Recursion, Datatypes and Arithmetic Expressions

2015/2016 1st Semester

CSIS0259 / COMP3259
Principles of Programming Languages

Goal of this Lecture

- To finish giving you a crash course on the basics of Haskell and Functional Programming:
  - We will learn about recursion and algebraic datatypes
- To build our first interpreter
  - An interpreter for a simple language of arithmetic

Suggested Reading

- Learn You a Haskell for Great Good! (up to Chapter 6)
  - http://learnyouahaskell.com

recommended!

- The Anatomy of Programming Languages
  - Chapter 2

Recursion

Recursion

Haskell programs are often defined using recursion:

```
countElements :: [a] -> Int
countElements []     = 0
countElements (x:xs) = 1 + countElements xs
```

recursive call

Good recursive definitions normally have:

- Base case(s): Cases where the program terminates.
  - For countElements the base case is `[]`.

- Recursive case(s): Cases where a step is taken towards the base cases.
  - For countElements the recursive case is (`x:xs`).
  - Recursive calls are normally done on **smaller** values.

Recursion
Demo

Recursion

There may be programs with bad recursion:

```
badcountElements1 :: [a] -> Int
badcountElements1 []      = 0
badcountElements1 xs      = 1 + badcountElements1 xs
```

why is this bad?

```
badcountElements2 :: [a] -> Int
badcountElements2 (x:xs) = 1 + badcountElements2 xs
```

why is this bad?

Recursion

There may be programs with bad recursion:

recursive call is not smaller!

```
badcountElements1 :: [a] -> Int
badcountElements1 []      = 0
badcountElements1 xs      = 1 + badcountElements1 xs
```

no base case!

```
badcountElements2 :: [a] -> Int
badcountElements2 (x:xs) = 1 + badcountElements2 xs
```

Datatypes

Datatypes

- How are Haskell lists defined? Conceptually they correspond to the following datatype:

```
data [a] = [] | a : [a] — pseudo-code
```

- Note: Haskell lists are actually built-in. The above definition is just for illustration purposes.

User-Defined Datatypes

- Haskell supports user-defined datatypes:

recursive definition

```
data ListInt = Nil | Cons Int ListInt
```

- New datatype definitions support their own pattern matching notation.

```
headListInt :: ListInt -> Int
headListInt Nil = error "Empty list!"
headListInt (Cons x xs) = x
```

Case Analysis

- Haskell also supports case analysis
  - Case analysis is an alternative way to use pattern matching
  - It is sometimes useful when we need to do something before we do pattern matching

Case Analysis

- Example: Defining a hasPositives function

positives :: [Int] -> [Int] — uses regular pattern matching
positives [] = []
positives (x:xs) = if (x > 0) then x : positives xs else positives xs

hasPositives :: [Int] -> Bool — uses case analysis
hasPositives xs =
  case positives xs of — we first need to call positives
    [] -> False
    (x:xs) -> True

Showing and Equality

- Some operations are useful for various datatypes
    - Examples: equality and conversion to a string
- Haskell provides an easy mechanism for supporting these operations:

```
data ListInt = Nil | Cons Int ListInt
                 deriving (Eq,Show)
```

Showing and Equality

- Using equality and show:

```
test :: ListInt -> ListInt -> String
test l1 l2 = if (l1 == l2) then
                 show l1
               else
                 “Not equal!”
```

Maybe

- An important concern in this course will be dealing with failure: interpreters may fail for some expressions.

- The Maybe type (and also the Either type) will be helpful

```
data Maybe a = Nothing | Just a
```

Demo

Expressions, Syntax and Evaluation

Resources

- Chapter 2 of “Anatomy of Programming Languages”

http://www.cs.utexas.edu/~wcook/anatomy/anatomy.htm

Programming Languages

- Wikipedia definition:
  - A programming language is an artificial language designed to communicate instructions to a machine, particularly a computer.
- A more abstract definition:
  - A (programming) language is a mechanism for expressing manipulations over certain types of values.

Language of Arithmetic

- Values: numbers
    - 1,2,3,10,100,6893,…
- Arithmetic expressions: using the language of arithmetic to denote some value
    - 2+3, 3 * 4 - 10, 4 - (-3)
- Values are not expressions!
- Multiple expressions can have the same value!

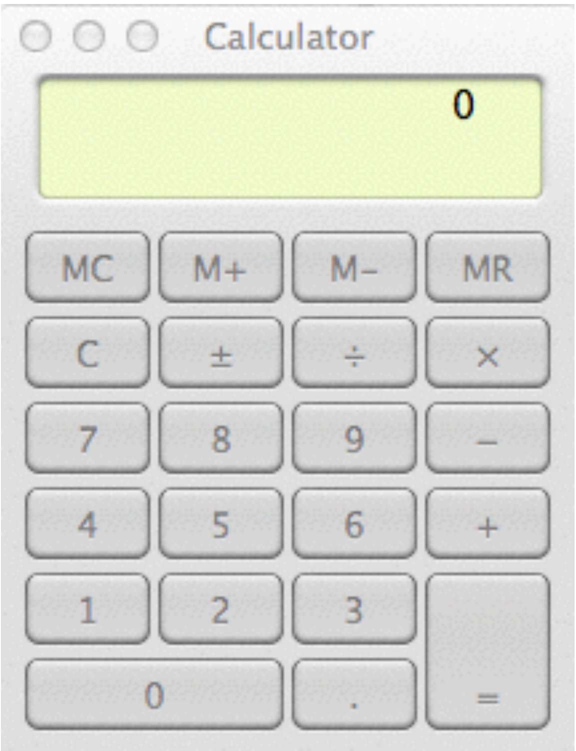
Expressions, Syntax and Evaluation

Three fundamental concepts:

- Expression: A combination of atomic components such as variables, values and operations over these values.
- Syntax: the syntax of an expression prescribes how the various components of the rules can be combined.
- Evaluation: gives the meaning of an expression in terms of a value.

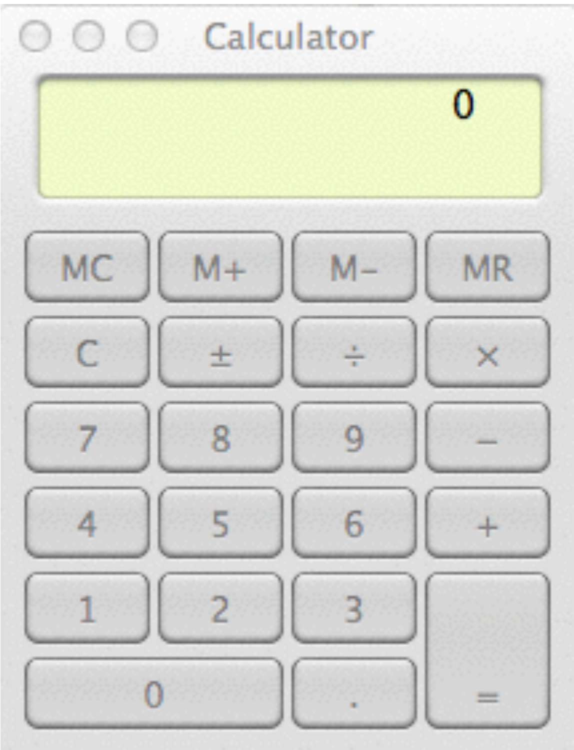
Representing Arithmetic Expressions

- Suppose that you are asked to implement a calculator in Haskell, what would you do?



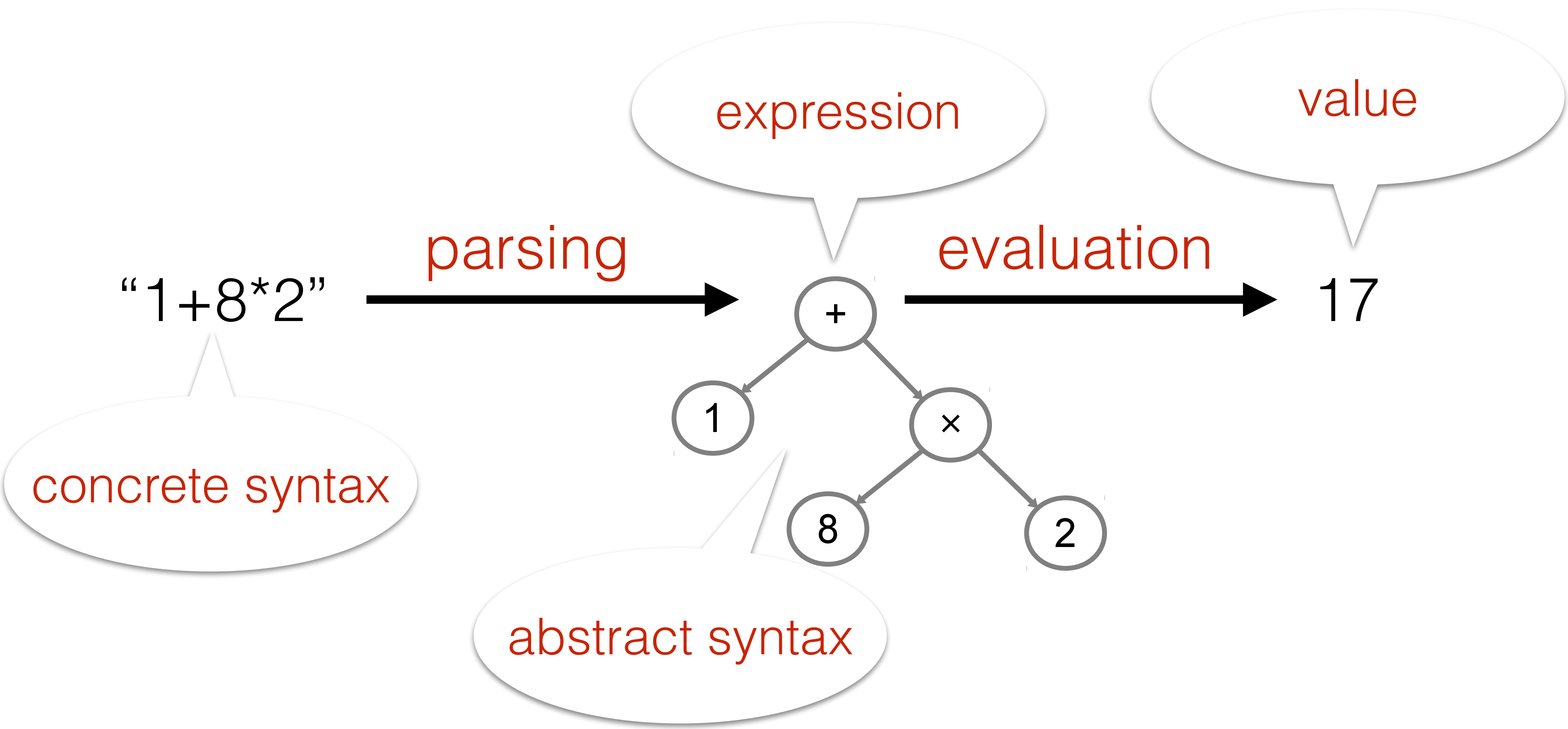
Representing Arithmetic Expressions

- Lets focus on two subproblems:
  - What would be a good representation of arithmetic expressions in Haskell?
  - How to evaluate arithmetic expressions?



Overview

An overview of the “architecture”:

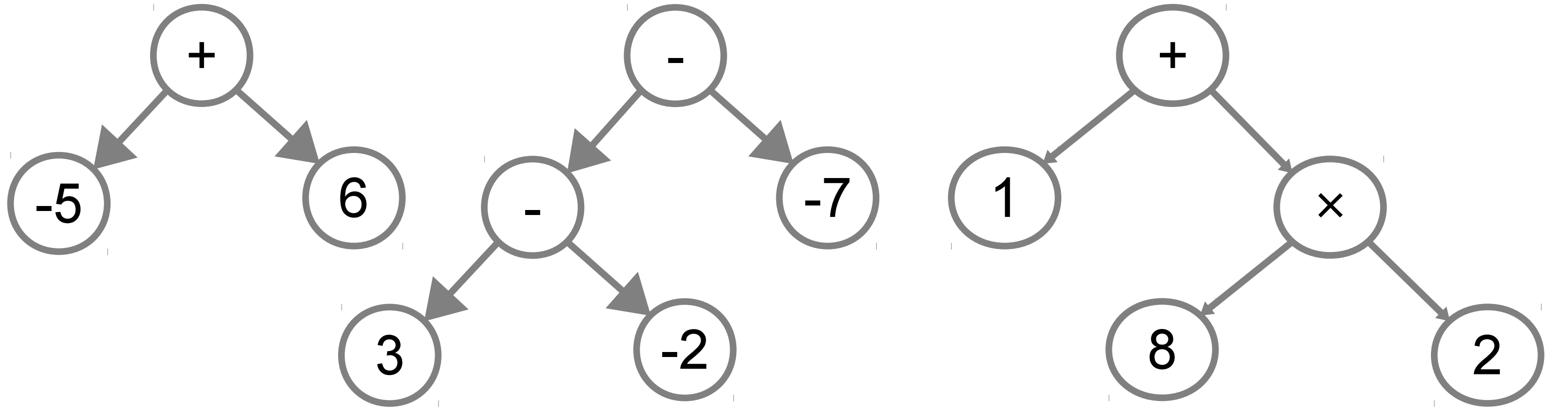


Syntax and Parsing

- Concrete Syntax of Arithmetic Expressions:

```
4
-5+6
3--2--7
3∗(8+5)
1+(8∗2)
1+8∗2
```

- Abstract Syntax of Arithmetic Expressions:

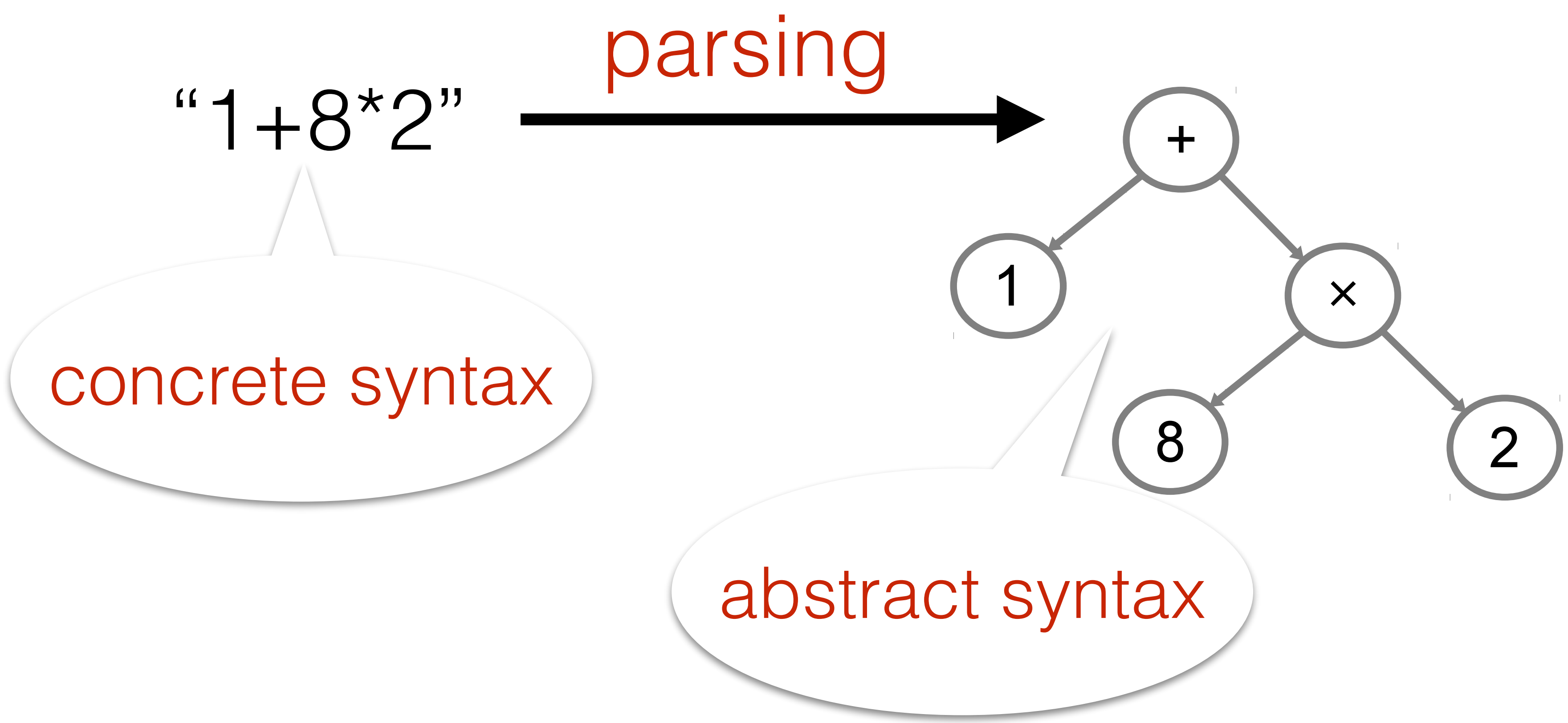


Syntax and Parsing

- Concrete Syntax: describes how the abstract concepts (such as expressions) in the language are represented as text.
  - Usually ambiguous; care need to be taken to define rules to rule out ambiguity.
- Abstract Syntax: A structured representation of the concepts in a language.
  - Unambiguous, but not as convenient/easy to understand by humans.

Syntax and Parsing

- Parsing: process that converts between concrete syntax and abstract syntax.



Abstract Syntax

- Abstract Syntax of arithmetic expressions

```
Exp ::= number | Exp + Exp | Exp - Exp | Exp * Exp | Exp / Exp
```

- How to represent this in Haskell?

Abstract Syntax

- Abstract Syntax of arithmetic expressions

```
Exp ::= number | Exp + Exp | Exp - Exp | Exp * Exp | Exp / Exp
```

- Abstract syntax can be represented nicely with Haskell datatypes

$$
\begin{array}{lll}
\mathbf{data}\ Exp = & Number & Int \\
\quad \mid & Add & Exp\ Exp \\
\quad \mid & Subtract & Exp\ Exp \\
\quad \mid & Multiply & Exp\ Exp \\
\quad \mid & Divide & Exp\ Exp
\end{array}
$$

Lets Build our First Interpreter!

Language and Meta-Language

In this course we will be using Haskell to implement other languages. In this lecture we will implement a simple language of arithmetic in Haskell.

- **Be careful not to confuse the two languages!**

Language and Meta-Language

To avoid confusion:

- we refer to the implementation language as the meta-language,
- and the language being implemented as the language.

- **What are the language and meta-language in this lecture?**

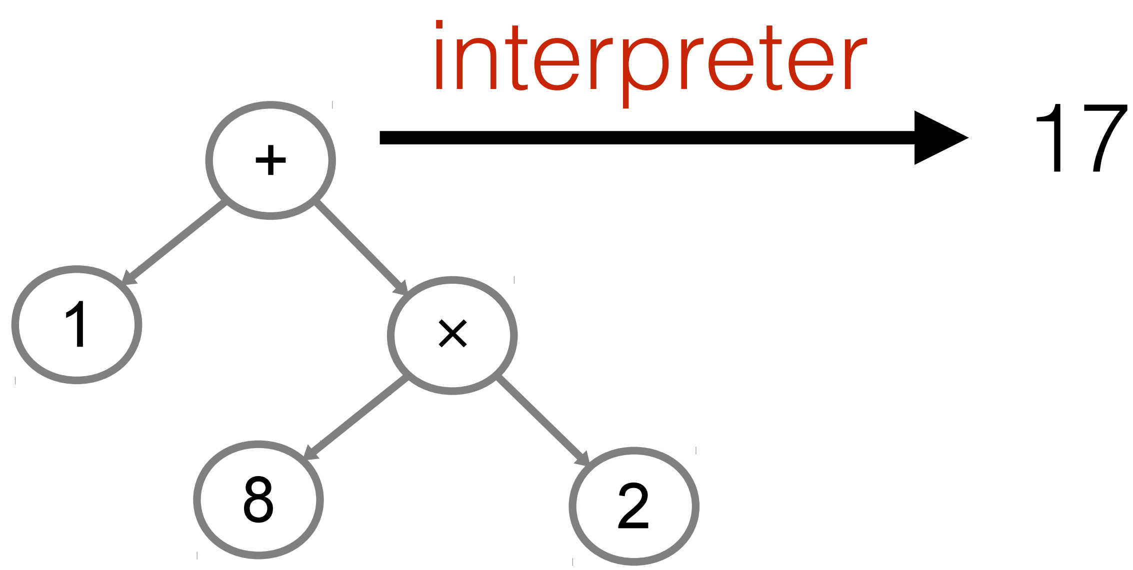
Language and Meta-Language

In this lecture:

- Haskell is the meta-language
- Arithmetic is the language

Evaluation: Compilation vs Interpretation

- Interpreter: Evaluates abstract syntax directly



Evaluation: Compilation vs Interpretation

- Compiler: Transforms abstract syntax into another language (possibly bytecode/assembly), which can be more efficiently executed afterwards.