It goes against the grain of modern education to teach students to program. What fun is there to making plans, acquiring discipline, organizing thoughts, devoting attention to detail, and learning to be self critical?

A. PERLIS

EBNF is a notation for formally describing syntax: how to write entities in a language. We use EBNF throughout this book to describe the syntax of Ada. But there is a more compelling reason to begin our study of programming with EBNF: it is a microcosm of programming. First, there is a strong similarity between the control forms in EBNF rules and the control structures in Ada: sequence; decision, repetition, recursion, and the ability to name descriptions. There is also a strong similarity between the process of writing descriptions in EBNF and writing programs in Ada: we must synthesize a candidate solution and then analyze it — to determine whether it is correct and easy to understand. Finally, studying EBNF introduces a level of formality that will continue throughout our study of programming and Ada.

OBJECTIVES

— Learn the control forms in EBNF: sequence, choice, option, and repetition

— Learn how to read and write syntactic descriptions with EBNF rules

— Explore the difference between syntax and semantics

— Learn the correspondence between EBNF rules and Syntax Charts

2.1 Languages and Syntax

FORTRAN and ALGOL

In the middle 1950's, computer scientists began to design high-level programming languages and build their compilers. The first two major successes were FORTRAN (FORmula 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, Naur, BNF and EBNF

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 Ada's syntax, because it results in more compact descriptions.

Chomsky's language hierarchy

In a parallel development, the linguist Noam Chomsky began work on the harder problem of 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 up through 0 in the Chomsky hierarchy. The power of Chomsky's type 2 notation is equivalent to BNF and 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.

2.2 EBNF Descriptions and Rules

EBNF descriptions via EBNF rules

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

character ⇐ separating the two sides; read this symbol as "is defined as". The LHS contains one (possibly hyphenated) word written in lower-case; it names the EBNF rule. The RHS supplies the definition associated with this name. It can include combinations of the four control forms explained in Table 2.1.

Table 2.1 Control Forms of Right-Hand Sides in EBNF Rules

Sequence	items appear left-to-right; their order is important
Choice	alternative items are separated by a | (stroke); one item is chosen from this list of alternatives; their order is unimportant
Option	an optional item is enclosed between [and] (square brackets); the item can either be included or discarded
Repetition	a repeatable item is enclosed between { and } (curly braces); the item can be repeated **zero** or more times

Characters with special meanings

EBNF rules can include six characters with special meanings: ⇐, |, [,], {, and }. Except for these characters and the names of EBNF rules, anything that appears in a RHS stands for itself: letters, digits, punctuation, parentheses, and any other printable characters.

An EBNF Description of Integers

A first EBNF description

The following EBNF rules describe how to write simple integers.* Their right-hand sides illustrate every control form available in EBNF.

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

An English interpretation of the control forms in these EBNF rules

— A `digit` is defined as any 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 the character + or -), followed by any `digit`, followed by a repetition of zero or more `digit` — each repeated `digit` is independently chosen from the list of alternatives in the `digit` rule. The RHS of `integer` combines all the control forms: sequence, option, choice, and repetition.

Abstraction via named EBNF rules

We can abstract the structure of an `integer` — where each `digit` appears — independently from the definition of `digit` — which defines only the possible choice of characters. For example, an `integer` written in base 2 has this same structure, even though the `digit` rule would be restricted to the choices 0 and 1.

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

Typesetting conventions

To make EBNF descriptions easy to read, we align their rule names, the ⇐ and 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 order in which EBNF rules appear is irrelevant, it is useful to write the rules in order of increasing complexity: with the right-hand sides in later EBNF rules referring to the names in the left-hand sides of earlier ones. Using this convention, the last EBNF rule names the main syntactic entity being described.

Language lawyers

Given an EBNF description, we must learn to interpret it as a language lawyer: determining whether a symbol — any sequence of characters — is legal or illegal according to the EBNF rules in that description. Computers perform expertly as language lawyers, even on the most complicated descriptions.

Proving Symbols Match EBNF Rules

Matching symbols with EBNF rules

To prove that a symbol is an `integer` we must match its characters with the characters in the `integer` rule, according to that rule's control forms. If there is an exact match, we recognize the symbol as a legal `integer`; otherwise we classify the symbol as illegal, according to the `integer` description.

7 is an `integer`

Proofs in English: To prove that the symbol `7` is an `integer`, we must start in the `integer` EBNF rule with the optional sign; in this case we discard the option. Next in the sequence, the symbol must contain a character that we can recognize as a `digit`; in this case we choose the `7` alternative from the RHS of the `digit` rule. Finally, we must repeat `digit` zero or more times; in this case we use zero repetitions. Every character in the symbol `7` has been matched against every character in the `integer` EBNF rule, according to its control forms. Therefore, we recognize `7` as a legal `integer` according to this EBNF description.

`+142` is an `integer`

We use a similar argument to prove that the symbol `+142` is an `integer`. Again we must start with the optional sign; in this case we include the option and choose the `+` alternative. Next in the sequence, the symbol must contain a character that we can recognize as a `digit`; in this case we choose the `1` alternative from the RHS of the `digit` rule. Finally, we must repeat `digit` zero or more times; in this case we use two repetitions. For the first `digit` we choose the `4` alternative, and for the second `digit` the `2` alternative: recall that each time we encounter a `digit`, we are free to choose any of its alternatives. Again, every character in the symbol `+142` has been matched against every character in the `integer` EBNF rule, according to its control forms. Therefore, `+142` is a also a legal `integer`.

`1,024` `A5` and `15-` are each Illegal

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. Likewise for the letter `A` in the symbol `A5`. 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 the minus follows the last `digit`, but the sequence in the `integer` rule requires that the sign precede the first `digit`. So according to our EBNF rules, none of these symbols

is recognized as a legal integer.* When viewing symbols as a language lawyer, we cannot appeal to intuition; we must rely solely on the EBNF description we are matching.

Proof rules

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 syntactic entity we are trying to match (in this example, integer); the last line must be the symbol we are matching. Each line is derived from the previous one according to the following rules.

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

Simplifying proofs

Combining rules 1 and 2 simplifies our proofs by allowing us to replace a left-hand side by one of the alternatives in its right-hand side in a single step. Figure 2.1 contains a tabular proof showing +142 is an integer.

A graphic illustration of proofs

Derivation Trees: We can illustrate a tabular proof more graphically by writing it as a **derivation tree**. The downward branches in such a tree correspond to the same 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 original symbol appears at the bottom of a derivation tree, when its characters are read left to right. Figure 2.1 contains a derivation tree showing +142 is an integer.

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

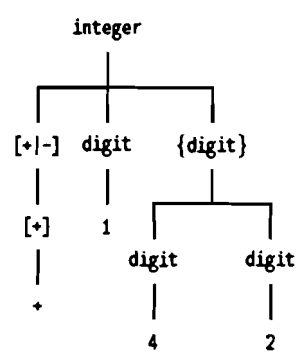


Figure 2.1 A Tabular Proof and its Derivation Tree showing +142 is an integer

*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 base 16, and the third is a negative number — sometimes written this way by accountants.

REVIEW QUESTIONS

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

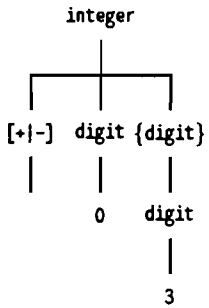
a. +42	e. -1492	i. 28	m. 0	q. +-7
b. +	f. 187	j. 187.0	n. forty two	r. 1 543
c. -0	g. drei	k. $15	o.	s. 1+1
d. VII	h. 25¢	l. 1000	p. 555-1212	t. 000193

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 03 is a legal integer.

Answer: Note how the omitted option ([+|-]) is drawn in the derivation tree.

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



2.3 More Examples of EBNF

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

Identical versus equivalent

```
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.

Names for rules do not change their meanings

*The words "identical" and "equivalent" have distinct meanings. Identical means "are exactly the same". Equivalent means "are the same within some context". Any two dollar bills have identical buying power. A dollar bill has equivalent buying power to four quarters in most contexts; but in a vending machine that requires exact change, it does not.

```
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.

Equivalent proofs

Complicated EBNF descriptions are easier to read and understand if they are named well: each name intuitively communicating 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.

Good rule names

Incorrect EBNF Descriptions for integer

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. The best description is the simplest one; so if these rules are equivalent to the previous ones, we have improved the description of integer.

A simpler syntax for integer? No!

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

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, choosing the plus; repeat digit three times, choosing 1, 4, and 2.

Almost equivalent

But there are two symbols that this description recognizes as legal, while the previous descriptions classify them as illegal: + and - (signs without following digits). The previous integer rules all require one digit followed by zero or more others; 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 plus; then repeat digit zero times. The proof for - is similar. Even the "empty symbol", which contains no characters, is recognized by this EBNF description as a legal integer: discard the sign option, then repeat digit zero times. Because of these three differences, this EBNF description of integer is not equivalent to the previous ones.

Three differences

Next we address the problem of describing how to write numbers with embedded commas: 1,024. We can easily extend the digit rule to allow a comma as one of its alternatives.

Numbers with 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: repeat comma-digit four times, choosing 1 , 0 2 and 4. But we can also prove that 1,,3,4 is a legal comma-integer, by using six repetitions

Good news, bad news

of comma-digit and choosing a comma for the second, third and fifth ones. By treating a comma as if it were a digit, numbers with well-placed commas are recognized as legal, but symbols with randomly placed commas also become legal. See Exercise 8 for a correct solution to this problem.

REVIEW QUESTIONS

1. Are 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 the sign in integer, or around +|- 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 recognize one or more digits.

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

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 is unsigned. Write an EBNF description for normalized-integer: 0, -1, and 193 are legal, but -01, 000193, +0, and -0 are not.

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}
```

2.4 Syntax versus Semantics

Form versus meaning

EBNF descriptions specify only syntax: the form in which some entity 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 out from the Sun" has an obvious meaning, but its meaning is contradicted by known astronomical facts.

Symbols and meanings

Two semantic issues are important in programming languages

— Can two different symbols have the same meaning?
— Can a symbol have two different meanings?

Two symbols, one meaning

One symbol, two meanings

The first issue is easy to illustrate. Everyone has a nickname; so two names can refer to the same person. The second issue is a bit more subtle; here the symbol we analyze is a sentence. Suppose you take a course that meets on 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 that meets every weekday. If your instructor for this 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 sentence may depend on its context.

Semantics and EBNF Descriptions

The semantics of `integer` in mathematics

Now we examine these semantic issues in relation to the 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 an even more special case, the symbols 0 and +0 and -0 all have the same value: the sign of zero is irrelevant. Finally, the symbols 000193 and 193 both have the same meaning: leading zeros do not effect a number's value.

Alternative `integer` semantics

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.

The semantics of measurements and pronunciation

Another example concerns how measurements are written: although 9.0 and 9.00 represent the same value, the former may indicate the quantity was measured to only two significant digits; the latter to three. Finally, when saying the name "Rich" and the adjective "rich", the upper-case letter is pronounced the same as the lower-case one.

The syntax and semantics of structured integers

Structured integers contain embedded underscores that separate groups of digits, indicating some important structure. We can use them to encode real-world information in an easy to read form: dates 2_10_1954, phone numbers 1_800_555_1212, and credit card numbers 314_159_265_358. Underscores can appear only between digits — not as the first or last character, and they cannot be adjacent. Here is an EBNF description that captures exactly these requirements.

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

Semantics of underscores

Semantically, underscores do not affect the value of a `structured-integer`: 1_800_555_1212 has the same meaning as 18005551212; when dialing either number, we press keys only for the characters representing a `digit`.

Describing semantics

In summary, EBNF descriptions specify syntax not semantics. When we describe the syntax of an Ada feature in EBNF, we will describe its semantics using a mixture of English definitions and illustrations. Computer scientists are just beginning to develop notations that describe the semantics of programming languages in simple and precise ways. In general, meaning is much harder to quantify than form.

REVIEW QUESTIONS

1. a. Find two dates that have the same meaning, when written naturally as structured integers. b. Propose a new format 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. Write the first date as 12_05_1987; write the second as 1_25_1987. These structured integers have different values.

2.5 Syntax Charts

Representing EBNF rules graphically

A **Syntax Chart** (SC) is a graphical notation for writing a syntax description. Figure 2.2 illustrates how to translate each EBNF control form into its equivalent SC. In each case we must follow the arrows from the beginning of the picture on the left, to its end on the right. In a sequence, we must go through each item. In a choice, we must go through one rung in a ladder of alternatives. In an option we must go through either the top rung containing the item, or the bottom that does not. Repetition is like option, but we can loop through the item in the top rung; this picture is the only one with a right-to-left arrow.

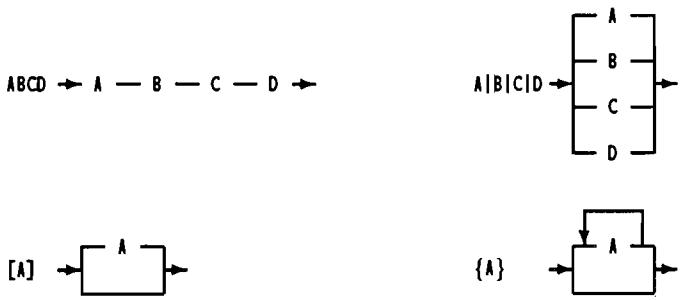


Figure 2.2 Translating EBNF rules into Syntax Charts

Combining and composing syntax charts

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 one big SC that contains no named EBNF rules, by replacing each named LHS with the SC for its RHS. Figure 2.3 shows the SC equivalents of the digit and original integer EBNF rule, and one large composed SC for integer.

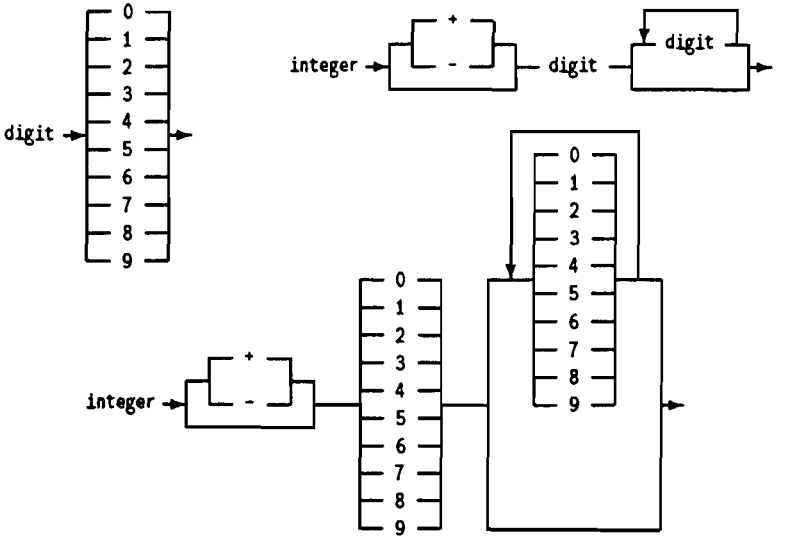


Figure 2.3 Syntax Charts for digit and integer, and a composed integer

Disambiguation of EBNF rules

The syntax charts in Figure 2.4 illustrate the RHS of three interesting EBNF rules. The first shows a repetition of two alternatives, where any number of intermixed As and Bs are legal: AAA, BB, or BABBA. A different choice can be made for each repetition. The second shows two alternatives where a repetition of As or a repetition of Bs are legal: AAA or BB, but not AB. Once the choice is made, only one of the characters can be repeated. Any symbol legal in this rule is legal in the first, but not vice versa.

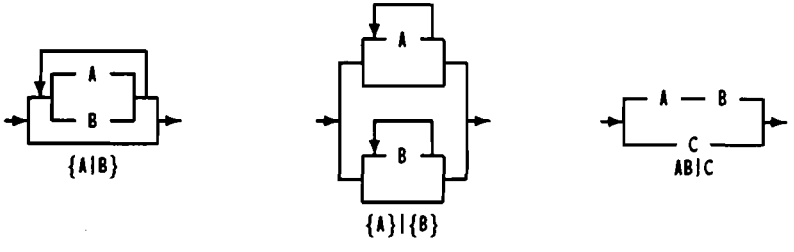


Figure 2.4 Syntax Charts Disambiguating Interesting EBNF Rules

A factoring technique

The third 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 both alternatives fully or use a second rule to "factor out" the alternatives.

```
simple ⇐ AB | AC          tail   ⇐ B | C
                          simple ⇐ Atail
```

EBNF versus 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, SCs are easier to use when classifying symbols; for advanced students EBNF descriptions, which are smaller, are easier to read and understand. Because beginning students become advanced ones, this book uses EBNF rules to describe Ada's syntax.

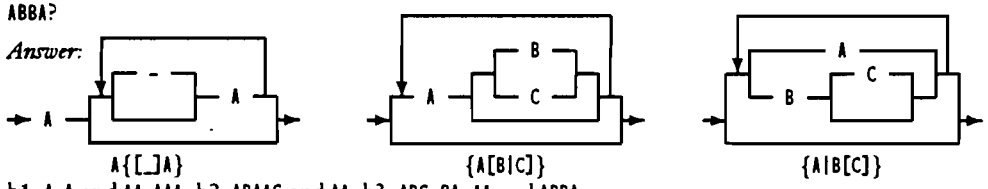
REVIEW QUESTIONS

1. a. Translate each of the following right-hand sides into a syntax chart.

   `A{[_]A}` `{A[B|C]}` `{A|B[C]}`

   b. Which symbols does the first RHS recognize as legal: `A_A`, `AA_AAA`, `_A`, `A_`, `A__A`? Which symbols do the second and third RHS recognize as legal: `ABAAC`, `ABC`, `BA`, `AA`, `ABBA`?

Answer:



b1. `A_A` and `AA_AAA`. b2. `ABAAC` and `AA`. b3. `ABC`, `BA`, `AA` and `ABBA`.

2.6 EBNF Descriptions of Sets

The syntax of sets

This section explores the syntax and semantics for writing sets of integers. Such sets start and end with parentheses, and contain zero or more integers separated by commas. The empty set `()`, a singleton set `(3)`, and a set containing the elements `(5,-2,11)` are all legal. Sets are illegal if they omit the parentheses, contain consecutive commas `(1,,3)` or extra commas `(,2)` or `(1,2,3,)`.

EBNF for sets

Given a description of `integer`, the following EBNF rules describe such sets. Note that the parentheses characters in `integer-set` stand for themselves; they are not used for grouping or any other special EBNF purpose.

```
integer-list ⇐ integer{,integer}
integer-set  ⇐ ([integer-list])
```

Proofs using `integer-set`

We can easily prove that the empty set is a legal `integer-set`: discard the `integer-list` option between the parentheses. For a singleton set we include this option but take zero repetitions after the first `integer` in `integer-list`. Figure 2.5

proves that (5,-2,11) is a legal integer-set. The tabular proof and its derivation tree are shortened by recognizing in one step that 5, -2, and 11 are each an integer. Like lemmas in mathematics, we use this information without proving it.

Status	Reason
integer-set	Given
([integer-list])	Replace integer-set by its RHS
(integer-list)	Include option
(integer{,integer})	Replace integer-list by its RHS
(5{,integer})	Lemma: 5 is an integer
(5,integer,integer)	Use two repetitions
(5,-2,integer)	Lemma: -2 is an integer
(5,-2,11)	Lemma: 11 is an integer

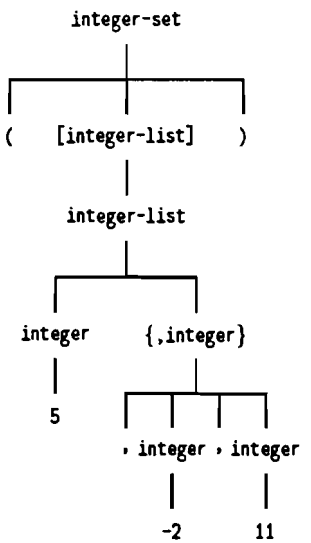


Figure 2.5 A Tabular Proof and its Derivation Tree showing (5,-2,11) is an integer-set

Set semantics

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

— Duplicate elements are irrelevant and can be removed: (1,3,5,1,3,3,5) is equivalent to (1,3,5).

— The order of the elements is irrelevant and can be rearranged: (1,5,3) is equivalent to (1,3,5) and (3,1,5) and all other permutations of these values.

Canonical sets

By convention, we write sets in an ordered form, starting with the smallest element and ending with the largest; each element is written 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.

An equivalent description

The following EBNF rules are an equivalent description for writing sets. Here, the option brackets are in the integer-list rule, not the integer-set rule.

```
integer-list ⇐ [integer{,integer}]
integer-set  ⇐ (integer-list)
```

Stylistic preferences

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 is in the other. Second, the new description

allows 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.

Sets containing Integer Ranges

The structure of ranges and sets

A range is a compact way to write a sequence of integers. Using the symbol .. to represent "up through", the range 2..5 specifies 2, 3, 4, and 5: the values 2 up through 5 inclusive. Using such a notation, we can write sets more compactly: (2..5,8,10..13,17..19,21) instead of (2,3,4,5,8,10,11,12,13,17,18,19,21). The following EBNF rules extend our description of integer-set to include ranges.

```
integer-range ⇐ integer[..integer]
integer-list  ⇐ integer-range{,integer-range}
integer-set   ⇐ ([integer-list])
```

Figure 2.6 proves that (1,3..7,15) is a legal integer-set.

Range semantics

Now we switch our focus to semantics and examine the exact meaning of ranges. For every pair of integers X and Y:

X ≤ Y: the range X..Y is equivalent to all integers between X and Y inclusive. By this rule, every integer X is equivalent to the range X..X.

X > Y: the range X..Y is the "null range" and contains no values.

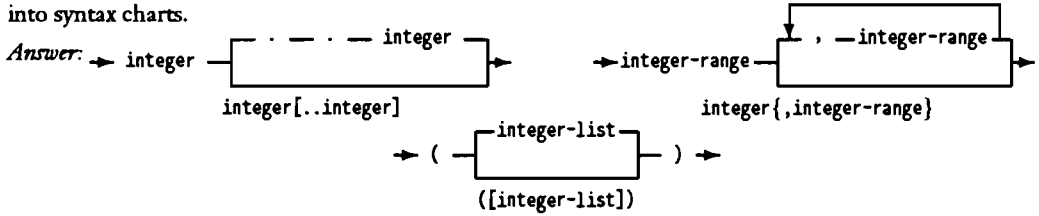
Canonical ranges

By convention, we do not use ranges to write single values nor ranges of two values (1,2 is more compact than 1..2) unless there are special reasons to do so.

REVIEW QUESTIONS

1. Translate the RHS of the integer-range, integer-list, and integer-set EBNF rules into syntax charts.

Answer:



2. Convert the following sets into canonical form; use ranges when appropriate.

   a. (1,5,9,3,7,11,9) c. (8,1,2,3,4,5,12,13,14,10) e. (1..3,8,2..5,12,4)
   b. (1..3,8,5..9,4) d. (2..5,7..10,1) f. (4..1,12,2,7..10,6)

   Answer:
   a. (1,3,5,7,9,11) c. (1..5,8,10,12..14) e. (1..5,8,12)
   b. (1..9) d. (1..5,7..10) f. (2,6..10,12)

3. The following EBNF description for integer-set is more compact than the original, but they are not equivalent. This one recognizes all the sets recognized by the original description, but it recognizes others as well; find one of these sets.

```
integer-list ⇐ integer{,integer|..integer}
integer-set  ⇐ ([integer-list])
```

Answer: This description allows "ranges" with more than one .. in them: (1..3..5).

Status	Reason
integer-set	Given
([integer-list])	Replace integer-set by its RHS
(integer-list)	Include option
(integer-range{,integer-range})	Replace integer-list by its RHS
(integer[..integer]{,integer-range})	Replace integer-range by its RHS
(integer{,integer-range})	Discard option
(1{,integer-range})	Lemma: 1 is an integer
(1,integer-range,integer-range)	Use two repetitions
(1,integer[..integer],integer-range)	Replace integer-range by its RHS
(1,3[..integer],integer-range)	Lemma: 3 is an integer
(1,3..integer,integer-range)	Include option
(1,3..7,integer-range)	Lemma: 7 is an integer
(1,3..7,integer[..integer])	Replace integer-range by its RHS
(1,3..7,15[..integer])	Lemma: 15 is an integer
(1,3..7,15)	Discard option

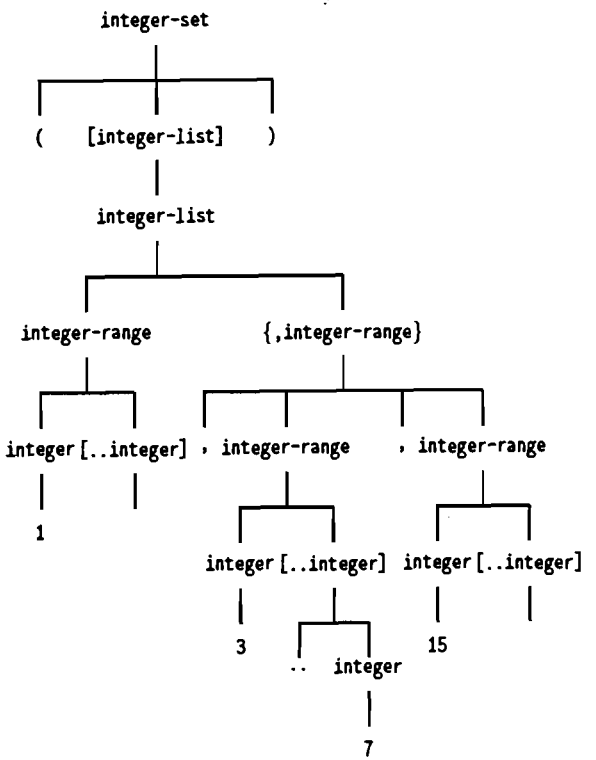


Figure 2.6 A Tabular Proof and its Derivation Tree showing (1,3..7,15) is an integer-set

2.7 Advanced EBNF (optional)

Recursion, and EBNF in EBNF

This section examines two advanced concepts in EBNF: recursive EBNF rules and using recursive EBNF rules to describe EBNF. In programming, recursion is a useful technique for specifying complex data structures and the subprograms that manipulate them.

Recursive EBNF Descriptions

Direct recursion

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 following directly recursive rule recognizes symbols containing any number of As, which we can describe mathematically as $A^n, n \geq 0$.

```
r1 ⇐ | A r1
```

A non-recursive alternative

The first alternative in this rule contains the empty symbol, which is recognized as a legal r1. Directly recursive rules must include at least one alternative that is not recursive, otherwise they describe only infinite-length symbols.

The meaning of recursive rules

The second alternative means that an A preceding anything that is recognized as an r1 is also recognized as an r1: so A is recognized as an r1 because it has an A preceding the empty symbol; likewise AA is also recognized as an r1, as is AAA, etc. Figure 2.7 is a tabular proof and its derivation tree, showing how AAA is recognized as a legal r1. If we require at least one A, this rule can be written more understandably as r1 ⇐ A | A r1, with A as the non-recursive alternative.

Status	Reason
r1	Given
A r1	Replace r1 by the second alternative in its RHS
AA r1	Replace r1 by the second alternative in its RHS
AAA r1	Replace r1 by the second alternative in its RHS
AAA	Replace r1 by the first (empty) alternative in its RHS

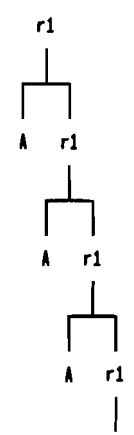


Figure 2.7 A Tabular Proof and its Derivation Tree showing AAA is an r1

Recursion and repetition

The recursive r1 rule is equivalent to r1 ⇐ {A}, which uses repetition instead. Recursion can always replace repetition, but the converse is not true,* because recursion is more powerful than repetition. For example, examine the following directly recursive EBNF description. It recognizes all symbols having some number of As followed by the same number of Bs: $A^nB^n, n \geq 0$. The description of these symbols cannot be written without recursion.

```
r2 ⇐ | Ar2B
```

The rule r2 ⇐ {A}{B} does not require the same number of As as Bs.

EBNF versus BNF

We just learned that repetition can always be replaced by recursion in an EBNF rule. We can also replace any option control form by an equivalent choice control form that contains an empty symbol. Using both techniques, we can rewrite our original integer description, or any other one, using only recursion, the choice control form, and the empty symbol. EBNF without the repetition or option control form extensions is called just BNF. The structure of each BNF rule is simpler, but descriptions written using them are often longer.

```
sign    ⇐ | + | -
digit   ⇐ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digits  ⇐ | digit digits
integer ⇐ sign digit digits
```

Recursive EBNF rules are limited too: $A^nB^nC^n, n \geq 0$

Even recursive EBNF rules are not powerful enough to describe all simple languages. 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: $A^nB^nC^n, n \geq 0$. To specify such a description, we need a more powerful notation: type 1 or type 0 in the Chomsky Hierarchy.

Describing EBNF using EBNF rules

Mutual recursion

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, all these rules are mutually described in terms of each other.

Boxed characters

For easier reading, these rules are grouped into three categories: character set related, LHS/RHS related (mutually recursive), and EBNF related. When a boxed character appears in a rule, it stands for itself, not its special meaning in EBNF: [|] denotes the stroke character, it does not separate alternatives in the rule in which it appears. The empty symbol appears as an empty box.

*The EBNF rule r1 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.

```
lower-case      ⇐ 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-case      ⇐ 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
other-character ⇐ -|_|"|#|&|'|(|)|*|+|,|.|/|:|;|<|=|>
character       ⇐ lower-case | upper-case | digit | other-character
empty-symbol    ⇐ []

lhs             ⇐ lower-case{[-]lower-case|[-]digit}
option          ⇐ [rhs]
repetition      ⇐ {rhs}
sequence        ⇐ empty-symbol | {character | lhs | option | repetition}
rhs             ⇐ sequence{|sequence}

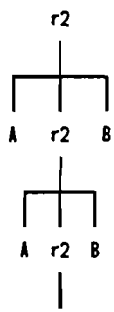
ebnf-rule       ⇐ lhs⇐rhs
ebnf-description ⇐ {ebnf-rule}
```

REVIEW QUESTIONS

1. a. Write a tabular proof that shows AAAABBBB is a legal r2. b. Draw a derivation tree showing AABB is a legal r2.

   Answer:

Status	Reason
r2	Given
A r2 B	Replace r2 by the second alternative in its RHS
AA r2 BB	Replace r2 by the second alternative in its RHS
AAA r2 BBB	Replace r2 by the second alternative in its RHS
AAAA r2 BBBB	Replace r2 by the second alternative in its RHS
AAAABBBB	Replace r2 by the first (empty) alternative in its RHS



2. Replace the integer-list rule by an equivalent directly recursive one.

   Answer: integer-list ⇐ integer | integer-list,integer

3. Rewrite: ebnf ⇐ {A[B|C]} as an equivalent description in BNF.

   Answer:

```
choice ⇐ | B | C
bnf    ⇐ | Achoicebnf
```

SUMMARY

EBNF descriptions

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 the right-hand sides of these rules: sequence, choice, option, and repetition. Proofs showing how a symbol matches an EBNF description were illustrated in English, and more formally as tabular proofs and their derivation trees. Throughout the chapter various EBNF descriptions of integers, ranges, and sets were proposed and analyzed according to their syntax and semantics. EBNF descriptions must be liberal enough to include all legal symbols, but restrictive enough to exclude all illegal symbols. Syntax Charts were seen to present graphically the same information contained in EBNF rules. Finally, this chapter

introduced recursive descriptions (direct and mutually recursive) and the latter's use in an EBNF description of EBNF.

EXERCISES

1. Write an EBNF description for `phone-number`, which describes telephone numbers written according to the following specifications. The description should be compact, and each rule should be well named.

   — Normal: a three digit exchange, followed by a dash, followed by a four digit number: `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: `1-(800)555-1212`

   — Interoffice: a `#` followed by either a 3 or a 5, followed by a dash, followed by a four digit number: `#5-1212`

2. The control forms in each of the following pairs are not equivalent. Find the simplest symbol that is classified differently by each control form in the pair.

   a1. `[A][B]` b1. `{A | B}` c1. `A | B`
   a2. `[A[B]]` b2. `{A} | {B}` c2. `[A][B]`

3. Simplify each of the following control forms (but preserve equivalence). For this problem, simpler means shorter or has fewer nested 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|AAAAA`

4. Write an EBNF description for numbers written in scientific notation, which scientists and engineers use to write very large and very small numbers compactly. Avogadro's number is written `6.02252x10↑23` and read as `6.02252` — called the mantissa — times 10 raised to the 23^{rd} power — called the exponent. Likewise, the mass of an electron is written `9.11x10↑-31` and earth's gravitational constant is written `9.8` — this number is pure mantissa; it is not multiplied by any power of ten.

   Numbers in scientific notation always contain at least one digit in the mantissa; if that digit is nonzero: (1) it may have a plus or minus sign preceding it, (2) it may be followed by a decimal point, which may be followed by more digits, and (3) 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 `5.3x10↑2.0` are illegal in scientific notation. **Hint**: my solution uses a total of five EBNF rules: `non-0-digit`, `digit`, `mantissa`, `exponent`, and `scientific-notation`.

5. Identify one advantage of writing dates as a `structured-integer` in the form: year, month, day (`1954_02_10`) instead of in the normal order (`02_10_1954`).

6. The following EBNF rules attempt to describe every possible `family-relation`. Here spaces are important, so all the characters do not run together.

```
contemporary        <= SISTER | BROTHER | WIFE     | HUSBAND
ancestor-descendent <= MOTHER | FATHER  | DAUGHTER | SON | NEPHEW | NIECE
side-relation       <= AUNT | UNCLE | COUSIN
close-relation      <= contemporary | ancestor-descendent | side-relation
far-relation        <= {GREAT} side-relation | {GREAT} [GRAND] ancestor-descendent
family-relation     <= far-relation | [STEP] close-relation
```

a. Classify each of the following symbols as a legal or illegal `family-relation`.

i. SISTER	iv. GRAND MOTHER	vii. GREAT UNCLE
ii. UNCLE LARRY	v. GRAND UNCLE	viii. STEP STEP BROTHER
iii. GREAT GREAT GRAND SON	vi. GRAND NIECE	ix. GREAT MOTHER

b. Based on your classification in ix, improve the `far-relation` rule to always require a GRAND after the last GREAT.

7. We can extend the previous EBNF rules for describing family relationships.

```
person ⇐ ME | MY family-relation | THE family-relation OF MY family-relation
```

This description specifies symbols such as MY GRAND MOTHER and THE MOTHER OF MY MOTHER. These symbols may denote the same person, although MY GRAND MOTHER may also refer to THE MOTHER OF MY FATHER, THE MOTHER OF MY STEP FATHER, etc. Write a simpler symbol that is equivalent to each of the following; in some cases, more than one answer may be correct.

a. THE SISTER OF MY MOTHER	e. THE SON OF MY FATHER
b. THE GRAND SON OF MY MOTHER	f. THE DAUGHTER OF MY GREAT GRAND MOTHER
c. THE WIFE OF MY FATHER	g. THE GRAND SON OF MY GREAT GREAT GRAND MOTHER
d. THE GRAND FATHER OF MY SON	h. THE FATHER OF MY AUNT

8. Write an EBNF description for `comma-integer`, which includes normalized unsigned or signed integers (no extraneous leading zeros) that have commas in only the correct places (separating thousands, millions, billions, etc.): 0; 213; -2,048; and 1,000,000. It should not recognize -0; 062; 0,516; 05,418; 54,32,12; or 5,,123 as legal.

9. a. Write an EBNF description for `structured-integer-set` that specifies an `integer-set` (Section 2.6) allowing sets and ranges that use `structured-integer` (Section 2.4).
b. How can a similar `comma-integer-set` allowing sets and ranges that uses `comma-integer` (Exercise 8) lead to a semantic problem?

10. Using the following rules, write an EBNF description for `train`. Let letters stand for each car in a train: E for Engine, C for Caboose, B for Boxcar, P for Passenger car, and D for Dining car. The railroad has four rules telling how to form trains.

a. Trains start with one or more Engines and end with one Caboose; neither can appear anywhere else.

b. Whenever Boxcars are used, they always come in pairs: BB, BBBB, etc.

c. There cannot be more than four Passenger cars in a series.

d. A single dining car must follow each series of passenger cars; it cannot appear anywhere else.

Train	Analysis
EC	The smallest train
EEEPPDBBPDBBBBC	A train showing all the cars
EEEPPDBBPDBBBB	Illegal by rule a — no caboose
EBBBC	Illegal by rule b — 3 boxcars in a row
EEPPPPPDBBC	Illegal by rule c — 5 passenger cars in a row
EEPPBBC	Illegal by rule d — no dining car after passenger cars
EEBBDC	Illegal by rule d — dining car after box car

11. The following message was seen on a bumper sticker: **Stinks Syntax.** What is the joke?

12. a. Write a directly recursive EBNF rule named `mp` that describes all symbols that have matching parentheses: (), ()()(), ()(()()), and (((())())(()(()))()). It should not recognize (, ())(, or (()() as legal. b. Write a tabular proof and its derivation tree showing how ()(()()) is recognized as legal.