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>*

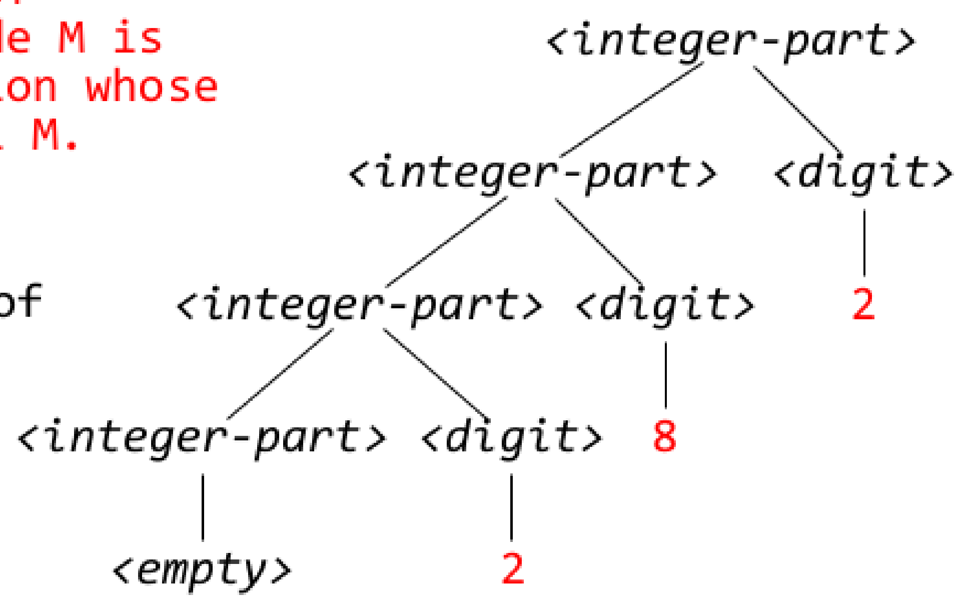
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

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

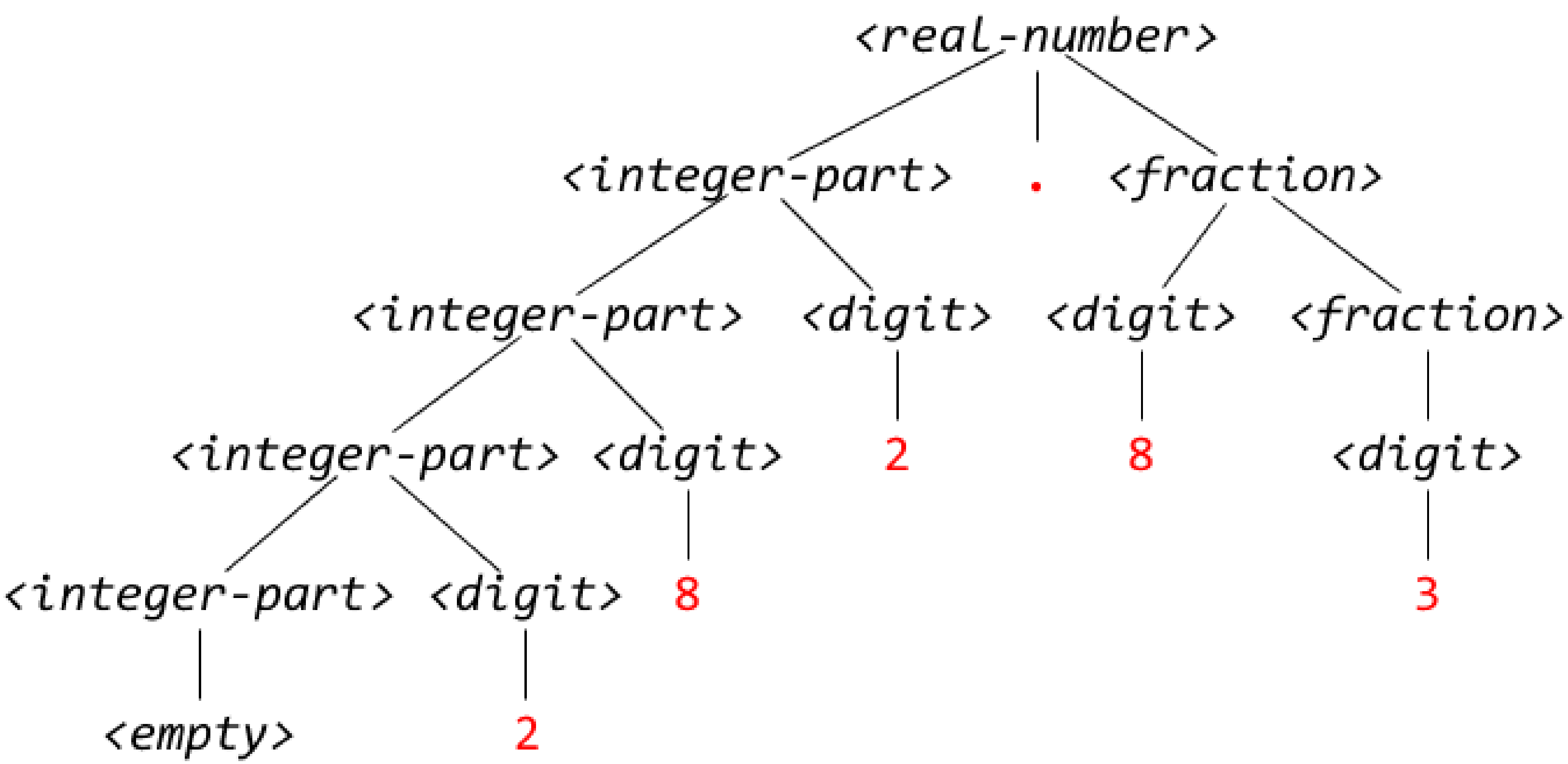
The left-to-right sequence of leaves that are not <empty> is 282, as required.

So the parse tree is complete!



Below is 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



Lexical Syntax: Tokens

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 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:
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**:

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*, *boolean*, and *null* literals in Java. *Each instance of such a token represents a fixed value* (*a* ***literal constant***) *of the associated kind*. 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*, *boolean*, and *null* literals in Java. *Each instance of such a token represents a fixed value* (*a **literal constant***) *of the associated kind*. Examples:

   Instances of Java's *floating-pt. literal token*: 2.3, 4.1f, 3e-4
   Instances of Java's *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 examples are: **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 examples are: **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 an entirely different role from that of an identifier, but which are ***not*** reserved words because it is 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 examples are: **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) ***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 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***.

Use of Grammars to Define *Syntactically* Valid Code

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, the language designer 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 to a limited extent only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "to a limited extent" means "only if" may only hold under certain conditions, and some exceptions are allowed.

Use of Grammars to Define *Syntactically* Valid Code

For many programming languages L, the language designer 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 to a limited extent only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "to a limited extent" means "only if" may only hold under certain conditions, and some exceptions 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*.

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

See https://euclid.cs.qc.cuny.edu/316/Syntactic-Validity.pdf for more on syntactic validity.

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[γ] = (γ | <empty>) means "γ is optional".
{ γ } = (<empty> | (γ) | (γ)(γ) | (γ)(γ)(γ) | ...) means "0 *or more* γs".

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 ***<u>infinite</u>*** collection of BNF productions, including productions such as

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