Programming Language Principles

Programming Language Theory

This Week's Topics

- Syntax, Semantics, Pragmatics
- How to define a language mathematically?
  - Formal Language Basics
  - **Backus-Naur Form (BNF)**
  - **Context-free Grammar (Syntax)**
  - **Parsing and Ambiguity**

We already learned Grammar and Language

- G = (V, T, S, P)
- A grammar defines how strings (sentences) of a language can be generated.
- We can use such grammars and notations we've learned for programming languages too.
- However, in PL, there is another notation for specifying grammars of programming languages.

Backus Naur Form

- Originally Backus Normal Form, developed by John Backus.
- After expanded and used by Peter Naur, the name was changed to ***Backus-Naur Form (BNF)*** by the suggestion of Donald Knuth.
- It is a notation technique for *context-free grammars*.

BNF

- ***Variables (or nonterminals)***: enclosed in brackets <, >
  - `<expression>, <term>, <operator>`
- ***Terminal symbols***: without any marking.
  - `int, void, for`
- Use ::= instead of →.
- Use ‘|’ to represent ‘or’.
  - `<bool-literal> ::= true|false`

Example: Real Number

- <real-num> ::= <int-part>.<frac-part>
- <int-part> ::= <digit>|<int-part><digit>
- <frac-part> ::= <digit>|<digit><frac-part>
- <digit> ::= 0|1|2|3|4|5|6|7|8|9
- Start nonterminal is <real-num>.

Left-most Derivation

- Derive the leftmost nonterminal first, if there are more than one nonterminal.

- 3.14

  - <real-num> ⇒ <int-part>.<frac-part>

  - ⇒ <digit>.<frac-part> ⇒ 3.<frac-part>

  - ⇒ 3.<digit><frac-part> ⇒ 3.1<frac-part>

  - ⇒ 3.1<digit> ⇒ 3.14

Right-most Derivation

- Let’s derive (())
- <balanced> ::= (<balanced>)<balanced>|ε
- <balanced> $\Rightarrow$ (<balanced>)$\boxed{\texttt{<balanced>}}$
- $\Rightarrow$ (<balanced>)ε $\Rightarrow$ ($\boxed{\texttt{<balanced>}}$)
- $\Rightarrow$ ((<balanced>)$\boxed{\texttt{<balanced>}}$) $\Rightarrow$ ((<balanced>)ε)
- $\Rightarrow$ (($\boxed{\texttt{<balanced>}}$)) $\Rightarrow$ ((ε)) $\Rightarrow$ (())

Extended BNF

- Or simply **EBNF**, has the same expressive power as BNF, but much simpler.
- { X } : repeat X 0 or more times.
    - `<statements> ::= {<statement>;}`
- [X] : X is optional. You can also use ‘?’ like regular expression style.
    - `<signed> ::= [‘-’]<num>`
    - `<signed> ::= ‘-’?<num>`

Extended BNF

- We can also use some regular expression like notations.
  - `*: <expr> ::= <digit>|ε`
    - `–> <expr> ::= <digit>*`
  - `+: <expr> ::= <digits>|<digit><digits>`
    - `–> <expr> ::= <digit>+`
- (X) : for grouping.
  - `<id> ::= <letter>|<id><letter>|<id><digit>`
    - `—> <id> ::= <letter> (<letter>|<digit>)*`

Real Number Again

- In **BNF**, let’s consider full spec. here.
- `<real-num> ::= ‘-’<num>|<num>`
- `<num> ::= <digits>|<digits>.<digits>`
- `<digits> ::= <digit>|<digit><digits>`
- `<digit> ::= 0|1|2|3|4|5|6|7|8|9`

Real Number Again

- In **EBNF**,
- `<real-num> ::= [‘-’] <digit>+ [‘.’<digit>+]`
- `<digit> ::= 0|1|2|3|4|5|6|7|8|9`
- A lot simpler than BNF.
- Using ‘?’ instead.
- `<real-num> ::= ‘-’? <digit>+ (‘.’<digit>+)?`

Context-free Language

- G = (V, T, S, P) is ***context-free***, if all productions in P have the form
    - $A \rightarrow x$
- where $A \in V$, $x \in (V \cup T)^*$.
- L is context-free iff. there exists a context-free grammar G such that L = L(G).
- Meaning that ***allowing only one variable on the left side***.

Why is it Context-Free?

- Suppose a grammar with productions contain something else on the left.
  - xAy ➞ b :   xAyb ⇒ bb **OK!**   xAb ⇒ bb **Wrong!**
  - xA ➞ c :   xAb ⇒ cb **OK!**   xAyb ⇒ cyb **OK!**   yAxb ⇒ ycxb **Wrong!**
  - Each production can only be applied to a certain sequence of strings (i.e. context).
- On the other hand, we can always replace a variable when it appears during derivation with context-free grammar.
  - A ➞ Ab | Bc
  - B ➞ Ba | b
  - `<expr> ::= <digit>|`ε

Parsing

- So far, we were talking about ‘generative’ aspect of grammars.
  - Given a grammar G, which set of strings can be derived by G?
- What if we want to know that, for a given string s of terminals,
  - whether or not $s \in L(G)$.

Parsing

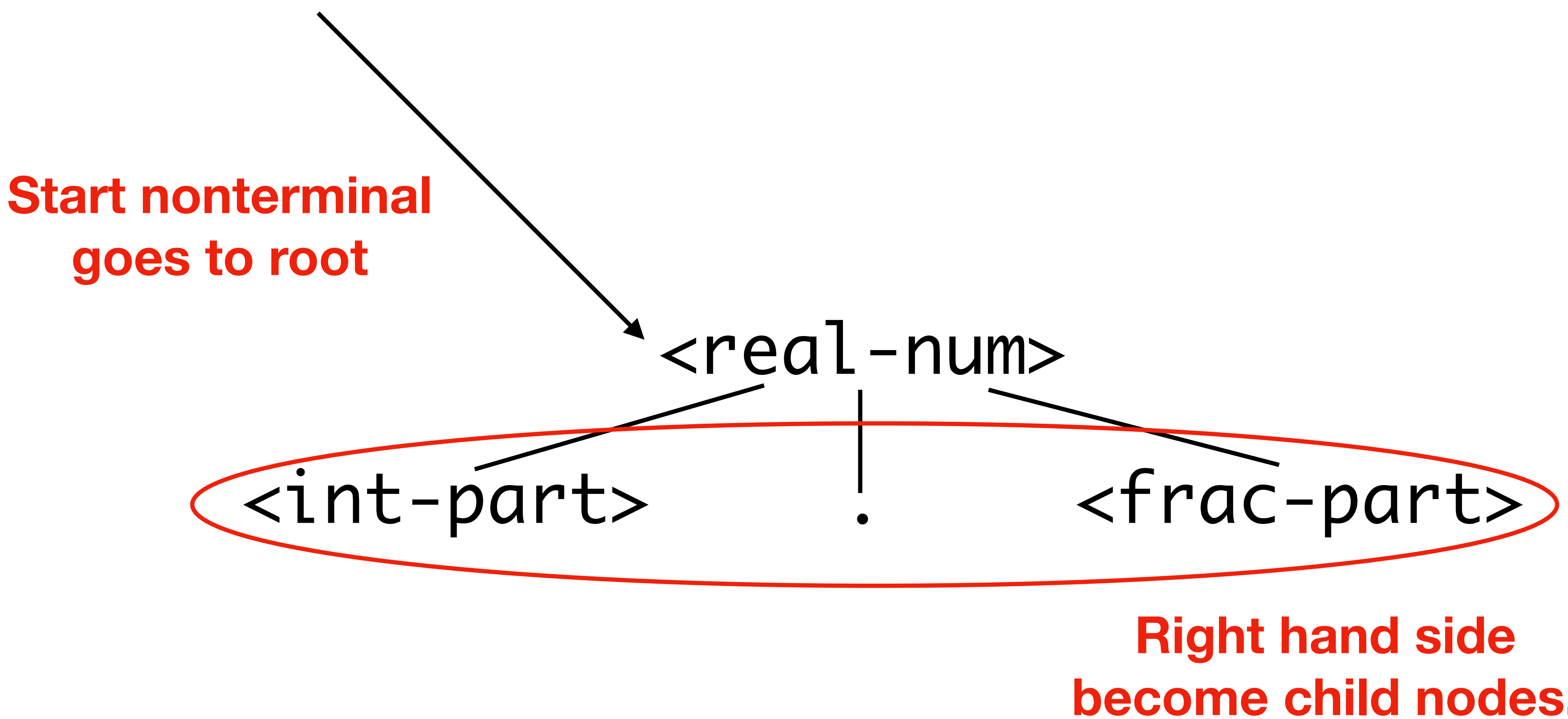
- **Parsing** is finding a sequence of productions by which a $w \in L(G)$ is derived.
- In other words, it answers whether w can be derived by G.
- Parse tree, top-down parsing, bottom-up parsing.

Parse Tree

- To verify an expression (or a string) can be derived by a given BNF, we can construct a **Parse Tree**.
- A parse tree should satisfy the following conditions.
  - All terminal nodes (leaf nodes) are either terminals or ε.
  - All intermediate nodes are nonterminals.
  - Each nonterminal is located on the left hand side, and the right hand side will be the nonterminal's children.
  - The root node is the start nonterminal.

Parsing 3.14

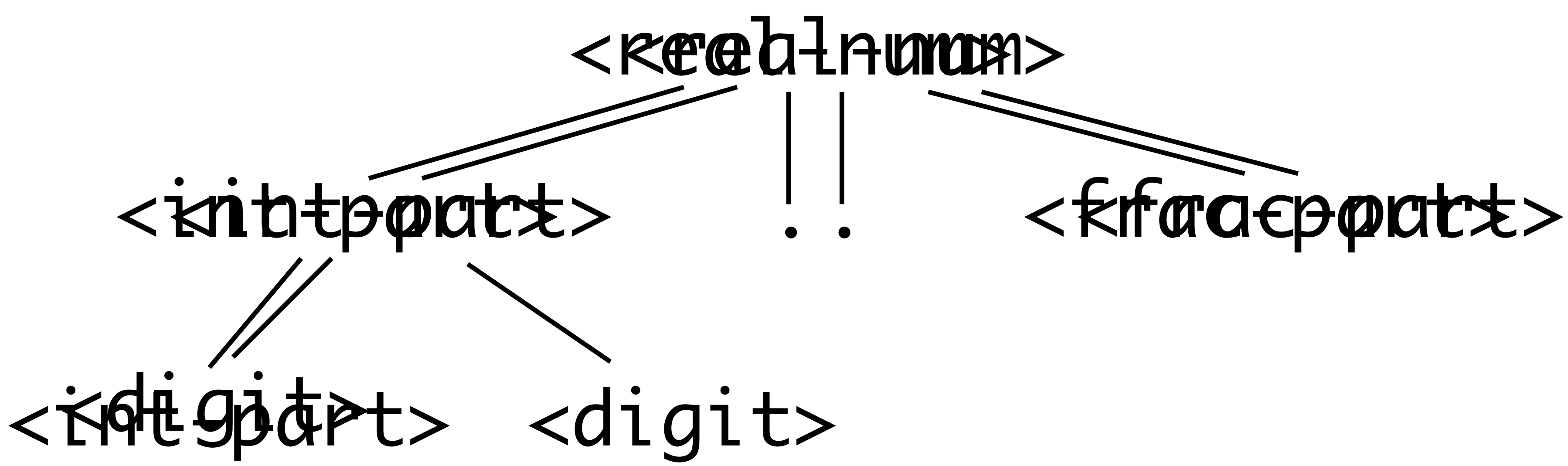
- 3.14
  - <real-num> ⇒ <int-part>.<frac-part>



Parsing 3.14

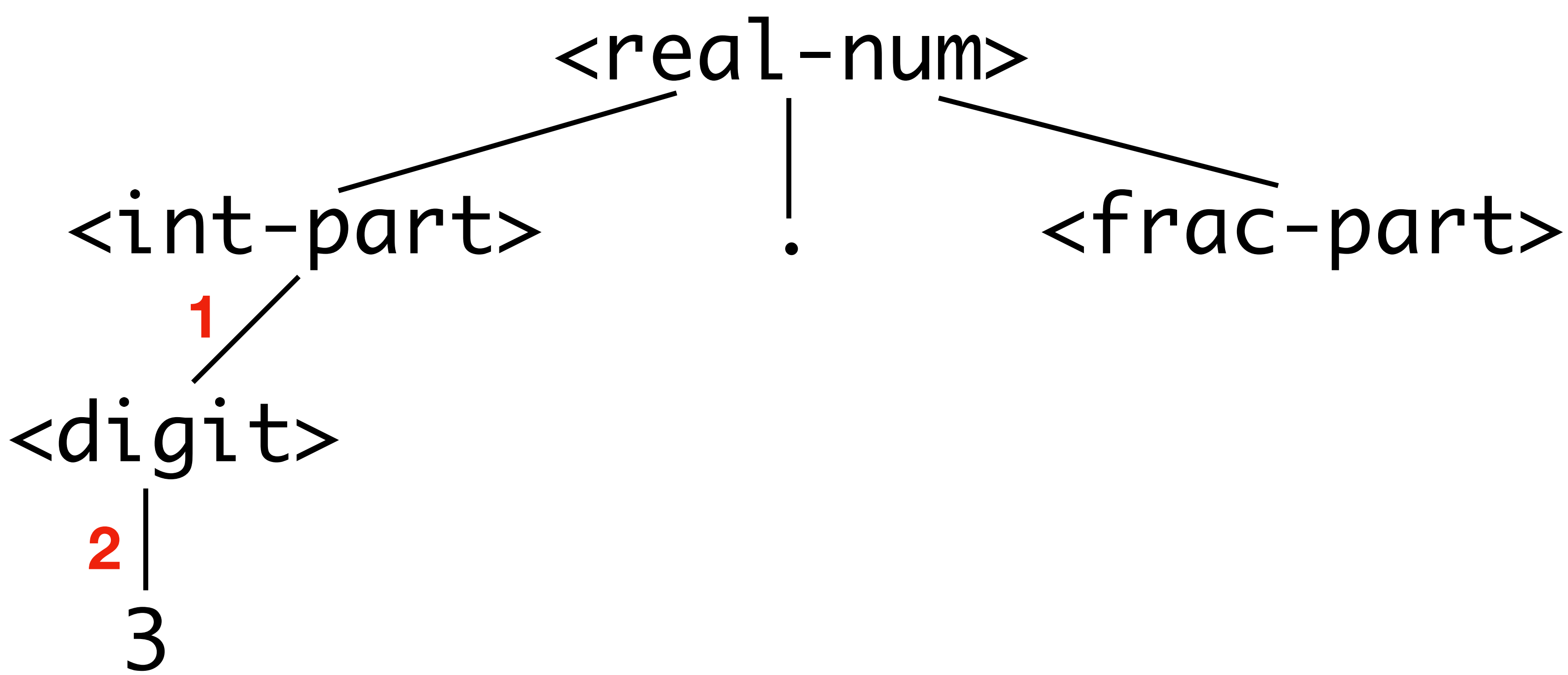
- <real-num> ⇒ <int-part>.<frac-part>
  - ⇒ <digit>.<frac-part> **try this**
  - ⇒ <int-part><digit>.<frac-part> **try this**

```
<int-part> ::=
     <digit>|<int-part><digit>
```



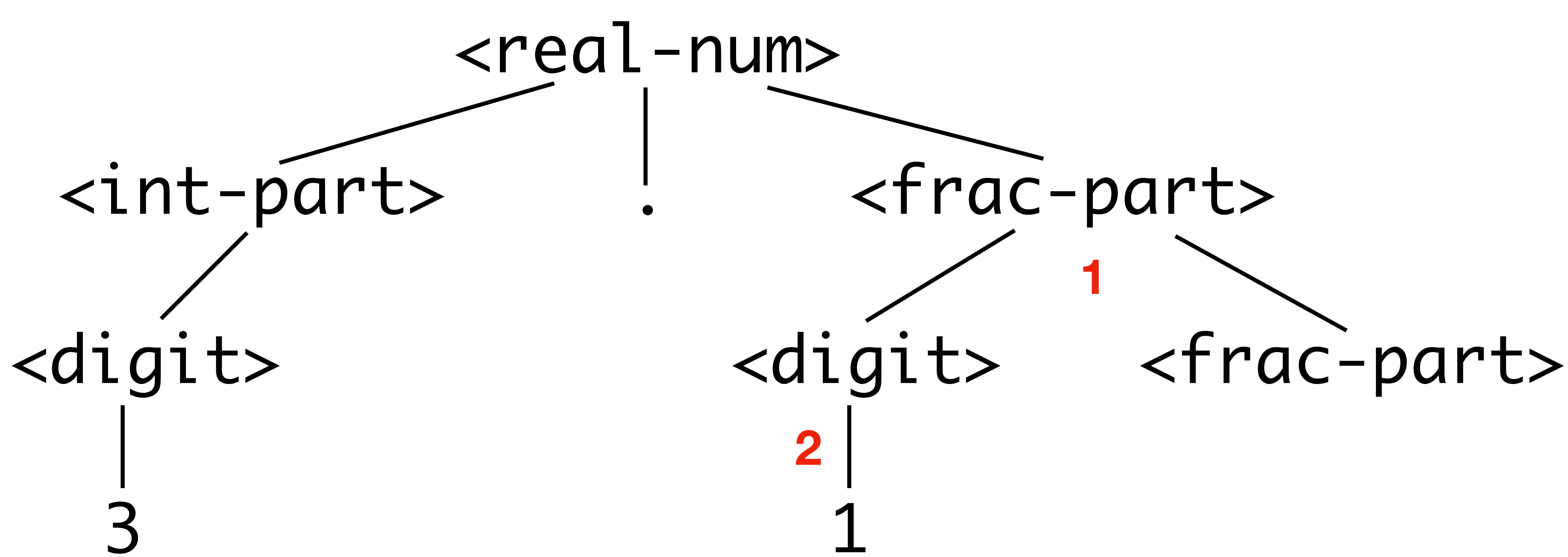
Parsing 3.14

- <real-num> ⇒ <int-part>.<frac-part>
    - $\overset{1}{\Rightarrow}$ <digit>.<frac-part> $\overset{2}{\Rightarrow}$ 3.<frac-part>



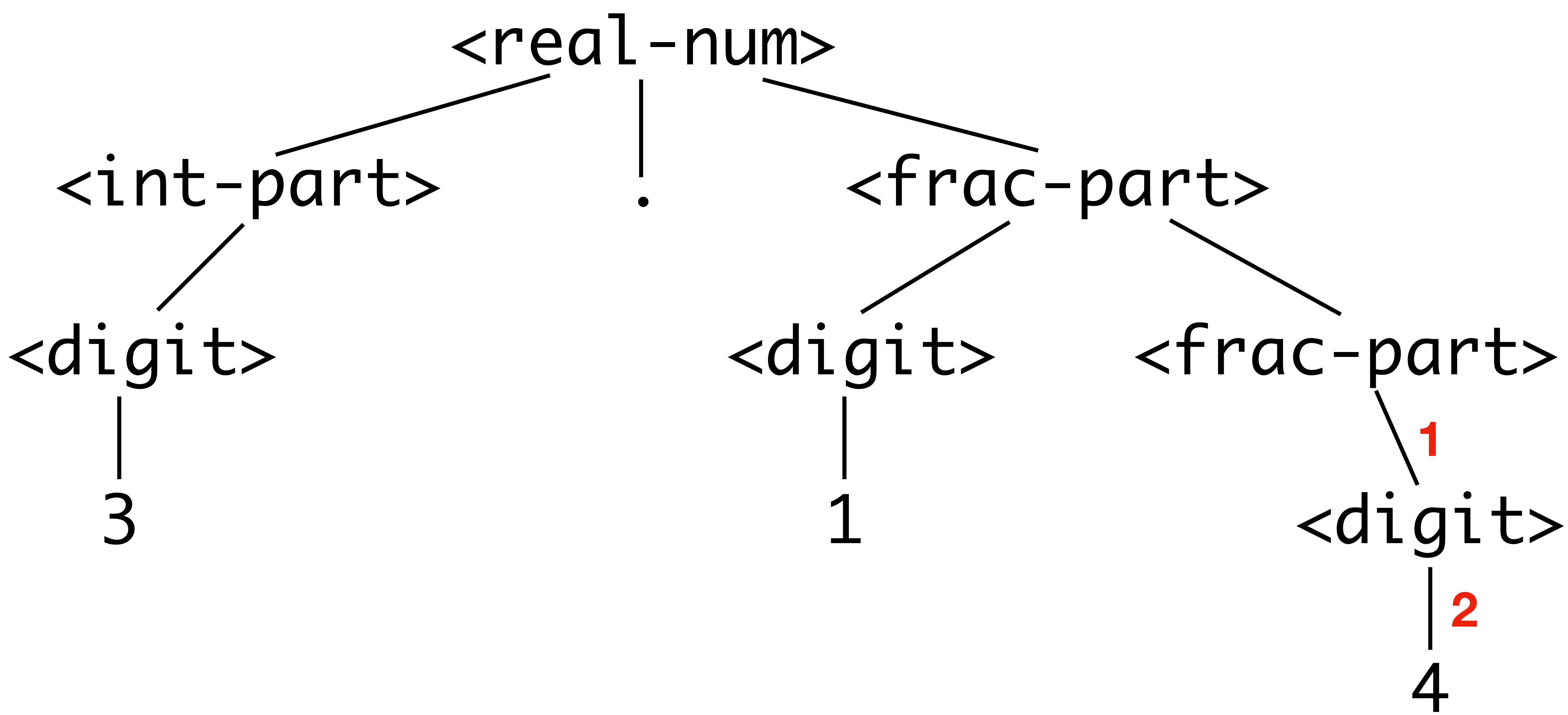
Parsing 3.14

- 3.<frac-part>
  - $\overset{1}{\Rightarrow}$ 3.<digit><frac-part> $\overset{2}{\Rightarrow}$ 3.1<frac-part>



Parsing 3.14

- 3.1<frac-part>
  - $\overset{1}{\Rightarrow}$ 3.1<digit> $\overset{2}{\Rightarrow}$ 3.14



Top-down Parsing

- **Top-down parsing** starts from the start nonterminal (i.e., root).
- For each round of parsing, ***it checks all possible productions*** to be applied to nonterminals.
- Hence it is also called **exhaustive search parsing**.
- <int-part> ::= <digit>|<int-part><digit>
    - <int-part>.<frac-part> ⇒ <digit>.<frac-part>
    - <int-part>.<frac-part>
⇒ <int-part><digit>.<frac-part>

Flaws in Top-down Parsing

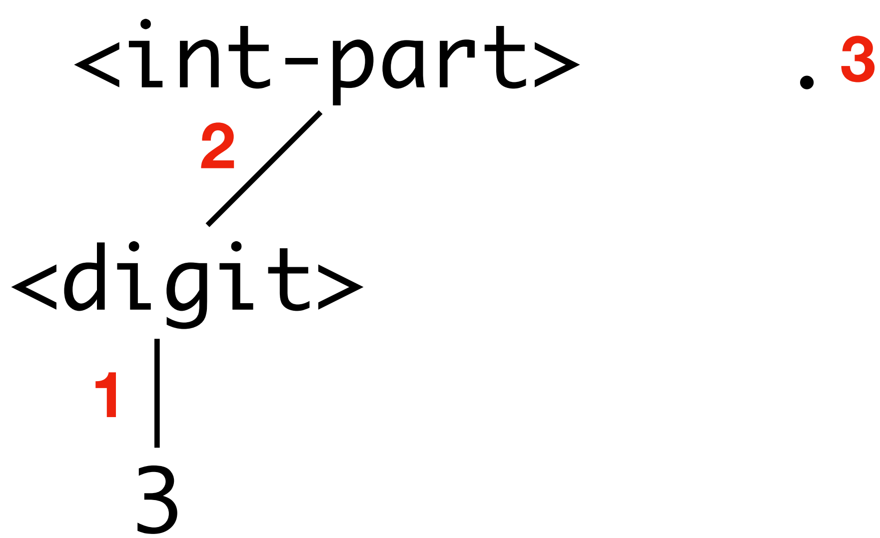
- It’s very tedious.
    - We have to verify every possible productions for each step, until we find the target expression.
    - This is not efficient way of parsing.
- It doesn’t terminate, if a given string ***w*** is not in L(G).
    - In other words, if **w** cannot be derived by given BNF, parsing will never end.

Bottom-up Parsing

- Conversely, we can ***reduce terminals*** of given string w to a nonterminal using BNF.
    - e.g.) `3.14` ⇒ `<digit>.14`
- Usually it reads the input text from left to right, and finds nonterminal to replace terminals in the text.

Parsing 3.14

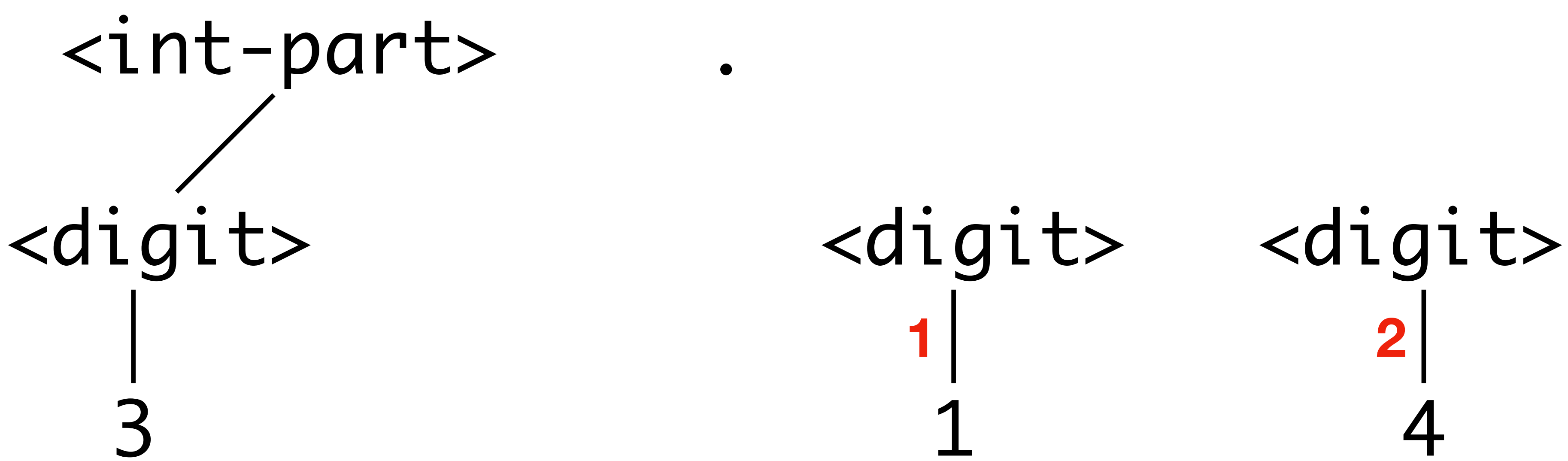
- 3.14
  - $\overset{1}{\Leftarrow}$ <digit>.14 $\overset{2}{\Leftarrow}$ <int-part>.14 $\overset{3}{\Leftarrow}$ <int-part>.14
    ?



Parsing 3.14

- <int-part>.14
    - $\overset{1}{\Leftarrow}$ <int-part>.<digit>4

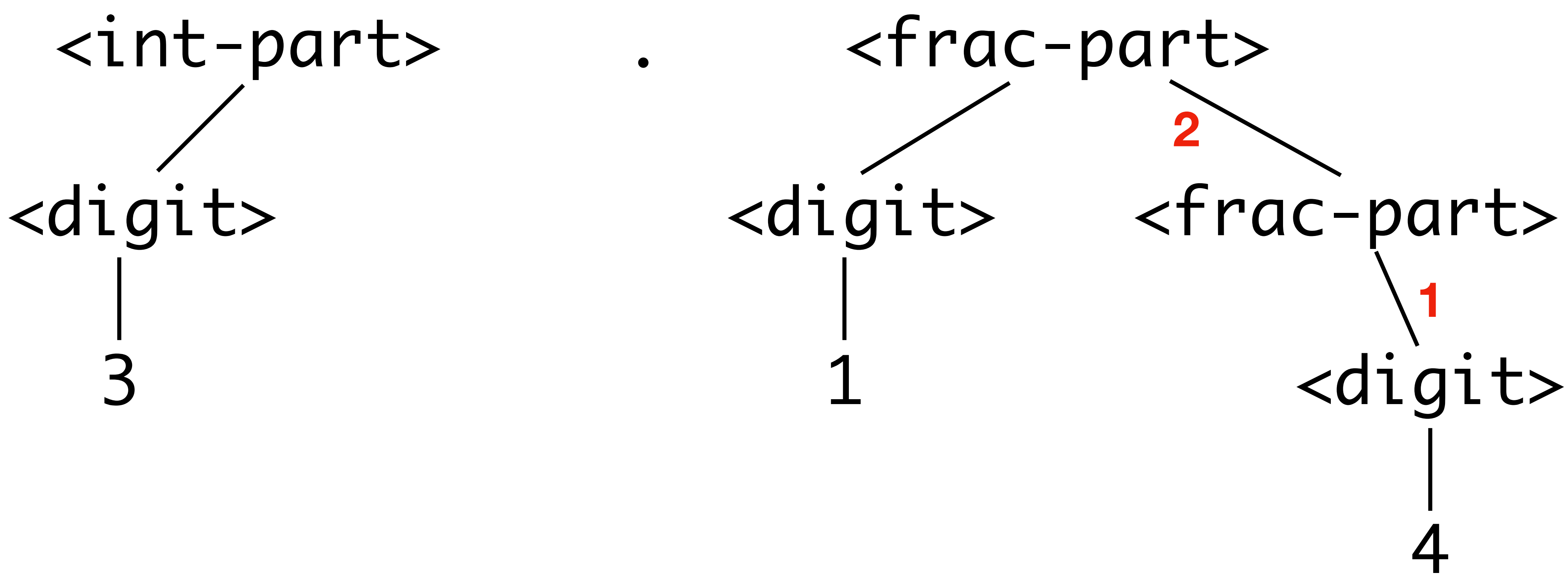
      ?
    - $\overset{2}{\Leftarrow}$ <int-part>.<digit><digit>



Parsing 3.14

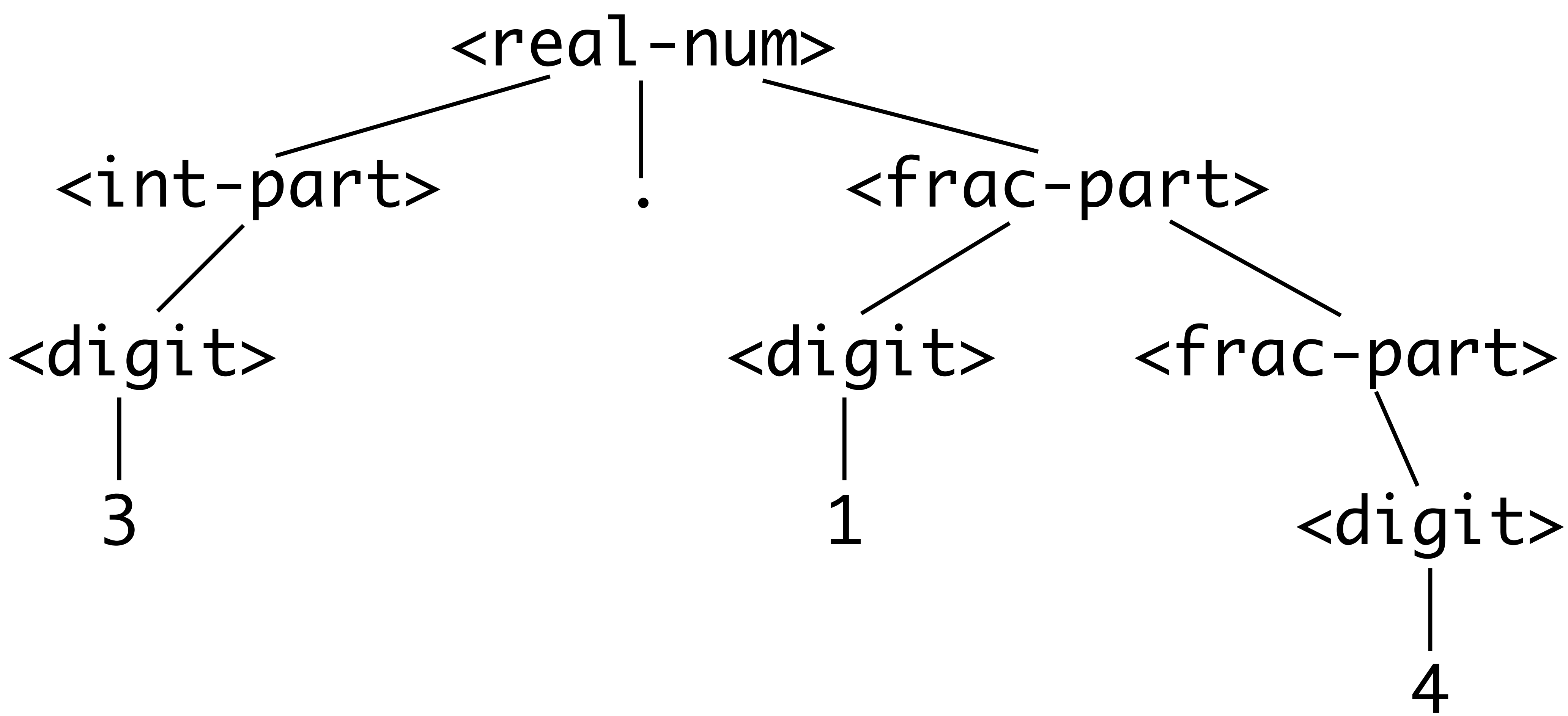
- <int-part>.<digit><digit>
  - $\stackrel{1}{\Leftarrow}$ <int-part>.<digit><frac-part>
  - $\stackrel{2}{\Leftarrow}$ <int-part>.<frac-part>

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



Parsing 3.14

- <int-part>.<frac-part> ⇐ <real-num>



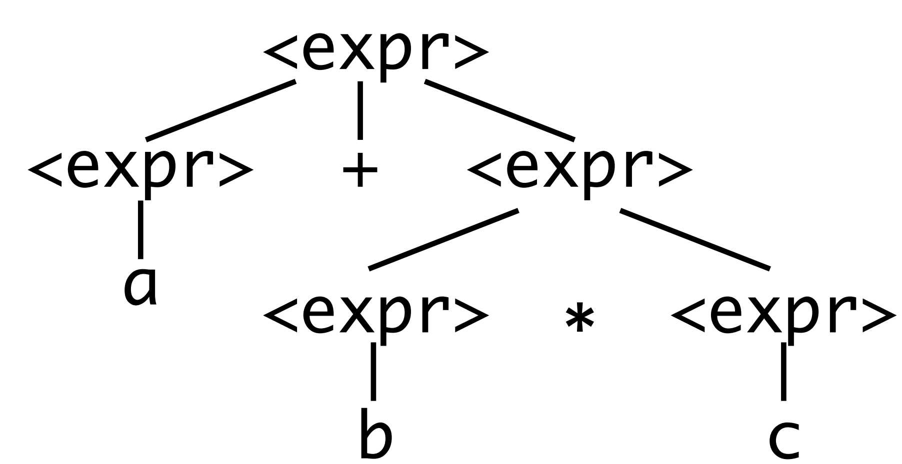
Ambiguity

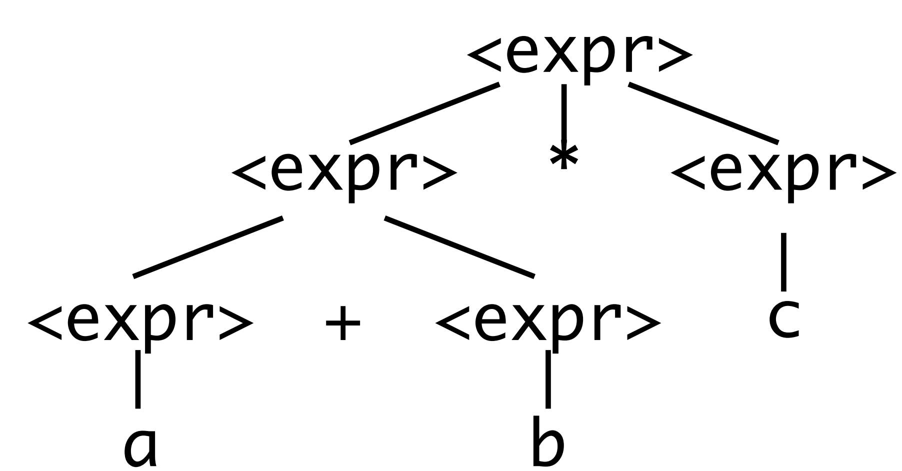
- If there exist more than one production, which one should be applied?
  - For `<digit>.14`, we can reduce `<digit>` into two different nonterminals.
  - `<int-part> ::= <digit>|<int-part><digit>`
  - `<frac-part> ::= <digit>|<digit><frac-part>`
  - For `<int-part>.<digit>4`, we can reduce `<digit>` further, or just move onto the next.

Ambiguity

- Let’s consider another example.
- <expr> ::= <expr> + <expr>
  | <expr> * <expr>
  | a | b | c
- Suppose we’re parsing a + b * c
- Whether we apply <expr>+<expr> or <expr>*<expr> first, there could be two possible parse trees.

<expr> ⇒ <expr>+<expr>





<expr> ⇒ <expr>*<expr>

Ambiguity

- Grammar itself has ambiguity.
- For an input, there are more than one interpretation.
- If a PL has more than one parse tree for the same input, we call the PL is ‘***ambiguous***’.
- For the previous example, we might use operator precedences.
    - This is ***not syntax***, ***but semantics***.
- It is necessary to design syntax carefully, so that ***syntactically correct statement is also semantically correct***.

To Resolve Ambiguity

- One way to resolve ambiguity is to rewrite the grammar.
- Think about the `a + b * c` example again.
-
```
<expr> ::= <expr> + <expr>
         | <expr> * <expr>
         | a | b | c
```
- We know that we have two parse trees for the expression, based on which operator (+, *) is considered first.

To Resolve Ambiguity

- We can introduce new nonterminals.

- ```
  <expr>  ::= <expr> + <expr*> | <expr*>
  <expr*> ::= <expr*> * <var> | <var>
  <var>   ::= a | b | c
  ```

- This example is not that difficult to resolve the ambiguity.

- But usually it is very hard to tell whether a grammar has ambiguity or not, and also to resolve it.
  ```

Summary

- BNF
- Context-free Grammar
- Parsing and Ambiguity