



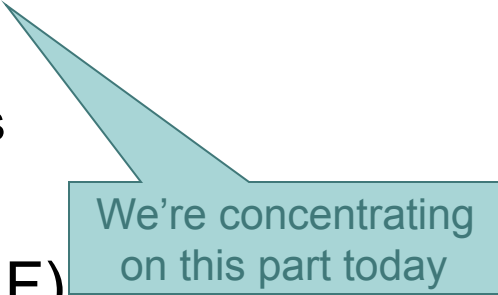
CS4450: Principles of Programming Languages

Language Processing: Syntactic Issues

Trees, “`data`” declarations, and tree programming

● ● ● | Today: Starting Language Processing

- Representing languages: Abstract Syntax
 - ...and how to represent AS in Haskell
 - Defining new types with `data` declarations
 - Abstract Syntax Trees
 - Introduction to Backus-Naur Form (BNF)
 - Wikipedia entry may be helpful
 - http://en.wikipedia.org/wiki/Backus-Naur_form
 - Read: Appendix J of Ramsey-Kamin text
- Defining Languages with Interpreters
 - Simple example: arithmetic language

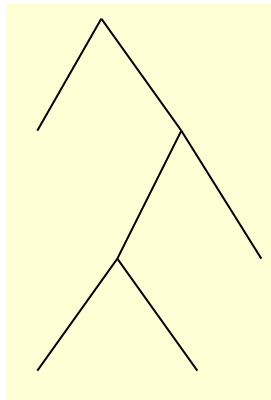


We're concentrating on this part today

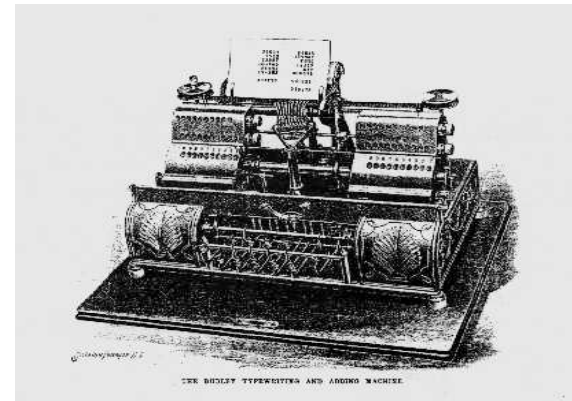


What is an interpreter?

Representation
of a program



Each expression/statement
defined in terms of Haskell

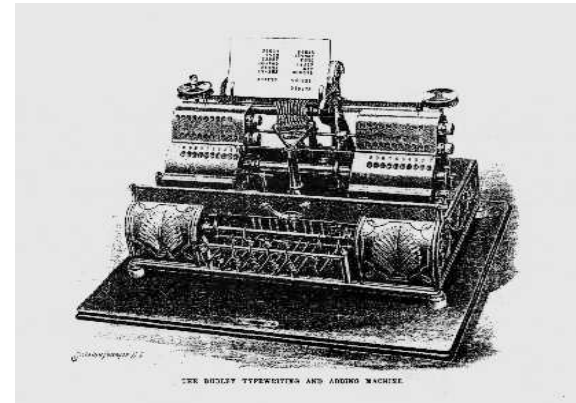
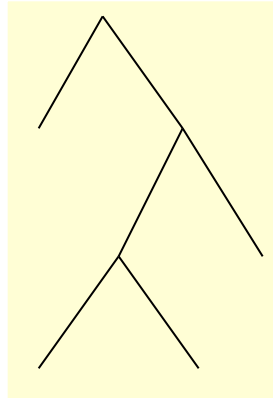




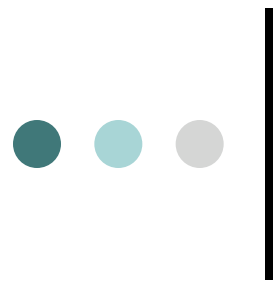
An interpreter is a function

Representation
of a program

Each expression/statement
defined in terms of Haskell

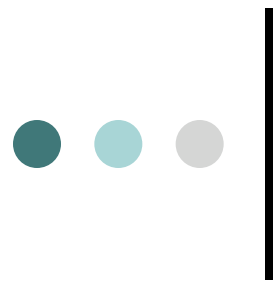


`interp :: AbstractSyntax` \longrightarrow "Computation of Values"



Abstract vs Concrete Language Syntax

- The **concrete syntax** of a language is the syntax in which the programmer writes
 - E.g., “(1 + x)” is in Haskell’s concrete syntax
 - Concrete syntax has various punctuation marks making it easy for the human programmer to read
 - Such punctuation is irrelevant to an interpreter or compiler



Abstract vs. Concrete Syntax

- **Abstract syntax** for a language is a syntactic form suitable for a language processing program like a compiler or interpreter
 - “(1 + x)” in abstract form might be:
(App (App (Op Plus) (Const 1)) (Var “x”))
 - Abstract syntax eliminates the punctuation, etc.,
 - records what the language processor needs to evaluate/compile the program



ImpCore's Concrete Syntax

Here is an ImpCore procedure declaration (page 6, Ramsey):

```
(define double (x) (+ x x))
```

Here is part of the Backus Naur form grammar for ImpCore's concrete syntax:

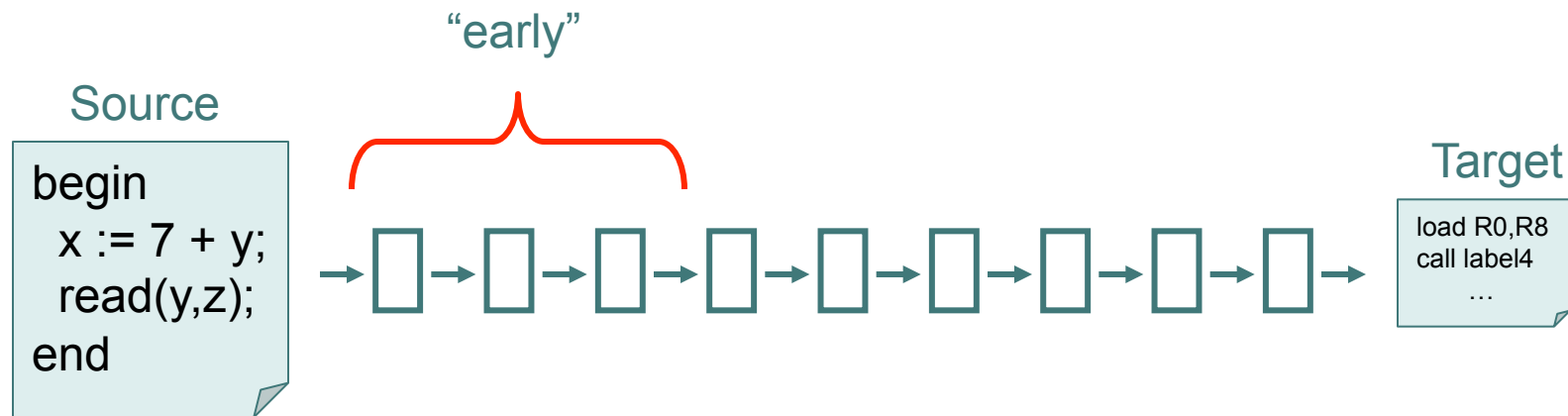
```
toplevel ::= (define function-name ( formals ) exp) | ...
```

Here is part of the BNF grammar for ImpCore's abstract syntax (p.12):

```
TopLevel ::= Define Name NameList Exp | ...
```

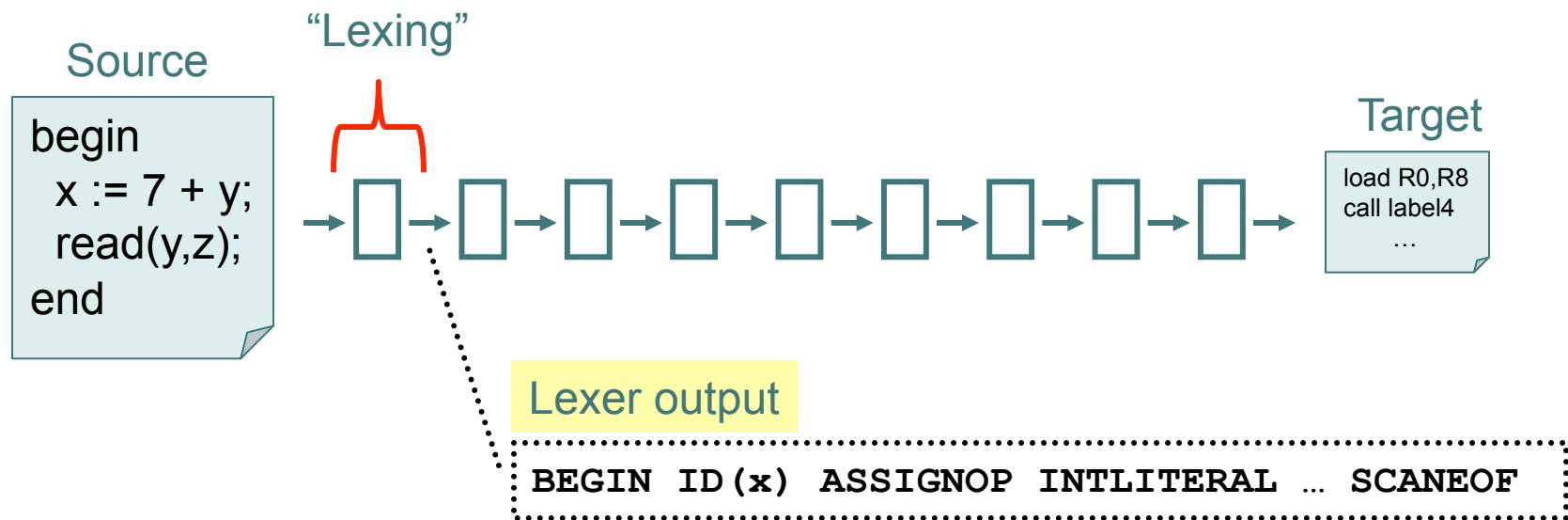
Observe that both are defined with BNF grammars

Language Processor “Phases”



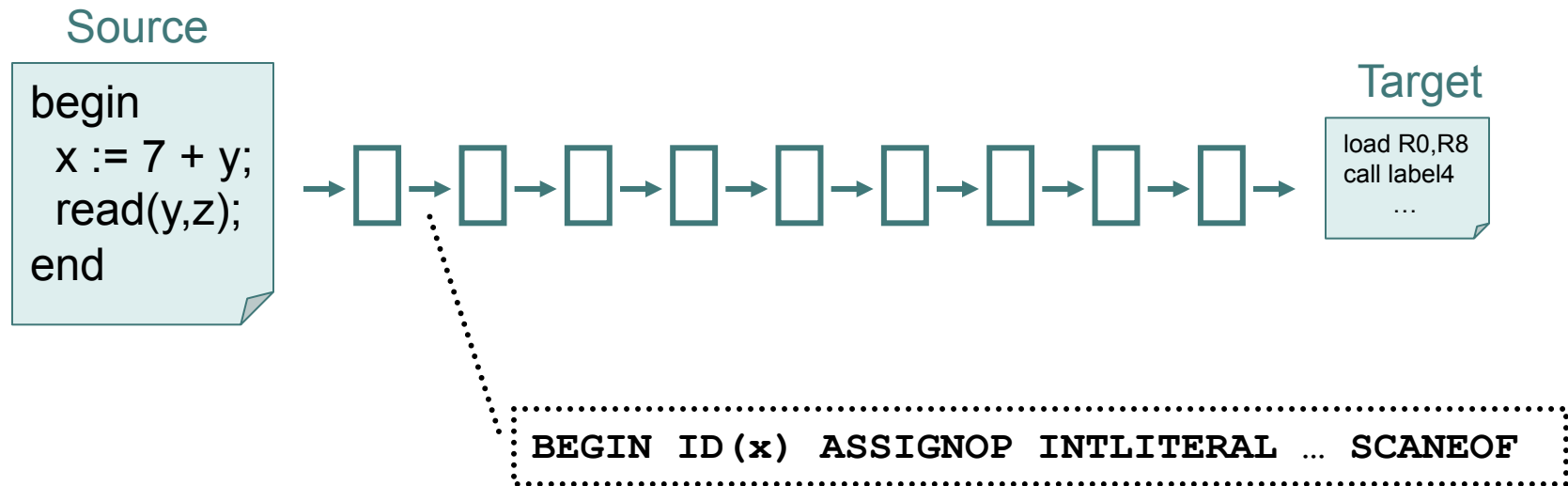
- early phases transform input sequence into tree representation (AST)
- “Lexing” turns source string into sequence of “words” or tokens
- “Parsing” checks that this token sequence is grammatically correct

● ● ● | Compiler phases



- early phases transform input sequence into tree representation (AST)
- ensure that input stream is, indeed, a program in the source language
- lexing, parsing, type-checking

Why Parsing?

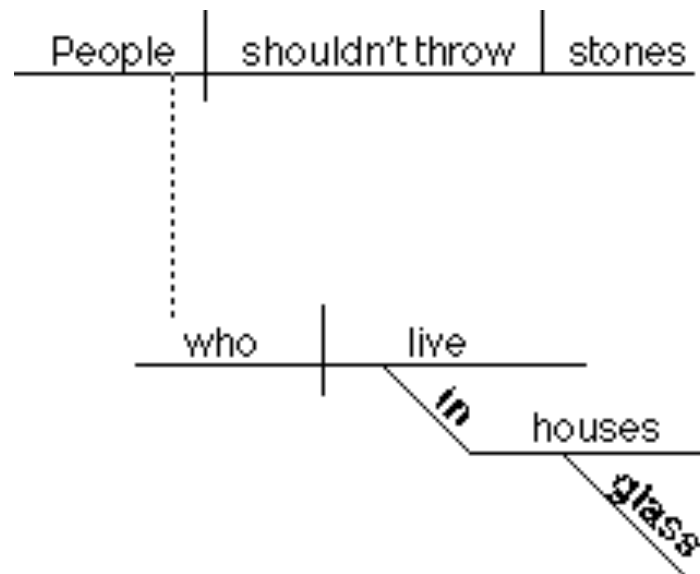


- Output of lexer is disorganized stream of symbols
 - how do you recognise the program structure in this stream?

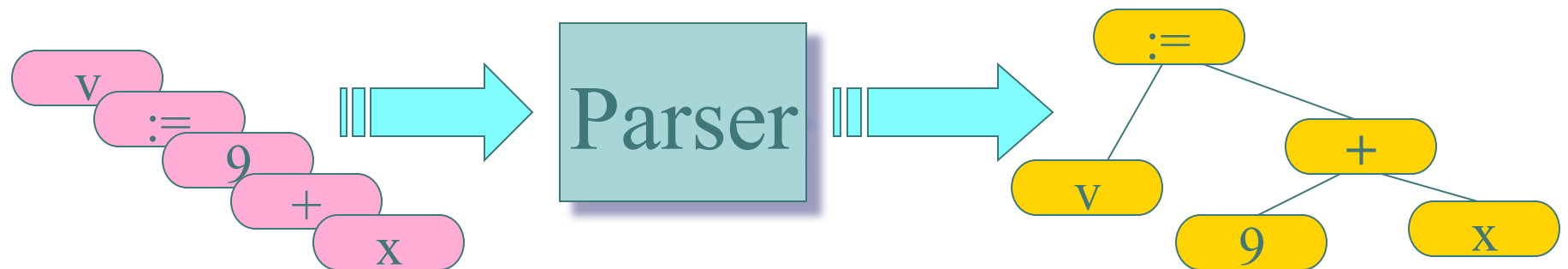


Parsing is like diagramming sentences

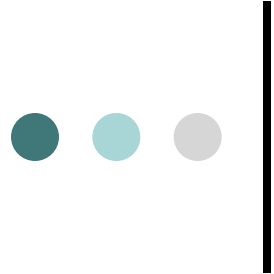
People who live in glass houses shouldn't throw stones.



● ● ● | Parsing up close



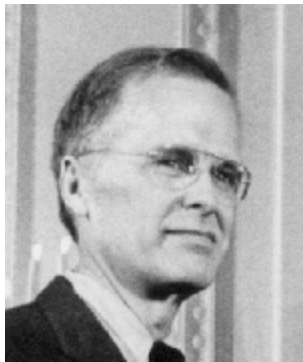
- Reads stream of tokens.
- Discovers structure in tokens.
- Produces a tree representing the source program
- Needs good error messages to help user.



Parsing and Grammars

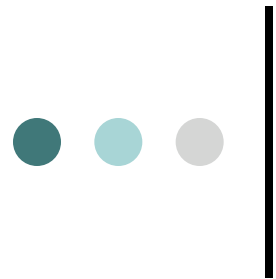
- We need to express correctness of language idioms.
 - $X + 0$ ← valid
 - $X + \text{while}(\dots) \{ \dots \}$ ← not valid
- Parsing is used to
 - Accept a valid sequence of tokens.
 - Build representations of programs called abstract syntax trees (AST)
- We will use Context-Free Grammars (CFG)
 - aka “BNF grammars”

We define the valid expressions using “BNF grammars”



- “BNF” stands for “Backus-Naur Form”
- John Backus (1928-) is one of the principal designers of FORTRAN
- In 1954, Backus publishes “Preliminary Report, Specifications for the IBM Mathematical FORMula TRANslating System, FORTRAN.”
 - Backus anticipated completion of the compiler in six months. Instead, it would take two years.
 - When completed in 1956, the compiler consisted of 25,000 lines of machine code.





Another example: lists-of-numbers LON

- (base) $() \in \text{LON}$
 - i.e., the empty list is a LON
- (ind.) if n is a number and $l \in \text{LON}$,
then $(n . l) \in \text{LON}$

Implicit assumption: that LON is the **smallest** set satisfying these conditions

Another example: lists-of-numbers LON

- $() \in \text{LON}$
 - i.e., the empty list is a LON
- if n is a number and $l \in \text{LON}$,
then $(n . l) \in \text{LON}$

The following Scheme expressions are in LON:

`() , (14 . ()) , (3 . (14 . ()))`

...or pretty-printed:

`() , (14) , (3 14)`



Backus-Naur form for LON

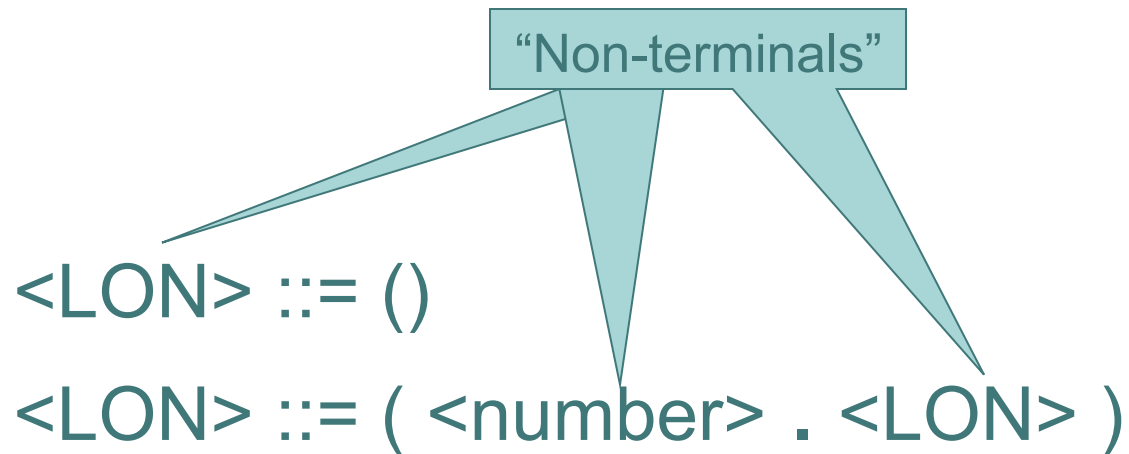
$\langle \text{LON} \rangle ::= ()$

$\langle \text{LON} \rangle ::= (\langle \text{number} \rangle . \langle \text{LON} \rangle)$

*This is a “BNF grammar”



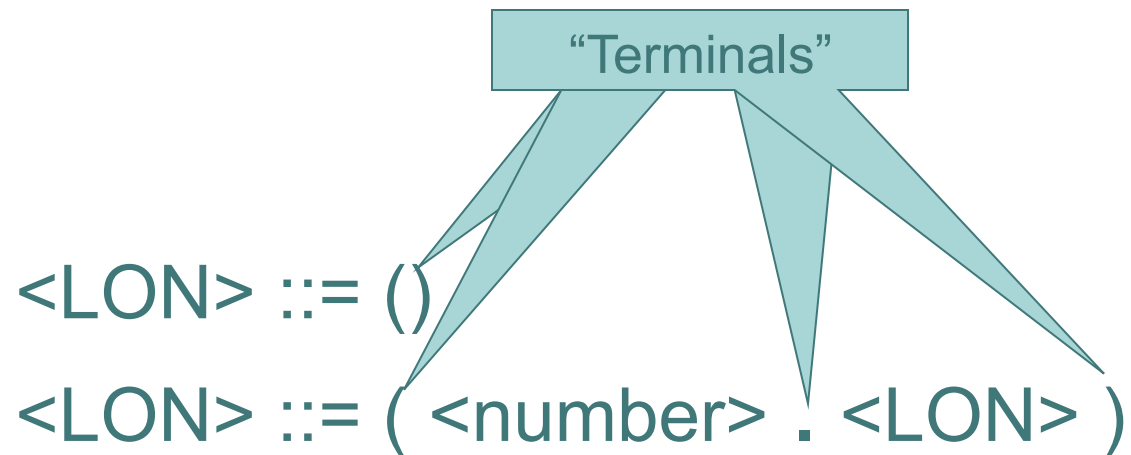
Backus-Naur form for LON



Non-terminals represent sets of items



Backus-Naur form for LON



Terminals are fixed symbols



Backus-Naur form for LON

“Productions”

$\langle \text{LON} \rangle ::= ()$

$\langle \text{LON} \rangle ::= (\langle \text{number} \rangle . \langle \text{LON} \rangle)$

Rules for forming objects are called “productions”



Backus-Naur form for LON

$\langle \text{LON} \rangle ::= ()$

$\langle \text{LON} \rangle ::= (\langle \text{number} \rangle . \langle \text{LON} \rangle)$

compare with

$() \in \text{LON}$

if n is a number and $l \in \text{LON}$, then $(n . l) \in \text{LON}$



Other extensions to BNF: alternation

$\langle \text{LON} \rangle ::= () \mid (\langle \text{number} \rangle . \langle \text{LON} \rangle)$

...can join productions together with “|”



Other extensions to BNF: Kleene star

$\langle \text{LON} \rangle ::= (\langle \text{number} \rangle^*)$

...can represent sequences of $\langle s \rangle$ with $\langle s \rangle^*$

$$\langle s \rangle^* = \{ x_1 \dots x_n \mid x_i \in \langle s \rangle \text{ and } 0 \leq i \leq n \}$$

- ● ● | Other extensions to BNF:
Kleene plus

$\langle \text{LON} \rangle ::= (\langle \text{number} \rangle^+)$

...can represent sequences of $\langle s \rangle$ with $\langle s \rangle^+$

$\langle s \rangle^+ = \{ x_1 \dots x_n \mid x_i \in \langle s \rangle \text{ and } 1 \leq i \leq n \}$



Simple arithmetic language

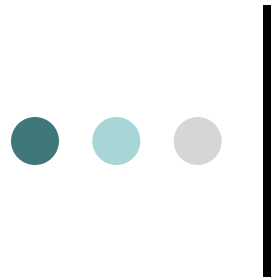
- Consider a small language consisting of:
 - integer constants,
 - +
 - *
- Valid expressions in the language:
 - 1, 2+3, 25 * 7 + 4
- Invalid expressions
 - 1 +, * 9 100, 1 + * 2,...
- **Q:** How do we precisely distinguish the valid from the invalid?



BNF grammar for expressions

1. $\text{Exp} \rightarrow \text{Int}$
2. $\text{Exp} \rightarrow \text{Exp} + \text{Exp}$
3. $\text{Exp} \rightarrow \text{Exp} * \text{Exp}$

- Each of lines 1-3 is a “production”
- “*Int*” and “+” “*” are “terminals”
 - meaning they are symbols/tokens
 - *Int* stands for any token: ..., -1, 0, 1, ...
- “Exp” is a “non-terminal”
 - meaning it doesn’t stand for a symbol



Recognizing the valid expressions

An sequence of tokens is an expression if, and only if, a derivation of it exists starting from Exp

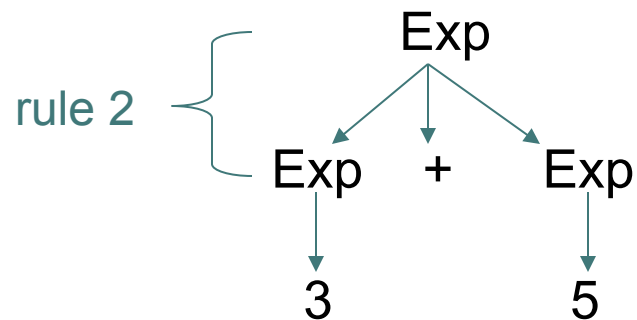
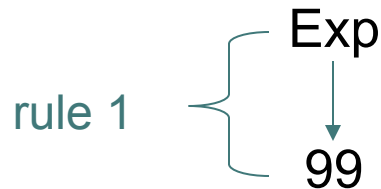
$$\text{Exp} \rightarrow_1 99$$

$$\begin{aligned} \text{Exp} &\rightarrow_2 \text{Exp} + \text{Exp} \\ &\rightarrow_1 3 + \text{Exp} \\ &\rightarrow_1 3 + 5 \end{aligned}$$

\therefore “99” and “3 + 5” are expressions

Recognizing the valid expressions

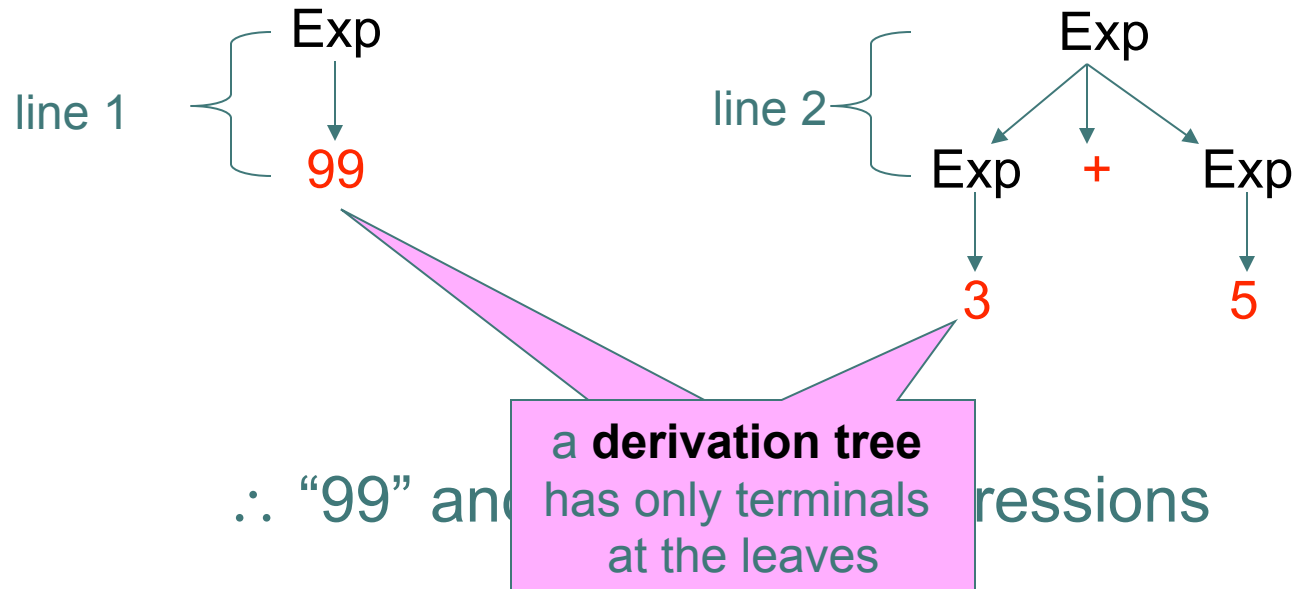
An sequence of tokens is an expression if, and only if, a derivation of it exists starting from Exp



∴ “99” and “3 + 5” are expressions

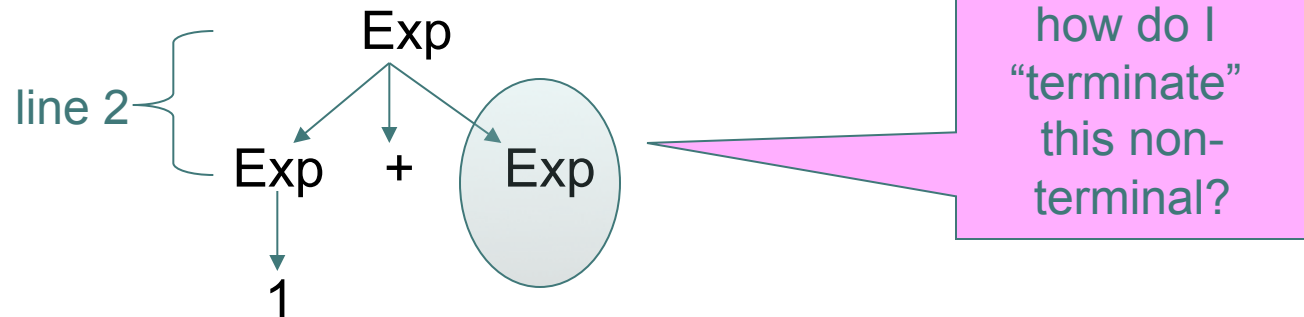
Recognizing the valid expressions

An sequence of tokens is an expression if, and only if, a derivation of it exists starting from Exp



Recognizing the valid expressions

Note that invalid expressions don't have derivations



\therefore "1 +" is not an expression



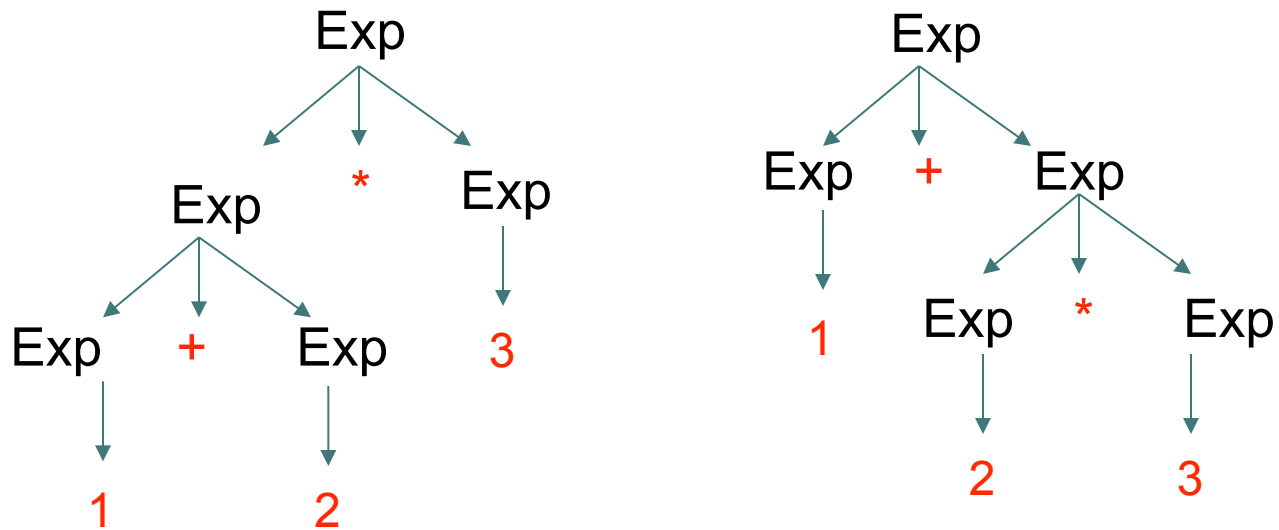
Basic ideas of parsing

- Grammars have languages as well
 - E.g., The language of `Exp` is any sequence of tokens for which a derivation exists
 - such a sequence is sometimes called a “sentence”
- Parsing asks the question:
 - given a sequence of tokens, does a derivation tree exist for it
 - and, if so, what is it?
- Subtlety: a grammar may be **ambiguous**, meaning that a sentence may have more than one derivation



Ambiguous Grammars

The expression grammar is ambiguous





What is a “language”?

- A **language** is a set of strings
 $\{ \text{“class”}, \text{“extends”}, \text{“+”}, \text{“899”}, \dots \}$
- Each **string** is a finite sequence of characters from an alphabet
 - This is what a lexer defines
- An **alphabet** is a set of characters
 $\{ a, b, c, \dots, 0, 1, \dots, 9, _, +, =, \dots \}$

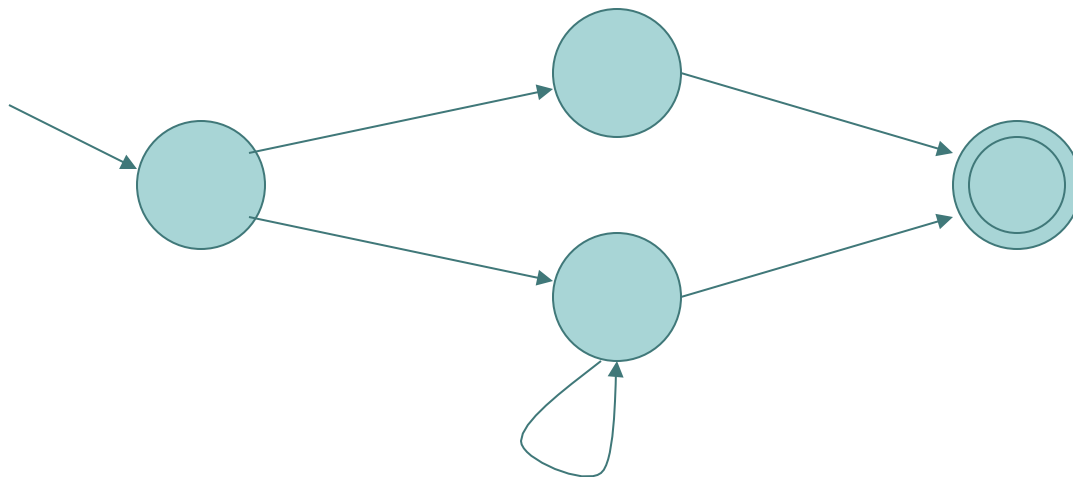
Question: Can we define languages with regular expressions/Finite Automata?

No, that’s why we need BNF.

● ● ● | Answer: yes ...

- Certainly can define sets of strings with REs/
FAs

{ Strings s | s is recognised by M }



● ● ● | ...and no!

- The kinds of languages we're interested in can not be recognised by a finite-state automaton!

Ex: the "balanced parentheses language L:

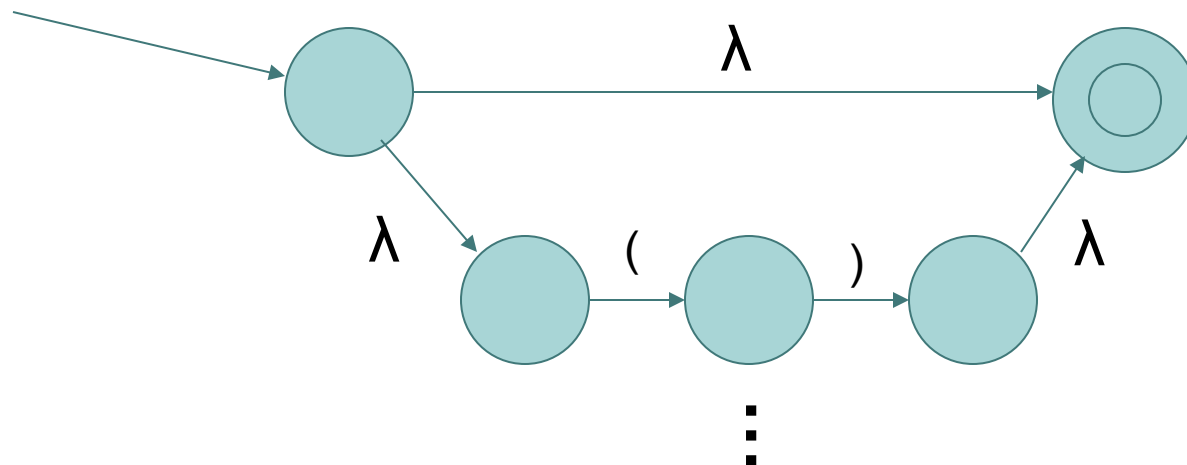
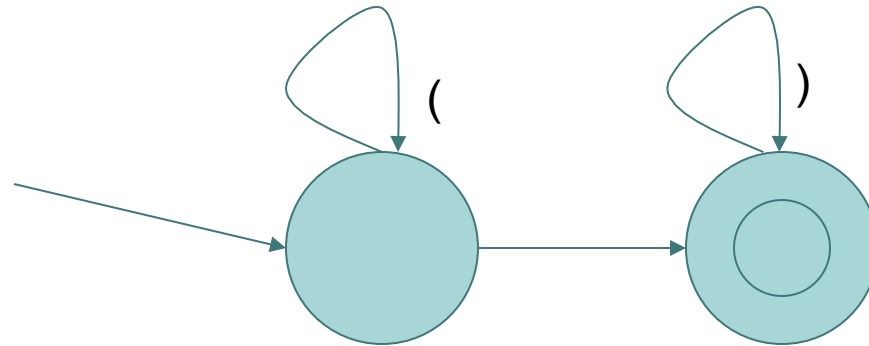
“” $\in L$

if $s \in L$, then $(s) \in L$

$L = \{ \text{“”}, (), (()), ((())), \dots \}$



Various (incorrect) solutions





Context-free Grammars (aka Backus Naur Form grammars)

- A Context-free grammar (CFG) describes a language.
- Grammars have sets of productions

symbol → symbol symbol symbol ... symbol

Zero or more symbols on right hand side.

Two types of symbols

- Terminals
 - From the alphabet of strings
 - These are your **Tokens** from the lexer.
 - The semantic value of Tokens/Terminals plays no part in the context free grammar.
- Non-terminals
 - Appears on the left hand side of a production.



Scheme's arithmetic is different from Haskell's

“Permissive typing”

```
> (+ 1 2.0)
3.0
>
```

“arbitrary arities”

```
> (+)
0
> (+ 1)
1
> (+ 1 2 3 4)
10
```

Haskell's type system would never allow this!

```
> :t (+)
(+) :: Num a => a -> a -> a
>(1 :: Int) + (2.0 :: Float)
error!
```



Backus-Naur Form (BNF)

Exp ::= (Op ExpList)

Exp ::= *Int* | *Real*

Op ::= + | - | * | /

ExpList ::= Exp ExpList

ExpList ::= λ

Describes the syntax of Scheme's arithmetic
N.b., it's recursive, too.

Q: How do we represent Exp in Haskell?

the empty string; unrelated to lambda expressions



Backus-Naur Form (BNF)

Exp ::= (Op ExpList)

Exp ::= *Int* | *Real*

Op ::= + | - | * | /

ExpList ::= Exp ExpList

ExpList ::= λ

- Symbols to the left of ::= are **non-terminal** symbols
- ... to the right of ::= that is not “|” is a **terminal** symbol
 - *Int* and *Real* are terminal symbols that stand for any integer or real number
- “ $X ::= X_1 \dots X_n$ ” is a production
- Exp, Op, ExpList are non-terminals
- $(,), Int, Real, +, *, -, /, \lambda$



Language defined by BNF

Q: How do we know that the sequence of symbols, (+ 1 2), is in the language of Exp?

Exp ::= (Op ExpList)

Exp ::= *Int* | *Real*

Op ::= + | - | * | /

ExpList ::= Exp ExpList

ExpList ::= λ

We construct a **derivation**:

```
Exp => ( Op ExpList )
    => ( + ExpList )
    => ( + Exp ExpList )
    => ( + 1 ExpList )
    => ( + 1 Exp ExpList )
    => ( + 1 2 ExpList )
    => ( + 1 2 )
```

* Parsing = Finding a Derivation



BNF gives rise to trees

Derivation of $(+ 1 2)$ represented as

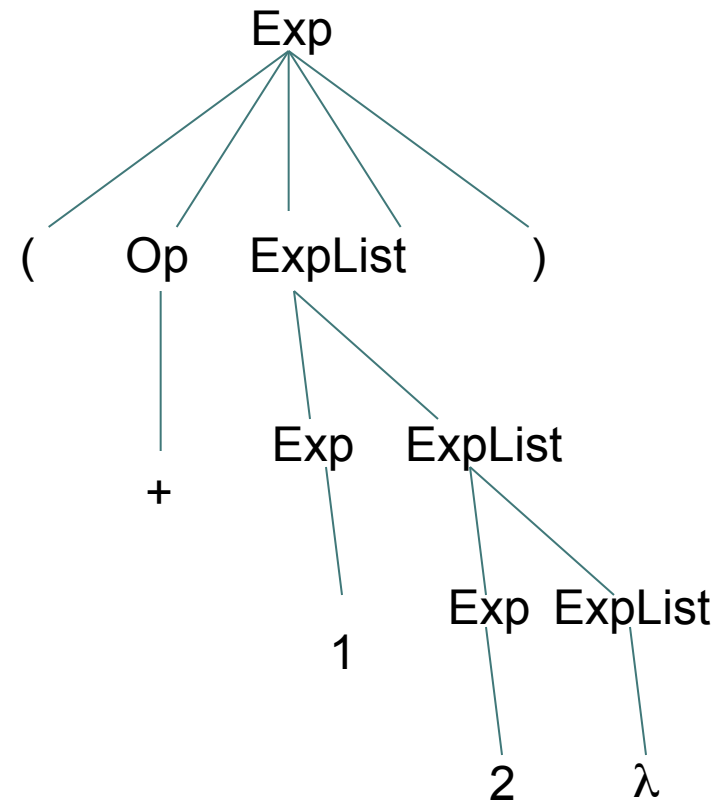
$\text{Exp} ::= (\text{Op ExpList})$

$\text{Exp} ::= \text{Int} \mid \text{Real}$

$\text{Op} ::= + \mid - \mid * \mid /$

$\text{ExpList} ::= \text{Exp ExpList}$

$\text{ExpList} ::= \lambda$



Programs in a language are usually represented as **trees**

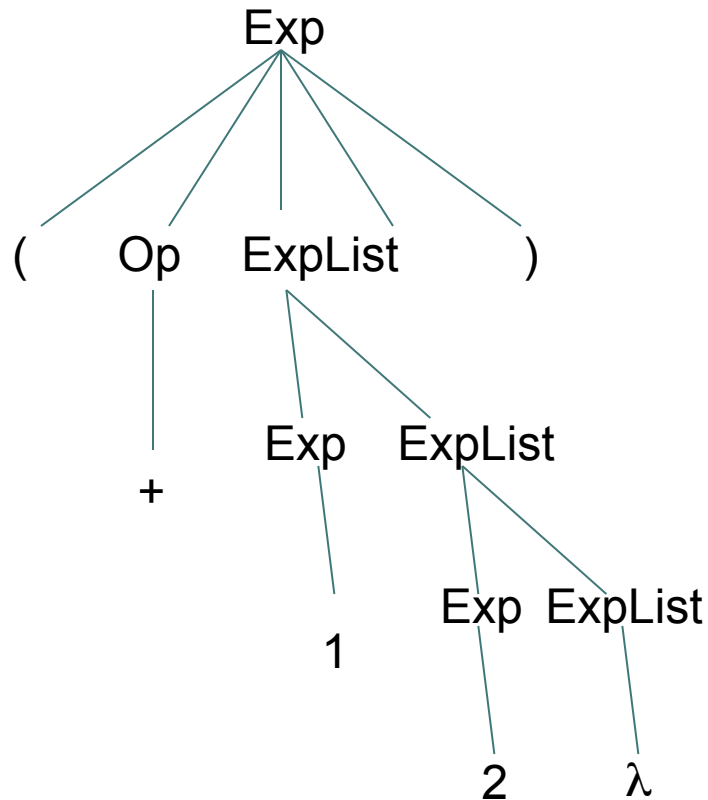


Abstract syntax trees (AST)

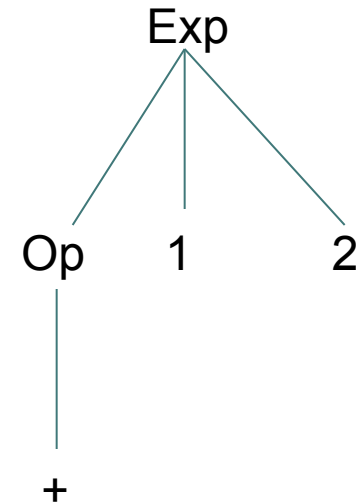
dispose of unnecessary details

Ex: (+ 1 2)

“concrete”



“abstract”

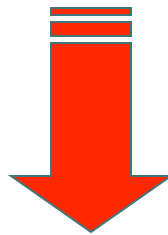




Simple composite data

```
data Value = I Int | R Float
```

```
Op ::= + | - | * | \
```



```
data Op = Plus | Minus | Times | Div
```



Structure of a data declaration

the type name being declared: aka “*type constructor*”

```
data Value = I Int | R Float
```

I and R are “*data constructors*”



Using `data` declaration

```
data Value = I Int | R Float
```

```
inc (I i) = I (i + 1)
```

```
inc (R r) = R (r + 1.0)
```

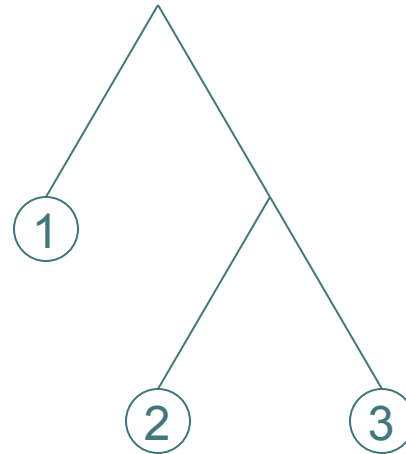
Haskell responds: `inc :: Value -> Value`

* N.b., the “**I**” and “**R**” have dual uses as constructors and patterns

● ● ● | Binary trees

A binary tree of `Int` values

②



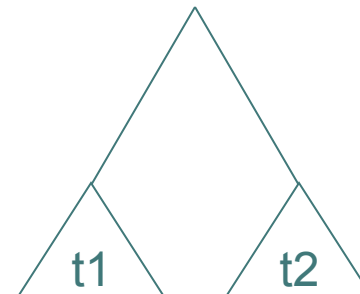
Let's figure out how to represent binary trees as a data declaration

- ● ● | Building Binary trees

- A “leaf” – i.e., a single node with value v



- Given two trees,  & 



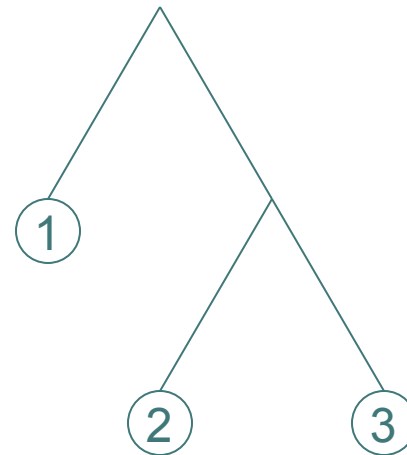


```
data Itree = Leaf Int  
          | Bnode Itree Itree
```

A binary tree of `Int` values

②

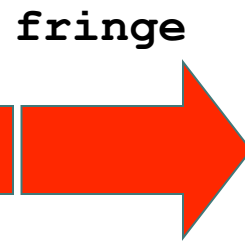
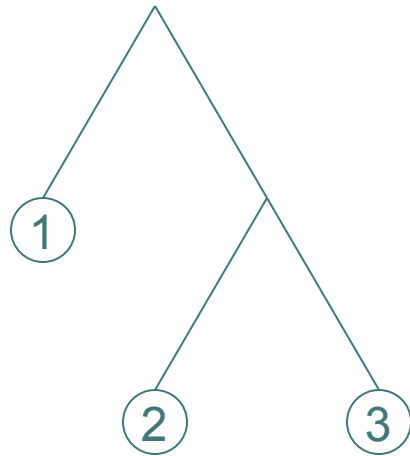
`(Leaf 2)`



`Bnode (Leaf 1)
(Bnode (Leaf 2) (Leaf 3))`



The “fringe” of an Itree



1,2,3



The “fringe” of an `Itree`

```
data Itree = Leaf Int
           | Bnode Itree Itree
```

```
fringe (Leaf i)      = ...
fringe (Bnode t1 t2) = ...
```

Just like lists with `[]` and `(x:xs)`, an `Itree` is either a `Leaf` or a `Bnode`
∴ programming with trees has an analogous “template”!

Let's try writing `fringe`

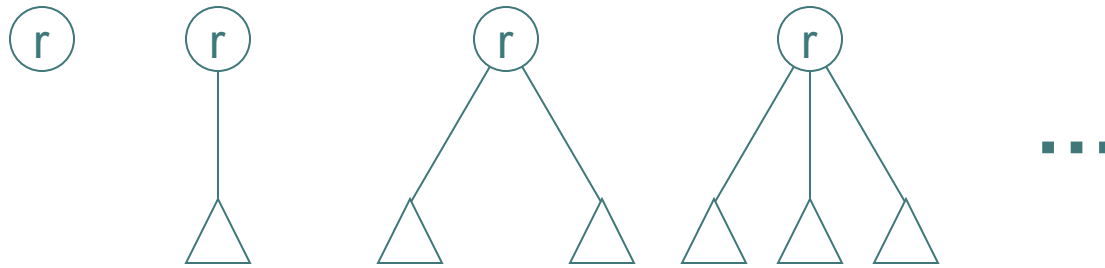


The “fringe” of an Itree

```
data Itree = Leaf Int
           | Bnode Itree Itree
```

```
fringe (Leaf i)      = [i]
fringe (Bnode t1 t2) = (fringe t1) ++ (fringe t2)
```

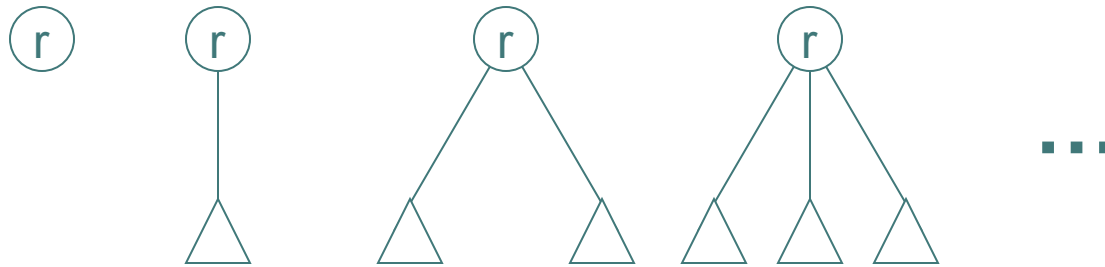
● ● ● | Trees of arbitrary branching degree



- The **branching degree** is the number of branches



Trees of arbitrary branching degree



```
data Atree = Leaf Int | Anode [Atree]
```

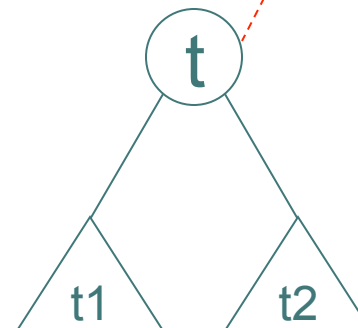
compare with the binary tree:

```
data Itree = Leaf Int  
           | Bnode Itree Itree
```

● ● ● | Building Tagged Binary trees

- Leaves unchanged 

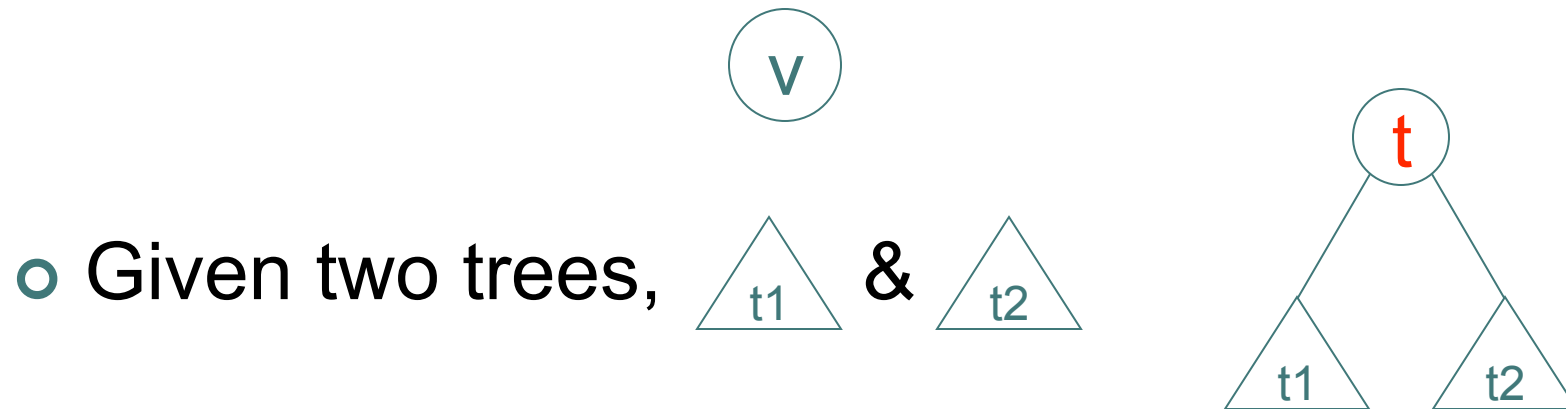
- Given two trees,



*sometimes a tag
is included on
“interior” nodes
--- for example,
an “Op”*

● ● ● | Building Tagged Binary trees

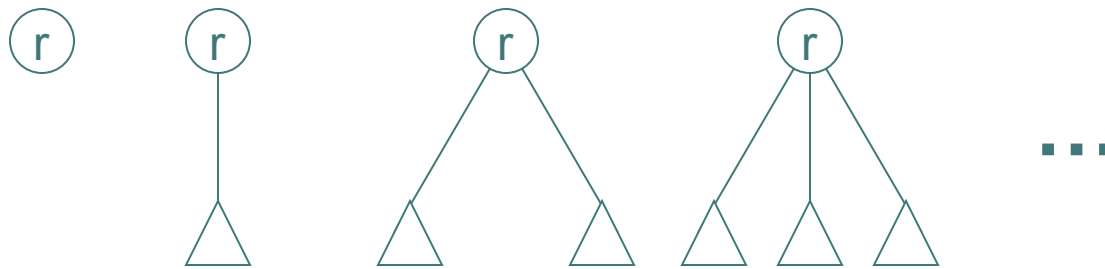
- Leaves are unchanged



```
data Itree = Leaf Int
           | Bnode Tag Itree Itree
```



Tagging Trees of arbitrary branching degree



```
data Atree = Leaf Int | Anode Tag [Atree]
```



...and back to Scheme

```
data Atree = Leaf Int | Anode Tag [Atree]
```

```
data Value = I Int | R Float
data Op     = Plus | Minus | Times | Div
data Exp    = Const Value | Aexp Op [Exp]
```

Const is like **Leaf**

Aexp is like **Anode**

Op is like **Tag**

● ● ● | Compare Exp with **Exp**

```
Exp ::= ( Op ExpList )  
Exp ::= Int | Real  
Op ::= + | - | * | \  
ExpList ::= Exp ExpList  
ExpList ::=  $\lambda$ 
```

```
data Value = I Int | R Float  
data Op     = Plus | Minus | Times | Div  
data Exp    = Const Value | Aexp Op [Exp]
```



Scheme AS in Haskell

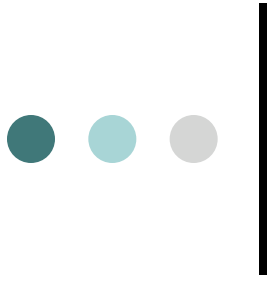
```
data Value = I Int | R Float
data Op    = Plus | Minus | Times | Div
data Exp   = Const Value | Aexp Op [Exp]
```

Scheme	Abstract Syntax Tree in Haskell
1	Const (I 1)
(*)	Aexp Times []
(+ 1)	Aexp Plus [Const (I 1)]
(+ 1 (*))	Aexp Plus [Const (I 1), Aexp Times []]



Next time

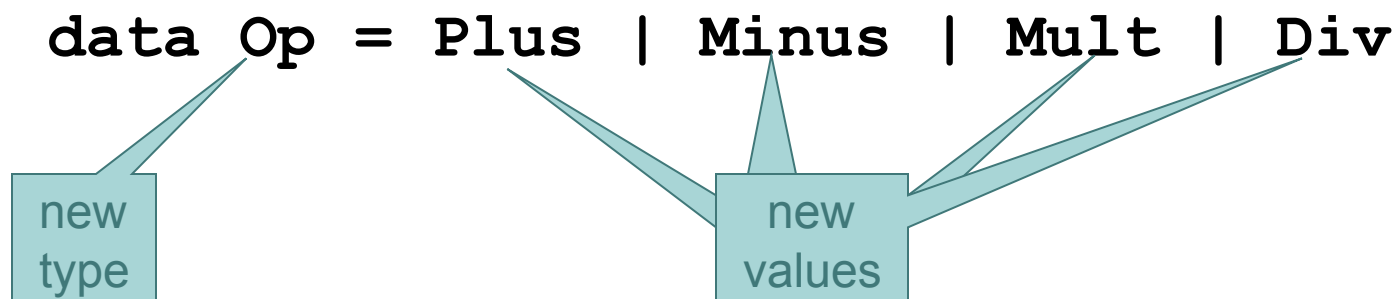
- More on Language Processing
 - Building an interpreter for Exp



The slides that follow may be of use to you. They cover material that we have discussed elsewhere.

• • • | Type declarations via data

- Thus far, we have only used built-in types
 - `Int [a] (a,b) a->b ...`
- The only declarations we've seen have been of functions
- Data declarations allow us to define new kinds of types



● ● ● | Programming with new types

Programming a function taking Bool input

```
f True = ...  
f False = ...
```

“design pattern” dictated
by the structure of input data:
`data Bool = True | False`

```
data Op = Plus | Minus | Mult | Div  
f :: Op -> ...  
f Plus = ...  
f Minus = ...  
f Mult = ...  
f Div = ...
```

ditto new data types



Maybe type

There's a built-in data type useful for representing errors

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

Generally speaking:

Just v is used to represent “I did what you asked and the value is **v**”

Nothing is used to represent “I couldn't do what you asked”

How **Maybe** is used:

If you have a function, $f :: a \rightarrow b$, that might crash for some **a**-values, Then, redefine it, $f' :: a \rightarrow \text{Maybe } b$, so that it returns **Nothing** instead of crashing. Also, so that $f' \text{ arg} = \text{Just } v \Leftrightarrow f \text{ arg} = v$



Using Maybe: an example

Recall the function, `nth`, which returns the n^{th} element of a list:

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

The problem with this function is that, if `n > length l`, then `nth n l` will crash

```
Hugs> nth 5 [1,2,3]
```

```
Program error: pattern match failure: nth 2 []
```

This can be fixed if we change the range of the function to `Maybe a`:

```
nth :: Int -> [a] -> Maybe a
nth _ []          = ?
nth 1 (x:xs)     = ?
nth (n+1) (x:xs) = nth n xs
```



Using Maybe: an example

Recall the function, `nth`, which returns the n^{th} element of a list:

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

The problem with this function is that, if $n > \text{length } l$, then `nth n l` will crash

```
Hugs> nth 5 [1,2,3]
```

```
Program error: pattern match failure: nth 2 []
```

This can be fixed if we change the range of the function to `Maybe a`:

```
nth :: Int -> [a] -> Maybe a
nth _ []          = Nothing
nth 1 (x:xs)      = Just x
nth (n+1) (x:xs) = nth n xs
```



Silly example

```
incthird l = case (nth 3 l) of
  (Just v) -> Just (v+1)
  Nothing  -> Nothing
```



Silly example

```
incthird 1 = case (nth 3 1) of
                (Just v) -> Just (v+1)
                Nothing  -> Nothing
```

There's a useful pattern here that propagates `Nothing`'s:

```
case <Could be Nothing> of
  (Just v) -> <do something with v>
  Nothing  -> Nothing    -- propagate the error
```

● ● ● | Propagation pattern

Let's capture this pattern as a higher-order function:

```
case <Could be Nothing> of
  (Just v) -> <do something with v>
  Nothing  -> Nothing    -- propagate the error
```

```
x `prop` f = case x of
  (Just v) -> f v
  Nothing  -> Nothing
```

What's the type of `prop`?

● ● ● | Propagation pattern

Let's rewrite incthird with prop:

```
incthird l = case (nth 3 l) of
    (Just v) -> Just (v+1)
    Nothing  -> Nothing
```

```
incthird l = ?
```

```
prop :: Maybe a -> (a -> Maybe b) -> Maybe b
x `prop` f = case x of
```

```
    (Just v) -> f v
    Nothing  -> Nothing
```



Propagation pattern

Let's rewrite `incthird` with `prop`:

```
incthird l = case (nth 3 l) of
              (Just v) -> Just (v+1)
              Nothing  -> Nothing
```

```
incthird l
= (nth 3 l) `prop` (\ v -> Just (v+1))
```

(`x `prop` f`) checks

1. if `x` fails (if so, whole thing fails);
2. otherwise, feeds `v` from `(Just v)` to `f`

```
prop :: Maybe a -> (a -> Maybe b) -> Maybe b
x `prop` f = case x of
              (Just v) -> f v
              Nothing  -> Nothing
```



Consider these two examples: What does this error message mean?

```
Hugs> 99  
99
```

```
Hugs> Plus  
ERROR - Cannot find "show" function for:  
*** Expression : Plus  
*** Of type    : Op
```

- ● ● | Int is in the “Show” class and Op isn’t

```
Hugs> :info Int
-- type constructor
data Int
-- instances:
instance Eq Int
...<snip>...
instance Show Int
```

Int is in many classes

```
Hugs> :info Op
-- type constructor
data Op
```

Op isn't in any class



Adding Op to the Show class

Defn. of Show

```
Hugs> :i Show
-- type class
class Show a where
  show :: a -> String
```

Declaring an Instance

```
instance Show Op where
  show Plus = "+"
  show Mult = "*"
  show Sub  = "-"
  show Div  = "/"
```

show :: Op -> String



Eq class is similar

```
instance Eq Op where  
  Plus == Plus = True  
  Mult == Mult = True  
  Sub == Sub = True  
  Div == Div = True  
  _ == _ = False
```