Chapter 1

EBNF: A Notation to Describe Syntax

> *Precise language is not the problem. Clear language is the problem.*
>
> Richard Feynman

Chapter Objectives

- Learn the four control forms in EBNF
- Learn to read and understand EBNF descriptions
- Learn to prove a symbol is legal according to an EBNF description
- Learn to determine if EBNF descriptions are equivalent
- Learn to write EBNF descriptions from specifications and exemplars
- Learn the difference between syntax and semantics
- Learn the correspondence between EBNF rules and syntax charts
- Learn about recursive EBNF rules

1.1 Introduction

We will use EBNF to describe the syntax of Python

EBNF is a notation for formally describing syntax: how to write features in a language. We will study ENBF in this chapter and then use it throughout the rest of this book to describe Python's syntax formally. But there is a more compelling reason to begin our study of programming with EBNF: it is a microcosm of programming itself.

Writing EBNF descriptions is similar to writing programs

First, there control forms in EBNF rules are strongly similar to the the basic control structures in Python: sequence; decision, repetition, and recursion; also similar is the ability to name descriptions and reuse these names to build more complex structures. There is also a strong similarity between the process of writing descriptions in EBNF and writing programs in Python: we must synthesize a candidate solution and then analyze it —to determine whether it is correct and simple. Finally, studying EBNF introduces a level of formality that we will employ throughout our study of programming and and Python.

1.2 Language and Syntax

John Backus helped developed the FORTRAN and ALGOL languages

In the middle 1950s, computer scientists began to design high–level programming languages and build their compilers. The first two major successes were FORTRAN (FORrmula TRANslator), developed by the IBM corporation in the United States, and ALGOL (ALGOrithmic Language), sponsored by a consortium of North American and European countries. John Backus led the effort to develop FORTRAN. He then became a member of the ALGOL design committee, where he studied the problem of describing the syntax of these programming languages simply and precisely.

Backus invented a notation (based on the work of logician Emil Post) that was simple, precise, and powerful enough to describe the syntax of any programming language. Using this notation, a programmer or compiler can determine whether a program is syntactically correct: whether it adheres to the grammar and punctuation rules of the programming language. Peter Naur, as editor of the ALGOL report, popularized this notation by using it to describe the complete syntax of ALGOL. In their honor, this notation is called Backus–Naur Form (BNF). This book uses Extended Backus–Naur Form (EBNF) to describe Python syntax, because using it results in more compact descriptions.

Backus developed a notation to describe syntax; Peter Naur then popularized its use: they are the B and N in EBNF

In a parallel development, the linguist Noam Chomsky began work on a harder problem: describing the syntactic structure of natural languages, such as English. He developed four different notations that describe languages of increasing complexity; they are numbered type 3 (least powerful) up through 0 (most powerful) in the Chomsky hierarchy. The power of Chomsky's type 2 notation is equivalent to EBNF. The languages in Chomsky's hierarchy, along with the machines that recognize them, are studied in computer science, mathematics, and linguistics under the topics of formal language and automata theory.

At the same time, linguist Noam Chomsky developed notations to describe the syntax of natural languages

1.3 EBNF Rules and Descriptions

An EBNF description is an unordered list of EBNF rules. Each EBNF rule has three parts: a left–hand side (LHS), a right-hand side (RHS), and the $\Leftarrow$ character separating these two sides; read this symbol as "is defined as". The LHS is one *italicized* word (possibly with underscores) written in lower–case; it names the EBNF rule. The RHS supplies a description of this name. It can include combinations of the four control forms explained in Table 1.1.

EBNF descriptions comprises a list of EBNF rules of the form: LHS $\Leftarrow$ RHS

Table 1.1: Control Forms of Right–Hand Sides

Sequence	Items appear left–to–right; their order in important.
Choice	Alternative items are separated by a | (stroke); one item is chosen from this list of alternatives; their order is unimportant.
Option	The optional item is enclosed between [and] (square–brackets); the item can be either included or discarded.
Repetition	The repeatable item is enclosed between { and } (curly–braces); the item can be repeated **zero** or more times; yes, we can chosose to repeat **zero** times, fact beginners often forget.

EBNF rules can include these six characters with special meanings: $\Leftarrow$, |, [,], {, and }. Except for these characters and the italicized names in EBNF rules, anything that appears in a RHS stands for itself. If we want to put any of

Special characters standing for themselves in EBNF rules appear in boxes these special characters standing for themself in a RHS, we put it in a box: so | means alternative but $\boxed{|}$ means the stroke character.

1.3.1 An EBNF Description of Integers

An EBNF description: *integer*

The following EBNF rules describe how to write simple integers.[1] Their RHS illustrates every control form available in EBNF.

EBNF Description: *integer*

digit ⇐ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
integer ⇐ [+|-]*digit*{*digit*}

What do these EBNF rules mean? We can paraphrase their meaning in English

We can paraphrase the meanings of these rules in English.

- A *digit* is defined as one of the ten alternative characters 0 through 9.
- An *integer* is defined as a sequence of three items: an optional sign (if it is included, it must be one of the alternatives + or -), followed by any *digit*, followed by a repetition of zero or more *digit*s, where each repeated digit is independently chosen from the list of alternatives in the *digit* rule.

The RHS of *integer* combines all the control forms in EBNF: sequence, option, choice, and repetition. We will see longer, more complicated EBNF descriptions, but their rules always use just these four control forms.

We adopt a few simple conventions for typesetting EBNF rules and descriptions

To make EBNF descriptions easier to read and understand, we align their rule names, the ⇐, and their rule definitions. Sometimes we put extra spaces in a RHS, to make it easier to read; such spaces do not change the meaning of the rule. Although the rules in EBNF descriptions are unordered, we will adopt the convention writing them in in order of increasing complexity: later rules often refer to the names of earlier ones. The last EBNF rule names the main syntactic structure being described: in this case *integer*.

1.4 Proving Symbols Match EBNF Rules

A language lawyer can prove whether a symbol is legal or illegal according to an EBNF description

Now that we know how to read an EBNF description, we must learn how to interpret its meaning like a language lawyer: given an EBNF description and a symbol —any sequence of characters— we must prove the symbol is legal or prove it is illegal, according to the description. Computers perform expertly as language lawyers, even on the most complicated descriptions.

We perform the proof by matching the characters in a symbol against the parts of the EBNF rule

To prove that a symbol is legal according to some EBNF rule, we must match all its characters with all the parts in the EBNF rule, according to that rule's control forms. If there is an exact match —we exhaust the characters in the symbol at the same time when exhaust the rule's control forms— we classify the symbol as legal according to that description and say it mathces; otherwise we classify the symbol as illegal and say it doesn't match.

[1] The EBNF descriptions in this chapter are for illustration purposes only: they do not describe any of Python's actual language features. Subsequent chapters use EBNF to describe Python.

1.4.1 Verbal Proofs (in English)

Proving in English that `7` is a legal *integer*

To prove in English that the symbol `7` matches the *integer* EBNF rule, we must start with the optional sign: the first of three parts in the sequence. In this case we discard the option, because it does not match the only character in the symbol. Next in the sequence, the symbol must match a character that is a *digit*; in this case, we choose the `7` alternative from the RHS of the *digit* rule, which matches the only character in the symbol. Finally, we must repeat *digit* zero or more times; in this case we use zero repetitions.

Success: `7` is an *integer*

Every character in the symbol `7` has been matched against every part of the *integer* EBNF rule, according to its control forms: we have exhausted each. Therefore, we have proven that `7` is a legal *integer* according to its EBNF description.

Proving `+142` is a legal *integer*

We use a similar argument to prove in English that the symbol `+142` matches the *integer* EBNF rule. Again we must start with the optional sign: the first of the three parts in the sequence. In this case we include this option and then choose the `+` alternative inside the option: we have now matched the first character in the symbol with the first part of *integer*'s sequence. Next in the sequence, the symbol must have a character that we can recognize as a *digit*; in this case we choose the `1` alternative from the RHS of the `digit` rule, which matches the second character in the symbol. Finally, we must repeat *digit* zero or more times; in this case we use two repetitions: for the first repetition we choose *digit* to be the `4` alternative, and for the second repetition we choose *digit* to be the `2` alternative. Recall that each time we encounter a *digit*, we are free to choose any of its alternatives.

Success: `+142` is an *integer*

Again, every character in the symbol `+142` has been matched against every part of the *integer* EBNF rule, according to its control forms: we have exhausted each. Therefore, we have proven that `+142` is also a legal *integer*.

Some short proofs that the symbols `1,024` `A15` and `15-` are not legal *integer*s

We can easily prove that `1,024` is an illegal *integer* by observing that the comma appearing in this symbol does not appear in either EBNF rule; therefore, the match is guaranteed to fail: the match fails after discarding the sign option and matching the first *digit*. Likewise for the letter `A` in the symbol `A15`. Finally, we can prove that `15-` is an illegal *integer* —not because it contains an illegal character, but because its structure is incorrect: in this symbol `-` follows the last digit, but the sequence in the *integer* rule requires that the sign precede the first digit: the match fails after discarding the sign option and matching two *digit*s, at which point the symbol still contains the character `-` while all the parts of the *integer* EBNF rule have been matched. So according to our rules for proof, none of these symbols is a legal *integer*.[2] When matching symbols as a language lawyer, we cannot appeal to intuition: we must rely solely on the EBNF description we are matching.

[2] All three symbols are legal integers under some interpretation: the first uses a comma to separate the thousands digit from the hundreds, the second is a valid number written in hexadecimal (base 16), and the third is a negative number —sometimes written this way by accountants to emphasize, at the end, whether a value is a debit or credit. But according to the *integer* EBNF rule, none is legal.

1.4.2 Tabular Proofs

A tabular proof is a more formal demonstration that a symbol matches an EBNF description. The first line in a tabular proof is always the name of the EBNF rule that specifies the syntactic structure we are trying to match the symbol against: in this example, *integer*. The last line is the symbol we are matching. Each line is derived from the previous one according to the following rules.

A tabular proof starts with the EBNF rule to match, and eventually generates from it all the characters in the symbol

1. Replace a name (LHS) by its definition (RHS)
2. Choose an alternative
3. Determine whether to include or discard an option
4. Determine the number of times to repeat

Combining rules 1 and 2 (1&2) simplifies our proofs by allowing us, in a single step, to replace a left–hand side by one of the alternatives in its right–hand side. The left side of Figure 1.1 shows a tabular proof that +142 is an *integer*.

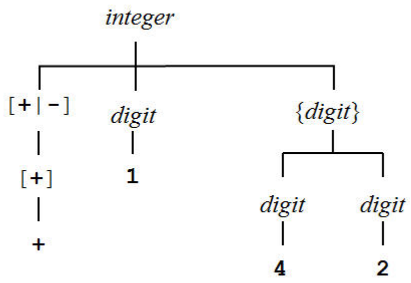
1.4.3 Derivation Trees

We illustrate a tabular proof more graphically by writing it as a derivation tree. The downward branches in such a tree illustrate the rules that allow us to go from one line to the next in a tabular proof. Although a derivation tree displays the same information as a tabular proof, it omits certain irrelevant details: the ordering of some decisions in the proof (e.g., which *digit* is replaced first). The ENBF rule appears at the root and the matching symbol appears in the leaves of a derivation tree, at the bottom when its characters are read left to right. The right side of Figure 1.1 shows a derivation tree for the tabular proof on the left, proving +142 is an *integer*.

A derivation tree illustrates a tabular proof graphically, with the ENBF rule name at the top (its root) and the symbol's characters at the bottom (its leaves)

Figure 1.1: A Tabular Proof and it Derivation Tree showing 142 is an *integer*

Status	Reason (rule #)
integer	Given
[+|−]*digit*{*digit*}	Replace *integer* by it RHS (1)
[+]*digit*{*digit*}	Choose + alternative (2)
+*digit*{*digit*}	Include option (3)
+1{*digit*}	Replace the first *digit* by 1 alternative (1&2)
+1*digit digit*	Use two repetitions(rule 4)
+14*digit*	Replace the first *digit* by 4 alternative (1&2)
+142	Replace the first *digit* by 2 alternative (1&2)



Section Review Exercises

1. Classify each of the following symbols as a legal or illegal *integer*. Note that part o. specifies a symbol containing no characters.

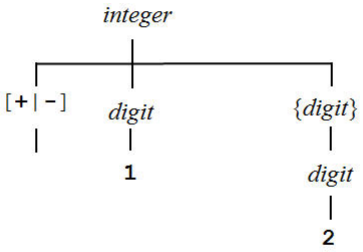
a. +42
b. +
c. -0
d. VII
e. -1492
f. 187
g. drei
h. 25¢
i. 2B
j. 187.0
k. $15
l. 1000
m. 0
n. forty--two
o.
p. 555-1212
q. +-7
r. 1 543
s. 1+1
t. 0007

Answer: Only a, c, e, f, l, m, and t are legal.

2. a. Write a tabular proof that `-1024` is a legal *integer*. b .Draw a derivation tree showing `12` is a legal *integer*.

   ANSWER: Note how the discarded [+|-] option is drawn in the derivation tree.

Status	Reason (rule #)
integer	Given
[+|-] *digit*{*digit*}	Replace *integer* by it RHS (1)
[-]*digit*{*digit*}	Choose - alternative (2)
-*digit*{*digit*}	Include option (3)
-1{*digit*}	Replace the first *digit* by 1 alternative (1&2)
-1*digit digit digit*	Use three repetitions (4)
-10*digit digit*	Replace the first *digit* by 0 alternative (1&2)
-102*digit*	Replace the first *digit* by 2 alternative (1&2)
-1024	Replace *digit* by 4 alternative (1&2)



1.5 Equivalent EBNF Descriptions

The following EBNF description is equivalent[3] to the one presented in the previous section. Two EBNF descriptions are equivalent if they recognize exactly the same legal and illegal symbols: for every possible symbol, both classify it as legal or both classify it as illegal —they never classify symbols differently.

When are two EBNF descriptions equivalent

EBNF Description: *integer* (equivalent, in 3 rules)

sign ⇐ +|-
digit ⇐ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
integer ⇐ [*sign*]*digit*{*digit*}

This EBNF description is not identical to the first, because it defines an extra *sign* rule that is then used in the *integer* rule. But these two EBNF descriptions are equivalent, because providing a named rule for [+|-] does not change which symbols are legal. In fact, even if the names of all the rules are changed uniformly, exactly the same symbols are recognized as legal.

Extra names do not change the meanings of EBNF descriptions; nor do different names for the rules

EBNF Description: *z* (really *integer* with different rule names)

x ⇐ +|-
y ⇐ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
z ⇐ [*x*]*y*{*y*}

Any symbol recognized as an `integer` by the previous EBNF descriptions is recognized as a *z* in this description, and vice–versa. Just exchange the names x, y, and z, for sign, digit, and integer in any tabular proof or derivation tree.

A symbol is a legal *integer* exactly when it is a legal *z*

Complicated EBNF descriptions are easier to read and understand if their rules are well–named, each name helping to communicate the meaning of its rule's definition. But to a language lawyer or compiler, names —good or bad— cannot change the meaning of a rule or the classification of a symbol.

EBNF rules are easier to understand if they are well–named, but the names do not affect the meanings of ENBF rules

[3] Equivalent means "are always the same within some context". For example, dollar bills are equivalent in their buying power. But a dollar bill only has equivalent buying power to four quarters in some contexts: when trying to buy a 75¢ item in a vending machine that requires exact change, the dollar bill does not have equivalent buying power.

1.5.1 Incorrect *integer* Descriptions

This section examines two EBNF descriptions that contain interesting errors. To start, we try to simplify the *integer* rule by removing the `digit` that precedes the repetition, thinking we can always repeat one more time. The best description is the simplest one; so, if this new rule is equivalent to to the previous one, we have improved the description of *integer*.

A simpler syntax for *integer*; but it is not equivalent to the previous one

EBNF Description: *integer* (simplified but not equivalent)

sign ⇐ `+`|`-`
digit ⇐ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
integer ⇐ [*sign*]{*digit*}

In fact, every symbol that is a legal *integer* in the previous EBNF descriptions is also legal in this one. For example, we can use this new EBNF description to prove that `+142` is an integer: include the *sign* option, choose the `+` alternative; repeat *digit* three times, choosing `1`, `4`, and `2`.

The rules are almost equivalent

But there are two simple symbols that this description recognizes as legal, which the previous descriptions classify as illegal: the one–character symbols `+` and `-` (just signs, without any following digits). The previous *integer* rules all require one *digit* followed by zero or more repetitions; but this *integer* rule contains just the repetition, which may be taken zero times. To prove `+` is a legal *integer*: include the *sign* option, choosing the `+` alternative; repeat *digit* zero times The proof that `-` is legal is similar.

But they classify two special one–character symbols differently

Also, the "empty symbol", a corner–case that contains no characters, is recognized by this EBNF description as a legal *integer*: discard the *sign* option; repeat *digit* zero times. Because of these three differences, this EBNF description of *integer* is not equivalent to the previous ones; so which one is correct? Intuitively, an *integer* is required to contain at least one *digit* so I would judge this *integer* rule to be incorrect. Note that equivalence is a formal property of EBNF, but correctness requires human judgement.

They also classify a corner case, zero–character symbol, differently

Next we address, and fail to solve, the problem of describing how to write numbers with embedded commas: e.g., `1,024` and other numbers where the thousands, millions, billions, ... position is followed by a comma. We can easily extend the *digit* rule to allow a comma as one of its alternatives: the comma character is not one of EBNF's special control forms, so it stands for itself.

A failed attempt to write numbers with embedded commas

EBNF Description: *comma_integer* (attempt to allow embedded commas)

sign ⇐ `+`|`-`
comma_digit ⇐ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `,`
comma_integer ⇐ [*sign*]*comma_digit*{*comma_digit*}

Using this description, we can prove that `1,024` is a legal *comma_integer*: discard the *sign* option; chose *digit* to be the `1` alternative; repeat *comma_digit* four times, choosing `,` first (a comma) then `0`, `2`, and `4`. But, we can also prove that `1,,3,4` and many other symbols with incorrectly embedded commas are legal according to *comma_integer*. So, we cannot treat a comma as if it were a *digit*; we need structurally more complicated EBNF rules to correctly classify exactly those symbols that have embedded commas in the correct locations. See Exercise 6for formal a statement of this problem (and a hint towards the needed structure).

Numbers with correctly embedded commas are legal according to this EBNF description; but so are numbers with incorrectly embedded commas

Section Review Exercises

1. Are either of the following EBNF descriptions equivalent to the standard ones for *integer*? Justify your answers.

sign	⇐ [+|-]	*sign*	⇐ [+|-]
digit	⇐ 0|1|2|3|4|5|6|7|8|9	*digit*	⇐ 0|1|2|3|4|5|6|7|8|9
integer	⇐ *sign digit*{*digit*}	*integer*	⇐ *sign*{*digit*}*digit*

   Answer: Each is equivalent. Left: it is irrelevant whether the option brackets appear around *sign* in the *integer* rule, or around + and - in the *sign* rule; in either case there is a way to include or discard the sign. Right: it is irrelevant whether the mandatory *digit* comes before or after the repeated ones; in either case one digit is mandated and there is a way to specify one or more digits.

2. Write an EBNF description for *even_integer* that recognizes only even integers: e.g., -6 and 34 are legal but 3 and -23 are not legal.

   Answer:
   sign ⇐ +|-
   even_digit ⇐ 0 | 2 | 4 | 6 | 8
   digit ⇐ *even_digit* | 1 | 3 | 5 | 7 | 9
   even_integer ⇐ [*sign*]{*digit*}*even_digit*

3. Normalized integers have no extraneous leading zeros, and zero must be unsigned. Write an EBNF description for *normalized_integer*. Legal: 0, -1, and 451. Illegal: -01, 007, +0, and -0.

   Answer:
   sign ⇐ +|-
   non_0_digit ⇐ 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
   digit ⇐ 0 | *non_0_digit*
   normalized_integer ⇐ 0 | [*sign*]*non_0_digit* {*digit*}

1.6 Syntax versus Semantics

EBNF descriptions specify only syntax: the form in which something is written. They do not specify semantics: the meaning of what is written. The sentence, “Colorless green ideas sleep furiously.” illustrates the difference between syntax and semantics: it is syntactically correct, because the grammar and punctuation are proper. But what does this sentence mean? How can ideas sleep? If ideas can sleep, what does it mean for them to sleep furiously? Can ideas have colors? Can ideas be both colorless and green? These questions all relate to the semantics, or meaning, of the sentence. As another example the sentence, ”The Earth is the fourth planet from the Sun” has an obvious meaning, but its meaning is contradicted by known astronomical facts.

Syntax = Form
Semantics = Meaning

Two semantic issues are important in programming languages:

- Can different symbols have the same meaning?
- Can one symbol have different meanings?

Can different symbols have the same meaning? Can one symbol have different meanings?

The first issue is easy to illustrate; the symbols we analyze is a name. Everyone has a nickname; so two names (two symbols) can refer to the same person. The second issue is a bit more subtle; here the symbol we analyze is the phrase “next

Different symbols can have the same meaning and one symbol can have different meanings depending on its context class" in a sentence. Suppose you take a course meeting Mondays, Wednesdays and Fridays. If your instructor says on Monday, "The next class is canceled." you know not to come to class on Wednesday. Now suppose you take another course meeting every weekday. If your instructor for that course says on Monday, "The next class is canceled." you know not to come to class on Tuesday. Finally, if it were Friday, "The next class is canceled." has the same meaning in both courses: there is no class on Monday. So the meaning of a a phrase (the symbol) in a sentence may depend on its context (what course you hear it in).

1.6.1 Semantics of the *integer* EBNF rule

Now we examine the semantic issues related to our EBNF description for *integer*. In a mathematical context, the meaning of a number is its value. In common usage, the symbols `1` and `+1` both have the same value: an omitted sign is considered equivalent to a plus sign. As a more special case, the symbols `0` and `+0` and `-0` all have the same value: the sign of zero is irrelevant.

Positive values can omit the plus sign; zero can have any sign

Generally, the symbols `000193` and `193` both have the same meaning: leading zeros do not effect a number's value. But there are contexts where `000193` and `193` have different meanings. I once worked on a computer where each user was assigned a six–digit account number; mine was `000193`. When I logged in, the computer expected me to identify myself with a six–digit account number; it accepted `000193` but rejected `193`.

Leading zeroes are ignored — sometimes

A final example concerns how measurements are written: although `9.0` and `9.0000` have the same value, the former may indicate the quantity was measured to only two significant digits; the latter to five.

Trailing zeroes can indicate a measurement's precision

1.6.2 Syntax and Semantics of the *integer_set* EBNF rule

Let us now explore the syntax and semantics of sets of integers. Such sets start and end with curly–braces, and contain zero or more *integer*s separated by commas. The empty set {}, a singleton set {`1`}, and a set containing the three values {`5,-2,11`} are all examples of legal sets. Sets are illegal if they omit either of the matching curly–braces or commas between *integer*s: e.g., adjacent commas {`1,,2`} or extra commas {`1,2,3,`} or other structural defects.

The syntax of sets: matching curly–braces, enclosing zero or more integers separated by commas

Given an EBNF description of *integer*, the following EBNF rules describe such a *integer_set*. Note that the open/close curly–braces in the *integer_list* rule means repetition; the open/close curly–braces in boxes in the *integer_set* rule means the open/close curly–brace character, not a repetition.

The EBNF description for *integer_set* comprises two EBNF rules

EBNF Description: *integer_set*

$integer\_list \Leftarrow integer\{,integer\}$

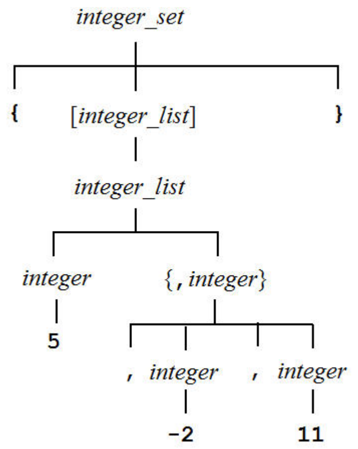
$integer\_set \Leftarrow \boxed{\texttt{\{}}[integer\_list]\boxed{\texttt{\}}}$

We can easily prove that the empty set is a legal *integer_set*: discard the *integer_list* option between the curly–braces. For a singleton set, we include the *integer_list* option, but use zero repetitions after the first *integer*. Figure 1.2 shows a tabular proof and its derivation tree that {`5,-2,11`} is a legal *integer_set*. We shorten this tabular proof and its derivation tree by using lemmas: we take as a lemma (without proof) that `5`, `-2`, and `11` are each an integer; we

We can easily prove the empty set and a singleton set are each a legal *integer_set* could fill in these details, but they would obscure the main result in the proof.

Figure 1.2: Proofs showing {`5,-2,11`} `is an` *integer_set*

Status	Reason (rule #)
integer_set	Given
$\boxed{\texttt{\{}}$[*integer_list*]$\boxed{\texttt{\}}}$	Replace *integer_set* by it RHS (1)
$\boxed{\texttt{\{}}$*integer_list*$\boxed{\texttt{\}}}$	Include option (3)
$\boxed{\texttt{\{}}$*integer*{`,`*integer*}$\boxed{\texttt{\}}}$	Replace *integer_list* by its RHS (1)
$\boxed{\texttt{\{}}$`5`{`,`*integer*}$\boxed{\texttt{\}}}$	Lemma: `5` is an *integer*
$\boxed{\texttt{\{}}$`5,`*integer*`,`*integer*$\boxed{\texttt{\}}}$	Use two repetitions (4)
$\boxed{\texttt{\{}}$`5,-2,`*integer*$\boxed{\texttt{\}}}$	Lemma: `-2` is an *integer*
$\boxed{\texttt{\{}}$`5,-2,11`$\boxed{\texttt{\}}}$	Lemma: `11` is an *integer*



Before finishing our discussion of the syntax of *integer_set*, let us re–examine the *integer_list* EBNF rule. It illustrates a common EBNF idiom: some number of values (here *integer*) separated from each other by some separator symbol (here comma). Notice in that rule the number of *integer* values is always one greater than the number of commas: there is one *integer* before the repetition; and inside the repetition there is one *integer* following every comma. When we study the syntax of Python, we will see many examples of this idiom.

The *entity* {*separator entity*} control form idiom is common in EBNF: e.g., *integer* {`,` *integer*}

Now we switch our focus to semantics and examine when two sets are equivalent. The rules involve duplicate values and the order of values.

Set semantics: whether a value is duplicated and the order of values in a set is irrelevant

- Duplicate values are irrelevant and can be removed: e.g., {`1,3,5,1,3,3,5`} is equivalent to {`1,3,5`}.
- The order of the values is irrelevant and can be rearranged: e.g., {`1,3,5`} is equivalent to {`1,5,3`} and {`3,1,5`} and all other permutations of these values.

By convention, we write sets in an ordered form, starting with the smallest value and ending with the largest; we write each value once. Such a form is called "canonical". It is impossible for our EBNF description to enforce these properties, which is why these rules are considered to be semantic, not syntactic.

We write sets in a canonical form: values in ascending order with no duplicates

The following EBNF rules are an equivalent description for writing *integer_set*. Here, the option brackets are in the *integer_list* rule, not the *integer_set* rule.

An an equivalent *integer_set* description

EBNF Description: *integer_set* (equivalent but more complex)
integer_list ⇐ [*integer*{*,integer*}]
integer_set ⇐ $\boxed{\texttt{\{}}$*integer_list*$\boxed{\texttt{\}}}$

There are two stylistic reasons to prefer the original description. First, it better balances the complexity between the EBNF rules: the repetition control form is in one rule, and the option control form rule is in the other; here both control forms are in the *integer_list* rule. Second, the new description allows

Prefer the first EBNF description of *integer_set*, because it is simpler *integer_list* to match the empty symbol, which contains no characters; this is a bit awkward and can lead to problems if this EBNF rule is used in others.

In summary, EBNF descriptions specify syntax, not semantics. When we describe the syntax of a Python language feature in EBNF, we will describe its semantics using a mixture of English definitions and illustrations. Computer scientists are still developing formal notations that describe the semantics of programming languages in clear and precise ways. In general, form is easier to describe than meaning.

Syntax is easier to describe formally than semantics

Section Review Exercises

1. Structured integers are unsigned and can contain embedded underscores that separate groups of digits, indicating some important structure. We can use structured integers to encode information in an easy to read form: dates 7_4_2012, phone numbers 1_800_555_1212, and credit card numbers 3141_5926_5358_9793. Underscores are legal only between digits: not as the first or last character in the symbol, and not adjacent to each other.

   Write a single EBNF rule that describes *structured_integer*, capturing exactly these requirements. Hint: reuse the *digit* rule and employ a variant of the idiom discussed above: a variant because there is no mandate that underscores appear between digits.

   Answer: *structured_integer* ⇐ *digit*{[_]*digit*}

2. Semantically, underscores do not affect the value of a *structured_integer*: e.g., `1_555_1212` has the same meaning as `15551212`; when dialing either number, we press keys for only the characters representing a *digit*.

   a. Find two dates that have the same meaning, when each is written as a different looking *structured_integer*. b. Propose a new semantic requirement for writing dates that alleviates this problem.

   Answer: a. The date December 5, 1987 is written as `12_5_1987`; the date January 25, 1987 is written as 1_25_1987. Both symbols have the same meaning: the value `1251987`. b. To alleviate this problem, always use two digits to specify a day, adding a leading zero if necessary. We write the first date as `12_05_1987` and the second as `1_25_1987`. These structured integers have different values.

1.7 Syntax Charts

A syntax chart is a graphical representation of an EBNF rule. Figure 1.3 illustrates how to translate each EBNF control form into its equivalent syntax chart. In each case, we must follow the arrows from the beginning of the picture to the end, staying on a path through all the characters in the symbol.

Syntax charts are a graphical representation of EBNF rules

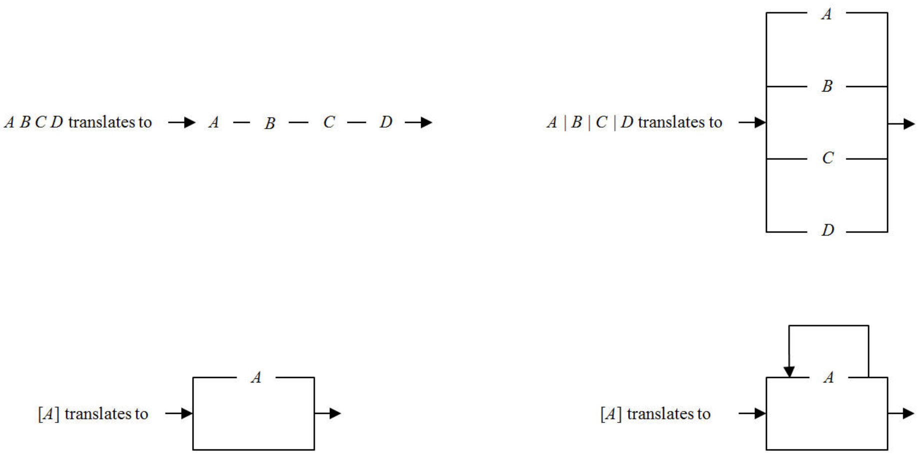
In a sequence, we must go through each item. In a choice, we must go through one rung in the ladder of alternatives. In an option we must go through either the top rung including the item, or the bottom rung discarding it. Repetition is like option, but we can additionally repeat the item in the top rung; this picture is the only one with a right–to–left arrow.

We can translate each EBNF control form to an equivalent syntax chart and traverse it from left to right

We can combine these control forms to translate the RHS of any EBNF rule into its equivalent syntax chart. We can also compose related syntax charts into

We can compose the syntax charts for an EBNF description into one large chart with no named EBNF rules

Figure 1.3: Syntax chart equivalents of EBNF rules



one big syntax chart that contains no named EBNF rules, by replacing each named LHS with the syntax chart for its RHS. Figure 1.4 shows the syntax chart equivalents of the *digit* and the original *integer* EBNF rules, and one large composed syntax chart for *integer*.

Syntax charts can help us disambiguate small EBNF rules

The syntax charts in Figure 1.5 illustrate the RHS of three interesting EBNF rules. The first shows a repetition of two alternatives, where any number of intermixed `A`s and `B`s are legal: `AAA`, `BB`, or `BABBA`. A different choice can be made for each repetition. The second shows two alternatives where a repetition of `A`s or a repetition of `B`s are legal: `AAA` or `BB`, but not `AB`: once the choice is made, only one of these characters can be repeated. Any symbol legal in this rule is legal in the first, but not vice versa.

We can use two EBNF rule to factor a common tail in two alternatives

The last illustration shows how the sequence and choice control forms interact: the stroke separates the first alternative (the sequence AB) from the second (just C). To describe the sequence of A followed by either B or C we must write something different: either both alternatives fully or use a second rule to "factor out" the alternatives.

$$\begin{array}{ll} & \textit{tail} \Leftarrow B \mid C \\ \textit{simple} \Leftarrow A\ B \mid A\ C & \textit{simple} \Leftarrow A\ \textit{tail} \end{array}$$

Which is better: EBNF or syntax charts?

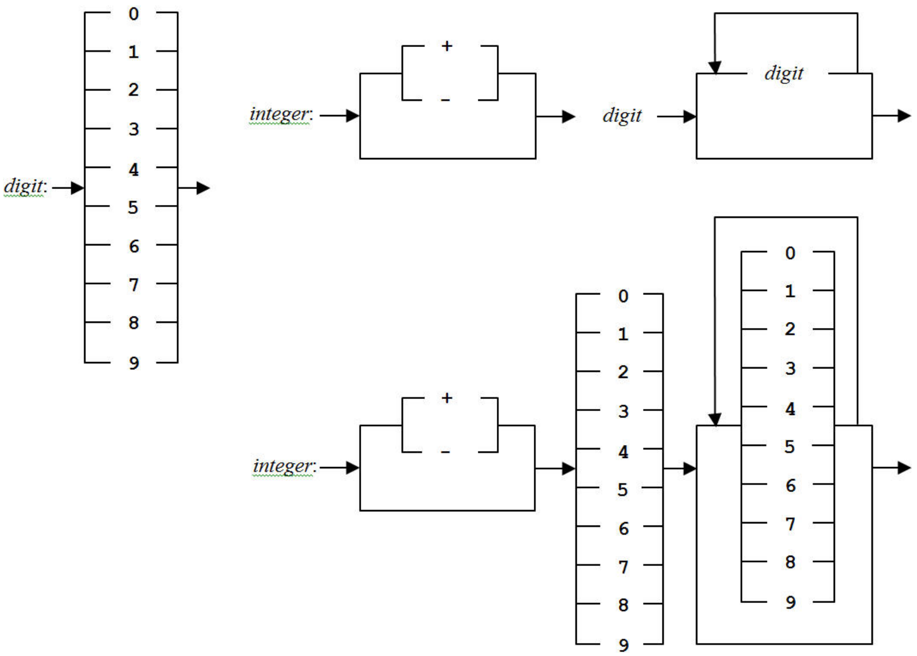
EBNF is a compact text–based notation; syntax charts present the same information, but in a graphical form. Which is better? For beginners, syntax charts are easier to use when proving whether symbols legal or illegal: we just follow the arrows to see if we can get from the start to the end. For more advanced students, EBNF descriptions are better: they are smaller and eventually are easier to read and understand. Because beginning students become advanced ones, this book uses EBNF rules —not syntax charts— to describe Python's syntax.

Section Review Exercises

1. a. Translate each of the following right–hand sides into its syntax chart.
   `A{[␣]A}`   `{A[B|C]}`   `{A | B[C]}`

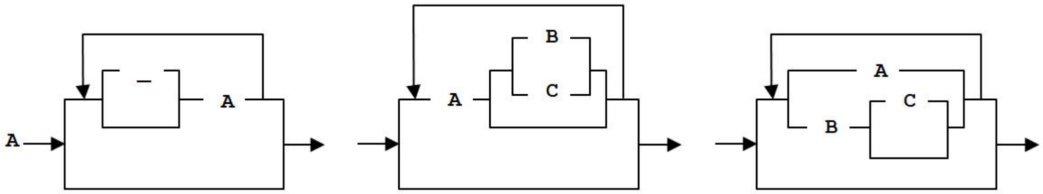
   b. Which symbols are legal according to the first RHS: `A␣A`, `AA␣AAA`, `␣A`,

Figure 1.4: Syntax charts for *digit* and *integer* and a composed *integer*



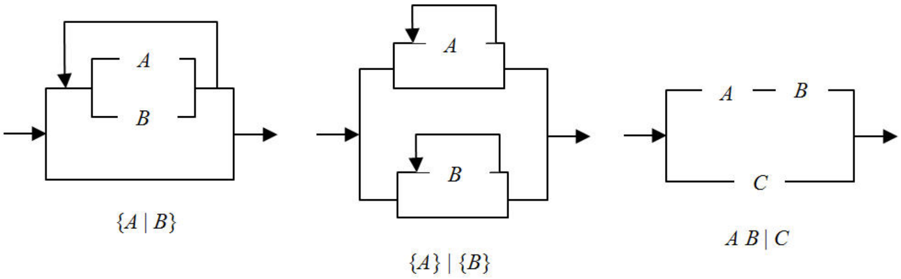
A_, A__A? c. Which symbols are legal according to the second and third RHS: ABAAC, ABC, BA, AA, ABBA.

Answer:



b. A_A and AA_AA. c. For the second, ABAAC, AA. For the third, ABC, BA, AA,

Figure 1.5: Syntax charts disambiguating Interesting ENBF Rules



{A | B}

{A} | {B}

A B | C

and ABBA.

1.8 Advanced EBNF: Recursion (optional)

This section examines two advanced concepts: recursive EBNF rules and using them to describe EBNF in EBNF. We will learn that recursion is more powerful than repetition and that we must use recursive EBNF rules to specify the structure of certain complicated symbols. In programming, recursion is a useful technique for specifying and processing complex data structures.

Recursion is a simple and powerful feature in EBNF

Recursive EBNF descriptions can contain rules that are "directly recursive" or "mutually" recursive: such rules use their names in a special way. A directly recursive EBNF rule uses its own name in its definition: the RHS of the EBNF rule refers to the name of its LHS. Let's look at an example to see how we avoid what looks like a circular definition. The following directly recursive EBNF rule[4] is very simple: it allows symbols containing any number of As, which we can describe mathematically as $\mathtt{A}^n$, where $n \geq 0$ (meaning n As, where n is greater than or equal to 0: zero As is the empty symbol).

Direct recursion is when the RHS of an EBNF rule refers to the name of its LHS

EBNF Description: r (a sequence of As, defined recursively)

$r \Leftarrow \mid \mathtt{A}r$

The first alternative in r contains the empty symbol, which is a legal r: it is a sequence of zero As. Directly recursive EBNF rules must include at least one alternative that is not recursive, otherwise they are circular and describe only infinite–length symbols. Often there is just one non–recursive alternative, which is the empty symbol, and is written first in the EBNF rule.

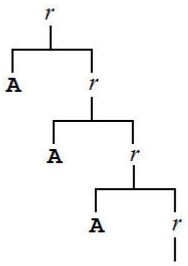
A directly recursive EBNF rule requires a RHS with at least one non–recursive alternative

The second alternative means that an A preceding anything that is a legal r is also recognized as an r; likewise, if we have an r in a tabular proof, we can replace it by an A followed by an r (or by the empty symbol, avoiding a completely circular definition). So A is a legal r because it has an A preceding the empty symbol (which is a legal r); likewise AA is also a legal r, because it has an A preceding an A (which we just showed was a legal r), etc. Figure 1.6 shows a tabular proof and its derivation tree, illustrating how AAA is a legal r. Finally, if we required at least one A, this rule can be written more understandably as $r \Leftarrow \mathtt{A} \mid \mathtt{A}r$, with A (not empty) as the non–recursive alternative.

In a recursive EBNF rule, we just substitute the LHS with any alternative in the RHS — something that might not or might contain the name of its LHS

Figure 1.6: A Tabular Proof and its Derivation Tree showing AAA is an r

Status	Reason (rule #)
r	Given
Ar	Replace r by the second alternative in its RHS (1&2)
AAr	Replace r by the second alternative in its RHS (1&2)
AAAr	Replace r by the second alternative in its RHS (1&2)
AAA	Replace r by the first (empty) alternative in it RHS (1&2)



[4] An equivalent description is $r \Leftarrow \mid r\mathtt{A}$, which is left–recursive instead of right–recursive.

The recursive EBNF rule r is equivalent to the non–recursive EBNF rule $r \Leftarrow \{\texttt{A}\}$, which uses repetition instead. Recursion can always replace repetition, but the converse is not true,[5] because recursion is more powerful than repetition. For example the following directly recursive EBNF description specifies that a symbol is legal if it has the same number of Bs following As: $\texttt{A}^n\texttt{B}^n$, where $n \geq 0$.

Recursion is more powerful than repetition

EBNF Description: *eq*

$eq \Leftarrow \mid \texttt{A}\ eq\ \texttt{B}$

This description cannot be written without recursion. The same symbols are all legal with the rule $eq \Leftarrow \{\texttt{A}\}\{\texttt{B}\}$, but by this rule does not (and cannot) specify that only equal number of As and Bs in a symbol are legal.

$eq \Leftarrow \{\texttt{A}\}\{\texttt{B}\}$ is not equivalent; there is no restriction on equal numbers of As and Bs

We asserted above that repetition can always be replaced by recursion. We can also replace any option control form by an equivalent choice control form that also contains an empty symbol. Using both techniques, we can rewrite our original *integer* description —or any other EBNF rules— using only the recursion and choice control forms and the empty symbol. In fact, the original definition of BNF had only recursion and choice; EBNF (developed by Niklaus Wirth) added the option and repetition control forms. So, although EBNF has more features and therefore is harder to learn, it has no more power than BNF, but its descriptions are often smaller and easier to understand.

EBNF, with option and repetition, is no more powerful than BNF, without these features: no special meanings for square–brackets and curly–braces; we can rewrite EBNF rules into BNF rules

EBNF Description: *integer* (using BNF not EBNF)

$sign \Leftarrow \mid \texttt{+} \mid \texttt{-}$
$digit \Leftarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}$
$digits \Leftarrow \mid digit\ digits$
$integer \Leftarrow sign\ digit\ digits$

Designers have to balance the complexity of learning and using a tool with the complexity building artifacts with the tool. This is a trade–off that we will revisit often when discussing features in the Python programming language.

Is it better to have a more complicated notation that is harder to learn that leads to simpler descriptions?

Even recursive EBNF rules are not powerful enough to describe all simple symbols. For example, they cannot describe symbols having some number of As, followed by the same number of Bs, followed by the same number of Cs: $\texttt{A}^n\texttt{B}^n\texttt{C}^n$, where $n \geq 0$. To specify such a description, we need a more powerful notation: type 1 or type 0 in the Chomsky Hierarchy; programming languages like Python are type 0, the most complicated/powerful in the hierarchy. So why do we use EBNF? Because it is the simplest notation that is powerful enough to describe the syntactic structures in a typical programming language.

Recursive EBNF rules have their limits too

1.8.1 Describing EBNF using Recursive EBNF rules

EBNF descriptions are powerful enough to describe their own syntax. Although such an idea may seem odd at first, recall that dictionaries use English to describe English. The EBNF rules describing EBNF illustrate mutual recursion: although no rule is directly recursive, the RHS of *rhs* is defined in terms of *sequence*, whose RHS is defined in terms of *option* and *repetition*, whose RHSs are defined in terms of *rhs*. Thus, these rules are mutually described in terms

EBNF is powerful enough to describe itself using mutual recursion

[5] The EBNF rule r is "tail–recursive": the recursive reference occurs at the end of an alternative. All "tail–recursive" EBNF rules can be replaced by equivalent EBNF rules that use repetition; but not all recursive rules tail–recusive; *eq* isn't.

of each other.

For easier reading, these rules are grouped into three categories: character–set related, LHS/RHS related (mutually recursive), and EBNF related. Recall that when a boxed character appears in an EBNF rule, it stands for itself, not its special meaning in EBNF. The empty symbol appears as an empty box.

The description is grouped into 3 categories, and the later rules use the names of the earlier rules

EBNF Description: *ebnf_description*

lower ⇐ a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
upper ⇐ A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
digit ⇐ 0|1|2|3|4|5|6|7|8|9
special ⇐ -|_|"|#|&|’|(|)|*|+|,|.|/|:|;|<|=|>
character ⇐ *lower* | *upper* | *digit* | *special*
empty ⇐ $\boxed{}$

lhs ⇐ *lower*{[_]*lower*}
option ⇐ $\boxed{\texttt{[}}$*rhs*$\boxed{\texttt{]}}$
repetition ⇐ $\boxed{\texttt{\{}}$*rhs*$\boxed{\texttt{\}}}$
sequence ⇐ *empty* | {*character* | *lhs* | *option* | *repetition*}
rhs ⇐ *sequence*{$\boxed{|}$*sequence*}

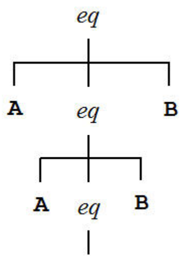
ebnf_rule ⇐ *lhs* ⇐ *rhs*
ebnf_description ⇐ {*enbf_rule*}

Section Review Exercises

1. a. Write a tabular proof that shows AAAABBBB is a legal *eq*. b. Draw a derivation tree showing AABB is a legal *eq*.

   Answer:

Status	Reason (rule #)
eq	Given
A *eq* B	Replace *eq* by the second alternative in its RHS (1&2)
AA *eq* BB	Replace *eq* by the second alternative in its RHS (1&2)
AAA *eq* BBB	Replace *eq* by the second alternative in its RHS (1&2)
AAAA *eq* BBBB	Replace *eq* by the second alternative in its RHS (1&2)
AAAABBBB	Replace *eq* by the first (empty) alternative in its RHS (1&2)



2. Replace the *integer_list* EBNF rule by an equivalent directly recursive one.

   Answer: *integer_list* ⇐ *integer* | *integer*,*integer_list*

3. Rewrite the EBNF rule *ebnf* ⇐ {A[B|C]} as an equivalent BNF description.

   Answer:
   choice ⇐ | B | C
   bnf ⇐ | A *choice bnf*

Chapter Summary

This chapter examined the use of EBNF rules to describe syntax. It started by discussing EBNF descriptions, named rules, and the control forms used in their right–hand sides: sequence, choice, option, and repetition. We saw how to write three different kinds of proofs to demonstrate whether or not a symbol was legal according to an EBNF description: in English, and more formally as tabular proofs and derivation trees. We saw various EBNF descriptions throughout the chapter, and analyzed each according to it syntax and semantics. To be correct, EBNF descriptions must be inclusive enough to include all legal symbols, but restrictive enough to exclude all illegal symbols. Sometimes different EBNF rules are equivalent: they classify all symbols exactly the same: legal in both or illegal in both. We saw that syntax charts present exactly the same information contained in EBNF rules, but more graphically graphically: we should be able to convert descriptions back and forth between EBNF rules and their syntax charts. Finally, this chapter discussed recursive descriptions (direct and mutual recursion) and the latter's use in an EBNF description of EBNF descriptions.

Chapter Exercises

1. The control forms in each of the following pairs are not equivalent. Find the simplest (shortest) symbol that is classified differently by each control form in the pair. Hint: try small combinations of `A` and `B`.
   a1. [A][B]   b1. {A | B}   c1. [A][B]
   a2. [A[B]]   b2. {A} | {B}   c2. A | B   he following are legal on the top
   and illegal on the bottom: a. `B`, b. `AB`, c. `AB`

2. Simplify each of the following control forms (but preserve equivalence). For this problem, simpler means shorter or has fewer nested control forms.
   a. A | B | A   c. [A]{A}   e. [A|B] | [B|A]   g. A| AB
   b. [A[A[A]]]   d. [A]{C} | [B]{C}   f. {[A|B][B|A]}   h. A | AA | AAA | AAAA

3. Write an EBNF description for `phone`, which describes telephone numbers written according to the following specifications.

   - Normal: a three digit exchange, followed by a dash, followed by a four digit number: e.g., `555-1212`
   - Long Distance: a `1`, followed by a dash, followed by a three digit area code enclosed in parentheses, followed by a three digit exchange, followed by a dash, followed by a four digit number: e.g., `1-(800)555-1212`
   - Interoffice: an `8` followed by a dash followed by a four digit number: e.g., `8-2404`.

   The description should be compact, and each rule should be well named.

4. Write an EBNF description for *sci_not*, numbers written in scientific notation, which scientists and engineers use to write very large and very small numbers compactly. Avogadro's number is written 6.02252xl0↑23 and read as 6.02252 —called the mantissa— times 10 raised to the 23rd power —called the exponent. Likewise, the mass of an electron is written 9.11xl0↑–31 and earth's gravitational acceleration constant is written 9.8 —this number is pure mantissa; it is not required to be multiplied by any power of ten. Numbers in scientific notation always contain at least one digit in the mantissa; if that digit is nonzero:

   - It may have a plus or minus sign preceding it.
   - It may be followed by a decimal point, which may be followed by more digits.
   - It may be followed by an exponent that specifies multiplication by ten raised to some non-zero unsigned or signed integer power.

The symbols `0.5`, `15.2`, `+0.0x10↑5`, `5.3x10↑02`, and `5.3xlO↑2.0` are all illegal in scientific notation. Hint: my solution uses a total of five EBNF rules: *non_0_digit*, *digit*, *mantissa*, *exponent*, and *sci_not*.

5. a. Write an EBNF description for `list`, which shows a list of `X`s punctuated according to the following rule: one `X` by itself, two `X`s separated by `and`, or a series of $n \geq 3$ `X`s where the first $n-1$ are separated by commas, with `and` separating the last two. Legal: `X`; `X and X`; `X, X and X`; and `X, X, X and X`. Illegal: empty symbol; `X,`; `X, X`; `X, and X`; `X, X, and X`; commas in other strange places.

   b. Write an EBNF description for `list`, which shows a list of `X`s punctuated according to the following rule: one `X` by itself, two `X`s separated by `and`, or a series of $n \geq 3$ `X`s where the first $n-1$ are ended by commas, with `and` appearing before the last `X`. Legal: `X`; `X and X`; `X, X, and X`; and `X, X, X, and X`. Illegal: empty symbol; `X,`; `X, X`; `X, and X`; `X, X and X`; commas in other strange places.

6. Write an EBNF description for `comma_integer`, which includes normalized unsigned or signed integers (no extraneous leading zeros) that have commas in all the correct places (separating thousands, millions, billions, etc.) and nowhere else. Legal: `0`; `213`; `-2,048`; and `1,000,000`. Illegal: `-0`; `062`; `0,516`; 05,418; `54,32,12`; and `5,,123`. Hint: What can you say is always true in legal numbers with commas and triples of digits?

7. Using the following rules, write an EBNF description for `train`. A single letter stands for each car in a train: `E`ngine, `C`aboose, `B`oxcar, `P`assenger car, and `D`ining car. There are four rules specifying how to form trains.

   - One or more Engines appear at the front; one Caboose at the end.
   - Boxcars always come in pairs: `BB`, `BBBB`, etc.
   - There cannot be more than four Passenger cars in a series.
   - One dining car must follow each series of passenger cars.

   These cars cannot appear anywhere other than these locations. Here are some legal and illegal exemplars.

Train	Analysis
EC	Legal: the smallest train
EEEPPDBBPDBBBBC	Legal: a train showing all the cars
EEBB	Illegal: no caboose (everything else OK)
EBBBC	Illegal: three boxcars in a row
EEPPPPPDBBC	Illegal: more than four passenger cars in a row
EEPPBBC	Illegal: no dining car after passenger cars
EEBBDC	Illegal: dining car after box car

8. The interaction of two syntactic structures can sometimes have an unexpected problematic semantic interaction. Briefly describe a bad interaction in an *integer_set* that specifies values that are *comma_integer*s.

9. A "range" is a compact way to write a sequence of integers. We will use the symbol `..` to mean "up through", so the range `2..5` denotes the values `2, 3, 4, and 5`: the values `2` up through `5` inclusive at both ends. Using such a notation, we can write sets more compactly: instead of `{2, 3, 4, 5, 8, 10, 11, 12, 13, 17, 18, 19, 21}` we could write

{`2..5, 8, 10..13, 17..19, 21`}. Semantically, for any range x`..`y the meaning of the range is

$x \leq y$ The range x`..`y is equivalent to all integers between x and y inclusive. By this rule, the range x`..`y is equivalent to just the value x.

$x > y$ The range x`..`y is equivalent to the "empty" range and contains no values.

By convention, we do not use ranges to write single values nor ranges of two values (`1,2` is more compact than `1..2`). With can define *integer_range* and use it to update our *integer_set* EBNF description.

EBNF Description: *integer_set* (updated to use *range)*

integer_range ⇐ *integer*[..*integer*]
integer_list ⇐ *integer_range*{,*integer_range*}
integer_set ⇐ $\boxed{\texttt{\{}}$[*integer_list*]$\boxed{\texttt{\}}}$

a. Given the semantics of sets of ranges, convert the following sets into canonical form, using ranges when appropriate.

a. {`1,5,9,3,7,11,9`}
b. {`1..3,8,5..9,4`}
c. {`8,1,2,3.4,5,12,13,14,10`}
d. {`2..5,7..10,1`}
e. {`1..3,8,2..5,12,4`}
f. {`4..1,12,2,7..10,6`}

g. The following EBNF description for *integer-set* is more compact than the previous one. But, they are not equivalent: this definition allows more sets than the previous definition. Find one of these sets.

EBNF Description: *integer_set* (more compact, but not equivalent)

integer_list ⇐ *integer*{`,`*integer*|`..`*integer*}
integer_set ⇐ $\boxed{\texttt{\{}}$[*integer_list*]$\boxed{\texttt{\}}}$

10. a. Write a directly recursive EBNF rule named *mp* that describes all symbols that have matching parentheses. Legal: `()`, and `()()()`, and `()(()())`, and `((()))())(()(()))()`. Illegal: `(`, and `())`, and `(()()`.
b. Show a tabular proof and its derivation tree proving `()(()())` is legal.

11. I once saw a bumper sticker that said, "Sucks Syntax". What is the joke?