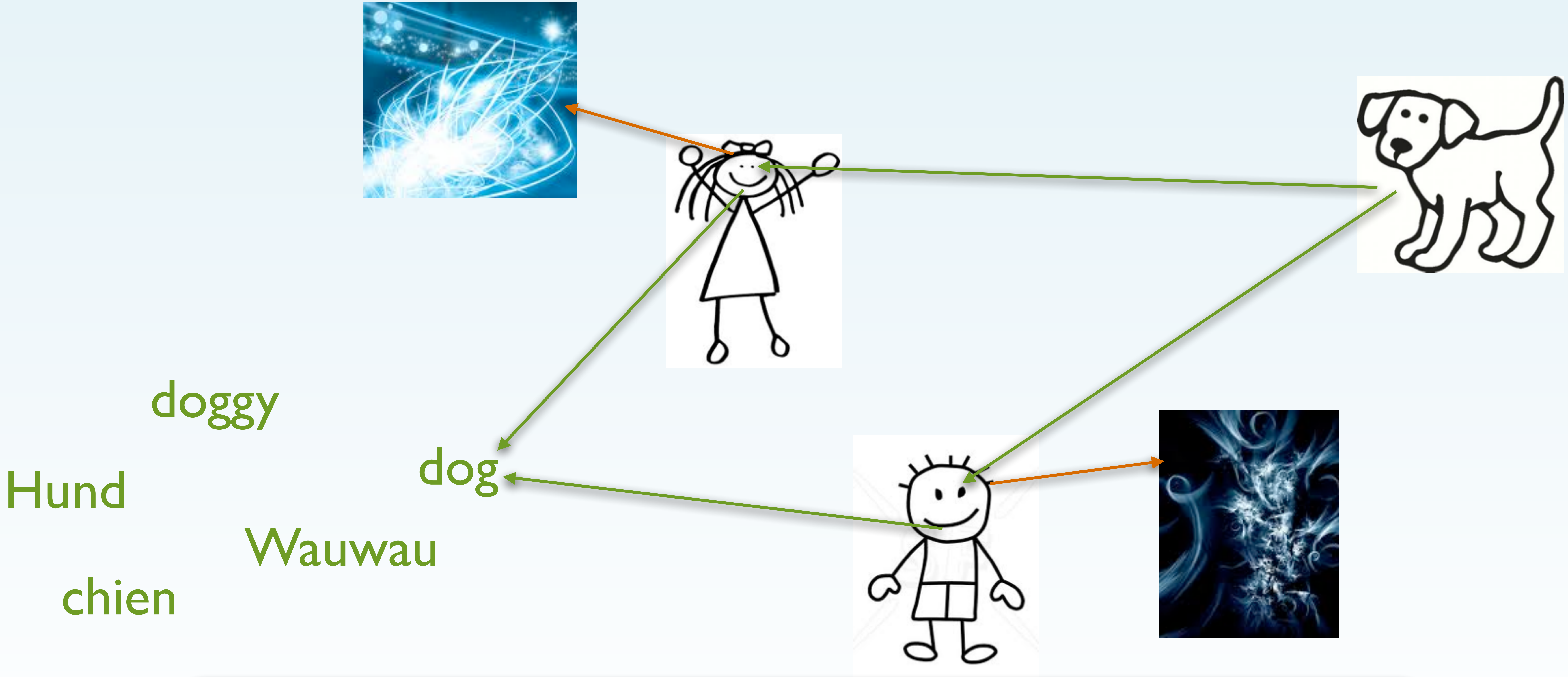
2 Syntax

For snoring?!
Hell, that's nothin'.
I once shot a man for ending a sentence in a preposition.

Sharing Thoughts



Syntax: Agreed-upon representation for semantic concepts

2 Syntax

Grammars & Derivation

Syntax Tree

Abstract vs. Concrete Syntax

Representing Grammars by Haskell Data Types

Well-Structured Sentences

The *syntax* of a language defines the set of all sentences.

How can syntax be defined?

Enumerate all sentences

Define rules to construct valid sentences

Grammar

Grammar

A grammar is given by a set of *productions* (or *rules*).

A ::= B C ...

LHS

RHS

A, *B*, *C* ... are *symbols* (= strings)

How are sentences generated by rules?

Start with one symbol and repeatedly expand symbols by RHSs of rules.

A grammar is called *context free* if all LHSs contain only 1 symbol.

Example Grammar

sentence	::=	noun verb noun	(R1)
noun	::=	dogs	(R2)
noun	::=	teeth	(R3)
verb	::=	have	(R4)

nonterminal symbols

(*do* appear on the LHS of some rule, i.e., *can* be expanded)

terminal symbols

(*do not* appear on the LHS of any rule, i.e. *cannot* be expanded)

Derivation

sentence	::=	noun verb noun	(R1)
noun	::=	dogs	(R2)
noun	::=	teeth	(R3)
verb	::=	have	(R4)

sentence	
noun verb noun	*apply rule* (R1)
dogs verb noun	*apply rule* (R2)
dogs have noun	*apply rule* (R4)
dogs have teeth	*apply rule* (R3)

Repeated rule application (i.e. replacing nonterminal by RHS) yields sentences.

Derivation Order

The order of rule application is *not* fixed.

sentence	::=	noun verb noun	(R1)
noun	::=	dogs	(R2)
noun	::=	teeth	(R3)
verb	::=	have	(R4)

sentence	
noun verb noun	(R1)
noun have noun	(R4)
noun have teeth	(R3)
dogs have teeth	(R2)

sentence	
noun verb noun	(R1)
noun verb teeth	(R3)
dogs verb teeth	(R2)
dogs have teeth	(R4)

Exercises

(1) Extend the "sentence" grammar to allow the creation of "and" sentences

sentence	::=	noun verb noun
noun	::=	dogs
noun	::=	teeth
verb	::=	have

(2) Write a grammar for binary numbers

(3) Derive the sentence 101

(4) Write a grammar for boolean expression built from the constants T and F and the operation not

(5) Derive the sentence not not F

Exercises

(1) Extend the "sentence" grammar to allow the creation of "and" sentences

sentence	::=	noun verb noun
noun	::=	dogs
noun	::=	teeth
verb	::=	have

sentence	::=	noun verb noun
sentence	::=	sentence and sentence
…		

(2) Write a grammar for binary numbers

digit	::=	0	(R1)
digit	::=	1	(R2)
bin	::=	digit	(R3)
bin	::=	digit bin	(R4)

bin	::=	0 bin	(R1)
bin	::=	1 bin	(R2)
bin	::=	ϵ	(R3)

"empty RHS"

Exercises

(3) Derive the sentence 101

digit	::= 0		(R1)
digit	::= 1		(R2)
bin	::= digit		(R3)
bin	::= digit bin		(R4)

bin	::= 0 bin	(R1)
bin	::= 1 bin	(R2)
bin	::= ϵ	(R3)

bin	
digit bin	(R4)
digit digit bin	(R4)
digit digit digit	(R3)
digit digit 1	(R2)
digit 0 1	(R1)
1 0 1	(R2)

bin	
1 bin	(R2)
1 0 bin	(R1)
1 0 1 bin	(R2)
1 0 1	(R3)

Exercises

(4) Write a grammar for boolean expression built from the constants T and F and the operation not

bool	::= T		(R1)
bool	::= F		(R2)
bool	::= not bool		(R3)

(5) Derive the sentence not not F

bool	
not bool	(R3)
not not bool	(R3)
not not F	(R2)

Why Grammar Matters ...

Video clip

2 Syntax

Grammars & Derivation

Syntax Tree

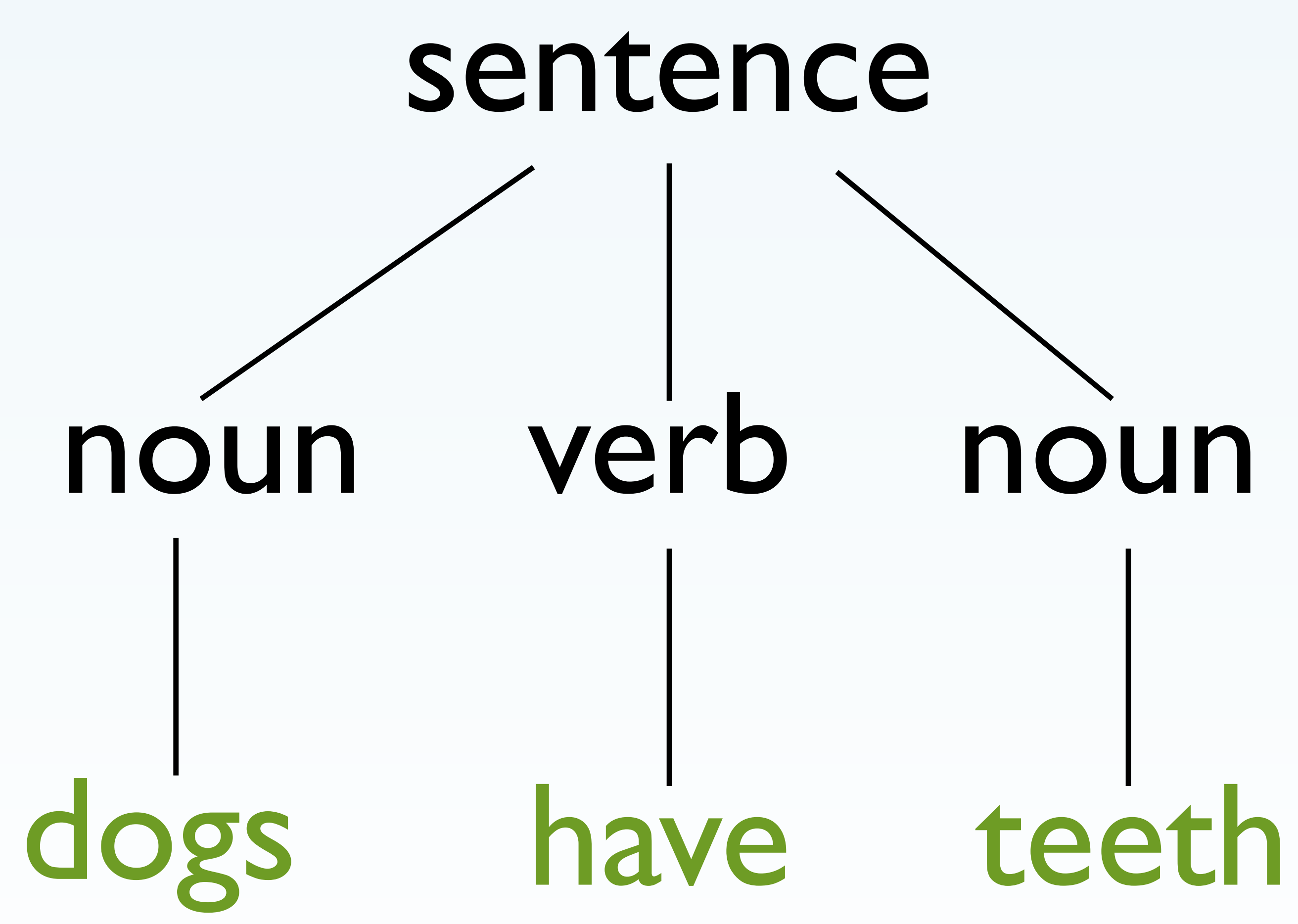
Abstract vs. Concrete Syntax

Representing Grammars by Haskell Data Types

Syntax Tree

A *syntax tree* is a structure to *represent* derivations.

Derivation is a *process* of producing a sentence according to the rules of a grammar.



sentence
noun verb noun
dogs verb noun
dogs have noun
dogs have teeth

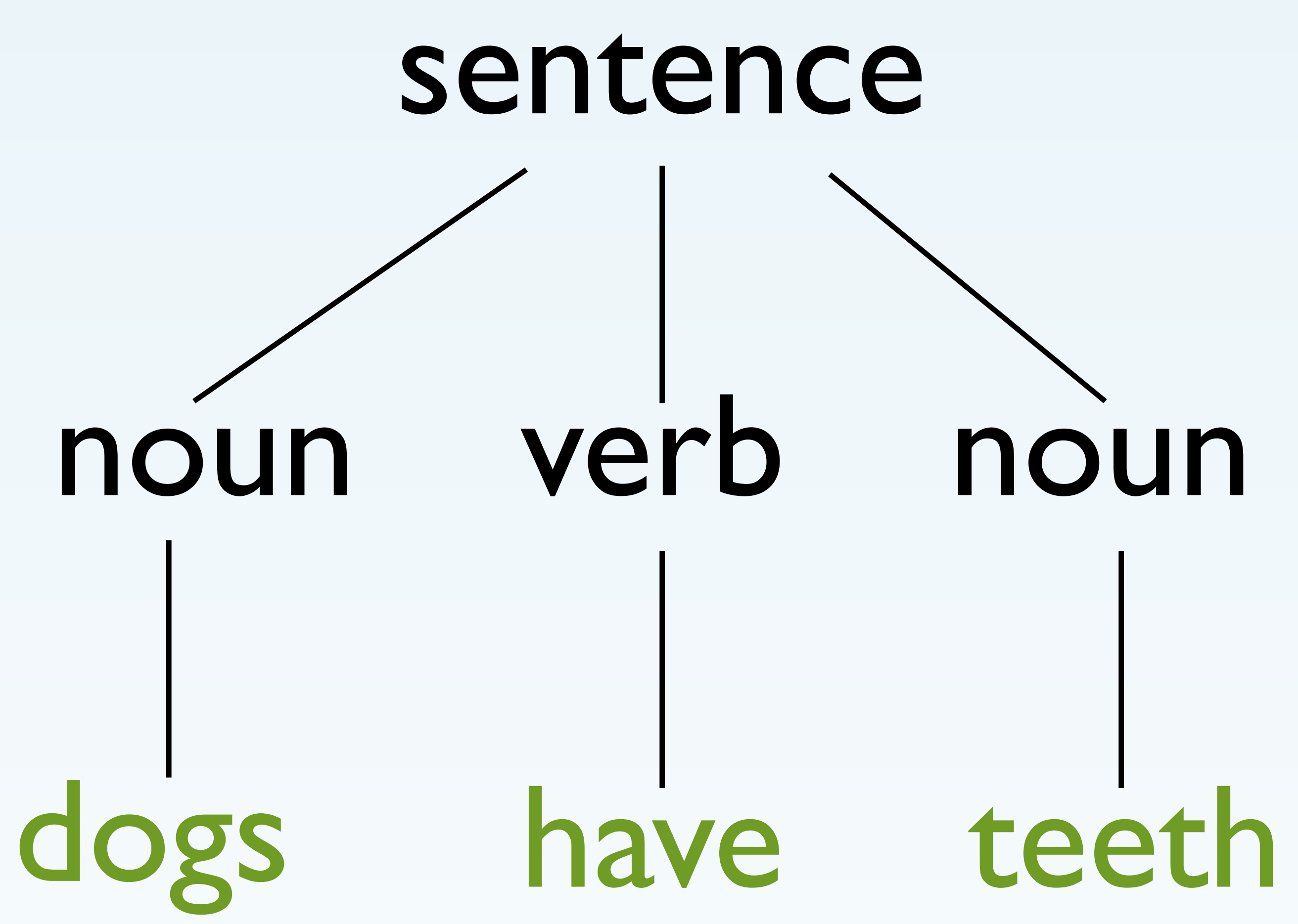
Observations About Syntax Trees

(1) Leaves contain *terminal symbols*

(2) Internal nodes contain *nonterminal symbols*

(3) Nonterminal in the root node indicates the *type* of the syntax tree

(4) Derivation order is *not* represented, which is a *Good Thing*, because the order is not important

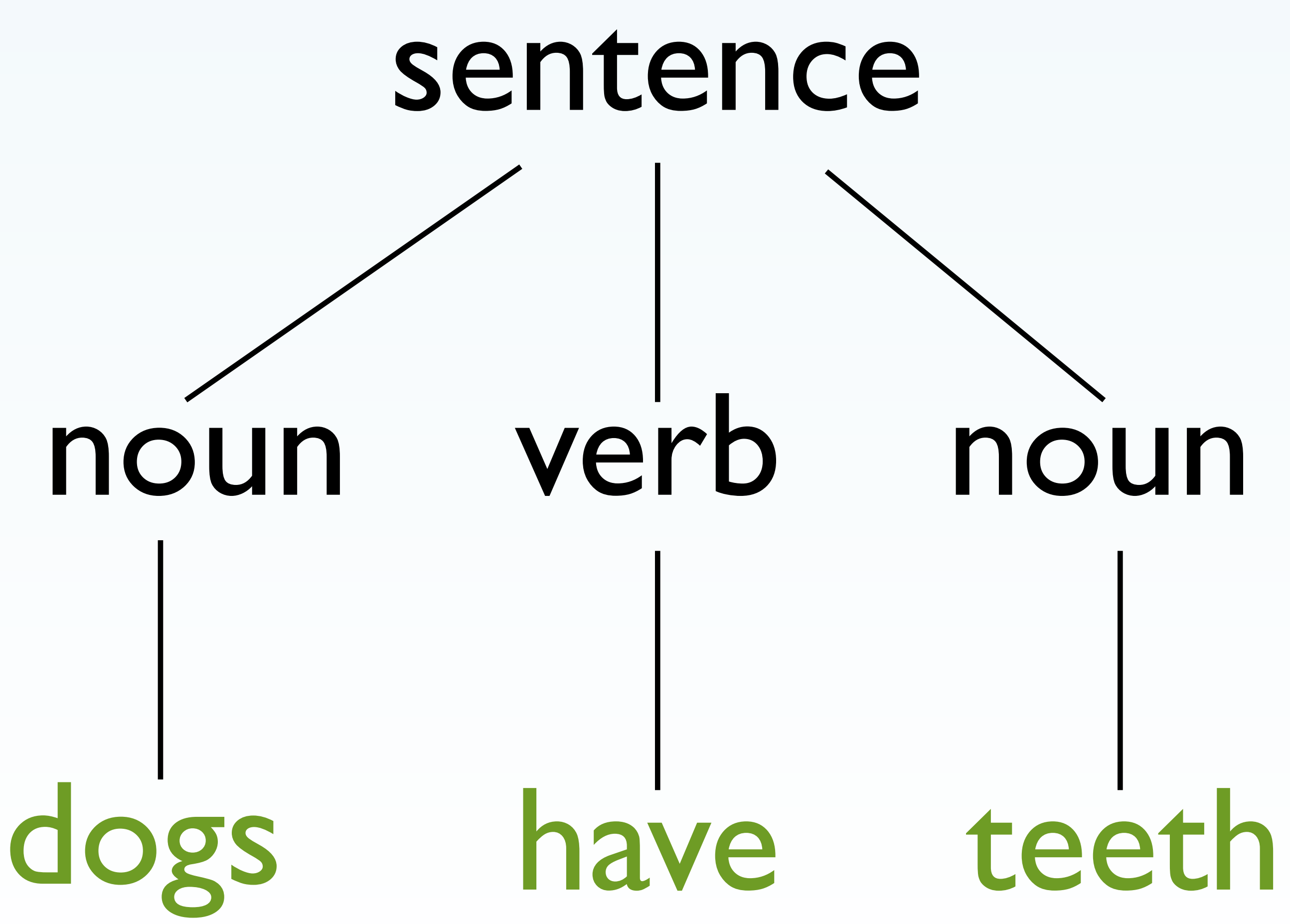


Alternative Representation

sentence	::=	noun verb noun	(R1)
sentence	::=	sentence and sentence	(R2)
noun	::=	dogs	(R3)
noun	::=	teeth	(R4)
verb	::=	have	(R5)

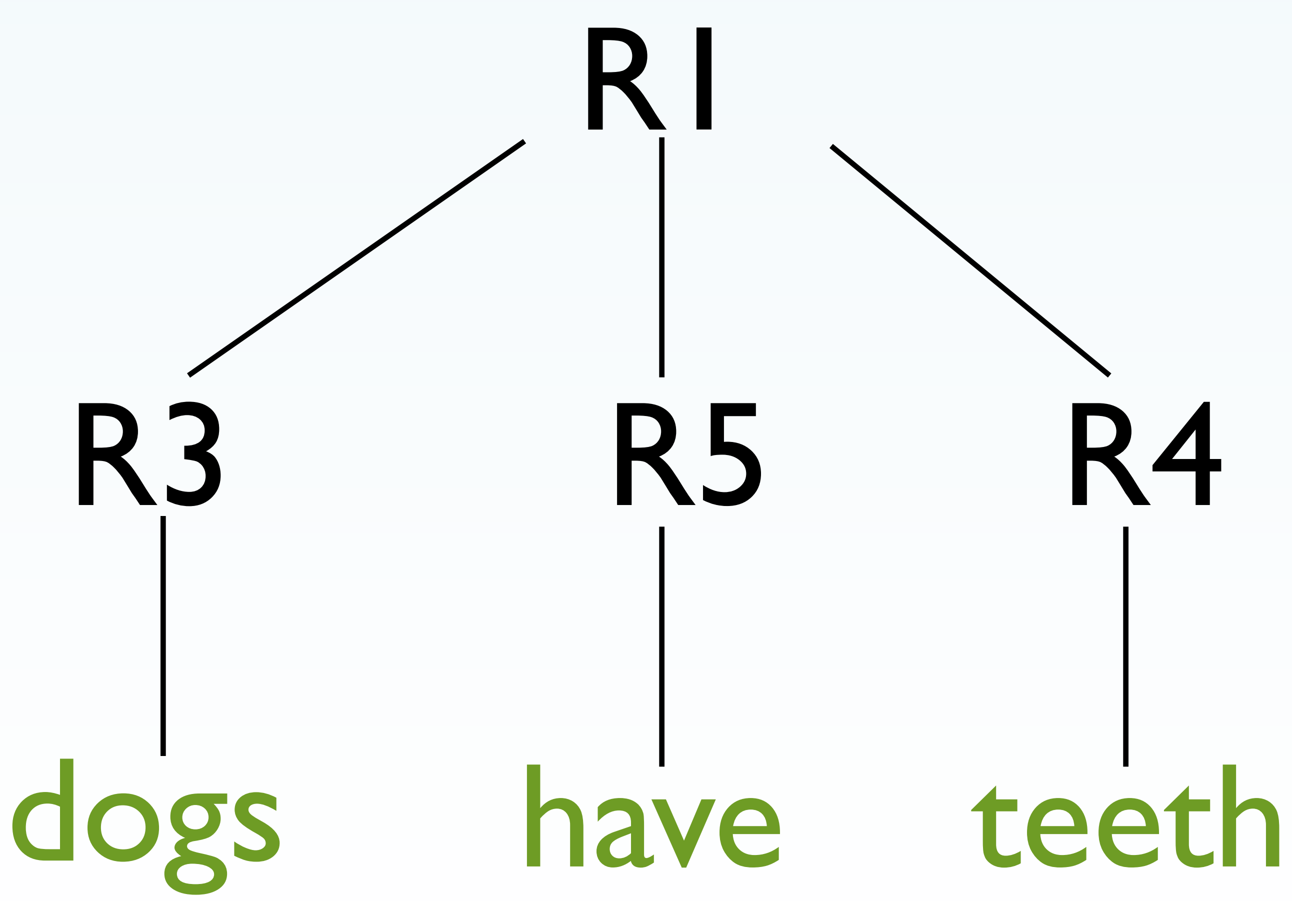
(1) Leaves contain *terminal symbols*

(2) Internal nodes contain *nonterminal symbols*



(1) Leaves contain *terminal symbols*

(2) Internal nodes contain *rule names*



Exercises

(1) Draw the syntax tree for the sentence 101

(2) Draw the syntax tree for the sentence not(not(F))

(3) Draw all syntax trees of type noun

(4) How many sentences/trees of type “stmt” can constructed with the following grammar?

```
cond  ::=  T
stmt  ::=  while cond do stmt
```

(5) How many with the following grammar?

```
cond  ::=  T
stmt  ::=  while cond do stmt
stmt  ::=  noop
```

Group Rules by LHS

```
sentence  ::=  noun verb noun             (R1)
            |  sentence and sentence      (R2)
noun      ::=  dogs | teeth               (R3, R4)
verb      ::=  have                       (R5)
```

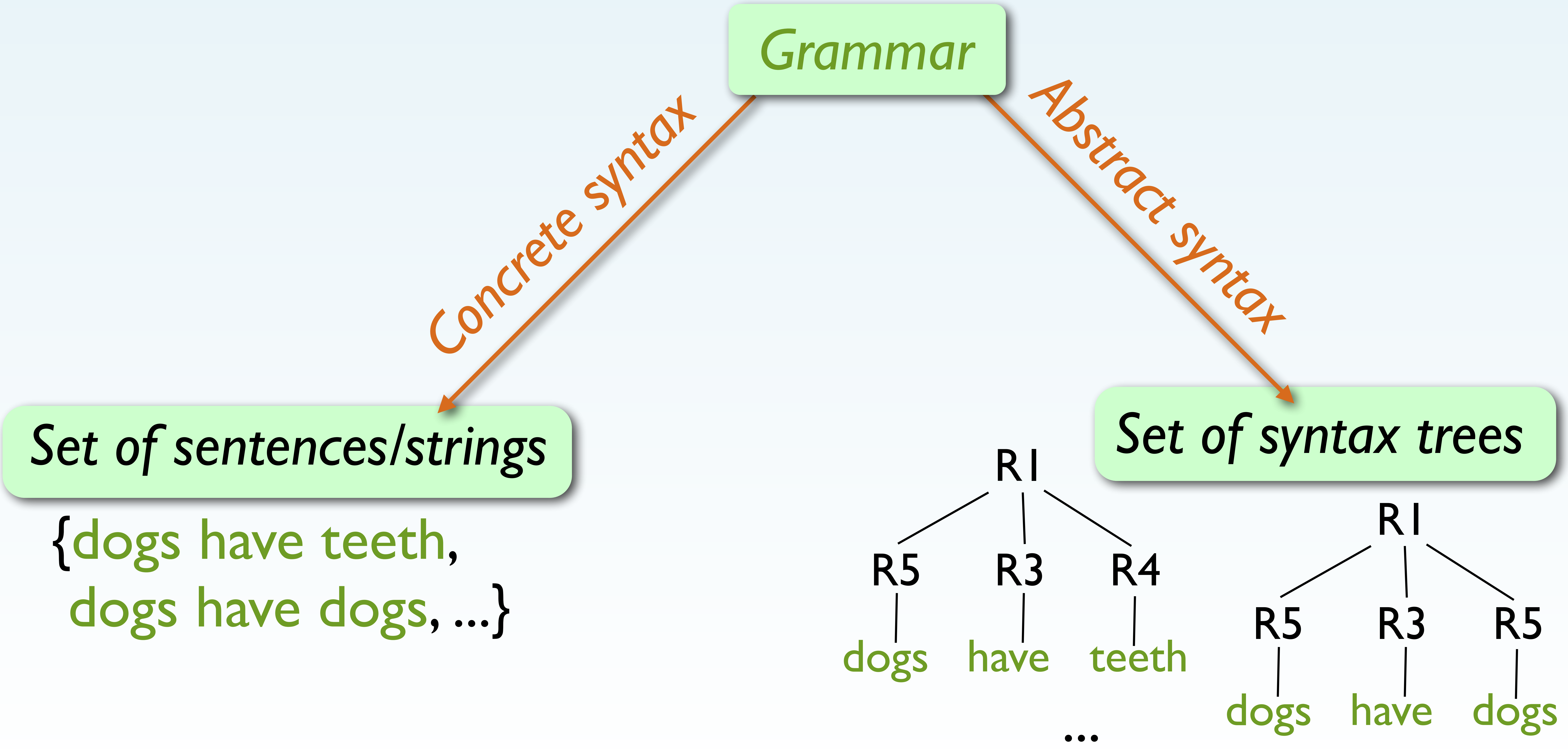
⇒ Grammar lists for each nonterminal all possible ways to construct a sentence of that kind.

Grammars can be defined in a modular fashion.

2 Syntax

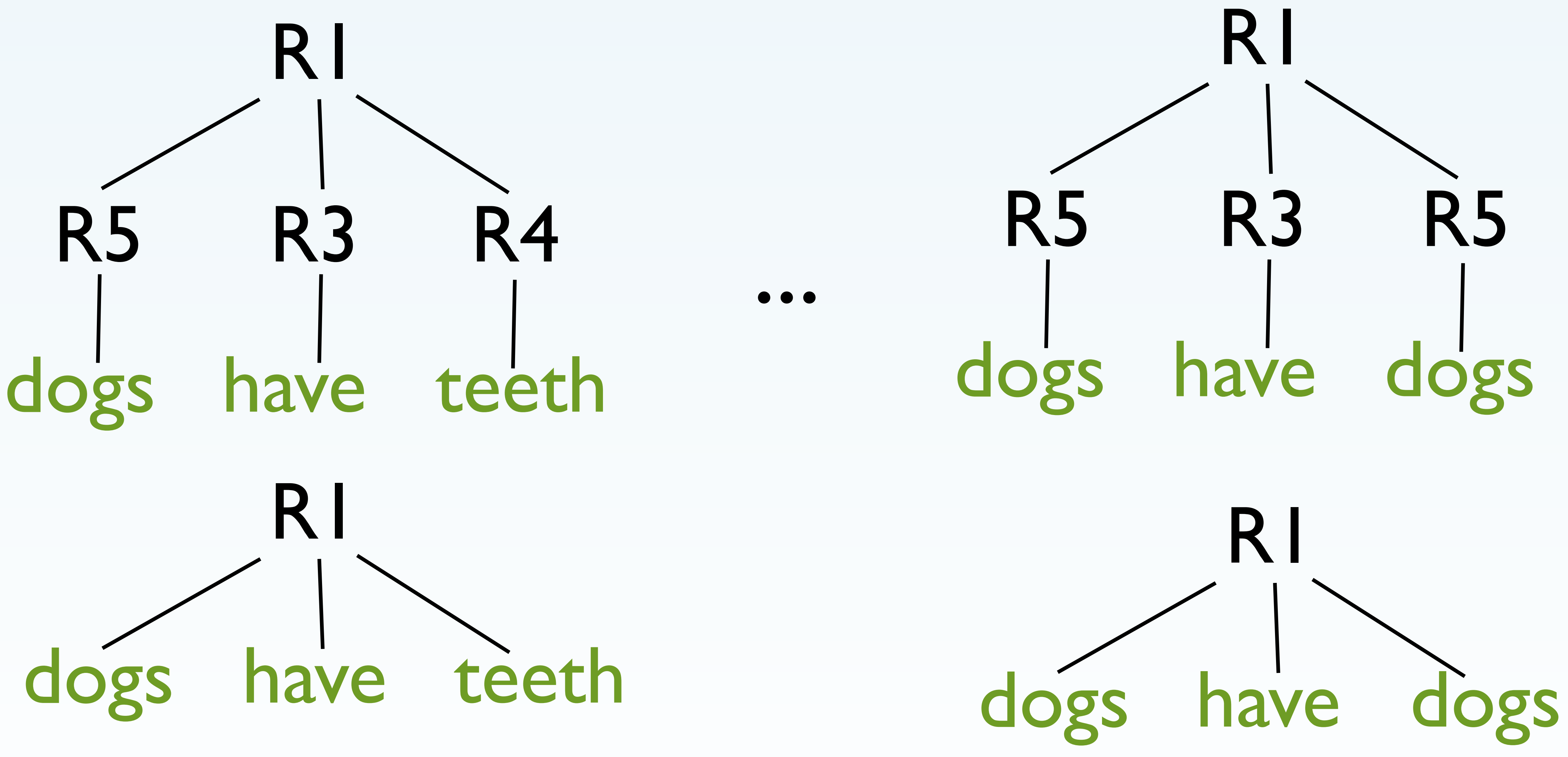
- Grammars & Derivation
- Syntax Tree
- Abstract vs. Concrete Syntax
- Representing Grammars by Haskell Data Types

Concrete vs. Abstract Syntax



Abstract Syntax

sentence	::=	noun verb noun	(R1)
	|	sentence and sentence	(R2)
noun	::=	dogs | teeth	(R3, R4)
verb	::=	have	(R5)



terminal symbols uniquely identify rules
(in this grammar)

How to Build Trees

top-down
- allocate space
- set record values
- subtrees are initially null and can be set later

bottom-up
- build subtrees first
- apply constructor to subtrees and values

```
public class Tree {
    Object val;
    Tree lft, rgt; }
```

```
public Tree (Object val, Tree lft, Tree rgt) {
   this.val = val;
   this.lft = lft;
   this.rgt = rgt; }
```

```
Tree left  = new Tree (new Integer(2), null, null);
Tree right = new Tree (new Integer(3), null, null);
Tree tree  = new Tree (new Integer(1), left, right);
```

Denoting Syntax Trees

sentence	::=	noun verb noun	(R1)
	|	sentence and sentence	(R2)
noun	::=	dogs | teeth	(R3, R4)
verb	::=	have	(R5)

Simple linear/textual representation:
Apply *rule names* to argument trees

R Tree-1 ... Tree-k

Use rule names as constructors:

R1 dogs have teeth

R2 (R1 dogs have teeth)
(R1 dogs have dogs)

Note: Parentheses are only used for linear notation of trees; they are not part of the abstract syntax

2 Syntax

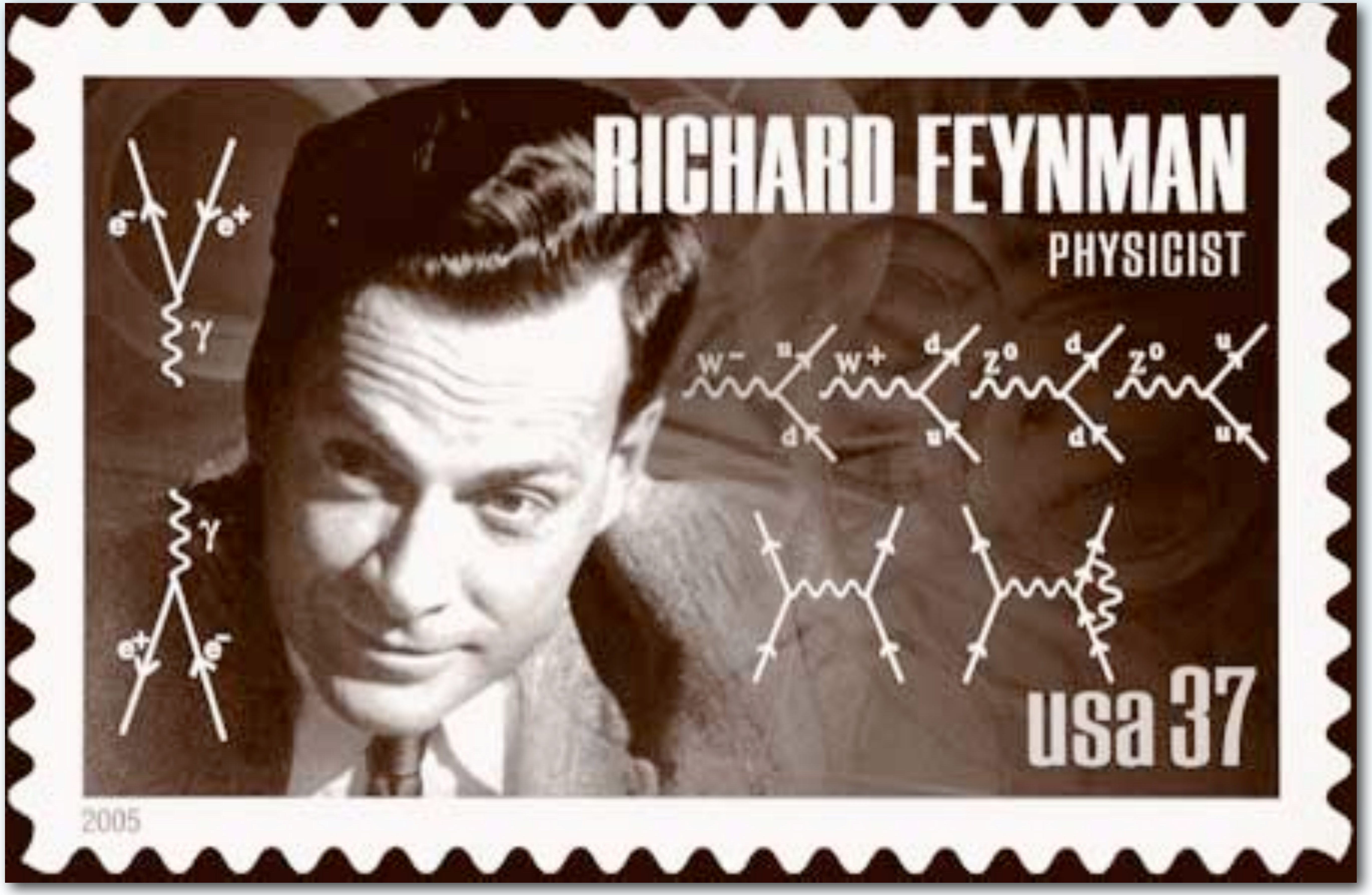
Grammars & Derivation

Syntax Tree

Abstract vs. Concrete Syntax

Representing Grammars by Haskell Data Types

Why Grammars in Haskell?



"What I cannot create, I do not understand."

Richard Feynman

Haskell Representation of Syntax Trees

Define a data type for each nonterminal
Define a constructor for each rule

```
sentence  ::=  noun verb noun
            |  sentence and sentence
noun      ::=  dogs | teeth
verb      ::=  have
```

```haskell
data Sentence = Phrase Noun Verb Noun
              | And Sentence Sentence
data Noun = Dogs | Teeth
data Verb = Have
```

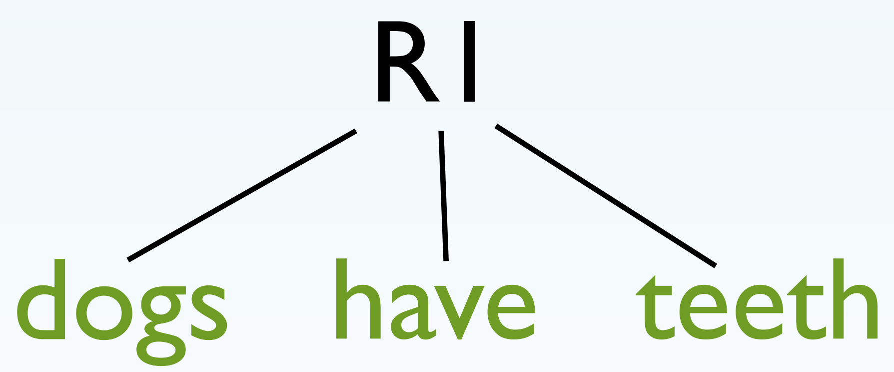
A syntax tree is represented by a Haskell value (built by data constructors)

To construct a syntax tree, apply a constructor to subtrees

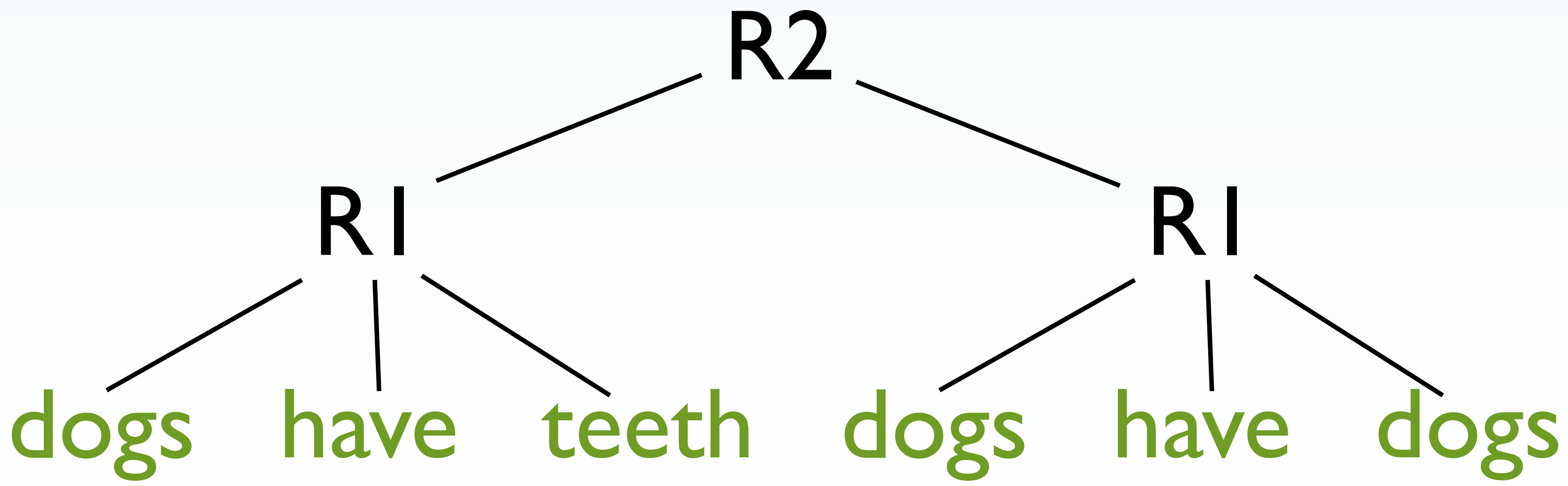
Haskell Representation of Syntax Trees

sentence	::=	noun verb noun	(R1)
	|	sentence and sentence	(R2)
noun	::=	dogs | teeth	(R3, R4)
verb	::=	have	(R5)

```haskell
data Sentence = R1 Noun Verb Noun
              | R2 Sentence Sentence
data Noun = Dogs | Teeth
data Verb = Have
```




```haskell
R1 Dogs Have Teeth
```

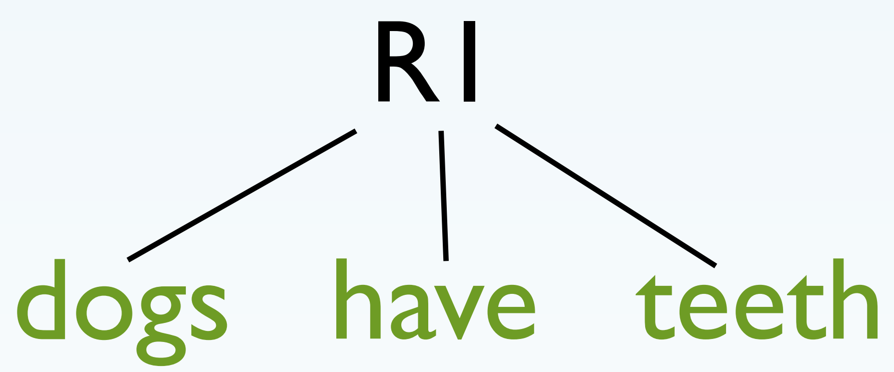



```haskell
R2 (R1 Dogs Have Teeth)
   (R1 Dogs Have Dogs)
```

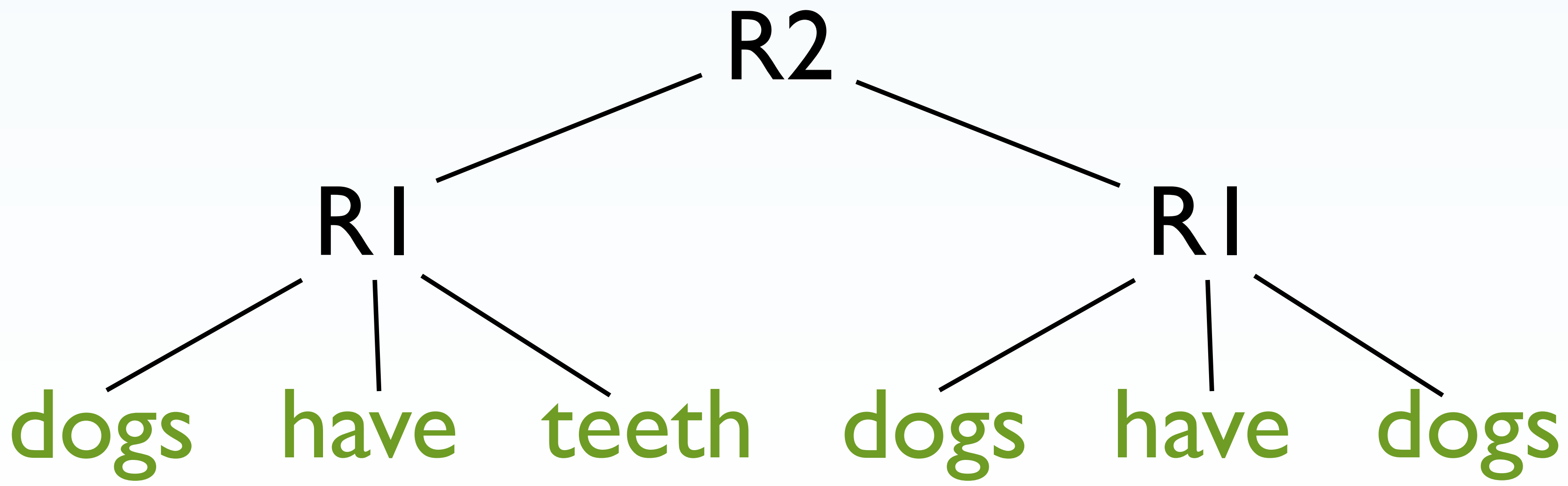
Haskell Representation of Syntax Trees


```
sentence  ::=  noun verb noun                (R1)
            |  sentence and sentence         (R2)
noun      ::=  dogs | teeth                  (R3, R4)
verb      ::=  have                          (R5)
```

```haskell
data Sentence = Phrase Noun Verb Noun
              | And Sentence Sentence
data Noun = Dogs | Teeth
data Verb = Have
```




```haskell
Phrase Dogs Have Teeth
```




```haskell
And (Phrase Dogs Have Teeth)
    (Phrase Dogs Have Dogs)
```

Haskell Demo ...

SentSyn.hs

Exercises

digit	::=	0	(R1)
digit	::=	1	(R2)
bin	::=	digit	(R3)
bin	::=	digit bin	(R4)

(1) Define a Haskell data type for binary numbers

(2) Represent the sentence 101 using constructors

(3) Define a Haskell data type for boolean expression including constants T and F and the operation not

(4) Represent the sentence not (not F)

(5) What is the type of `T` ?
What is the type of `Not T` ?
What is the type of `Not` ?
What is the type of `Not Not` ?

```
data BExpr = T | F | Not BExpr
```

More Exercises

(1) Define a Haskell data type for Peano-style natural numbers (constructed by 0 and successor)
(*Note*: numbers *cannot* be constructors.)

(2) Represent the sentence 3 using constructors

(3) Extend the number data type by constructors for representing addition and multiplication

(4) Represent the sentence 2*(3+1) using constructors

(5) Explain how the construction of syntactically incorrect sentences is prevented by Haskell's type system
(*Hint*: Try to build incorrect sentences)

Abstract Grammar

Abstract grammar contains:
(1) exactly one unique terminal symbol in each rule
(2) no redundant rules

Concrete grammar

cond ::= T | not cond | (cond)
stmt ::= while cond { stmt }
| noop

Haskell Data Type
=
Abstract grammar

```
data Cond = T | Not Cond
data Stmt = While Cond Stmt
          | Noop
```

Abstract Syntax Tree

Concrete grammar

```
cond ::= T | not cond | (cond)
stmt ::= while cond { stmt }
       | noop
```

```
while not(not(T)) {
  while T { noop }
}
```

Sentence

Haskell Data Type
=
Abstract grammar

```haskell
data Cond = T | Not Cond
data Stmt = While Cond Stmt
          | Noop
```

```haskell
While (Not (Not T))
      (While T Noop)
```

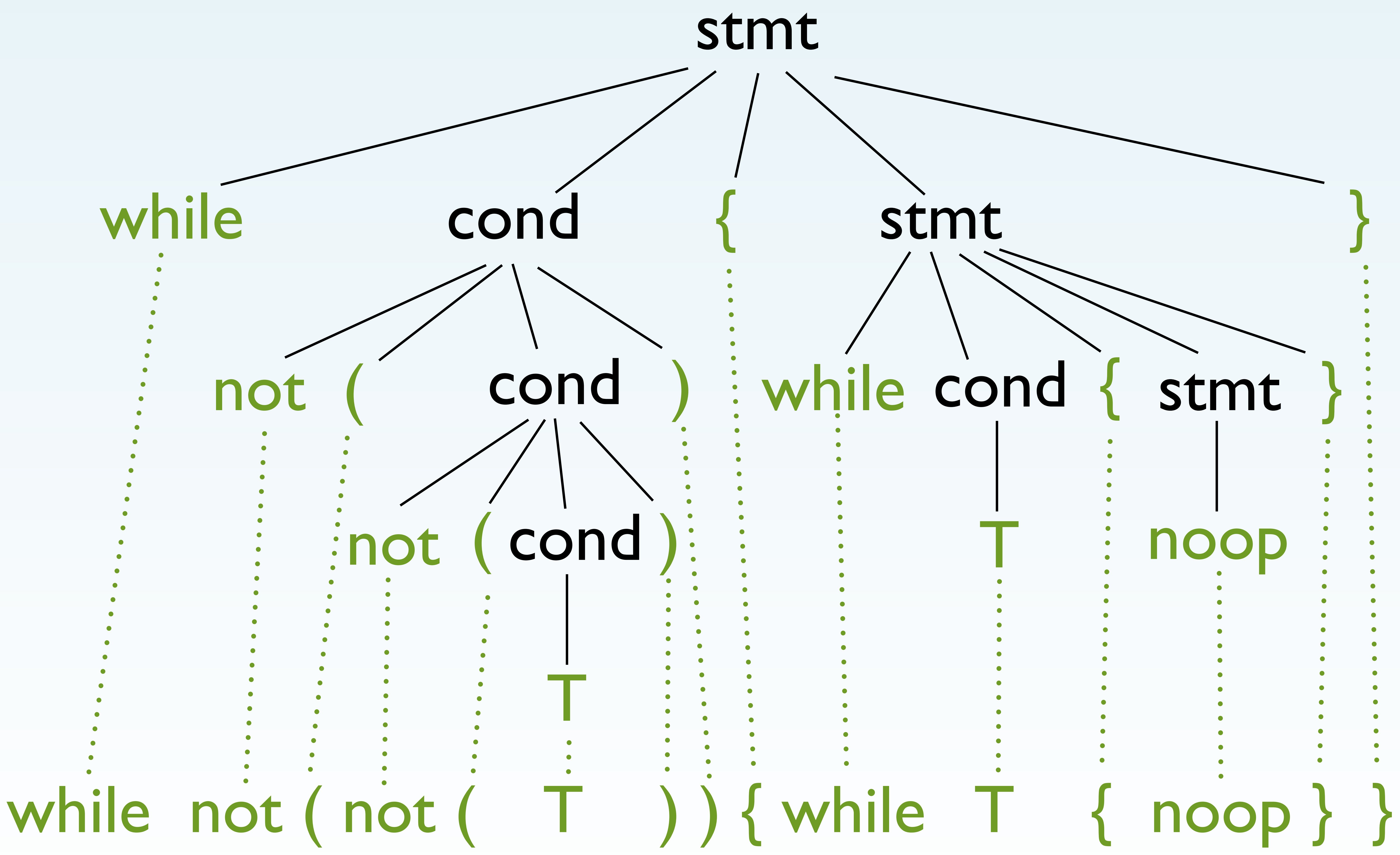
Abstract Syntax Tree
=
Value of Haskell Data Type

Exercises

(1) Draw the syntax tree for the following sentence

```
while not(not(T)) {
   while T { noop }
}
```

cond ::= T | not (cond)
stmt ::= while cond { stmt }
 | noop

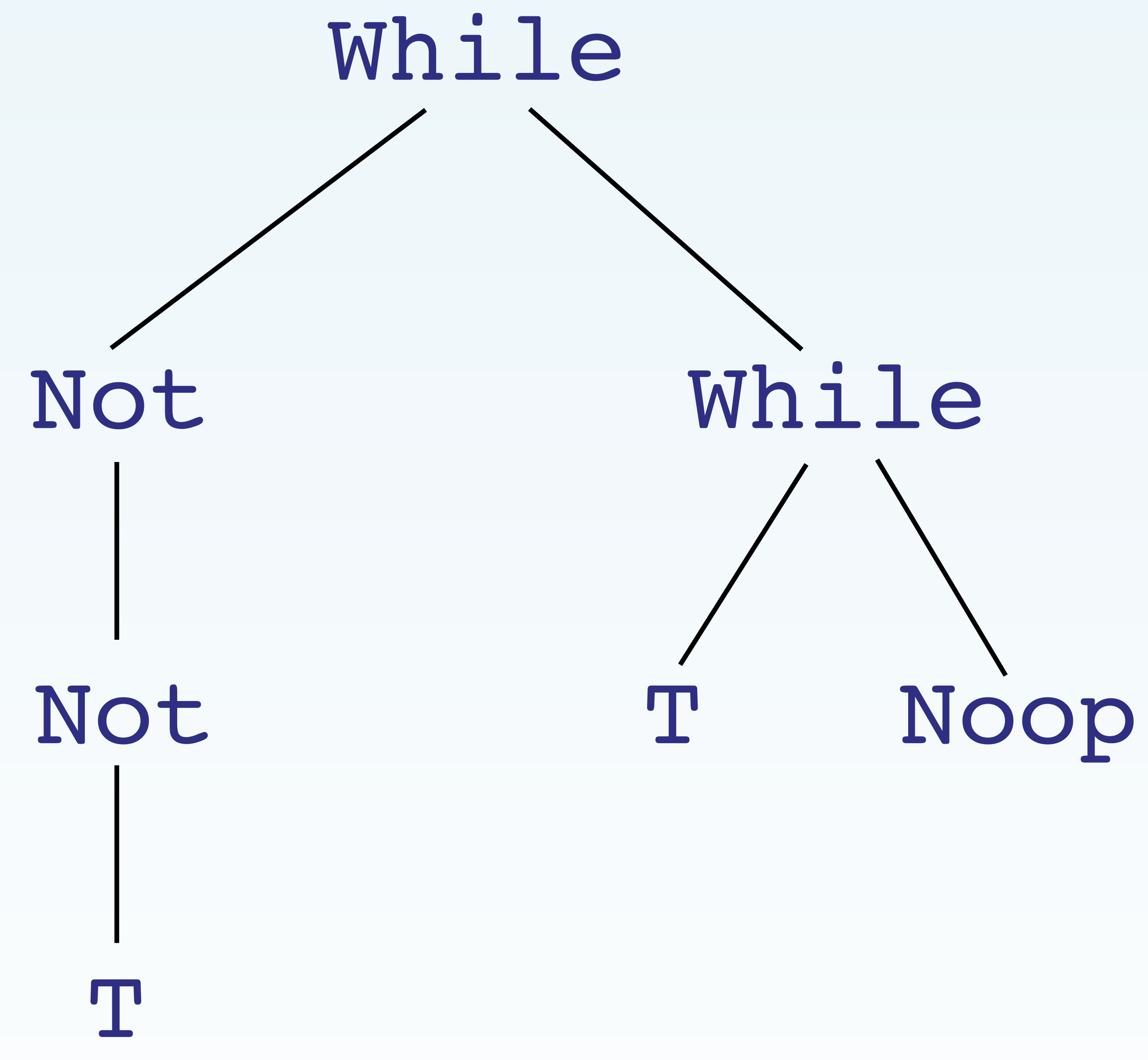


Exercises

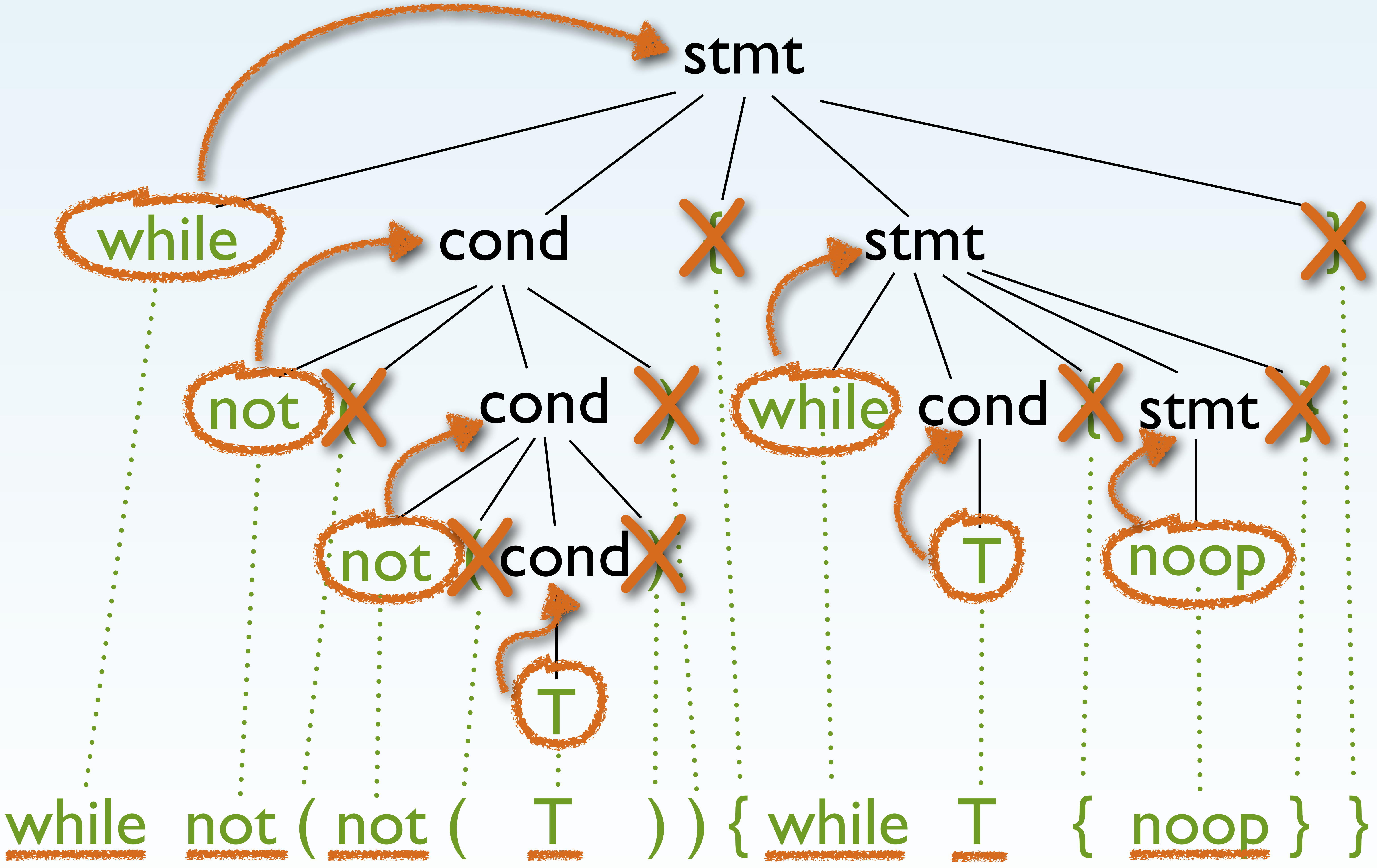
(2) Draw the following *abstract* syntax tree

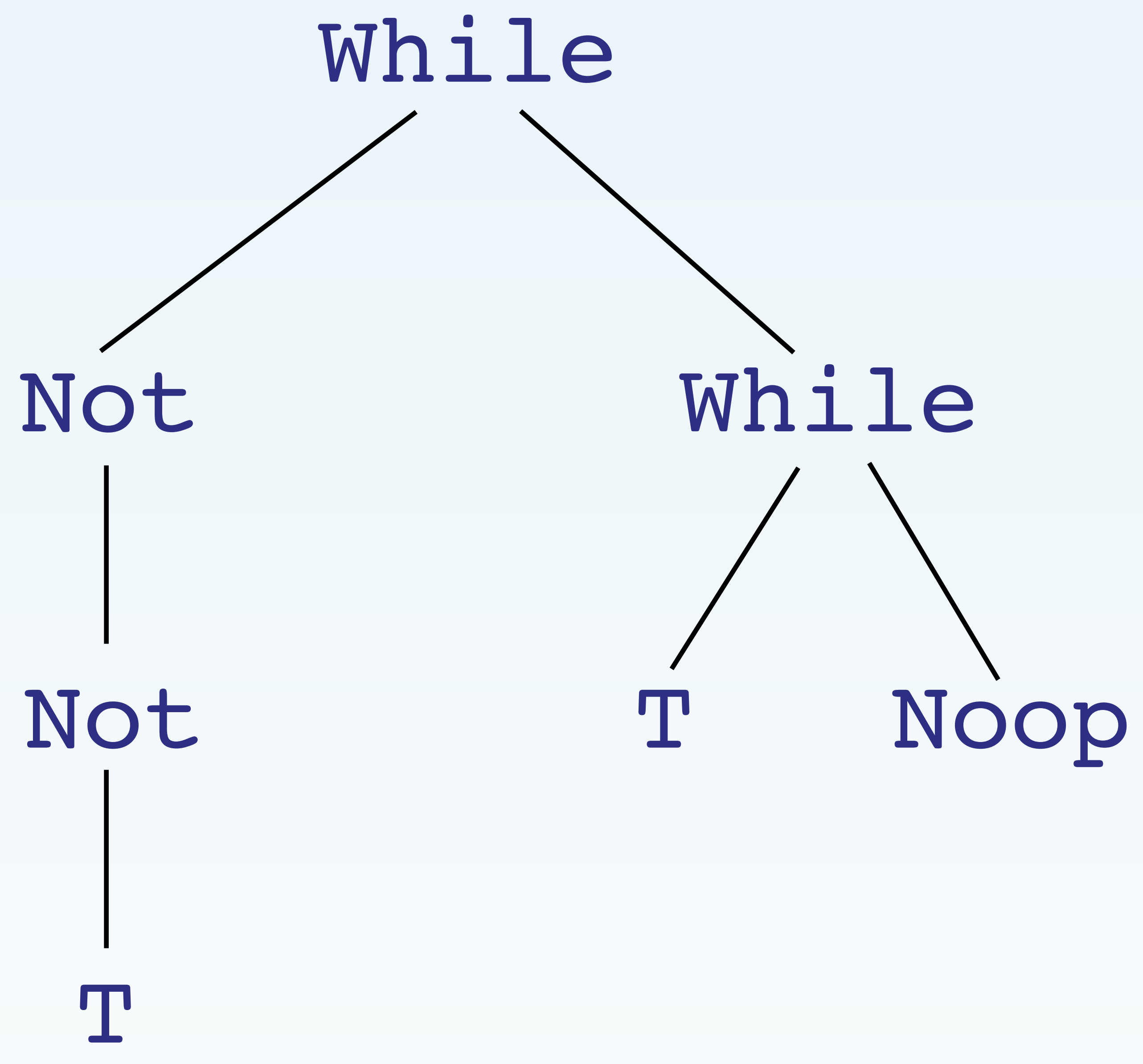
```
While (Not (Not T))
      (While T Noop)
```

```
data Cond = T | Not Cond
data Stmt = While Cond Stmt
          | Noop
```



From Concrete To Abstract Syntax






```
While (Not (Not T))
      (While T Noop)
```

Why Two Kinds of Syntaxes?

Abstract syntax:
- more concise
- represents essential language structure
- basis for analyses and transformations

```
data Cond = T | Not Cond
data Stmt = While Cond Stmt
          | Noop
```

Concrete syntax:
- more verbose and often better readable
- extra keywords and symbols help parser

```
cond ::= T | not cond | (cond)
stmt ::= while cond { stmt }
       | noop
```

Exercises

(1) Define abstract syntax for the following grammar

```
con  ::=  0 | 1
reg  ::=  A | B | C
op   ::=  MOV con TO reg
       |  MOV reg TO reg
       |  INC reg BY con
       |  INC reg BY reg
```

(2) Refactor grammar and abstract syntax by introducing a nonterminal to represent a con *or* a reg

(3) What are the elements of a Haskell data type definition?

Pretty Printing

A *pretty printer* creates a string from a syntax tree.

A *parser* extracts a syntax tree from a string.

CS 480

```
cond ::= T | not cond | (cond)
stmt ::= while cond { stmt }
       | noop
```

```
data Cond = T | Not Cond
data Stmt = While Cond Stmt
          | Noop
```

```
while not(not(T)) {
  while T { noop }
}
```

pretty printer

parser

```
While (Not (Not T))
      (While T Noop)
```

Haskell Demo ...

SentSyn.hs
SentPP.hs
BoolSyn.hs
BoolPP.hs

Exercise

(1) Define a pretty printer for the following abstract syntax

```
data Cond = T | Not Cond
data Stmt = While Cond Stmt
          | Noop
```

that produces output according to the following grammar

```
cond ::= T | not cond | (cond)
stmt ::= while cond { stmt }
       | noop
```

Haskell Demo ...

Stmt.hs

Grammar Rules for Lists

A string is a sequence of zero or more characters

Concrete syntax

string ::= char string | ϵ
char ::= a | b | c | …

aac

Abstract syntax

```
data Str = Seq Chr Str | Empty
data Chr = A | B | C | …
```

```
Seq A (Seq A (Seq C Empty))
```

Using built-in `Char` and list types

```
data Str = Seq [Char]
```

```
Seq ['a','a','c']
```

Using built-in `String` type

```
type String = [Char]
```

```
"aac"
```

```
['a','a','c']
```

Grammar Rules for Lists (2)

A number is a sequence of one or more digits

Concrete syntax

num ::= digit num | digit
digit ::= 1 | 2 | 3 | …

211

Abstract syntax

```
data Num = S Digit Num | D Digit
data Digit = One | Two | …
```

```
S Two (S One (D One))
```

Using built-in `Int` *and list types*

```
data Num = S [Int]
```

```
S [2,1,1]
```

Using built-in `Int` *type*

```
type Num = Int
```

```
211
```

Grammar Rules for Lists (3)

A qualified adjective is a list of adverbs followed by an adjective

adv* ::= adv adv* | ϵ

abbreviation

Concrete syntax

qadj ::= adv* adj
adv ::= really | frigging
adj ::= awesome | …

really really frigging awesome

Abstract syntax

```
data QAdj = Q [Adv] Adj
type Adv = String
type Adj = String
```

```
Q ["really","really","frigging"] "awesome"
```

Representing List in Abstract Grammars

Zero or more *A*'s

As ::= *A* *As* | ϵ
B ::= … *As* …

abbreviation →

B ::= … A^* …

Abstract syntax

```
data B = Con … [A] …
```

As ::= *A* *As* | *A*
B ::= … *As* …

abbreviation →

B ::= … A^+ …

One or more *C*'s

Data Types vs. Types

Type definitions just give names to type expressions, while Data definitions introduce constructors that build new objects

```
type Point = (Int,Int)

(3,5) :: Point
(3,5) :: (Int,Int)
```

```
data Point = Pt Int Int

Pt 3 5 :: Point
```

~~Pt 3 5 :: (Int,Int)~~

Design rule. Data definitions are used when:

- more than one constructor is needed
- pretty printing is required
- representation might be hidden (ADT)

Translating Grammars Into Data Types

(1) Represent each *basic nonterminal* by a *built-in type* (names, symbols, etc. by `String`, numbers by `Int`)

(2) For each *other nonterminal*, define a *data type*

(3) For each *production*, define a *constructor*

(4) *Argument types* of constructors are given by the production's *nonterminals*

```
❷exp  ::= ❶num | exp+exp | (exp)
stmt  ::= ❸while exp❹{ stmt }
        | noop
```

```haskell
data ❷Exp = N ❶Int | Plus Exp Exp
data Stmt = ❸While Exp Stmt❹
          | Noop
```

Note Carefully!

```
exp   ::=  num | exp+exp | (exp)
stmt  ::=  while exp { stmt }
        |  noop
```

- *Each case of a data type must have a constructor! (Even if no terminal symbol exists in the concrete syntax.)*

Constructor is indispensable!

```
data Exp = N Int | Plus Exp Exp
data Stmt = …
```

A perfectly valid alternative!

```
Plus (Exp,Exp)
```

- *Argument types of constructors may be grouped into pairs (or other type constructors).*

```
type EPair = (Exp,Exp)

data Exp = …
         | Plus EPair
         | Times EPair
```

--- END OF SLIDES ---

BACKUP SLIDES FOLLOW

Concept Summary