

Syntax (continued)

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

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

Derivation

- To **determine** if a string is **valid** according to a grammar.
- A derivation is **a series of replacements** defined by the productions.
- **Leftmost derivation**: In each step of a derivation, apply production rule to the **leftmost** symbol.
- **Rightmost derivation**: In each step of a derivation, apply production rule to the **rightmost** 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**.
- Bottom-up approach
 - Replace all terminal symbols with nonterminal symbols and the 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).
- **Example1**: Given the formal definition of integer, check whether 312 is a valid integer using top-down approach.

$$Integer \rightarrow Digit \mid Digit\ Integer$$

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

- Topdown approach

$$\begin{aligned} Integer &\Rightarrow DigitInteger \\ &\Rightarrow 3Integer \\ &\Rightarrow 3DigitInteger \\ &\Rightarrow 31Integer \\ &\Rightarrow 31Digit \\ &\Rightarrow 312 \end{aligned}$$

- **Example2:** Given the formal definition of integer, check whether 312 is a valid integer using bottom-up approach.

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

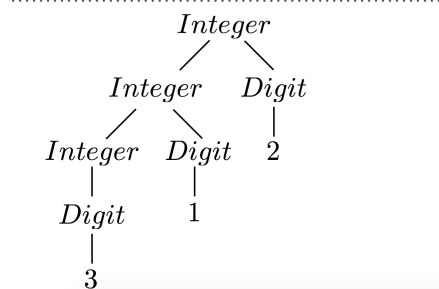
- Bottom-up approach

$$\begin{aligned} 312 &\Rightarrow Digit12 \\ &\Rightarrow Integer12 \\ &\Rightarrow IntegerDigit2 \\ &\Rightarrow Integer2 \\ &\Rightarrow IntegerDigit \\ &\Rightarrow Integer \end{aligned}$$

Parse Tree

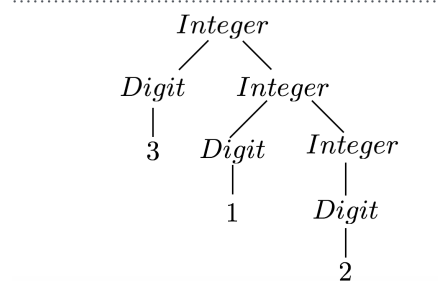
- A [graphical form](#) of derivation. The root of the tree (drawn at the top) is the start symbol.
- Each derivation step corresponds to a new subtree. The parent of each subtree is a nonterminal and its children are the non-terminals and terminals for the right-hand side of the appropriate rule (and 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]

$$Integer \rightarrow Digit | Integer Digit$$

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


- 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.
- Example: Draw the parse tree for 312 by by using the following rule [bottom-up, rightmost]

$$Integer \rightarrow Digit | Digit Integer$$

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


- The above derivation is representative of a **LR(k)** parser that starts with the leftmost leaf of the parse tree.
- Notice that the 3, 1, and 2 appear in the correct order (as we read the leaves from left to right), and that if we read the tree from top to bottom, the rightmost digit is resolved “first” (it is higher up in the tree) (and that is why we refer to this as a rightmost derivation even though we went left to right when we constructed it). Also notice how the left-recursion appears as the chain of Integer non-terminals going from the root via the leftmost children.
- For shortish examples like this, if I need to draw the parse tree using a grammar with left-recursion, I just read the string from right to left and build it top-down. This isn’t how I would program it, but it is how I do it manually (again, only for small examples).

- In both top-down and bottom-up derivations, the parser has to be smart about selecting the appropriate production rule. When you are doing it by hand, you can basically scan ahead to make sure you are using the rule that will allow you to finish the derivation (i.e. if there is a + sign, then use the E+T rule rather than the T rule).
- But in these examples we have no reason to prefer one parse tree for another (well, that isn't quite true because we may prefer the top-down approach to the bottom-up approach, but I mean that the end-product doesn't make us have a preference).
- Let's look at parse trees that aren't just part of the number-parsing. Let's look in a more complicated context - parsing mathematical expressions. Then we will care which parse tree we have.

Different Parse Trees Lead to Different Expressions

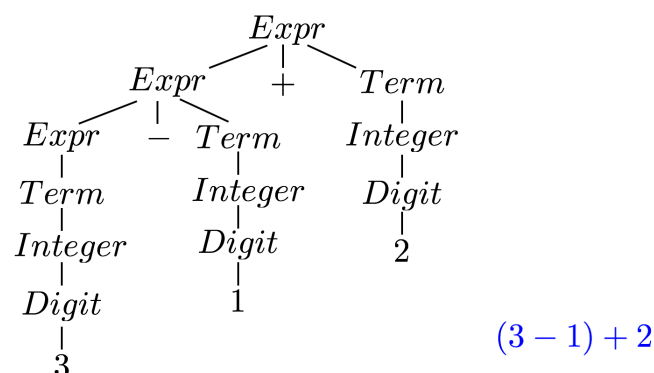
- The order of non-terminals on the right side is important to generating parse trees.
- **Example 1:** Draw a parse tree for $3 - 1 + 2$ using the following rules

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

$$Term \rightarrow Integer$$

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

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



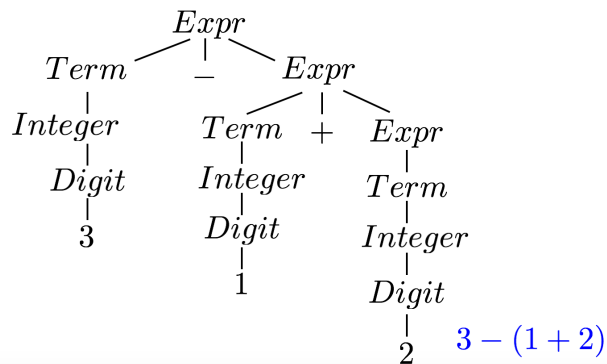
This is a grammar written with left-recursion, and, if I were to build the tree by hand, I would scan from right to left and build it top-down.

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

$$Expr \rightarrow Term + Expr | Term - Expr | Term$$

$$Term \rightarrow Integer$$

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

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


- 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

Addresss	100	101	102	103
Value	0A	0B	0C	0D

Little Endian

Addresss	100	101	102	103
Value	0D	0C	0B	0A

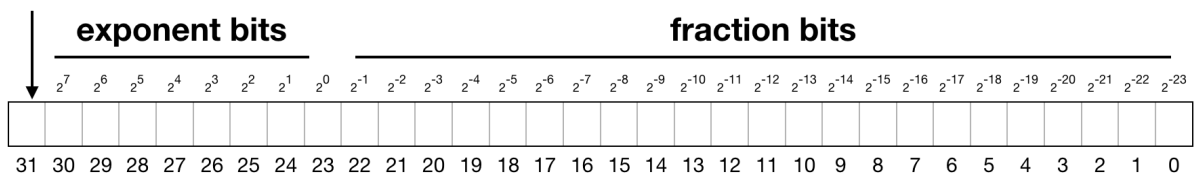
Floating Point Format (for HW1)

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

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

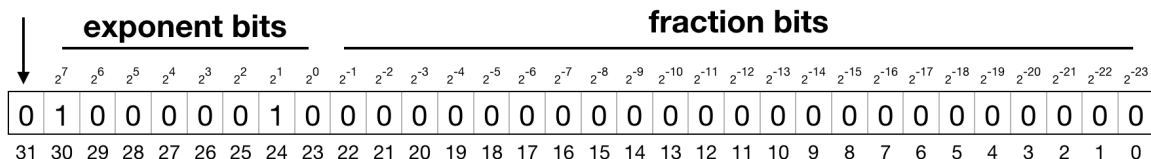
- The leftmost bit (bit 31) is the sign bit (1 indicates negative, 0 indicates positive).
- The next 8 bits (bits 30 through 23) are used as an unsigned 8-bit integer (let's call that `exp_bits`) and contain the exponent. $e = \text{exp_bits} - 127$.
- The remaining 23 bits (bits 22 through 0) represent 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 8.0

sign bit



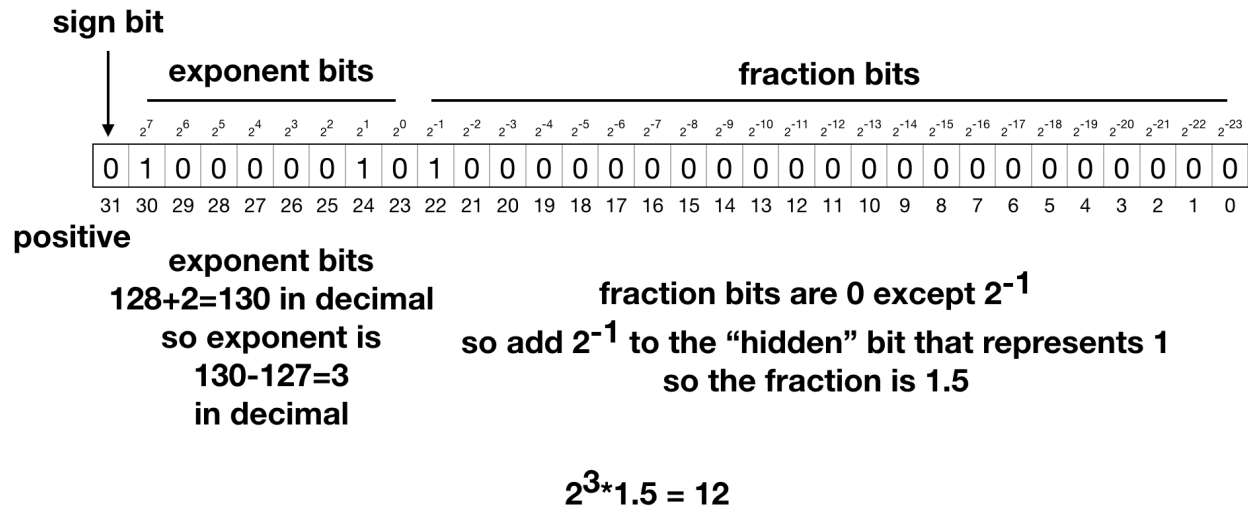
positive

exponent bits
 $128 + 2 = 130$ in decimal
 so exponent is
 $130 - 127 = 3$
 in decimal

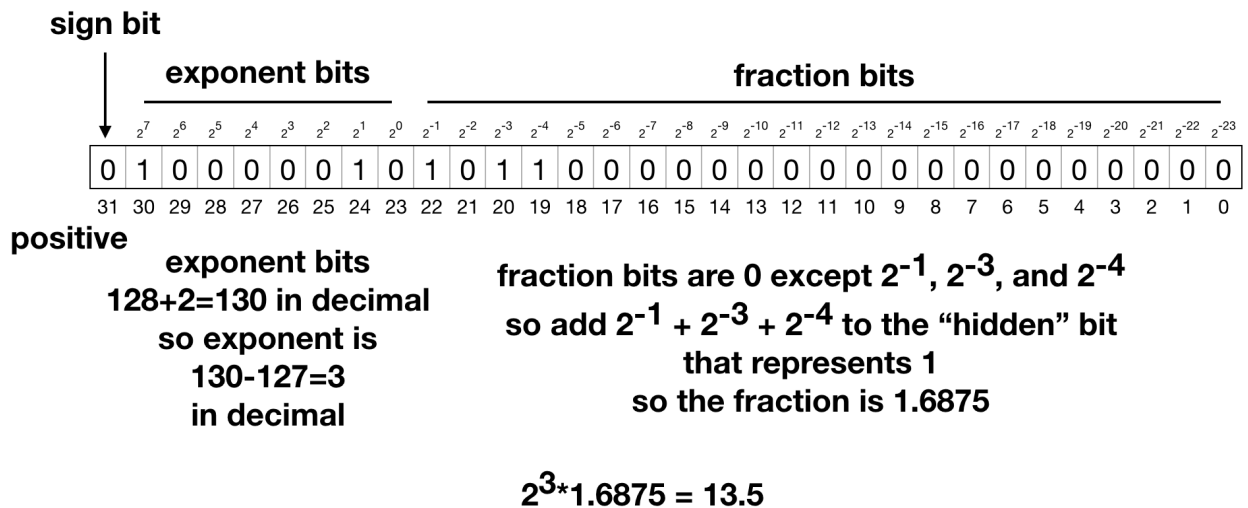
fraction bits are 0
 so add 0 to the "hidden" bit that represents 1
 so the fraction is 1

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

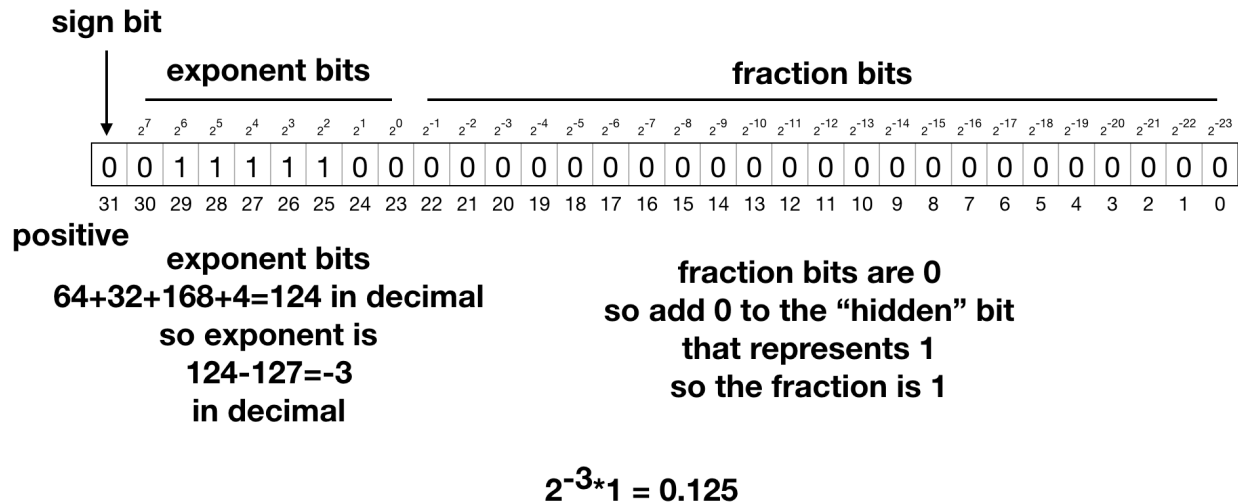
- **Example 2:** Look at the floating point representation of 12.0



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



- **Example 4:** Look at the floating point representation of 0.125 (i.e. $1/8$)



- Very small numbers (2-127) can be represented, as can very large numbers (2128).
- But when we perform arithmetic with them, we are constrained. What happens when we add two numbers of different orders of magnitude (i.e. different exponents)? Think about how you would do that by hand - you would represent both numbers with the same exponent. And, if you have a fixed number of digits to represent the number, you would lose information when you represent your smaller number with the larger exponent. The computer is forced to follow a similar process, so we lose information - we call this round-off error.
- There is a trade-off between the range of the numbers that can be represented (more bits in the exponent gives us a wider range) and the precision (more bits in the fraction allows us to be more precise in representing the number).