Context-Free Grammars

Grammars were invented by Chomsky in the mid-1950s for describing natural languages. In the late 1950s, one of Chomsky’s types of grammar (Type 2 or ***context-free*** grammar) was reinvented by Backus and proposed as a way to specify the syntax of the new language Algol.

Context-Free Grammars

Grammars were invented by Chomsky in the mid-1950s for describing natural languages. In the late 1950s, one of Chomsky's types of grammar (Type 2 or ***context-free*** grammar) was reinvented by Backus and proposed as a way to specify the syntax of the new language Algol.

This proposal was adopted in the **Algol 60 Report** (edited by Naur), an influential document considered to have done an excellent job of specifying Algol 60.

Context-Free Grammars

Grammars were invented by Chomsky in the mid-1950s for describing natural languages. In the late 1950s, one of Chomsky's types of grammar (Type 2 or ***context-free*** grammar) was reinvented by Backus and proposed as a way to specify the syntax of the new language Algol.

This proposal was adopted in the **Algol 60 Report** (edited by Naur), an influential document considered to have done an excellent job of specifying Algol 60.

The grammar notation used in the Algol 60 Report is now called "Backus Naur Form" or **BNF**.

Like many authors (but unlike Sethi) we use the term ***BNF*** more loosely, to simply mean "*a commonly used notation for writing context-free grammars*", and we refer to grammars written in such a notation as ***BNF specifications***.

$$
\begin{array}{rcl}
\langle expression \rangle & ::= & \langle expression \rangle + \langle term \rangle \\
& | & \langle expression \rangle - \langle term \rangle \\
& | & \langle term \rangle \\
\langle term \rangle & ::= & \langle term \rangle * \langle factor \rangle \\
& | & \langle term \rangle \, / \, \langle factor \rangle \\
& | & \langle factor \rangle \\
\langle factor \rangle & ::= & \textbf{number} \\
& | & \textbf{name} \\
& | & (\, \langle expression \rangle \,)
\end{array}
$$

A grammar written in BNF notation on p. 46 of Sethi (p. 47 in the course reader).

Figure 2.10 BNF syntactic rules for arithmetic expressions.

On p. 42, Sethi gives this equivalent grammar that is written in a similar notation. ***We will consider this notation to be BNF*,**

$$
\begin{array}{lcl}
E & ::= & E + T \mid E - T \mid T \\
T & ::= & T * F \mid T \, / \, F \mid F \\
F & ::= & \textbf{number} \mid \textbf{name} \mid (\, E \,)
\end{array}
$$

Figure 2.6 A grammar for arithmetic expressions.

$$
\begin{aligned}
\langle expression \rangle &::= \langle expression \rangle + \langle term \rangle \\
&\quad|\ \langle expression \rangle - \langle term \rangle \\
&\quad|\ \langle term \rangle \\
\langle term \rangle &::= \langle term \rangle * \langle factor \rangle \\
&\quad|\ \langle term \rangle \,/\, \langle factor \rangle \\
&\quad|\ \langle factor \rangle \\
\langle factor \rangle &::= \textbf{number} \\
&\quad|\ \textbf{name} \\
&\quad|\ (\ \langle expression \rangle\ )
\end{aligned}
$$

A grammar written in BNF notation on p. 46 of Sethi (p. 47 in the course reader).

Figure 2.10 BNF syntactic rules for arithmetic expressions.

On p. 42, Sethi gives this equivalent grammar that is written in a similar notation. ***We will consider this notation to be BNF***, even though it isn't exactly the same as the notation used in the Algol 60 Report and Sethi does not call it BNF.

$$
\begin{aligned}
E &::= E + T \mid E - T \mid T \\
T &::= T * F \mid T / F \mid F \\
F &::= \textbf{number} \mid \textbf{name} \mid (\ E\ )
\end{aligned}
$$

Figure 2.6 A grammar for arithmetic expressions.

We will use the term ***<u>grammar</u>*** to mean "context-free grammar"; we will not use other types of grammar.

-

-

-

-

We will use the term ***<u>grammar</u>*** to mean "context-free grammar"; we will not use other types of grammar.

- A grammar is relatively concise way to precisely specify certain (possibly infinite) *sets of finite sequences of symbols*; those symbols are referred to as ***terminals*** of the grammar.
-
-
-

We will use the term ***grammar*** to mean "context-free grammar"; we will not use other types of grammar.

- A grammar is relatively concise way to precisely specify certain (possibly infinite) *sets of finite sequences of symbols*; those symbols are referred to as ***terminals*** of the grammar.
- Each of the specified sets of finite sequences of terminals is denoted by a ***nonterminal*** of the grammar.
-
-

We will use the term ***grammar*** to mean "context-free grammar"; we will not use other types of grammar.

- A grammar is relatively concise way to precisely specify certain (possibly infinite) *sets of finite sequences of symbols*; those symbols are referred to as ***terminals*** of the grammar.
- Each of the specified sets of finite sequences of terminals is denoted by a ***nonterminal*** of the grammar.
- One of the nonterminals is regarded as the “most important”: It is called the ***starting nonterminal*** (or *start symbol* or *sentence symbol*); the set of sequences of terminals it denotes is called the ***language*** *generated by* (or *language of*) the grammar.
-

We will use the term ***grammar*** to mean "context-free grammar"; we will not use other types of grammar.

- A grammar is relatively concise way to precisely specify certain (possibly infinite) *sets of finite sequences of symbols*; those symbols are referred to as ***terminals*** of the grammar.
- Each of the specified sets of finite sequences of terminals is denoted by a ***nonterminal*** of the grammar.
- One of the nonterminals is regarded as the “most important”: It is called the ***starting nonterminal*** (or *start symbol* or *sentence symbol*); the set of sequences of terminals it denotes is called the ***language*** *generated by* (or *language of*) the grammar.
- We commonly think of the other nonterminals as being defined in order that they may be used in defining the starting nonterminal.

$$\langle \textit{real-number} \rangle ::= \langle \textit{integer-part} \rangle \; . \; \langle \textit{fraction} \rangle$$

$$\langle \textit{integer-part} \rangle ::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle$$

$$\langle \textit{fraction} \rangle ::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle$$

$$\langle \textit{digit} \rangle ::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

In the above grammar:

$$
\begin{array}{rcl}
\langle \textit{real-number} \rangle & ::= & \langle \textit{integer-part} \rangle \; . \; \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle & ::= & \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle & ::= & \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

In the above grammar:

The following characters are the 11 ***terminals***:

. 0 1 2 3 4 5 6 7 8 9

A ***terminal*** of a grammar is a constant symbol that is ***not*** defined by the grammar.

$$
\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \;.\; \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

In the above grammar:

The following characters are the 11 ***terminals***:

. 0 1 2 3 4 5 6 7 8 9

A ***terminal*** of a grammar is a constant symbol that is ***not*** defined by the grammar.

The following are the 4 ***nonterminals***:

<real-number> *<integer-part>* *<fraction>* *<digit>*

A ***nonterminal*** of a grammar is a variable that denotes a set of finite sequences of terminals. For example,

⟨*real-number*⟩	::=	⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩	::=	⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩	::=	⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

In the above grammar:

The following characters are the 11 ***terminals***:

. 0 1 2 3 4 5 6 7 8 9

A ***terminal*** of a grammar is a constant symbol that is ***not*** defined by the grammar.

The following are the 4 ***nonterminals***:

<real-number> *<integer-part>* *<fraction>* *<digit>*

A ***nonterminal*** of a grammar is a variable that denotes a set of finite sequences of terminals. For example, *<digit>* denotes the set {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}.

$$
\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \;.\; \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

In the above grammar:

⟨*real-number*⟩ ::= ⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩ ::= ⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

In the above grammar:

There are 15 rules called ***productions***. Each production:

- has a left side that is a *single nonterminal*, and
- has a right side that is a *sequence of 0 or more terminals and/or nonterminals*.

The “vertical bar” symbol | means:

The left side of this production is the same as the left side of the ***previous*** *production.*

Example: The **3rd** production of the above grammar is

$$
\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \; . \; \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \, \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \, \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

In the above grammar:

There are 15 rules called ***productions***. Each production:

- has a left side that is a *single nonterminal*, and
- has a right side that is a *sequence of 0 or more terminals and/or nonterminals*.

The “vertical bar” symbol | means:

The left side of this production is the same as the left side of the ***previous*** *production.*

Example: The **3**rd production of the above grammar is

<integer-part> ::= <integer-part> <digit>

$$\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \,.\, \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

⟨*real-number*⟩ ::= ⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩ ::= ⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

Grammar notation is "free format": We can insert whitespace characters, ***including newlines***, between symbols without changing the specified grammar!

For example, the 2nd and 3rd productions

<integer-part> ::= *<digit>* | *<integer-part>* *<digit>*

of the above grammar could be ***rewritten*** as:

⟨*real-number*⟩ ::= ⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩ ::= ⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

Grammar notation is “free format”: We can insert whitespace characters, ***including newlines***, between symbols without changing the specified grammar!

For example, the 2nd and 3rd productions

<integer-part> ::= <digit> | <integer-part> <digit>

of the above grammar could be ***rewritten*** as:

<integer-part> ::= <digit>
| <integer-part> <digit>

$$
\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \;.\; \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

Grammar notation is “free format”: We can insert whitespace characters, ***including newlines***, between symbols without changing the specified grammar!

For example, the 2nd and 3rd productions

    <integer-part> ::= *<digit>* | *<integer-part>* *<digit>*

of the above grammar could be ***rewritten*** as:

    <integer-part> ::= *<digit>*
                      | *<integer-part>* *<digit>*

Intuitively, a production *N* ::= ... means “any ... is an *N*”.

$$
\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \;.\; \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

$$
\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \,.\, \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

<real-number> is the ***starting nonterminal*** of the above grammar.

In this course, we use the convention that ***unless otherwise indicated***, the starting nonterminal of a grammar is the nonterminal on the left side of the ***first*** production:

⟨*real-number*⟩ ::= ⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩ ::= ⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

<real-number> is the ***starting nonterminal*** of the above grammar.

In this course, we use the convention that ***unless otherwise indicated***, the starting nonterminal of a grammar is the nonterminal on the left side of the ***first*** production:

If you write a grammar and want *some other* nonterminal to be its starting nonterminal, you must ***explicitly indicate*** which nonterminal is the starting nonterminal!

$$
\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \,.\, \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

⟨*real-number*⟩ ::= ⟨*integer-part*⟩ . ⟨*fraction*⟩
⟨*integer-part*⟩ ::= ⟨*digit*⟩ | ⟨*integer-part*⟩ ⟨*digit*⟩
⟨*fraction*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*fraction*⟩
⟨*digit*⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Figure 2.3 BNF rules for real numbers.

A grammar given by Sethi to specify unsigned floating point literals in a simple language.

<*empty*> denotes the empty string; other people write ϵ or λ to denote the empty string.

Example: Changing the 2nd production above from *<integer-part> ::= <digit>* to *<integer-part> ::= <empty>* will allow a number with ***no digits before the point*** (e.g., .213) to belong to the language of the grammar.

Note that <*empty*> is ***neither*** a terminal ***nor*** a nonterminal!

Parse Trees

Q. Exactly which sequences of terminals belong to the set of sequences of terminals that is denoted by a given nonterminal ***N*** of a grammar?

Parse Trees

Q. Exactly which sequences of terminals belong to the set of sequences of terminals that is denoted by a given nonterminal ***N*** of a grammar?

A. A sequence of terminals t_1 ... t_k belongs to the set of sequences denoted by a nonterminal ***N*** *if and only if* there is a **parse tree with root *N* that generates** t_1 ... t_k.

Parse Trees

Q. Exactly which sequences of terminals belong to the set of sequences of terminals that is denoted by a given nonterminal ***N*** of a grammar?

A. A sequence of terminals $t_1 \dots t_k$ belongs to the set of sequences denoted by a nonterminal ***N*** *if and only if* there is a **parse tree with root *N* that generates** $t_1 \dots t_k$.

Unless otherwise indicated, the term ***parse tree*** means *parse tree whose root is the starting nonterminal*.

So we can say that sequence of terminals $t_1 \dots t_k$ belongs to the language of a grammar *if and only if* there is a **parse tree that generates** $t_1 \dots t_k$.

Comment:

Parse Trees

Q. Exactly which sequences of terminals belong to the set of sequences of terminals that is denoted by a given nonterminal ***N*** of a grammar?

A. A sequence of terminals t_1 ... t_k belongs to the set of sequences denoted by a nonterminal ***N*** *if and only if* there is a **parse tree with root *N* that generates** t_1 ... t_k.

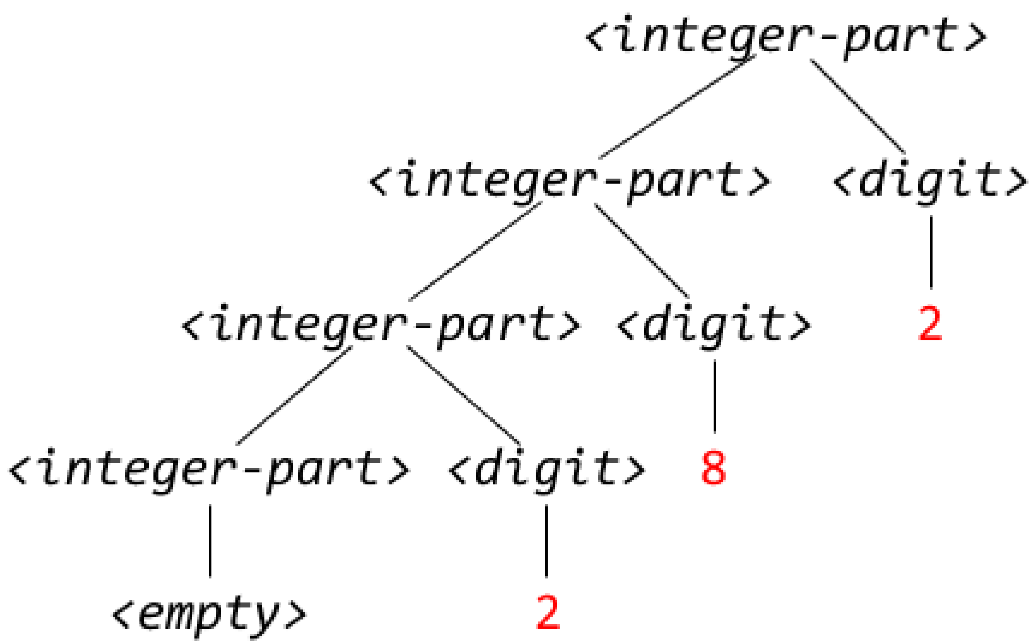
Unless otherwise indicated, the term ***parse tree*** means *parse tree* <u>*whose root is the starting nonterminal*</u>.

So we can say that sequence of terminals t_1 ... t_k belongs to the language of a grammar *if and only if* there is a **parse tree that generates** t_1 ... t_k.

Comment: Instead of using parse trees, we can also answer the above question using the concept of a <u>***derivation***</u> that is introduced on pp. 40 – 41 of Sethi.

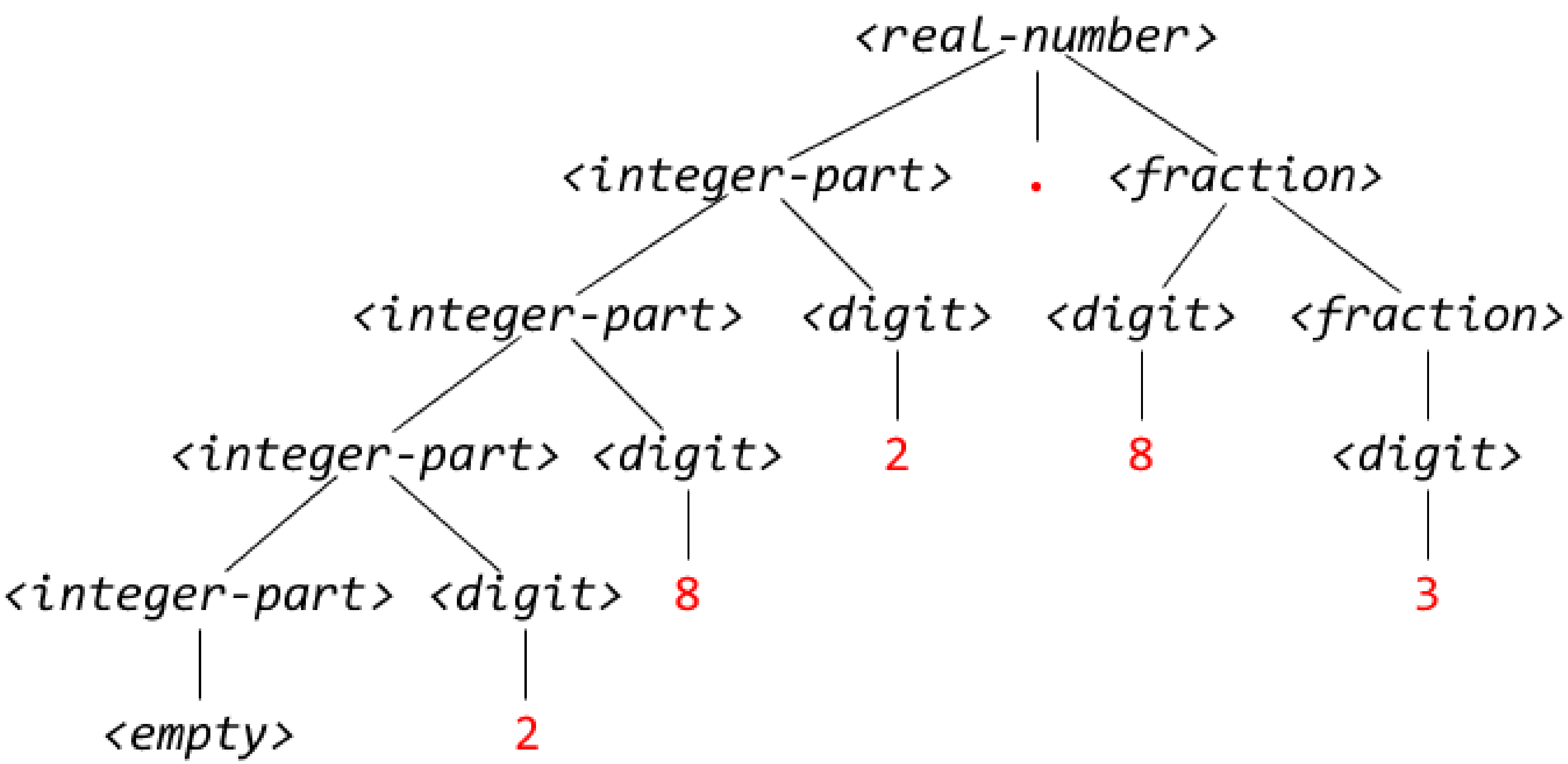
Below is a parse tree, whose root is *<integer-part>*, that shows 282 belongs to the set of sequences denoted by *<integer-part>* in the following grammar:

<real-number> ::= *<integer-part>* . *<fraction>*
<integer-part> ::= *<empty>* | *<integer-part>* *<digit>*
<fraction> ::= *<digit>* | *<digit>* *<fraction>*
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Below is a parse tree that shows 282.83 belongs to the language of the same grammar:

<real-number> ::= *<integer-part>* . *<fraction>*
<integer-part> ::= *<empty>* | *<integer-part>* *<digit>*
<fraction> ::= *<digit>* | *<digit>* *<fraction>*
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Given a nonterminal ***N*** and a ***parse tree with root N*** is an ordered rooted tree with the following properties:

Given a nonterminal ***N*** and a ***parse tree with root N*** is an ordered rooted tree with the following properties:

1. The root is the nonterminal ***N***.

Given a nonterminal ***N*** and a ***parse tree with root N*** is an ordered rooted tree with the following properties:

1. The root is the nonterminal ***N***.
2. Each leaf either is a terminal or is <*empty*>; moreover, a leaf that is <*empty*> has no sibling.

Given a nonterminal ***N*** and a ***parse tree with root N*** is an ordered rooted tree with the following properties:

1. The root is the nonterminal ***N***.
2. Each leaf either is a terminal or is <*empty*>; moreover, a leaf that is <*empty*> has no sibling.
3. Each internal node is a nonterminal.

Given a nonterminal ***N*** and a ***parse tree with root N*** is an ordered rooted tree with the following properties:

1. The root is the nonterminal ***N***.
2. Each leaf either is a terminal or is <*empty*>; moreover, a leaf that is <*empty*> has no sibling.
3. Each internal node is a nonterminal.
4. The left-to-right sequence of children of any internal node ***M*** is the right side of a production whose left side is the nonterminal ***M***.

Given a nonterminal ***N*** and a ***parse tree with root N*** is an ordered rooted tree with the following properties:

1. The root is the nonterminal ***N***.
2. Each leaf either is a terminal or is <*empty*>; moreover, a leaf that is <*empty*> has no sibling.
3. Each internal node is a nonterminal.
4. The left-to-right sequence of children of any internal node ***M*** is the right side of a production whose left side is the nonterminal ***M***.

Unless otherwise indicated, the term ***parse tree*** means *parse tree whose root is the starting nonterminal*.

Given a nonterminal ***N*** and a ***parse tree with root N*** is an ordered rooted tree with the following properties:

1. The root is the nonterminal ***N***.
2. Each leaf either is a terminal or is <*empty*>; moreover, a leaf that is <*empty*> has no sibling.
3. Each internal node is a nonterminal.
4. The left-to-right sequence of children of any internal node ***M*** is the right side of a production whose left side is the nonterminal ***M***.

Unless otherwise indicated, the term ***parse tree*** means *parse tree whose root is the starting nonterminal*.

Given terminals t_1, ..., t_k, a ***parse tree with root N that generates*** t_1 ... t_k (or ***parse tree with root N of*** t_1 ... t_k) is

Given a nonterminal ***N*** and a ***parse tree with root N*** is an ordered rooted tree with the following properties:

1. The root is the nonterminal ***N***.
2. Each leaf either is a terminal or is <*empty*>; moreover, a leaf that is <*empty*> has no sibling.
3. Each internal node is a nonterminal.
4. The left-to-right sequence of children of any internal node ***M*** is the right side of a production whose left side is the nonterminal ***M***.

Unless otherwise indicated, the term ***parse tree*** means *parse tree whose root is the starting nonterminal*.

Given terminals t_1, ..., t_k, a ***parse tree with root N that generates*** t_1 ... t_k (or ***parse tree with root N of*** t_1 ... t_k) is a parse tree with root ***N*** for which the left-to-right sequence of leaves that are not <*empty*> is t_1, ..., t_k.

RECALL:

Q. Exactly which sequences of terminals belong to the set of sequences of terminals that is denoted by a given nonterminal ***N*** of a grammar?

A. A sequence of terminals t_1 ... t_k belongs to the set of sequences denoted by a nonterminal ***N*** *if and only if* there is a **parse tree with root *N* that generates** t_1 ... t_k.

Unless otherwise indicated, the term ***parse tree*** means *parse tree whose root is the starting nonterminal.*

So we can say that sequence of terminals t_1 ... t_k belongs to the language of a grammar *if and only if* there is a **parse tree that generates** t_1 ... t_k.

Comment: Instead of using parse trees, we can also answer the above question using the concept of a ***derivation*** that is introduced on pp. 40 – 41 of Sethi.

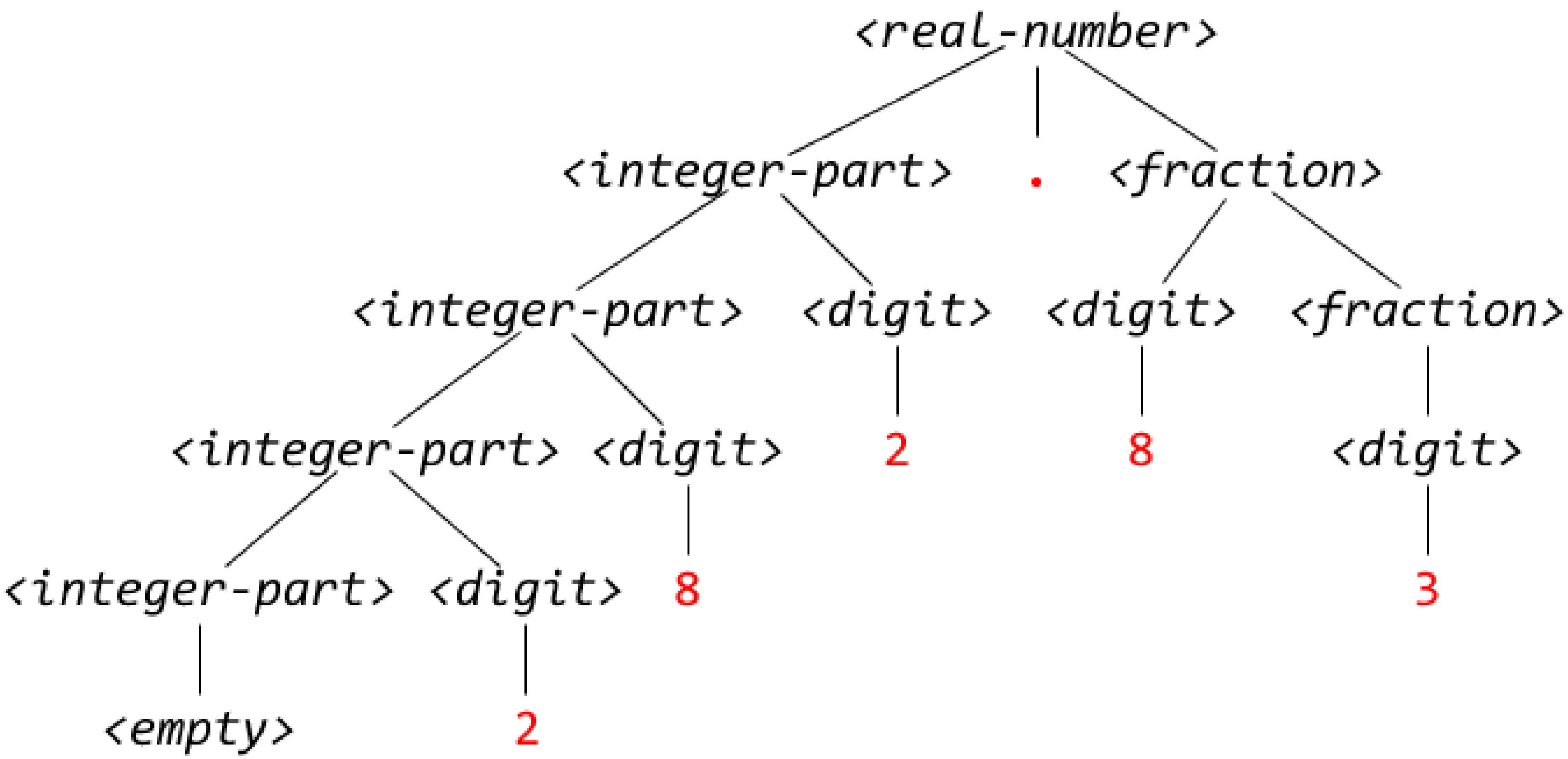
Below is a parse tree that shows 282.83 belongs to the language of the same grammar:

<real-number> ::= *<integer-part>* . *<fraction>*

<integer-part> ::= *<empty>* | *<integer-part>* *<digit>*

<fraction> ::= *<digit>* | *<digit>* *<fraction>*

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Use of Grammars to Define *Syntactically* Valid Code

An important part of the work of a compiler or interpreter is ***lexical analysis*** or ***lexical scanning***.

Lexical analysis decomposes the source program into **token instances** (i.e., **instances of *tokens***).
Ten examples of tokens of a C-like language are:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Use of Grammars to Define *Syntactically* Valid Code

An important part of the work of a compiler or interpreter is ***lexical analysis*** or ***lexical scanning***.

Lexical analysis decomposes the source program into **token instances** (i.e., **instances of *tokens***).
Ten examples of tokens of a C-like language are:
; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*.

For Java:
3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**
2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

Note:

Use of Grammars to Define *Syntactically* Valid Code

An important part of the work of a compiler or interpreter is ***lexical analysis*** or ***lexical scanning***.

Lexical analysis decomposes the source program into **token instances** (i.e., **instances of *tokens***). Ten examples of tokens of a C-like language are:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*.

For Java:

3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**

2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

Note: In sec. 2.3 of Sethi, the tokens **IDENTIFIER** and **UNSIGNED-INT-LITERAL** are called **name** and **number**, and a token instance is called a ***spelling***.

If a piece of source code should be decomposed by a compiler into a sequence of token instances t_1 ... t_n in which each t_i is an instance of token T_i, we say T_1 ... T_n is the ***sequence of tokens*** of that source code.

Java Example: **IDENTIFIER = UNSIGNED-INT-LITERAL ;**
is the sequence of tokens of **x23 = 4;**

If a piece of source code should be decomposed by a compiler into a sequence of token instances t_1 ... t_n in which each t_i is an instance of token T_i, we say T_1 ... T_n is the ***sequence of tokens*** of that source code.

Java Example: **IDENTIFIER = UNSIGNED-INT-LITERAL ;**
is the sequence of tokens of **x23 = 4;**

For many programming languages L, we can construct a grammar G (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by G if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly valid L source file.

- "possibly valid" means

- "roughly speaking" means

If a piece of source code should be decomposed by a compiler into a sequence of token instances $t_1 \dots t_n$ in which each t_i is an instance of token T_i, we say $T_1 \dots T_n$ is the ***sequence of tokens*** of that source code.

Java Example: **IDENTIFIER = UNSIGNED-INT-LITERAL ;**
is the sequence of tokens of **x23 = 4;**

For many programming languages L, we can construct a grammar *G* (whose terminals are L's tokens) such that:

$T_1 \dots T_n$ belongs to the language generated by *G* if (and, roughly speaking, only if) $T_1 \dots T_n$ is the sequence of tokens of a possibly valid L source file.

- "possibly valid" means "can be successfully compiled under certain conditions" (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means

If a piece of source code should be decomposed by a compiler into a sequence of token instances t_1 ... t_n in which each t_i is an instance of token T_i, we say T_1 ... T_n is the ***sequence of tokens*** of that source code.

Java Example: **IDENTIFIER = UNSIGNED-INT-LITERAL ;**
is the sequence of tokens of **x23 = 4;**

For many programming languages L, we can construct a grammar G (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by G if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly valid L source file.

- "possibly valid" means "can be successfully compiled under certain conditions" (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some (but not too many) exceptions to the condition's "only if" part are allowed.

For many programming languages L, we can construct a grammar G (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by G if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly valid L source file.

- "possibly valid" means "can be successfully compiled under certain conditions" (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some (but not too many) exceptions to the condition's "only if" part are allowed.

For many programming languages L, we can construct a grammar *G* (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by *G* if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly valid L source file.

- "possibly valid" means "can be successfully compiled under certain conditions" (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some (but not too many) exceptions to the condition's "only if" part are allowed.

For many programming languages L, we can construct a grammar *G* (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by *G* if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly valid L source file.

- "possibly valid" means "can be successfully compiled under certain conditions" (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some (but not too many) exceptions to the condition's "only if" part are allowed.

We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the grammar *G*.

Note: Replacing one identifier with another and replacing a literal constant with another of the same type (e.g., changing 9/x to 3/y) will ***not*** affect the syntactic validity of a source file, as it won't change its sequence of tokens!

EBNF: Extended BNF

EBNF notation supplements BNF notation with (...), [...], and { ... } to allow simpler specifications.

(γ_1 | ... | γ_k) means

[...] =

{ ... } means

EBNF: Extended BNF

EBNF notation supplements BNF notation with (...), [...], and { ... } to allow simpler specifications.

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[...] =
{ ... } means

EBNF: Extended BNF

EBNF notation supplements BNF notation with (...), [...], and { ... } to allow simpler specifications.

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[...] = (... | <empty>) means "... is optional".
{ ... } means

EBNF: Extended BNF

EBNF notation supplements BNF notation with (...), [...], and { ... } to allow simpler specifications.

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[...] = (... | <empty>) means "... is optional".
{ ... } means "*zero or more* occurrences of ...".

EBNF: Extended BNF

EBNF notation supplements BNF notation with (...), [...], and { ... } to allow simpler specifications.

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[...] = (... | <empty>) means "... is optional".
{ ... } means "*zero or more* occurrences of ...".

Examples

Expr ::= Term (+ | –) Term

is equivalent to the following 2 BNF productions:

EBNF: Extended BNF

EBNF notation supplements BNF notation with (...), [...], and { ... } to allow simpler specifications.

(γ_1 | ... | γ_k) means “pick any one of γ_1, ..., γ_k”.
[...] = (... | <empty>) means “... is optional”.
{ ... } means “*zero or more* occurrences of ...”.

Examples

```
Expr ::= Term (+ | -) Term
```

is equivalent to the following 2 BNF productions:

```
Expr ::=   Term + Term
         | Term - Term
```

EBNF: Extended BNF

EBNF notation supplements BNF notation with (...), [...], and { ... } to allow simpler specifications.

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[...] = (... | <empty>) means "... is optional".
{ ... } means "*zero or more* occurrences of ...".

Examples

```
Expr ::= Term (+ | –) Term
```

is equivalent to the following 2 BNF productions:

```
Expr ::=    Term + Term
          | Term – Term
```

```
Expr ::= [+ | –] Term (+ | –) Term
```

is equivalent to

```
Expr ::= (+ | – | <empty>) Term (+ | –) Term
```

which is equivalent to these 6 BNF productions:

EBNF: Extended BNF

EBNF notation supplements BNF notation with (...), [...], and { ... } to allow simpler specifications.

$(\gamma_1 \mid \dots \mid \gamma_k)$ means “pick any one of $\gamma_1, \dots, \gamma_k$”.
[...] = (... | <empty>) means “... is optional”.
{ ... } means “*zero or more* occurrences of ...”.

Examples

```
Expr ::= Term (+ | –) Term
    is equivalent to the following 2 BNF productions:
Expr ::=    Term + Term
          | Term – Term

Expr ::= [+ | –] Term (+ | –) Term
    is equivalent to
Expr ::= (+ | – | <empty>) Term (+ | –) Term
    which is equivalent to these 6 BNF productions:
Expr ::=    + Term + Term | – Term + Term | Term + Term
          | + Term – Term | – Term – Term | Term – Term
```

(γ_1 | ... | γ_k) means “pick any one of γ_1, ..., γ_k”.
[...] = (... | <empty>) means “... is optional”.
{ ... } means “*zero or more* occurrences of ...”.

Examples

```
   Expr ::= Term (+ | –) Term
      is equivalent to the following 2 BNF productions:
   Expr ::=    Term + Term
             | Term – Term

   Expr ::= [+ | –] Term (+ | –) Term
      is equivalent to
   Expr ::= (+ | – | <empty>) Term (+ | –) Term
      which is equivalent to these 6 BNF productions:
   Expr ::=    + Term + Term | – Term + Term | Term + Term
             | + Term – Term | – Term – Term | Term – Term
```

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[...] = (... | <empty>) means "... is optional".
{ ... } means "*zero or more* occurrences of ...".

Examples

```
Expr ::= Term (+ | –) Term
   is equivalent to the following 2 BNF productions:
Expr ::=    Term + Term
          | Term – Term

Expr ::= [+ | –] Term (+ | –) Term
   is equivalent to
Expr ::= (+ | – | <empty>) Term (+ | –) Term
   which is equivalent to these 6 BNF productions:
Expr ::=    + Term + Term | – Term + Term | Term + Term
          | + Term – Term | – Term – Term | Term – Term
```

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[...] = (... | <empty>) means "... is optional".
{ ... } means "*zero or more* occurrences of ...".

Examples

```
Expr ::= Term (+ | -) Term
```

is equivalent to the following 2 BNF productions:

```
Expr ::=    Term + Term
          | Term - Term
```

```
Expr ::= [+ | -] Term (+ | -) Term
```

is equivalent to

```
Expr ::= (+ | - | <empty>) Term (+ | -) Term
```

which is equivalent to these 6 BNF productions:

```
Expr ::=    + Term + Term | - Term + Term | Term + Term
          | + Term - Term | - Term - Term | Term - Term
```

```
Expr ::= Term {(+ | -) Term}
```

is equivalent to an ***infinite*** collection of BNF productions, including productions such as

```
Expr ::= Term + Term + Term - Term + Term - Term - Term
```

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

-
-
-

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

- Replace each $(x_1 \mid \ldots \mid x_k)$ with a new nonterminal (D, say) that is defined by these k productions:

  $$D ::= x_1 \mid \ldots \mid x_k$$

-

-

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

- Replace each (x_1 | ... | x_k) with a new nonterminal (D, say) that is defined by these k productions:

  D ::= x_1 | ... | x_k
- Replace each [x_1 | ... | x_k] with a new nonterminal (D, say) that is defined by these k+1 productions:

  D ::= *<empty>* | x_1 | ... | x_k
-

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

- Replace each $(x_1 \mid \ldots \mid x_k)$ with a new nonterminal (D, say) that is defined by these k productions:

  D ::= $x_1 \mid \ldots \mid x_k$

- Replace each $[x_1 \mid \ldots \mid x_k]$ with a new nonterminal (D, say) that is defined by these k+1 productions:

  D ::= *<empty>* | $x_1 \mid \ldots \mid x_k$

- Replace each $\{x_1 \mid \ldots \mid x_k\}$ with a new nonterminal (D, say) that is defined by these k+1 productions:

  D ::= *<empty>* | $Dx_1 \mid \ldots \mid Dx_k$

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

- Replace each $(x_1 \mid \ldots \mid x_k)$ with a new nonterminal (D, say) that is defined by these k productions:

  D ::= $x_1 \mid \ldots \mid x_k$

- Replace each $[x_1 \mid \ldots \mid x_k]$ with a new nonterminal (D, say) that is defined by these k+1 productions:

  D ::= *<empty>* | $x_1 \mid \ldots \mid x_k$

- Replace each $\{x_1 \mid \ldots \mid x_k\}$ with a new nonterminal (D, say) that is defined by these k+1 productions:

  D ::= *<empty>* | $Dx_1 \mid \ldots \mid Dx_k$

Here k may be 1. Thus { *Digit* } can be replaced with a new nonterminal (*DigitSeq*, say) that is defined by:

DigitSeq ::= *<empty>* | *DigitSeq Digit*

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}        (*)
```

into a finite set of BNF productions.

Example: We now use the above method to translate

Expr ::= [+ | -] Term {(+ | -) Term}   (*)

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes:

Example: We now use the above method to translate

Expr ::= [+ | −] Term {(+ | −) Term}   (*)

into a finite set of BNF productions.

1. First, replace (+ | −) with a nonterminal Op defined by: Op ::= + | −

   (*) becomes: Expr ::= [+ | −] Term {Op Term}   (**)

Example: We now use the above method to translate

Expr ::= [+ | −] Term {(+ | −) Term}   (*)

into a finite set of BNF productions.

1. First, replace (+ | −) with a nonterminal Op
   defined by: Op ::= + | −
   (*) becomes: Expr ::= [+ | −] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes:

Example: We now use the above method to translate

Expr ::= [+ | −] Term {(+ | −) Term}   (*)

into a finite set of BNF productions.

1. First, replace (+ | −) with a nonterminal Op defined by: Op ::= + | −
   (*) becomes: Expr ::= [+ | −] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes: Expr ::= [+ | −] Term Rest   (***)

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}          (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op defined by: Op ::= + | -

   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)

2. Next, replace {Op Term} with a nonterminal Rest defined by: Rest ::= <*empty*> | Rest Op Term

   (**) becomes: Expr ::= [+ | -] Term Rest   (***)

3. Finally, replace [+ | -] with a nonterminal OptSign defined by OptSign ::= <*empty*> | + | -

   (***) becomes:

Example: We now use the above method to translate

Expr ::= [+ | −] Term {(+ | −) Term}   (*)

into a finite set of BNF productions.

1. First, replace (+ | −) with a nonterminal Op
   defined by: Op ::= + | −
   (*) becomes: Expr ::= [+ | −] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes: Expr ::= [+ | −] Term Rest   (***)
3. Finally, replace [+ | −] with a nonterminal OptSign
   defined by   OptSign ::= <*empty*> | + | −
   (***) becomes: Expr ::= OptSign Term Rest

Example: We now use the above method to translate

Expr ::= [+ | -] Term {(+ | -) Term}   (*)

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes: Expr ::= [+ | -] Term Rest   (***)
3. Finally, replace [+ | -] with a nonterminal OptSign
   defined by OptSign ::= <*empty*> | + | -
   (***) becomes: Expr ::= OptSign Term Rest

The result is the following set of 8 BNF productions:

Expr ::= OptSign Term Rest
OptSign ::= <*empty*> | + | -
Rest ::= <*empty*> | Rest Op Term
Op ::= + | -

While the above method is general, it will often ***not*** find a simplest BNF equivalent of the given EBNF rule.

For example, here is a simpler BNF equivalent of the EBNF rule Expr ::= [+ | −] Term {(+ | −) Term} considered above:

While the above method is general, it will often ***not*** find a simplest BNF equivalent of the given EBNF rule.

For example, here is a simpler BNF equivalent of the EBNF rule Expr ::= [+ | −] Term {(+ | −) Term} considered above:

Expr ::= Term | + Term | − Term | Expr + Term | Expr − Term

While the above method is general, it will often ***not*** find a simplest BNF equivalent of the given EBNF rule.

For example, here is a simpler BNF equivalent of the EBNF rule Expr ::= [+ | −] Term {(+ | −) Term} considered above:

Expr ::= Term | + Term | − Term | Expr + Term | Expr − Term

EBNF can be used just like BNF to define what it means for a source code file to be "***syntactically*** valid":

For many programming languages L, we can construct ~~a grammar~~ an EBNF specification *G* (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by *G*

if (and, roughly speaking, only if) T_1 ... T_n is the

sequence of tokens of a possibly valid L source file.

We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the ~~grammar~~ EBNF specification *G*.

Note: Replacing one identifier with another and replacing a literal constant with another of the same type (e.g., changing 9/x to 3/y) will ***not*** affect the syntactic validity of a source file, as it won't change its sequence of tokens!

A Rule to Follow When Writing EBNF Specifications

In EBNF, when any of the characters | ( ) [ ] { } is a terminal, that terminal should be ***put in single quotes*** to make it clear that the character is ***not*** being used with its EBNF meaning!

Sethi says the following about this on p. 47 of his book (p. 48 of the course reader):

Symbols such as { and }, which have a special status in a language description, are called *metasymbols*.

EBNF has many more metasymbols than BNF. Furthermore, these same symbols can also appear in the syntax of a language—the index i in $A[i]$ is not optional—so care is needed to distinguish tokens from metasymbols. Confusion between tokens and metasymbols will be avoided by enclosing tokens within single quotes if needed, as in '('.

An EBNF version of the grammar in Fig. 2.6 is

$$
\begin{aligned}
\langle \textit{expression} \rangle \quad &::= \quad \langle \textit{term} \rangle \ \{\ (+ \mid -)\ \langle \textit{term} \rangle\ \} \\
\langle \textit{term} \rangle \quad &::= \quad \langle \textit{factor} \rangle \ \{\ (* \mid /)\ \langle \textit{factor} \rangle\ \} \\
\langle \textit{factor} \rangle \quad &::= \quad \text{'('}\ \langle \textit{expression} \rangle\ \text{')'} \mid \textbf{name} \mid \textbf{number}
\end{aligned}
$$

Frequently Useful EBNF Equivalences

An ***EBNF form*** is "an expression that can be the right side of a valid EBNF rule".

Frequently Useful EBNF Equivalences

An ***EBNF form*** is “an expression that can be the right side of a valid EBNF rule”. More precisely, γ is an ***EBNF form*** just if one of the following is true:

Frequently Useful EBNF Equivalences

An ***EBNF form*** is "an expression that can be the right side of a valid EBNF rule". More precisely, γ is an ***EBNF form*** just if one of the following is true:

1. γ is a terminal, or a nonterminal, or <empty>.
2. γ is (e) for some EBNF form e.
3. γ is e_1 ... e_n for some EBNF forms e_1, ..., e_n.
4. γ is e_1 | ... | e_n for some EBNF forms e_1, ..., e_n.
5. γ is [e] for some EBNF form e.
6. γ is {e} for some EBNF form e.

Frequently Useful EBNF Equivalences

An ***EBNF form*** is "an expression that can be the right side of a valid EBNF rule". More precisely, γ is an ***EBNF form*** just if one of the following is true:

1. γ is a terminal, or a nonterminal, or <empty>.
2. γ is (e) for some EBNF form e.
3. γ is e_1 ... e_n for some EBNF forms e_1, ..., e_n.
4. γ is e_1 | ... | e_n for some EBNF forms e_1, ..., e_n.
5. γ is [e] for some EBNF form e.
6. γ is { e } for some EBNF form e.

Now let α and β be any EBNF forms. Then:

-

-

Frequently Useful EBNF Equivalences

An ***EBNF form*** is "an expression that can be the right side of a valid EBNF rule". More precisely, γ is an ***EBNF form*** just if one of the following is true:

1. γ is a terminal, or a nonterminal, or <empty>.
2. γ is (e) for some EBNF form e.
3. γ is e_1 ... e_n for some EBNF forms e_1, ..., e_n.
4. γ is e_1 | ... | e_n for some EBNF forms e_1, ..., e_n.
5. γ is [e] for some EBNF form e.
6. γ is {e} for some EBNF form e.

Now let α and β be any EBNF forms. Then:

- A ::= A (α) is equivalent to
  | β
-

Frequently Useful EBNF Equivalences

An ***EBNF form*** is “an expression that can be the right side of a valid EBNF rule”. More precisely, γ is an ***EBNF form*** just if one of the following is true:

1. γ is a terminal, or a nonterminal, or <empty>.
2. γ is (e) for some EBNF form e.
3. γ is e_1 ... e_n for some EBNF forms e_1, ..., e_n.
4. γ is e_1 | ... | e_n for some EBNF forms e_1, ..., e_n.
5. γ is [e] for some EBNF form e.
6. γ is { e } for some EBNF form e.

Now let α and β be any EBNF forms. Then:

- A ::= A (α) is equivalent to A ::= (β) { α }
  | β
-

Frequently Useful EBNF Equivalences

An ***EBNF form*** is "an expression that can be the right side of a valid EBNF rule". More precisely, γ is an ***EBNF form*** just if one of the following is true:

1. γ is a terminal, or a nonterminal, or <empty>.
2. γ is (e) for some EBNF form e.
3. γ is e_1 ... e_n for some EBNF forms e_1, ..., e_n.
4. γ is e_1 | ... | e_n for some EBNF forms e_1, ..., e_n.
5. γ is [e] for some EBNF form e.
6. γ is {e} for some EBNF form e.

Now let α and β be any EBNF forms. Then:

- A ::= A (α) is equivalent to A ::= (β) {α}
  | β
- A ::= (α) A is equivalent to
  | β

Frequently Useful EBNF Equivalences

An ***EBNF form*** is “an expression that can be the right side of a valid EBNF rule”. More precisely, γ is an ***EBNF form*** just if one of the following is true:

1. γ is a terminal, or a nonterminal, or <empty>.
2. γ is (e) for some EBNF form e.
3. γ is e_1 ... e_n for some EBNF forms e_1, ..., e_n.
4. γ is e_1 | ... | e_n for some EBNF forms e_1, ..., e_n.
5. γ is [e] for some EBNF form e.
6. γ is {e} for some EBNF form e.

Now let α and β be any EBNF forms. Then:

- A ::= A (α) | β is equivalent to A ::= (β) {α}
- A ::= (α) A | β is equivalent to A ::= {α} (β)

Example: Translate `Expr ::= [+ | -] Term {(+ | -) Term}` into BNF *without introducing new nonterminals*.

Solution:

Example: Translate Expr ::= [+ | -] Term {(+ | -) Term}
into BNF *without introducing new nonterminals*.

Solution:

Expr ::= [+ | -] Term {(+ | -) Term}
is equivalent to
A ::= (β) {α}
if A is , β is , and α is .

Example: Translate `Expr ::= [+ | -] Term {(+ | -) Term}` into BNF *without introducing new nonterminals*.

Solution:

`Expr ::= [+ | -] Term {(+ | -) Term}`
is equivalent to
A ::= (β) {α}
if A is Expr, β is , and α is .

Example: Translate `Expr ::= [+ | -] Term {(+ | -) Term}`
into BNF *without introducing new nonterminals*.

Solution:

`Expr ::= [+ | -] Term {(+ | -) Term}`
is equivalent to
A ::= (β) {α}
if A is Expr, β is [+ | -] Term, and α is .

Example: Translate `Expr ::= [+ | -] Term {(+ | -) Term}` into BNF *without introducing new nonterminals*.

Solution:

`Expr ::= [+ | -] Term {(+ | -) Term}`
is equivalent to
`A ::=` $(\beta)\,\{\alpha\}$
if `A` is `Expr`, β is `[+ | -] Term`, and α is `(+ | -) Term`.

Example: Translate `Expr ::= [+ | -] Term {(+ | -) Term}` into BNF *without introducing new nonterminals*.

Solution:

`Expr ::= [+ | -] Term {(+ | -) Term}`
is equivalent to
A ::= (β) {α}
if A is `Expr`, β is `[+ | -] Term`, and α is `(+ | -) Term`.

So, since
A ::= (β) {α}   is equivalent to   A ::= A (α)
| β

we deduce that

`Expr ::= [+ | -] Term {(+ | -) Term}`
is equivalent to

Example: Translate `Expr ::= [+ | -] Term {(+ | -) Term}`
into BNF *without introducing new nonterminals*.

Solution:

`Expr ::= [+ | -] Term {(+ | -) Term}`
is equivalent to
A ::= (β) {α}
if A is Expr, β is [+ | -] Term, and α is (+ | -) Term.

So, since
A ::= (β) {α} is equivalent to A ::= A (α)
| β

we deduce that

`Expr ::= [+ | -] Term {(+ | -) Term}`
is equivalent to

```
Expr ::= Expr ((+ | -) Term)
      |  [+ | -] Term
```

Example: Translate `Expr ::= [+ | -] Term {(+ | -) Term}` into BNF *without introducing new nonterminals*.

Solution:

```
Expr ::= [+ | -] Term {(+ | -) Term}
   is equivalent to
A ::= (β) {α}
   if A is Expr, β is [+ | -] Term, and α is (+ | -) Term.
```

So, since

```
   A ::= (β) {α}   is equivalent to   A ::= A (α)
                                          |  β
```

we deduce that

```
Expr ::= [+ | -] Term {(+ | -) Term}
   is equivalent to
Expr ::= Expr ((+ | -) Term)
      |  [+ | -] Term
```

which can be expanded into these five BNF productions:

Example: Translate `Expr ::= [+ | -] Term {(+ | -) Term}` into BNF *without introducing new nonterminals*.

Solution:

`Expr ::= [+ | -] Term {(+ | -) Term}`
is equivalent to
A ::= (β) {α}
if A is `Expr`, β is `[+ | -] Term`, and α is `(+ | -) Term`.

So, since
A ::= (β) {α}   is equivalent to   A ::= A (α)
| β

we deduce that

```
Expr ::= [+ | -] Term {(+ | -) Term}
```

is equivalent to

```
Expr ::= Expr ((+ | -) Term)
       |  [+ | -] Term
```

which can be expanded into these five BNF productions:

```
Expr ::= Expr + Term | Expr - Term | + Term | - Term | Term
```

2nd Example: Translate the five BNF productions

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

into a non-recursive EBNF rule.

Solution:

2nd Example: Translate the five BNF productions

Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term

into a non-recursive EBNF rule.

Solution:

Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term

can be rewritten in EBNF as

2nd Example: Translate the five BNF productions

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

into a non-recursive EBNF rule.

Solution:

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

can be rewritten in EBNF as

```
Expr ::= Expr (+ Term | – Term)
      |  (<empty> | + | –) Term
```

which can be simplified to

2nd Example: Translate the five BNF productions

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

into a non-recursive EBNF rule.

Solution:

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

can be rewritten in EBNF as

```
Expr ::= Expr (+ Term | – Term)
      |  (<empty> | + | –) Term
```

which can be simplified to

```
Expr ::= Expr ((+ | -) Term)
      |  [+ | –] Term                                    (*)
```

2nd Example: Translate the five BNF productions

Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term

into a non-recursive EBNF rule.

Solution:

Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term

can be rewritten in EBNF as

Expr ::= Expr (+ Term | – Term)
 | (<empty> | + | –) Term

which can be simplified to

Expr ::= Expr ((+ | -) Term)
 | [+ | –] Term **(*)**

This is of the form

A ::= A (α) if A is , α is ,
 | β and β is .

2nd Example: Translate the five BNF productions

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

into a non-recursive EBNF rule.

Solution:

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

can be rewritten in EBNF as

```
Expr ::= Expr (+ Term | – Term)
      |  (<empty> | + | –) Term
```

which can be simplified to

```
Expr ::= Expr ((+ | -) Term)
      |  [+ | –] Term                                        (*)
```

This is of the form

A ::= A (α) if A is Expr, α is ,
| β and β is .

2nd Example: Translate the five BNF productions

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

into a non-recursive EBNF rule.

Solution:

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

can be rewritten in EBNF as

```
Expr ::= Expr (+ Term | – Term)
      |  (<empty> | + | –) Term
```

which can be simplified to

```
Expr ::= Expr ((+ | -) Term)
      |  [+ | –] Term                                         (*)
```

This is of the form

A ::= A (α) if A is Expr, α is (+ | –) Term,
| β and β is .

2nd Example: Translate the five BNF productions

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

into a non-recursive EBNF rule.

Solution:

```
Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term
```

can be rewritten in EBNF as

```
Expr ::= Expr (+ Term | – Term)
      |  (<empty> | + | –) Term
```

which can be simplified to

```
Expr ::= Expr ((+ | -) Term)
      |  [+ | –] Term                                     (*)
```

This is of the form

A ::= A (α)   if A is Expr, α is (+ | –) Term,
   | β and β is [+ | –] Term.

2nd Example: Translate the five BNF productions

`Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term`

into a non-recursive EBNF rule.

Solution:

`Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term`

can be rewritten in EBNF as

```
Expr ::= Expr ((+ | -) Term)
      |  [+ | –] Term                                        (*)
```

This is of the form

A ::= A (α) if A is Expr, α is (+ | –) Term,

| β and β is [+ | –] Term.

2nd Example: Translate the five BNF productions

Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term

into a non-recursive EBNF rule.

Solution:

Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term

can be rewritten in EBNF as

Expr ::= Expr ((+ | -) Term)
| [+ | –] Term **(*)**

This is of the form

A ::= A (α) if A is Expr, α is (+ | –) Term,
| β and β is [+ | –] Term.

So, since A ::= A (α) is equivalent to A ::= $(\beta)\,\{\alpha\}$
| β

(*) is equivalent to:

2nd Example: Translate the five BNF productions

Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term

into a non-recursive EBNF rule.

Solution:

Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term

can be rewritten in EBNF as

Expr ::= Expr ((+ | -) Term)
    | [+ | –] Term **(*)**

This is of the form

A ::= A (α) if A is Expr, α is (+ | –) Term,
  | β and β is [+ | –] Term.

So, since A ::= A (α) is equivalent to A ::= (β) {α}
  | β

(*) is equivalent to:

Expr ::= ([+ | –] Term) {(+ | –) Term}

This can be simplified to:

2nd Example: Translate the five BNF productions

`Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term`

into a non-recursive EBNF rule.

Solution:

`Expr ::= Term | + Term | – Term | Expr + Term | Expr – Term`

can be rewritten in EBNF as

```
Expr ::= Expr ((+ | -) Term)
      |  [+ | –] Term                                             (*)
```

This is of the form

A ::= A (α) if A is Expr, α is (+ | –) Term,
| β and β is [+ | –] Term.

So, since A ::= A (α) | β is equivalent to A ::= (β) $\{\alpha\}$

(*) is equivalent to:

`Expr ::= ([+ | –] Term) {(+ | –) Term}`

This can be simplified to:

`Expr ::= [+ | –] Term {(+ | –) Term}`