Programming Language Principles

Programming Language Theory

Topics

- How to implement a programming language?
- Syntax, Semantics, Pragmatics
- How to define a language syntax?
    - Backus-Naur Form (BNF)
    - Parsing and Ambiguity

How to Implement a PL?

- Human: high-level languages are more easy to understand.
- Computer: it can only understand machine instructions.
- PL implementation: implement a process of ***translating high-level language to low-level language***, so that a program written by human can be directly executed by a computer.
- Such translation is done by a compiler or an interpreter.
- Eventually *PL implementation is equivalent to compiler or interpreter implementation*.

Compiler vs. Interpreter

- Both a compiler and an interpreter convert code written by human to low-level code which a machine can execute.
- Compiler
    - complete code → executable program.
- Interpreter
    - read & evaluate expressions → execute instructions.

Compiler vs. Interpreter

- Compiler
  - Focus on execution performance and efficiency of a generated executable program.
  - Relatively difficult to connect code and execution.
  - Find errors at compile time.
- Interpreter
  - Easy to implement, but slower.
  - Runtime errors → directly linked to code.
  - Can execute partial code (only some expressions).

Compiler vs. Interpreter

C++

```cpp
#include<iostream>
using namespace std;

int main() {
    int a = 10, b = 5;
    cout << a / b << endl;

    return 0;
}
```

Compiler

Python

```python
>>> a = 10
>>> b = 5
>>> a / b
2.0
>>> 
```

Interpreter

Compiler

- Convert code in high-level language to machine instructions executable in a target machine.

- Translator between humans and machines.

- A compiler generates code in an object (target) language, and its output is often called an object file.

- Then these object files are combined into one executable program.

Compilation Steps

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Intermediate Code Generation
- Code Optimization
- Code Generation

Lexical Analysis

- The first phase of a compiler.
- Convert source code into a series of **tokens**.
  - e.g.) keywords, literals, identifiers, numbers, operators, etc.
  - int a = 10; → int (keyword), a (identifier), = (operator), 10 (number literal), ; (symbol).
- Remove whitespaces and comments.
- If a token is invalid, it causes an error.

Syntax Analysis

- Now we have a series of tokens.
- In syntax analysis step, we verify whether the sequence of the tokens follows correct syntax.
- This step is often called “Parsing”.
- Producing Abstract Syntax Tree (AST) or Parse Tree, which represents syntactic structure of source code.
- Code which cannot be parsed → syntactically incorrect!

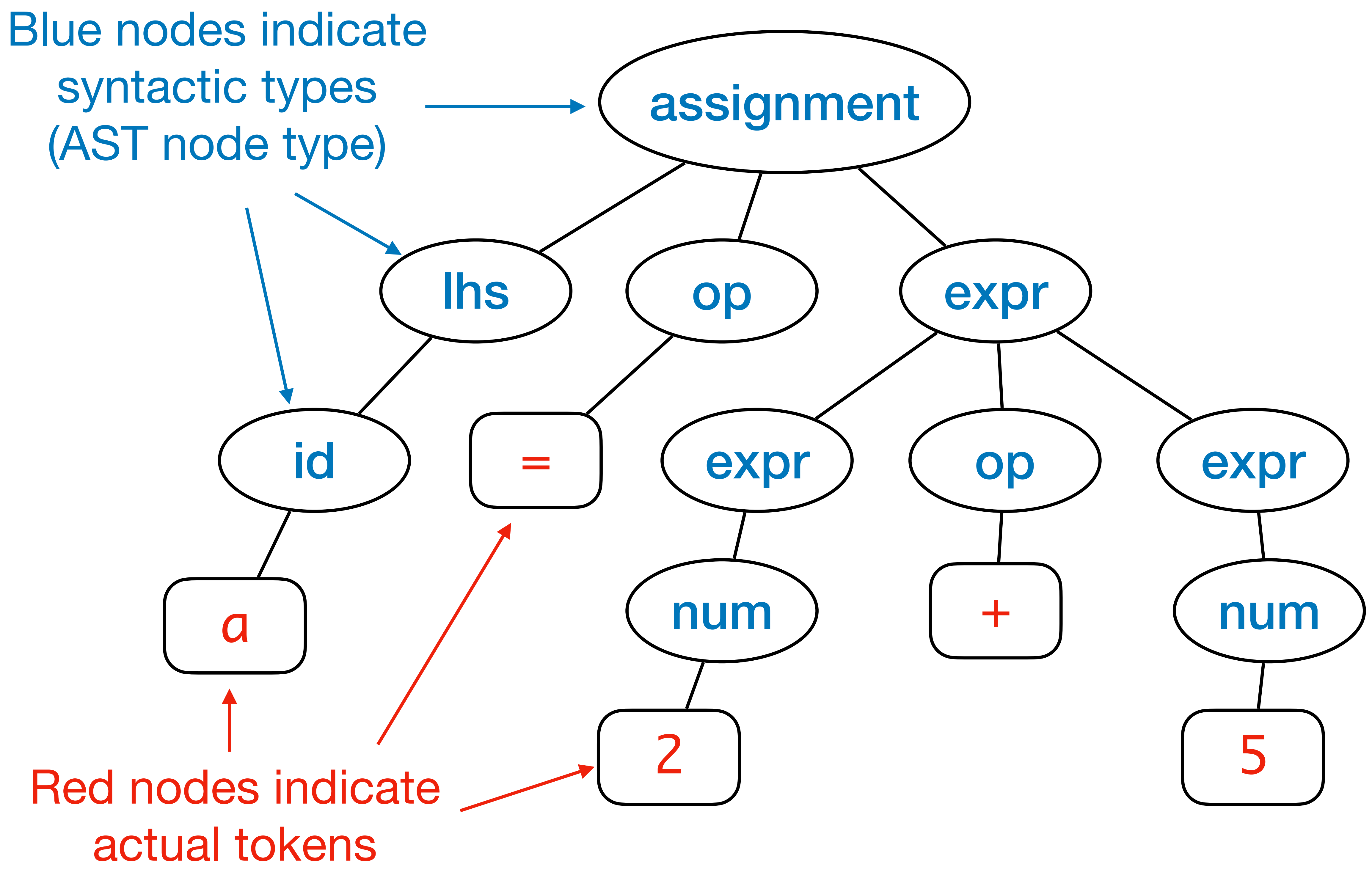
Syntax Analysis

```
Code: a = 2 + 5;
```

Lexical Analysis

```
         a (id)
         = (op)
         2 (num)
Tokens:  + (op)
         5 (num)
         ; (sym)
```

Parse Tree



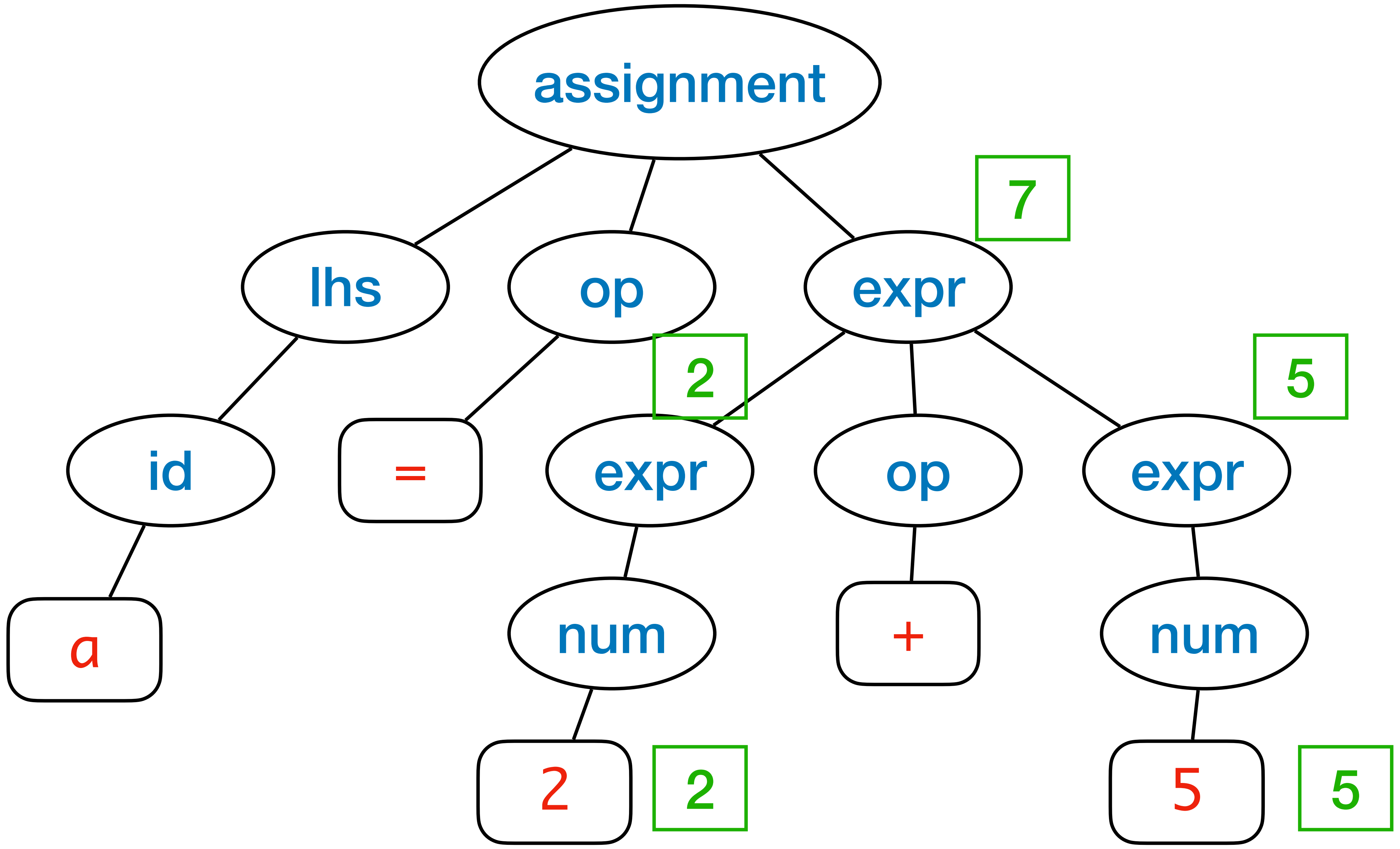
<assignment> → <lhs> <op> <expr>

Semantic Analysis

- Parse tree is “annotated” or “decorated” by semantic analysis.
- The information gathered during this step will be used for code generation.
- Syntax analysis gives nothing about what the code **means**.
- Hence we need to figure out the meaning of code - at least evaluate them.
- Often combined with syntax analysis.

Semantic Analysis

Augmented Parse Tree



It means that 7 will be assigned to a

$a \leftarrow 7$

Intermediate Code Generation

- The final outcome of a compiler is often ***machine dependent***.
- Hence machine code is not a desirable target to apply ***machine independent code optimizations***.
- Instead, a compiler generates intermediate code, which is independent to target machines.
- Then it optimizes code first, and further translate them to instructions for a target machine.

Code Optimization

- A compiler automatically performs code optimizations before it generates an executable program.
  - e.g.) dead code elimination, common subexpression elimination, copy propagation, loop optimization, etc.
  - `a=i*j+k; b=i*j*k; → tmp=i*j; a=tmp+k; b=tmp*k;`
- Such optimization might improve your code significantly.
- One of the most important advantage compared to an interpreter.

Code Generation

- Takes intermediate code as an input and produces an equivalent target program.
- Requirements
  - The output code must be correct.
  - The output code should use resource of a target machine effectively.
  - Code generator itself should run efficiently.

Interpreter

- Directly read source code, and execute tasks corresponds to the code.
- Sometimes it implements a virtual computer runs on a real computer, then executes code on the virtual computer.
- It is more easy to implement, occupies less memory than a compiler, but it’s slower.
- e.g.) Python, ML, Scheme, Prolog.

REPL

- ***R**ead-**E**val-**P**rint **L**oop*.
- An interpreter actually repeats the above three tasks.
- Doesn’t require whole program, it simply evaluates what it reads.
- You can write down code at runtime → easy to use.

Syntax vs. Semantics vs. Pragmatics

- ***Syntax*** is about the ***form*** of programs.
- ***Semantics*** is about the ***meaning*** of programs.
- ***Pragmatics*** is the meaning of programs in a certain context.

Syntax vs. Semantics vs. Pragmatics

- ***Syntax***:
  - A mouse is kicking a cat. → **OK!**
  - mouse a cat is a kicking. → **Wrong!**
- ***Semantics***:
  - A mouse is kicking a cat. → Ah, wait… what?
- ***Pragmatics***:
  - The mouse is Jerry and the cat is Tom. → Aha! it’s possible.

Focus on Syntax

- Among the three, we're more interested in ***Syntax*** in this course.
- Before discuss about the others, we need to know how to define a programming language first.
- To talk about the meaning of a program, we first need to say what is a correctly written program.

Formal Language

- At this point, we're more interested in how to define the form of a program.
- How can we determine what is right or wrong for a PL?
- How can we decide whether given code is correctly written in a PL?

Formal Language

- A ***formal language*** is an abstraction of the general characteristics of programming languages.
- A formal language consists of a *set of symbols* and some *rules of formation* by which these symbols can be combined.
- These are defined as a ***grammar*** of the formal language.

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>.
  - When we have a real number, we try to ***derive*** the number starting from it.

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>)<balanced>
- $\Rightarrow$ (<balanced>)ε $\Rightarrow$ (<balanced>)
- $\Rightarrow$ ((<balanced>)<balanced>) $\Rightarrow$ ((<balanced>)ε)
- $\Rightarrow$ ((<balanced>)) $\Rightarrow$ ((ε)) $\Rightarrow$ (())

Extended BNF

- Or simply **EBNF**, has the same expressive power as BNF, but much simpler.
- Employ additional notations.
  - {X}, [X], *, +, (X)
- Verbose and complex BNF expressions are shortened and make them easy to understand.

EBNF Notations

- { X } : repeat X, 0 or more times.
  - `<statements> ::= {<statement>;}`
  - `<statements>` $\Rightarrow^*$ `<statement>; ... <statement>;`
  - `<statements>` $\Rightarrow^*$ `<statement>;`
  - `<statements>` $\Rightarrow^*$ ε

EBNF Notations

- [X] : X is optional. You can also use ‘?’ like regular expression style.
  - `<signed> ::= [‘-’]<num>`
  - `<signed> ::= ‘-’?<num>`
  - `<num> ::= 1|2|3|4`
  - `<signed> ⇒* 1  <signed> ⇒* -1`
`<signed> ⇒* 2  <signed> ⇒* -4`

EBNF Notations

- We can also use some regular expression like notations.
    - *: `<stmts> ::= <stmt>;|<stmts><stmt>;|`ε
        - `-> <stmts> ::= <stmt>;* repeat 0 or more`
    - +: `<digits> ::= <digit>|<digit><digits>`
        - `-> <digit> ::= <digit>+ repeat at least once`

EBNF Notations

- (X) : for grouping. Symbols are applied to the whole grouped terminals, nonterminals.
  - `<nums> ::= (+|-)*<num>(,<num>)+`
  - +, - are repeated 0 or more times.
  - For every repetition, we can choose + or -.
    - e.g.) ++--+-<num>,<num>,<num>
  - After one <num>, ",<num>" will be added at least once, or more.
    - e.g.) +<num>,<num> or <num>,<num> or <num>,<num>,<num> ...

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>+)?`

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.

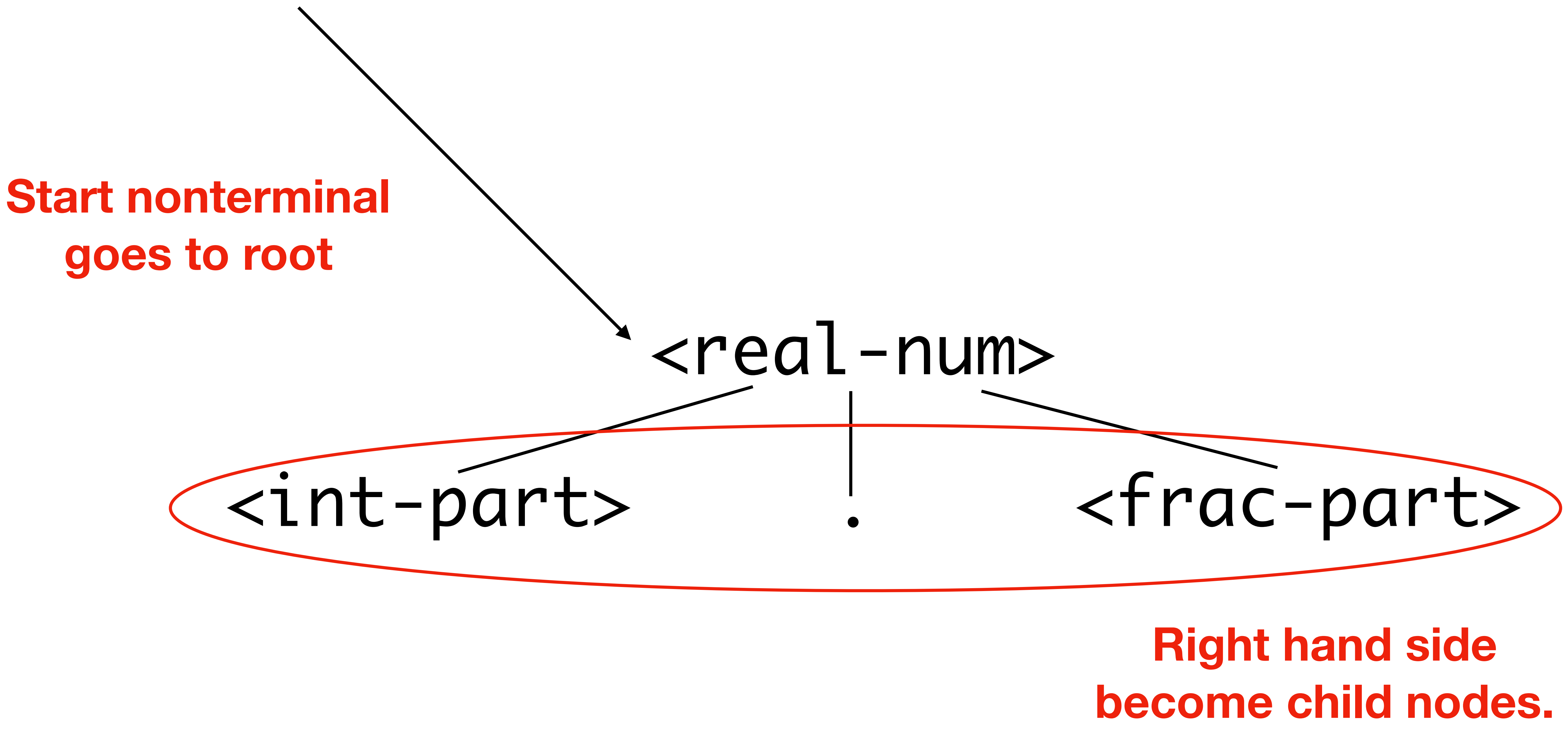
  - All intermediate nodes are nonterminals.

  - 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

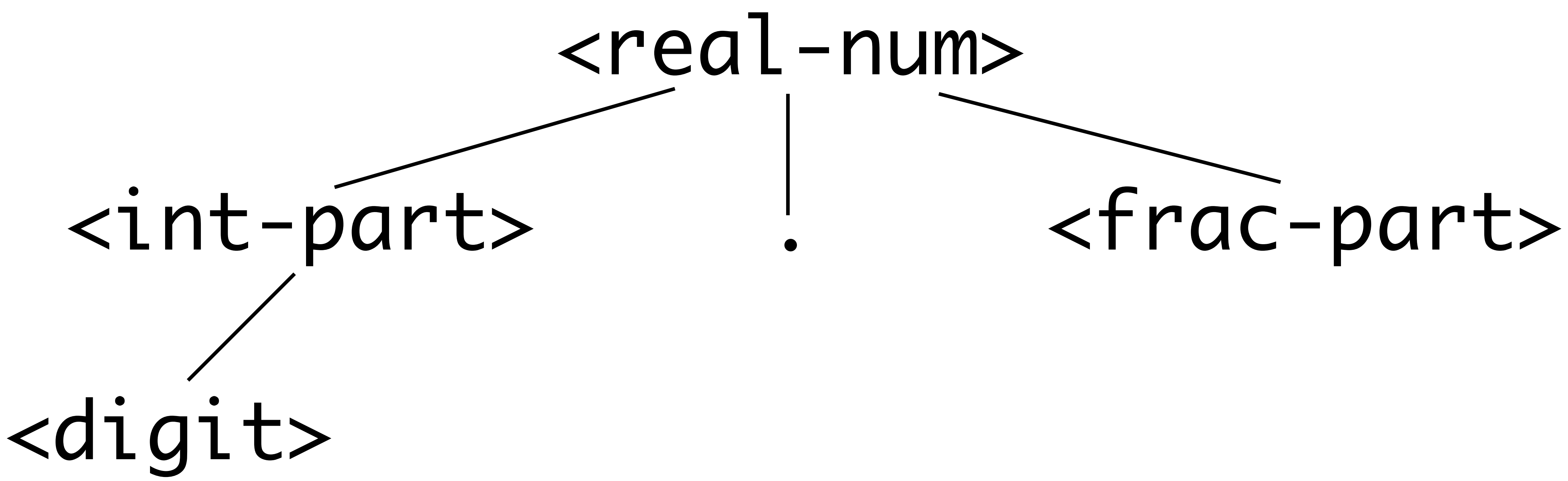
- 3.14
  - <real-num> ⇒ <int-part>.<frac-part>



Parsing 3.14

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

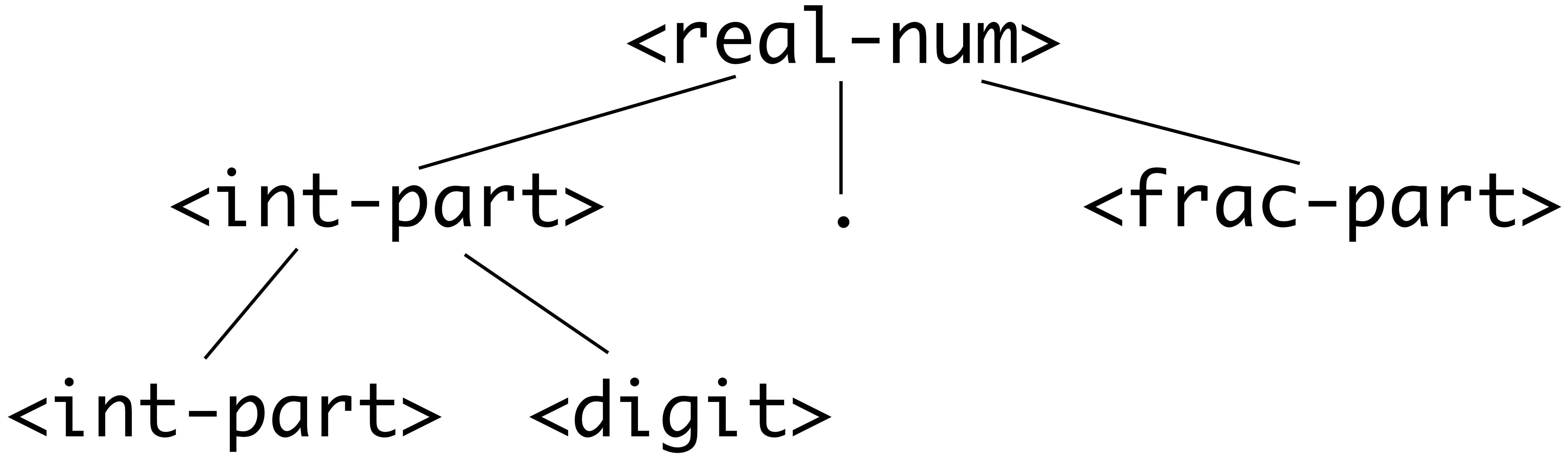
<int-part> ::=
    <digit>|<int-part><digit>



Parsing 3.14

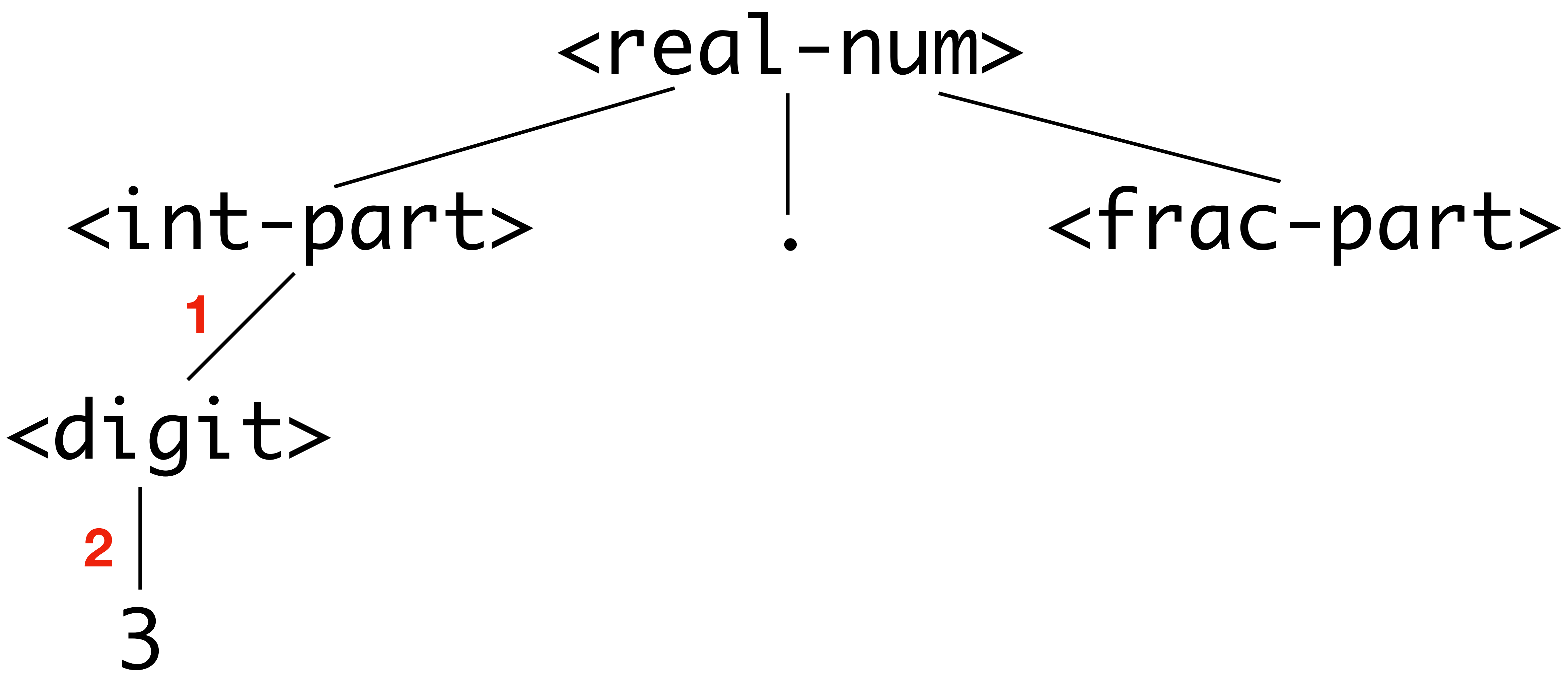
- <real-num> ⇒ <int-part>.<frac-part>
  - ⇒ <digit>.<frac-part>
  - ⇒ <int-part><digit>.<frac-part> **try this**

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



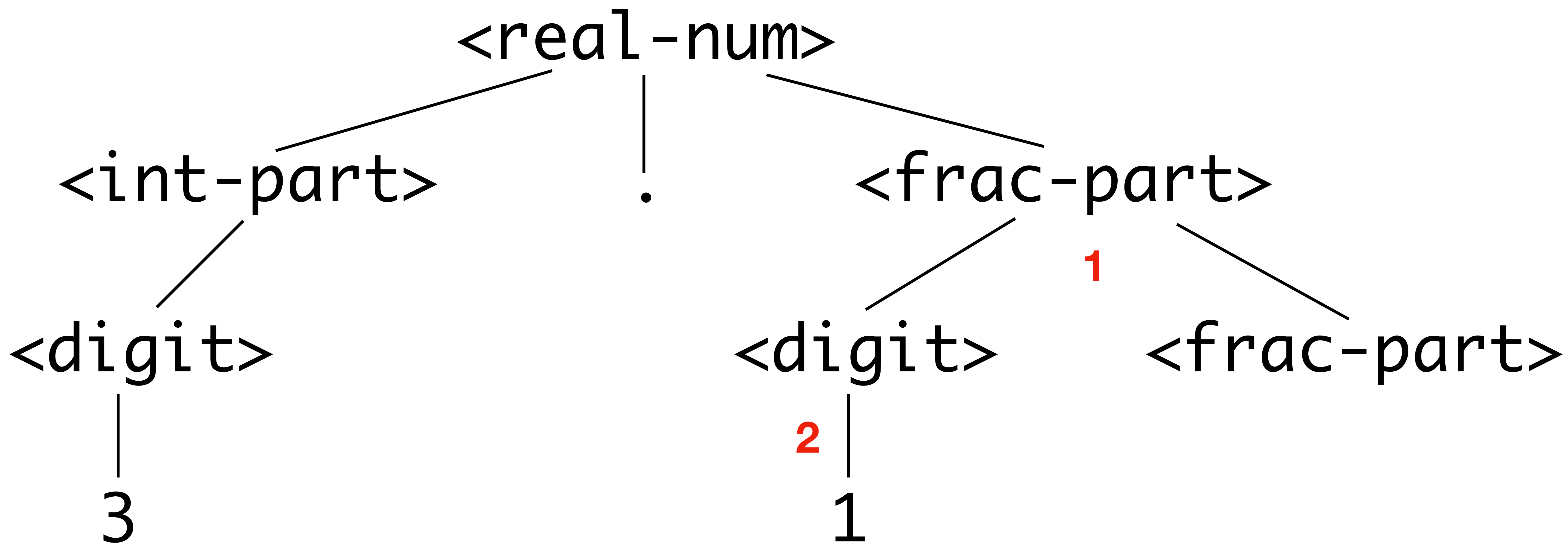
Parsing 3.14

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



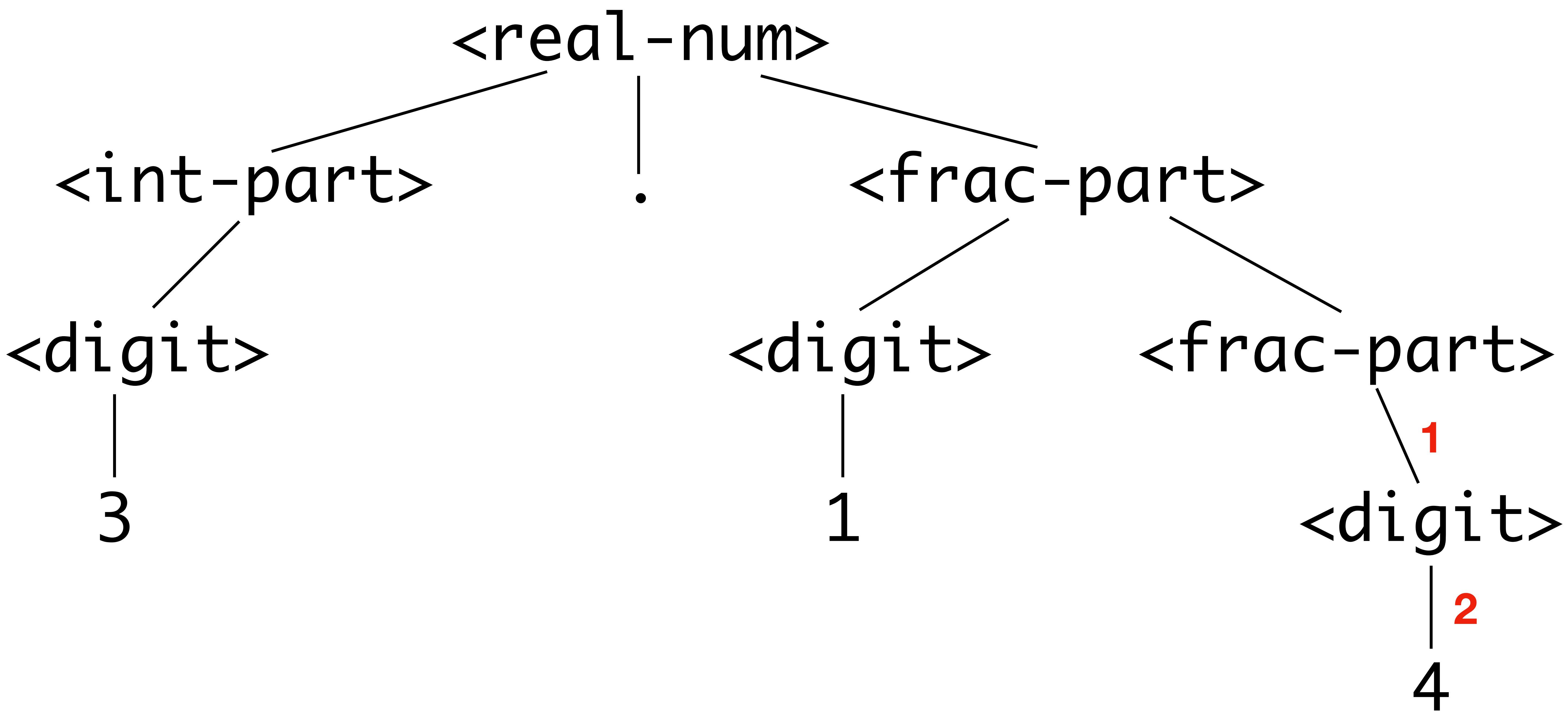
Parsing 3.14

- 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 simply ask the compiler to try every possibility until it finds the right one.
  - This is not efficient way of parsing.
- Non-termination.
  - If a given string cannot be derived by given BNF, parsing will never end.

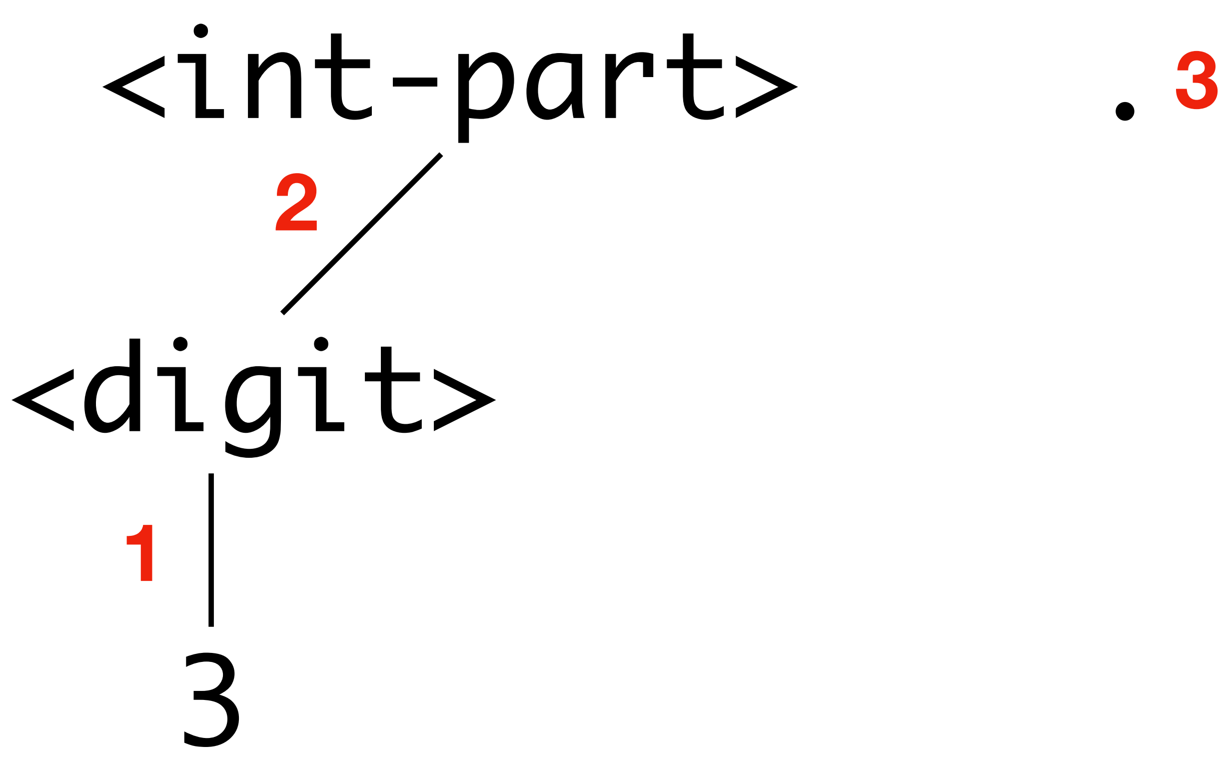
Bottom-up Parsing

- 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.
  - This is the method which compilers are using.

Parsing 3.14

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

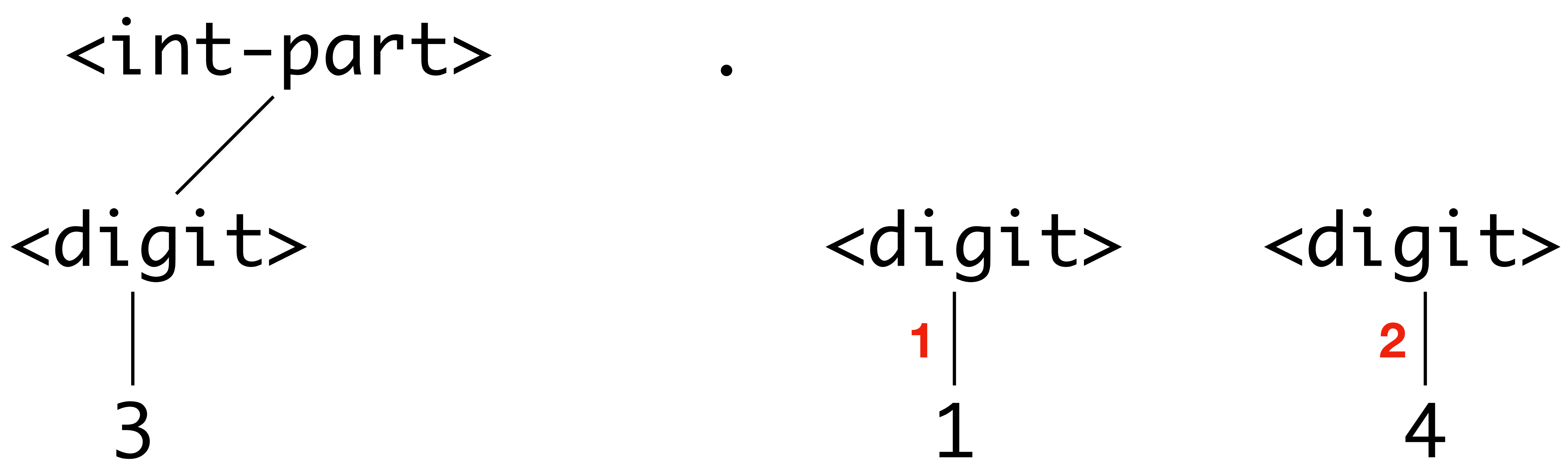
?



Parsing 3.14

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

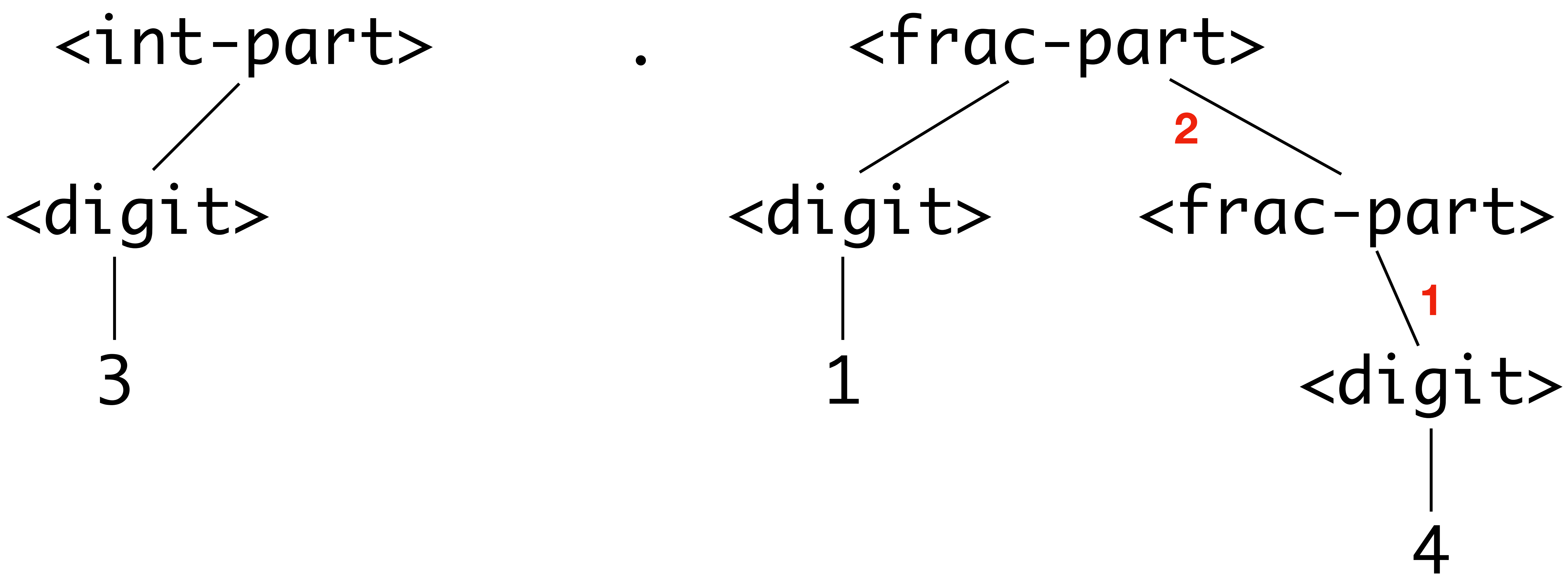
    ?
  - $\overset{2}{\Leftarrow}$ <int-part>.<digit><digit>



Parsing 3.14

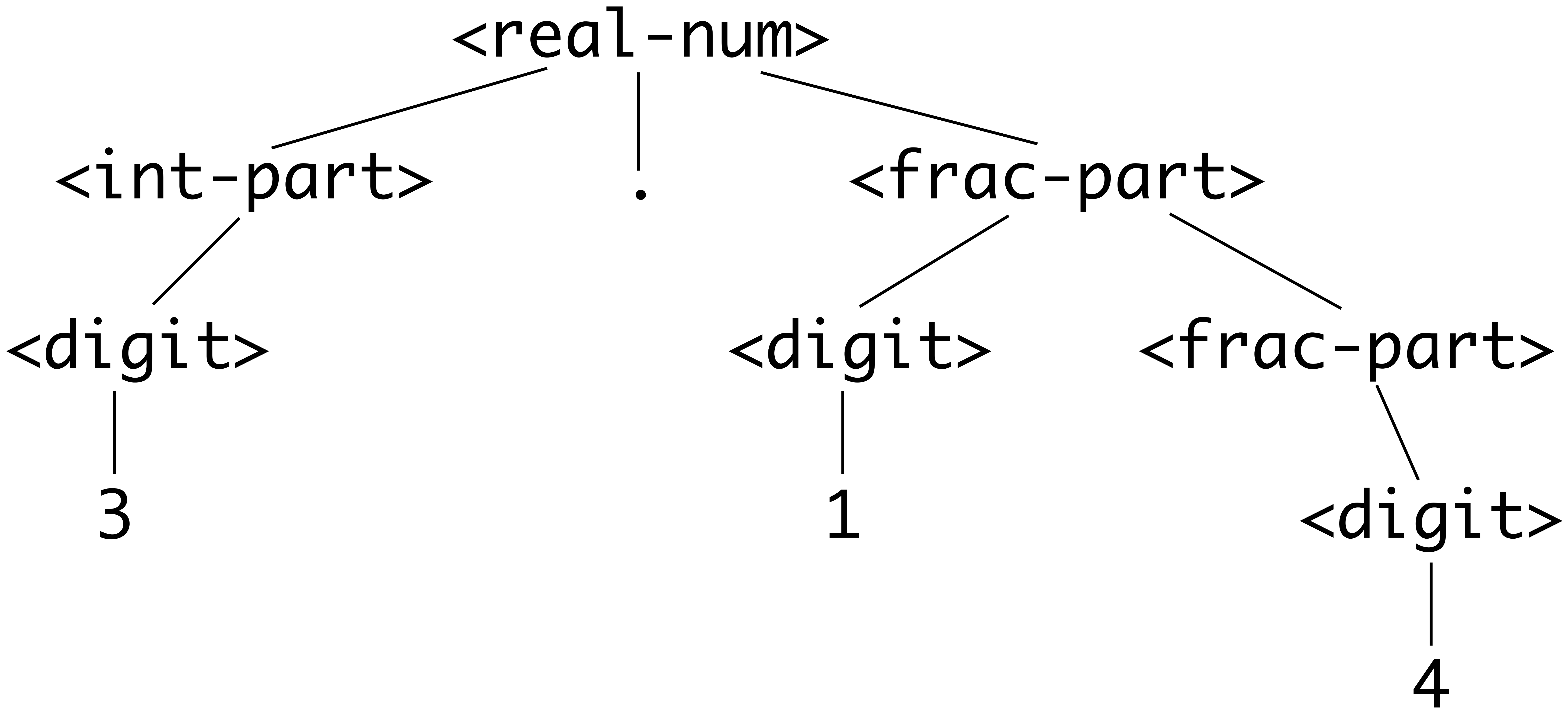
- <int-part>.<digit><digit>
  - $\stackrel{1}{\Leftarrow}$ <int-part>.<digit><frac-part>
  - $\stackrel{2}{\Leftarrow}$ <int-part>.<frac-part>

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



Parsing 3.14

- <int-part>.<frac-part> ⇐ <real-num>



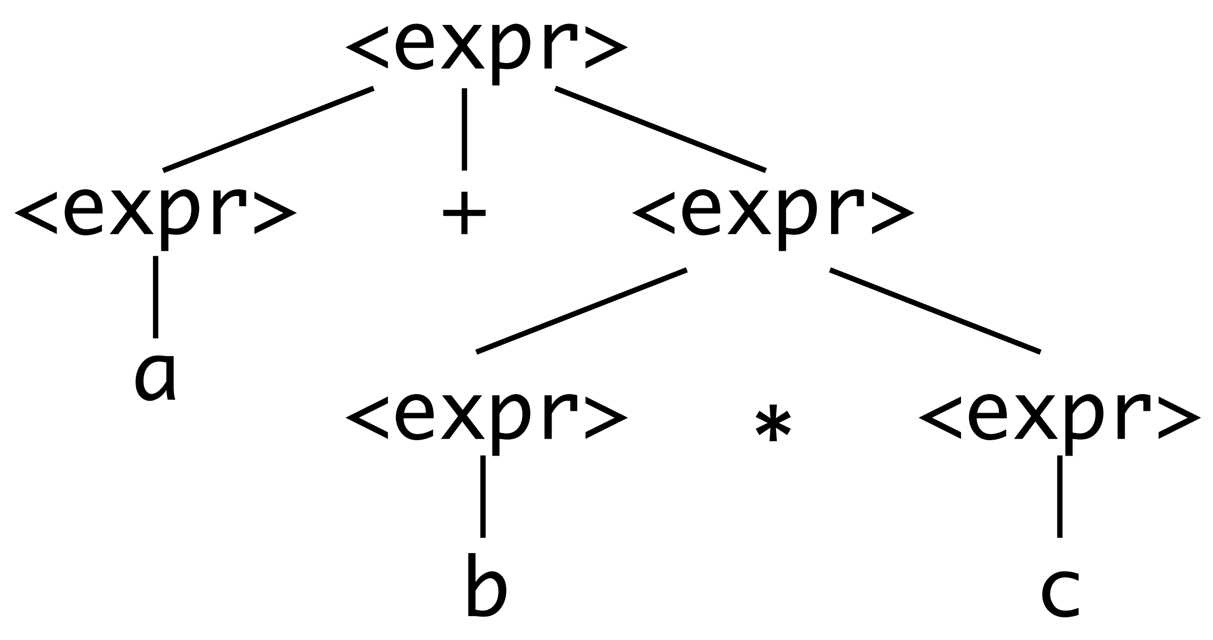
Ambiguity

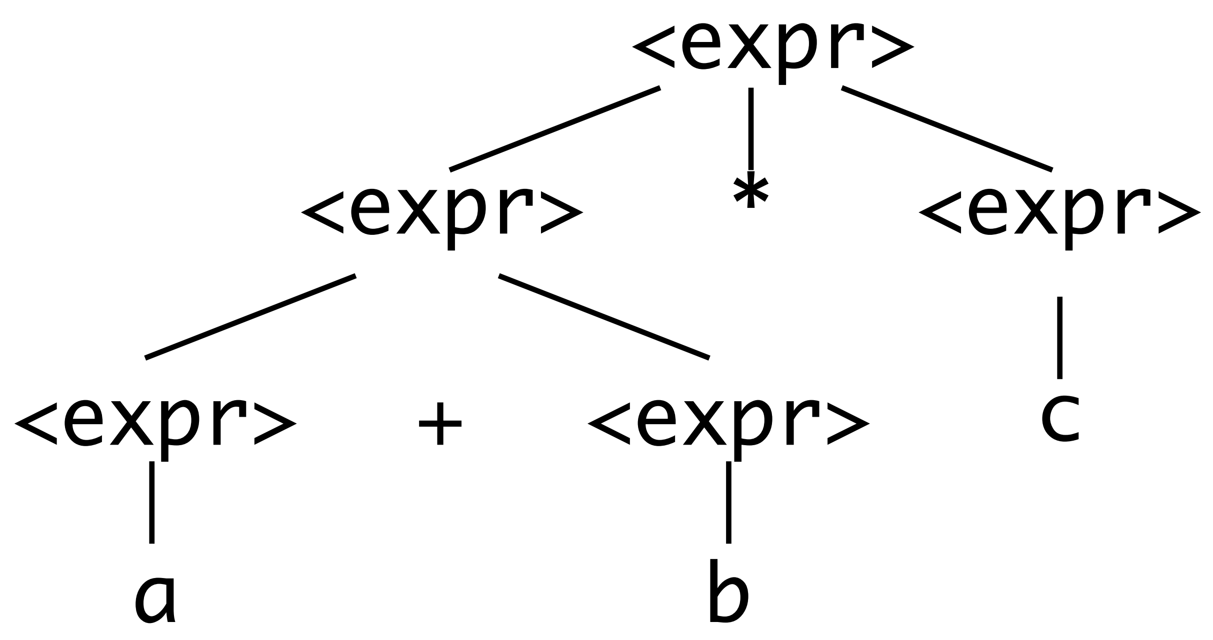
- If there exist more than one production, which one should be applied?
  - For `<digit>.14`, we can reduce `<digit>` into two different nonterminals.
  - `<int-part> ::= <digit>|<int-part><digit>`
  - `<frac-part> ::= <digit>|<digit><frac-part>`
  - For `<int-part>.<digit>4`, we can reduce `<digit>`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

- Compiler vs. Interpreter
- Syntax, Semantics, and Pragmatics
- Formal Language and BNF
- Parsing and Ambiguity