Context-Free Syntax of Programming Languages

Context-Free Grammars

Grammars were invented by Chomsky in the mid-1950s for describing natural languages. A notation equivalent to one of his types of grammar (***context-free***/***Type 2*** grammars) was proposed by Backus at ICIP 1959 to specify the syntax of IAL (= Algol 58), an early version of Algol 60.

Backus's notation was improved by Naur and used in the **Algol 60 Report** (edited by Naur), an influential document that did an excellent job of specifying Algol 60.

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

Notes: The notation was first called "Backus Normal Form": Knuth proposed the name "Backus Naur Form" in 1964 in a letter https://dl.acm.org/doi/10.1145/355588.365140 published in the *Communications of the ACM*. Backus makes interesting remarks on the genesis of BNF in the 3rd video at https://amturing.acm.org/award_winners/backus_0703524.cfm; a transcript of the entire interview is available here.

Context-Free Grammars

Grammars were invented by Chomsky in the mid-1950s for describing natural languages. A notation equivalent to one of his types of grammar (***context-free***/***Type 2*** grammars) was proposed by Backus at ICIP 1959 to specify the syntax of IAL (= Algol 58), an early version of Algol 60.

Backus's notation was improved by Naur and used in the **Algol 60 Report** (edited by Naur), an influential document that did 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. We'll use it to mean "*a commonly used notation for writing context-free grammars that specify the syntax of programming language constructs*". We'll refer to grammars written in such a notation as ***BNF specifications***.

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

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 so Sethi does ***not*** call it BNF.

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

Figure 2.6 A grammar for arithmetic expressions.

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

- A grammar is a 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 auxiliary nonterminals that are defined for use in defining the starting nonterminal.

⟨*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}.

⟨*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.

A grammar consists of finitely many rules called ***productions***.

The above grammar has 15 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 next production ***has the same left side as the previous production; we'll only show its right side here.***

Example The **2nd** & **3rd** productions of the above grammar are:

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

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

⟨*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>

A production *N ::= ...* means *any ... is an N*; for example, *<integer-part> ::= <integer-part> <digit>* means *concatenation of an <integer-part> and a <digit> (in that order) gives an <integer-part>*.

⟨*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.

Proof that 313 ∈ <*integer-part*>:

⟨*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.

Proof that 313 ∈ <*integer-part*>:

3 ∈ <*digit*> by the 9th production. (A)

∴ (B)

(C)

∴ (D)

∴ **QED**

⟨*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.

Proof that 313 ∈ <*integer-part*>:

	3 ∈ <*digit*>	by the 9th production.	(A)
∴	3 ∈ <*integer-part*>	by (A) and the 2nd production.	(B)
			(C)
∴			(D)
∴			**QED**

⟨*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.

Proof that 313 ∈ <*integer-part*>:

	3 ∈ <*digit*>	by the 9th production.	(A)
∴	3 ∈ <*integer-part*>	by (A) and the 2nd production.	(B)
	1 ∈ <*digit*>	by the 7th production.	(C)
∴			(D)
∴			**QED**

⟨*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.

Proof that 313 ∈ <*integer-part*>:

	3 ∈ <*digit*>	by the 9th production.	(A)
∴	3 ∈ <*integer-part*>	by (A) and the 2nd production.	(B)
	1 ∈ <*digit*>	by the 7th production.	(C)
∴	31 ∈ <*integer-part*>	by (B,C) and the 3rd production.	(D)
∴			**QED**

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

Figure 2.3 BNF rules for real numbers.

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

Proof that 313 ∈ <*integer-part*>:

	3 ∈ <*digit*>	by the 9th production.	(A)
∴	3 ∈ <*integer-part*>	by (A) and the 2nd production.	(B)
	1 ∈ <*digit*>	by the 7th production.	(C)
∴	31 ∈ <*integer-part*>	by (B,C) and the 3rd production.	(D)
∴	313 ∈ <*integer-part*>	by (D,A) and the 3rd production.	**QED**

⟨*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.

Proof that 313 ∈ <*integer-part*>:

	3 ∈ <*digit*>	by the 9th production.	(A)
∴	3 ∈ <*integer-part*>	by (A) and the 2nd production.	(B)
	1 ∈ <*digit*>	by the 7th production.	(C)
∴	31 ∈ <*integer-part*>	by (B,C) and the 3rd production.	(D)
∴	313 ∈ <*integer-part*>	by (D,A) and the 3rd production.	**QED**

As we'll soon see, this fact can instead be proved using a ***parse tree*** *with root <integer-part>*.

Yet another way to prove the same fact is to use the concept of a ***derivation***. That concept is introduced on pp. 40 – 41 of Sethi, which the Syntax-Reading-and-Exercises-B document on Brightspace asks you to read as part of reading assignment 1.

```
⟨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, then you must ***explicitly indicate*** which nonterminal is the starting nonterminal!

⟨*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!

Things to Remember

- Any symbol that appears on the ***Left*** side of a production of a grammar is a **nonterminal** of the grammar.
- A symbol (other than <*empty*>, ϵ, or λ) that does ***not*** appear on the left side of any production of a grammar but appears on the right side of one or more productions is a **terminal** of the grammar.

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

A **nonterminal** of a grammar is a variable that is defined by the grammar. Each nonterminal denotes a *set of finite sequences of terminals*; the set of sequences denoted by the **starting nonterminal** is called the ***language*** *generated by* (or ***language*** *of*) the 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's a **parse tree whose root is *N* that generates** t_1 ... t_k.

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

So, putting ***N*** = *the starting nonterminal*, a sequence of terminals t_1 ... t_k belongs to *the set denoted by the starting nonterminal* *if and only if* there's a **parse tree that generates** t_1 ... t_k.

Equivalently, a sequence of terminals t_1 ... t_k belongs to *the language generated by a grammar* *if and only if* there's a **parse tree 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 ... t_k belongs to the set of sequences denoted by a nonterminal ***N*** *if and only if* there's a **parse tree whose root is *N* that generates** t_1 ... t_k.

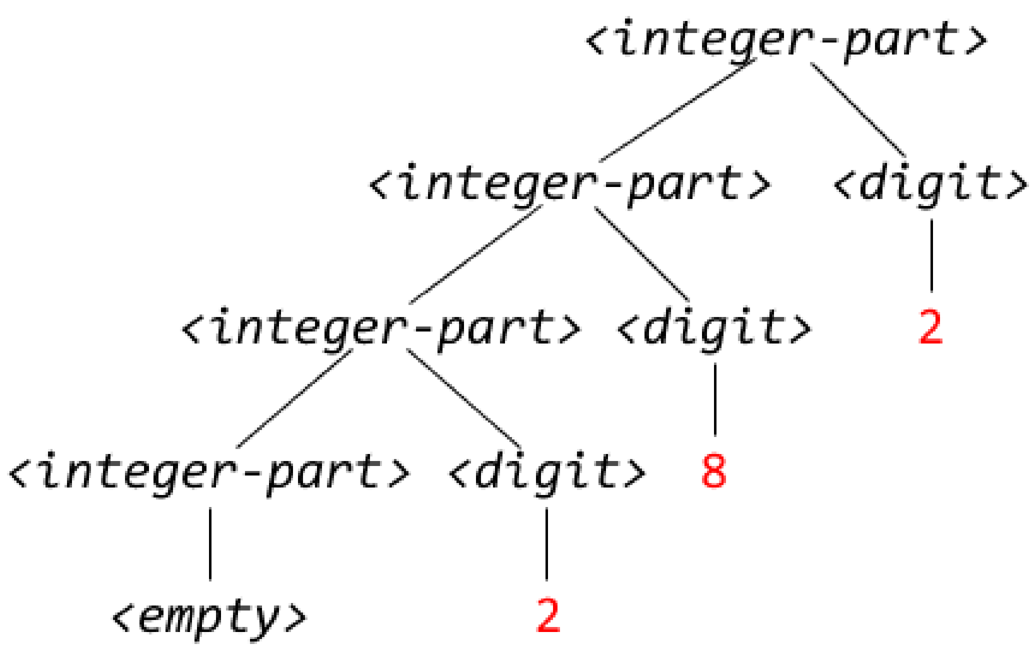
Unless otherwise indicated, the term ***parse tree*** means "*parse tree whose root is the starting nonterminal*":

Therefore, a sequence of terminals t_1 ... t_k belongs to *the language generated by a grammar* *if and only if* there's a **parse tree that generates** t_1 ... t_k.

COMMENT The above question can also be answered using the concept of a ***derivation***, which is introduced on pp. 40-41 of Sethi. (The Syntax-Reading-and-Exercises-B document on Brightspace asks you to read those pages as part of reading assignment 1.)

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



Note: This is just a picture to show what parse trees look like.

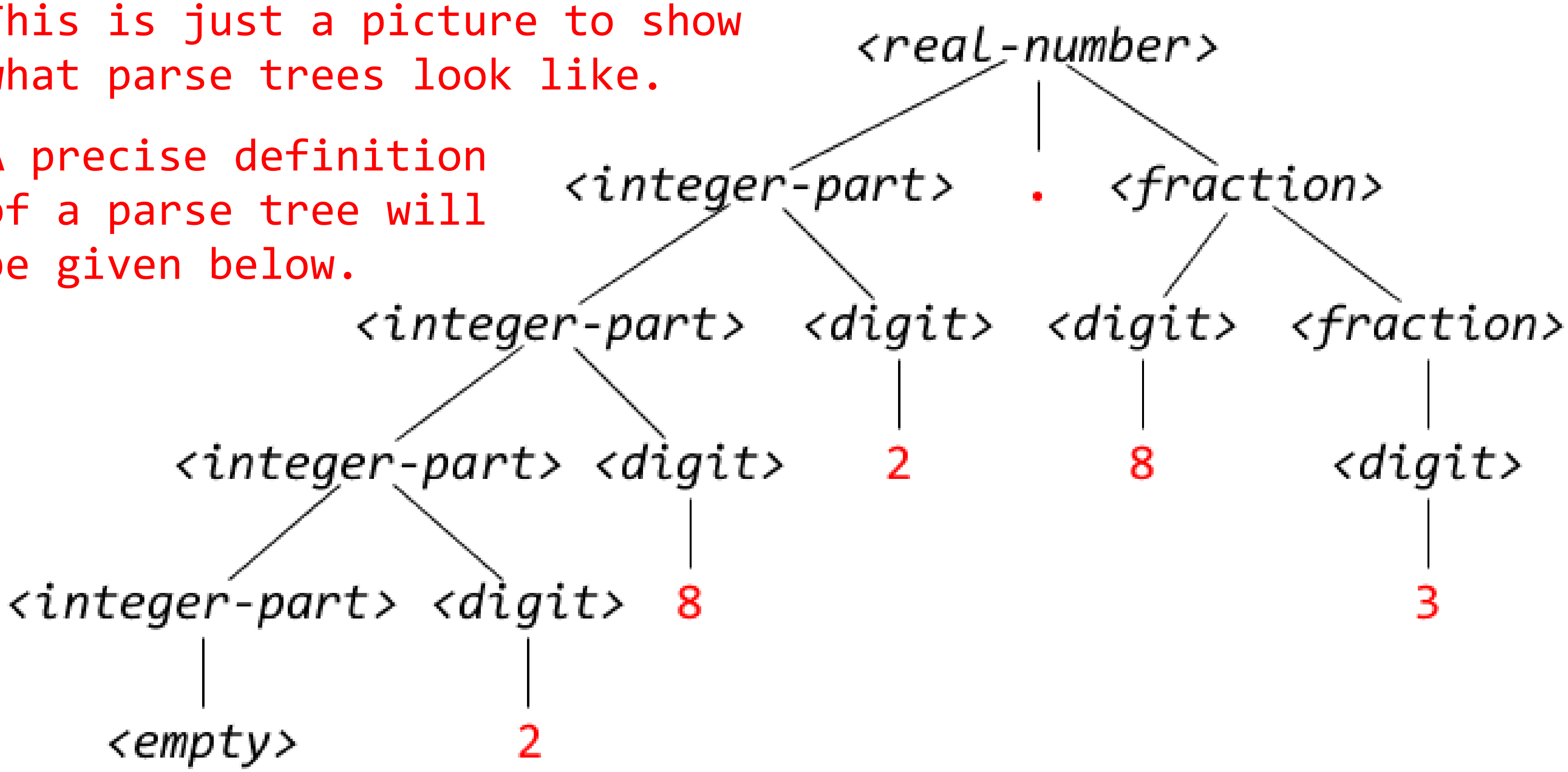
A precise definition of a parse tree will be given below.

The parse tree below shows 282.83 is in the ***set denoted by the starting nonterminal*** (i.e., the ***language***) of this 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

This is just a picture to show what parse trees look like.

A precise definition of a parse tree will be given below.



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

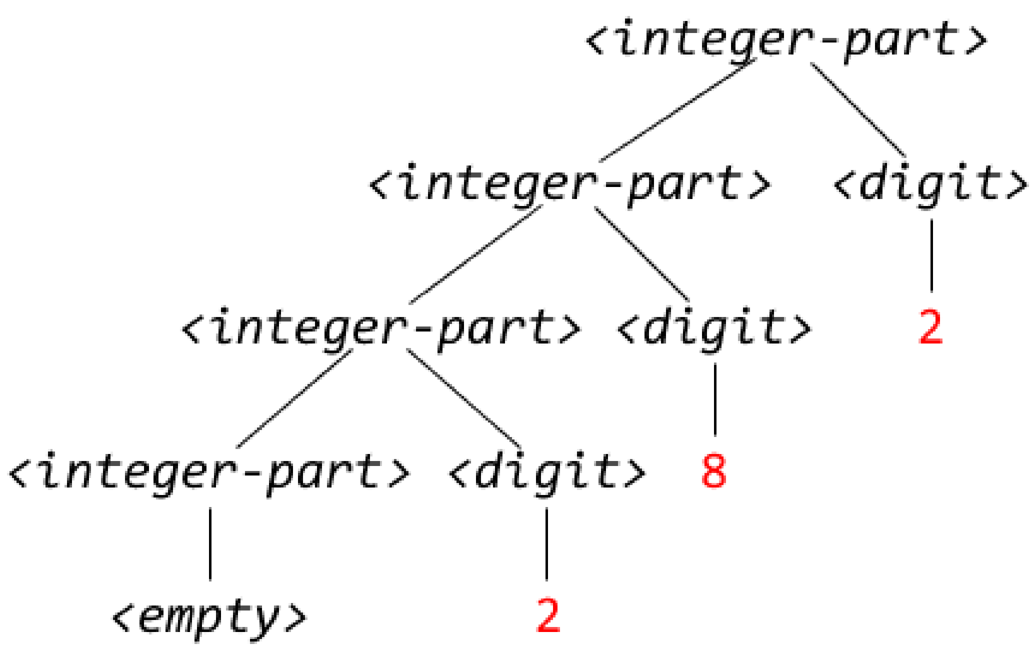
1. The **root** is the nonterminal ***N***.
2. Each **leaf** either is a terminal or is *<empty>*; moreover, a leaf that is *<empty>* has no sibling.
3. Each **internal node** is a nonterminal.
4. The left-to-right sequence of children of any internal node ***M*** is the right side of a production whose 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*** $t_1 \ldots t_k$ (or ***parse tree with root N for*** $t_1 \ldots t_k$ or ***parse tree with root N of*** $t_1 \ldots t_k$) is a parse tree with root ***N*** for which the left-to-right sequence of leaves that are not *<empty>* is $t_1 \ldots t_k$.

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



This is a parse tree with root *<integer-part>* that generates the following sequence of three terminals:

2 8 2

The parse tree below shows 282.83 is in the ***set denoted by the starting nonterminal*** (i.e., the ***language***) of this grammar:

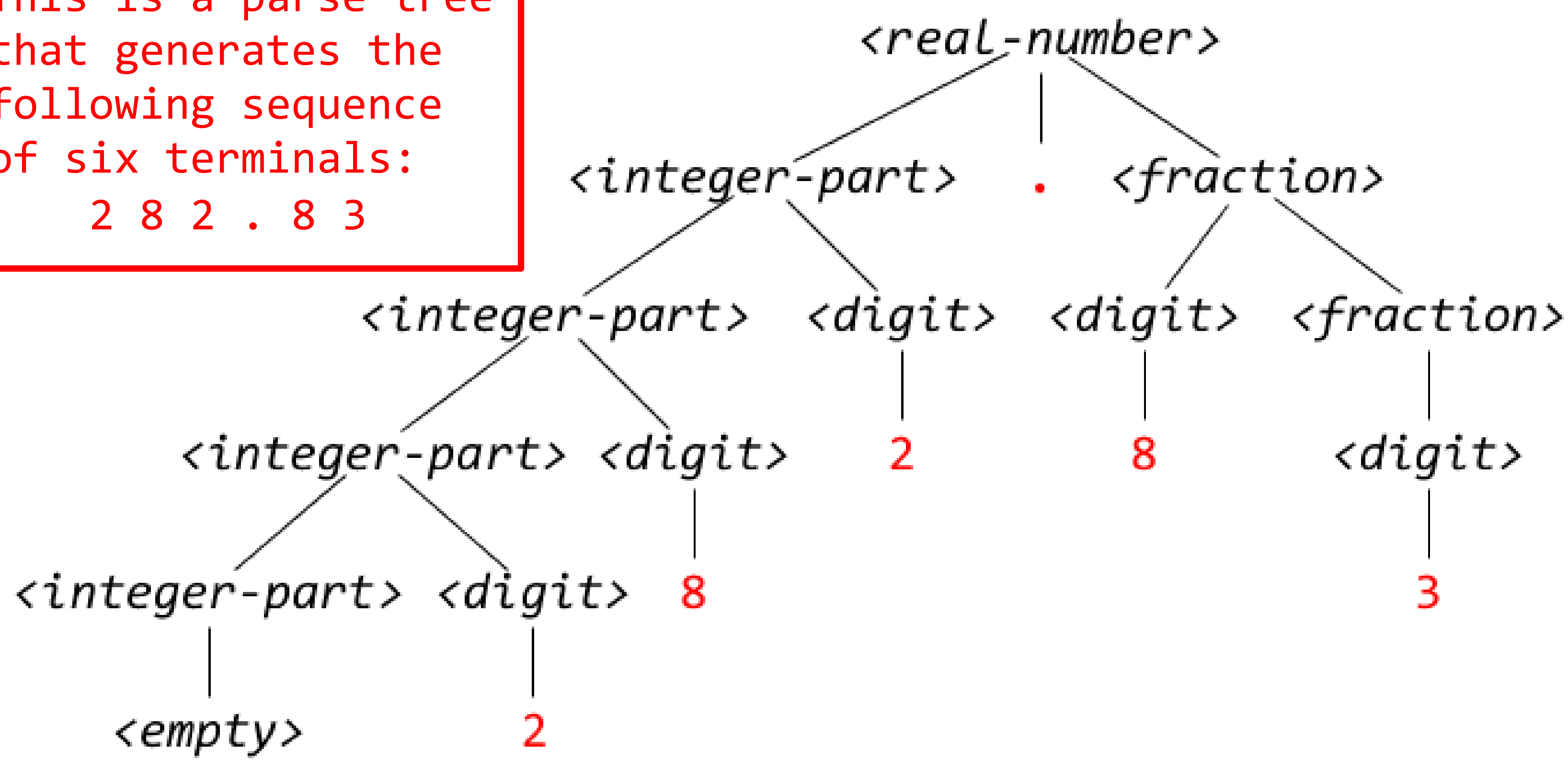
<real-number> ::= *<integer-part>* . *<fraction>*

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

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

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

This is a parse tree that generates the following sequence of six terminals:
2 8 2 . 8 3



RECALL:

Given a nonterminal ***N***, 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 terminals t_1, ..., t_k, a ***parse tree with root N that generates*** 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.
- 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.

Let's draw 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
```

The root of this parse tree
is <integer-part>.

<integer-part>

Let's draw 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>*

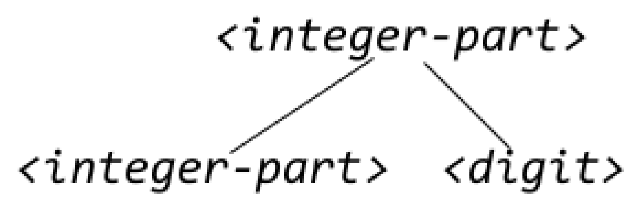
<integer-part>* ::=** *<empty>* | ***<integer-part> <digit>

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

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

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.

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



Let's draw 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>*

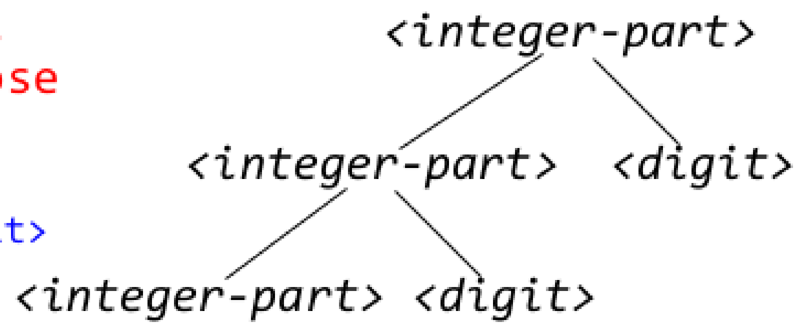
<integer-part>* ::=** *<empty>* | ***<integer-part> <digit>

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

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

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.

Re-using production:
<integer-part> ::= <integer-part> <digit>

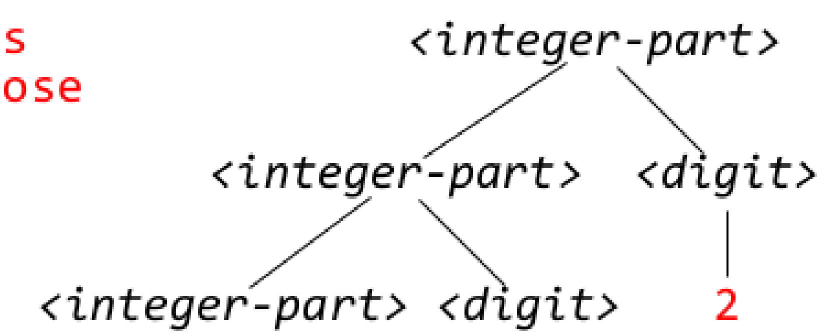


Let's draw a parse tree, whose root is *<integer-part>*, that shows 282 belongs to the set of sequences denoted by *<integer-part>* in the following grammar:

<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

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.

Using production:
<digit> ::= 2

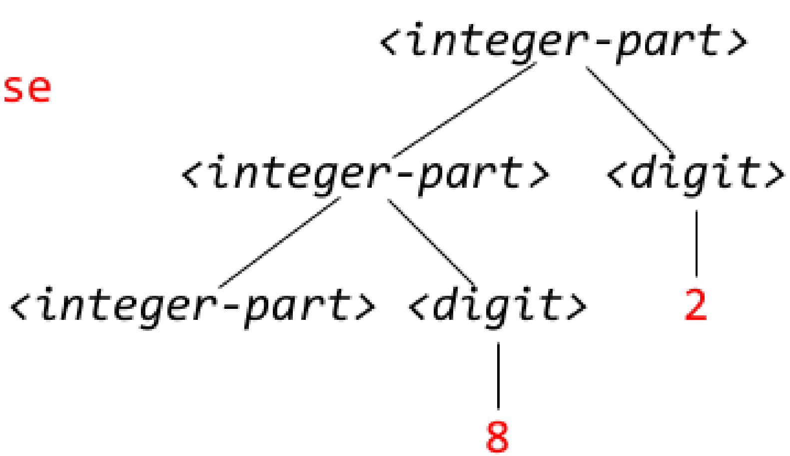


Let's draw a parse tree, whose root is *<integer-part>*, that shows 282 belongs to the set of sequences denoted by *<integer-part>* in the following grammar:

<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

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.

Using production:
<digit> ::= 8



Let's draw 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>*

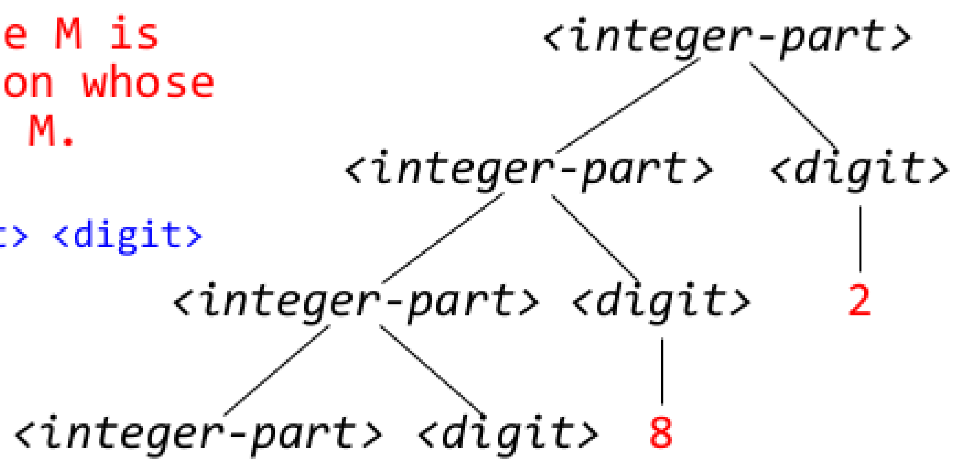
<integer-part>* ::=** *<empty>* | ***<integer-part> <digit>

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

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

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.

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

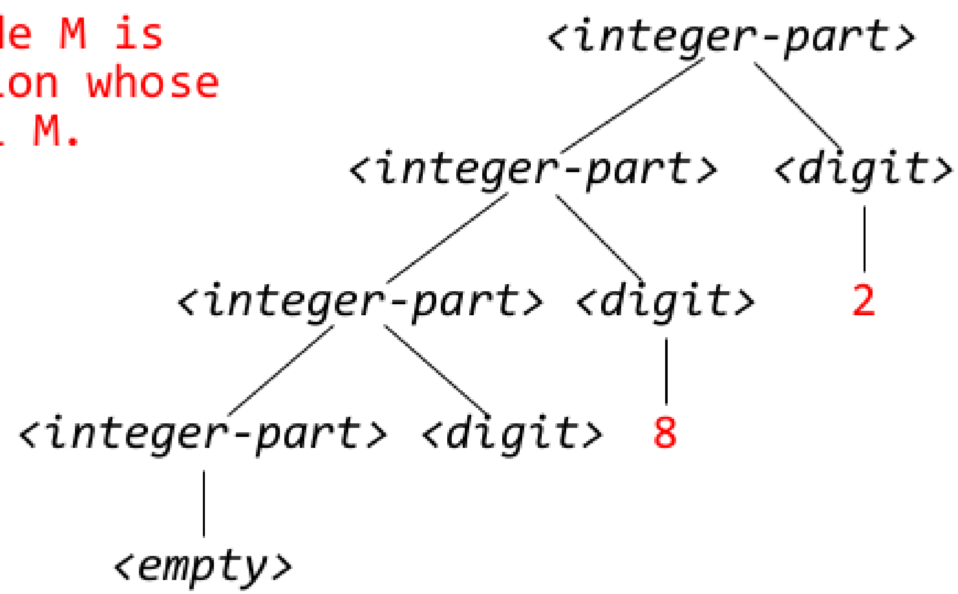


Let's draw a parse tree, whose root is *<integer-part>*, that shows 282 belongs to the set of sequences denoted by *<integer-part>* in the following grammar:

<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

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.

Using production:
<integer-part> ::= <empty>

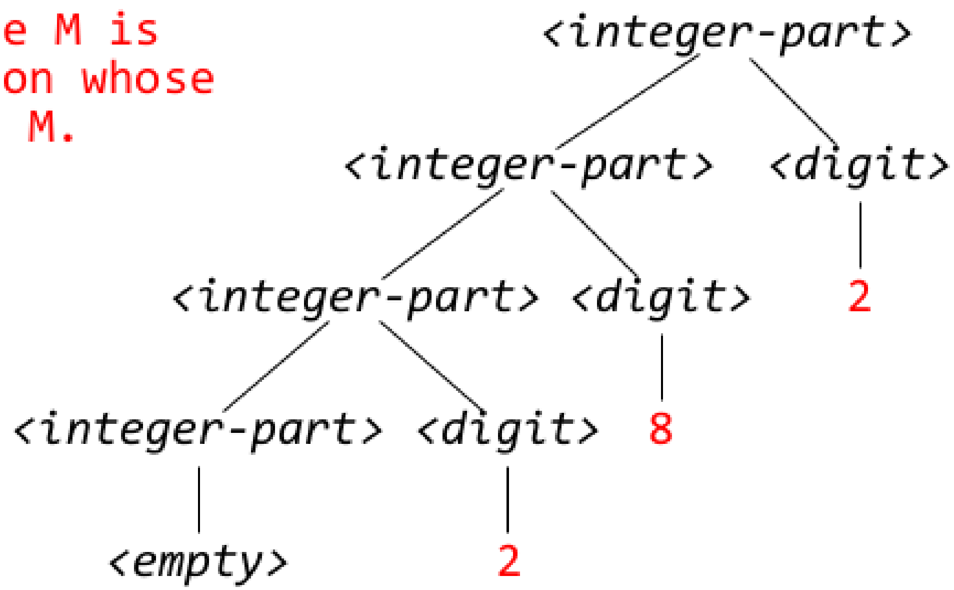


Let's draw a parse tree, whose root is *<integer-part>*, that shows 282 belongs to the set of sequences denoted by *<integer-part>* in the following grammar:

<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

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.

Using production:
<digit> ::= 2



Let's draw 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:

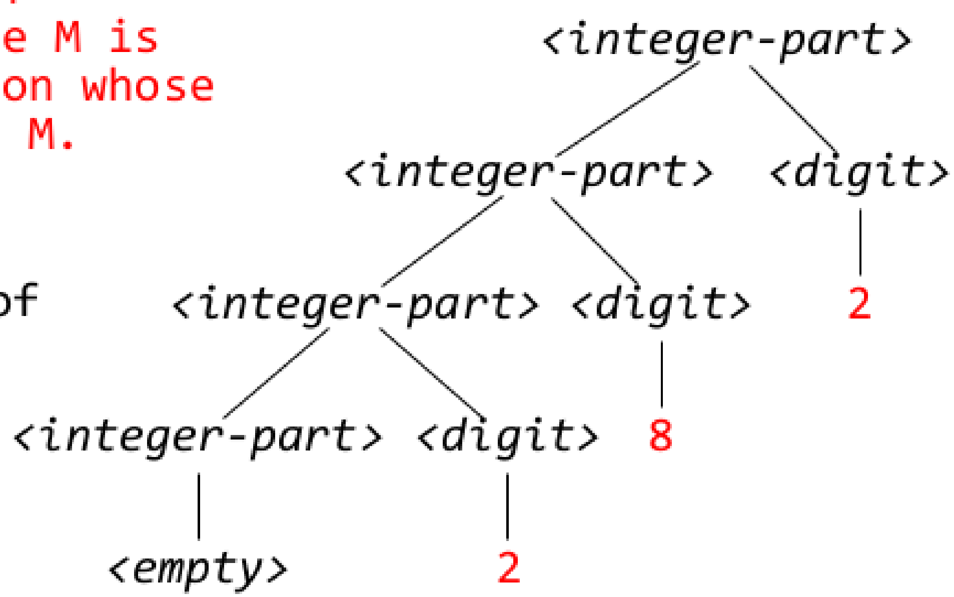
<real-number> ::= *<integer-part>* . *<fraction>*
<integer-part> ::= *<empty>* | *<integer-part>* *<digit>*
<fraction> ::= *<digit>* | *<digit>* *<fraction>*
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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.

Using production:
<digit> ::= 2

The left-to-right sequence of leaves that are not <empty> is 282, as required.

So the parse tree is complete!



RECALL:

The set of sequences of terminals denoted by the **starting nonterminal** of a grammar is called the ***language*** *generated by* (or ***language*** *of*) that grammar.

So a sequence of terminals $t_1 \ldots t_k$ belongs to the language of a grammar *if and only if* there is a **parse tree**, whose root is the **starting nonterminal, that generates** $t_1 \ldots t_k$.

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

So we can simply say:

- A sequence of terminals $t_1 \ldots t_k$ belongs to the language of a grammar *if and only if* there is a **parse tree that generates** $t_1 \ldots t_k$.

Exercise: Draw a parse tree that shows 282.83 belongs to the language of this grammar:

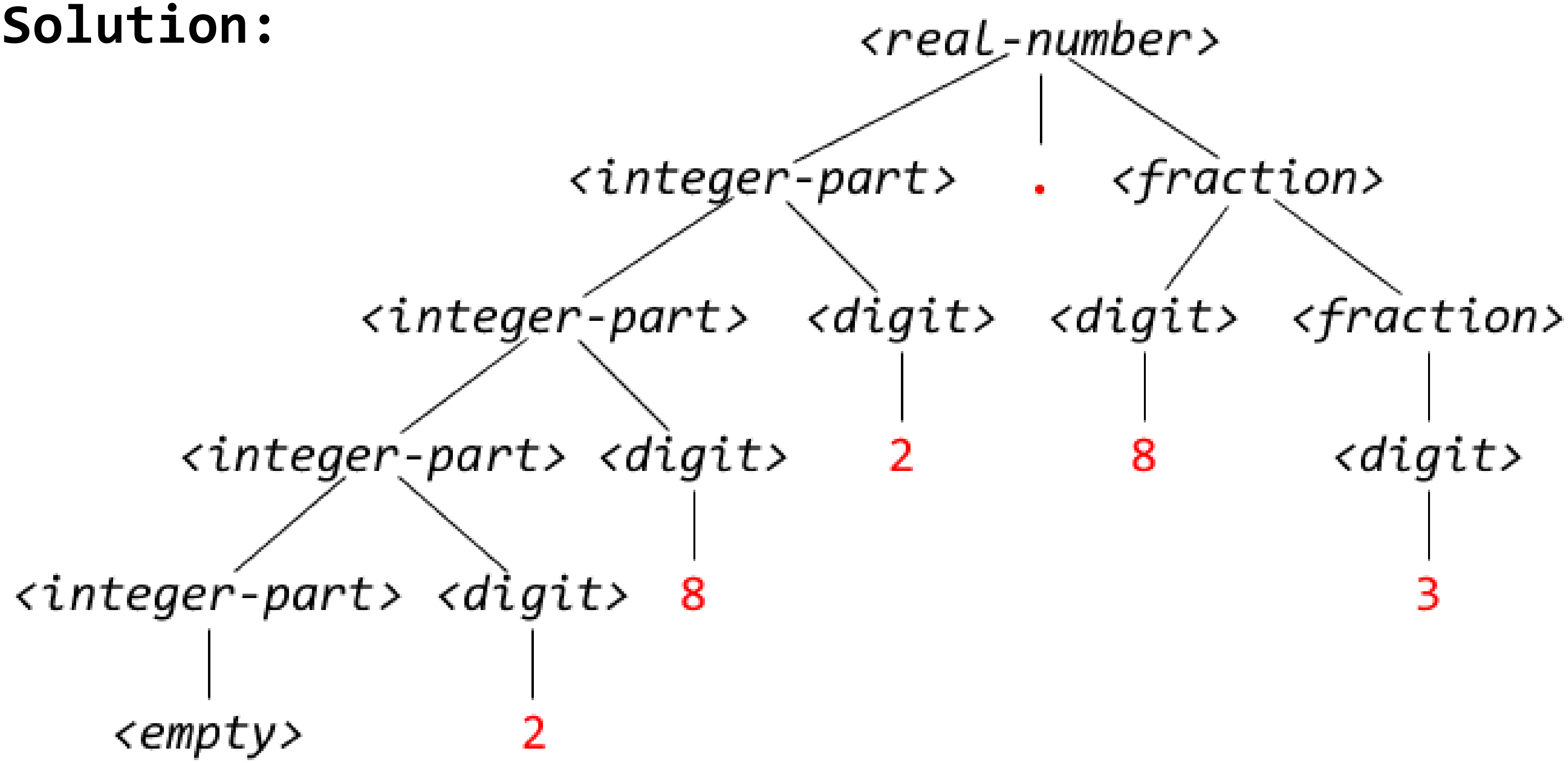
<real-number> ::= *<integer-part>* . *<fraction>*

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

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

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

Solution:



Lexical Syntax: Tokens

An important part of the work of a typical compiler or interpreter is ***lexical analysis*** (also called ***lexical scanning***).

Lexical analysis decomposes the source program into **token instances** (i.e., **instances of *tokens***).
Ten examples of tokens of a language might be:

; < -- -) { 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 or C++:
3 instances of **IDENTIFIER**: **x prevVal pi_2**
3 instances of **UNSIGNED-INT-LITERAL**: **23 0x1A1D 5210101115L**

If a token has just one instance, then it can be denoted by the instance--e.g., **if** denotes the token whose only instance is **if**.

Notes: 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***.
Many authors call a token instance a ***lexeme***.

Lexical Syntax: The Five Kinds of Token

Ten examples of tokens of a language might be:

; < -- -) { 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 or C++**:

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

3 instances of **UNSIGNED-INT-LITERAL**: **23 0x1A1D 5210101115L**

For most programming languages, ***there are 5 kinds of token***:

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.
2. There are tokens called ***literals***, each of which is associated with one kind of value--e.g., *integer*, *floating-point*, *character*, *string*, and *boolean* literals in Java/C++. *Each instance of such a token represents a fixed value (a* ***literal constant****) of the associated kind*. Java/C++ examples:

Lexical Syntax: The Five Kinds of Token

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.
2. There are tokens called ***literals***, each of which is associated with one kind of value--e.g., *integer*, *floating-point*, *character*, *string*, and *boolean* literals in Java/C++. *Each instance of such a token represents a fixed value* (*a* ***literal constant***) *of the associated kind*. Java/C++ examples:
   Instances of the *floating-pt. literal* *token*: 2.3, 4.1f, 3e-4
   Instances of the *string literal* *token*: "The cat", "apple"
3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java/C++ examples: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java/C++ examples: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
   - Reserved words are also called ***keywords***.

Note: In some languages there are "words" that have a different role from that of an identifier, but which are identifiers rather than reserved words as it's legal to use them as identifiers in some contexts: *Such "words" are also called keywords*. In Lisp, special operator names (e.g., IF, LET, QUOTE) are keywords of this kind: They can be used as identifiers, as in

(defun f (if let quote) (+ if let quote)),

though it'd be a bad idea to write such code.

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java/C++ examples: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
4. For each ***operator*** (e.g., **!**, *****, **++**, **+=**, **>=**, **&&**, **:**, **?** in Java/C++) ***there is a token whose only instance is that operator***.
5. Languages usually have certain other characters or sequences of characters that are used as a "punctuation" symbols. Java/C++ examples: **,**, **;**, **.**, **{**, **}**, **[**, **]**, **(**, **)**

   These are called ***delimiters*** or ***separators***. For each of them ***there's a token whose only instance is that symbol***.

A ***lexical syntax specification*** of a programming language specifies ***its tokens*** and ***the sequence of token instances into which any given piece of source code should be decomposed***.