \text{[b]} \end{array}}$$

In the context of a reduction semantics, such relations are known as **notions of reduction.** A minor modification of this definition yields a relation $\asymp_{\mathbf{r}}$ that is reflexive:

$$\begin{array}{rcll} (\mathtt{f} \bullet B_1) & \asymp_{\mathbf{r}} & B_1 & [\mathsf{a}] \\ (\mathtt{t} \bullet B_1) & \asymp_{\mathbf{r}} & \mathtt{t} & [\mathsf{b}] \\ B_1 & \asymp_{\mathbf{r}} & B_1 & [\mathsf{c}] \end{array}$$

An alternative way of defining $\asymp_{\mathbf{r}}$ is to extend $\mathbf{r}$ and explicitly constrain the new relation to be reflexive:

$$\begin{array}{lcll} B_1 & \asymp_{\mathbf{r}} & B_2 \text{ if } B_1 \,\mathbf{r}\, B_2 & [\mathsf{ab}] \\ B_1 & \asymp_{\mathbf{r}} & B_1 & [\mathsf{c}] \end{array}$$

The relation $\asymp_{\mathbf{r}}$ is therefore called the **reflexive closure** of the $\mathbf{r}$ relation. We define yet another relation by adding symmetry and transitivity constraints:

$$\begin{array}{lcll} B_1 & \approx_{\mathbf{r}} & B_2 \text{ if } B_1 \,\mathbf{r}\, B_2 & [\mathsf{ab}] \\ B_1 & \approx_{\mathbf{r}} & B_1 & [\mathsf{c}] \\ B_2 & \approx_{\mathbf{r}} & B_1 \text{ if } B_1 \approx_{\mathbf{r}} B_2 & [\mathsf{d}] \\ B_1 & \approx_{\mathbf{r}} & B_3 \text{ if } B_1 \approx_{\mathbf{r}} B_2 & [\mathsf{e}] \\ & & \quad \text{and } B_2 \approx_{\mathbf{r}} B_3 & \end{array}$$

The $\approx_{\mathbf{r}}$ relation is the **symmetric-transitive closure** of $\asymp_{\mathbf{r}}$, and it is the **reflexive-symmetric-transitive closure** or **equivalence closure** of $\mathbf{r}$.

1·3 Semantics as an Equivalence Relation.

The running example of B and $\mathbf{r}$ suggests how a programming language can be defined through syntax and relations on syntax—or, more specifically, as a set B of abstract syntax trees and a relation $\mathbf{r}$ on this set. In fact, an alert reader might begin to suspect that B is a grammar for boolean expressions with t for true, f for false, and • as the "or" operator. The relation $\approx_{\mathbf{r}}$ equates pairs of B expressions that have the same (boolean) value.

Indeed, using the constraints above, we can show that $(\mathtt{f} \bullet \mathtt{t}) \approx_{\mathbf{r}} (\mathtt{t} \bullet \mathtt{t})$, just as $\mathsf{false} \vee \mathsf{true} = \mathsf{true} \vee \mathsf{true}$:

$$\dfrac{(\mathtt{f} \bullet \mathtt{t}) \approx_{\mathbf{r}} \mathtt{t}\ [\mathsf{a}] \qquad \dfrac{(\mathtt{t} \bullet \mathtt{t}) \approx_{\mathbf{r}} \mathtt{t}\ [\mathsf{b}]}{\mathtt{t} \approx_{\mathbf{r}} (\mathtt{t} \bullet \mathtt{t})}\,[\mathsf{d}]}{(\mathtt{f} \bullet \mathtt{t}) \approx_{\mathbf{r}} (\mathtt{t} \bullet \mathtt{t})}\,[\mathsf{e}]$$

It does not follow, however, that • is exactly like a boolean "or". If we wished to establish this connection, we would have to prove general claims about •, such as $(B_1 \bullet \mathtt{t}) \approx_{\mathbf{r}} \mathtt{t}$ for any expression B_1.

Put differently, there is generally a gap between the semantics of a programming language and properties of this semantics language that we might like to know. For various purposes, the properties of a semantics are as important as the values it relates to expressions or programs. For example, if $\bullet$ really satisfied the laws of "or", then a compiler might safely optimize $(B_1 \bullet \mathtt{t})$ as $\mathtt{t}$. Similarly, if the semantics of the language guaranteed that a number can never be added to anything other than another number, the implementation of the semantics need not check the arguments of an addition operation to ensure that they are numbers.

1·4 **Semantics via Reduction.**

The $\approx_{\mathbf{r}}$ relation should remind the reader of $=$ from arithmetic and algebra in primary school. Just like we teach students in this setting to use such equational reasoning for all kinds of purposes, we can use the $\approx_{\mathbf{r}}$ relation to prove the equivalence of certain expressions. In general, though, the relation does not suggest how to get from an arbitrary B to either $\mathtt{t}$ or $\mathtt{f}$—which is what we really need to build an interpreter for a semantics.

In this sense, the $\mathbf{r}$ relation is actually more useful than $\approx_{\mathbf{r}}$. Both cases in the definition of $\mathbf{r}$ relate an expression to a smaller expression. Also, for any expression B, either B is $\mathtt{t}$ or $\mathtt{f}$, or $\mathbf{r}$ relates B to at most one other expression. As a result, we can think of $\mathbf{r}$ as a **single-step reduction**, corresponding to the way that an interpreter might take a single evaluation step in working towards a final value.

Using $\mathbf{r}$, it is then possible to define $\twoheadrightarrow_{\mathbf{r}}$ as the reflexive-transitive closure of $\mathbf{r}$:

$$\begin{array}{lll} B_1 & \twoheadrightarrow_{\mathbf{r}} & B_1 \\ B_1 & \twoheadrightarrow_{\mathbf{r}} & B_2 \ \text{ if } \ B_1 \,\mathbf{r}\, B_2 \\ B_1 & \twoheadrightarrow_{\mathbf{r}} & B_2 \ \text{ if } \ B_1 \twoheadrightarrow_{\mathbf{r}} B_3 \text{ and } B_3 \twoheadrightarrow_{\mathbf{r}} B_2 \end{array}$$

This yields a **multi-step reduction** relation. In particular, the multi-step relation $\twoheadrightarrow_{\mathbf{r}}$ maps a single expression to many other expressions but to at most one of $\mathtt{t}$ or $\mathtt{f}$.

The relations $\mathbf{r}$ and $\twoheadrightarrow_{\mathbf{r}}$ are intentionally asymmetric, emphasizing that evaluation should proceed in a specific direction towards a value. For example, given the expression $(\mathtt{f} \bullet (\mathtt{f} \bullet (\mathtt{t} \bullet \mathtt{f})))$, we can show that there exits a sequence of **reduction**s from it to $\mathtt{t}$:

$$\begin{array}{lll} (\mathtt{f} \bullet (\mathtt{f} \bullet (\mathtt{t} \bullet \mathtt{f}))) & \mathbf{r} & (\mathtt{f} \bullet (\mathtt{t} \bullet \mathtt{f})) \\ & \mathbf{r} & (\mathtt{t} \bullet \mathtt{f}) \\ & \mathbf{r} & \mathtt{t} \end{array}$$

Each blank line in the left column is implicitly filled by the expression in the right column from the previous line. Each line is then a step in an argument that $(\mathtt{f} \bullet (\mathtt{f} \bullet (\mathtt{t} \bullet \mathtt{f}))) \twoheadrightarrow_{\mathbf{r}} \mathtt{t}$.

Exercise 1.2. Show that $(\mathtt{f} \bullet (\mathtt{f} \bullet (\mathtt{f} \bullet \mathtt{f}))) \rightsquigarrow\!\!\!\!\rightarrow_{\mathbf{r}} \mathtt{f}$ by constructing a reduction sequence based on the $\mathbf{r}$ one-step relation.

1·5 Reduction in Context.

How does the expression $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f})$ reduce? According to $\mathbf{r}$ or $\rightsquigarrow\!\!\!\!\rightarrow_{\mathbf{r}}$, it does not reduce at all. Intuitively, $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f})$ should reduce to $(\mathtt{t} \bullet \mathtt{f})$, by simplifying the first sub-expression according to $(\mathtt{f} \bullet \mathtt{t})\,\mathbf{r}\,\mathtt{t}$. Nothing in the definition of $\mathbf{r}$ matches $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f})$ as the source expression, however. That is, we can only reduce expressions of the form $(\mathtt{f} \bullet B)$ and $(\mathtt{t} \bullet B)$. While the expression on the right-hand side of the outermost $\bullet$ can be arbitrary, the expression on the left-hand side must be $\mathtt{f}$ or $\mathtt{t}$.

If we wish to reduce such B expressions to answers, we must extend the $\mathbf{r}$ relation to a relation that supports the reduction of sub-expressions.

$$\boxed{\begin{array}{rcllr} B_1 & \rightarrow_{\mathbf{r}} & B_2 & \text{if } B_1\,\mathbf{r}\,B_2 & [\mathsf{a}] \\ (B_1 \bullet B_2) & \rightarrow_{\mathbf{r}} & (B_1' \bullet B_2) & \text{if } B_1 \rightarrow_{\mathbf{r}} B_1' & [\mathsf{b}] \\ (B_1 \bullet B_2) & \rightarrow_{\mathbf{r}} & (B_1 \bullet B_2') & \text{if } B_2 \rightarrow_{\mathbf{r}} B_2' & [\mathsf{c}] \end{array}}$$

The $\rightarrow_{\mathbf{r}}$ relation is the **compatible closure** of the $\mathbf{r}$ relation. Like $\mathbf{r}$, $\rightarrow_{\mathbf{r}}$ is a single-step reduction relation, but $\rightarrow_{\mathbf{r}}$ allows the reduction of any sub-expression within an expression. The reducible expression is called the **redex**, and the text surrounding a redex is its **context**.

In particular, the $\rightarrow_{\mathbf{r}}$ relation includes $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) \rightarrow_{\mathbf{r}} (\mathtt{t} \bullet \mathtt{f})$. We can demonstrate this inclusion with the following proof tree:

$$\dfrac{\dfrac{(\mathtt{f} \bullet \mathtt{t})\,\mathbf{r}\,\mathtt{t}}{(\mathtt{f} \bullet \mathtt{t}) \rightarrow_{\mathbf{r}} \mathtt{t}}\,[\mathsf{a}]}{((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) \rightarrow_{\mathbf{r}} (\mathtt{t} \bullet \mathtt{f})}\,[\mathsf{b}]$$

Continuing with $\rightarrow_{\mathbf{r}}$, we can reduce $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f})$ to $\mathtt{t}$:

$$\begin{array}{rcl} ((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) & \rightarrow_{\mathbf{r}} & (\mathtt{t} \bullet \mathtt{f}) \\ & \rightarrow_{\mathbf{r}} & \mathtt{t} \end{array}$$

Finally, if we define $\twoheadrightarrow_{\mathbf{r}}$ to be the reflexive–transitive closure of $\rightarrow_{\mathbf{r}}$, then we get $((\mathtt{f} \bullet \mathtt{t}) \bullet \mathtt{f}) \twoheadrightarrow_{\mathbf{r}} \mathtt{t}$. Thus, $\twoheadrightarrow_{\mathbf{r}}$ is the natural **reduction relation** generated by $\mathbf{r}$.

In general, the mere reflexive closure $\asymp_{\mathbf{r}}$, equivalence closure $\approx_{\mathbf{r}}$, or reflexive-transitive closure $\rightsquigarrow\!\!\!\!\rightarrow_{\mathbf{r}}$ of a relation $\mathbf{r}$ is uninteresting. What we are most often interested in is the compatible closure $\rightarrow_{\mathbf{r}}$ and its reflexive-transitive closure $\twoheadrightarrow_{\mathbf{r}}$. Those two correspond to typical notions of expression evaluation and interpretation. In addition, the equivalence closure $=_{\mathbf{r}}$ of $\rightarrow_{\mathbf{r}}$ is interesting because it relates expressions that produce the same result.

Exercise 1.3. Explain why $(\texttt{f} \bullet ((\texttt{t} \bullet \texttt{f}) \bullet \texttt{f})) \not\twoheadrightarrow_{\mathbf{r}} \texttt{t}$.

Exercise 1.4. Show that $(\texttt{f} \bullet ((\texttt{t} \bullet \texttt{f}) \bullet \texttt{f})) \twoheadrightarrow_{\mathbf{r}} \texttt{t}$ with a reduction sequence based on $\rightarrow_{\mathbf{r}}$.

1·6 **Evaluation Functions.**

The $\twoheadrightarrow_{\mathbf{r}}$ relation brings us close to a useful notion of evaluation, but we are not there yet. While $((\texttt{f} \bullet \texttt{t}) \bullet \texttt{f}) \twoheadrightarrow_{\mathbf{r}} \texttt{t}$, it is also the case that $((\texttt{f} \bullet \texttt{t}) \bullet \texttt{f}) \twoheadrightarrow_{\mathbf{r}} (\texttt{t} \bullet \texttt{f})$ and $((\texttt{f} \bullet \texttt{t}) \bullet \texttt{f}) \twoheadrightarrow_{\mathbf{r}} ((\texttt{f} \bullet \texttt{t}) \bullet \texttt{f})$. For an evaluator, however, we are only interested in whether a B evaluates to a result and a result is either f or to t; anything else is irrelevant.

We use two definitions to state this desire formally. The first specifies once and for all what we consider a result R for B "programs."

$$\boxed{\begin{array}{rcl} R & = & \texttt{t} \\ & | & \texttt{f} \end{array}}$$

Obviously, R is a subset of B ($R \subset B$) because all results are also expressions of our "programming language."

Our second definition specifies evaluation as the $\mathit{eval}_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}}$ relation, which maps each expression to a result.

$$\boxed{\begin{array}{rcl} \mathit{eval}_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}} : B & \longrightarrow & R \\ \mathit{eval}_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}}(B) & = & \begin{cases} \texttt{f} & \text{if } B \twoheadrightarrow_{\mathbf{r}} \texttt{f} \\ \texttt{t} & \text{if } B \twoheadrightarrow_{\mathbf{r}} \texttt{t} \end{cases} \end{array}}$$

Here, we're using yet another notation to define a relation. This particular notation is suggestive of a **function**, i.e., a relation that maps each element to at most one element. We use the function notation because $\mathit{eval}_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}}$ must be a function if it is going to make sense as an evaluator (for a deterministic programming language).

The name of the function comes with both a subscript and a superscript. Naturally, the former just says that the function is based on the relation $\mathbf{r}$, while the latter tells us that the definition of the evaluation function is based on the relation $\twoheadrightarrow_{\mathbf{r}}$. We decorate the name of the function with both because there are many alternative definitions. For example, the following definition uses the equivalence relation based on $\mathbf{r}$ instead of the reduction relation.

$$\boxed{\mathit{eval}_{\mathbf{r}}^{=_{\mathbf{r}}}(B) = \begin{cases} \texttt{f} & \text{if } B =_{\mathbf{r}} \texttt{f} \\ \texttt{t} & \text{if } B =_{\mathbf{r}} \texttt{t} \end{cases}}$$

The equivalence relation $=_{\mathbf{r}}$ is of course just the compatible, reflexive, transitive and symmetric closure of $\mathbf{r}$. Defining an evaluation function via $=_{\mathbf{r}}$ shows that a program's computation really just generalizes the notion of calculation from algebra.

Exercise 1.5. Among the relations $\mathbf{r}$, $\asymp_{\mathbf{r}}$, $\approx_{\mathbf{r}}$, $\rightsquigarrow\!\!\!\twoheadrightarrow_{\mathbf{r}}$, $\rightarrow_{\mathbf{r}}$, $\twoheadrightarrow_{\mathbf{r}}$, and $=_{\mathbf{r}}$ which are functions? For each non-function relation, find an expression and two expressions that it relates to.

Exercise 1.6. Use the above definitions to find the results of $eval_{\mathbf{r}}^{=_{\mathbf{r}}}$(((f • t) • t)) and $eval_{\mathbf{r}}^{\twoheadrightarrow_{\mathbf{r}}}$(((f • t) • f))

1·7 Notation Summary.

The following table summarizes the notions and the notations introduced so far.

name	definition	intuition
$\_$	the base relation on members of an expression grammar	a single "reduction" step with no context
$\rightarrow_{\_}$	the compatible closure of $\_$ with respect to the expression grammar	a single step within a context
$\twoheadrightarrow_{\_}$	the reflexive–transitive closure of $\rightarrow_{\_}$	multiple evaluation steps (zero or more)
$=_{\_}$	the symmetric–transitive closure of $\twoheadrightarrow_{\_}$	equates expressions that produce the same result
$eval_{\_}^{\_}$	a relation projected to a range (results)	complete evaluation based on $\twoheadrightarrow_{\_}$ or $=_{\_}$
$eval_{\_}$	a generic *eval* relation	