Inductive definitions and proofs

June 3, 2025

1 Inductively defined sets

We represent inductively defined sets in the form of a set of inference rules. Suppose we want to define the set S, each inference rule takes the following general form

$$\frac{P_0 \qquad P_1 \qquad \dots \qquad P_n}{a \in S}\, R_0$$

Each premise P_i above the horizontal line is a logical formula that either does not involve S at all or a logical formula of the form $b \in S$ where S does not appear in b.

Each inference rule can be read as an implication that describes the elements that inhabit the set S. Rule R_0, for example, can be read as the proposition $(P_0 \wedge P_1 \dots \wedge P_n) \to a \in S$. When the premise is empty, the statement simply states that the conclusion is true. Note that shuffling the order of $P_0 \dots P_n$ doesn't change the definition of the set S since the underlying proposition should remain equivalent.

Variables that appear free in each rule are implicitly quantified. As an example, recall the following rules we have seen in class.

$$\frac{}{0 \in S}\, R_0$$

$$\frac{n \in S}{n + 4 \in S}\, R_4$$

$$\frac{n \in S}{n + 6 \in S}\, R_6$$

Here, the proposition corresponding to rule R_4 is $\forall n \in S, n + 4 \in S$ where the variable n is universally quantified.

An element a is in the set S if and only if there exists a proof of $a \in S$ using the rules. While it's possible to describe the proofs in English, we can write proofs in the form of *derivation trees*, a formal mathematical object. Here is an example of a derivation tree that shows $10 \in S$.

$$\dfrac{\dfrac{\dfrac{}{0 \in S}\,R_0}{4 \in S}\,R_4}{10 \in S}\,R_6$$

The derivation tree for $10 \in S$ looks more like a sequence than a tree because the rules for the set S only contain at most one premise.

Consider the following inductive definition of the set *beautiful*.

$$\dfrac{}{0 \in \textit{beautiful}}\,B_0$$

$$\dfrac{}{3 \in \textit{beautiful}}\,B_3$$

$$\dfrac{}{5 \in \textit{beautiful}}\,B_5$$

$$\dfrac{n \in \textit{beautiful} \qquad m \in \textit{beautiful}}{m + n \in \textit{beautiful}}\,B_n$$

Here is a derivation tree for $11 \in$ *beautiful*.

$$\dfrac{\dfrac{\dfrac{}{3 \in \textit{beautiful}}\,B_3 \quad \dfrac{}{3 \in \textit{beautiful}}\,B_3}{6 \in \textit{beautiful}}\,B_n \qquad \dfrac{}{5 \in \textit{beautiful}}\,B_5}{11 \in \textit{beautiful}}\,B_n$$

Since B_n has two premises, invoking B_n to prove $11 \in$ *beautiful* requires us to provide two subproofs/subtrees $6 \in$ *beautiful* and $5 \in$ *beautiful*.

Exercise 1. Write a derivation tree for $9 \in$ *beautiful* using only rules B_3 and B_n.

Exercise 2. Write a derivation tree for $9 \in$ *beautiful* that involves at least one usage of rule B_0.

2 Inductive proofs

Given a set S defined by some inference rules, rule induction says that to show that S is a subset of some set R, it suffices to show for each inference rule defining S, the proposition corresponds to the inference rule after replacing S by R holds.

For example, given some set R, to prove that *beautiful* $\in R$, it suffices to show that the propositions correspond to the following rules hold:

$$\frac{}{0 \in R}\ B_0$$

$$\frac{}{3 \in R}\ B_3$$

$$\frac{}{5 \in R}\ B_5$$

$$\frac{n \in R \qquad m \in R}{m + n \in R}\ B_n$$

Thus, to show that $S \subseteq R$, the rule of induction says it is sufficient to prove the following statements.

- $0 \in R$
- $3 \in R$
- $5 \in R$
- $\forall n\, m, n \in R \wedge m \in R \implies m + n \in R$

Exercise 3. Prove by induction that S only contains natural numbers. For each rule, explicitly write down the statement you need to prove and then show why it's true.

Exercise 4. Try proving that S only contains odd numbers, which is a false statement as $3 + 3 = 6 \in S$. Again, for each rule, write down the statement you need to prove. Which rule fails to hold?

Suppose we want to prove the statement $\forall a \in S, P(a)$ where P is a predicate over objects. We can prove the statement through the induction principle by instantiate R with the set $\{a \mid P(a)\}$. Thus, to prove $\forall a \in S, P(a)$, it suffices to show the following statements.

- $P(0)$ is true.
- $P(3)$ is true.
- $P(5)$ is true.
- $\forall n\, m, P(n) \wedge P(m) \implies P(m + n)$ is true.

For example, to prove that all elements in *beautiful* are linear combinations of 3 and 5, we can instantiate P with $P(a) := \exists m\ n \in \mathbb{N}, a = 3m + 5n$. Then by induction, it suffices to show that the following statements hold.

- 0 is a linear combination of 3 and 5.
- 3 is a linear combination of 3 and 5.
- 5 is a linear combination of 3 and 5.
- If n and m are both linear combinations of 3 and 5, then $m + n$ is a linear combination of 3 and 5.

Don't forget that we are not done yet! The above process helps us find what needs to be proven by invoking the induction principle. We still need to check that all the propositions hold.

As you get more familiar with inductive proofs, you should be able to perform the rewriting from *beautiful* to R in your head and directly prove the statement that corresponds to each rule. With more complicated definitions, however, it is sometimes useful to explicitly write down the induction principle.

Of course, so far we are only talking about how to obtain the induction principle, but we never asked *why* this style of reasoning is correct. Justifying the validity of induction is a topic we will cover later in class.

3 Inductively defined data types

In this section, we use the knowledge developed in section 1 to define various data types that will be useful when defining languages. As we will see, inductively defined data types really are just an instance of the inductively defined sets we have already seen, except that we expect them to satisfy some constraints so we can define recursive functions over them.

3.1 Sets of Symbols

In set theory, the notion of strings or symbols is not a primitive concept. However, as we are defining programming languages, it is convenient to assume that there exists a set named *Symbol* that consist of strings from the English and Greek alphabet.

It is important to distinguish between symbols and metavariables that we can quantify over. Therefore, to avoid ambiguity, we will sometimes preceed strings from the set *Symbol* with a single quote $'$. Using this notation, we have $'\lambda,' foo,' bar,' plus,' + \in \mathit{Symbol}$. Importantly, the symbol $'+$ is really a string and is not to be confused with the mathematical function $+$ over numbers.

Here is an example of why the distinction between symbols and metavariables are important. Consider the following two rules.

$$\frac{n \in \mathbb{N}}{n \in S}\, R_n$$

$$\frac{}{'n \in S}\, R'_n$$

Rule R_n corresponds to the universally quantified statement $\forall n \in \mathbb{N}, n \in S$, which says all natural numbers n are in the set S. Rule R'_n, on the other hand, corresponds to $'n \in S$, the proposition that says S contains the single symbol $'n$.

3.2 Inductively Defined Lists and Recursive Functions

Given a set A, we inductively define the set $\mathbf{List}_A$ with the following two inference rules.

$$\frac{}{'\mathbf{Nil} \in \mathbf{List}_A}\, L_0$$

$$\frac{a \in A \qquad l \in \mathbf{List}_A}{(a, l) \in \mathbf{List}_A}\, L_1$$

Thus, an element in $\mathbf{List}_A$ should take the general form $(a_0, (a_1, \ldots, (a_n, '\mathbf{Nil})))$ where $a_0, \ldots, a_n \in A$. We write $(a_0\ a_1 \ldots\ a_n)$ as a shorthand notation for the same list to avoid nested parentheses. Note that when the list is empty, we use the notation $()$ to denote the symbol $'\mathbf{Nil}$. For example, we write $(1\ 2)$ for the 2-element list of natural numbers instead of $(1, (2, '\mathbf{Nil}))$.

Given a non-empty list $(a, l) \in \mathbf{List}_A$, applying the projection operator π_1 gives us $\pi_1(a, l) = a \in A$, the head of the list, whereas applying the projection operator π_2 gives us $\pi_2(a, l) = l$, the remainder of the list.

However, there are more interesting functions that we'd like to define over lists. For example, an operation that computes the length of a list, or a function that appends two lists. That's where recursive definitions become useful.

The principle of recursion says to define a function f from $\mathbf{List}_A$ to some set T, it suffices to provide two equations about f, one for rule L_0 and one for rule L_1.

$$f('\mathbf{Nil}) = e_0$$
$$f((a, l)) = e_1$$

The expression e_0 must not contain any reference to f. The expression e_1 can refer to its arguments a and l and $f(l)$.

Following the recursion principle, we can define the length function as follows.

$$\mathbf{length}('\mathbf{Nil}) = 0$$

$$\mathbf{length}((a, l)) = 1 + \mathbf{length}(l)$$

Exercise 5. Recall that (1 2 3) is a shorthand for $(1, (2, (3, {}'\mathbf{Nil})))$. Following the equations of the **length** function, show that $\mathbf{length}((1\ 2\ 3)) = 3$.

Exercise 6. Consider the following equations.

$$g({}'\mathbf{Nil}) = 0$$

$$g((a, l)) = 1 + g((a, l))$$

Explain why these two equations *do not* induce a valid recursive definition for a function $g : \mathbf{List}_A \to \mathbb{N}$.

In particular, explain intuitively why applying g to the singleton list (1) fail to produce a natural number.

Exercise 7. The **stutter** function takes a list as input and produces a list with double the size by duplicating each element. For example, given the list (1 2 3) as input, we want **stutter** to produce the list (1 1 2 2 3 3) as its output.

Write a recursive definition of $\mathbf{stutter} : \mathbf{List}_A \to \mathbf{List}_A$ by filling out the right hand side of the following equations.

$$\mathbf{stutter}({}'\mathbf{Nil}) = \ldots$$

$$\mathbf{stutter}((a, l)) = \ldots$$

Exercise 8. By rule induction, show that for all $a \in \mathbf{List}_A$, $\mathbf{length}(\mathbf{stutter}(a)) = 2 \cdot \mathbf{length}(a)$.

In Exercises 1 and 2, we have seen that there can be multiple derivation trees that prove $9 \in$ *beautiful*. The definition of $\mathbf{List}_A$, on the other hand, satisfies the following uniqueness property about derivation trees.

Proposition 1. For all $a \in \mathbf{List}_A$, there exists a unique derivation tree with a conclusion that $a \in \mathbf{List}_A$.

Proof. We proceed by rule induction over $\mathbf{List}_A$, which requires us to show the following two statements.

- There is a unique derivation tree for ${}'\mathbf{Nil} \in \mathbf{List}_A$.
- If $a \in A$ and there is a unique derivation tree for $l \in \mathbf{List}_A$, then there exists a unique derivation tree for $(a, l) \in \mathbf{List}_A$.

□

Exercise 9. Finish off the proof above by showing that both statements hold.

Remark. In most cases, the uniqueness derivation tree property can be quickly checked by looking at the conclusion of each rule. In the case of $\mathbf{List}_A$, ${}'\mathbf{Nil}$ and (a, l) are clearly different objects so given $a \in \mathbf{List}_A$ there will be no ambiguity which rule can fire.

The uniqueness of derivation trees is essential for the recursion principle to work. Consider, for example, the following set of recursive equations for the set *beautiful*.

$$h(0) = 1$$

$$h(3) = 1$$

$$h(5) = 1$$

$$h(m + n) = h(m) + h(n)$$

The equations do not induce a function because a single input can correspond to multiple different outputs: $h(3) = h(0) + h(3) = 1 + 1 = 2$, $h(3) = 1$.

Exercise 10. Consider the following definition of the set *Even*.

$$\frac{}{0 \in \mathit{Even}}\ E_0$$

$$\frac{n \in \mathit{Even}}{n + 2 \in \mathit{Even}}\ E_1$$

- First, show that there can be only one derivation tree for every proof of $a \in \mathit{Even}$.
- Next, write a recursive function p from *Even* to $\mathbb{N}$ that divides the input by 2.
- Finally, use rule induction to show the correctness of p by proving $\forall a \in \mathit{Even}, 2 \cdot p(a) = a$

4 An Arithmetic Language

In this section, we use the concepts we have learned in previous sections to define a simple language that describes arithmetic.

4.1 Syntax

The arithmetic language allows us to write expressions that you can write in a simple calculator:

$$4 + (9 * 7)$$

$$(3 * 3) * 3$$

However, instead of working with strings of characters, we skip the parsing step completely and define the *abstract syntax tree* (AST) of the language as an inductively defined sets. By working with ASTs, we no longer have to worry about operator precedence and parentheses while reasoning about the language. In practice, a parser is needed to turn well-formed strings representing algorithmic expressions into their corresponding ASTs.

The set **Arith** is defined by the following set of rules. For simplicity, we consider multiplication and addition as our only operators.

$$\frac{n \in \mathbb{N}}{n \in \textbf{Arith}}\ A_0$$

$$\frac{a \in \textbf{Arith} \qquad b \in \textbf{Arith}}{('\textbf{Add}\ a\ b) \in \textbf{Arith}}\ A_1$$

$$\frac{a \in \textbf{Arith} \qquad b \in \textbf{Arith}}{('\textbf{Mult}\ a\ b) \in \textbf{Arith}}\ A_2$$

The arithmetic expression is either a number (by rule A_0), or a 3-element list containing a symbol $'$**Add** or $'$**Mult** and two subexpressions a and b.

For consicion, I'll ignore the $'$ symbol and the parentheses when there's no ambiguity. For example, we write **Add** (**Mult** 1 2) 3 as an arithmetic expression that corresponds to $(1 \cdot 2) + 3$.

Are **Add** (**Mult** 1 2) 3 and 5 equal arithmetic expressions? The answer is that it depends. Without first assigning meaning to our language, we can only revert back to the set-theoretic equality, where the number 5 is distinct from **Add** (**Mult** 1 2) 3, which is really a list of three elements.

However, morally, we'd like to define an equivalence relation that identifies these two terms since they both represent the number 5. Such an equivalence relation can be specified by defining *semantics* for our language.

In the following sections, we will explore two different ways of assigning meaning to our language.

4.2 Denotational Semantics

Denotational semantics consist of a function that maps the syntactic terms into something that we already understand and have an existing notion of equality. In the case of **Arith**, we can assign a denotational semantics by defining a recursive function from **Arith** to $\mathbb{N}$.

Of course, before we start writing the recursive equations, we should check that **Arith** has the property that there is at most one derivation tree for each element in it. This is trivially the case since the number n and the lists **Add** a b and **Mult** a b are clearly distinct from one another. Now we define the function $I : \textbf{Arith} \to \mathbb{N}$ with the following equations.

$$I(n) = n$$

$$I(\textbf{Add}\ a\ b) = I(a) + I(b)$$

$$I(\textbf{Mult}\ a\ b) = I(a) \cdot I(b)$$

From the definition of I, we define the relation $\simeq \subseteq \textbf{Arith} \times \textbf{Arith}$ as follows.

$$a \simeq b := I(a) = I(b)$$

Note that the $=$ in $I(a) = I(b)$ is the set-theoretic equality, which behaves the way we expect over natural numbers.

Exercise 11. Show that $\mathbf{Add}\ (\mathbf{Mult}\ 1\ 2)\ 3 \simeq 5$.

Exercise 12. Let **mirror** be a recursive function from **Arith** to **Arith** defined as follows.

$$\begin{aligned}
&\mathbf{mirror}(n) = n\\
&\mathbf{mirror}(\mathbf{Add}\ a\ b) = \mathbf{Add}\ (\mathbf{mirror}(b))\ (\mathbf{mirror}(a))\\
&\mathbf{mirror}(\mathbf{Mult}\ a\ b) = \mathbf{Mult}\ (\mathbf{mirror}(b))\ (\mathbf{mirror}(a))
\end{aligned}$$

Given an arithmetic expression representing $4 + (7 \cdot 8)$ as input, what does the **mirror** function return? Try out a few more examples to get a better understanding of what the function does, then prove through induction that **mirror** is idempotent, meaning that $\forall a \in \mathbf{Arith}, \mathbf{mirror}(\mathbf{mirror}(a)) = a$.

Finally, show that **mirror** preserves the meaning of arithmetic expressions, meaning that $\forall a \in \mathbf{Arith}, a \simeq \mathbf{mirror}(a)$.

Exercise 13. We can extend our arithmetic language with the minus operator by adding the following rule.

$$\frac{a \in \mathbf{Arith} \qquad b \in \mathbf{Arith}}{\mathbf{Minus}\ a\ b \in \mathbf{Arith}}\ A_3$$

Extend the interpretation function I with an equation for this new rule. Prove that the **mirror** function Exercise 12 is still idempotent. Next, find an arithmetic expression a such that $\mathbf{mirror}(a) \simeq a$ doesn't hold.

4.3 Small-Step Operational Semantics

We define the small-step operational semantics for **Arith** by defining the following inductively defined relation $\mathbf{Step} \subseteq \mathbf{Arith} \times \mathbf{Arith}$.

$$\frac{m \in \mathbb{N} \qquad n \in \mathbb{N}}{(\textbf{Add}\ m\ n, m + n) \in \textbf{Step}}\ S_0$$

$$\frac{m \in \mathbb{N} \qquad n \in \mathbb{N}}{(\textbf{Mult}\ m\ n, m \cdot n) \in \textbf{Step}}\ S_1$$

$$\frac{a \notin \mathbb{N} \qquad (a, a') \in \textbf{Step}}{(\textbf{Add}\ a\ b, \textbf{Add}\ a'\ b) \in \textbf{Step}}\ S_2$$

$$\frac{a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad (b, b') \in \textbf{Step}}{(\textbf{Add}\ a\ b, \textbf{Add}\ a\ b') \in \textbf{Step}}\ S_3$$

$$\frac{a \notin \mathbb{N} \qquad (a, a') \in \textbf{Step}}{(\textbf{Mult}\ a\ b, \textbf{Mult}\ a'\ b) \in \textbf{Step}}\ S_4$$

$$\frac{a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad (b, b') \in \textbf{Step}}{(\textbf{Mult}\ a\ b, \textbf{Mult}\ a\ b') \in \textbf{Step}}\ S_5$$

Exercise 14. Show that $(\textbf{Add}\ (\textbf{Mult}\ 3\ 4)\ (\textbf{Add}\ 2\ 3), \textbf{Add}\ 12\ (\textbf{Add}\ 2\ 3)) \in$ *Step* by explicitly writing down the derivation tree.

Next, explain why $(\textbf{Add}\ (\textbf{Mult}\ 3\ 4)\ (\textbf{Add}\ 2\ 3), \textbf{Add}\ (\textbf{Mult}\ 3\ 4)\ 5) \in$ *Step* is *not* true by showing that none of the rules can be applied.

In PL papers, a directed arrow such as $\rightsquigarrow$ is used to name the stepping relation. Furthermore, the infix notation $a \rightsquigarrow b$ is used to denote $(a, b) \in \rightsquigarrow$. Using this new notation, we can write the operational semantics in a more readable format. The induction principles should work the same way, but if you ever get confused, you can always rewrite $a \rightsquigarrow b$ into $(a, b) \in \rightsquigarrow$, which can be useful when you are trying to figure out the induction principles.

$$\frac{m \in \mathbb{N} \qquad n \in \mathbb{N}}{\textbf{Add}\ m\ n \rightsquigarrow m + n}\, S_0$$

$$\frac{m \in \mathbb{N} \qquad n \in \mathbb{N}}{\textbf{Mult}\ m\ n \rightsquigarrow m \cdot n}\, S_1$$

$$\frac{a \notin \mathbb{N} \qquad a \rightsquigarrow a'}{\textbf{Add}\ a\ b \rightsquigarrow \textbf{Add}\ a'\ b}\, S_2$$

$$\frac{a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad b \rightsquigarrow b'}{\textbf{Add}\ a\ b \rightsquigarrow \textbf{Add}\ a\ b'}\, S_3$$

$$\frac{a \notin \mathbb{N} \qquad a \rightsquigarrow a'}{\textbf{Mult}\ a\ b \rightsquigarrow \textbf{Mult}\ a'\ b}\, S_4$$

$$\frac{a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad b \rightsquigarrow b'}{\textbf{Mult}\ a\ b \rightsquigarrow \textbf{Mult}\ a\ b'}\, S_5$$

4.4 A more concise representation

In this section, we present a more concise way of presenting the rules. First, we define **Op** to be the finite set of symbols $\{\textbf{Add}, \textbf{Mult}\}$ (the $'$ mark is omitted for concision). Of course, we can also present it inductively, though it's not quite necessary since the set is finite.

$$\frac{}{\textbf{Mult} \in \textbf{Op}} \qquad \frac{}{\textbf{Add} \in \textbf{Op}}$$

The syntax of the arithmetic language is then defined as follows.

$$\frac{n \in \mathbb{N}}{n \in \textbf{Arith}'}\, A_0'$$

$$\frac{op \in \textbf{Op} \qquad a \in \textbf{Arith}' \qquad b \in \textbf{Arith}'}{op\ a\ b \in \textbf{Arith}'}\, A_1'$$

It is not hard to show that $\textbf{Arith}'$ defines the exact same set elements as **Arith** through induction (you would need to do it twice for each direction).

Denotational semantics for $\textbf{Arith}'$ can be broken in two steps. First, we define the function I_o that maps symbols in the set **op** to their corresponding binary functions over natural numbers.

$$I_o(\textbf{Mult}) = x\ y \mapsto x \cdot y$$
$$I_o(\textbf{Add}) = x\ y \mapsto x + y$$

Then we can define our interpretation function I' with the following recursive equations:

$$\begin{aligned}&I'(n) = n, \text{ where } n \in \mathbb{N}\\ &I'(op\ a\ b) = f(I'(a), I'(b)), \text{ where } f = I_o(op)\end{aligned}$$

Exercise 15. Show that $I'(\textbf{Add}\ (\textbf{Mult}\ 1\ 2)\ 3) = 5$ by carefully unfolding the definitions of I' and I_o.

Remark. The function $I_o : \textbf{Op} \to (\mathbb{N} \times \mathbb{N} \to \mathbb{N})$ takes an element from the set **Op** and returns a binary operator over numbers. Functions that takes another function as input or returns another function as output are referred to as *higher-order functions*. In the definition of $I'(op\ a\ b)$, I introduced the local definition f so the whole concept of functions returning other functions feel less intimidating, but if you are comfortable, you can always inline the definition of f and directly write the recursive case as:

$$I'(op\ a\ b) = (I_o(op))(I'(a), I'(b))$$

The operational semantics can benefit the most from the concise definition as we can group multiple similar cases together.

$$\frac{op \in \textbf{Op} \qquad m \in \mathbb{N} \qquad n \in \mathbb{N}}{op\ m\ n \rightsquigarrow (I_o(op))(m, n)}\ S_0'$$

$$\frac{op \in \textbf{Op} \qquad a \notin \mathbb{N} \qquad a \rightsquigarrow a'}{op\ a\ b \rightsquigarrow op\ a'\ b}\ S_1'$$

$$\frac{op \in \textbf{Op} \qquad a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad b \rightsquigarrow b'}{op\ a\ b \rightsquigarrow op\ a\ b'}\ S_2\text{'}$$

Extending our language with the minus operator is now as simple as extending the set **Op** as $\{\textbf{Mult}, \textbf{Minus}, \textbf{Add}\}$ and adding a new case for $I_o(\textbf{Minus}) = x\ y \mapsto x - y$. The operational semantics and the denotational semantics for $\textbf{Arith}'$ are now capable of handling the minus operation.

4.5 BNF notation

A different way of specifying the syntax of **Arith** is by using the Backus-Naur form (a.k.a the BNF notation). I won't go over what precisely constitutes a BNF grammar specification, but with the tools we have developed so far, we can view BNF notation as a concise way of writing inductively defined data types / languages.

Here is how **Arith** (the concise version in the previous section) can be specified using BNF notation.

$$\begin{aligned} n &\in \mathbb{N} \\ op &:= \textbf{Mult} \mid \textbf{Add} \mid \textbf{Minus} \\ a &:= op\ a\ a \mid n \end{aligned}$$

Each clause $t := t_0 \mid t_1 \ldots \mid t_n$ can be read as t is a metavariable ranging over an inductive set whose elements are of the form one of the t_is.

For example, the metavariable op in this case ranges over the finite set containing $\{\textbf{Mult}, \textbf{Add}, \textbf{Minus}\}$.

The metavariable a ranges over the set of arithmetic terms, which is either an operator op followed by two expressions a of the same set, or a natural number.

The choice of metavariable is important because of the implicit information associated with the set that the metavariable ranges over. For example, the definition $a := \textbf{op}\ b\ a$ does not make sense because we don't have a clause of the form $b := \ldots$.

It is possible, however, to assign multiple metavariable names to the same clause. For example, the clause $a, b := op\ a\ b \mid n$ is equivalent to the clause $a := op\ a\ a \mid n$. In the former, it's understood that both a and b range over the same set of terms that we are defining.

Exercise 16. Write the inductive definition that corresponds to the following BNF grammar for the untyped lambda calculus, which we will cover later in class.

$$\begin{aligned} x &\in \textbf{Symbol} \\ t &:= x \mid \textbf{LAM}\ x\ t \mid \textbf{APP}\ t\ t \end{aligned}$$

Explicitly spelling out the names of the operators as **Mult** or **Add** is very tedious. In some textbooks, you might see a BNF grammar for the arithmetic language presented as follows.

$$\begin{aligned} n &\in \mathbb{N} \\ a &:= a + a \mid a \cdot a \mid n \end{aligned}$$

The same notation for usual arithmetic is reused for specifying the grammar of the language we are trying to define. This style of definition is quite common in practice, but can be very confusing.

For example, we can immediately tell that **Mult** 1 2 is a syntax tree of the language we are defining, and it is a distinct tree from **Mult** 2 1. However, if we use the notation from above, how do we know if $1 \cdot 2$ and $2 \cdot 1$ are equal? If we read $1 \cdot 2$ and $2 \cdot 1$ as syntax trees, then they are not the same. However, if we read $1 \cdot 2$ and $2 \cdot 1$ as applications of the multiplication operator on the numbers 1 and 2, then they both are the number 2. This ambiguity issue can be avoided by choosing the syntax carefully to not collide with language from the metatheory (set theory or Rocq), or simply clarifying whenever ambiguity arises.

As an example of notation usage, the untyped lambda calculus from Exercise 16 can be written as follows.

$$\begin{aligned} x &\in \textbf{Symbol} \\ t &:= x \mid \lambda x.t \mid t\ t \end{aligned}$$

Here, the . that appears in $\lambda x.t$ does not have any meaning. It's simply there to make lambda abstractions look prettier. In fact, both . and λ are implicitly assumed to be symbols as there are no clauses of the form $. := \ldots$ or $\lambda := \ldots$.

Back to the example with arithmetic expressions. We can rewrite the BNF grammar more concisely as follows.

$$\begin{aligned} n &\in \mathbb{N} \\ \circ &:= + \mid \cdot \\ a &:= a \circ a \mid n \end{aligned}$$

In this definition, $\cdot$ is not just a symbol, but a metavariable that ranges over $-$ and $+$. Thus, the language contains expressions of the form $3 + 4$, $9 \cdot 3$ but *not* $3 \circ 4$.

Suppose instead we chose to define the grammar as follows.

$$\begin{aligned} n &\in \mathbb{N} \\ a &:= a \circ a \mid n \end{aligned}$$

Then $\circ$ is implicitly treated as a symbol, and our language only contains expressions of the shape $3 \circ (4 \circ 5)$ but not $3 + (4 + 5)$.