CS16 Week 2 Part 1

Kyle Dewey

Overview

- Numeral Systems
- Doubles (`double`)
- Characters (`char` **and** `char*`)
- `printf`
- Arithmetic
- Expressions

Last Time...

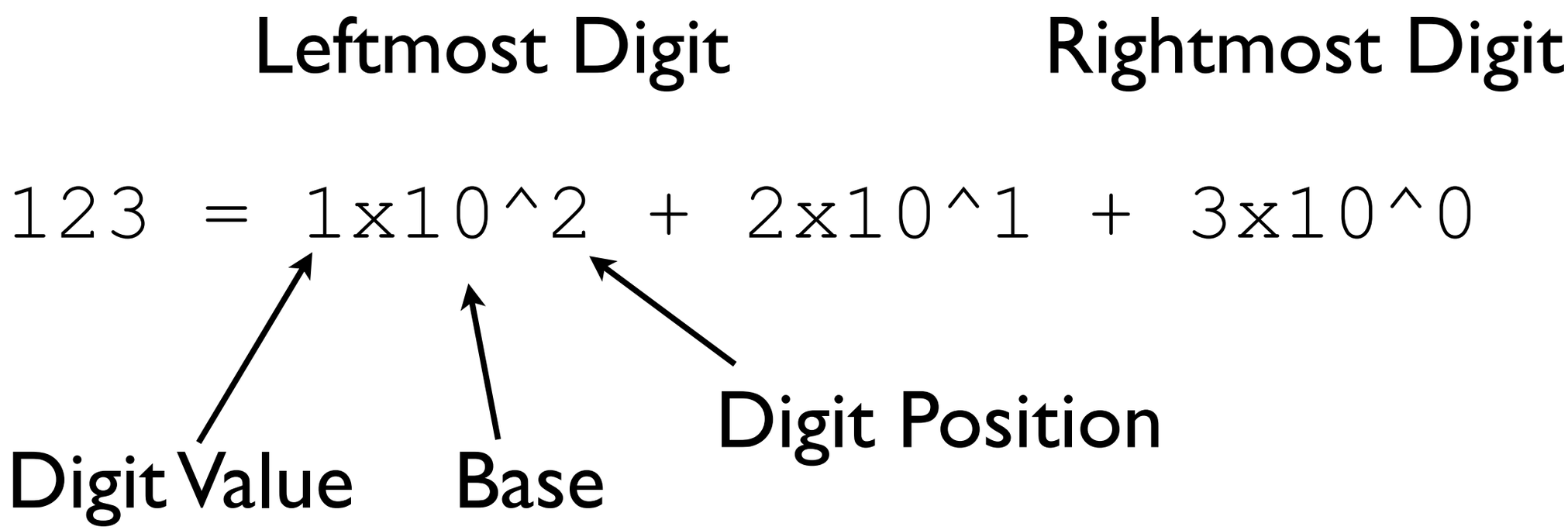
- Data stored in binary digits (bits)
- Different data types use different numbers of bits
- The more bits there are, the more things that can be stored

Numeral Systems

- May have heard that a number is in base X, where X is some integer
- The base roughly states how many possibilities are in one digit
- Humans work in base 10 (0-9)
- Computers work in base 2 (0,1)

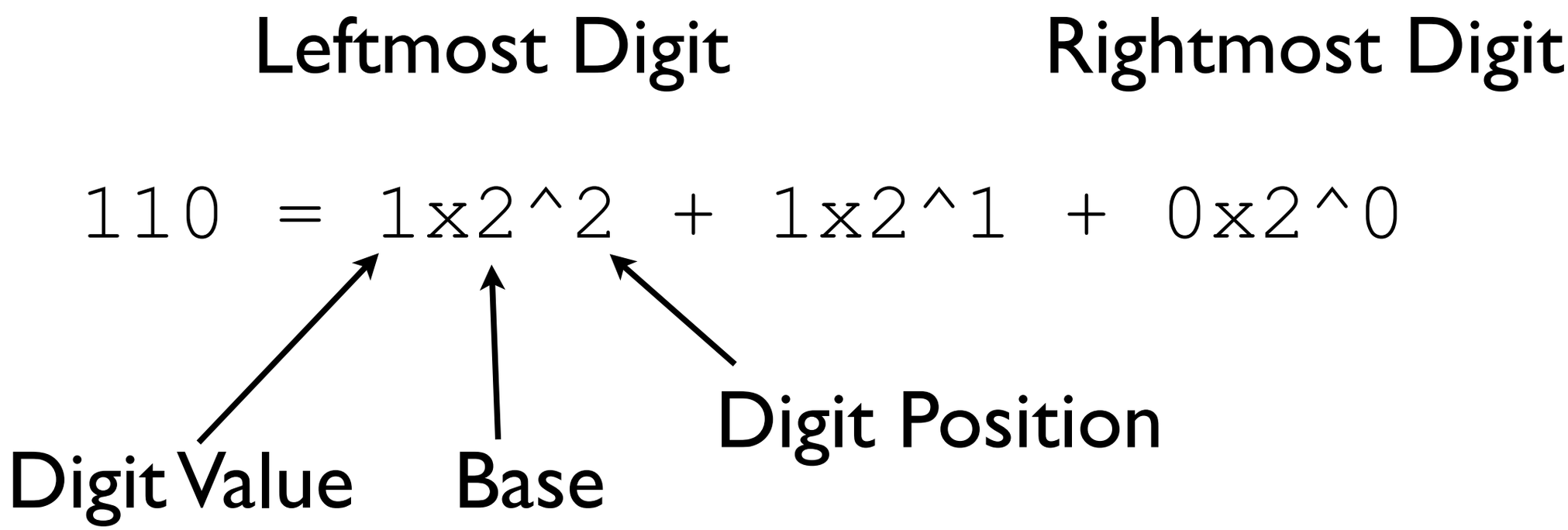
Numeral Systems

- How do we understand what number "123" is in base 10?



Numeral Systems

- Same rule applies to other bases
- What about "110" in base 2?



Why Binary?

- Easy at the circuitry level
- Either on or off
- Merely a different representation of the same thing

Problem

- Recall: `chars` are of 8 bits (1 byte)
- Intended to store characters, but they can also store small integers
- What does `0` look like in an `unsigned char`?

Answer

- Recall: `chars` are of 8 bits (1 byte)
- What does `0` look like in an `unsigned char` in binary?

```
00000000
```

Problem

- Recall: `chars` are of 8 bits (1 byte)
- What is the maximal value of this?
- What does this maximal value look like in binary?

Answer

- Recall: `chars` are of 8 bits (1 byte)
- What is the maximal value of this?
- What does this maximal value look like in binary?

```
11111111
```

Problem

- Recall: `char`s are of 8 bits (1 byte)
- What is the middlemost value?

Answer

- Recall: `char`s are of 8 bits (1 byte)
- What is the middlemost value?

```
10000000
```

Problem

- Recall: `chars` are of 8 bits (1 byte)

```
unsigned char x = 255;
x = x + 1;
// what is x?
```

Overflow

- When there is not enough room to store the result
- Data ends up being truncated

```
  11111111
+        1
----------
 100000000
```

Size of Data Type

Problem

- Recall: `chars` are of 8 bits (1 byte)

```
unsigned char x = 0;
x = x - 1;
// what is x?
```

Underflow

- Same sort of problem, but in the opposite direction
- They always loop around to the greatest value in the opposite direction

```
 00000000
-        1
---------
 11111111
```

Hexadecimal

- Base 16
- Digits: 0-9 and A-F
- Much more compact than binary
  - Only two hexadecimal digits per byte (four bits apiece)
- Typically start with `0x` by convention, as with `0x3A`

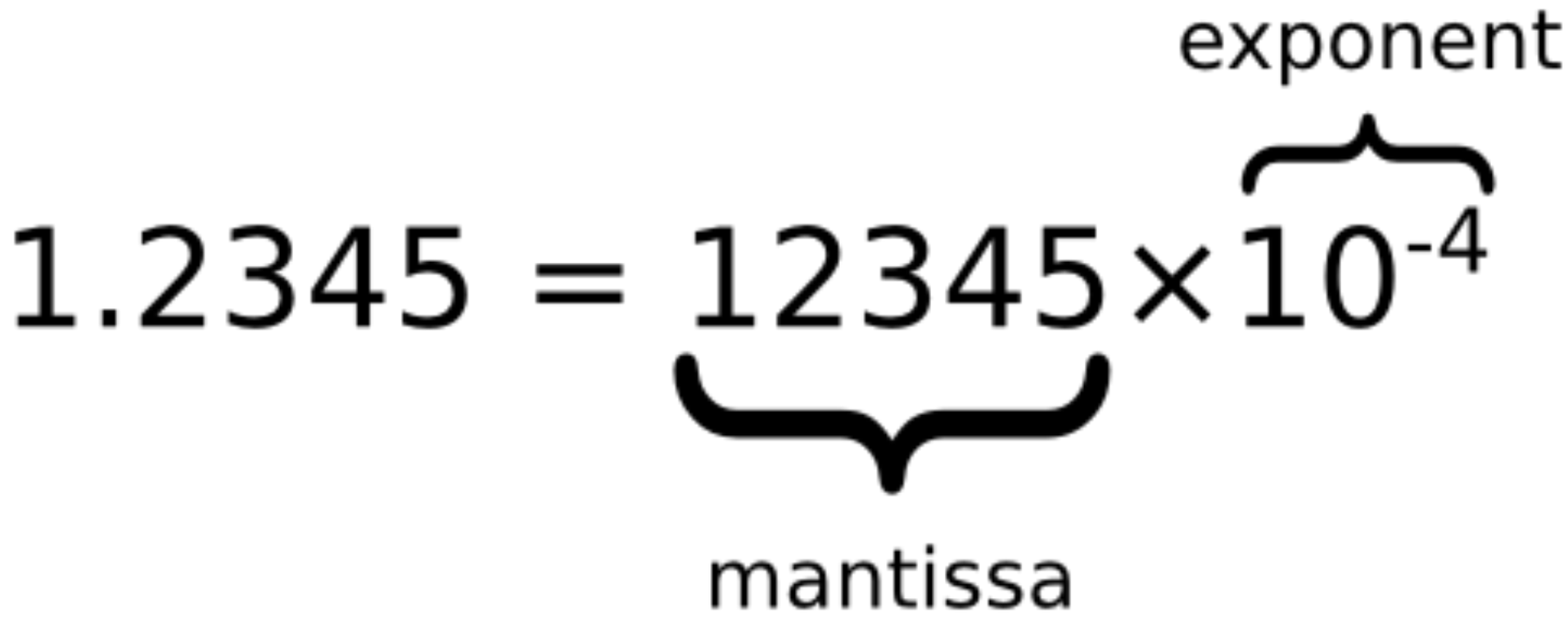
Hexadecimal

```
0: 0000
1: 0001
2: 0010
3: 0011
4: 0100
5: 0101
6: 0110
7: 0111
8: 1000
9: 1001
A: 1010
B: 1011
C: 1100
D: 1101
E: 1110
F: 1111
```

`128` **(base 10) =** `10000000` **(base 2) =** `80` **(base 16)**

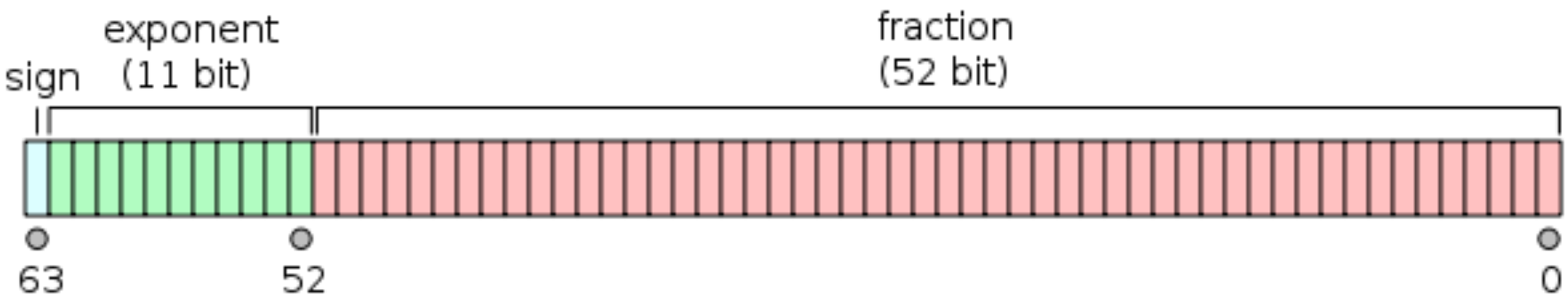
`double` Representation

- Internally, scientific notation is used



`double` Representation

- Fraction is essentially mantissa
- 15 significant digits
- Many more (not so important) details



Question

- Is `double` an appropriate data type for representing currency?
- Why / why not?

Answer

- No
- Using base 2 to represent a base 10 number
- The significant digits may be correct for a single calculation, but this is not the only part!

Answer

- `1.03 - 0.42 =`
  `0.6100000000000001`

- Slight error can add up as more calculations are performed

Characters

`char`

- As already stated, these can hold small integers too
- Intended to hold individual characters

Question

- How does the machine "know" if you mean an integer or a character?

Answer

- All about context

```
char c = 5;
c = c + 2;
```

```
char c = 'c';
```

So what are they?

- Everything ends up being a binary bit pattern
- Established mappings of different bit patterns to characters
- Other stuff deals with this translation

ASCII

- One such popular mapping
- All characters are 8 bits long
- For a long time, the only mapping

ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	Y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Words

- What about words and sentences?
- Need multiple characters
- Brings up a number of issues

Question

- How does C represent words (“strings” of characters)?

Answer

- Sequentially
- Null byte at the end

```
“Hello” = ‘H’,’e’,’l’,’l’,’o’,’\0’
```

Question

- Why the terminating null byte?

Answer

- Where does the string end?
- Everything is just a bunch of bits underneath

```
char* foo = “foo”;
char* bar = “bar”;
```

```
'f','o','o','b','a','r'
```

vs.

```
'f','o','o','\0','b','a','r','\0'
```

Question

- The null byte is just a convention
- There is another convention of varying popularity
- What could it be?

Answer

- Fundamentally, we need to know when the string ends
- Can put this information right in the front

"foobar"

6:'f','o','o','b','a','r'

Question

- What are the pros and cons of each?
- Why choose one over the other?

printf

`printf`

- "Prints" something to the terminal
- The "something" can be:
  - `char`
  - `int`
  - `double` (with a number of formatting options)
  - Strings of `chars`

Simple Example

```
printf( “foobar” );
```

Formatting String

- Takes a formatting string specifying what to print and how to print it
- A mini-language for specifying how to print

Formatting String

- Things beginning with % have special meaning
- Specify a placeholder for something

Placeholders

- %c: Character (`char`)
- %i: Signed decimal integer (`int`)
- %u: Unsigned decimal integer (`unsigned int`)
- %s: Character string (`char*` or `char[]`)

Placeholders

- The things they hold for must be specified afterwards

Examples

```
char myChar = ‘a’;
int mySignedInt = -2;
unsigned int myUnsignedInt = 5;
char* myString = “myString”;
...
printf( “%c”, myChar );
printf( “%i”, mySignedInt );
printf( “%u”, myUnsignedInt );
printf( “%s”, myString );
printf( “%c%i”, myChar, mySignedInt);
printf( “%c %i”, myChar, mySignedInt);
printf( “%s%c”, myString, myChar );
```

Other Bases

- `%x` for hexadecimal
- `ch` supports the special `%b` for a binary representation

What about...

```
int x = 0;
printf( “%i” );
printf( “hi”, x );
```

What about...

- Using `gcc`

```
int x = 0;
printf( “%i” ); // only a warning!
printf( “hi”, x ); // not even a warning
```

What about...

```
int x = -2;
printf( “%u”, x );
```

What about...

```
int x = -2;
printf( “%u”, x );
```

- Ends up printing ‘`4294967295`’
- It’s all just bits - it’s up to you to say what they means

Or perhaps...

```
char* foobar = “foobar”;
printf( “%u”, foobar );
```

Or perhaps...

```
char* foobar = “foobar”;
printf( “%u”, foobar );
```

- Ends up printing wherever foobar is in memory

Or even...

```
unsigned int x = 5;
printf( “%s”, x );
```

Or even...

- `ch` will straight-up not let you do this
- `gcc` will issue a warning at compilation
- At runtime...who knows?
- Will walk memory until it either finds a null byte or it tries to read something it is not allowed to

The Point

- `printf` is not very intelligent
- Relies on the programmer to specify what kind of data it should print out

`printf` **and** `double`

- The placeholder is %f
- Can specify the minimum number of digits to show using . *`number`*

Examples

```
printf( “%f”, 512.5 );
// prints 512.500000

printf( “%.2f”, 512.5 );
// prints 512.50
```

Special Characters

- Include newlines and tabs
- Quotes are another problem, given that the string itself is specified with quotes

Special Characters

- Use backslash (\) for specifying these
- Newline: `\n`
- Tab: `\t`
- Single quote: `\'`
- Double Quote: `\"`
- Backslash: `\\`
- Null byte: `\0`

Question

- What does the following print out?

```
printf( “foo\0bar” );
```

Answer

- By C's convention, strings stop at a null byte (`\0`)
- As far as `printf` is concerned, the string stopped after `foo`

And much, much more

- There are a bunch of additional things that `printf` can do
  - Specify more specific types not covered
  - Specify field length
  - Parameterized field / precision lengths
  - Justification...
- Chances are good that `printf` will already handle your printing needs

Arithmetic Expressions

Arithmetic

- C has a number of basic arithmetic operators
  - Addition: +
  - Subtraction: –
  - Multiplication: *
  - Division: /

Arithmetic

- Modulus (get remainder of an integer divided by an integer): %
  - `5 % 2 = 1`
  - `2 % 5 = 2`
  - `6 % 3 = 0`

Precedences

- The basic arithmetic precedence rules still apply. High to low:
  - *, /, %
  - +, −
- Parenthesis can be added to change precedence of whatever is in the parenthesis

Expressions

- Arithmetic expressions return values
- Anything that returns a value is an expression
  - The importance of this will be discussed later with functions

Expressions

- One final value derived by evaluating subexpressions
- A **recursive** process
  - More detail will be given later in the discussion of recursion

Example

```
3 * 2 + 7 - 8 * 2
```

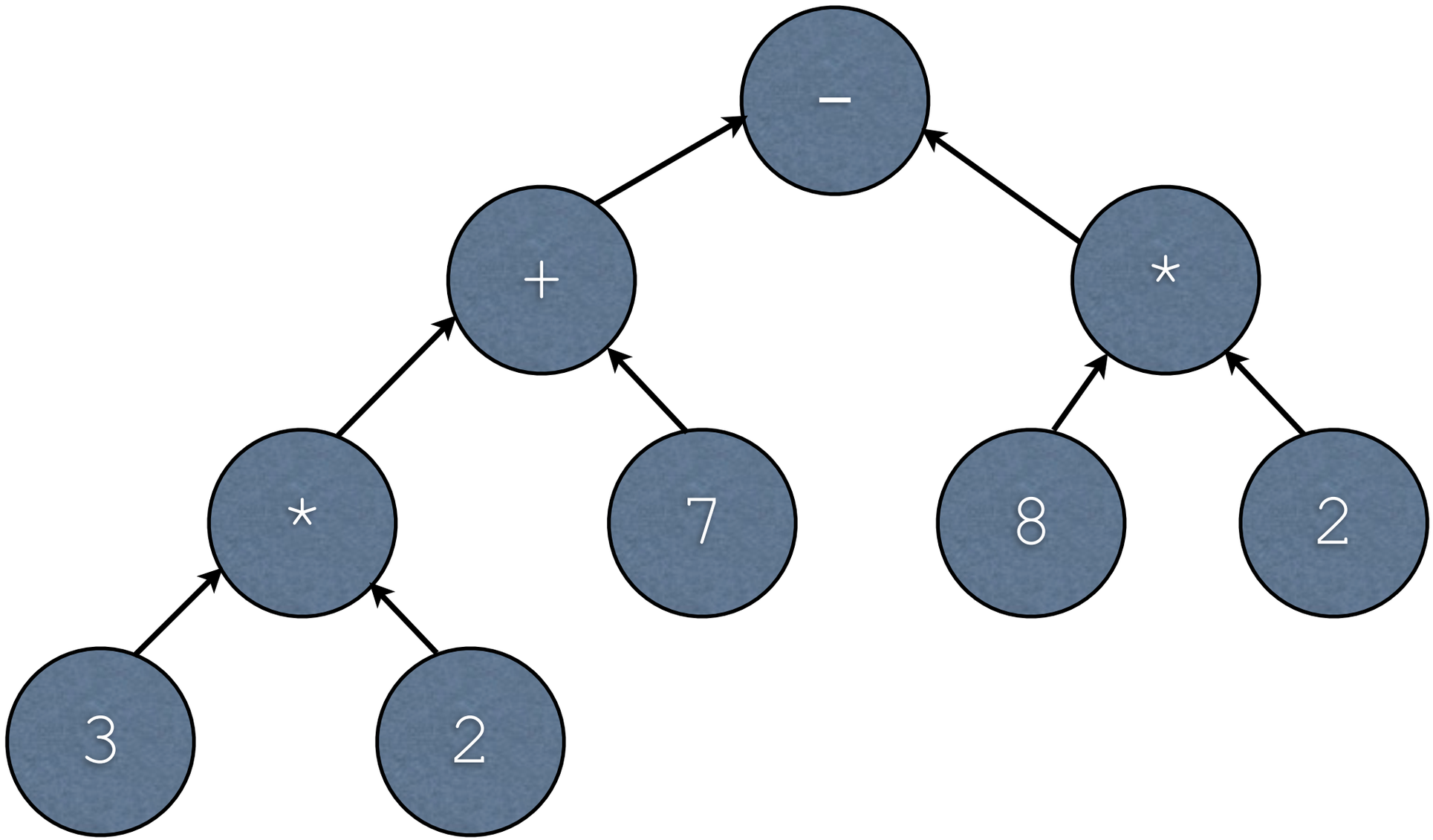
```
(3 * 2) + 7 - (8 * 2)
```

```
((3 * 2) + 7) - (8 * 2)
```

```
(((3 * 2) + 7) - (8 * 2))
```

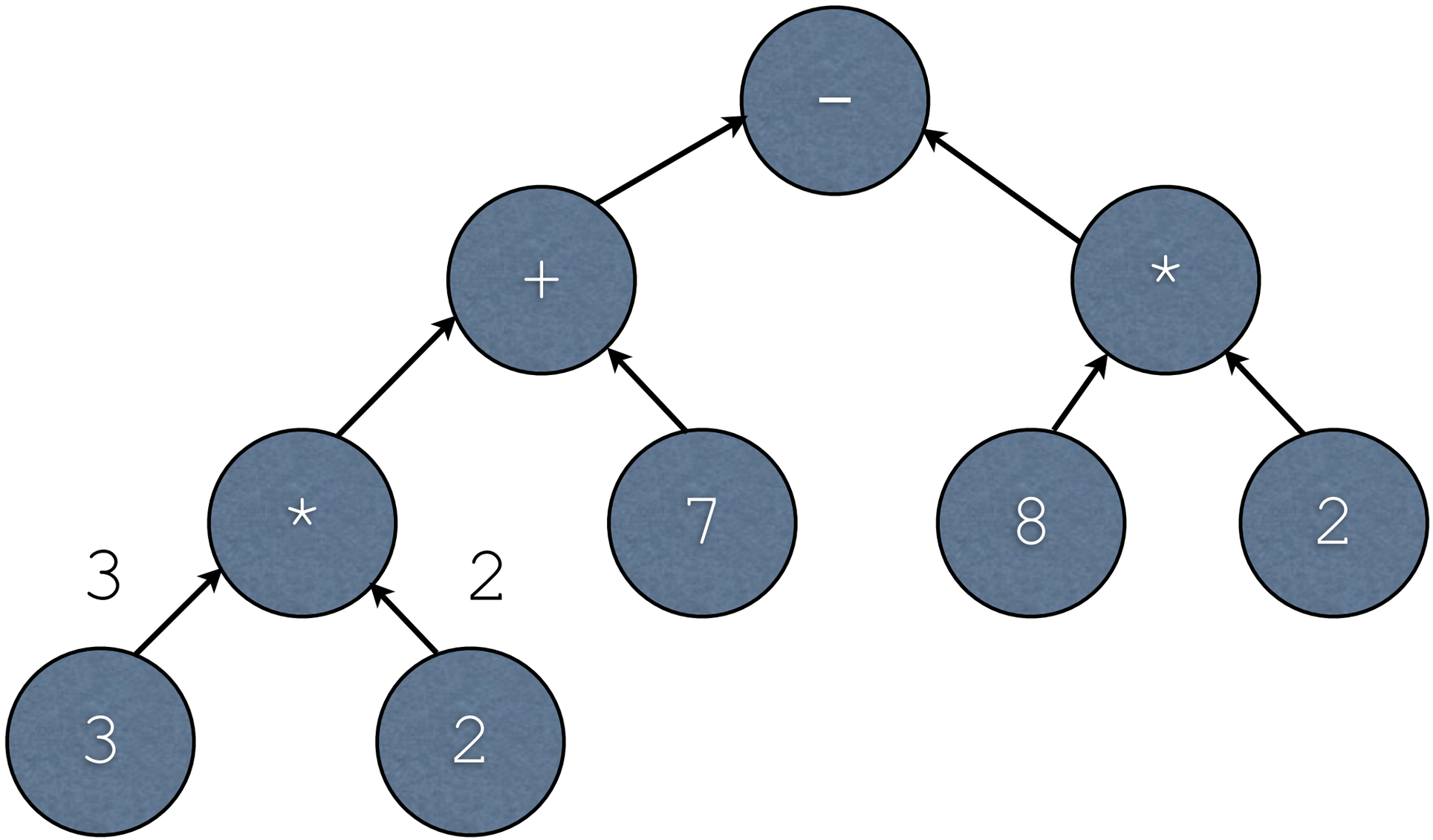
Example

```
((((3 * 2) + 7) - (8 * 2))
```



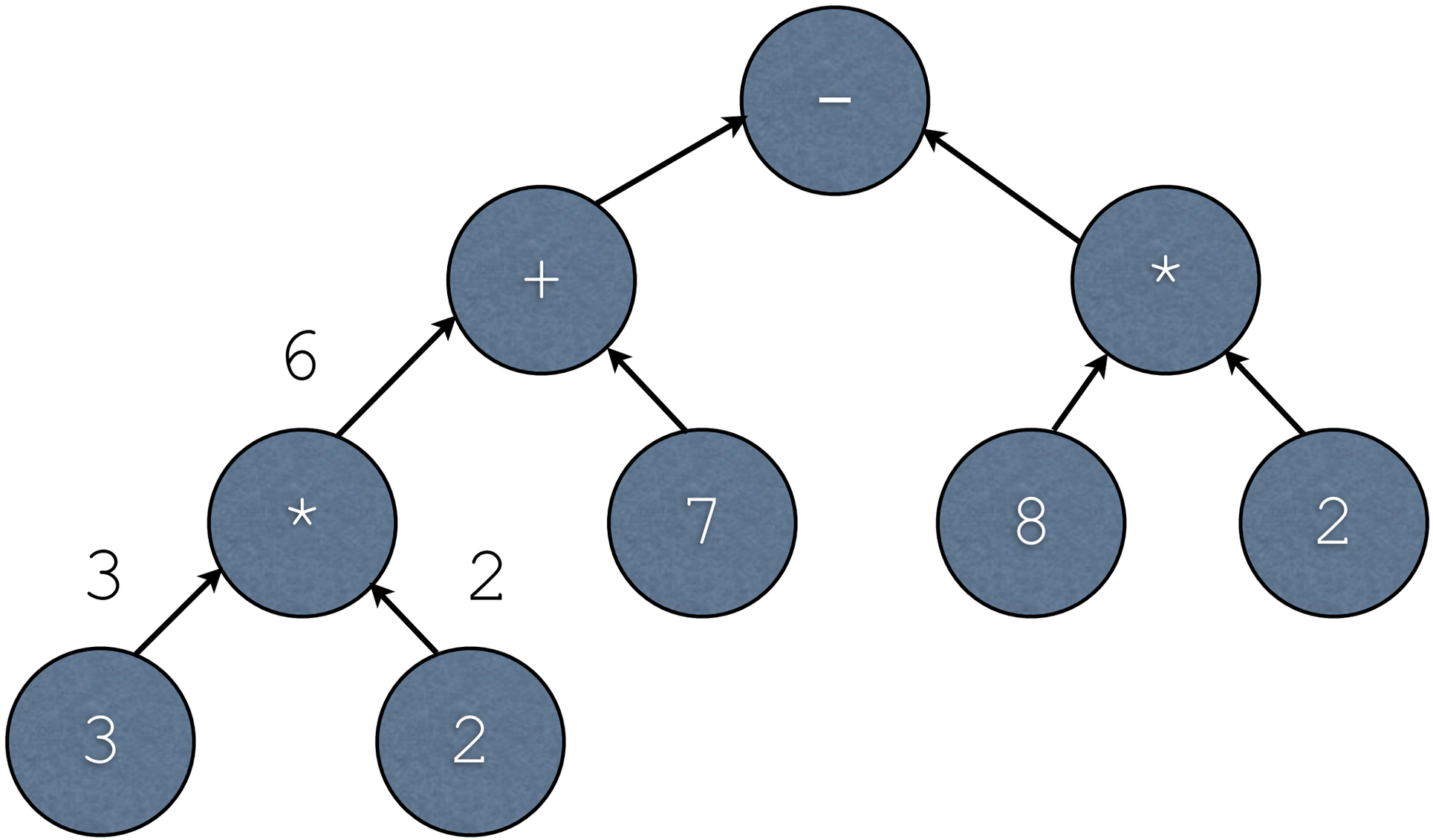
Example

```
((3 * 2) + 7) - (8 * 2))
```



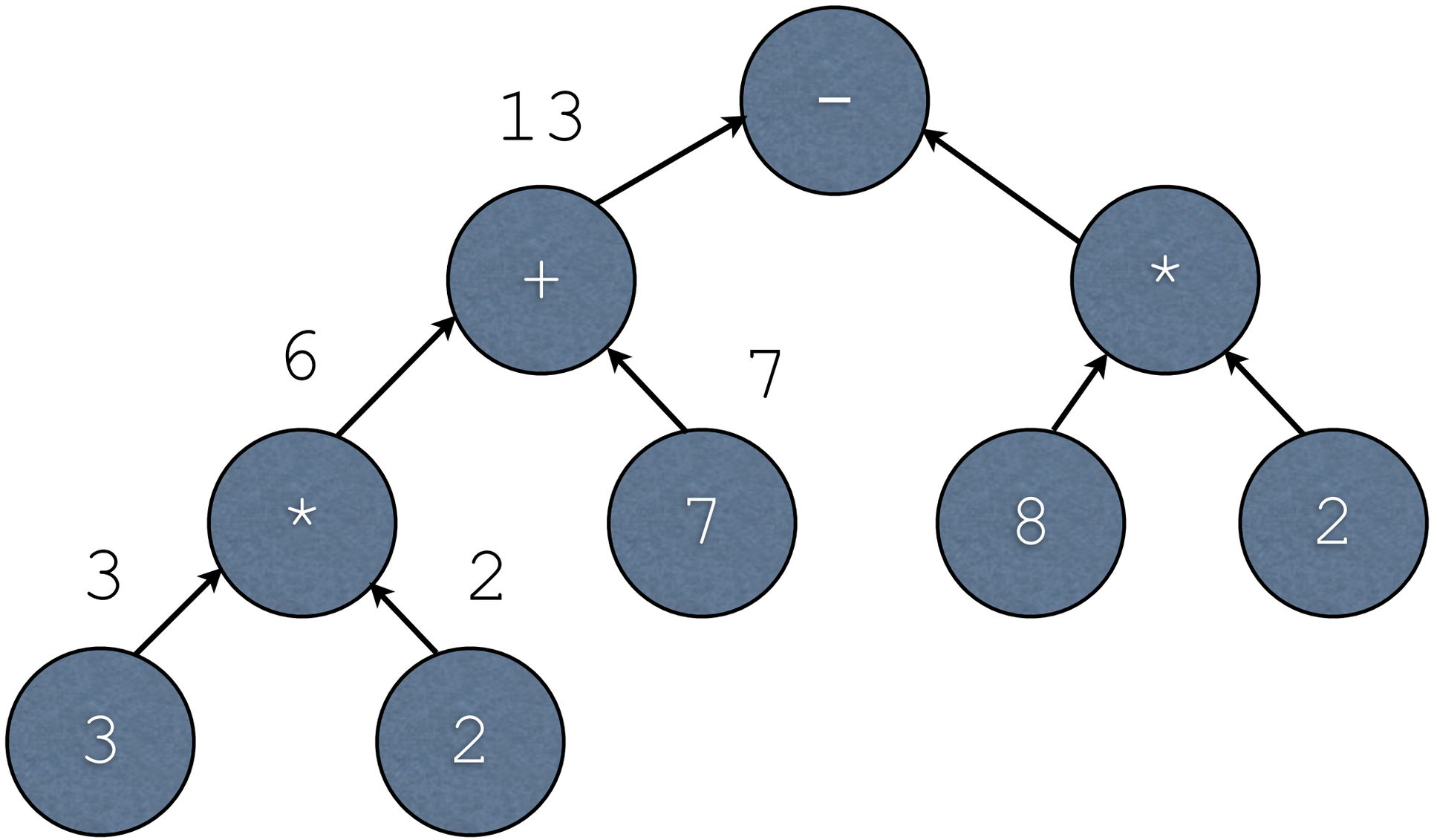
Example

```
((((3 * 2) + 7) - (8 * 2))
```



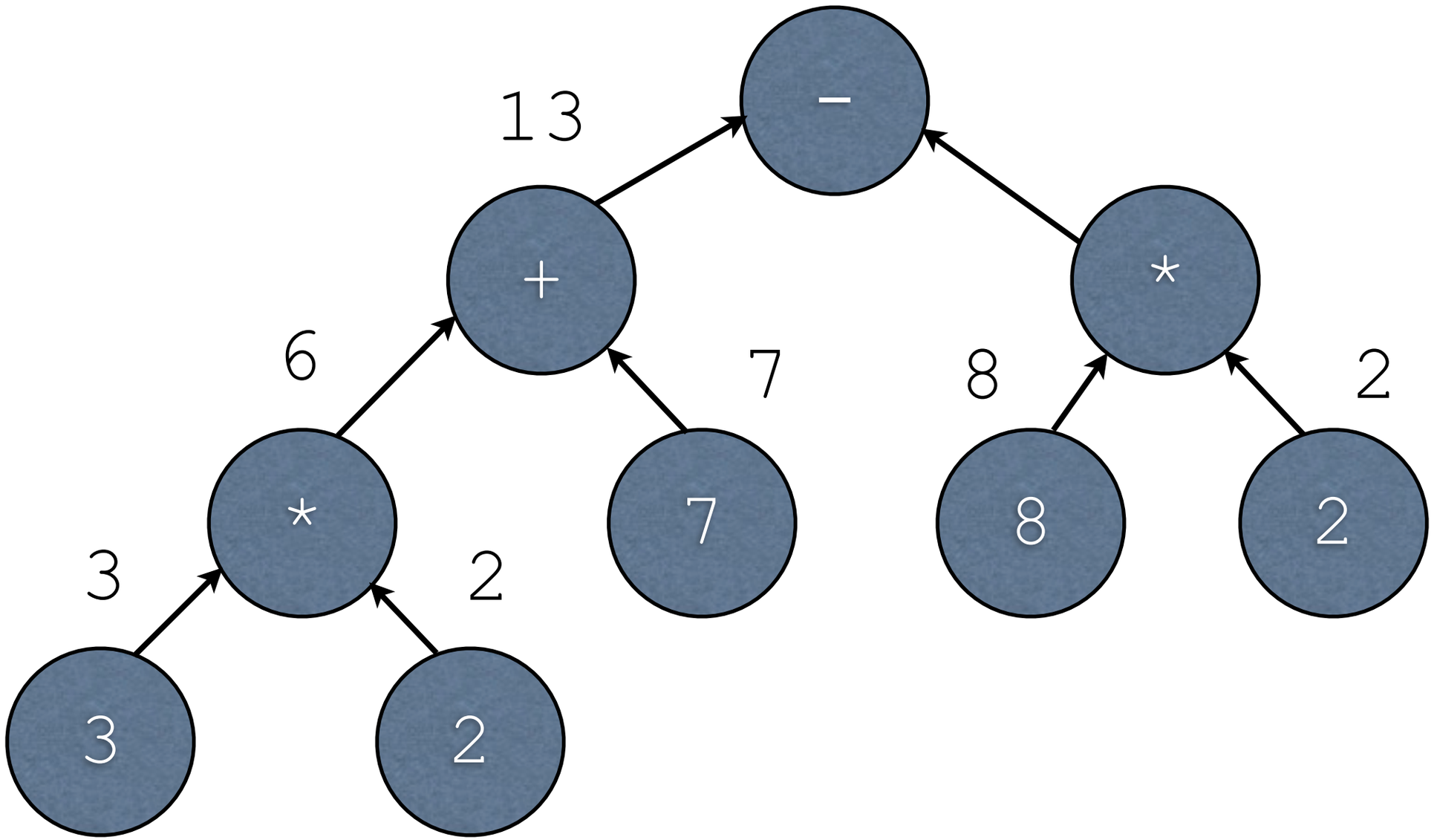
Example

```
((3 * 2) + 7) - (8 * 2))
```



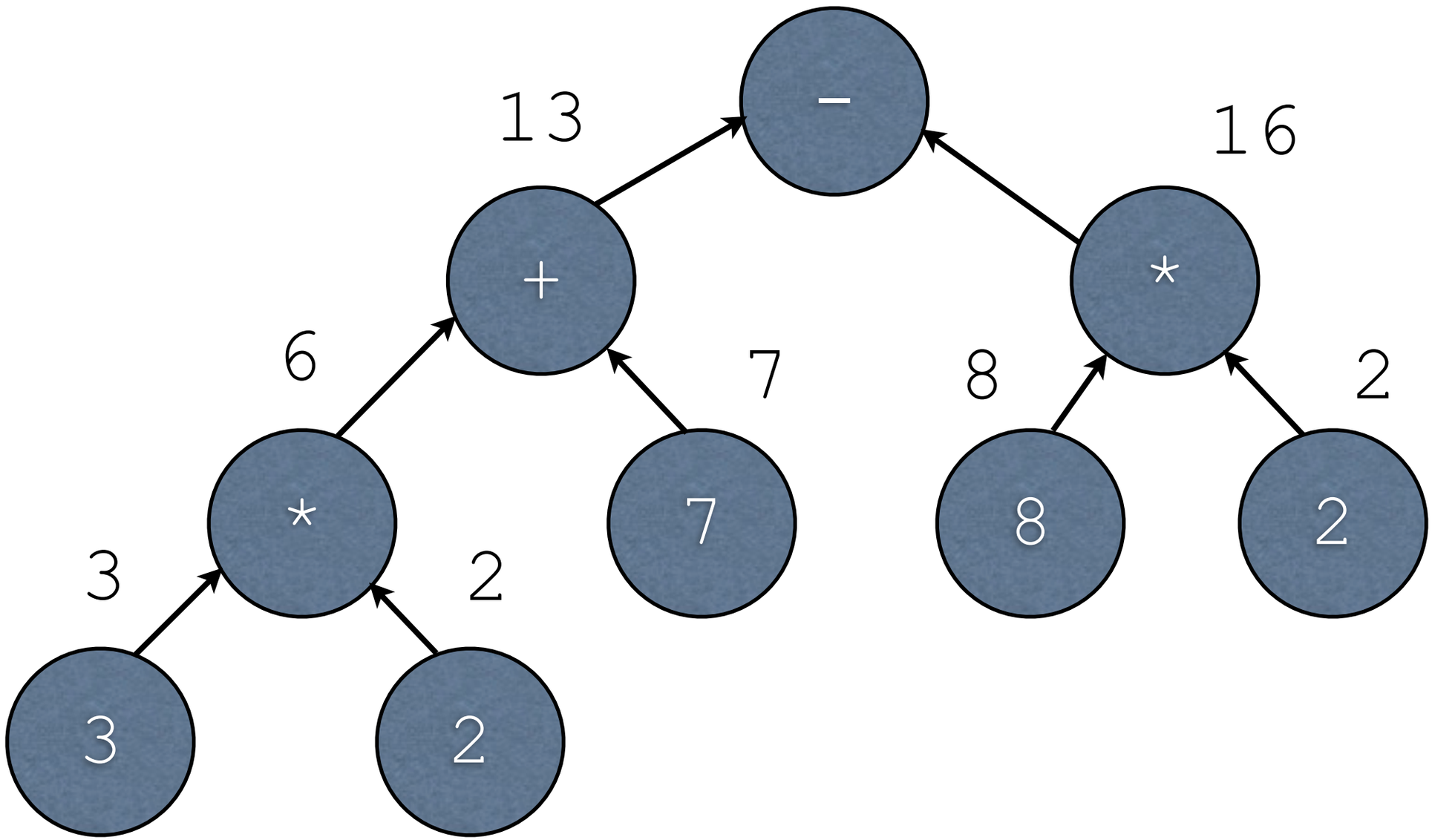
Example

```
((3 * 2) + 7) - (8 * 2))
```



Example

```
(((3 * 2) + 7) - (8 * 2))
```



Example

```
(((3 * 2) + 7) - (8 * 2))
```