

## Syntax (continued)

---

### BNF Derivation

- To **determine** if a string is **valid** according to a grammar.
- A derivation is **a series of replacements** defined by the productions.
- LL(k) grammars
  - LL(k) grammars may contain right recursion, e.g.  $L \rightarrow D L \mid D$ , but not left recursion
  - **Leftmost derivation**: In each step of a derivation, apply production rule to the **leftmost** symbol.
  - **Top-down approach**
    - The derivation **begins** from the **start symbol**.
    - The production rules **iteratively replace** nonterminal symbols with nonterminal and terminal symbols **until** there are **no more nonterminal** symbols in the string.
    - If the **terminal symbols match** the string, the string is **valid**.
- LR(k) grammars
  - LR(k) grammars may contain left recursion, e.g.  $L \rightarrow L D \mid D$ , but not right recursion
  - **Rightmost derivation**: In each step of a derivation, apply production rule to the **rightmost** symbol. We use a rightmost derivation if the grammar is LR(k) (which means it)
  - **Bottom-up approach**
    - Replace all terminal symbols with nonterminal symbols and then continue to replace nonterminal symbols until the only remaining symbol is the start symbol.
    - **Start from** the **terminal** symbols (leaves) and work **up towards the start symbol** (root).
    - The procedure for determining if the string is legal (according to the given grammar) involves a stack (Note this doesn't tell you how to build the tree.)
      - while there are more tokens
        - Check to see if the tokens currently on the stack are the appropriate right-hand-side for a rule.
        - If so, then pop those items off the stack and push the left-hand-side symbol onto the stack (this is called a **reduce** operation)
        - If not, then push the next token onto the stack

- **Example1:** Given the formal definition of integer written in an LL grammar, check whether 31 is a valid integer. Since it is LL, we use the top-down approach.

$$\begin{aligned} \text{Integer} &\rightarrow \text{Digit} \mid \text{Digit Integer} \\ \text{Digit} &\rightarrow 0 \mid \dots \mid 9 \end{aligned}$$

- Topdown approach

$$\begin{aligned} \text{Integer} &\Rightarrow \text{Digit Integer} \\ &\Rightarrow 3 \text{ Integer} \\ &\Rightarrow 3 \text{ Digit} \\ &\Rightarrow 31 \end{aligned}$$

- **Example2:** Given the formal definition of integer written in an LR grammar, check whether 31 is a valid integer. Since it is an LR grammar, we use the bottom-up approach. I will be drawing a stack sideways (bottom on left, top on right)

$$\begin{aligned} \text{Integer} &\rightarrow \text{Digit} \mid \text{Integer Digit} \\ \text{Digit} &\rightarrow 0 \mid \dots \mid 9 \end{aligned}$$

- Bottom-up approach
- Push 3, stack is [3]
- Reduce 3 to *Digit*, stack is [*Digit*]
- Reduce *Digit* to *Integer*, stack is [*Integer*]
- Push 1, stack is [*Integer*, 1]
- Reduce 1 to *Digit*, stack is [*Integer*, *Digit*]
- Reduce *Integer Digit* to *Digit*, stack is [*Integer*]

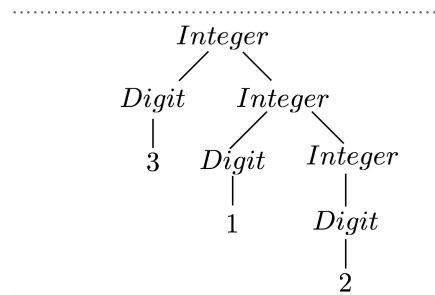
$$\begin{aligned} 312 &\Rightarrow \text{Digit } 1 \ 2 \\ &\Rightarrow \text{Integer } 1 \ 2 \\ &\Rightarrow \text{Integer Digit } 2 \\ &\Rightarrow \text{Integer } 2 \\ &\Rightarrow \text{Integer Digit} \\ &\Rightarrow \text{Integer} \end{aligned}$$

## Parse Tree

- A **graphical form** of derivation. The root of the tree (drawn at the top) is the start symbol.
- Each derivation step corresponds to a new subtree. The parent of each subtree is a nonterminal and its children are the non-terminals and terminals for the right-hand side of the appropriate rule (and if there is more than one option on the right-hand side, then just the appropriate option is represented, e.g. an Integer node may have both a Digit and an Integer node for children, or it may have just a Digit as an only child).
- The leaves of the tree are terminals.
- Reading the leaves of the tree from left to right should be the same as reading the original string.
- Example: Draw the parse tree for 312 by using the following rules [top-down, leftmost]

$$Integer \rightarrow Digit | Digit Integer$$

$$Digit \rightarrow 0 | \dots | 9$$



- The above derivation is representative of a recursive descent (**LL(k)**) parser that starts with the top of parse tree.
- Notice that as we read the tree from top to bottom, the left-most digit (3) is resolved “first” (it is higher up on the tree). Also notice how the right-recursion appears as the chain of Integer non-terminals going from the root via the rightmost children.



## EBNF

- To increase the clarity and brevity of syntax descriptions, Extended BNF (EBNF) was introduced.
- Based on BNF, EBNF includes more metasympols:
  - Curly braces  $\{\}$ : include the enclosed symbols 0 or more times
  - Parentheses  $()$ : include the enclosed symbols 1 or more times
  - Square brackets  $[]$ : indicate an optional sequence of symbols
- Example: Describe the integer formally in EBNF

$$Integer \rightarrow Digit\{Digit\}$$

- The above production cannot avoid the situation where an integer with multiple digits starts with 0. How to rewrite the production to avoid the situation?

$$\begin{aligned} Digit &\rightarrow 0 \mid \dots \mid 9 \\ nonzeroDigit &\rightarrow 1 \mid \dots \mid 9 \\ Integer &\rightarrow nonzeroDigit\{Digit\} \end{aligned}$$

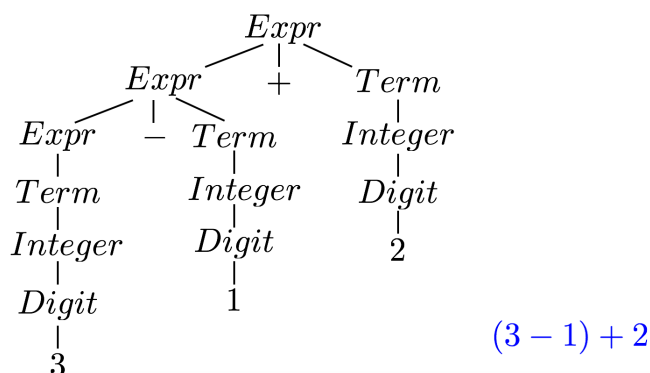
or

$$\begin{aligned} zero &\rightarrow 0 \\ nonzeroDigit &\rightarrow 1 \mid \dots \mid 9 \\ Integer &\rightarrow zero \mid nonzeroDigit\{zero \mid nonzeroDigit\} \end{aligned}$$

## Different Parse Trees Lead to Different Expressions

- The order of non-terminals on the right side is important to generating parse trees.
- Example 1:** Draw a parse tree for  $3 - 1 + 2$  using the following rules

$$\begin{aligned} Expr &\rightarrow Expr + Term \mid Expr - Term \mid Term \\ Term &\rightarrow Integer \\ Integer &\rightarrow Digit\{Digit\} \\ Digit &\rightarrow 0 \mid \dots \mid 9 \end{aligned}$$



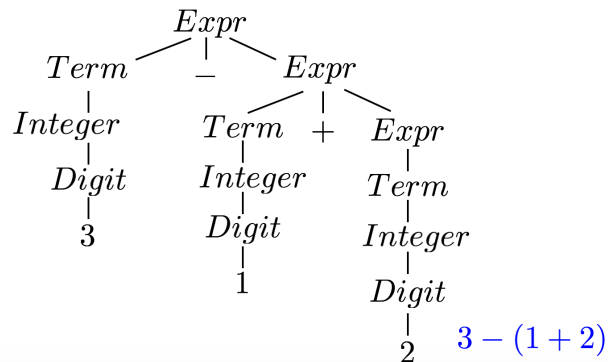
- **Example 2:** Draw a parse tree for  $3 - 1 + 2$  using the following rules

$$Expr \rightarrow Term + Expr \mid Term - Expr \mid Term$$

$$Term \rightarrow Integer$$

$$Integer \rightarrow Digit\{Digit\}$$

$$Digit \rightarrow 0 \mid \dots \mid 9$$



- In the first  $3 - 1 + 2$  example, the result is interpreted as  $(3 - 1) + 2$  (+ is higher than -). In the second  $3 - 1 + 3$  example, the result is interpreted as  $3 - (1 + 2)$ . (different associativities indicated by the different placement of parentheses)
  - Maintaining the order of operations is essential in programming languages.
  - In order to specify hierarchies of operations, grammars can become extremely complex
-

---

## Big Endian vs. Little Endian (for project 1)

- Memory can be thought as one large array containing bytes
- Use “address” to refer to the array location
- Each address stores one element, which is typically one byte (byte-addressable)
- To store 32-bit integer, like 5, we need 4 bytes so 4 slots of RAM array of bytes. Should the 00000101 byte be at the first or fourth byte of the 4 bytes?
- Big Endian: stores the leftmost significant byte in the lowest numerical byte address, so 00000101 is in the 4th byte
- Little Endian: stores the rightmost significant byte in the lowest numerical byte address, so 00000101 is in the 1st byte

Integer: 0x0A0B0C0D:

Big Endian

| Address | 100 | 101 | 102 | 103 |
|---------|-----|-----|-----|-----|
| Value   | 0A  | 0B  | 0C  | 0D  |

Little Endian

| Address | 100 | 101 | 102 | 103 |
|---------|-----|-----|-----|-----|
| Value   | 0D  | 0C  | 0B  | 0A  |

---

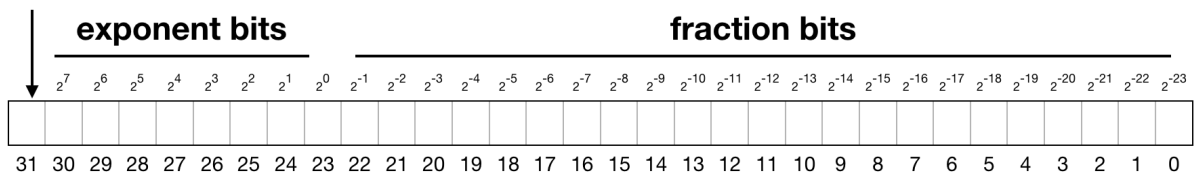
## Floating Point Format (for HW1)

Floating point numbers are represented in the IEEE-754 format. A single precision float is represented by 32 bits. These 32 bits are split up into 3 parts - the sign bit, the exponent, and the fraction. The number being represented is computed according to

$$\pm 2^e \times \text{fraction}$$

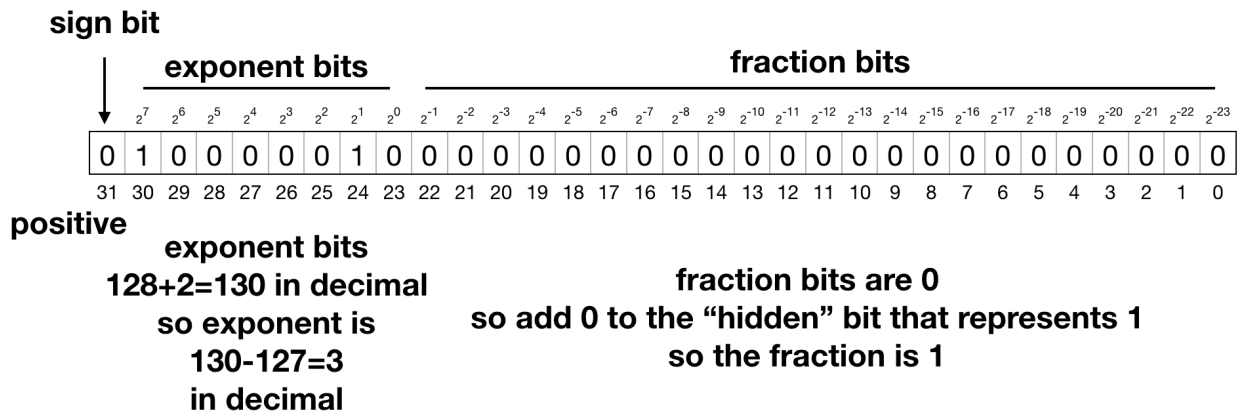
- The leftmost bit (bit 31) is the sign bit (1 indicates negative, 0 indicates positive).
- The next 8 bits (bits 30 through 23) are used as an unsigned 8-bit integer (let's call that `exp_bits`) and contain the exponent.  $e = \text{exp\_bits} - 127$ .
- The remaining 23 bits (bits 22 through 0) represent the fraction, also known as the mantissa or significand. This represents a number between 1 and 2 (it approaches 2). Even though we have only 23 bits, we treat it as a 24-bit number with the first bit always as a 1. That first ("hidden") left-most bit represents  $2^0 = 1$ . The next bit represents  $2^{-1} = 1/2$ , the next is  $2^{-2} = 1/4$ , ..., and the last bit is  $2^{-23}$ . We add the value for all bits with 1's in them. I.e. if the  $2^{-1}$  bit has a 1 in it, then we add 1 (because the hidden bit is always 1) to  $2^{-1} = 1/2$  to get 1.5. The fraction is 1.5.

### sign bit



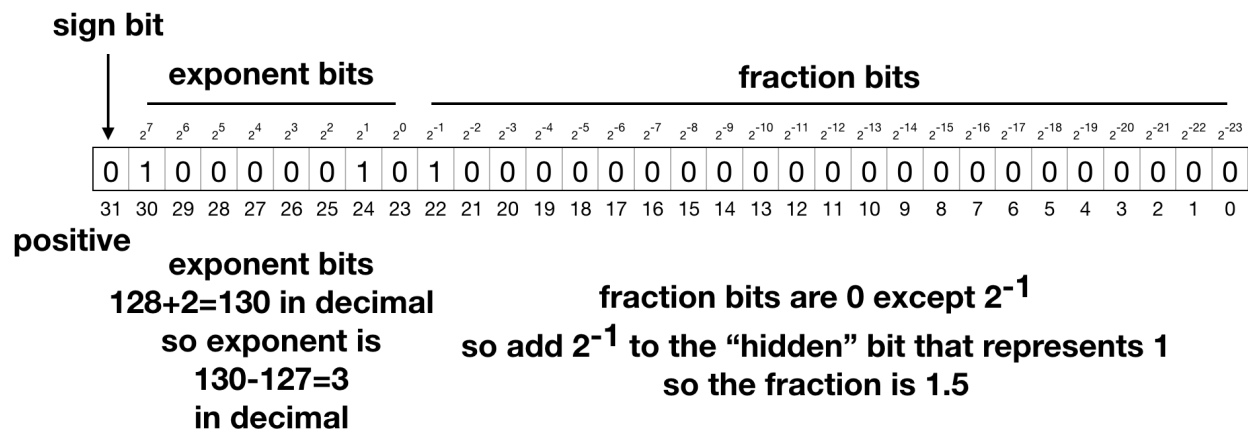
- Check out this website that shows how floating point numbers are represented. You can edit either the decimal representation or the binary representation: <https://www.h-schmidt.net/FloatConverter/IEEE754.html>

- **Example 1:** Look at the floating point representation of 8.



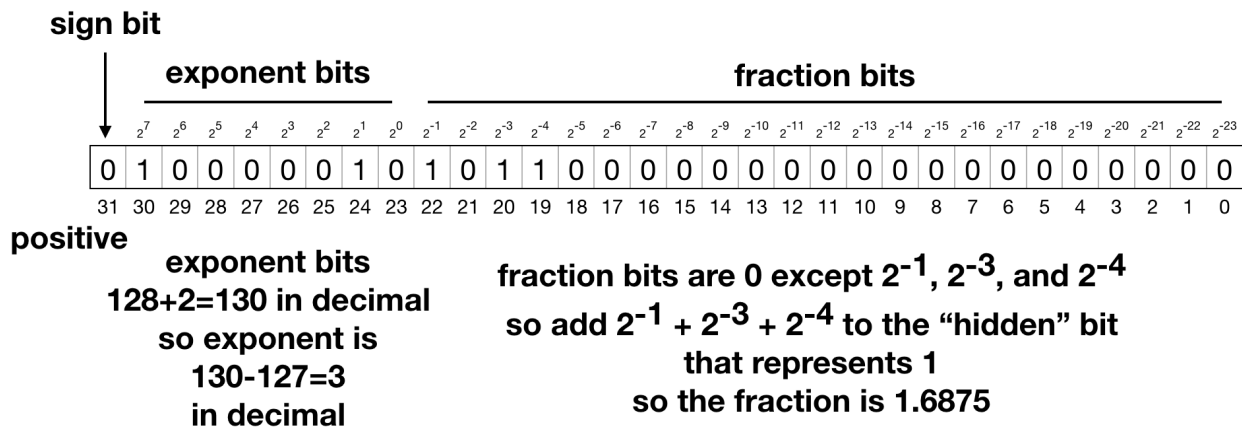
$$2^3 \times 1 = 8$$

- **Example 2:** Look at the floating point representation of 12.0



$$2^3 \times 1.5 = 12$$

- **Example 3:** Look at the floating point representation of 13.5



$$2^3 \times 1.6875 = 13.5$$

