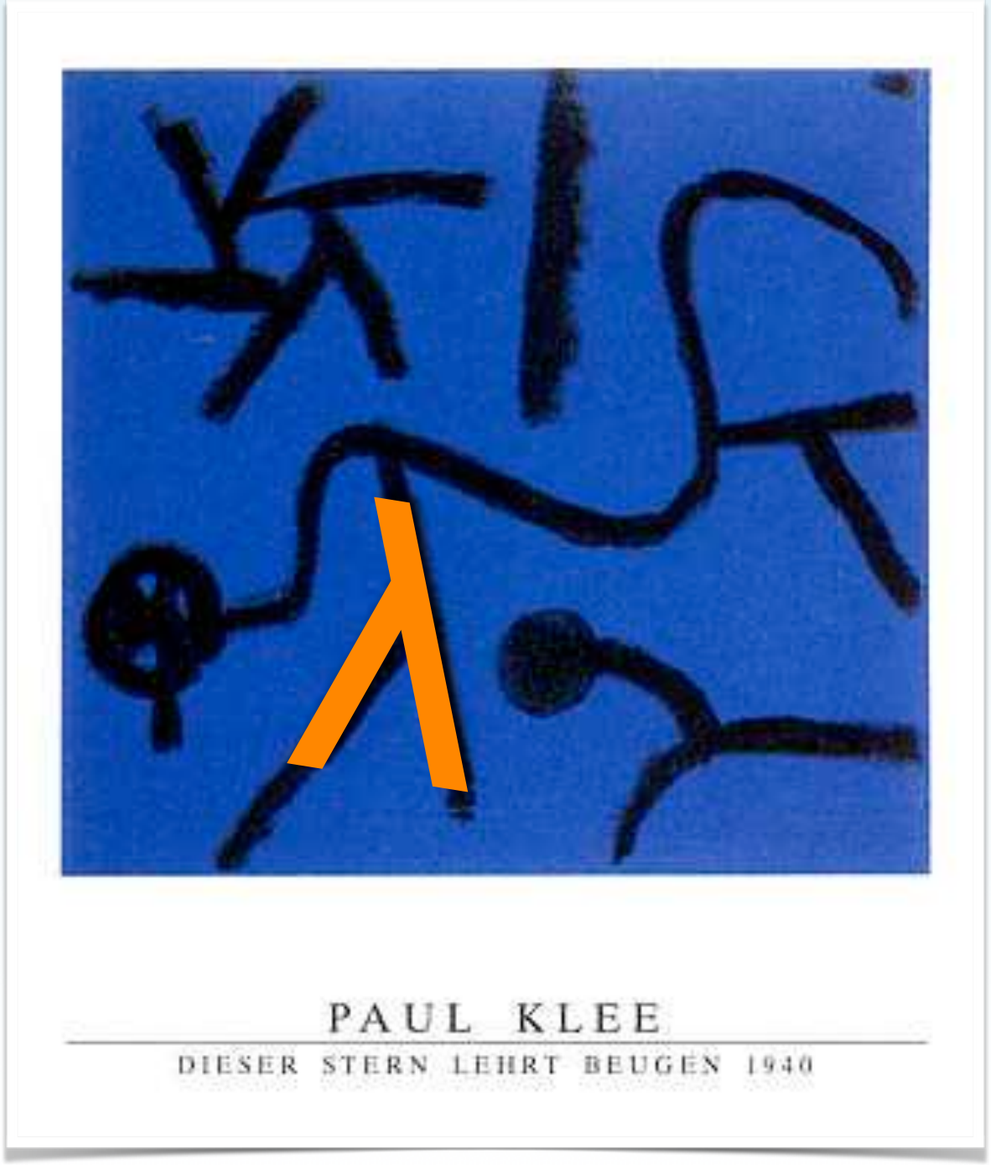
1 Haskell



Change vs. Description

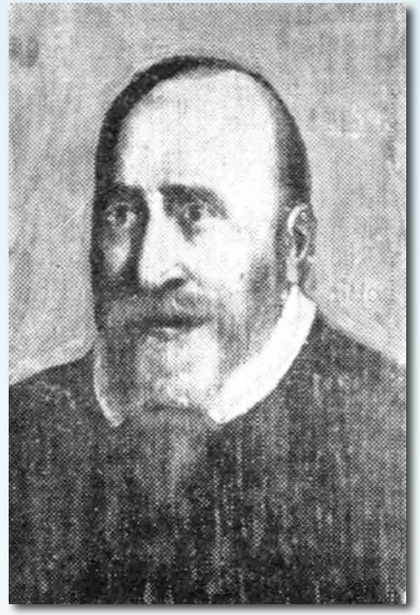
```
private void quicksort(int low, int high) {
     int i = low, j = high;
     int pivot = numbers[low + (high-low)/2];

     while (i <= j) {
          while (numbers[i] < pivot) {
               i++;
          }
          while (numbers[j] > pivot) {
               j--;
          }
          if (i <= j) {
               exchange(i, j);
               i++;
               j--;
          }
     }
     if (low < j)
          quicksort(low, j);
     if (i < high)
          quicksort(i, high);
}

private void exchange(int i, int j) {
     int temp = numbers[i];
     numbers[i] = numbers[j];
     numbers[j] = temp;
}
```

Quicksort in Java

invented the "=" sign in 1557



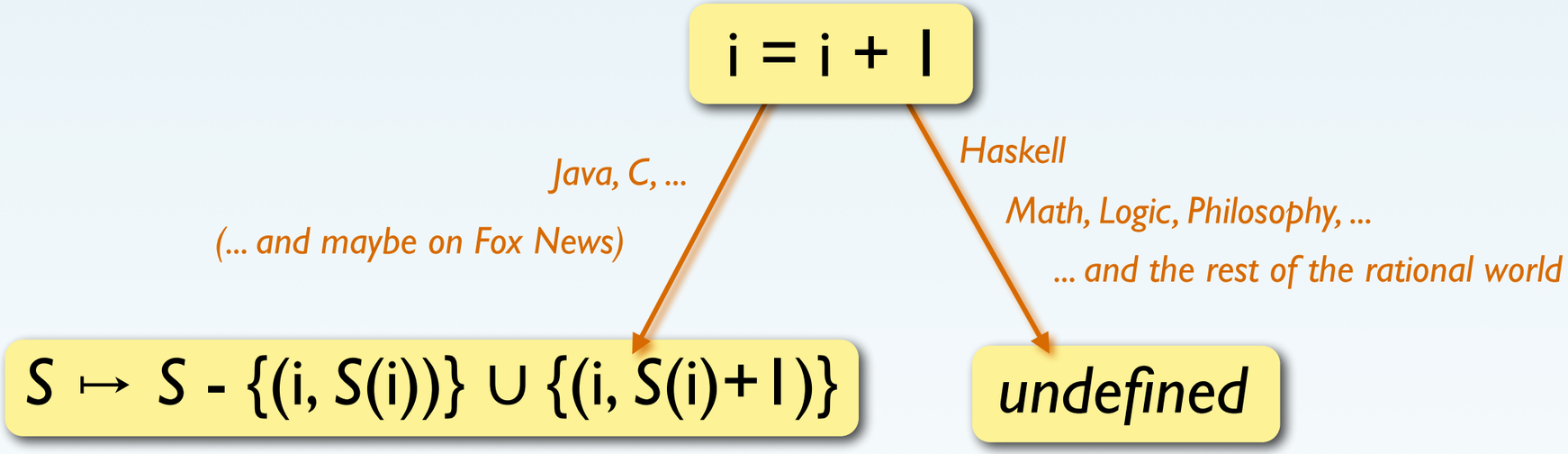
Robert Recorde

Same symbol – completely different meaning!

```
qsort :: Ord a ⇒ [a] → [a]
qsort []     = []
qsort (x:xs) = qsort [y | y←xs, y<=x] ++ [x] ++
               qsort [y | y←xs, y>x]
```

Quicksort in Haskell

The Meaning of "="



$i_{new} = i_{old} + 1$

In Haskell:
No state, No assignment!

(There are monads ...)

Computation in Haskell

$S \rightarrow T$

Function that maps values of type S to values of type T

Description of Computation:
Equations of how to construct output from input

Example: reversing a list

`[Int]` : *list of Int*
`[a]` : *list of anything*

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Concatenate 2 lists

list of single element `x`

Empty list

Nonempty list with first element and `x` *rest list* `xs`

Zoom Poll

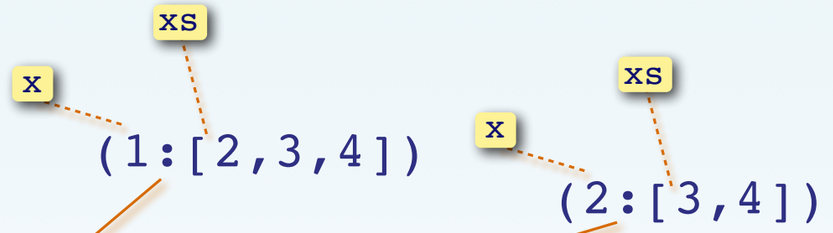
About Recursion

Substituting Equals for Equals

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Pattern Matching:
(1) Conditional
(2) Bindings

```
(:)  ::  a  → [a] → [a]
(++) :: [a] → [a] → [a]
```




```
reverse [1,2,3,4] =
reverse [2,3,4] ++ [1] =
reverse [3,4] ++ [2] ++ [1] =
reverse [4] ++ [3] ++ [2] ++ [1] =
reverse [] ++ [4] ++ [3] ++ [2] ++ [1] =
[] ++ [4] ++ [3] ++ [2] ++ [1] =
[4,3,2,1]
```

Learning Haskell: Baby Steps

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions (Type Signatures, Parameters, Equations)*
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types (Constructors, Pattern Matching)*

Exercises

Write an expression that computes the larger value of 5 and 6 (don't use the function max)

*Write an expression that tests whether 2*3 is equal to 3+3*

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Exercises

Define a function that tests whether a number is positive using pattern guards

```
positive :: Int → Bool
positive x = if x>=0 then True else False
```

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Exercises

Define a function for computing Fibonacci numbers

Define a function that tests whether a number is even

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Exercises

```
(:)  ::  a  → [a] → [a]
(++) :: [a] → [a] → [a]
```

```
head :: [a] → a
head (x:_) = x
```

```
tail :: [a] → [a]
tail (_:xs) = xs
```

```
sum :: [Int] → Int
sum []     = 0
sum (x:xs) = x + sum xs
```

Define the function `length :: [a] → Int`

Evaluate the expressions that don't contain an error

```
xs = [1,2,3]
```

```
sum xs + length xs
xs ++ length xs
xs ++ [length xs]
[sum xs, length xs]
[xs, length xs]
```

```
5:xs
xs:5
[tail xs,5]
[tail xs,[5]]
tail [xs,xs]
```

Zoom Poll

Haskell list functions

From Iteration to Recursion

f

```
result = Start
while Condition[argument] do {
  result = Update[result,argument]
  argument = Simplify[argument]
}
```

```
f argument | Condition[argument] = Update[f(Simplify[argument]),argument]
           | otherwise            = Start
```

fac

```
result = 1
while argument > 1 do {
  result = result * argument
  argument = argument - 1
}
```

```
fac argument | argument > 1 = fac(argument - 1) * argument
             | otherwise    = 1
```

```
fac 1 = 1
fac n = n * fac (n-1)
```

From Iteration to Recursion

f

```
result = Start
while Condition[argument] do {
  result = Update[result,argument]
  argument = Simplify[argument]
}
```

```
f argument | Condition[argument] = Update[f(Simplify[argument]),argument]
           | otherwise            = Start
```

length

```
result = 0
while not (null (list)) do {
  result = result + 1
  list = list.next
}
```

```
length list | not (null list) = length (tail list) + 1
            | otherwise       = 0
```

```
length []     = 0
length (_:xs) = 1 + length xs
```

3 Ways to Define Functions

Recursion

```
sum :: [Int] → Int
sum xs = if null xs then 0
         else head xs + sum (tail xs)
```

```
head :: [a] → a
head (x:_) = x
```

```
tail :: [a] → [a]
tail (_:xs) = xs
```

Pattern Matching

(1) *Case analysis*

```
sum :: [Int] → Int
sum []     = 0
sum (x:xs) = x + sum xs
```

(2) *Data decomposition*

Higher-Order Functions

```
sum :: [Int] → Int
sum = foldr (+) 0
```

variables & recursion not needed!

Learning Haskell

- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Higher-Order Functions

HOFs ≡ Control Structures

```
map f [x1, ..., xk] = [f x1, ..., f xk]
```

```
map :: (a → b) → [a] → [b]
map f []     = []
map f (x:xs) = f x:map f xs
```

Loop for processing elements independently

```
fold f u [x1, ..., xk] = x1 `f` ... `f` xk `f` u
```

Loop for aggregating elements

```
fold :: (a → b → b) → b → [a] → b
fold f u []     = u
fold f u (x:xs) = x `f` (fold f u xs)
```

```
sum = fold (+) 0
fac n = fold (*) 1 [2 .. n]
```

Higher-Order Functions

Function composition

```
(f . g) x = f (g x)
```

```
(.) :: (b → c) → (a → b) → a → c
f . g = \x → f (g x)
```

```
plus2 = succ . succ
odd = not . even
snd = head . tail
drop2 = tail . tail
```

```
succ :: Int → Int
even :: Int → Bool
not :: Bool → Bool
head :: [a] → a
tail :: [a] → [a]
```

Exercises

Is the function `th` *well defined?*
If so, what does it do and what is its type?

```
th :: ?
th = tail . head
```

```
(.) :: (b → c) → (a → b) → a → c
```

```
head :: [a] → a
head (x:_) = x
```

```
tail :: [a] → [a]
tail (_:xs) = xs
```

What does the expression `map f . map g` *compute?*
How can it be rewritten?

Exercises

Implement `revmap` *using pattern matching*

```
map :: (a → b) → [a] → [b]
map f []     = []
map f (x:xs) = f x:map f xs
```

```
reverse :: [a] → [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Implement `revmap` *using function composition*

```
(.) :: (b → c) → (a → b) → a → c
```

Exercises

Find expressions to …

```
xs = [1,2,3]
ys = [xs,[7]]
```

… *increment elements in xs by 1*

… *increment elements in ys by 1*

… *find the last element in xs*

Define the function

```
last :: [a] → a
```

Evaluate all the expressions that don't contain an error

```
map sum xs
map sum ys
last ys
map last ys
last (last ys)
```

Learning Haskell

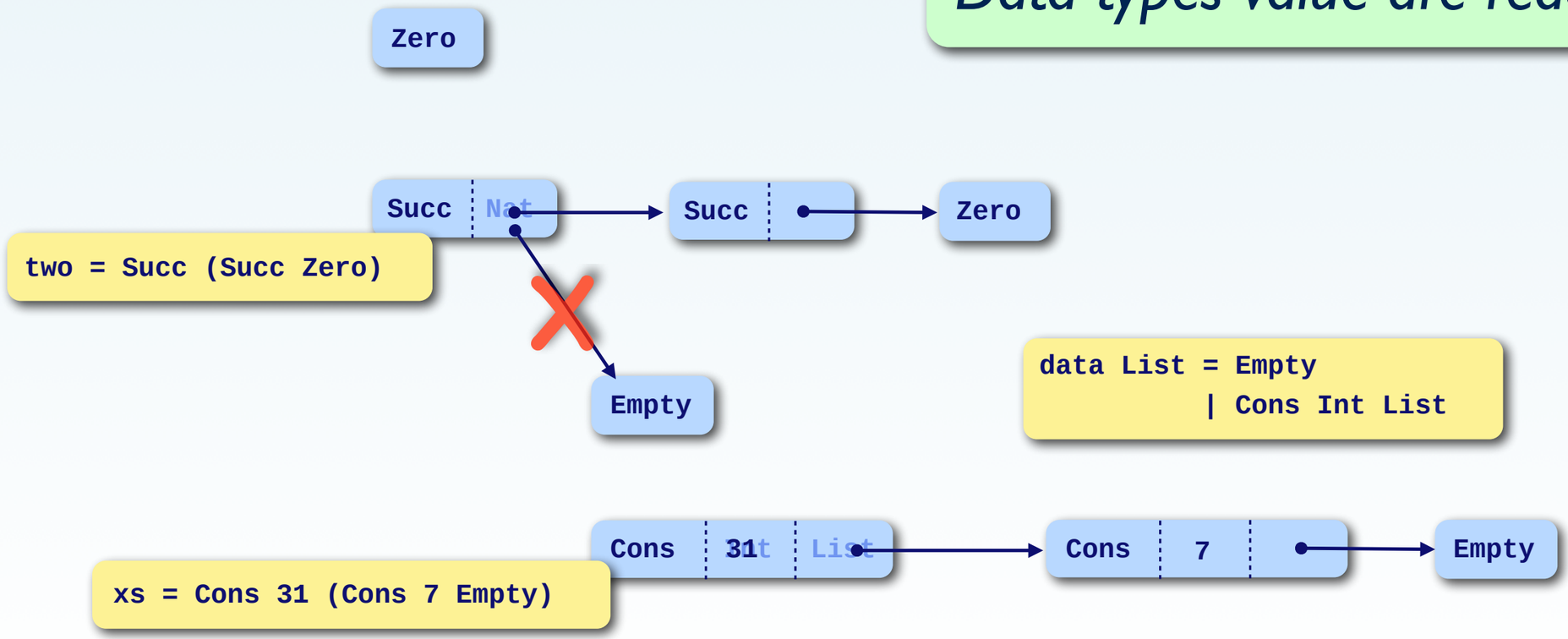
- *Values* and *Basic Types*
- *Expressions* (applying functions to values and expressions)
- *Function Definitions* (*Type Signatures*, *Parameters*, *Equations*)
- *Pattern Guards*
- *Recursion*
- *Lists* and *Pattern Matching*
- *Higher-Order Functions*
- *Data Types* (*Constructors*, *Pattern Matching*)

Data Constructors

≈ C++/Java object constructor, but:
(1) Inspection by pattern matching!
(2) Immutable!

Data types value are read-only

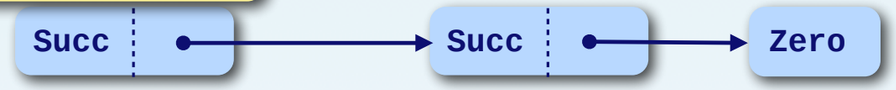
```
data Nat = Zero
         | Succ Nat
```



More on Data Constructors

```
data Nat = Zero
         | Succ Nat
```

```
two = Succ (Succ Zero)
```

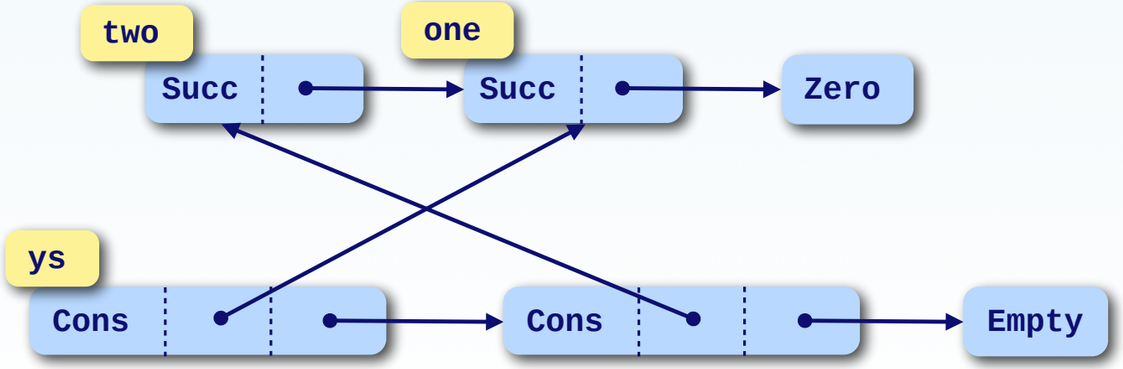



```
data List = Empty
          | Cons Int List
```

```
xs = Cons 1 (Cons 2 Empty)
```


```
data List = Empty
          | Cons Nat List
```

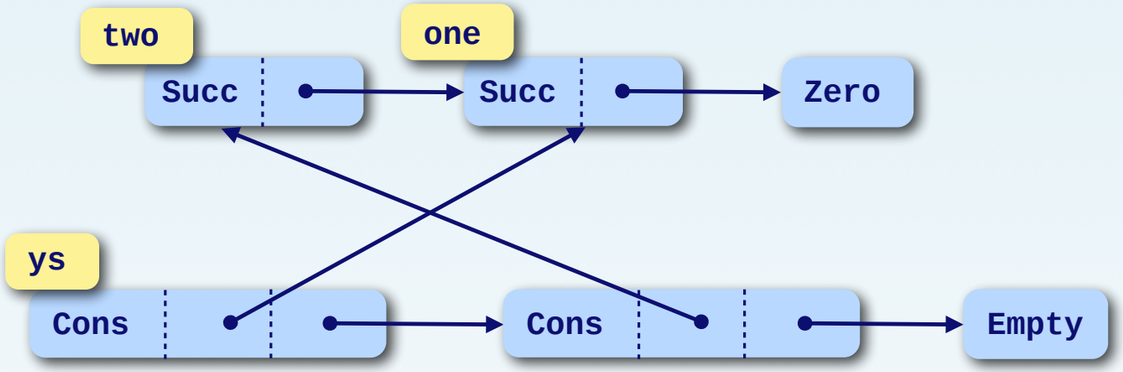
```
one = Succ Zero
two = Succ one
ys = Cons one (Cons two Empty)
```



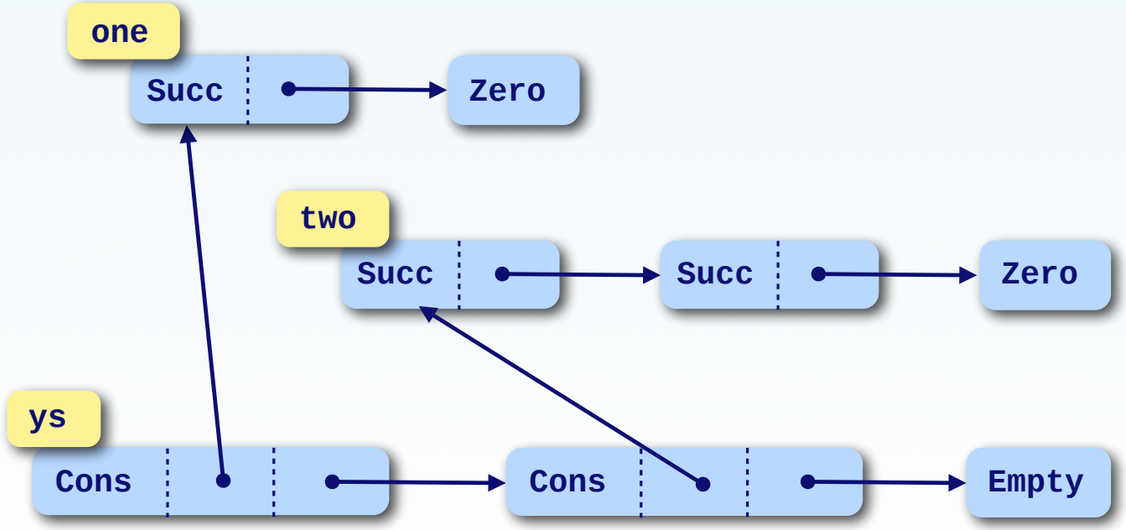
Avoiding Sharing

```
data List = Empty
          | Cons Nat List
```

```
one = Succ Zero
two = Succ one
ys = Cons one (Cons two Empty)
```




```
one = Succ Zero
two = Succ (Succ Zero)
ys = Cons one (Cons two Empty)
```



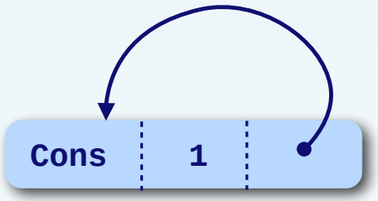
Cyclic Data Structures

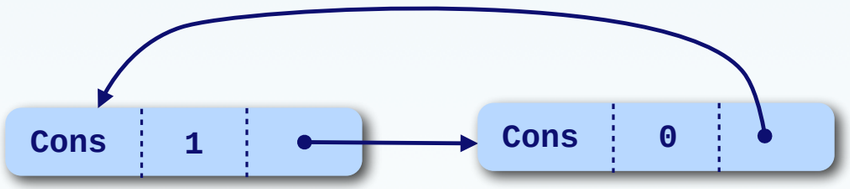
```
data List = Empty
          | Cons Int List
```

```
xs = Cons 1 (Cons 2 Empty)
```

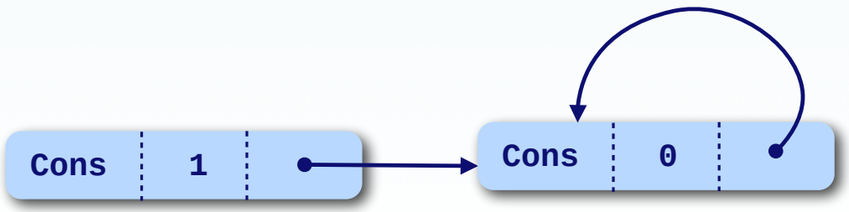
Intensional description of an infinite list

```
ones = Cons 1 ones
morse = Cons 1 (Cons 0 morse)
```






```
zeros = Cons 0 zeros
big = Cons 1 zeros
```

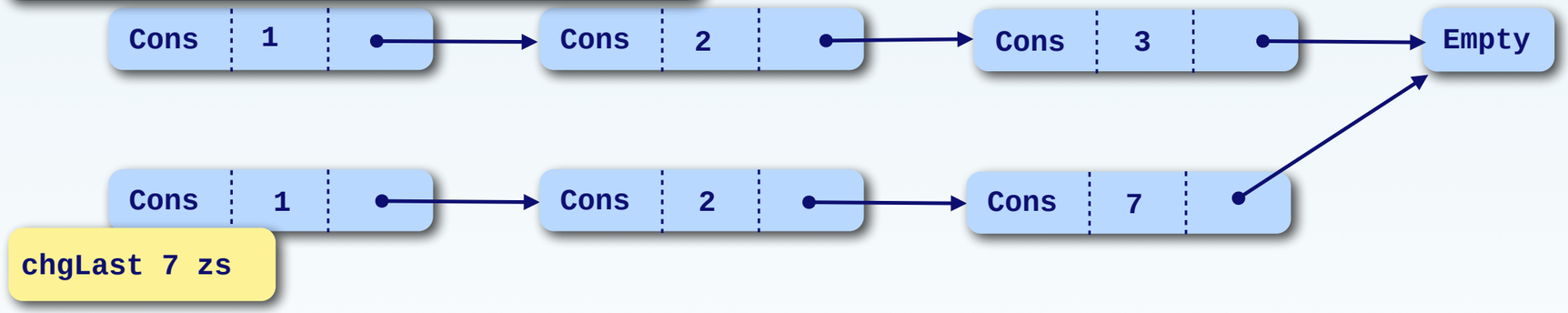


Changing Data Structures

```
data List = Empty
          | Cons Int List
```

Data types value are read-only

```
zs = Cons 1 (Cons 2 (Cons 3 Empty))
```




```
chgLast :: Int → List → List
chgLast y [x]    = [y]
chgLast y (x:xs) = x:chgLast y xs
```

Summary: Haskell so far

- Functions (vs. state manipulation)
- No side effects
- Higher-Order Functions (i.e. flexible control structures)
- Recursion
- Data Types (constructors *and* pattern matching)
- More Haskell features:
  list comprehensions, *where blocks*,

“Curried” Dinners are More Spicy

```
Experience dinner(Drink d, Entree e, Dessert f){…}
```

```
dinner(wine, pasta, pie)
```

Must provide all arguments at once


```
dinner :: Drink → Entree → Dessert → Experience
```

```
dinner wine :: Entree → Dessert → Experience
```

Partial function application is possible

Currying

Curry-Howard Isomorphism

The values of a type are the proofs for the proposition represented by it.

Proof for Proposition

`Program : Type`

```
even :  Int → Bool
sort :  List → SortedList
```

```
(a,b) → c
```

$A \wedge B \Rightarrow C$

$(A \wedge B) \Rightarrow C$

$\neg(A \wedge B) \vee C$

$\neg A \vee \neg B \vee C$

$A \Rightarrow (\neg B \vee C)$

$A \Rightarrow (B \Rightarrow C)$

```
a → b → c
```

≡

```
a → (b → c)
```

Haskell?

Research in PL ?

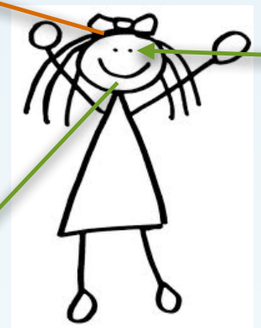
Research Experience as Undergrad ?

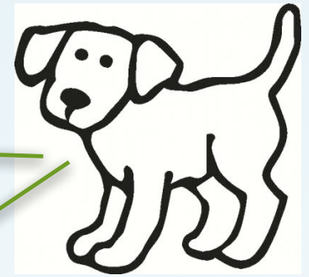
Grad School ?

2 Syntax

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

Sharing Thoughts





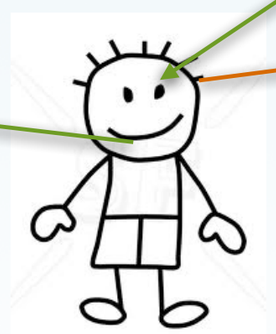
doggy

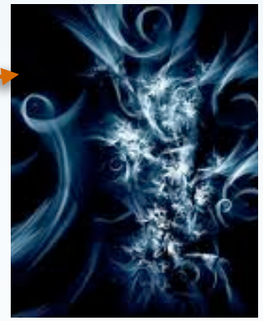
Hund

dog

Wauwau

chien





Syntax: Agreed-upon representation for semantic concepts

2 Syntax

Grammars & Derivation

Syntax Tree

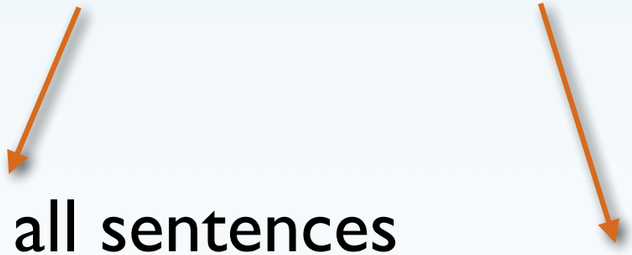
Abstract vs. Concrete Syntax

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?



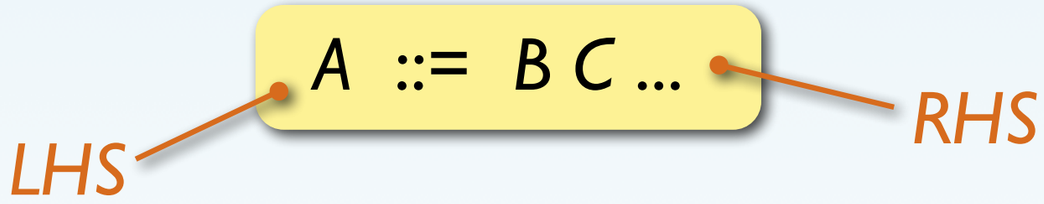
Enumerate all sentences

Define rules to construct valid sentences

Grammar

Grammar

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



A, *B*, *C* ... 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	
digit bin	(R4)
digit digit bin	(R4)
digit digit digit	(R3)
digit digit 1	(R2)
digit 0 1	(R1)
1 0 1	(R2)

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

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

WARNING: The video contains R-rated language!

2 Syntax

Grammars & Derivation

Syntax Tree

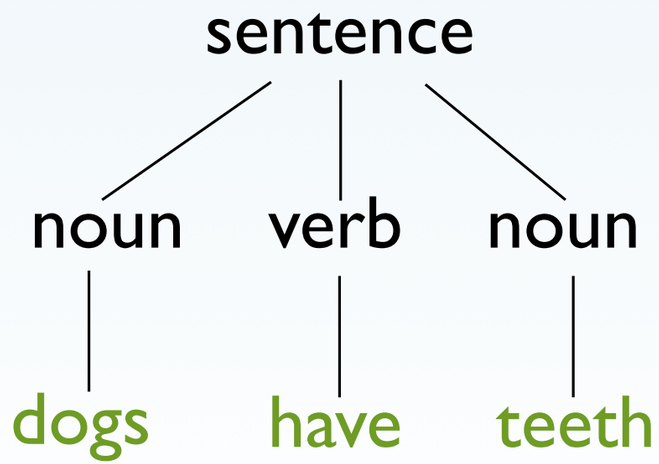
Abstract vs. Concrete Syntax

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

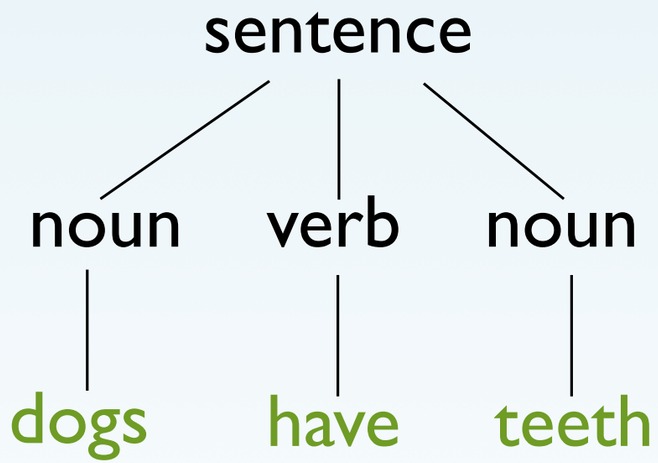
Observations About Syntax Trees

(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

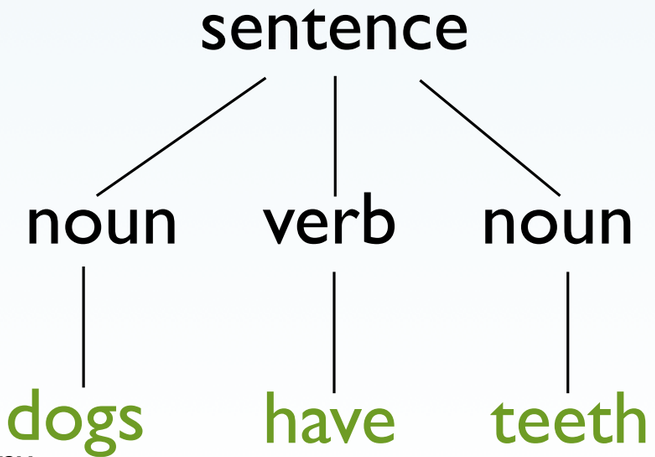


Alternative Representation

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

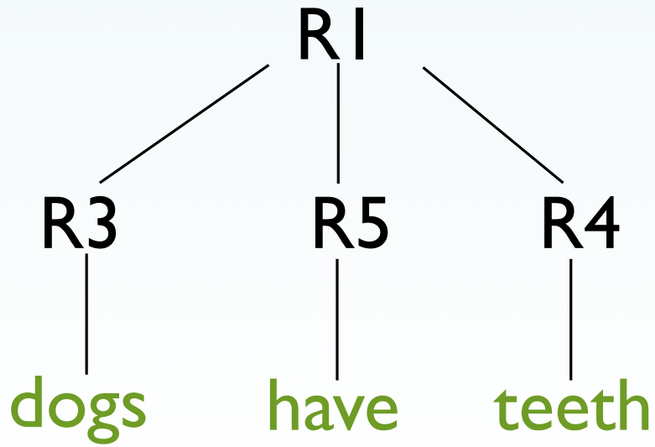
(1) Leaves contain *terminal symbols*

(2) Internal nodes contain *nonterminal symbols*



(1) Leaves contain *terminal symbols*

(2) Internal nodes contain *rule names*



Exercises

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

(1) Draw the syntax tree for the sentence 101

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

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

(3) Draw all syntax trees of type noun

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

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

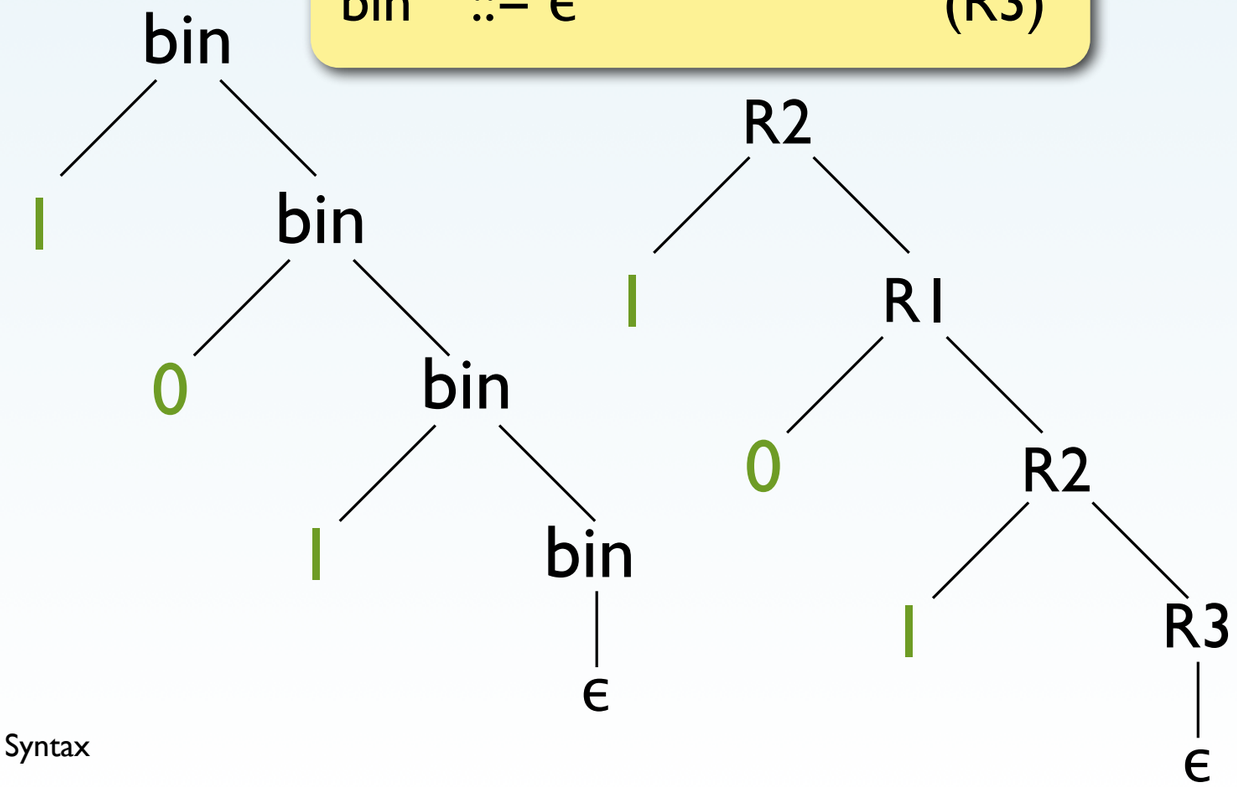
(5) How many with the following grammar?

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

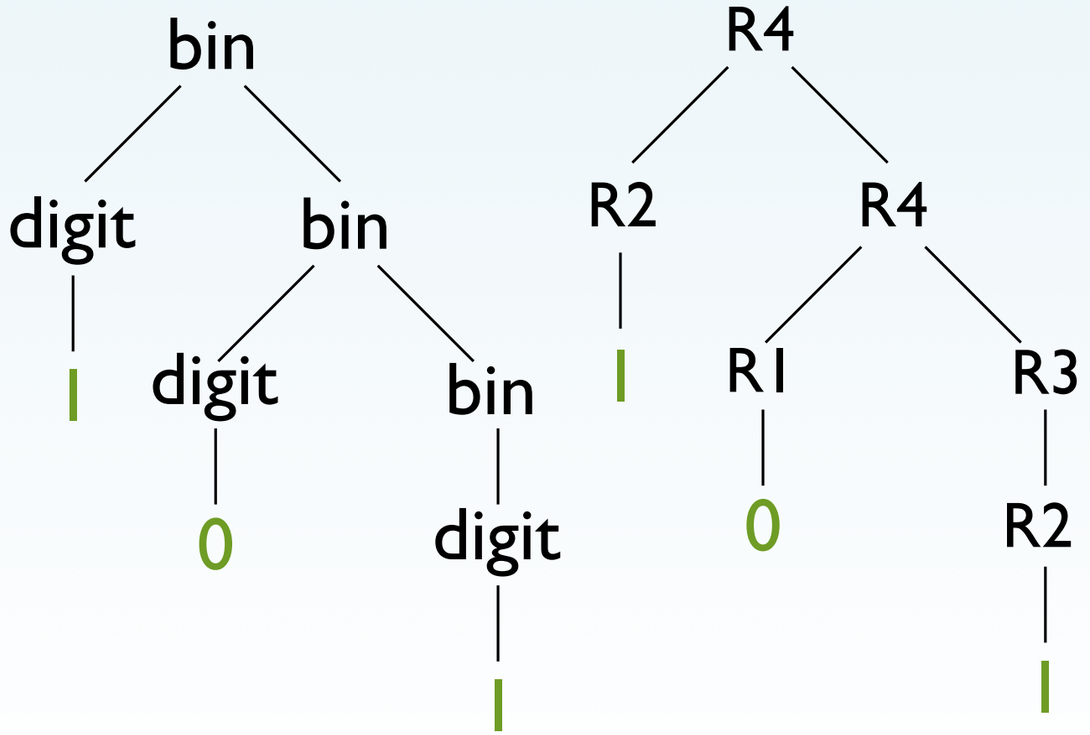
Exercises

(1) Draw the syntax tree for the sentence 101

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

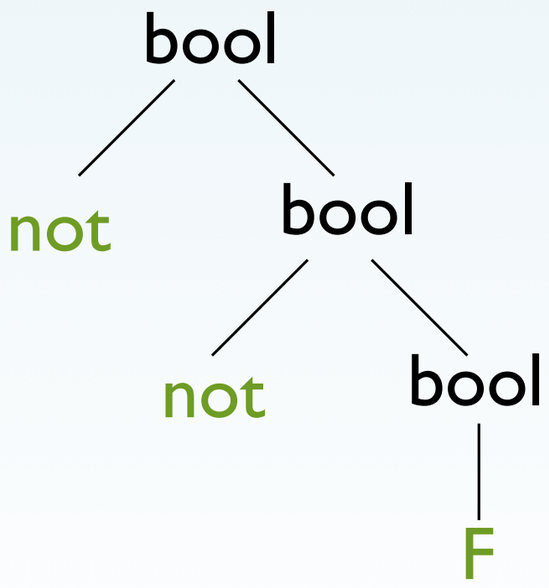


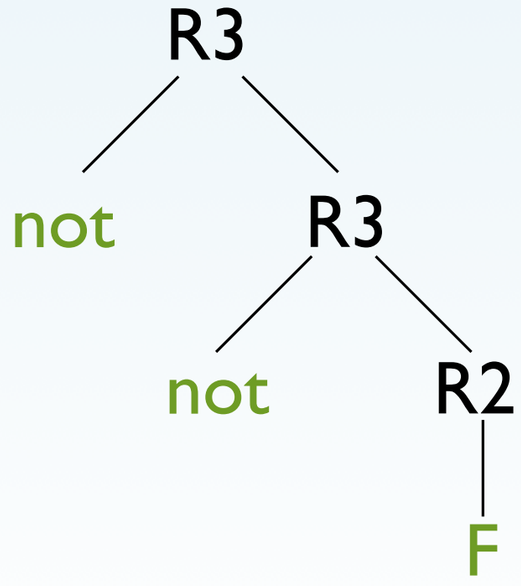
digit	::= 0	(R1)
digit	::= 1	(R2)
bin	::= digit	(R3)
bin	::= digit bin	(R4)



Exercises

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



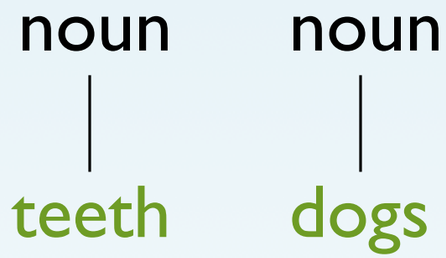


bool	::= T	(R1)
bool	::= F	(R2)
bool	::= not bool	(R3)

Exercises

(3) Draw all syntax trees of type noun

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



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

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

zero
(since stmt has no base case)

(5) How many with the following grammar?

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

infinitely many
(since stmt can be expanded arbitrarily often)

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

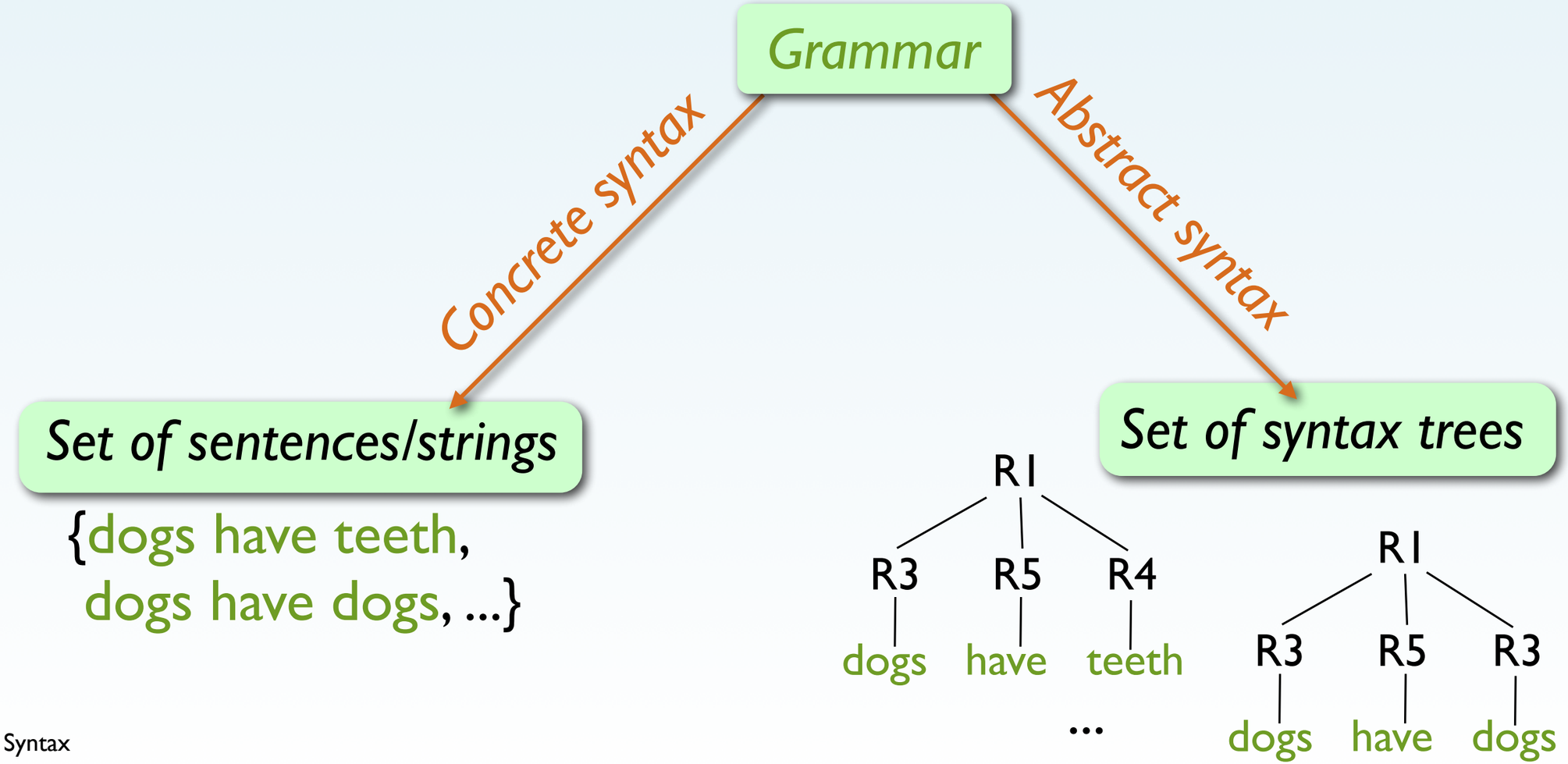
Grammars & Derivation

Syntax Tree

Abstract vs. Concrete Syntax

Representing Grammars by Haskell Data Types

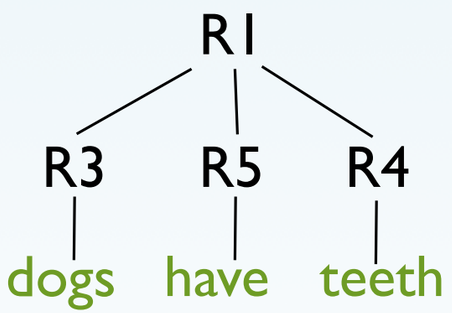
Concrete vs. Abstract Syntax



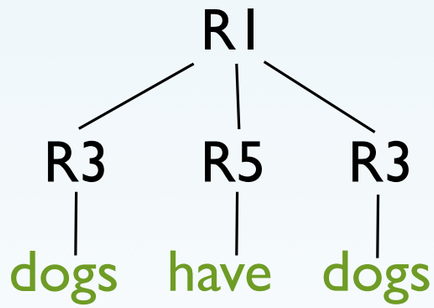
Abstract Syntax

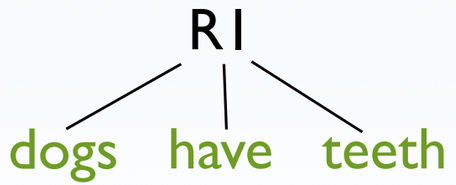
Grammar → *Abstract syntax* → *Set of syntax trees*

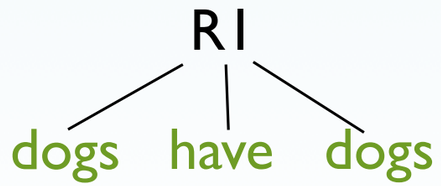
sentence	::=	noun verb noun	(R1)
	|	sentence and sentence	(R2)
noun	::=	dogs | teeth	(R3, R4)
verb	::=	have	(R5)



...







terminal symbols uniquely identify rules (in this grammar)

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

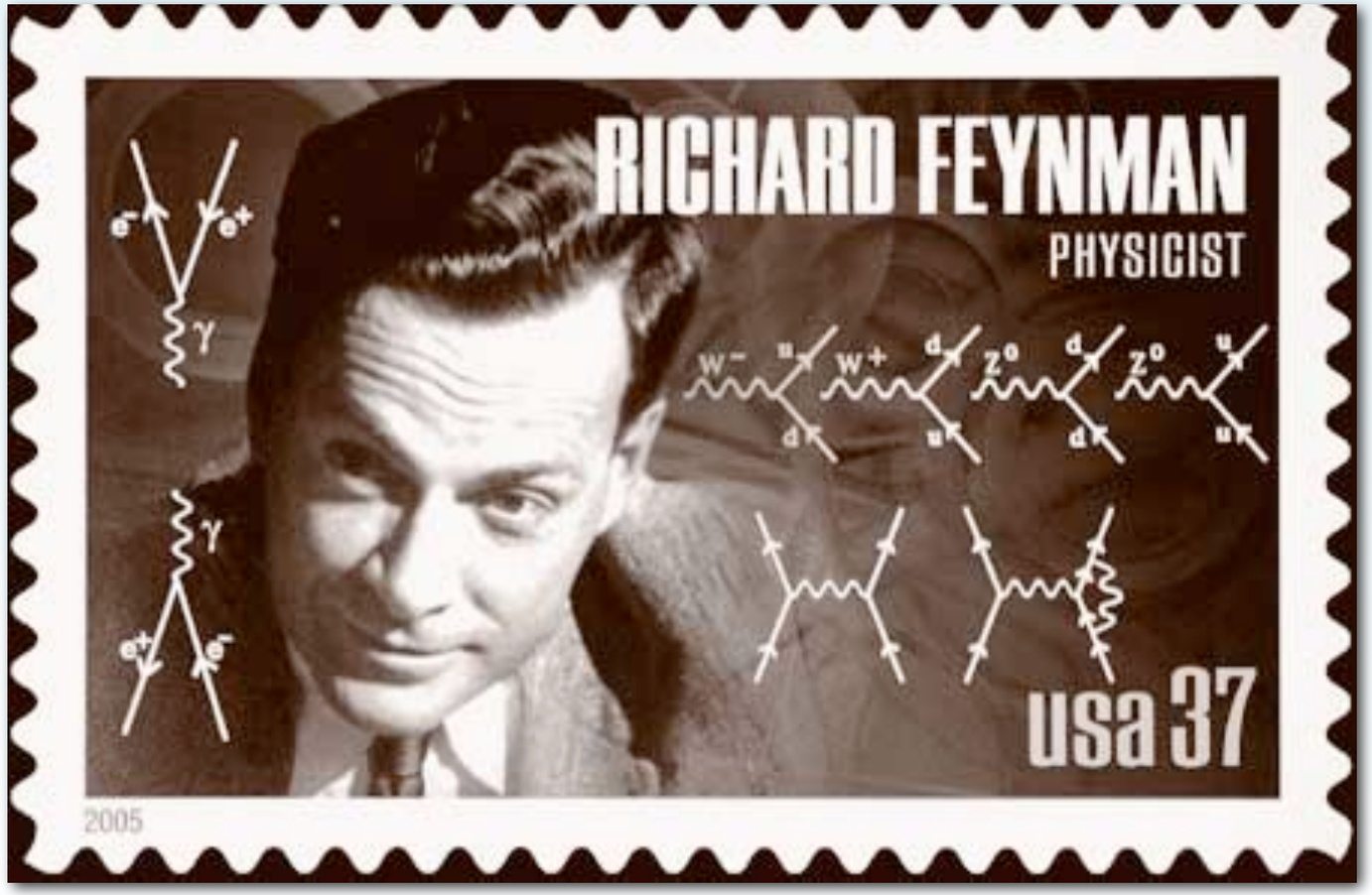
Grammars & Derivation

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
```

```
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)

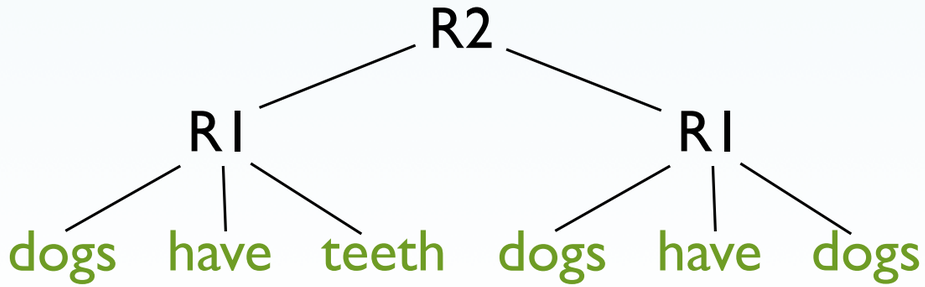
To construct a syntax tree, apply a constructor to subtrees

Haskell Representation of Syntax Trees

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

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


```
R1 Dogs Have Teeth
```



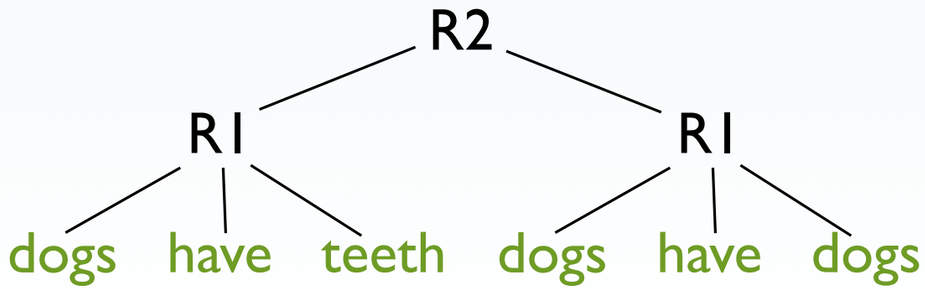

```
R2 (R1 Dogs Have Teeth)
   (R1 Dogs Have Dogs)
```

Haskell Representation of Syntax Trees

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

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


```
Phrase Dogs Have Teeth
```




```
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` ?

More Exercises

(1) Define a Haskell data type for Peano-style natural numbers
(constructed by 0 and successor)
(*Note*: numbers *cannot* be constructors; you must use names such as Zero.)

(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

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

```
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 }
}
```

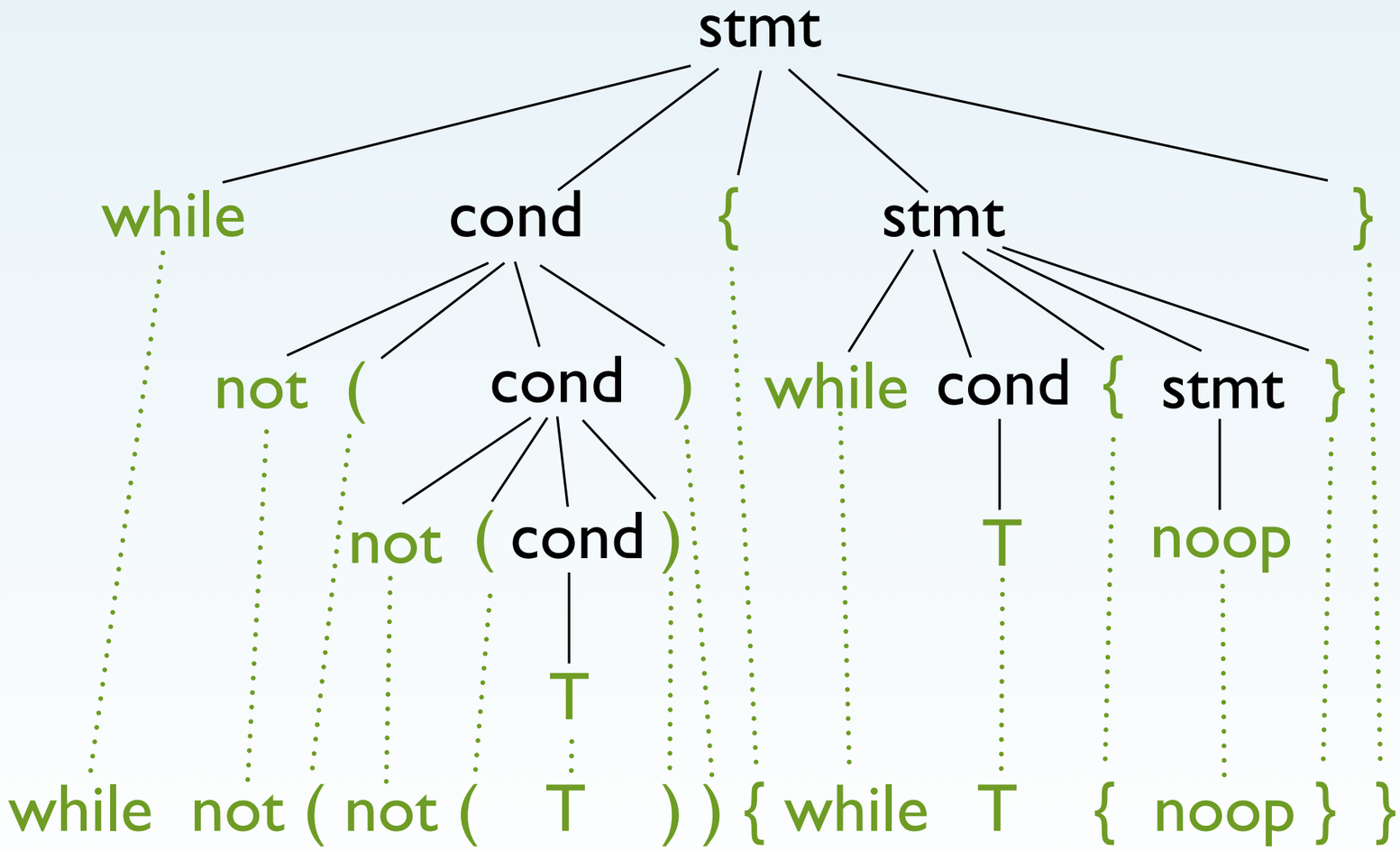
```
cond ::= T | not (cond)
stmt ::= while cond { stmt }
       | noop
```

Exercises

(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

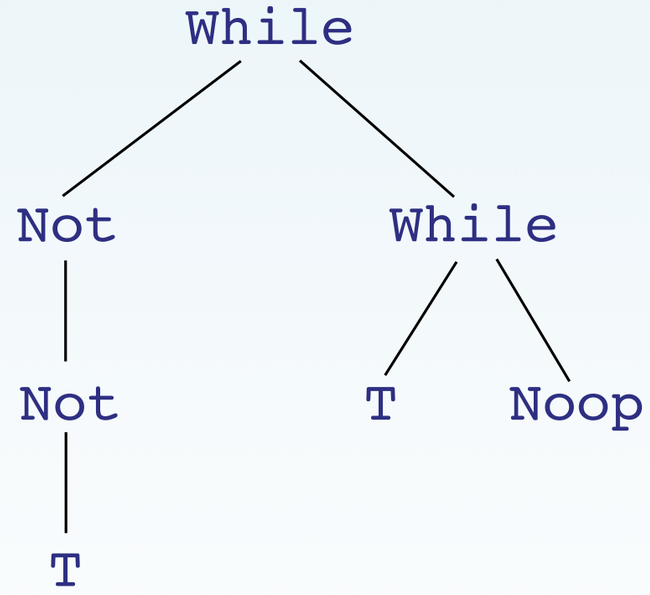
```
While (Not (Not T))
      (While T Noop)
```

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

Exercises

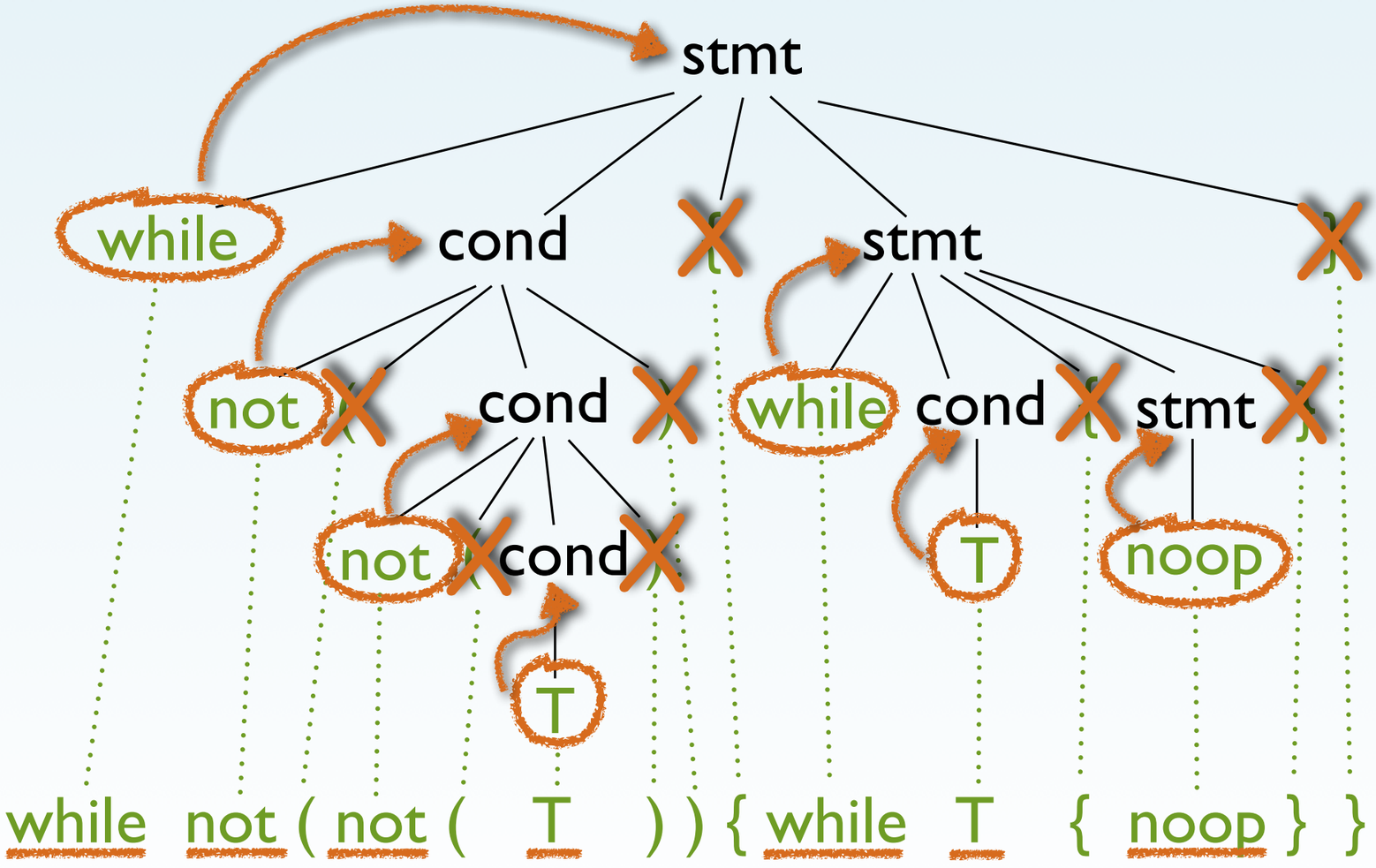
(2) Draw the following *abstract* syntax tree

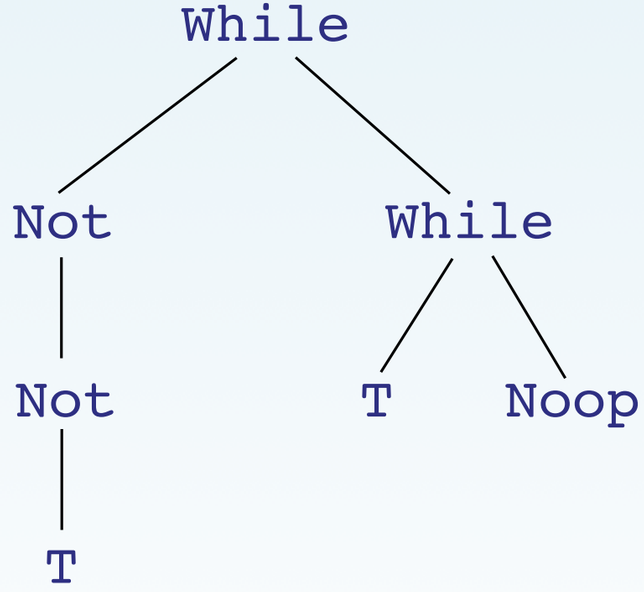
```
While (Not (Not T))
      (While T Noop)
```




```
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
```

Acceptable (Data) Types for Abstract Syntax

Acceptable (data) type:

- *must* represent all sentences of language, (not more), not less

Names and order of arguments of constructors don't matter

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

```
data Cond = Y | Z Cond
data Stmt = Loop Cond Stmt
          | A
```

Acceptable (Data) Types for Abstract Syntax

Acceptable (data) type:

- *may* represent multiple productions with one constructor

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

```
data BExpr = Const Bool | Not BExpr
```

Acceptable (Data) Types for Abstract Syntax

Acceptable (data) type:

- *may* be able to represent one sentence in different ways

```
data Ints = One Int | Add Int Ints
```

```
Add 2 (Add 4 (One 5))                    [2,4,5]
```

```
data Ints = One Int | Join Ints Ints
```

```
Join (One 2) (Join (One 4) (One 5))
Join (Join (One 2) (One 4)) (One 5)
```

```
type Ints = [Int]
```

can represent empty list

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]
```

Strictly speaking, incorrect

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

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 Lists in Abstract Grammars

Zero or more *A*'s

As ::= *A As* | ϵ
B ::= … *As* …

abbreviation →

B ::= … A^* …

One or more *A*'s

As ::= *A As* | *A*
B ::= … *As* …

abbreviation →

B ::= … A^+ …

Abstract syntax

```
data B = Con … [A] …
```

```
data B = Con … (A,[A]) …
```

Zoom Poll

Alternative abstract syntax

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 exp ❹{ stmt }
| noop

```
data ❷Exp = N ❶Int | Plus Exp Exp
data Stmt = ❸While Exp ❹Stmt
          | 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.)*

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

Constructor is indispensable!

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

A perfectly valid alternative!

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

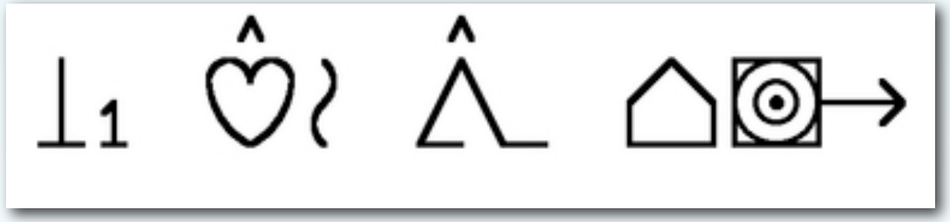
```
type EPair = (Exp,Exp)

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

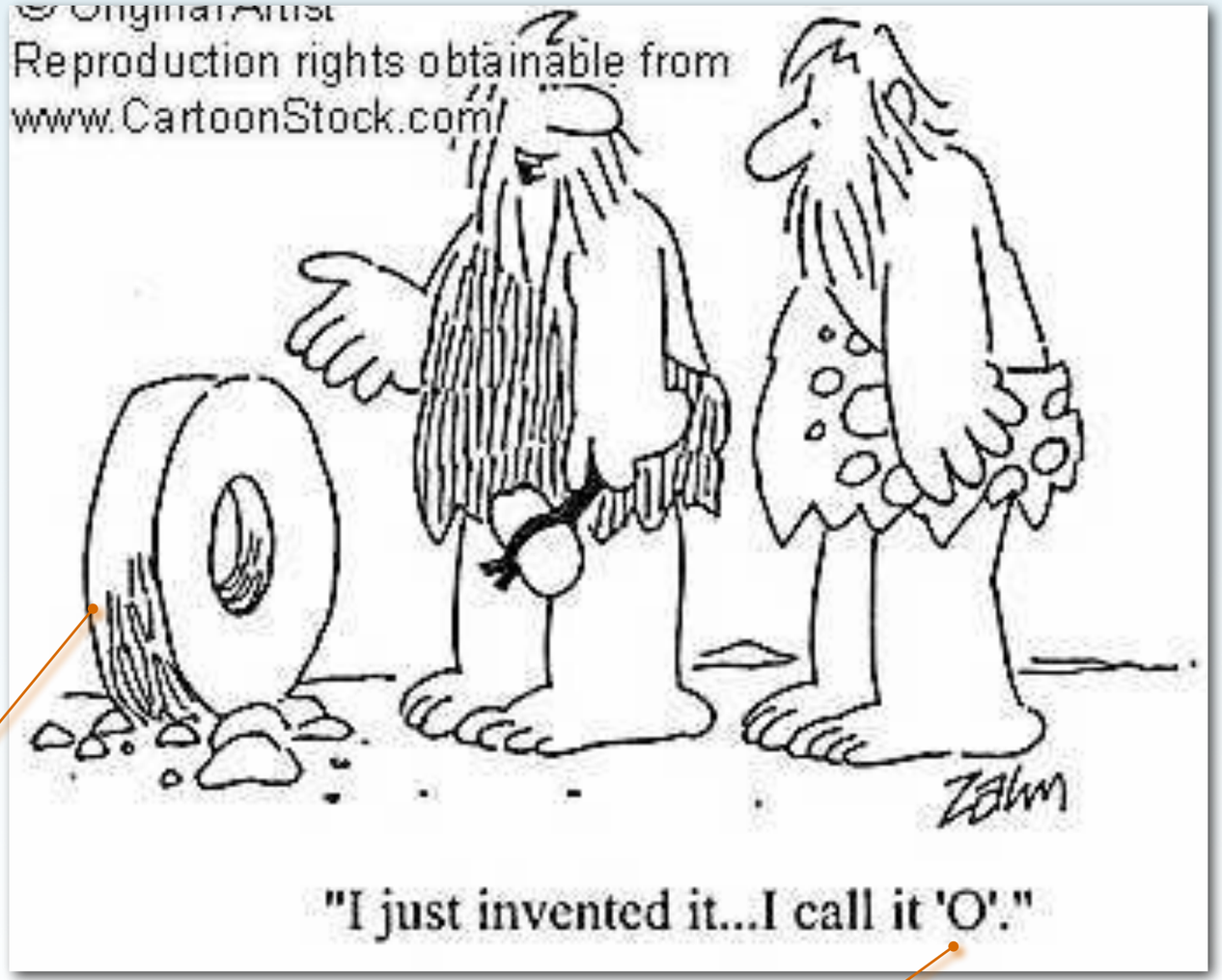
3 Semantics


HOW GOES THE OBEDIENCE TRAINING?
GREAT! WATCH THIS.
HEEL.
NEXT WEEK WE START ON SEMANTICS.
7-14
©2010 Creators.com
www.JohnHartStudios.com

3 Semantics



en.wikipedia.org/wiki/Blissymbols



Semantics

Syntax

3 Semantics

Why semantics?

What is semantics?

Semantics of simple expression languages

Elements of semantic definitions

Examples: Shape & Move languages

Advanced semantic domains

Translating Haskell into denotational semantics

Haskell as a metalanguage

Modus Ponens

$$\frac{S \rightarrow G \quad S}{G}$$

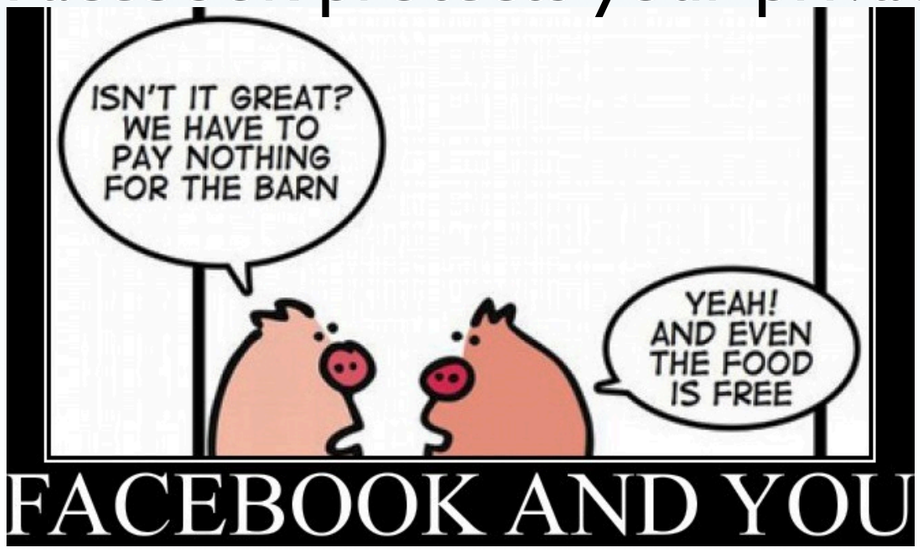
Why Semantics ?

Haskell B. Curry, 1900-1982

If this statement is true, then God exists

If this statement is true, then God does not exist

If this statement is true, then Facebook protects your privacy



Curry's Paradox

Prior, A. N., 1955. "Curry's Paradox and 3-Valued Logic", *Australasian Journal of Philosophy* 33:177-82

See also: John Allen Paulos: *Irreligion,* Hill and Wang 2008

Why Semantics ?

Recursion without a base case

S = If S is true, then God exists

S = If S is true, then God does not exist

S = If S is true, then Facebook protects your privacy

Why Semantics ?

Access to non-local variables

```
{
  int x=2;
  int f(int y) {return y+x;};
  {
    int x=4;
    printf("%d", f(3));
  }
}
```

Output? 5

Why Semantics ?

Swap the values of two variables

```
{
   int x=1;
   int y=8;

   y = x;
   x = y;
}
```

Effect?

```
y: 1
x: 1
```

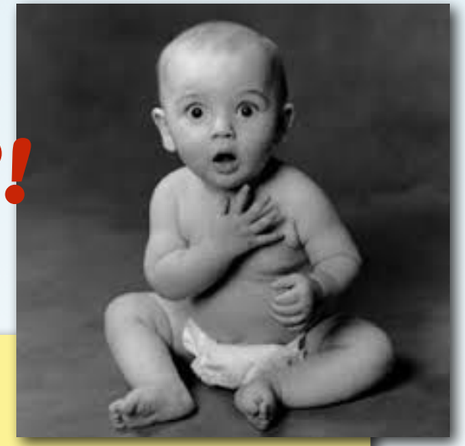
```
{
   int x=1;
   int y=8;
   int z;

   z = y;
   y = x;
   x = z;
}
```

Effect?

```
y: 1
x: 8
```

What?!



```
{
   int x=1;
   int y=8;

   y = x + 0*(x = y);
}
```

Effect?

```
y: 1
x: 8
```

Why Semantics ?

- Understand what *program* constructs do
- Judge the correctness of a *program*
  (compare expected with observed behavior)
- Prove properties about *languages*
- Compare *languages*
- Design *languages*
- Specification for *language* implementations

Syntax: Form of programs
Semantics: Meaning of programs

The Meaning of Programs

What is the meaning of a program?

It depends on the language!

Language	*Meaning*
Boolean expressions	Boolean value
Arithmetic expressions	Integer
Imperative Language	State transformation
Logo	Picture

Denotational Semantics of a language:
Transformation of representation
(abstract syntax → semantic domain)

Simple Examples

BoolSyn.hs
BoolSem.hs

ExprSyn.hs
ExprSem.hs

Exercises

(1) Extend the boolean expression language by an and operation (abstract syntax and semantics)

(2) Extend the arithmetic expressions by multiplication and division (abstract syntax and semantics)

(3) Define a Haskell function to apply DeMorgan's laws to boolean expression, i.e., a function to transform any expression not (x and y) into (not x) or (not y) (and accordingly for not (x or y))

Defining Semantics in 3 Steps

Example Language
"*Arithmetic expressions*"

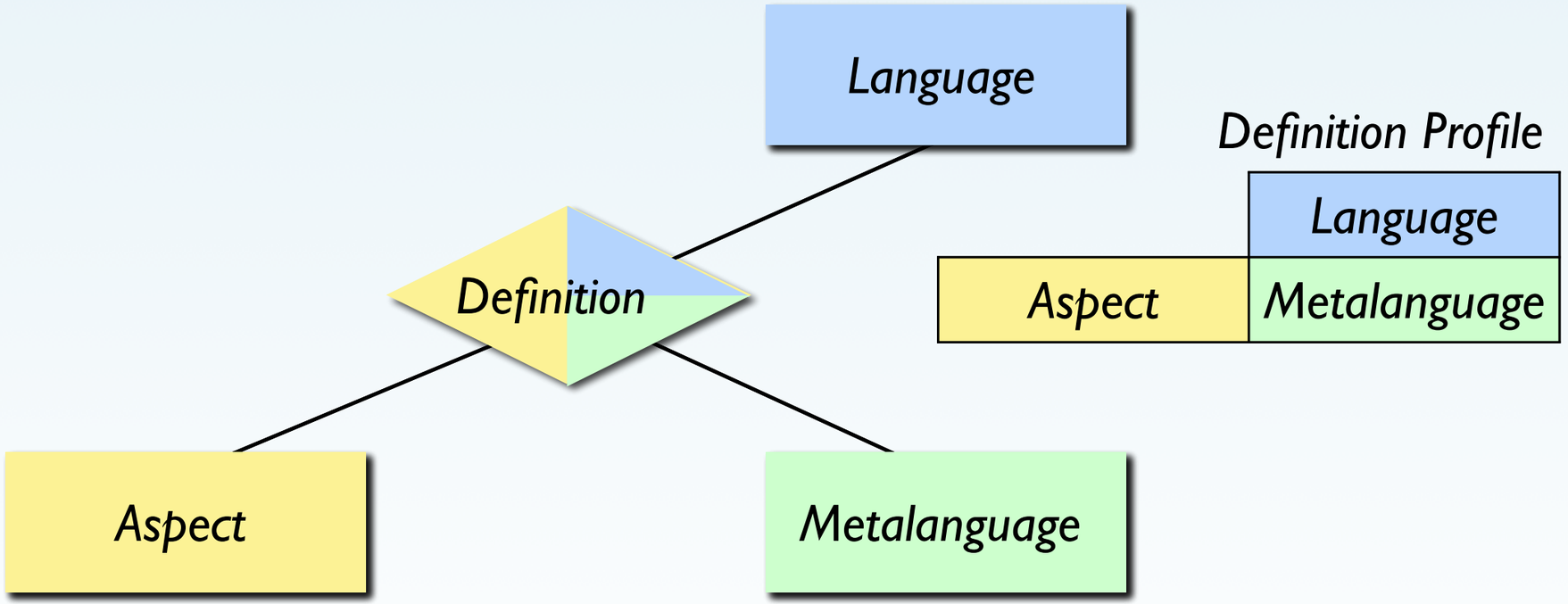
S:	`Expr`
D:	`Int`
$[\![\cdot]\!]$:	`sem :: Expr → Int`

(1) Define the *abstract syntax* *S*,
i.e. set of syntax trees

(2) Define the *semantic domain* *D*,
i.e. the representation of semantic values

(3) Define the *semantic function* / *valuation* $[\![\cdot]\!] : S \to D$
that maps trees to semantic values

Language Definitions



Example Expression Language

Expr

Syntax | Haskell

```
data Expr = N Int
          | Plus Expr Expr
          | Neg Expr
```

Semantic Domain

Expr

Semantics | Haskell

```
sem :: Expr → Int
sem (N i)       = i
sem (Plus e e') = sem e + sem e'
sem (Neg e)     = -(sem e)
```

Semantic Function

Expr

Syntax | Grammar

$$Expr \quad ::= \quad Num \mid Expr{+}Expr \mid {-}Expr$$

Expr

Semantics | Math

$$[\![\cdot]\!] : Expr \to Int$$
$$[\![n]\!] = n$$
$$[\![e{+}e']\!] = [\![e]\!] + [\![e']\!]$$
$$[\![{-}e]\!] = -[\![e]\!]$$

Syntactic Symbol

Semantic Operation

Related: Expressions vs. Values

```
tail [1 .. 4]
```

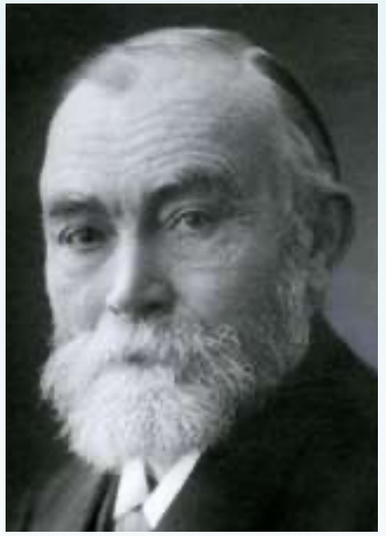
semantics →

```
[2,3,4]
```

or

```
2:3:4:[]
```

~~`[2 .. 4]`~~



Gottlob Frege:
Sense vs. Reference

```
3 + 5
```

semantics →

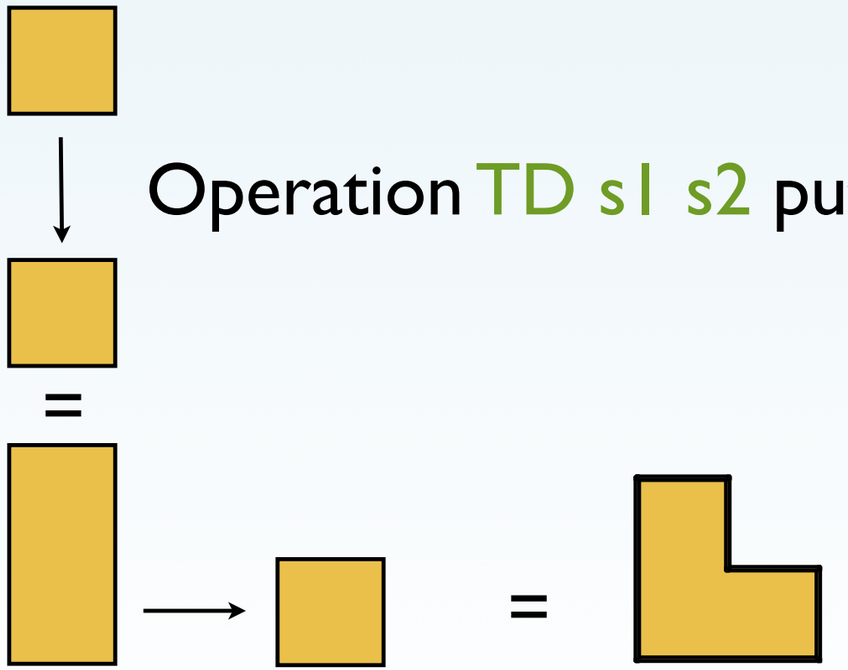
```
8
```

~~`2 + 6`~~

`3*5` *and* `5+5+5` *have different sense (meaning) but the same referent*

Example: Shape Language

A language for constructing bitmap images: an image is either a pixel or a vertical or horizontal composition of images

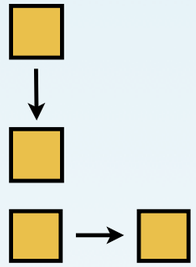


Operation TD s1 s2 puts s1 on top of s2

Operation LR s1 s2 puts s1 left next to s2

Abstract Syntax

```
data Shape = X
           | TD Shape Shape
           | LR Shape Shape
```

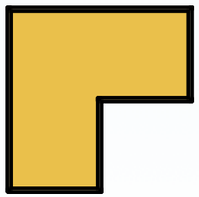


Example:

```
LR (TD X X) X
```

```
TD X (LR X X)
```

```
TD (LR X X) X
```



LR aligns at bottom

TD aligns at left

... part of semantics

Semantic Domain

How to represent a bitmap image?

```
data Shape = X
           | TD Shape Shape
           | LR Shape Shape
```

```
type Image = Array (Int,Int) Bool
```

Drawback: size is fixed, operations require complicated bit shifting

```
type Pixel = (Int,Int)
type Image = [Pixel]
```

```
LR (TD X X) X
```

semantics →

```
[(1,1),(1,2),(2,1)]
```

Semantic Function (1)

Approach: Translate individual shapes separately into *pixel lists* and then compose pixel lists

```
data Shape = X
           | TD Shape Shape
           | LR Shape Shape
```

semantics →

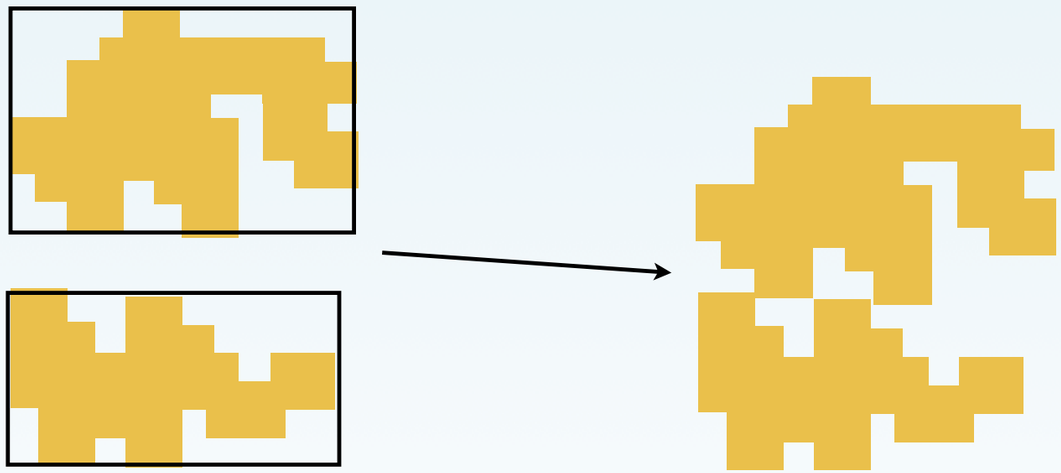
```
type Pixel = (Int,Int)
type Image = [Pixel]
```

Base case: Individual pixel

```
sem :: Shape -> Image
sem X = [(1,1)]
```

Semantic Function (2)

How can we compose (horizontally and vertically) two pixel lists images without overlapping?



Take bounding boxes and adjust y-coordinates of top shape by height of bottom shape

```
sem (TD s1 s2) = adjustY ht p1 ++ p2
    where p1 = sem s1
          p2 = sem s2
          ht = maxY p2
```

Semantic Function (3)

```
sem (TD s1 s2) = adjustY ht p1 ++ p2
    where p1 = sem s1
          p2 = sem s2
          ht = maxY p2
```

```
maxY :: [(Int,Int)] -> Int
maxY p = maximum (map snd p)
```

```
adjustY :: Int -> [(Int,Int)] -> [(Int,Int)]
adjustY ht p = [(x,y+ht) | (x,y) <- p]
```

Exercise

```
sem (TD s1 s2) = adjustY ht p1 ++ p2
    where p1 = sem s1
          p2 = sem s2
          ht = maxY p2

maxY :: [(Int,Int)] -> Int
maxY p = maximum (map snd p)

adjustY :: Int -> [(Int,Int)] -> [(Int,Int)]
adjustY ht p = [(x,y+ht) | (x,y) <- p]
```

Define the functions:

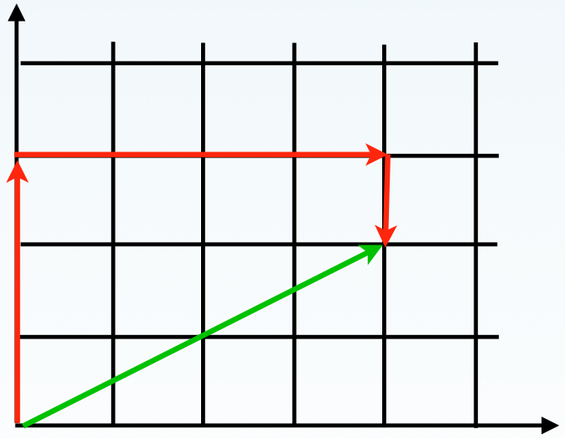
```
sem (LR s1 s2)      maxX      adjustX
```

Example

Shape.hs
ShapePP.hs

Example: Move Language

A language describing vector-based movements in the 2D plane.
A *step* is an *n*-unit horizontal or vertical move,
a *move* is a sequence of steps.



Go Up 3;
Go Right 4;
Go Down 1

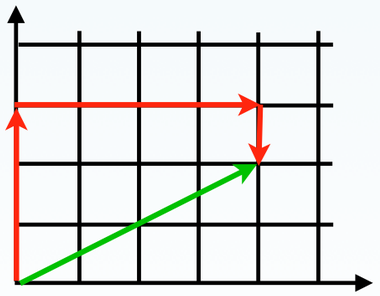
Abstract Syntax

```
data Dir  = Lft | Rgt | Up | Dwn

data Step = Go Dir Int

type Move = [Step]
```

Example:



```
[Go Up 3, Go Rgt 4, Go Dwn 1]
```

Exercises

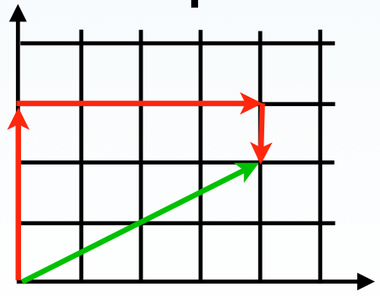
(1) Give a type definition for the data type `Step`

```
data Step = Go Dir Int
```

(2) Define the data type `Move` without using built-in lists

```
type Move = [Step]
```

(3) Write the move `[Go Up 3, Go Rgt 4, Go Dwn 1]` using the representation from (1) and(2)



Semantic Domain

What is the meaning of a move?

```
data Dir  = Lft | Rgt | Up | Dwn

data Step = Go Dir Int

type Move = [Step]
```

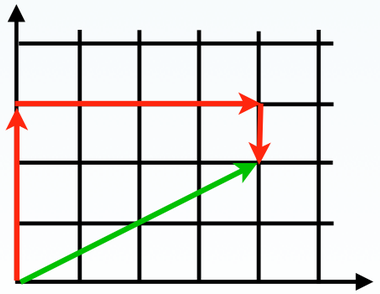
The distance traveled, the final position, or both.

```
type Pos = (Int,Int)
```

```
[Go Up 3, Go Rgt 4, Go Dwn 1]
```

semantics

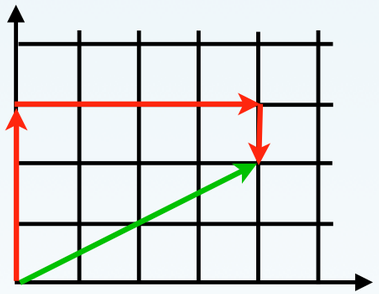
```
(4,2)
```



Semantic Function

```
sem :: Move -> Pos
sem []           = (0,0)
sem (Go d i:ss) = (dx*i+x,dy*i+y)
                  where (dx,dy) = vector d
                        (x,y)   = sem ss
```

pattern matching in definitions



```
vector :: Dir -> (Int,Int)
vector Lft = (-1,0)
vector Rgt = (1,0)
vector Up  = (0,1)
vector Dwn = (0,-1)
```

Example

Move.hs

Exercises

```
sem :: Move -> Pos
sem []           = (0,0)
sem (Go d i:ss) = (dx*i+x,dy*i+y)
                  where (dx,dy) = vector d
                        (x,y)   = sem ss
```

Define the semantic function for the move language for the semantic domain

```
type Dist = Int
```

Define the semantic function for the move language for the semantic domain

```
type Trip = (Dist,Pos)
```

Advanced Semantic Domains

The story so far: Semantic domains were mostly simple types (such as `Int` or `[(Int,Int)]`)

How can we deal with language features, such as *errors*, *union types*, or *state*?

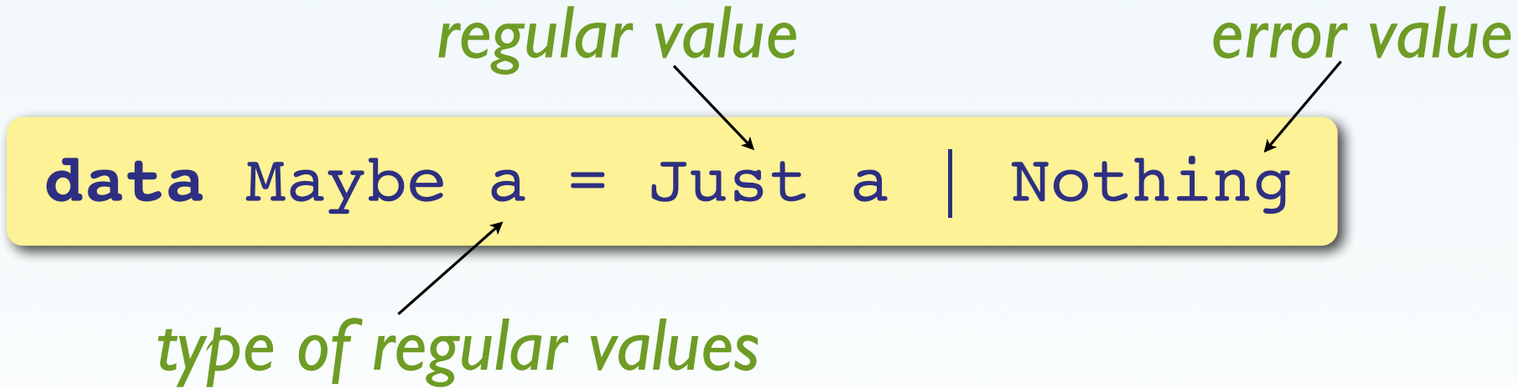
(1) *Errors*: Use the `Maybe` *data type*

(2) *Union types*: Use corresponding *data types*

(3) *State*: Use *function types*

Error Domains

If `T` *is the type representing "regular" values, define the semantic domain as* `Maybe T`

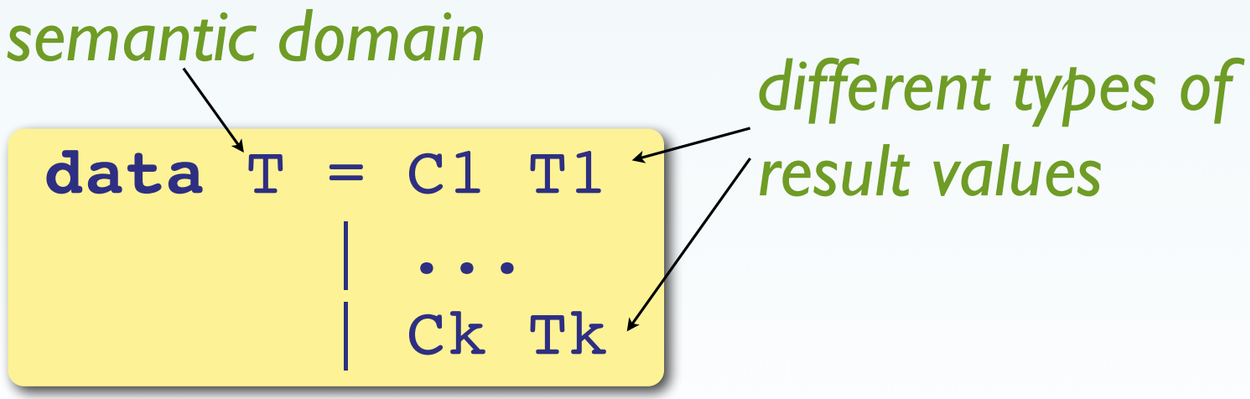


Example

ExprErr.hs

Union Domains

If `T1` *...* `Tk` *are types representing different semantic values for different nonterminals, define the semantic domain as a data type with k constructors.*



Special Case: Binary Union Domains

If `T1` and `T2` *are types representing different semantic values for different nonterminals, define the semantic domain as a data type with 2 constructors.*

Or: Use the `Either` *data type.*

```
data Either a b = Left a | Right b
```

```
data T = C1 T1
       | C2 T2
```

```
type T = Either T1 T2
```

```
data Val = I Int
         | B Bool
```

```
type Val = Either Int Bool
```

Example

Expr2.hs

Exercises

(1) Extend the semantic domain for the two-type expression language to include errors

```
data Val = I Int
         | B Bool
```

(2) Extend the semantic function for the two-type expression language to handle errors

Function Domains

If a language operates on a state that can be represented by a type `T`*, define the semantic domain as a function type* `T -> T`

```
type D = T -> T
```

```
sem :: S -> D
```

=

```
sem :: S -> (T -> T)
```

=

```
sem :: S -> T -> T
```

Semantic function takes state as an additional argument

Example

RegMachine.hs

Exercises

(1) Extend the machine language to work on two registers A and B

```
data Op = LD Int
        | INC
        | DUP
```

(2) Define a new semantic domain for the extended language

```
type RegCont = Int

type D = RegCont -> RegCont
```

(3) Define the semantics functions for the extended language

RegMachine2.hs

Zoom Poll

Semantic Domain

Translating Haskell into Mathematical Denotational Semantics

(1) Replace *type definitions* by *sets (should actually be CPOs)*

(2) Replace *patterns* by *grammar productions (and replace nonterminals by variables)*

(3) Replace *function names* by *semantic brackets* that enclose only syntactic objects

Semantics	Expr
Haskell	

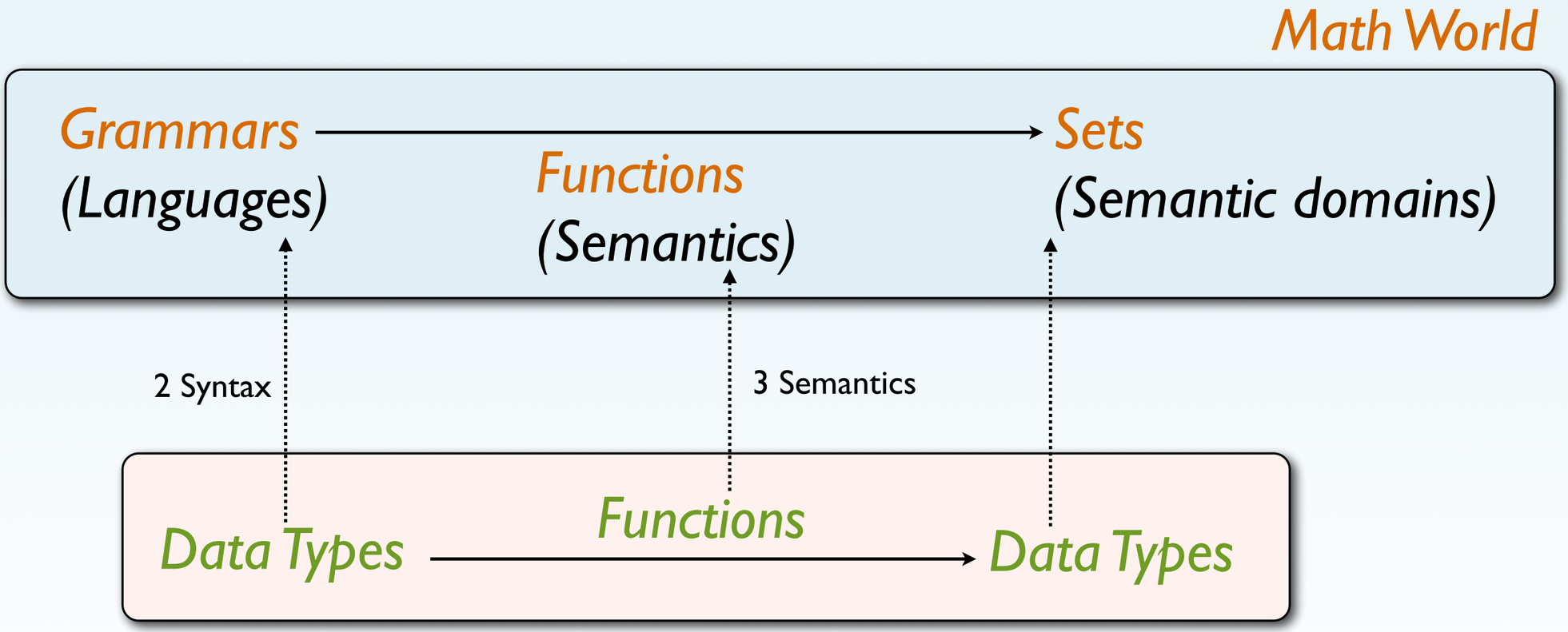
```
sem :: Expr → Int      ❶
sem (N i)        = i
sem (Plus e e')  = sem e + sem e'   ❷ ❸
sem (Neg e)      = -(sem e)
```

Semantics	Expr
Math	

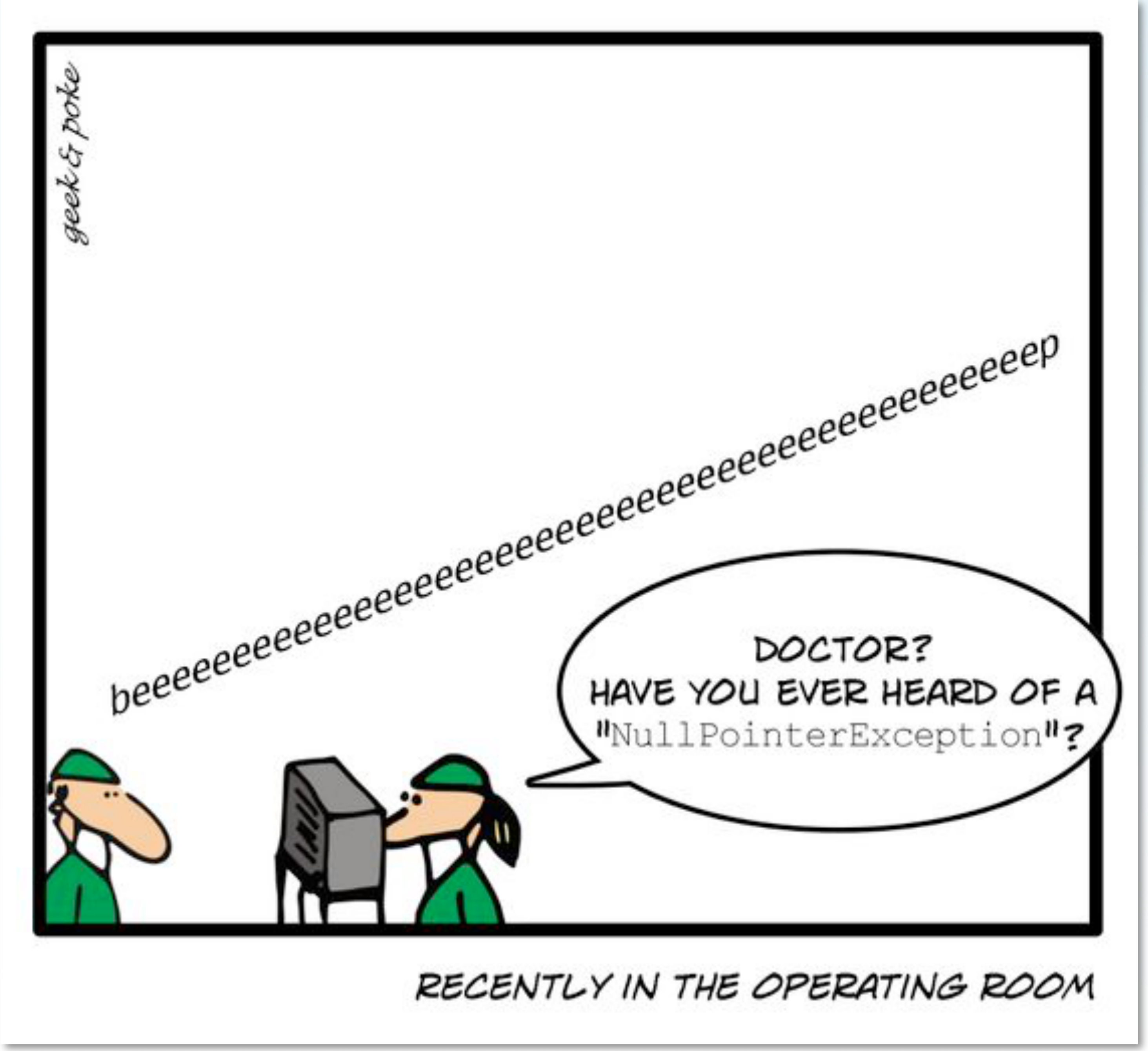
$$[\![\cdot]\!] : Expr \to Int \quad ❶$$
$$[\![n]\!] = n$$
$$[\![e + e']\!] = [\![e]\!] + [\![e']\!] \quad ❷$$
$$[\![-e]\!] = -[\![e]\!] \quad ❸$$

$$Expr ::= Num \mid Expr + Expr \mid -Expr$$

Haskell as a Mathematical Metalanguage



4 Types



geek & poke
beeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeep
DOCTOR?
HAVE YOU EVER HEARD OF A
"NullPointerException"?
RECENTLY IN THE OPERATING ROOM

4 Types

Types, Type Errors, Type Systems

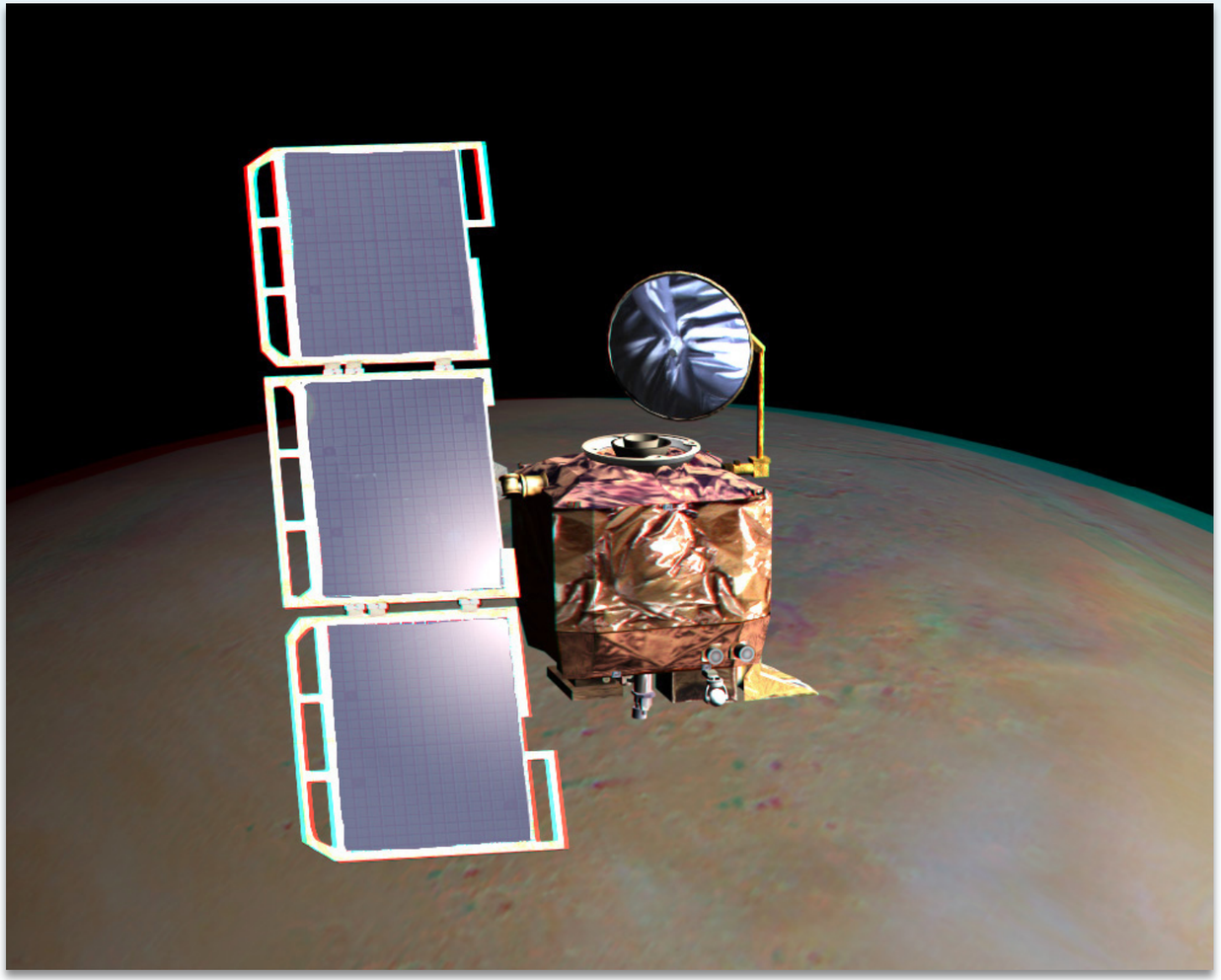
Why Type Checking?

Static vs. Dynamic Typing

Polymorphism

Parametric Polymorphism

Lost ...



mars.jpl.nasa.gov/msp98/orbiter/

More Motivation ...

Video clip

First-Cause Arguments

Plato, Aristotle, Aquinas, ...

Kalam Cosmological Argument
(Al-Ghazali, William Lane Craig, ...)

(1) Whatever begins to exist has a cause.
(2) The Universe began to exist.
(3) Therefore, the Universe had a cause.

Fallacy of Composition

Composition Fallacy

Incorrect Reasoning

Individual things a, b, c, ... of type A have property P.
Therefore, A has property P.

Atoms are invisible. Cats are made of atoms.
Therefore, cats are invisible.

Every natural number has a successor.
Therefore, the type Nat has a successor.

Everything in the universe has a cause.
Therefore, the universe has a cause.

Confusing types & values

JavaScript Madness

```
> "5" - 3
2

> "5" + 3
53

> 5 + "3"
53

> 5 + +"3"
8
```

```
> "5" + +"3"
53

> "5" + +"3" - 2
51

> "5" + +"3" + - 2
53-2
```

```
> "foo" + +"foo"
fooNaN

> +"foo" + +"foo"
NaN

> +"foo" + "foo"
NaNfoo
```

What is a Type (System)?

- *Type*
  Collection of PL elements that share the same behavior
- *Type System*
  Formal system to characterize the types of PL elements and their interactions
  Can be used to prove the *absence of type errors*
- *Purpose of Type Systems*
  Ensure meaningful interaction of different program parts, i.e., ensure *absence of type errors*

Type Errors

- *Type Error*
  Illegal combination of PL elements
  (typically: applying an operation to a value of the wrong type)
- *Why are type errors bad?*
  Lead to program crashes
  Cause incorrect computations

Why Types are a Good Thing

(1) Types provide *precise documentation* of programs

(2) Types *summarize* a program on an abstract level

(3) Type correctness means *partial correctness* of programs;
a type checker delivers partial *correctness proofs*

(4) Type systems can *prevent* runtime *errors*
(and can save a lot of debugging)

(5) Type information can be exploited for *optimization*

Things to Know About Type Systems

(1) Notion of Type Safety

(2) Strong vs. Weak Typing

(3) Static vs. "Dynamic Typing"

(4) Approximation & Undecidability of Static Typing

(5) Type Checking vs. Type Inference

(6) Polymorphism (*parametric*, subtype, ad hoc)

Example: Expression Language with 2 Types

Expr2.hs

Type Safety

Type Safety

A programming language is called *type safe* if all type errors are detected

Type Safe Languages

Lisp (*ridiculous type system*)
Java
Haskell
Expr + eval
Expr + evalDynTC

Unsafe Languages (type casts, pointers)

C
C++
JavaScript (“foo” * 4 = NaN)

Exercises

(1) Implement an unsafe `eval` function for the language `Expr`

```
data Expr = N Int
          | Plus Expr Expr
          | Equal Expr Expr
          | Not Expr
```

(a) Use `Int` as the semantic domain

(b) Map boolean values to `0` and `1`

(2) Evaluate unsafe expressions

Expr2Unsafe.hs

Strong vs. Weak Typing

Strong Typing

Each value has one precisely determined type

Weak Typing

Values can be interpreted in different types (e.g. "17" can be used as a string or number, or 0 can be used as a number or boolean)

In practice: Only strongly typed languages are safe (although strong typing does not guarantee safety)

A Type Checker for the Expression Language

Expr2.hs

TypeCheck.hs

Dynamic vs. Static Typing

Dynamic Typing

Types are checked during runtime

Static Typing

Types and type errors are found during compile time

Statically Typed

Haskell

(Java)

Expr + evalStatTC

Dynamically Typed

Lisp

Python

Expr + evalDynTC

Static Typing is Conservative

What is the type of the following expression?

```
if 3>4 then "hello" else 17
```

Under dynamic typing: Int
Under static typing: type error

How about:

```
f x = if test x then x+1 else False
```

Under dynamic typing: ?
Under static typing: type error

Exercises

(1) What is the type of the following function under static and dynamic typing?

```
f x = if not x then x+1 else x
```

(2) What is the type of the following function under static and dynamic typing?

```
f x = f (x+1) * 2
```

Zoom Poll

Type Checking

Undecidability of Static Typing

```
test :: Int -> Bool
f x = if test x then x+1 else not x
```

`f` is *type correct* if `test x` yields `True`
`f` contains a *type error* if `test x` yields `False`

Since `test x` might not terminate, we cannot determinate the value statically, because of the undecidability of the halting problem.

Static typing *approximates* by assuming a type error when type correctness cannot be shown

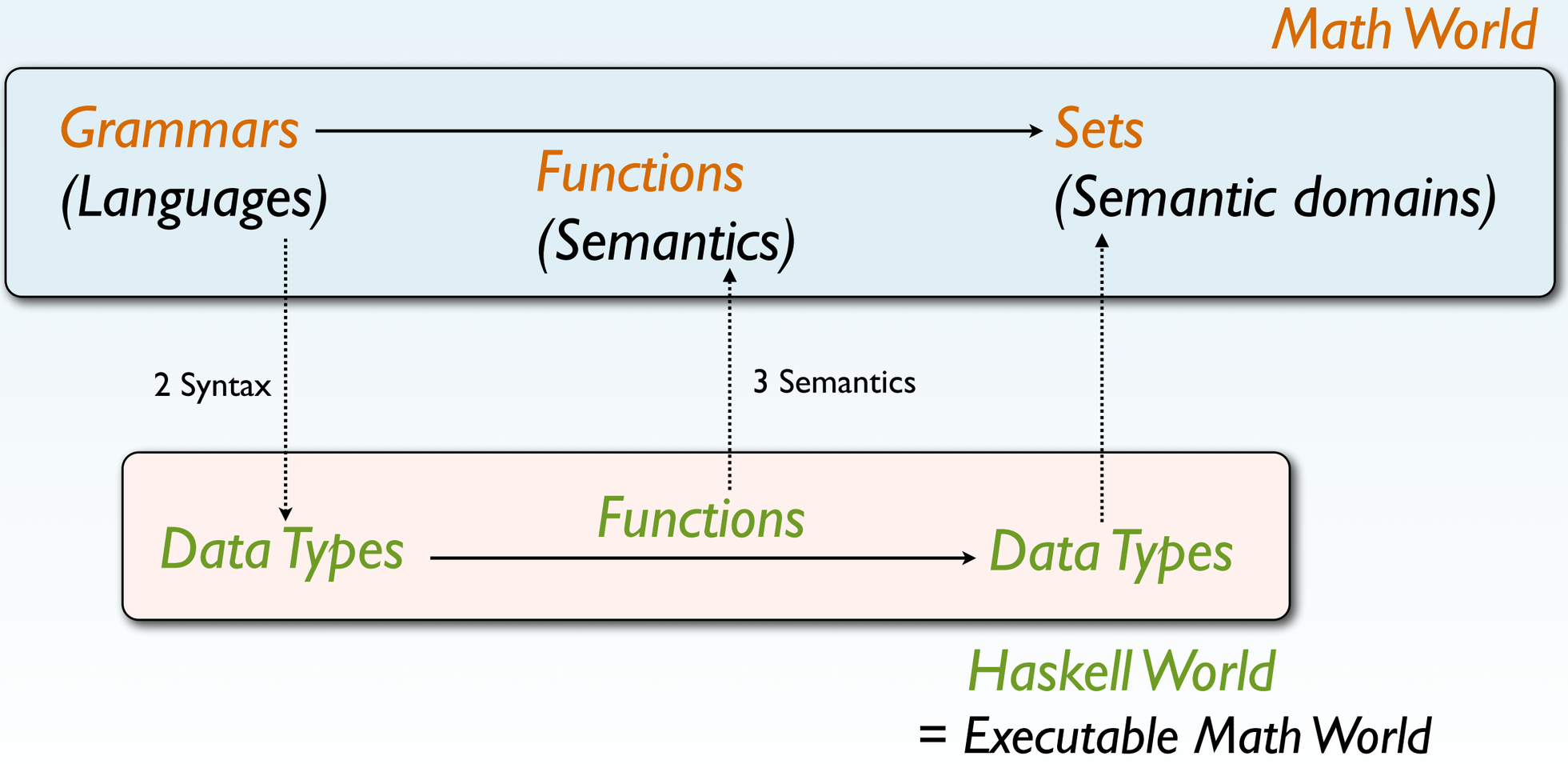
Advantages & Disadvantages of Static/Dynamic Typing

	Advantage	*Disadvantage*
Static Typing	prevents type errors smaller & faster code early error detection (saves debugging)	rejects some o.k. programs
Dynamic Typing	detects type errors fewer programming restrictions faster compilation (& development?)	no guarantees slower execution released programs may stop unexpectedly with type errors

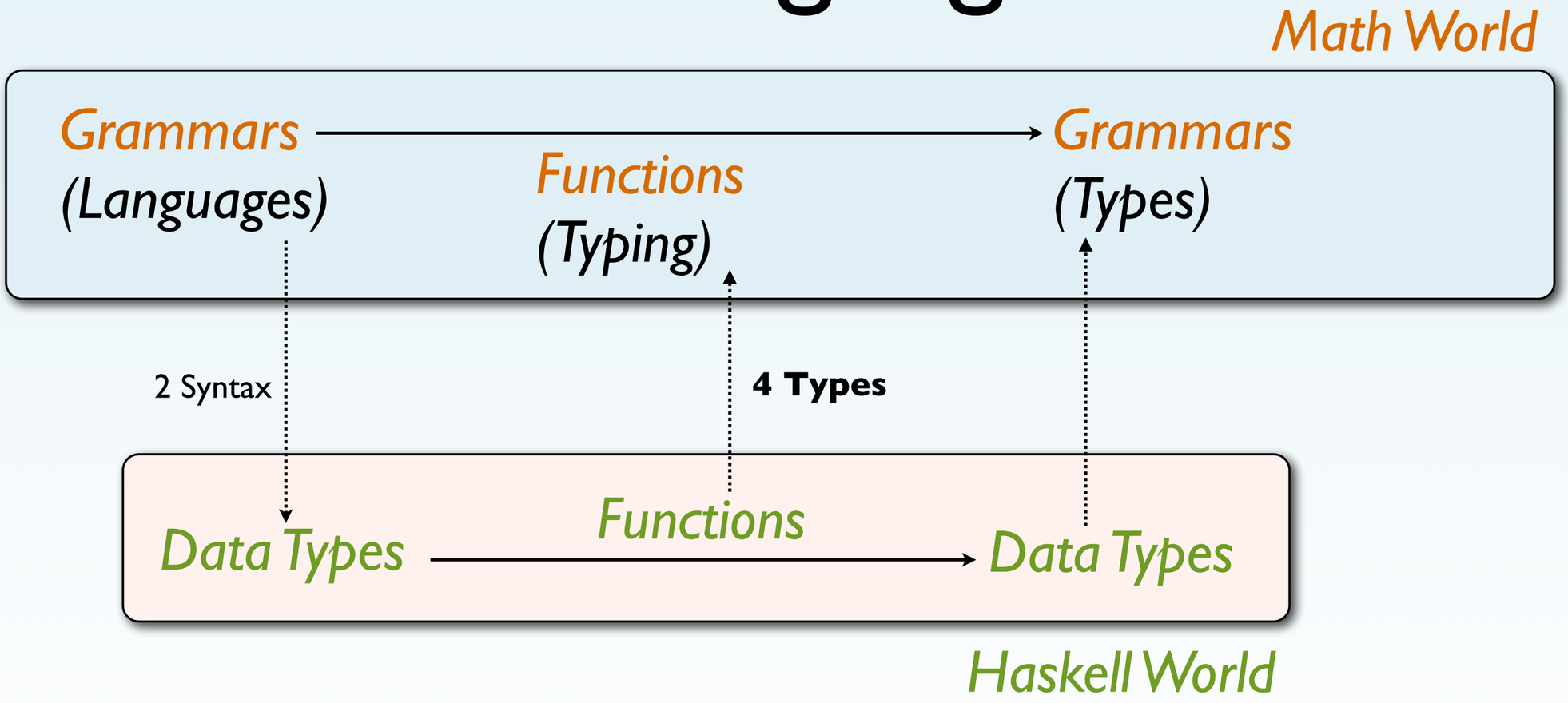
A Type Checker for Arithmetic Language with Pairs

ExprPair.hs
ExprNPair.hs

Haskell as a Mathematical Metalanguage



Haskell as a Mathematical Metalanguage



Typing = ***Static Semantics***

Semantics = *Dynamic Semantics*

Polymorphism

A value (function, method, ...) is *polymorphic* if it has more than one type

Different forms of polymorphism can be distinguished based on:

(a) relationship between types

(b) implementation of functions

Forms of Polymorphism

Parametric Polymorphism

(a) All types match a common "type pattern"

(b) One implementation, i.e., there is only one function

Ad Hoc Polymorphism (aka *Overloading*)

(a) Types are unrelated

(b) Implementation differs for each type, i.e., different functions are referred to by the same name

Subtype Polymorphism

(a) Types are related by a subtype relation

(b) One implementation (methods can be applied to objects of any subtype)

Parametric Polymorphism

Haskell demo

5 Names & Scope

Why Names?

Video clip

5 Names & Scope

Scope & Blocks

Activation Records & Runtime Stack

Scope of Functions and Parameters

Static vs. Dynamic Scoping

Implementation of Static Scoping

Implementation of Recursion

Meaning of Names

Oxford English Dictionary | The definitive record of the English language

trondhjemite, *n*.

Pronunciation: /ˈtrɒndheɪmaɪt/

Etymology: < German *trondhjemit* (V. M. Goldschmidt 1916, in ...

Geol.

Any leucocratic tonalite, esp. one in which the plagioclase is oligoclase.

Jill likes oranges. Jane likes apples.
She enjoys eating them.

Scope of Symbols

Scope of a symbol:
All *locations* in a program where the symbol is visible

that particular definition

Things to know about scope

Blocks (limited scope)
Nested blocks (shadowing)
Runtime stack & activation records
Non-local variables
Static vs. dynamic scoping

Blocks

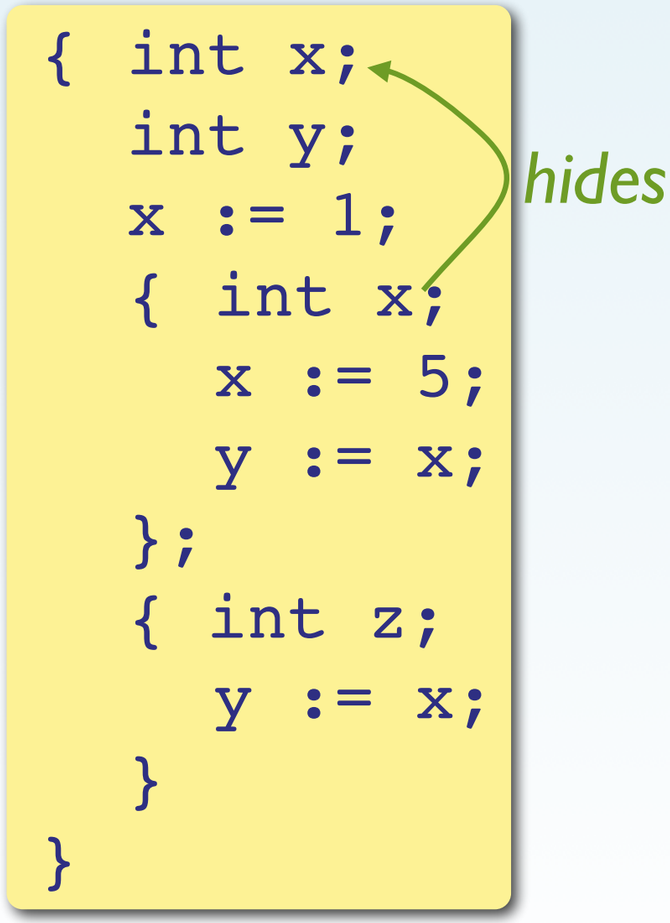
```
{  int x;❶
   int y;❷
  ❶x := 1;
   {  int x;❸
     ❸x := 5;
     ❷y := ❸x;
   };
   {  int z;❹
     ❷y := ❶x;
   }
}
```

A *block* consists of a group of declarations and
(a) a sequence of statements (in imperative languages)
(b) an expression (in functional languages)

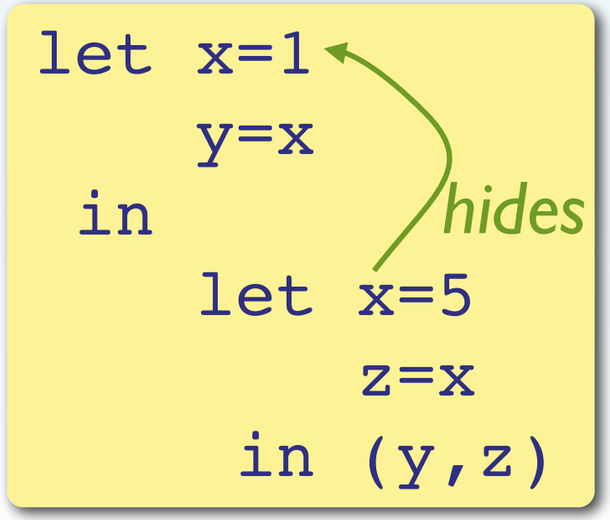
```
let ❶x=1
    ❷y=❶x
 in
    let ❸x=5
        ❹z=❸x
     in (❷y,❹z)
```

Observe references to
local and *non-local* variables

Nested Blocks: Shadowing



Declarations in inner blocks can temporarily hide declarations in enclosing blocks



Homonyms & Synonyms

A name is a homonym if it has more than one meaning.

$$C \neq C' \Rightarrow \texttt{sem}\ C\ x \neq \texttt{sem}\ C'\ x$$

context is needed for disambiguation

Two names x and y are synonyms if they have the same meaning.

$$\texttt{sem}\ x = \texttt{sem}\ y$$

Activation Records

Local variables are kept in memory blocks, called *activation records*, on the *runtime stack*

Enter/leave block: push/pop activation record on/off the runtime stack

```
{  int x;
   int y;
   x := 1;
   {  int x;
      x := 5;
      y := x;
   };
   {  int z;
      y := x;
   }
}
```

[]	
[⟨x:?, y:?⟩]	push
[⟨x:1, y:?⟩]	
[⟨x:?⟩, ⟨x:1, y:?⟩]	push
[⟨x:5⟩, ⟨x:1, y:?⟩]	
[⟨x:5⟩, ⟨x:1, y:5⟩]	
[⟨x:1, y:5⟩]	pop
[⟨z:?⟩, ⟨x:1, y:5⟩]	push
[⟨z:?⟩, ⟨x:1, y:1⟩]	
[⟨x:1, y:1⟩]	pop
[]	pop

A Simplified Model

A declaration of a group of variables is equivalent to a corresponding group of nested blocks for each variable

```
{  int x;
   int y;
   int z;
   x := 1;
   y := x;
}
```

≡

```
{  int x;
   {  int y;
      {  int z;
         x := 1;
         y := x;
      }
   }
}
```

```
let x=1
    y=2
 in x+y
```

≡

```
let x=1
 in  let y=2
      in x+y
```

... we can use activation records of single variables

Simplified Activation Records & Stacks

Enter/leave block:
push/pop activation record
on/off the runtime stack

```
let x=1
 in let y=2
     in x+y
```

[]	
[x:1]	push
[y:2, x:1]	push
[x:1]	pop
[]	pop

Exercise

What is the value of the following expression?

```
let x=1 in (let x=2 in x,x)
```

Example ...

Var.hs
(Variables and Definitions)

Scope of Functions and Parameters

```
{int x;
 {int f(int y){return y+1};
  x := f(1);

 }
}
```

[]	
[x:?]	push
[f:{}, x:?]	push
[y:1, f:{}, x:?]	push
[f:{}, x:2]	pop
[x:2]	pop
[]	pop

Dynamic Scoping

non-local variable

```
{int x;
 x := 1;
 {int f(int y){return y+x};
  {int x;
   x := 2;
   x := f(3);
  }
 }
}
```

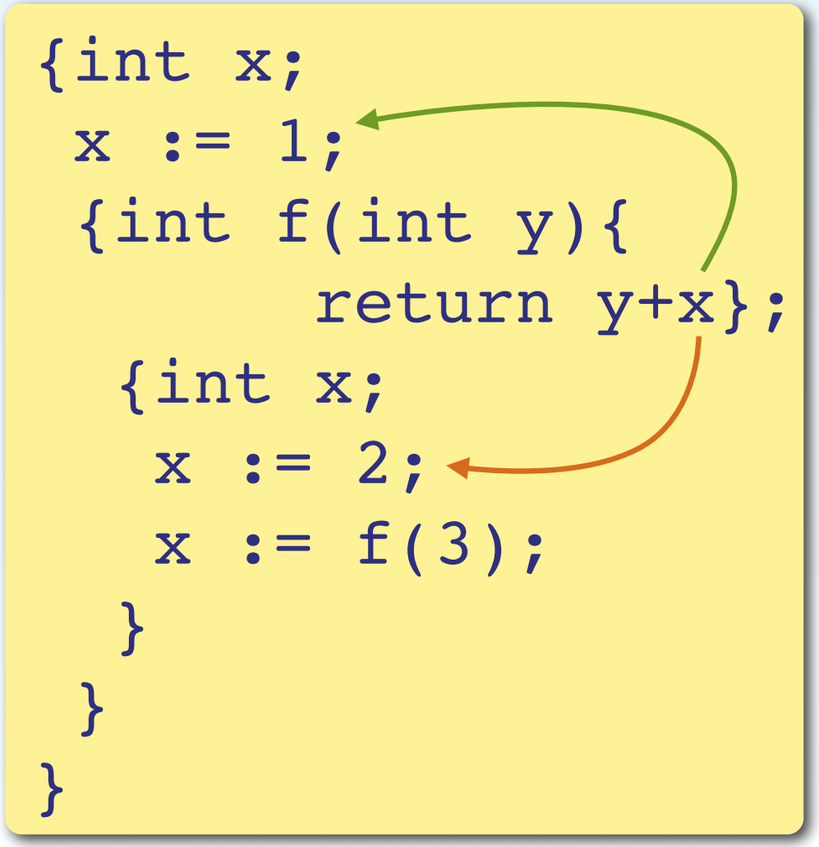
[]	
[x:?]	push
[x:1]	
[f:{}, x:1]	push
[x:?, f:{}, x:1]	push
[x:2, f:{}, x:1]	
[y:3, x:2, f:{}, x:1]	push
[x:5, f:{}, x:1]	pop
[f:{}, x:1]	pop
[x:1]	pop
[]	pop

Dynamic Scoping

Example

FunDynScope.hs
(Functions)

Static vs. Dynamic Scoping



Static scoping: A non-local name refers to the variable that is visible (= in scope) at the *definition* of a function

Dynamic scoping: A non-local name refers to the variable that is visible (= in scope) at the *use* of a function

Static Scoping

non-local variable

```
{int x;
 x := 1;
 {int f(int y){return y+x};
  {int x;
   x := 2;
   x := f(3);
  }
 }
}
```

[]	
[x:?]	push
[x:1]	
[f:{}, x:1]	push
[x:?, f:{}, x:1]	push
[x:2, f:{}, x:1]	
[y:3, x:2, f:{}, x:1]	push
[x:4, f:{}, x:1]	pop
[f:{}, x:1]	pop
[x:1]	pop
[]	pop

Static Scoping

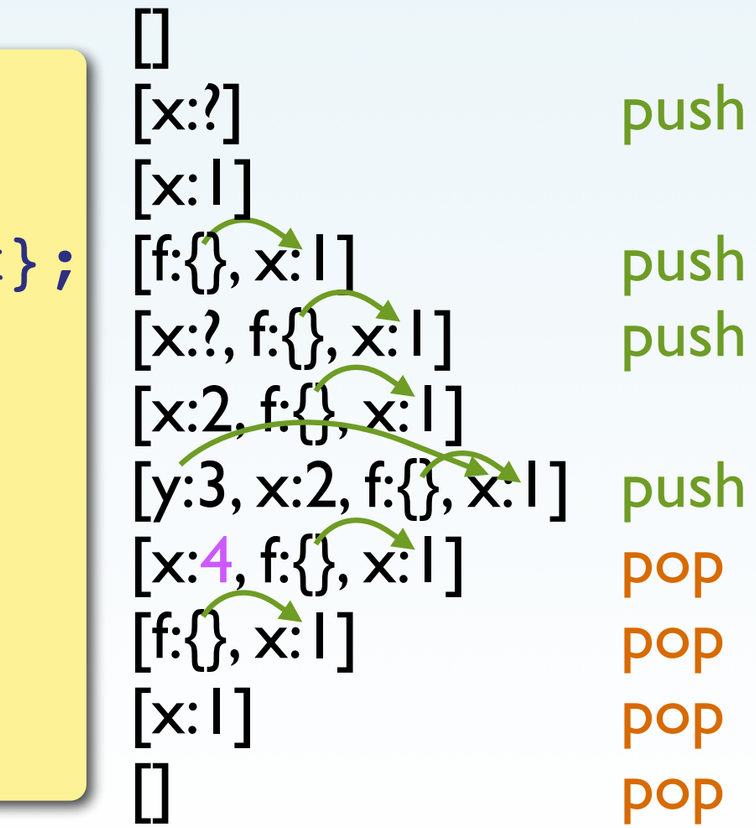
Implementation of Static Scoping

access link

How? Store a *pointer* to the previous activation record in the runtime stack with function definition

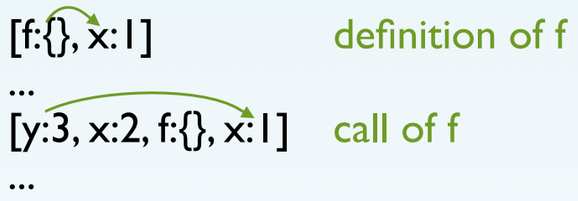
Goal: remember earlier definitions together with function definition

```
{int x;
 x := 1;
 {int f(int y){return y+x};
  {int x;
   x := 2;
   x := f(3);
  }
 }
}
```

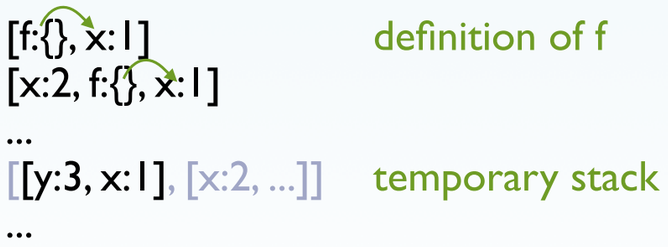


Two Interpretations of Access Links

When a function `f` (with parameter `y`) is called:



(a) Push activation record for `f` onto the runtime stack. *Follow access links* when searching for variables.



(b) Push activation record for `f` onto a temporary stack (the remainder of the runtime stack pointed to by the access link). *Evaluate `f` on temporary stack.*

Example

FunStatScope.hs
(Closures)

Dynamic vs. Static Scope: Runtime Stack

```
data Val = ...
         | F Name Expr

eval s (Fun x e)  = F x e
eval s (App f e') = case eval s f of
                      F x e → eval ((x,eval s e'):s) e
                      _     → Error
```

```
data Expr = ...
          | Fun Name Expr
```

```
data Val = ...
         | C Name Expr Stack

eval s (Fun x e)  = C x e s
eval s (App f e') = case eval s f of
                      C x e s' → eval ((x,eval s e'):s') e
                      _        → Error
```

Exercise

Show the development of the runtime stack under *static* and *dynamic* scoping for the execution of the following code.

```
{int y := 1;
 {int z := 0;
  {int f(int x){return y+x};
   {int g(int y){return f(2)};
    z := g(3);


   }
  ...
```

Exercise

Show the development of the runtime stack under *static* and *dynamic* scoping for the execution of the following code.

```
{int z := 0;
 {int f(int x){return x+1};
  {int g(int y){return f(y)};
   {int f(int x){return x-1};
    z := g(3);


   }
  ...
```

Implementation of Recursion

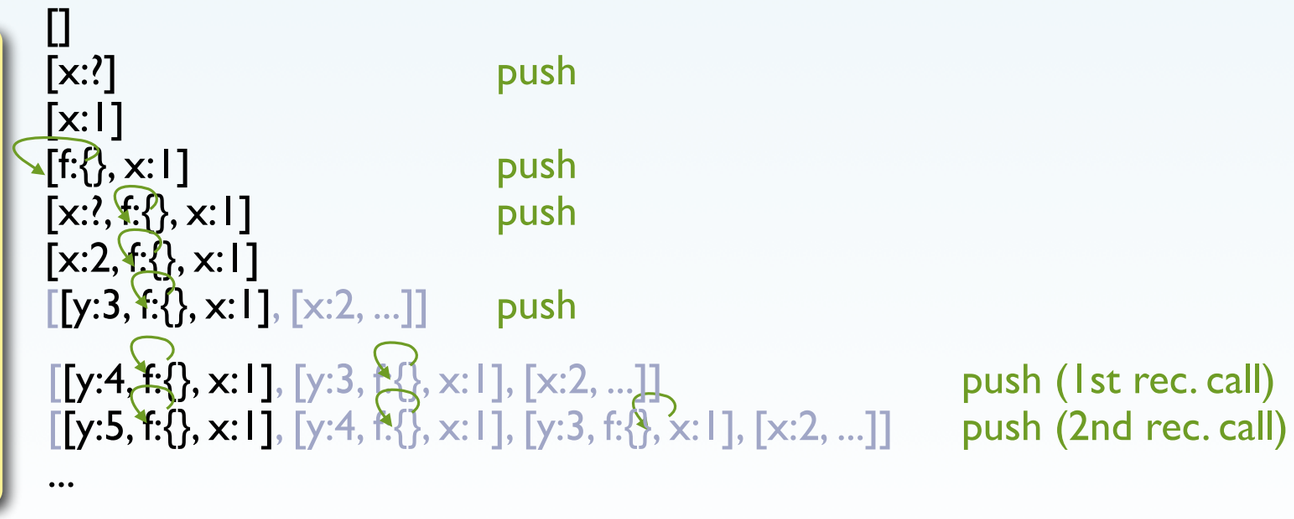
Problem: Need access to function definition when evaluating the function body

works for the 2nd interpretation of access links

Solution: Let *access link* point to the *very same* activation record in the runtime stack containing the function definition

```
{int x;
 x := 1;
 {int f(int y){return f(x+y)};
  {int x;
   x := 2;
   x := f(3);

 }
}
```



Example

FunRec.hs

7. Parameter Passing

Example

(Formal) Parameter

```
void f(int x, int y) {
     y := x+1
};
```

```
x := 3;
z := 1;
f(2*x,z);
```

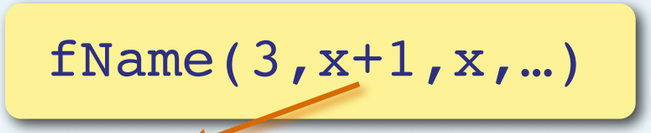
Actual Parameter, or: Argument

What is the value of z?

... it depends on the *parameter passing mechanism.*

Function Definition & Use

Function Call



6

[… ⟨x: 5⟩ …]

❶ *Evaluate argument expressions*

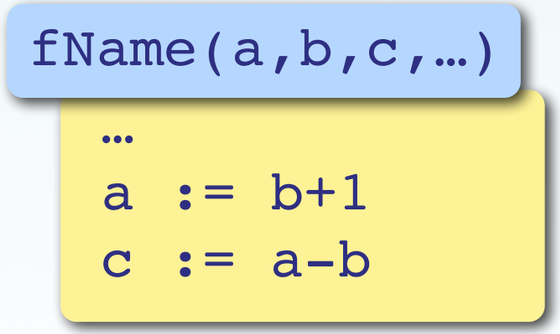
[⟨a: ?, b: ?, c: ?⟩, … ⟨x: 5⟩ …]

❷ *Create activation record with parameter names on runtime stack*

[⟨a: 3, b: 6, c: x⟩, … ⟨x: 5⟩ …]

❸ *Store values and locations in activation record*

❹ *Transfer control to function and run code*



[⟨a: 7, b: 6, c: x⟩, … ⟨x: 5⟩ …]

[⟨a: 7, b: 6, c: x⟩, … ⟨x: 1⟩ …]

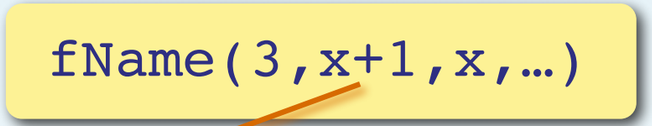
N/A

❺ *Pass values back to call site*

[… ⟨x: 1⟩ …]

❻ *Remove activation record from runtime stack*

Call-By-Value



6

[… ⟨x: 5⟩ …]

[⟨a: ?, b: ?, c: ?⟩, … ⟨x: 5⟩ …]

[⟨a: 3, b: 6, c: 5⟩, … ⟨x: 5⟩ …]

[⟨a: 7, b: 6, c: 5⟩, … ⟨x: 5⟩ …]

[⟨a: 7, b: 6, c: 1⟩, … ⟨x: 5⟩ …]

[… ⟨x: 5⟩ …]

❶ *Evaluate argument expressions*

❷ *Create activation record with parameter names on runtime stack*

❸ *Store values ~~and locations~~ in activation record*

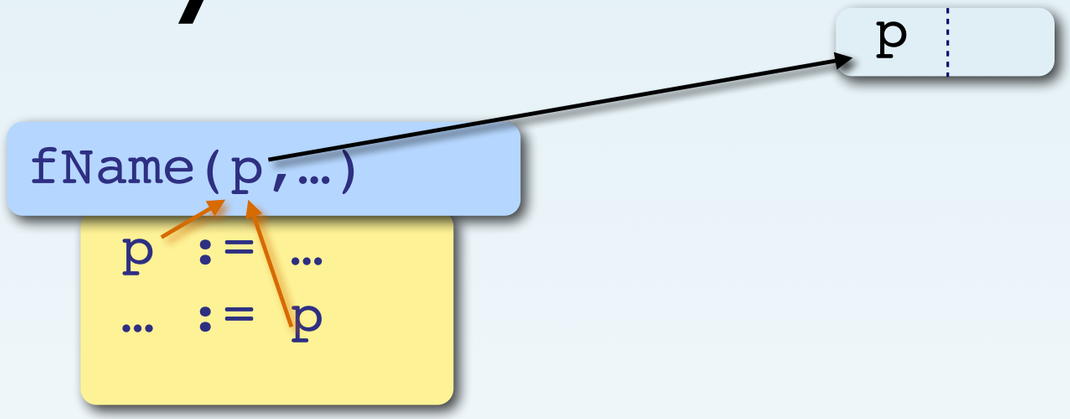
❹ *Transfer control to function and run code*

```
fName(a,b,c,…)
  …
  a := b+1
  c := a-b
```

❺ ~~*Pass values back to call site*~~

❻ *Remove activation record from runtime stack*

Call-By-Value



Synopsis

- *Evaluate argument*
- *Bind resulting value to parameter*
- *Look up value when needed*
- *Local assignments ok, but are lost after return from function call*

Exercise: Compute under CBV

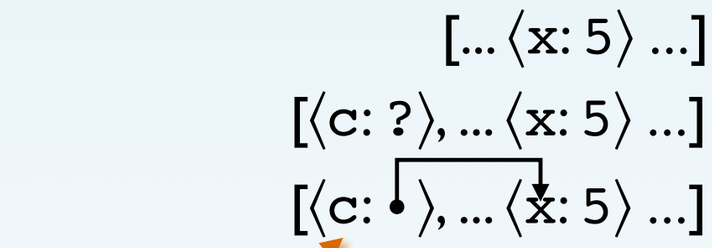
```
 1  { int z;
 2    int y;
 3    y := 5;
 4    { int f(int x){
 5        x := x+1;
 6        y := x-4;
 7        x := x+1;
 8        return x;
 9      };
10      z := f(y)+y;
11    };
12  }
```

```
   []
 1 [z:?]
 2 [y:?, z:?]
 3 [y:5, z:?]
 4 [f:{}, y:5, z:?]
10 >>
        [x:5, f:{}, y:5, z:?]
      5 [x:6, f:{}, y:5, z:?]
      6 [x:6, f:{}, y:2, z:?]
      7 [x:7, f:{}, y:2, z:?]
      8 [res:7, x:7, f:{}, y:2, z:?]
   <<
10 [f:{}, y:2, z:9]
...
```

Call-By-Reference

```
fName(3,x+1,x,…)
```

Only variables can be arguments



Alias, or: Synonym

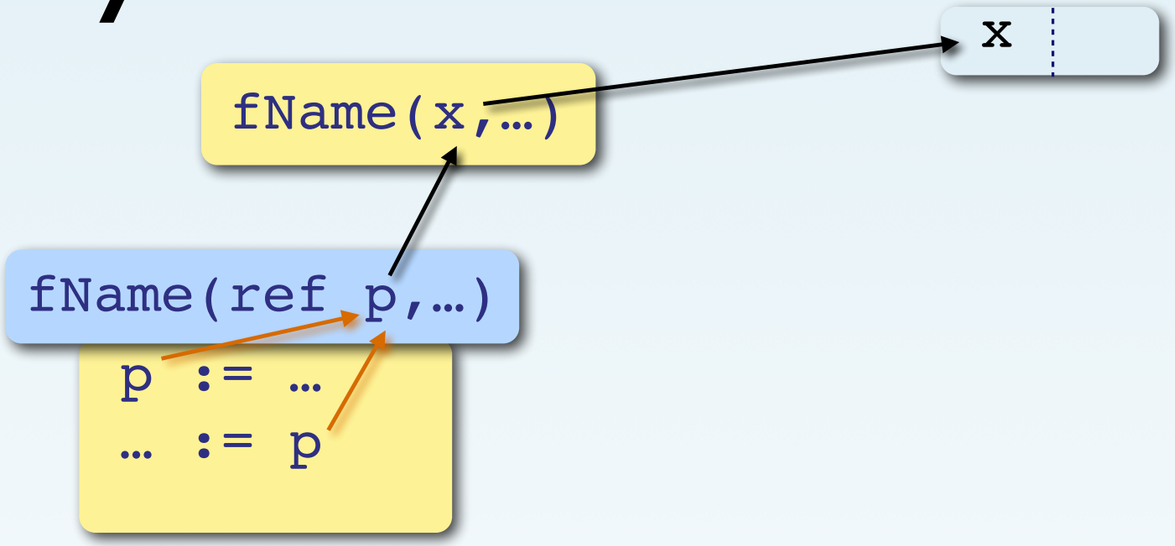
1. ~~*Evaluate argument expressions*~~
2. *Create activation record with parameter names on runtime stack*
3. *Store ~~values and~~ locations in activation record*
4. *Transfer control to function and run code*

```
fName(ref c,…)
…
c := c-2
```

Indicating parameter passing schema

5. ~~*Pass values back to call site*~~
6. *Remove activation record from runtime stack*

Call-By-Reference



Synopsis

- *Parameter* `p` *points to variable* `x` *passed as argument*
- *Parameter* `p` *is just another name for* `x`
- *Every read/write access to* `p` *acts on* `x`
- *Only variables can be passed as arguments*

Exercise: Compute under CBR

```
 1  {int z;
 2   int y;
 3   y := 5;
 4   {int f(ref int x){
 5       x := x+1;
 6       y := x-4;
 7       x := x+1;
 8       return x;
 9   };
10   z := f(y)+y;
11  };
12 }
```

```
   []
 1 [z:?]
 2 [y:?, z:?]
 3 [y:5, z:?]
 4 [f:{}, y:5, z:?]
10 >>
       [x->y, f:{}, y:5, z:?]
     5 [x->y, f:{}, y:6, z:?]
     6 [x->y, f:{}, y:2, z:?]
     7 [x->y, f:{}, y:3, z:?]
     8 [res:3, x->y, f:{}, y:3, z:?]
       <<
10 [f:{}, y:3, z:6]
...
```

Call-By-Value-Result

```
fName(3,x+1,x,…)
```

Only variables can be arguments

[… ⟨x: 5⟩ …]

❶ ~~*Evaluate argument expressions*~~

[⟨c: ?⟩, … ⟨x: 5⟩ …]

❷ *Create activation record with parameter names on runtime stack*

[⟨c: 5⟩, … ⟨x: 5⟩ …]

❸ *Store values ~~and locations~~ in activation record*

❹ *Transfer control to function and run code*

```
fName(valres c,…)
  …
  c := c-2
```

Indicating parameter passing schema

[⟨c: 3⟩, … ⟨x: 5⟩ …]

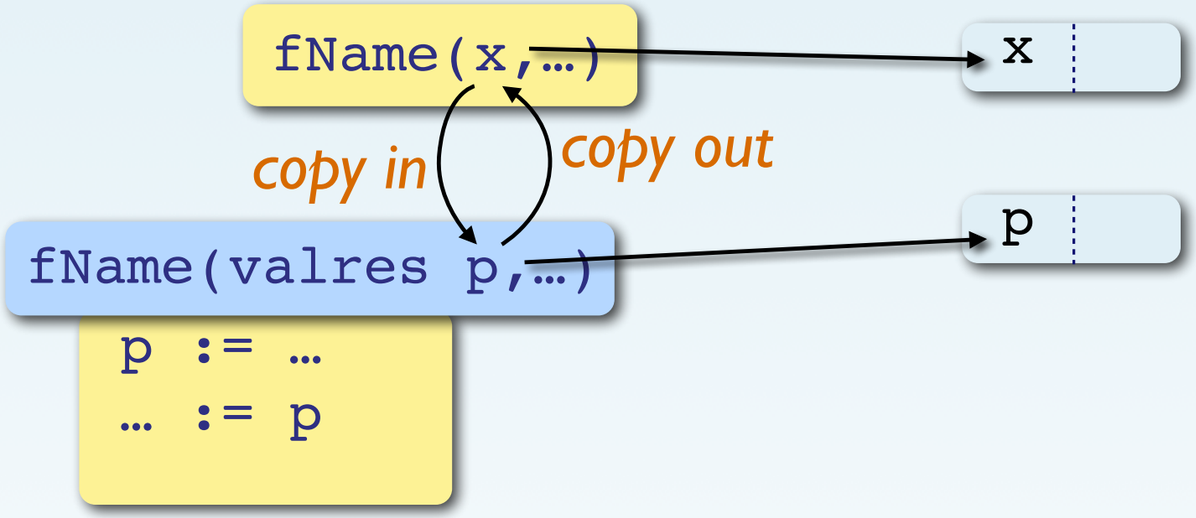
[⟨c: 3⟩, … ⟨x: 3⟩ …]

❺ *Pass values back to call site*

[… ⟨x: 3⟩ …]

❻ *Remove activation record from runtime stack*

Call-By-Value-Result



Synopsis

- *Copy value of argument variable* `x` *to parameter* `p`
- *Every read/write access to* `p` *acts on* `p`
- *Copy value of parameter* `p` *back to argument variable* `x`
- *Only variables can be passed as arguments*

Call-By-Value-Result & Function Results

When exactly does the copy-out step happen when returning from a function call?
Before or after the function result is set?

```
int f(valres int y) {
  y := 7;
  return 5
}
```

[⟨x: 0⟩ ...]
[⟨y: 0⟩, ⟨x: 0⟩ ...]
[⟨y: 7⟩, ⟨x: 0⟩ ...]
[⟨res: 5, y: 7⟩, ⟨x: 0⟩ ...]

Does it matter?

Yes!

```
x := f(x);
```

"before" [⟨x: 5⟩ ...] *"after"* [⟨x: 7⟩ ...]

defined behavior

Exercise: Compute under CBVR

```
 1  {int z;
 2   int y;
 3   y := 5;
 4   {int f(valres x){
 5       x := x+1;
 6       y := x-4;
 7       x := x+1;
 8       return x;
 9   };
10   z := f(y)+y;
11  };
12 }
```

```
   []
 1 [z:?]
 2 [y:?, z:?]
 3 [y:5, z:?]
 4 [f:{}, y:5, z:?]
10 >>
       [x:5, f:{}, y:5, z:?]
     5 [x:6, f:{}, y:5, z:?]
     6 [x:6, f:{}, y:2, z:?]
     7 [x:7, f:{}, y:2, z:?]
     8 [res:7, x:7, f:{}, y:2, z:?]
     9 [res:7, x:7, f:{}, y:7, z:?]
       <<
10 [f:{}, y:7, z:14]
...
```

Exercise: Compute under CBVR (2)

```
 1  {int z;
 2   int y;
 3   y := 5;
 4   {int f(valres x){
 5       x := x+1;
 6       y := x-4;
 7       x := x+1;
 8       return x;
 9   };
10   z := y+f(y);
11  };
12 }
```

```
   []
 1 [z:?]
 2 [y:?, z:?]
 3 [y:5, z:?]
 4 [f:{}, y:5, z:?]
10 >>
        [x:5, f:{}, y:5, z:?]
      5 [x:6, f:{}, y:5, z:?]
      6 [x:6, f:{}, y:2, z:?]
      7 [x:7, f:{}, y:2, z:?]
      8 [res:7, x:7, f:{}, y:2, z:?]
      9 [res:7, x:7, f:{}, y:7, z:?]
        <<
10 [f:{}, y:7, z:12]
...
```

Comparison: CBV, CBR, CBVR

```
 1  {int z;
 2   int y;
 3   y := 5;
 4   {int f(? x){
 5      x := x+1;
 6      y := x-4;
 7      x := x+1;
 8      return x;
 9   };
10   z := f(y)+y;
11  };
12 }
```

CBV

```
  [x:5, f:{}, y:5, z:?]
5 [x:6, f:{}, y:5, z:?]
6 [x:6, f:{}, y:2, z:?]
7 [x:7, f:{}, y:2, z:?]
8 [res:7, x:7, f:{}, y:2, z:?]
<<
10 [f:{}, y:2, z:9]
```

CBR

```
  [x->y, f:{}, y:5, z:?]
5 [x->y, f:{}, y:6, z:?]
6 [x->y, f:{}, y:2, z:?]
7 [x->y, f:{}, y:3, z:?]
8 [res:3, x->y, f:{}, y:3, z:?] <<
10 [f:{}, y:3, z:6]
```

```
  [x:5, f:{}, y:5, z:?]
5 [x:6, f:{}, y:5, z:?]
6 [x:6, f:{}, y:2, z:?]
7 [x:7, f:{}, y:2, z:?]
8 [res:7, x:7, f:{}, y:2, z:?]
9 [res:7, x:7, f:{}, y:7, z:?] <<
10 [f:{}, y:7, z:14]
```

CBVR

Exercise: Compute under CBV

```
1 { int x := 5;
2   int f(int y) {
3     x := y-4;
4     y := y+1;
5     return y;
6   };
7   int z := x+f(x);
8 }
```

```
2 [f:{}, x:5]
7 >> call f:
        [y:5, f:{}, x:5]
     3 [y:5, f:{}, x:1]
     4 [y:6, f:{}, x:1]
     5 [res:6, y:6, f:{}, x:1]
   << return from f
7 [z:11, f:{}, x:1]
```

Exercise: Compute under CBR

```
1 { int x := 5;
2   int f(int y) {
3     x := y-4;
4     y := y+1;
5     return y;
6   };
7   int z := x+f(x);
8 }
```

```
2 [f:{}, x:5]
7 >> call f:
        [y->x, f:{}, x:5]
     3 [y->x, f:{}, x:1]
     4 [y->x, f:{}, x:2]
     5 [res:2, y->x, f:{}, x:2]
   << return from f
7 [z:7, f:{}, x:2]
```

Exercise: Compute under CBVR

```
1 { int x := 5;
2   int f(int y) {
3     x := y-4;
4     y := y+1;
5     return y;
6   };
7   int z := x+f(x);
8 }
```

```
2 [f:{}, x:5]
7 >> call f:
        [y:5, f:{}, x:5]
     3 [y:5, f:{}, x:1]
     4 [y:6, f:{}, x:1]
     5 [res:6, y:6, f:{}, x:1]
   << return from f
7 [z:11, f:{}, x:6]
```

Call-By-Name

```
fName(x+1,…)
```

[… ⟨x: 5⟩ …]

❶ ~~*Evaluate argument expressions*~~

[⟨a: ?⟩, … ⟨x: 5⟩ …]

❷ *Create activation record with parameter names on runtime stack*

[⟨a: x+1⟩, … ⟨x: 5⟩ …]

❸ *Store ~~values and locations~~ expressions in activation record*

❹ *Transfer control to function and run code*

```
fName(name a,…)
    z := a-2
    x := 2
    z := 2*a
```

local variable

Indicating parameter passing schema

[⟨z: 4, a: x+1⟩, … ⟨x: 5⟩ …]

[⟨z: 4, a: x+1⟩, … ⟨x: 2⟩ …]

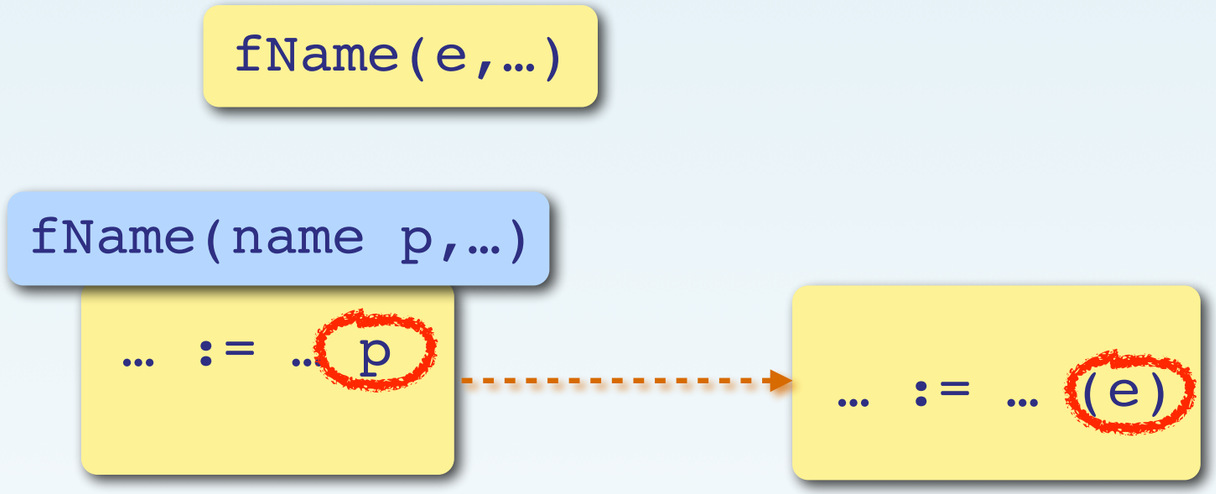
[⟨z: 6, a: x+1⟩, … ⟨x: 2⟩ …]

❺ ~~*Pass values back to call site*~~

[… ⟨x: 2⟩ …]

❻ *Remove activation record from runtime stack*

Call-By-Name



Synopsis

- *Bind argument expression* `e` *to parameter* `p`
- *Every read access to* `p` *evaluates* `e` *anew*
- *Imagine substituting* `(e)` *for* `p` *in the body of* `fName`
- *Assignments to parameter are not allowed*

Call-By-Need

```
fName(x+1,…)
```

[… ⟨x: 5⟩ …]

❶ ~~*Evaluate argument expressions*~~

[⟨a: ?⟩, … ⟨x: 5⟩ …]

❷ *Create activation record with parameter names on runtime stack*

[⟨a: x+1⟩, … ⟨x: 5⟩ …]

❸ *Store ~~values and locations~~ expressions, replaced by values in activation record*

❹ *Transfer control to function and run code*

```
fName(need a,…)
    z := a-2
    x := 2
    z := 2*a
```

local variable

Indicating parameter passing schema

[⟨z: ?, a: x+1⟩, … ⟨x: 5⟩ …]

[⟨z: 4, a: 6⟩, … ⟨x: 5⟩ …]

[⟨z: 4, a: 6⟩, … ⟨x: 2⟩ …]

[⟨z: 12, a: 6⟩, … ⟨x: 2⟩ …]

❺ ~~*Pass values back to call site*~~

[… ⟨x: 2⟩ …]

❻ *Remove activation record from runtime stack*

Call-By-Need

Synopsis

- *Similar to call-by-name, except:*
- *First read access to* `p` *evaluates* `e` *to* `v` *and stores* `v` *in* `p`
- *Subsequent accesses to* `p` *retrieve* `v`

Exercise: Compute under CBN

```
1 { int y;
2   y := 4;
3   { int f(int x) {
4       y := 2*x;
5       return (y+x);
6     };
7     y := f(y+3);
8   };
9 }
```

```
2 [y:4]
3 [f:{}, y:4]
7 >>
      [x:y+3, f:{}, y:4]      |    x ⇒ 7
    4 [x:y+3, f:{}, y:14]     |    x ⇒ 17
    5 [res:31, x:y+3, f:{}, y:14]
    <<
7 [f:{}, y:31]
```

Exercise: Compute under CBNeed

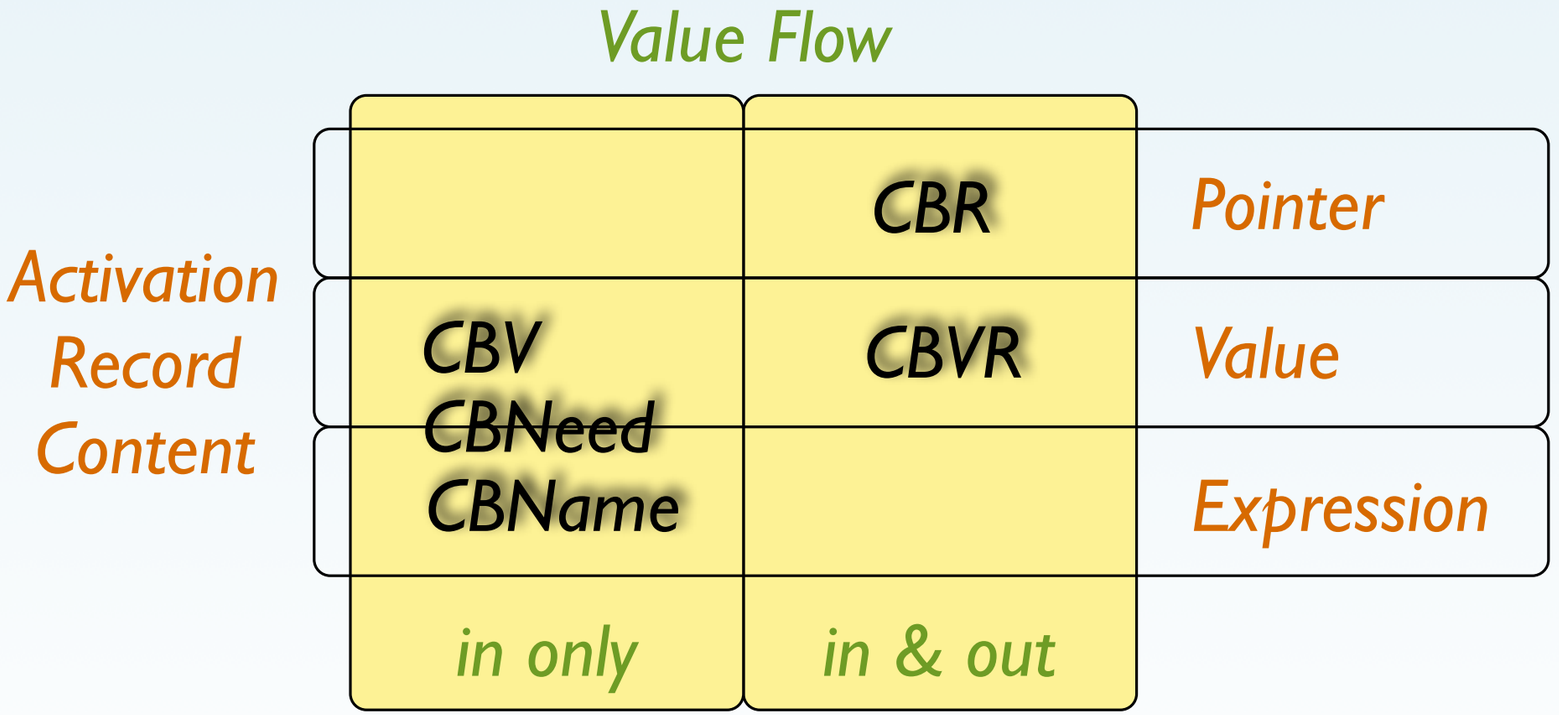
```
1 { int y;
2   y := 4;
3   { int f(int x) {
4       y := 2*x;
5       return (y+x);
6     };
7     y := f(y+3);
8   };
9 }
```

```
2 [y:4]
3 [f:{}, y:4]
7 >>
      [x:y+3, f:{}, y:4]          |    x ⇒ 7
    4 [x:7,   f:{}, y:14]         |    x ⇒ 7
    5 [res:21, x:7, f:{}, y:14]
    <<
7 [f:{}, y:21]
```

Comparison

	Call-By-...				
	Value	***Reference***	***Value-Result***	***Name***	***Need***
restriction on argument		var only	var only		
restriction on parameter				no assignment	no assignment
what is stored in activation record	value	pointer	value	expr	expr/value
how to access parameter value	lookup	deref	lookup	eval	eval, then lookup
on return from call			copy value to actual param		

Classification



Example Languages

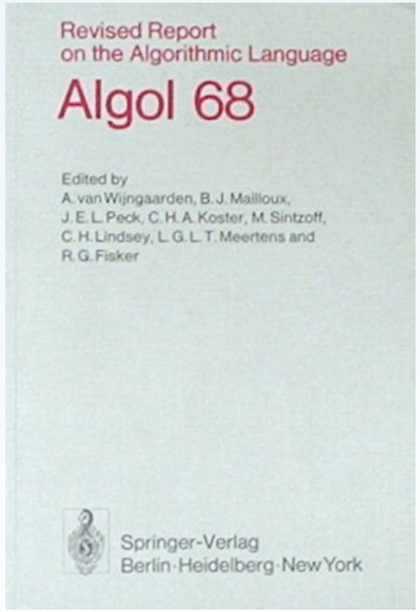
Call-by-Value

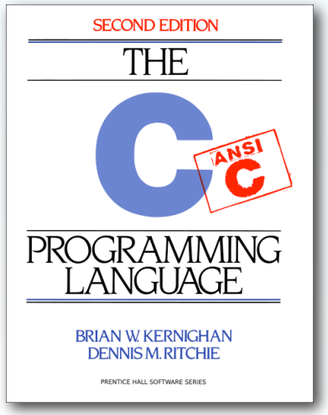
Call-by-Reference

Call-by-Value-Result

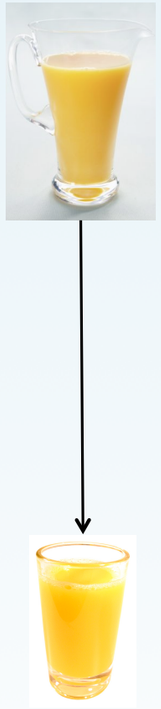
Call-by-Name

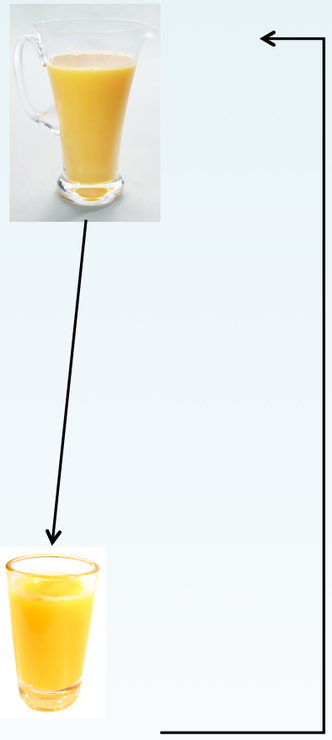
Call-by-Need





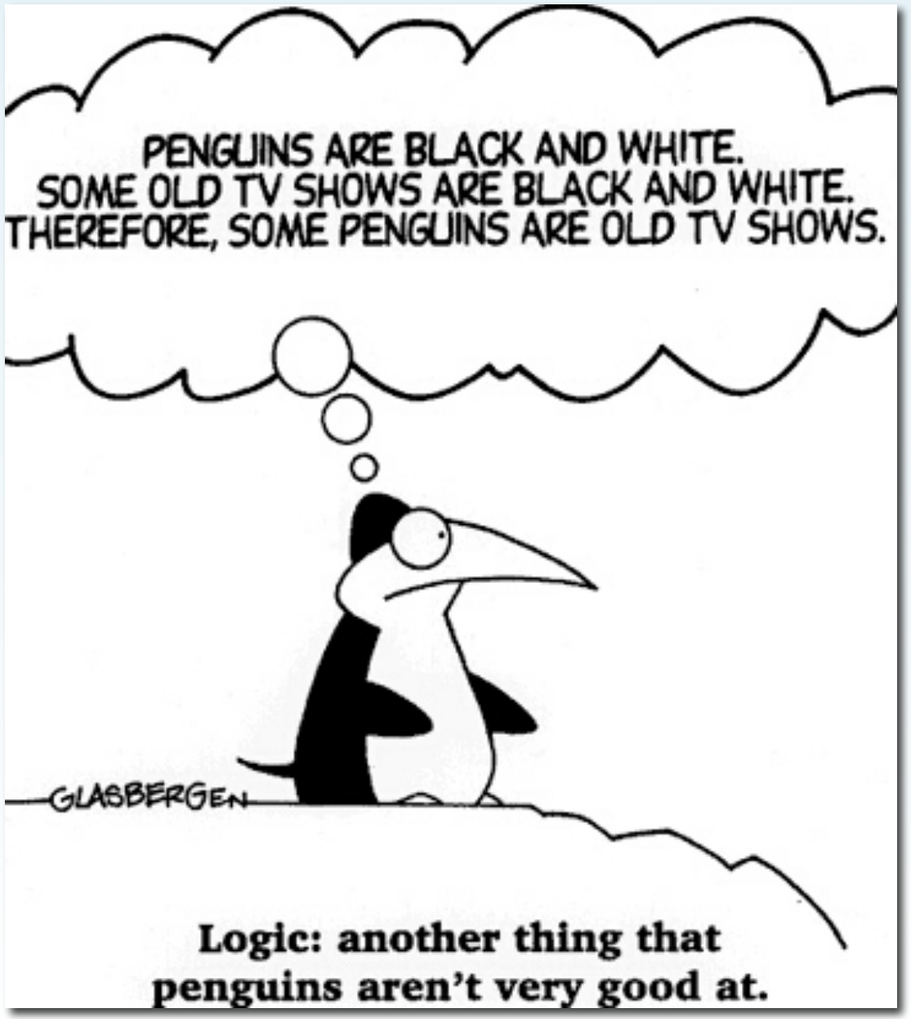
Pass the Orange Juice …





Call By:	Value	Reference	Value/Result	Name	Need

9 Prolog


PENGUINS ARE BLACK AND WHITE.
SOME OLD TV SHOWS ARE BLACK AND WHITE.
THEREFORE, SOME PENGUINS ARE OLD TV SHOWS.
GLASBERGEN
Logic: another thing that penguins aren't very good at.

9 Prolog

Introduction

Predicates & Goals

Rules

Satisfying Goals, Recursion & Backtracking

Equality

Terms & Lists

Arithmetic

The Cut & Negation

Prolog Pitfalls

9.1 Introduction

What is Prolog?

- *Untyped logic* programming language
- Computations are expressed by *rules* that define *relations* on objects
- Running a program/computing a value: Formulating a *goal* or *query*
- Result of program execution: Yes/No answer and a *binding of free variables*
- No higher-order predicates

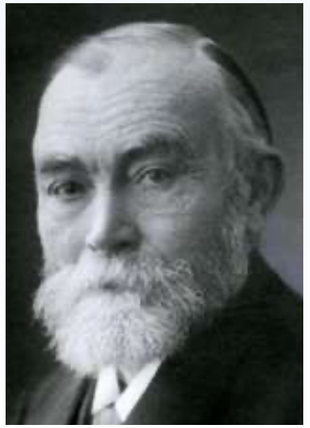
Logic: A Tool for Reasoning

Syllogism (logical argument) (*Aristotle, 350 BCE*)

E.g.: *All humans are mortal.*
Socrates is human.
Therefore: Socrates is mortal.

First-Order Logic (*Gottlob Frege, 1879 "Begriffsschrift"*)

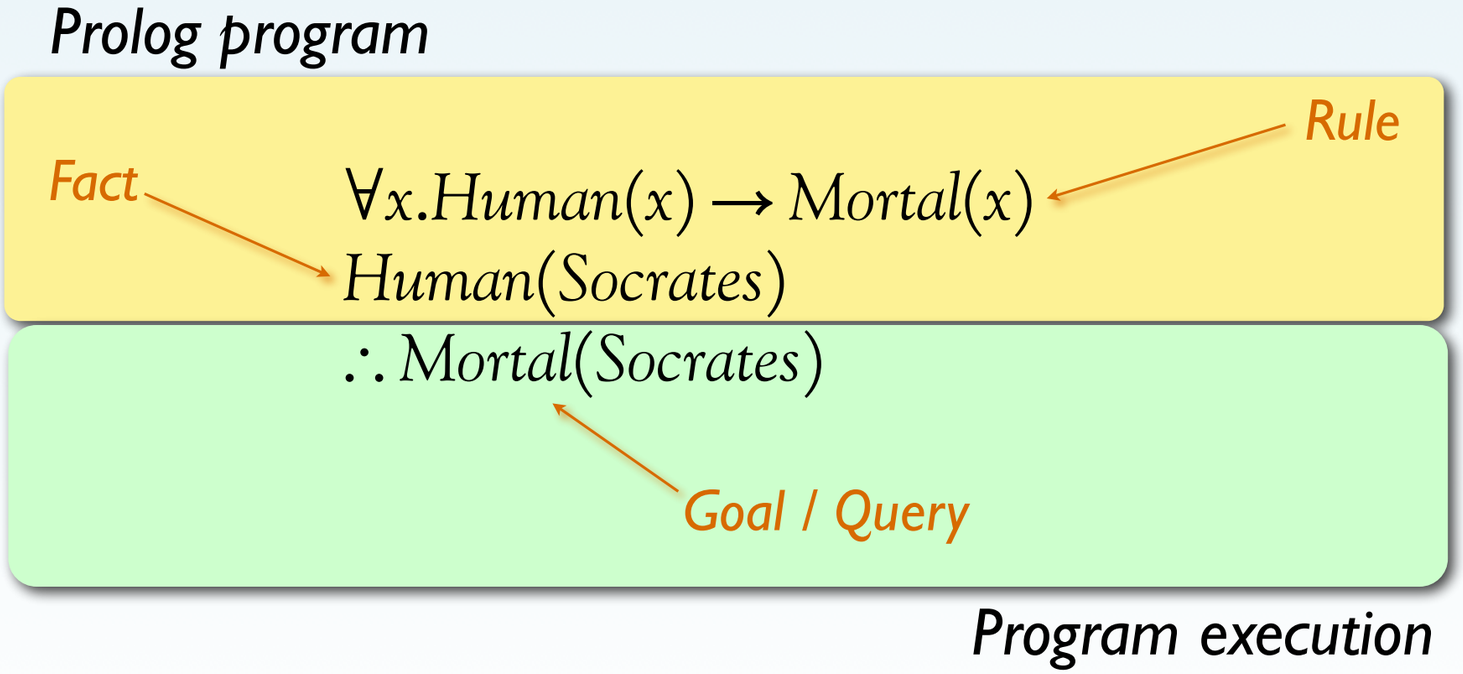
E.g.: $\forall x.Human(x) \rightarrow Mortal(x)$
$Human(Socrates)$
$\therefore Mortal(Socrates)$



Using Logic

Video Clip

Logic & Programming



Predicates, Functions & Sets

Nullary Predicate ≅ *Set A*

$$Term = \{\texttt{true}, \texttt{false}, \texttt{not true}, \texttt{not false}, \ldots\}$$

≈ *SQL query over table (with schema A, A×B, …)*

Unary Predicate (over A) ≅ *A* → *Bool* ≅ *Subset of A*

$$Even : \mathbb{N} \to \mathbb{B} = \{(\texttt{0}, \texttt{true}), (\texttt{1}, \texttt{false}), (\texttt{2}, \texttt{true}), \ldots\} \cong \{\texttt{0}, \texttt{2}, \texttt{4}, \ldots\} \subseteq \mathbb{N}$$

Binary Predicate (over A and B) ≅ *A* × *B* → *Bool* ≅ *Subset of A* × *B*

$$< : \mathbb{N} \times \mathbb{N} \to \mathbb{B} = \{((\texttt{0}, \texttt{0}), \texttt{false}), ((\texttt{0}, \texttt{1}), \texttt{true}), \ldots, ((\texttt{5}, \texttt{3}), \texttt{false}), \ldots, ((\texttt{5}, \texttt{7}), \texttt{true}), \ldots\}$$
$$\cong \{(\texttt{0}, \texttt{1}), \ldots, (\texttt{5}, \texttt{7}), \ldots\} \subseteq \mathbb{N} \times \mathbb{N}$$

SWI-Prolog

swipl [<option> ...] [-f <file> ...]

?- <goal>.	solve goal
?- [<file>].	load definitions from a file (file name w/o ".pl" extension)
?- help.	help on predicates
?- trace.	Turn on tracing of next goal
?- halt.	quit

On flip:
/usr/local/apps/swipl/current/bin/swipl

9.2 Predicates & Goals

The basic entities of Prolog are *predicates*
(Note: *predicate* ≅ *relation* ≅ *set*)

color = { red, blue }

```
color(red).
color(blue).
```

likes = { (john,red), (mary,red), (john,mary) }

fact

```
likes(john,red).
likes(mary,red).
likes(john,mary).
```

rules with empty bodies define facts

For each set S there is a corresponding predicate
$P_S(x) :\Leftrightarrow x \in S$
(*characteristic function*)

Exercises

```
likes(john,red).
likes(mary,red).
likes(john,mary).
```

Define the successor predicate on the first 5 natural numbers

Define the less than predicate on the first 3 natural numbers

Goals & Queries

A *goal* (or *query*) looks like a fact, but is a question:

Prolog prompt

```
?- color(red).
true.
```

Goal

Answer

```
?- likes(mary,red).
true.
```

```
?- color(green).
false.
```

closed world assumption

"No" means "cannot be derived" from the currently known facts

Answers are sought in a "*database*" of given facts (and rules):

Prolog program

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

likes.pl

Variables

Goals may contain variables

```
?- color(X).
X = red ↵
```

satisfied with answer

```
?- color(X).
X = red ;
X = blue.
```

want more evidence

```
?- likes(X,blue).
false.
```

```
?- likes(john,X).
X = red ;
X = mary.
```

```
?- likes(X,Y).
X = john
Y = red ;
X = mary
Y = red ;
X = john
Y = mary.
```

enumerating a predicate

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

Exercises

Which goal finds the successor of 3?

Which goal finds the predecessor of 2?

Write the goal to find all numbers that are greater than 1

```
succ(1,2).
succ(2,3).
succ(3,4).
succ(4,5).
succ(5,6).
```

```
lt(1,2).
lt(2,3).
lt(1,3).
```

Variables are "1st order"

Variables cannot be used for predicates:

⇒ *We can only ask whether a particular predicate holds.*

⇒ *We cannot ask which predicate(s) hold for objects.*

Conjunctions

Do John and Mary like each other?

```
?- likes(john,mary), likes(mary,john).
false.
```

Comma denotes logical and

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

Is there anything that John and Mary both like?

```
?- likes(john,X), likes(mary,X).
X = red ;
false.
```

X must be bound to the same value

No other solutions found

Which colors does John like?

```
?- likes(john,X), color(X).
X = red ;
false.
```

9.3 Rules

A *rule* consists of a *head* and a *body*

```
marry(X,Y) :- likes(X,Y), likes(Y,X).
```

:- denotes ⇐ (read "if")

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

```
?- marry(john,Y).
false.
```

Rules can contain *free variables*

```
friends(X,Y) :- likes(X,Z), likes(Y,Z).
```

```
?- friends(X,Y).
X = john
Y = mary
```

Are there any other solutions? Which?

More on Rules

Predicates can be defined through multiple rules:

```
marry(X,Y) :- likes(X,Y), likes(Y,X).
marry(X,Y) :- likes(X,Y), isRich(X).
marry(john,Y).
```

Facts are rules without a body

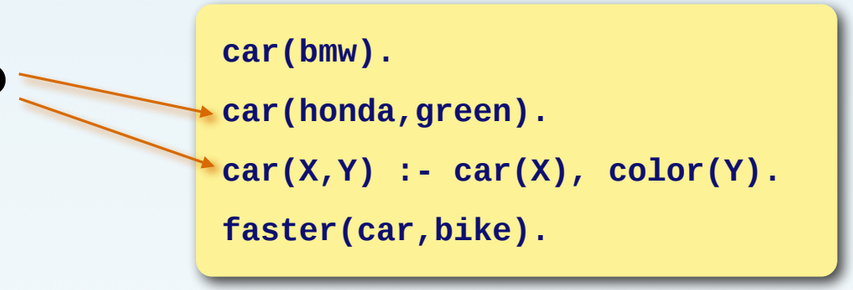
Facts are unconditional

```
color(red).
color(blue).
likes(john,red).
likes(mary,red).
likes(john,mary).
```

Overloading

Predicates are identified by name *and* arity.

car/2 refers to



```
car(bmw).
car(honda,green).
car(X,Y) :- car(X), color(Y).
faster(car,bike).
```

```
?- car(X).
X = bmw.
```

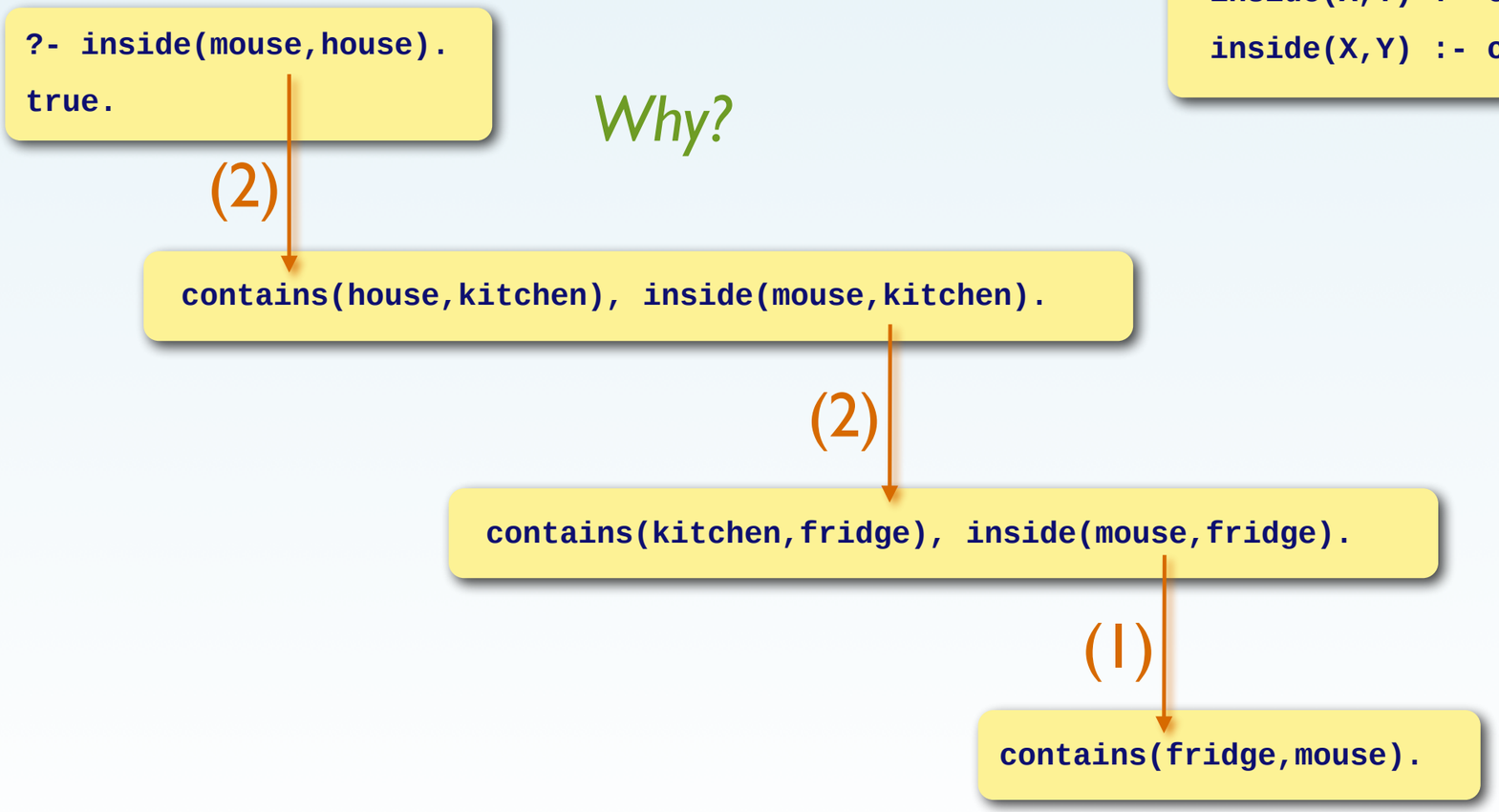
```
?- car(X,Y).
X = honda,
Y = green ;
X = bmw,
Y = red ;
X = bmw,
Y = blue.
```

References to predicates often include/require the predicate's arity

Recursive Rules

```
inside(X,Y) :- contains(Y,X).                  /* (1) */
inside(X,Y) :- contains(Y,Z), inside(X,Z).     /* (2) */
```

```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
```



Why?

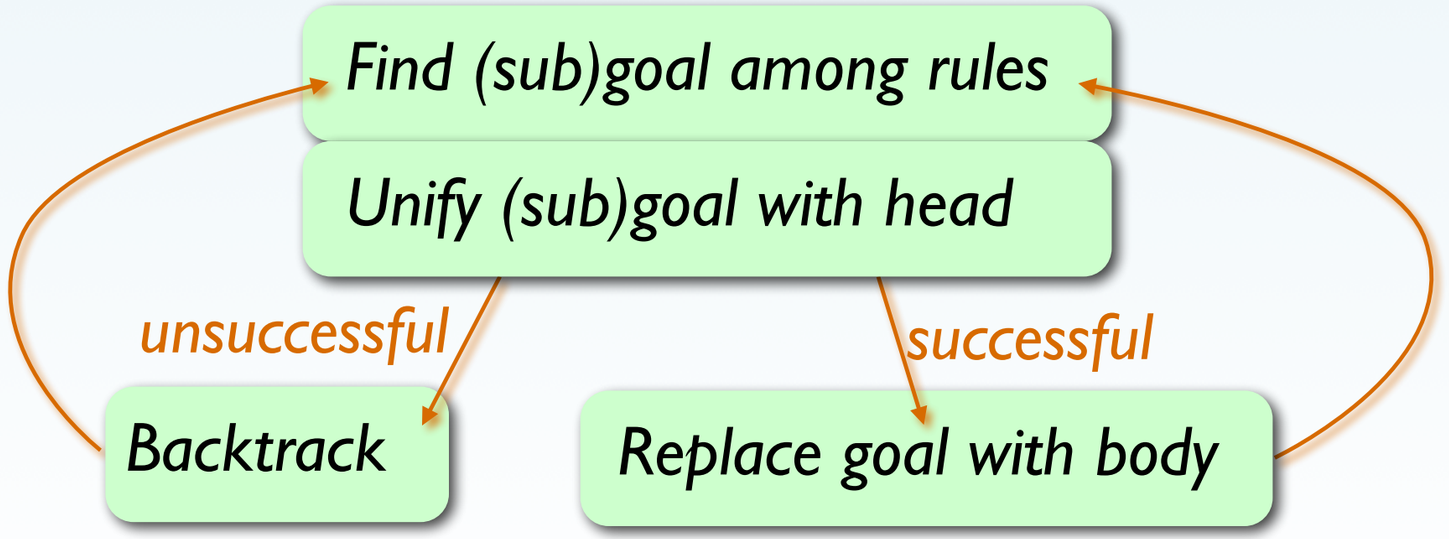
9.4 Satisfying Goals

Prolog's computational model

Logical view:

Try to refute negated goal (using SLD resolution)

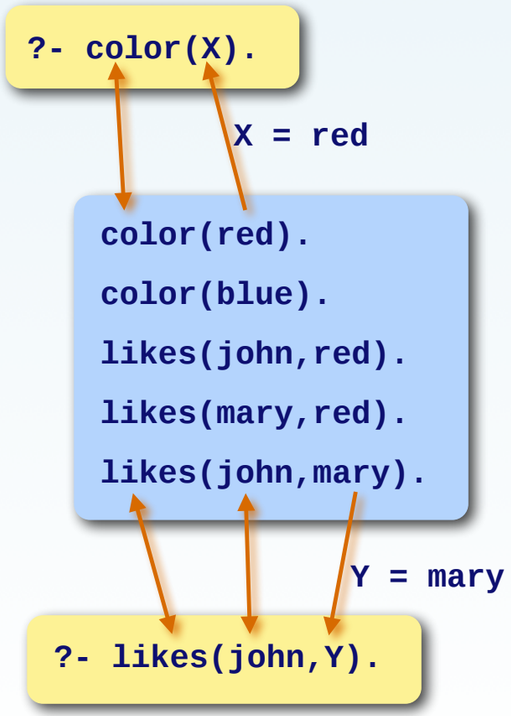
Operational view:



“Fact” Queries

Simplified view:

A goal/query is a pattern that is *matched* against *facts*.



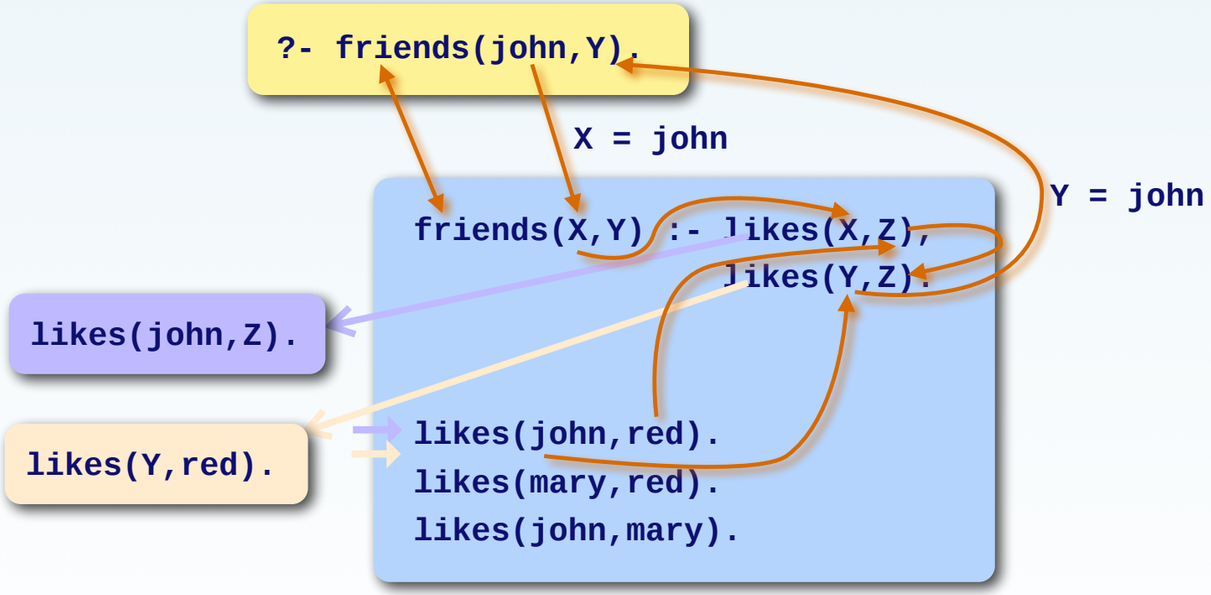
Variables in goals: Result parameters

Data flows from database to call

"View" Queries

General view:

A goal/query is a pattern that is *unified* with *rule heads*.

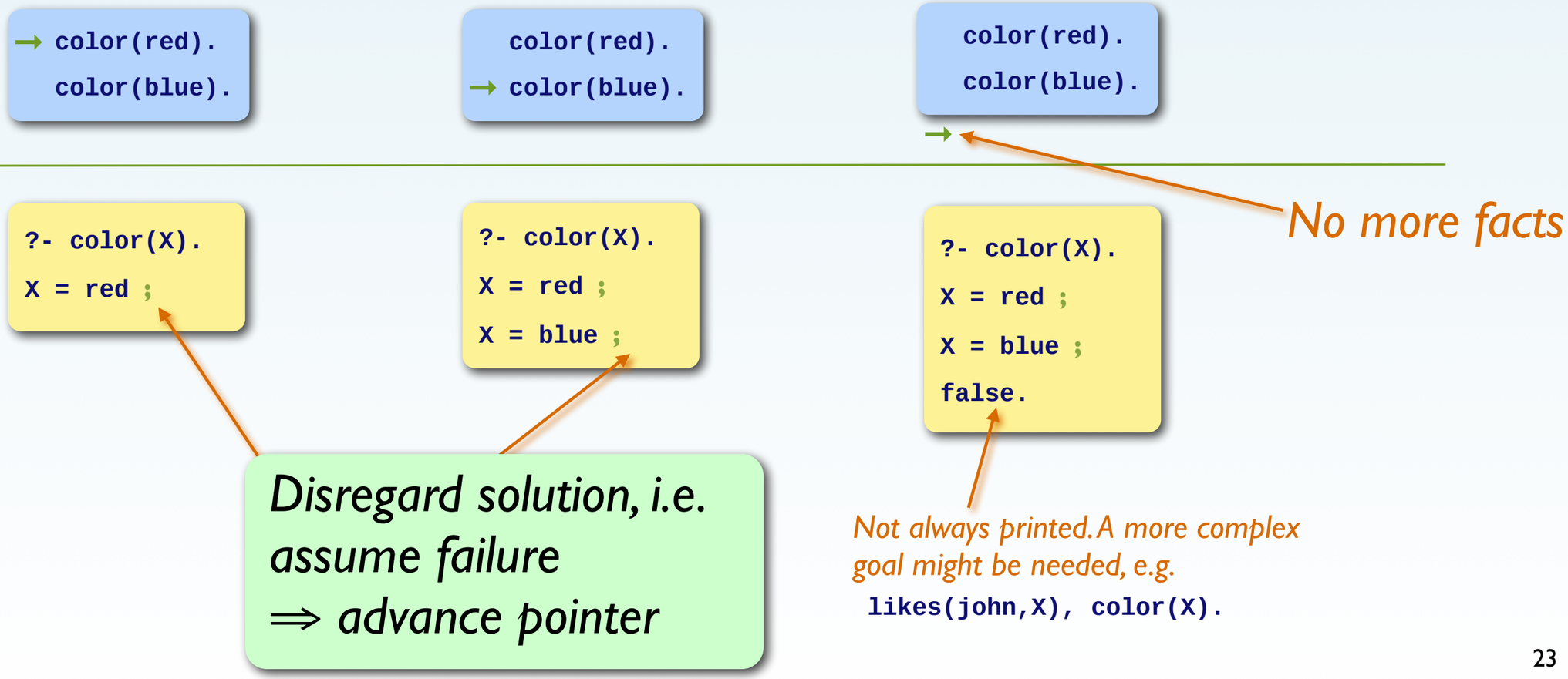


Variables in heads:
Value or result parameters

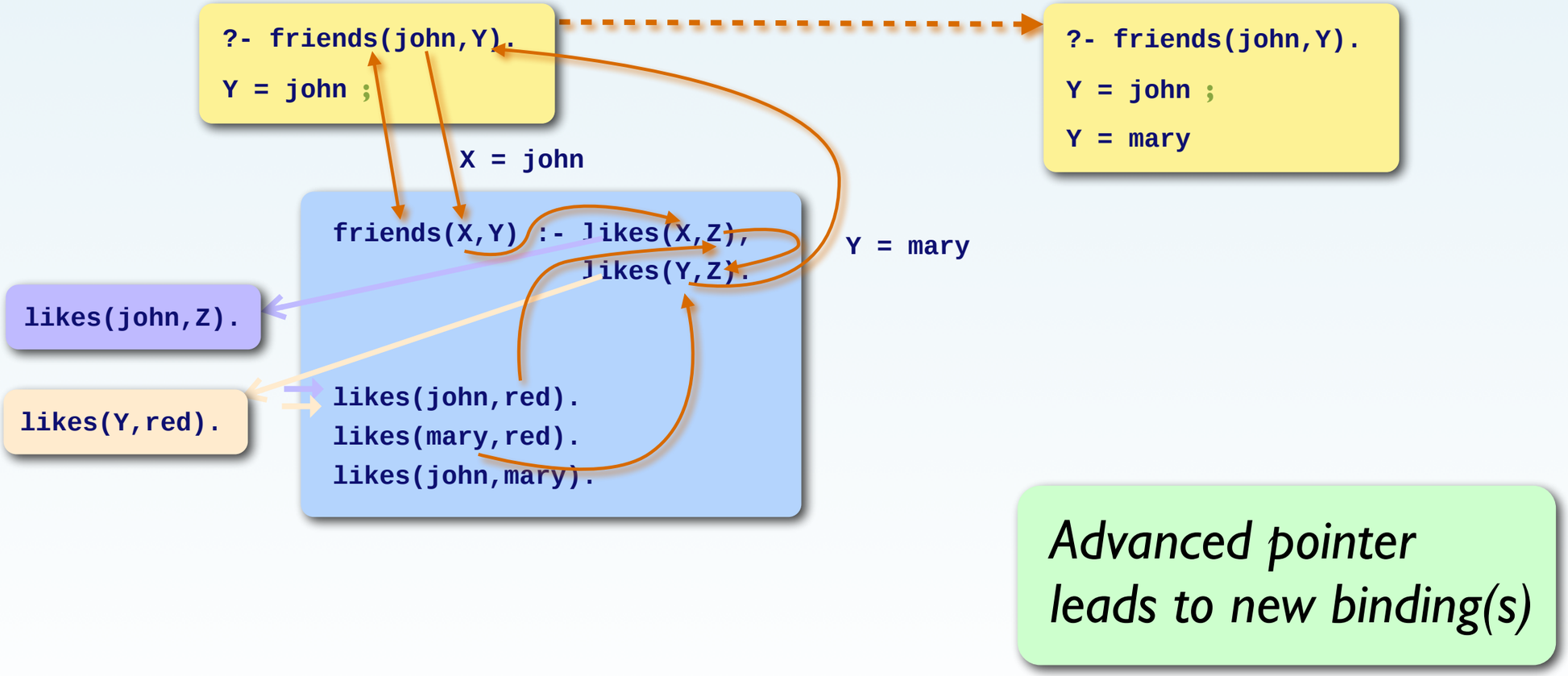
Data flows in both directions

Re-Satisfying Goals

For each (sub-)goal, imagine having a pointer "→" in the Prolog database.



Multiple Predicate Instances



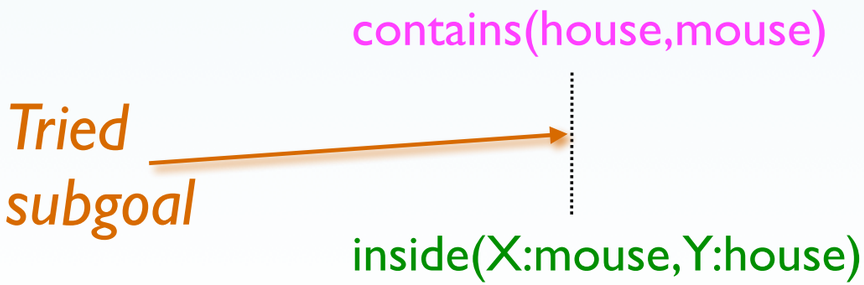
Advanced pointer leads to new binding(s)

Tree Model

A goal that is (tried to be) satisfied by a rule
H :- P_1, ..., P_n is extended by P_1, ..., P_n.
⇒ leads to a tree of goals.

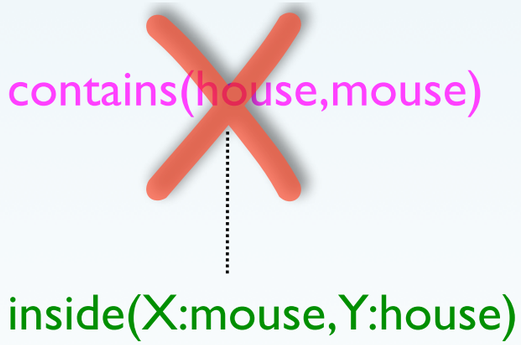
(1) To satisfy inside(mouse,house):
- select 1st rule for inside
- bind X and Y (to mouse & house)
- create subgoal contains(house,mouse)
  ⇒ create new marker "→"

```
→ contains(house,bathroom).
  contains(house,kitchen).
  contains(kitchen,fridge).
  contains(fridge,mouse).
→ inside(X,Y) :- contains(Y,X).
  inside(X,Y) :- contains(Y,Z),
                 inside(X,Z).
```



Revoking Subgoals

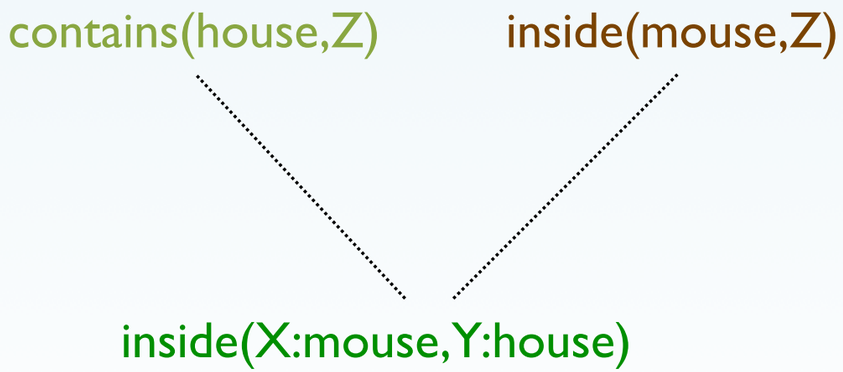
(2) Marker "→" for goal contains(house,mouse) scans the whole database, but isn't able to find such a fact (or a matching head of a rule)
⇒ goal fails and is revoked.




```
→ contains(house,bathroom).
  contains(house,kitchen).
  contains(kitchen,fridge).
  contains(fridge,mouse).
→ inside(X,Y) :- contains(Y,X).
  inside(X,Y) :- contains(Y,Z),
                 inside(X,Z).
```

Expanding Subgoals

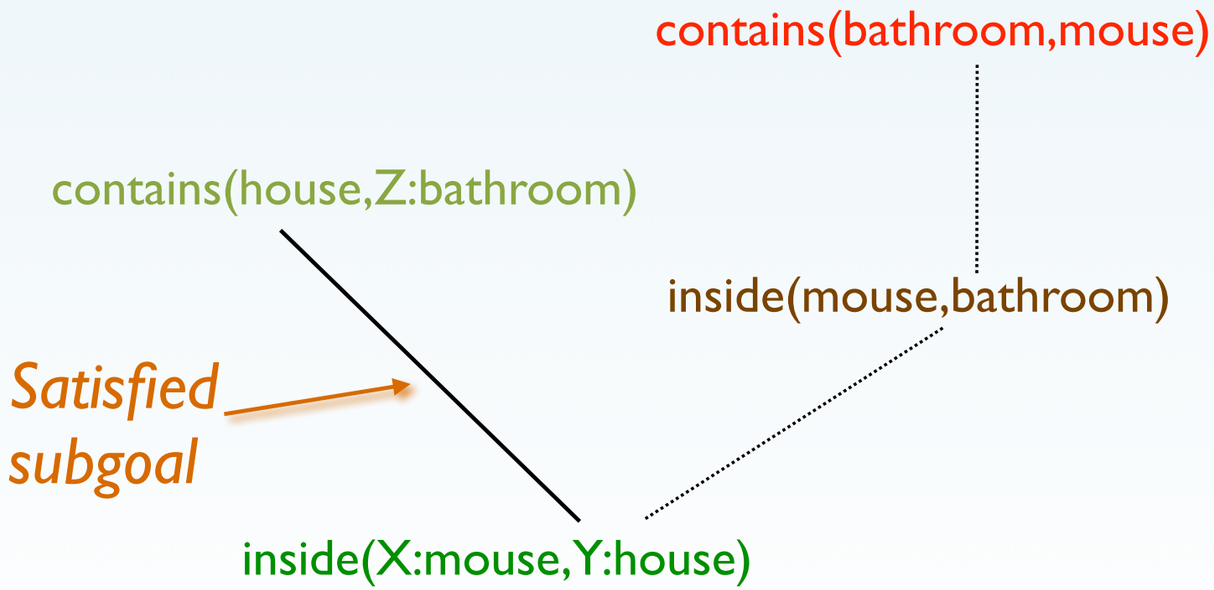
(3) Marker "→" for goal inside(mouse,house) is advanced
⇒ create new subgoals contains(house,Z) and inside(mouse,Z)
together with two new markers.




```
→ contains(house,bathroom).
  contains(house,kitchen).
  contains(kitchen,fridge).
  contains(fridge,mouse).
→ inside(X,Y) :- contains(Y,X).
  inside(X,Y) :- contains(Y,Z),
                 inside(X,Z).
```

Lots of Markers ...

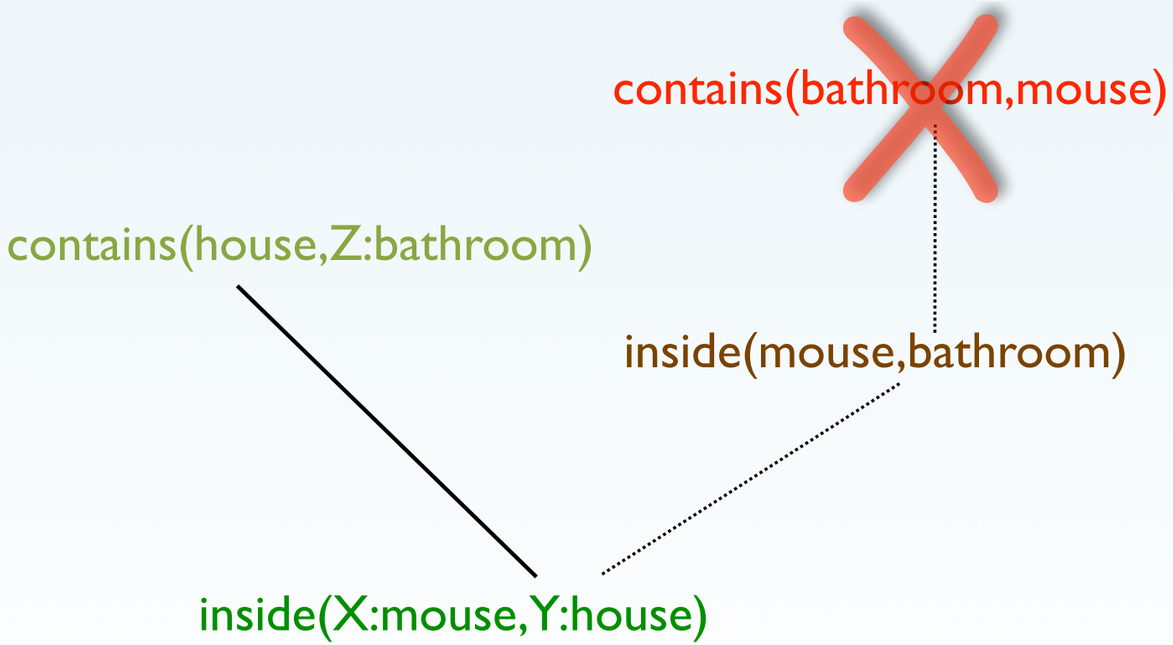
(4) Z is bound to bathroom, which causes the "contains" goal to succeed. The goal inside(mouse,bathroom) is expanded to contains(bathroom,mouse) (plus new marker "→").




```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
inside(X,Y) :- contains(Y,X).
inside(X,Y) :- contains(Y,Z),
               inside(X,Z).
```

Lots of Markers ...

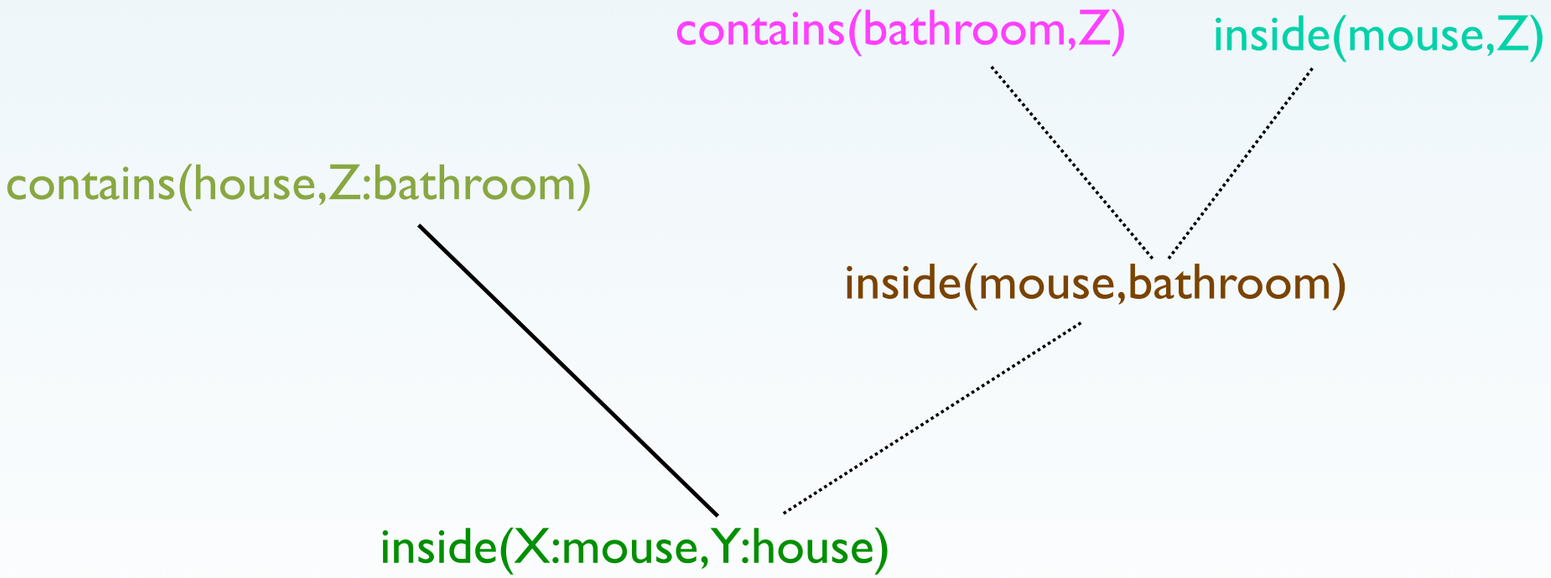
(5) contains(bathroom,mouse) cannot be satisfied and must be revoked; marker "→" for goal inside(mouse,bathroom) is advanced.




```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
inside(X,Y) :- contains(Y,X).
inside(X,Y) :- contains(Y,Z),
               inside(X,Z).
```

Even more markers ...

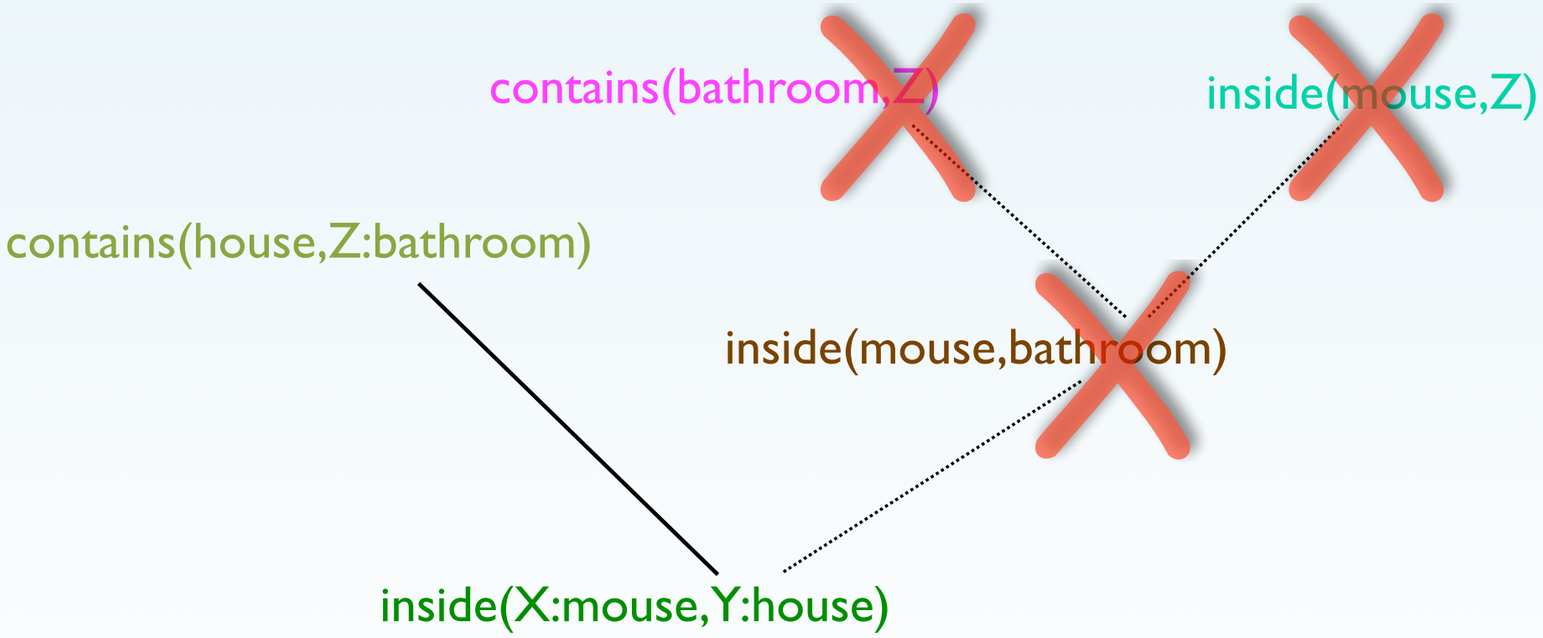
(6) 2nd rule for inside(mouse,bathroom) is expanded (and two more markers are created), but cannot succeed because no fact matches contains(bathroom,...).




```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
inside(X,Y) :- contains(Y,X).
inside(X,Y) :- contains(Y,Z),
               inside(X,Z).
```

Backtracking

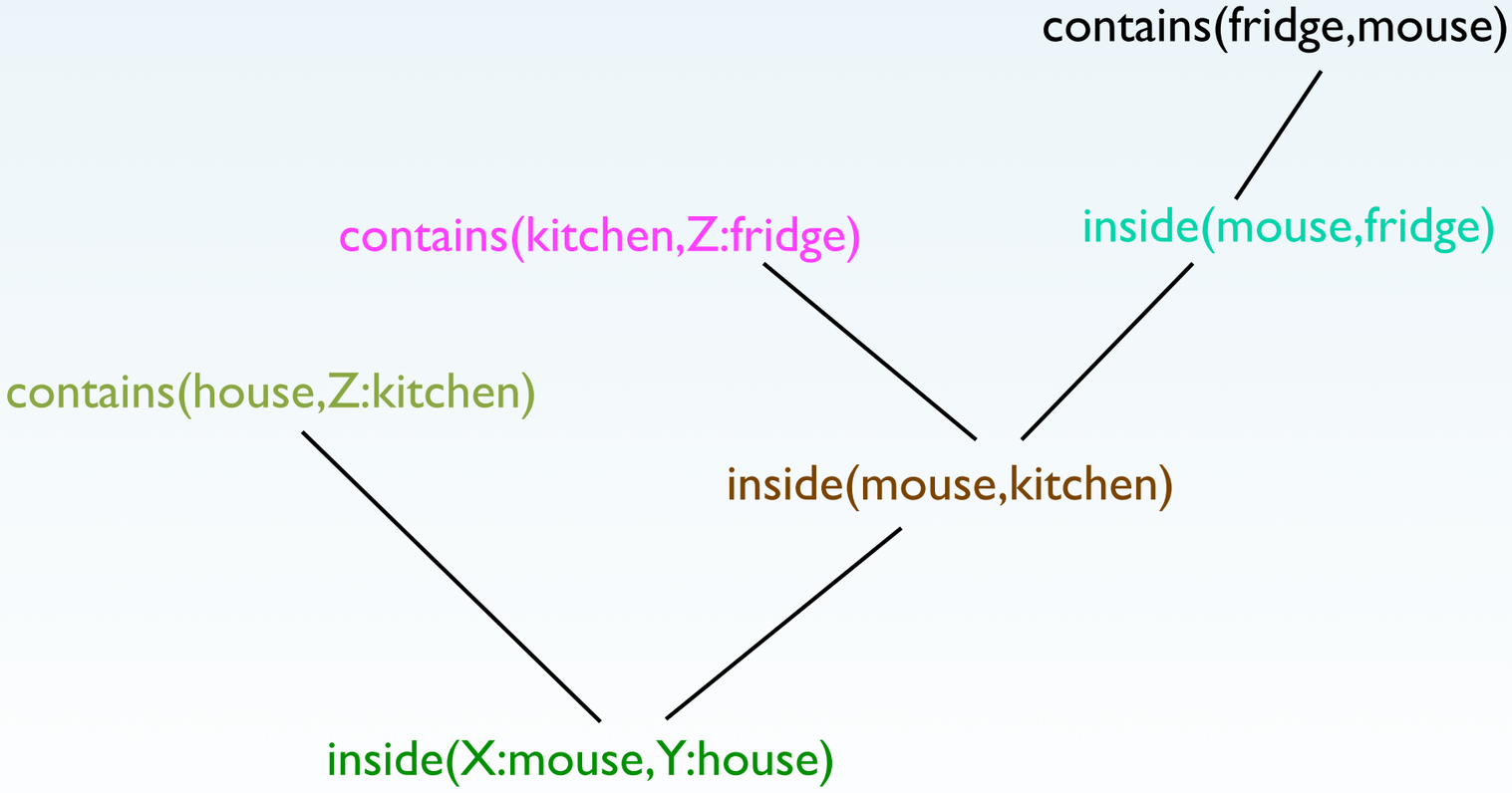
(7) Both subgoals are revoked. Since there is no other way of satisfying inside(mouse,bathroom), this goal is revoked, too. This fact forces the advance of the "→" pointer.




```
→ contains(house,bathroom).
  contains(house,kitchen).
  contains(kitchen,fridge).
  contains(fridge,mouse).
  inside(X,Y) :- contains(Y,X).
→ inside(X,Y) :- contains(Y,Z),
                 inside(X,Z).
```

Resatisfying Goals

(8) The previous steps are repeated, now with "kitchen" bound to Z. In the recursive call to "contains", Z will be bound to fridge; the first rule for inside(mouse,Z) succeeds.




```
contains(house,bathroom).
contains(house,kitchen).
contains(kitchen,fridge).
contains(fridge,mouse).
inside(X,Y) :- contains(Y,X).
inside(X,Y) :- contains(Y,Z),
               inside(X,Z).
```

Exercises

Given the predicates female/1, male/1, and parent/2, define the following predicates.

(1) father/2 and mother/2

(2) grandfather/2, and grandmother/2

(3) child/2, son/2, and daughter/2

```
male(tywin).
male(jaime).
male(tyrion).
male(joffrey).
male(tommen).

female(joanna).
female(cersei).
female(myrcella).
```

```
parent(tywin,jaime).
parent(tywin,cersei).
parent(tywin,tyrion).
parent(joanna,jaime).
parent(joanna,cersei).
parent(joanna,tyrion).
parent(jaime,joffrey).
parent(jaime,tommen).
parent(jaime,myrcella).
parent(cersei,joffrey).
parent(cersei,tommen).
parent(cersei,myrcella).
```

Exercises

Given the predicates female/1, male/1, and parent/2, define the following predicates.

(4) grandchild/2, grandson/2, and granddaughter/2

(5) ancestor/2

(6) sibling/2, brother/2, and sister/2

```
male(tywin).
male(jaime).
male(tyrion).
male(joffrey).
male(tommen).

female(joanna).
female(cersei).
female(myrcella).
```

```
parent(tywin,jaime).
parent(tywin,cersei).
parent(tywin,tyrion).
parent(joanna,jaime).
parent(joanna,cersei).
parent(joanna,tyrion).
parent(jaime,joffrey).
parent(jaime,tommen).
parent(jaime,myrcella).
parent(cersei,joffrey).
parent(cersei,tommen).
parent(cersei,myrcella).
```

Exercises

Given the predicates female/1, male/1, and parent/2, define the following predicates.

(7) aunt/2 and uncle/2

(8) mate/2

(9) incest/2

```
male(tywin).
male(jaime).
male(tyrion).
male(joffrey).
male(tommen).

female(joanna).
female(cersei).
female(myrcella).
```

```
parent(tywin,jaime).
parent(tywin,cersei).
parent(tywin,tyrion).
parent(joanna,jaime).
parent(joanna,cersei).
parent(joanna,tyrion).
parent(jaime,joffrey).
parent(jaime,tommen).
parent(jaime,myrcella).
parent(cersei,joffrey).
parent(cersei,tommen).
parent(cersei,myrcella).
```

9.5 Equality

Terms can be "made" equal through substitution

Terms are identical

Terms can be evaluated to the same number

Different forms of equality between terms:

(1) *Unification* = and \=

(2) *Equivalence* == and \==

(3) *Evaluation* =:= and =\=

Unification Equality

Unification equality

Two terms are equal when they can be instantiated so that they become identical.

```
?- 3=3.
true.
```

```
?- X=3.
X = 3.
```

```
?- X=Y.
X = Y.
```

```
?- likes(X,red)=likes(john,Y).
X = john,
Y = red.
```

Term

```
?- car(red)=car(X).
X = red.
```

Compare with evaluation equality

```
?- 3=4.
false.
```

```
?- 3+1=4.
false.
```

Unification

Mapping from variables to terms

A *unifier* for two terms T and T' is a *substitution* for variables, σ, such that σ(T) = σ(T').

```
?- 3=3.
true.
```

σ={}
σ(3) = 3 = σ(3)

```
?- X=3.
X = 3.
```

σ={X→3}
σ(X) = 3 = σ(3)

```
?- X=Y.
X = Y.
```

σ={X→Y}
σ(X) = Y = σ(Y)

```
?- likes(X,red)=likes(john,Y).
X = john,
Y = red.
```

σ={X→john,Y→red}
σ(likes(X,red)) = likes(john,red) = σ(likes(john,Y))

```
?- likes(X,red)=Y.
Y = likes(X,red).
```

σ={Y→likes(X,red)}
σ(likes(X,red)) = likes(X,red) = σ(Y)

Equivalence

Equivalence

Two terms are equivalent if they are identical.

```
?- 3==3.
true.
```

```
?- X==3.
false.
```

```
?- X==Y.
false.
```

```
?- X=3, X==3.
X = 3.
```

```
?- X=Y, X==Y.
X = Y.
```

```
?- X==Y, X=Y.
false.
```

Different from object/ reference equality

Evaluation Equality

Evaluation Equality

Two terms are evaluation equal if they evaluate to the same number.

```
?- 3+1=:=4.
true.
```

```
?- X=3, X*2=:=X+3.
X = 3.
```

```
?- 3+1==4.
false.
```

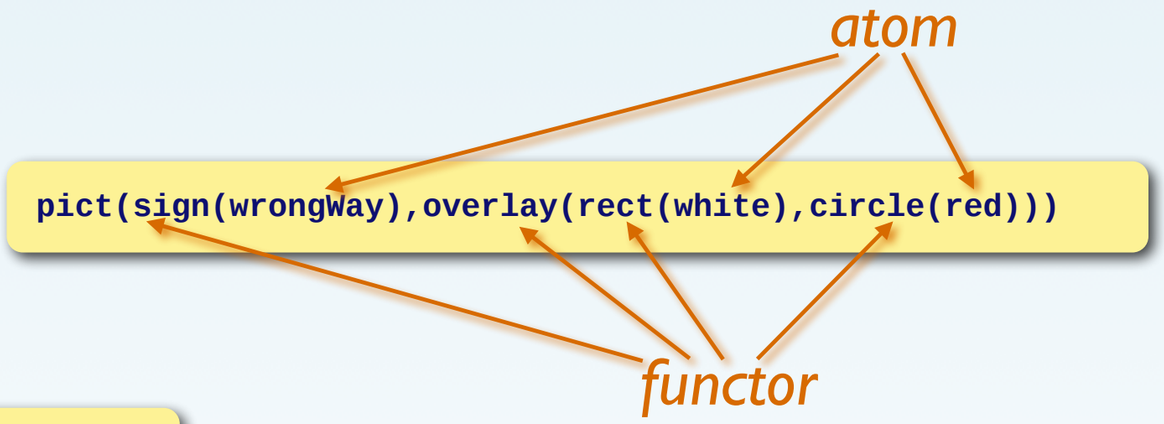
```
?- 3+1=4.
false.
```

```
?- 3=:=X.
ERROR: =:=/2: Arguments are not
sufficiently instantiated
```

```
?- X*2=:=X+3, X=3.
ERROR: =:=/2: Arguments are …
```

9.6 Terms & Lists

Terms denote structured objects




```
?- owns(john,X).
X = bike(blue).
```

```
?- owns(john,bike(X)).
X = blue.
```

```
?- owns(X,bike(Y)).
X = john,
Y = blue ;
X = mike,
Y = black.
```

```
owns(john,bike(blue)).
owns(mike,bike(black)).
owns(ann,caddilac(black)).
owns(ann,caddilac(silver)).
```

Variables cannot be used to match functors

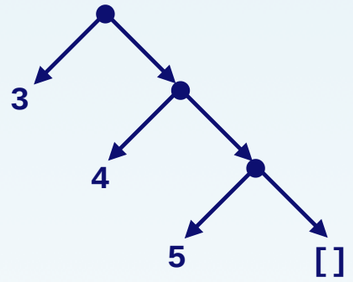
```
?- owns(john,X(blue)).
```

Lists

Lists are terms with special syntax

`[]`

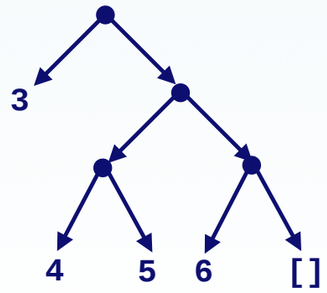
Empty list



`[3,4,5]` = `.(3,.(4,.(5,[])))`

Cons functor

`[3,[4,5],6]`



Lists can be heterogeneous

```
[3, little, pigs]
```

```
has(mary,[lamb(little),lamp(big),lame(pig)])
```

```
[walk(4),wait(2),bus(7),walk(2)]
```

List Patterns

Head *Tail*

```
?- story([X|Y]).
X = 3,
Y = [little, pigs].
```

```
story([3,little,pigs]).
```

```
?- story([X,Y|Z]).
X = 3,
Y = little,
Z = [pigs].
```

```
?- story([X,Y,Z]).
X = 3,
Y = little,
Z = pigs.
```

```
?- story([X,Y,Z|V]).
X = 3,
Y = little,
Z = pigs,
V = [].
```

```
?- story([X,Y,Z,V]).
false.
```

Pattern in Prolog		*Pattern in Haskell*
`[X,Y|Z]`	≈	`x:y:z`
`[X,Y,Z]`	≈	`[x,y,z]`

List Predicates

Wildcard, matches anything

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```

Test

```
?- member(3,[2,3,4,3]).
true ;
true.
```

```
?- member(3,[2,3,4,3,1]).
true ;
true ;
false.
```

```
?- member(2,[2,3,4,3]).
true ;
false.
```

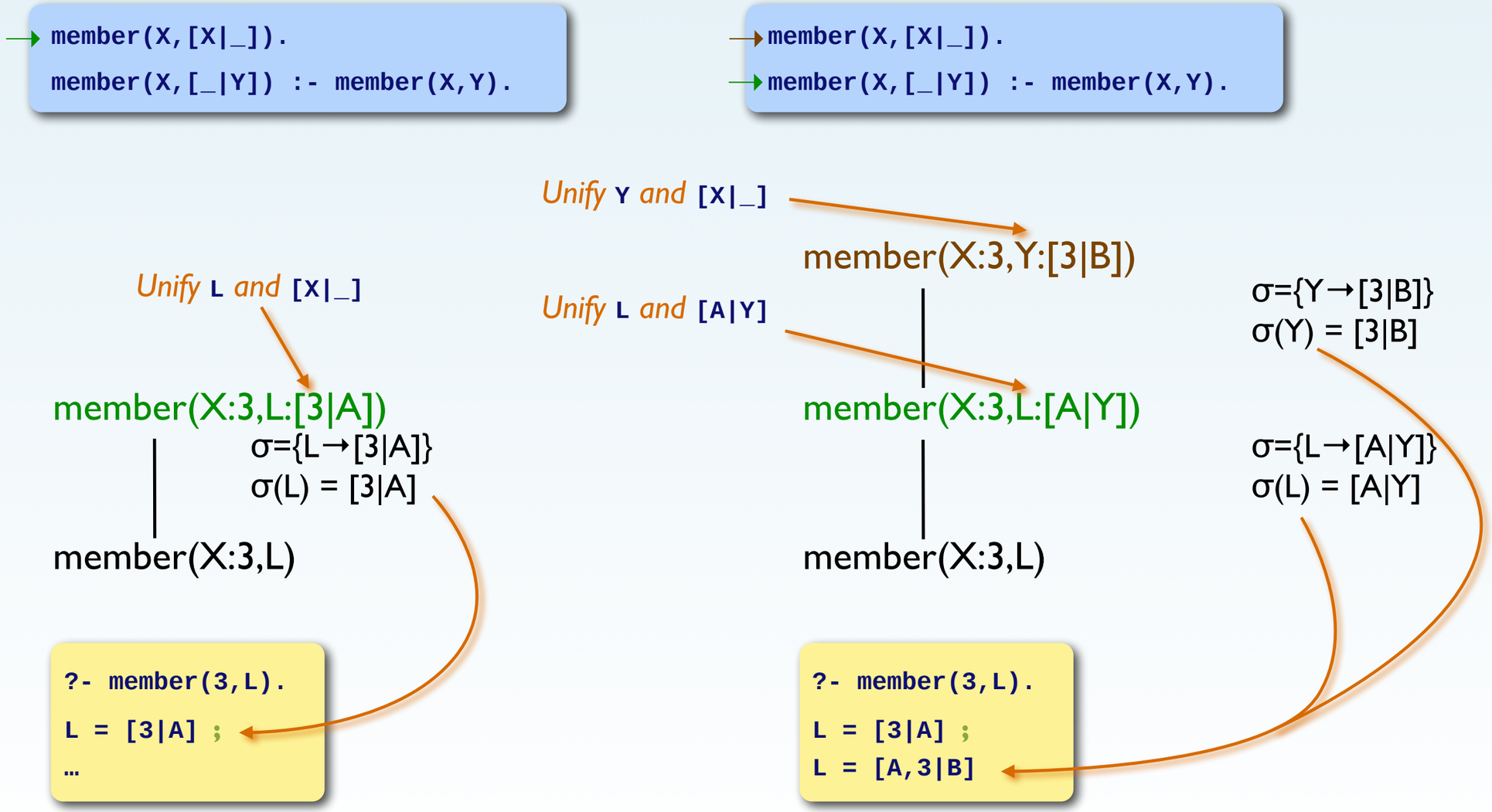
Traversal

```
?- member(X,[3,4]).
X = 3 ;
X = 4.
```

```
?- member(3,L).
L = [3|A] ;
L = [A,3|B] ;
L = [A,B,3|C]
…
```

Generation

Solving member(3,L).



The append/3 predicate

Constructing result in argument

```
append([], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

Only one way (of several) to read the definition!

```
?- append([2,3], [a,[c],d], L).
L = [2,3,a,[c],d].
```

```
?- append([2,3], Y, [2,3,a,[c],d]).
Y = [a,[c],d].
```

```
?- append(X, [a,[c],d], [2,3,a,[c],d]).
X = [2,3].
```

```
?- append(X, Y, [3,4,5]).
X = [],
Y = [3,4,5] ;
X = [3],
Y = [4,5] ;
X = [3,4],
Y = [5] ;
X = [3,4,5],
Y = [] ;
false.
```

Exercises

Define predicate del/3, such that in del(X, L, M) M is equal to the list L with the first occurrence of X removed.

What is the output of del(a, [a,b], L)?

Exercises

Avoid unwanted results in the definition of del(X, L, M).

```
del(X, [],    []).
del(X, [X|L], L).
del(X, [Y|L], [Y|M]) :- del(X, L, M).
```

What is the output of del(a, [a,b], L)?

Exercises

Define a predicate sublist/2, such that sublist(S, L) holds if the list S is a sublist of L.

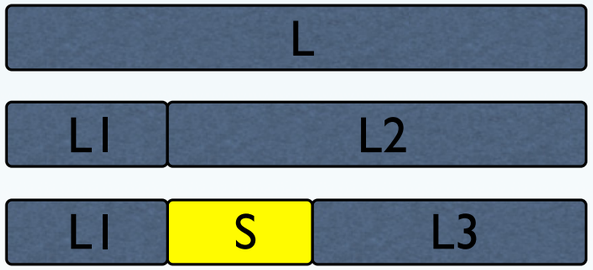
Examples:

sublist([c,d,e], [a,b,c,d,e,f]) yes

sublist([c,e], [a,b,c,d,e,f]) no

Hint: use append/3

Hint: S is a sublist of L if L can be decomposed into L1 and L2 and L2 can be decomposed into S and L3.



9.7 Arithmetic

Prolog Operators: =, \=, >, ... +, -, * , ...

X is <exp> binds result of <exp> to X

Unification vs. evaluation equality

```
?- X is 3*5.
X = 15.
```

```
?- X = 3*5.
X = 3*5.
```

`=:=` *evaluate and check*

`is` *evaluate and create binding*

```
?- 8 is X*2.
ERROR: is/2: Arguments are not
sufficiently instantiated
```

```
?- 8 = X*2.
false.
```

```
?- 4*2 = X*2.
X = 4.
```

```
?- fac(X,6).
ERROR: fac/2: Arguments are not
sufficiently instantiated
```

```
fac(1,1).
fac(N,M) :- K is N-1, fac(K,L), M is L*N.
```

Exercises

Define a predicate length/2, such that N in length(L, N) is equal to the length of the list L.

Can we swap the goals in the body of the second rule?

What does the goal length([a,b,c],N) compute if we use the following, alternative rule?

```
length([_|L], N) :- length(L, M), N = M+1.
```

Exercises

Simplify the following rule for length.

```
length([_|L], N) :- length(L, M), N = M+1.
```

Using the above version, which goal can compute the length of a list ?

9.8 The Cut & Negation

The cut "!" prevents backtracking:

- *do not resatisfy goals*
- *never advance any preceding markers*

The cut is a **non-logical**, **imperative** feature!

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y).
```

```
?- member(3,[3,4,3,3]).
true ;
true ;
true.
```

```
?- member(3,L).
L = [3|A] ;
L = [A,3|B] ;
L = [A,B,3|C]
…
```

`memberchk` *in SWIPL*

```
member(X,[X|_]) :- !.
member(X,[_|Y]) :- member(X,Y).
```

```
?- member(3,[3,4,3,3]).
true.
```

```
?- member(3,L).
L = [3|A].
```

```
member(X,[X|_]).
member(X,[_|Y]) :- member(X,Y), !.
```

```
?- member(3,[3,4,3,3]).
true ;
true.
```

```
?- member(3,L).
L = [3|A] ;
L = [A,3|B].
```

Negation

... can be tricky

Video Clip

Negation Predicate

```
likes(mary,X) :- animal(X), not(snake(X)).
```

```
not(P) :- P, !, fail.
not(P).
```

If `P` *is true, then* `not(P)` *is false*

Otherwise, `not(P)` *is true*

```
animal(dog).
animal(python).
snake(python).
```

```
?- likes(mary,dog).
true.
```

```
?- likes(mary,cobra).
false.
```

```
?- likes(mary,python).
false.
```

```
?- likes(mary,cat).
false.
```

Negation as Failure

```
not(P) :- P, !, fail.
not(P).
```

```
mother(mary).
mother(eve).
hadSex(eve).
```

```
virginBirth(X) :- mother(X), not(hadSex(X)).
```

```
?- virginBirth(eve).
false.
```

```
?- virginBirth(mary).
true.
```

```
?- virginBirth(X).
X = mary ;
false.
```

```
virginBirth(X) :- not(hadSex(X)), mother(X).
```

```
?- virginBirth(X).
false.
```

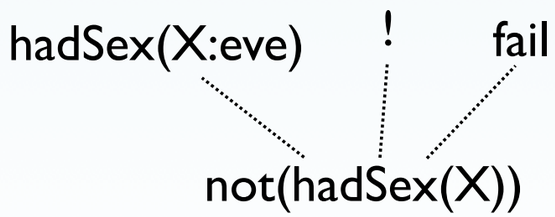
Negation as Failure

```
not(P) :- P, !, fail.
not(P).
```

```
mother(mary).
mother(eve).
hadSex(eve).
```

```
?- virginBirth(X).
false.
```

```
virginBirth(X) :- not(hadSex(X)), mother(X).
```



not(hadSex(X))

=

not(∃X. hadSex(X))

=

∀X. not(hadSex(X))

≠

∃X. not(hadSex(X))

Barber Paradox

In a town, the barber shaves all males that do not shave themselves. Does the barber shave himself?

```
shaves(barber,X) :- male(X), not(shaves(X,X)).

male(barber).
```

```
?- shaves(barber,barber).
ERROR: Out of local stack
```

More on Negation

Video Clip

9.9 Prolog Pitfalls

- Misspelled predicates
- Wrong arity of predicates
- Missing rule cases
- Too general rules (e.g. 'catch-all' rules)
- Unintentional unification (use one name for variables that should be different)
- Left-recursion in rules:

```
path(A,C) :- path(A,B), edge(B,C).
path(A,A).
```

- Infinite unification (as in: X = [X]) (use unify_with_occurs_check)