

CS 381-001 Programming Language Fundamentals

4 credits

OSU catalog course description including pre-requisites/co-requisites: An introduction to the concepts found in a variety of programming languages. Programming languages as tools for problem solving. A brief introduction to languages from a number of different paradigms. **Prerequisites:** CS 261 and (MTH 231 or CS225)

CS 381

Instructor: Julianne Coffman

Office Hours: M&W 12:30-2:00pm in KEC 3109

E-mail: coffmaju@oregonstate.edu

Email should be a secondary with the primary contact being Canvas messaging.

Meetings: M&W 10:00 – 11:50am in Linc 210 (1/9 to 3/17)

Textbooks: *Concepts of Programming Languages* by Robert W. Sebesta, 12th Edition.

CS 381 Class Notes by Dr. Martin Erwig

TAs : Information on Canvas

Learning Objectives: On completion of the course, students should be able to perform the following tasks.

1. **Create** functional programs using *algebraic data types* and *recursive functions*.
2. **Produce and explain** the *type* and *result* of an expression in the context of functional programming.
3. **Produce** an *abstract syntax* for a language given its concrete syntax.
4. **Create** a *denotational semantics* for a language given its abstract syntax and an informal specification of its behavior.
5. **Produce and explain** the behavior of a program under *static* vs. *dynamic typing*, and discuss the benefits and drawbacks of each approach.
6. **Produce and explain** a program's output under *static* vs. *dynamic scoping* of names.
7. **Produce and explain** a program's output under different parameter passing schemes, such as *call-by-value* vs. *call-by-name* vs. *call-by-need*.
8. **Create** logic programs and express queries using *predicates*.

Grade Evaluation: Your course grade will be based on the following:

Homework 50%

Quizzes 30%

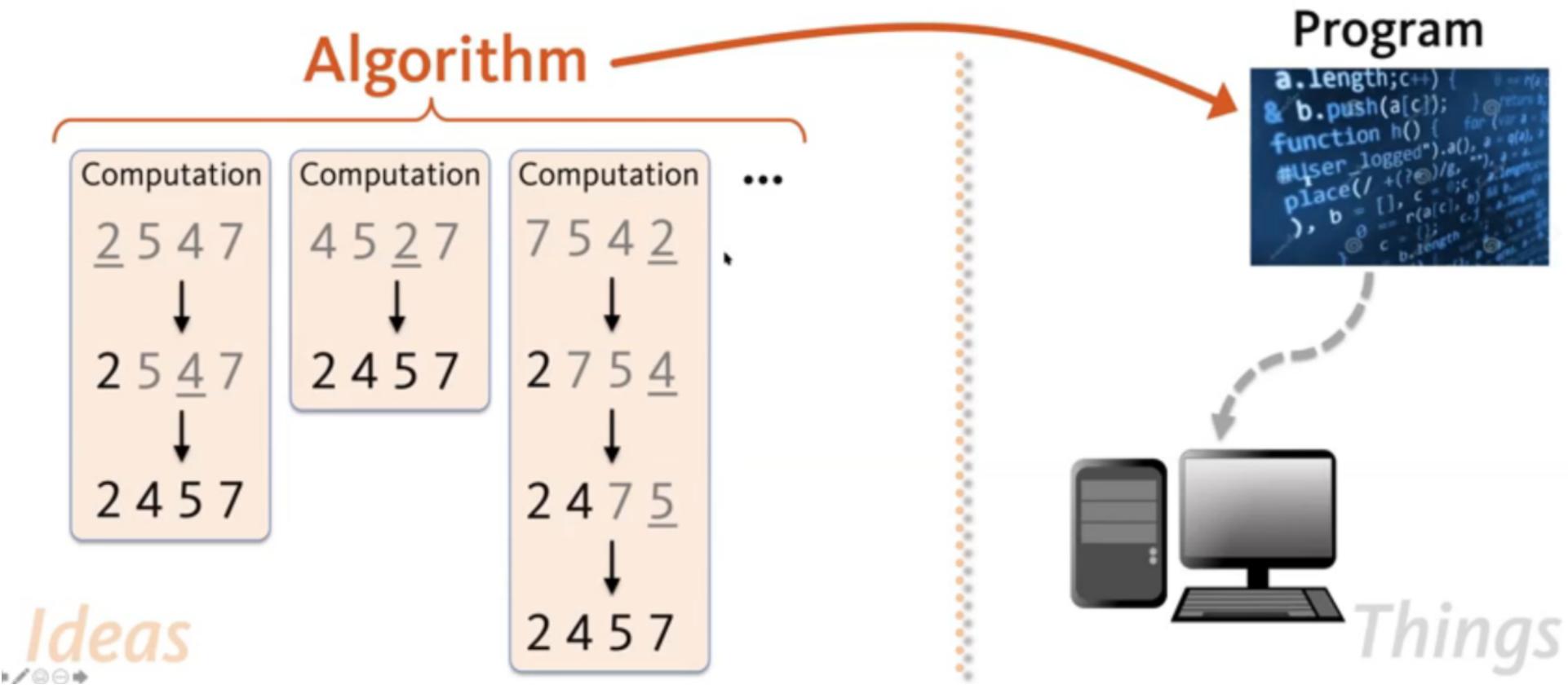
Final Exam 20%

TOTAL -----100%

Week	Topics	Readings
1	Intro, Haskell	LN: Unit 1- Introduction CPL: Ch 1- Preliminaries CPL: Ch 2 - Evolution of Major Languages & Paradigms
1&2	Haskell	LN: Unit 2 – Haskell Haskell Tutorials
3	Syntax	CPL: Ch 3 – Sections 1-3 Describing Syntax LN: Unit 3 – Syntax
4&5	Semantics	LN: Unit 4 – Semantics
6	Types	LN: Unit 5 – Types CPL: Ch 6 – Data Types
7	Scope	LN: Unit 6 – Scope CPL: Ch 5 – Names, Binding & Scope CPL: Ch 7 – Expressions and Statements
8	Parameter Passing	LN: Unit 7 – Parameter Passing CPL: Ch 9 – Subprograms
9&10	Prolog	LN: Unit 8 – Prolog CPL: Ch 16 – Logical Programming Languages

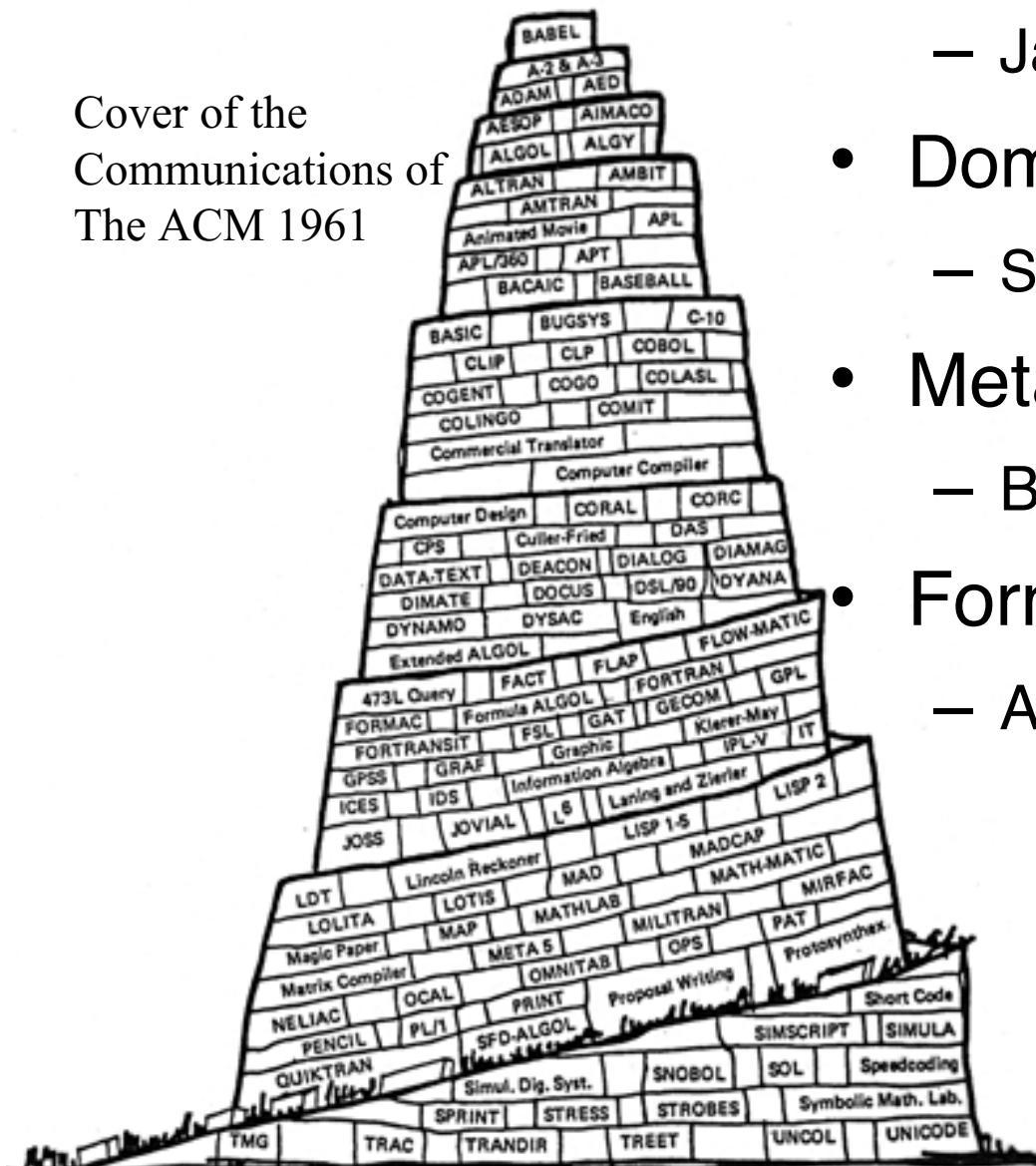
LN: Lecture Notes for Programming Language Fundamentals by Dr Martin Erwin
 CPL: Concepts of Programming Languages by Robert Sebesta

What is a Programming Language? What is Programming?



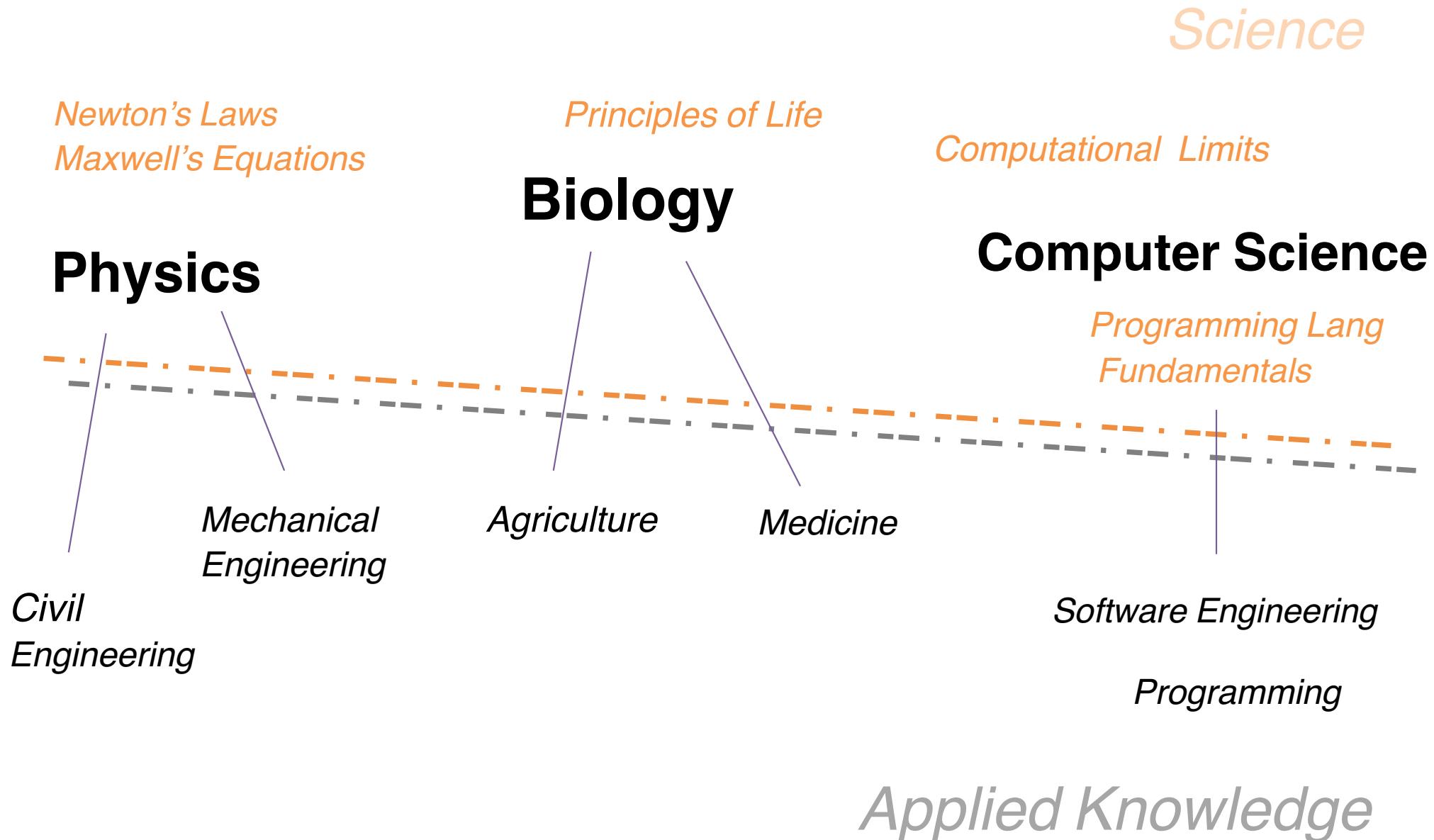
Languages in CS

Cover of the
Communications of
The ACM 1961

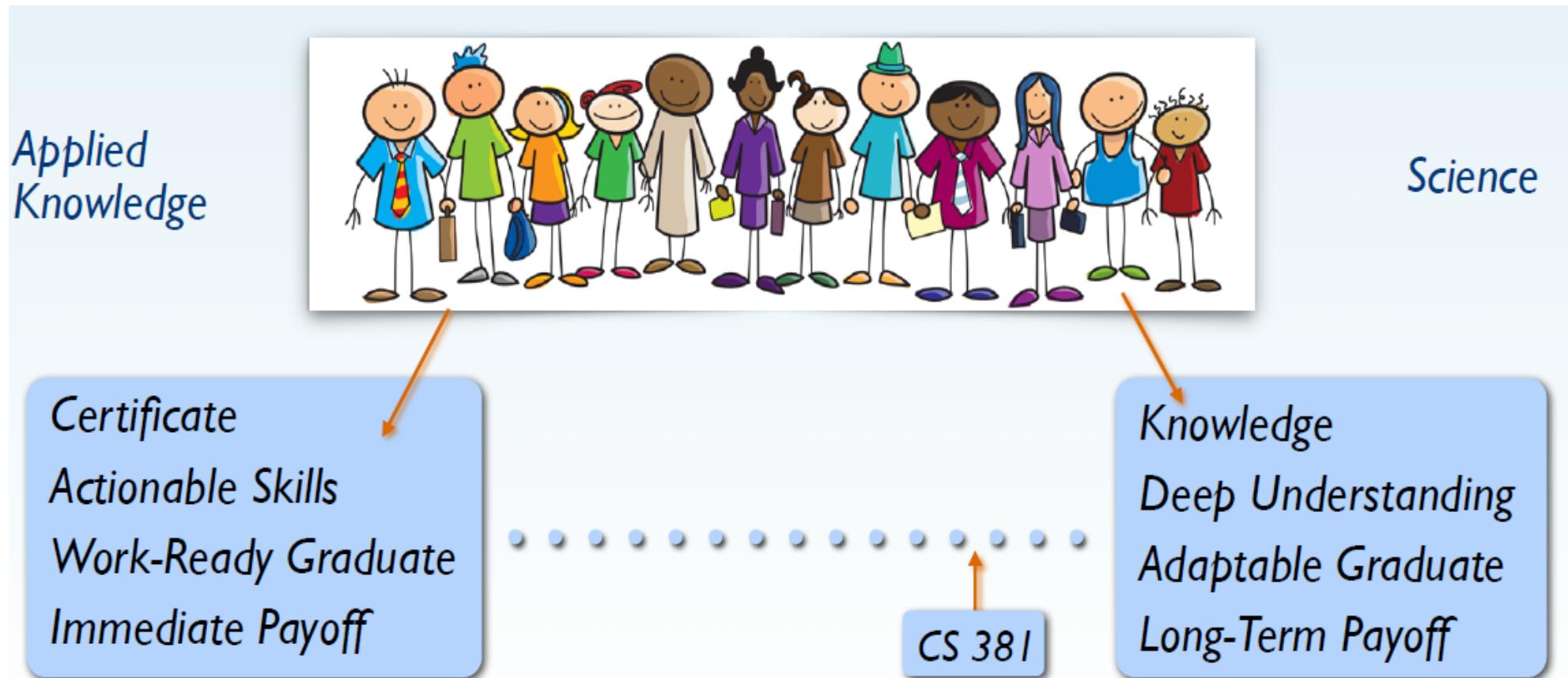


- Programming Languages
 - Java, C, Python, Haskell, Prolog, ...
- Domain-Specific Languages
 - SQL, HTML, R, LaTex, Excel, ...
- Metalanguages
 - BNF (grammars), Rule Systems, ...
- Formal Models
 - Automata, Logic(s), ...

Science vs. Engineering



Programming vs Computer Science



What is CS 381 About?

What exactly is a programming language?

Recursion?

Functional Programming

```
int x;  
int f(int z) {x:= f(y)...};  
{ int x;  
  f(3)  
}
```

Value or Address?

When to evaluate?

Parameter Passing

Which x?

Scoping

x := f(y) * g(z-1)

“Determine” value of RHS and “assign it” to x

How to express computation?

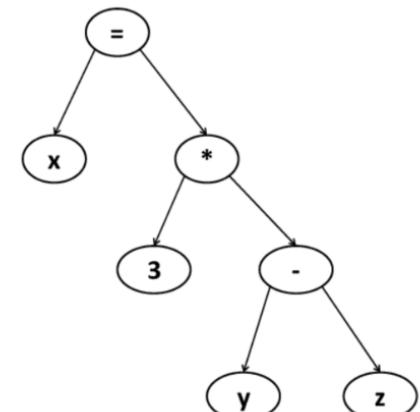
Programming Paradigms

Semantics

Based on what?

Abstract Syntax

AST



The Role of Haskell in CS 381

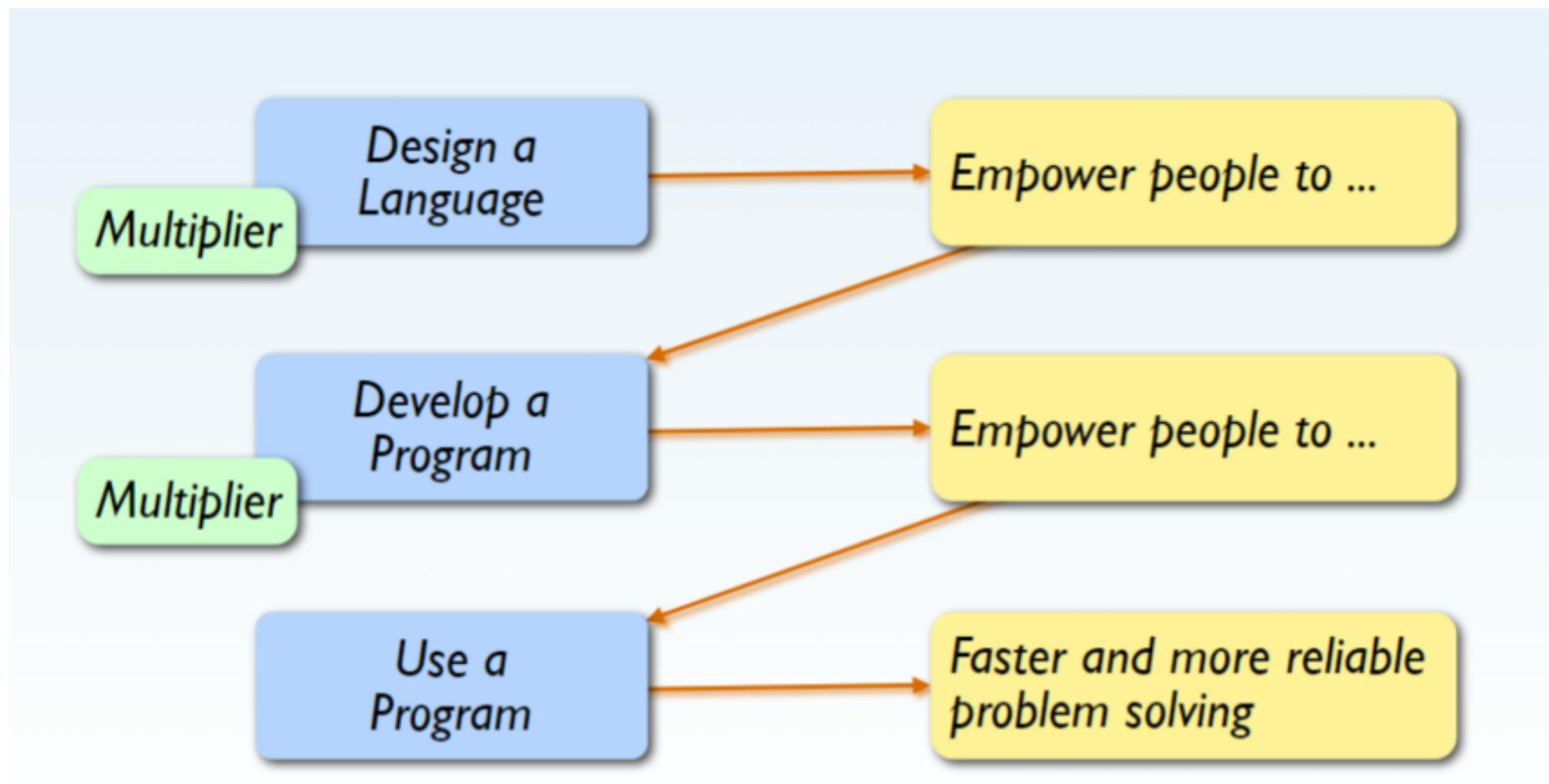
- Example of a non-imperative programming paradigm
- Tool for describing language concepts: syntax, semantics, scope, typing
- Executable PL Theory

Metalanguage			
	English	Math	Haskell
Precise	✗	✓	✓
Checkable	✗	✗	✓
Executable	✗	✗	✓

Learning Objectives

	Class Section							
	1 Haskell	2 Abstract Syntax	3 Semantics	4 Types	5 Scope	6 Parameter Passing	7 Paradigms	8 Prolog
1 Functional Programming	✓	✓	✓	✓			✓	
2 Functional Programming	✓	✓	✓	✓			✓	
3 Abstract Syntax		✓						
4 Denotational Semantics			✓					
5 Static & Dynamic Typing				✓				
6 Static & Dynamic Scoping					✓			
7 Parameter Passing						✓		
8 Logic Programming							✓	✓

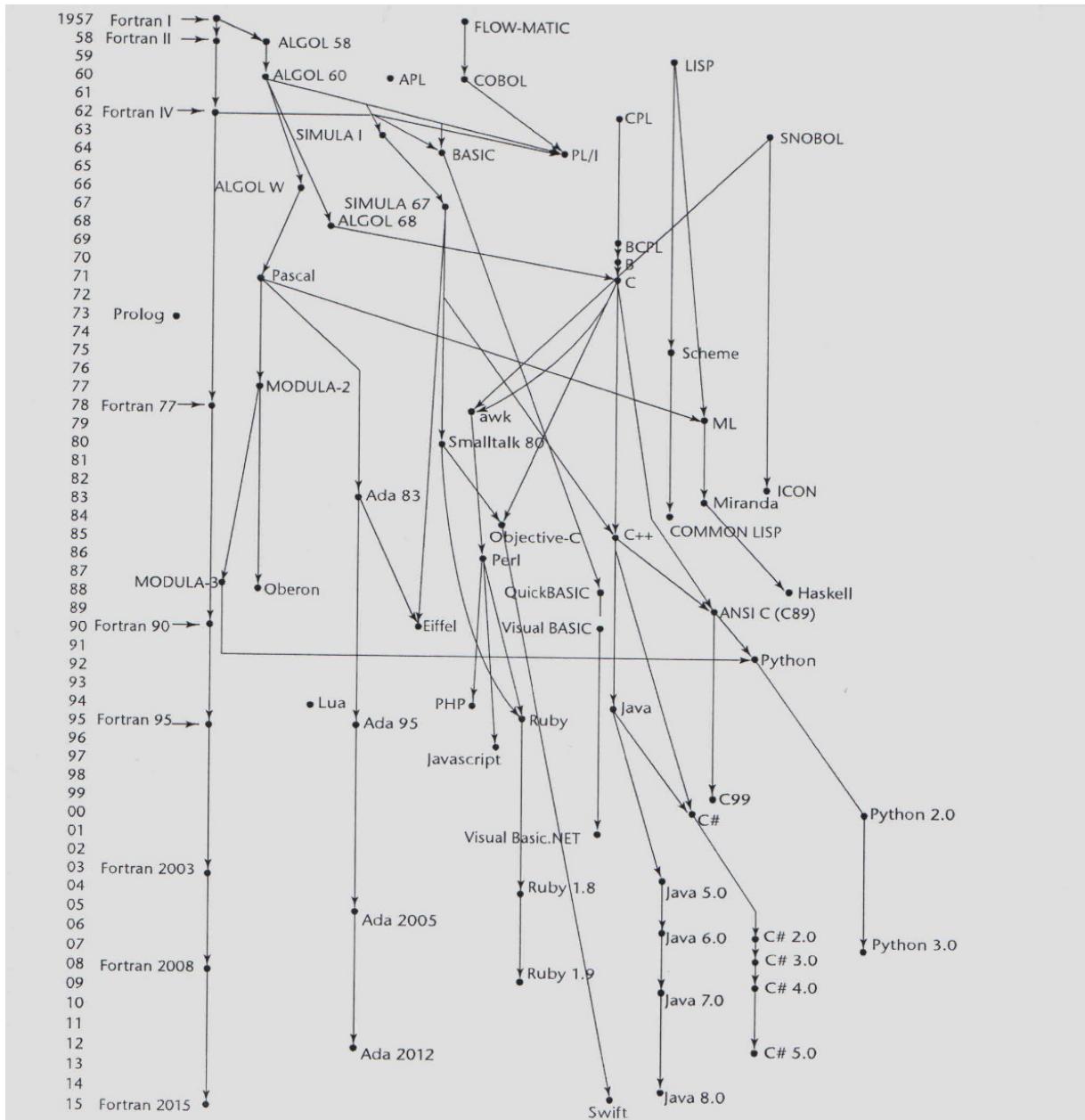
Impacts of Languages & Programs



Practical Benefits of Studying Programming Languages

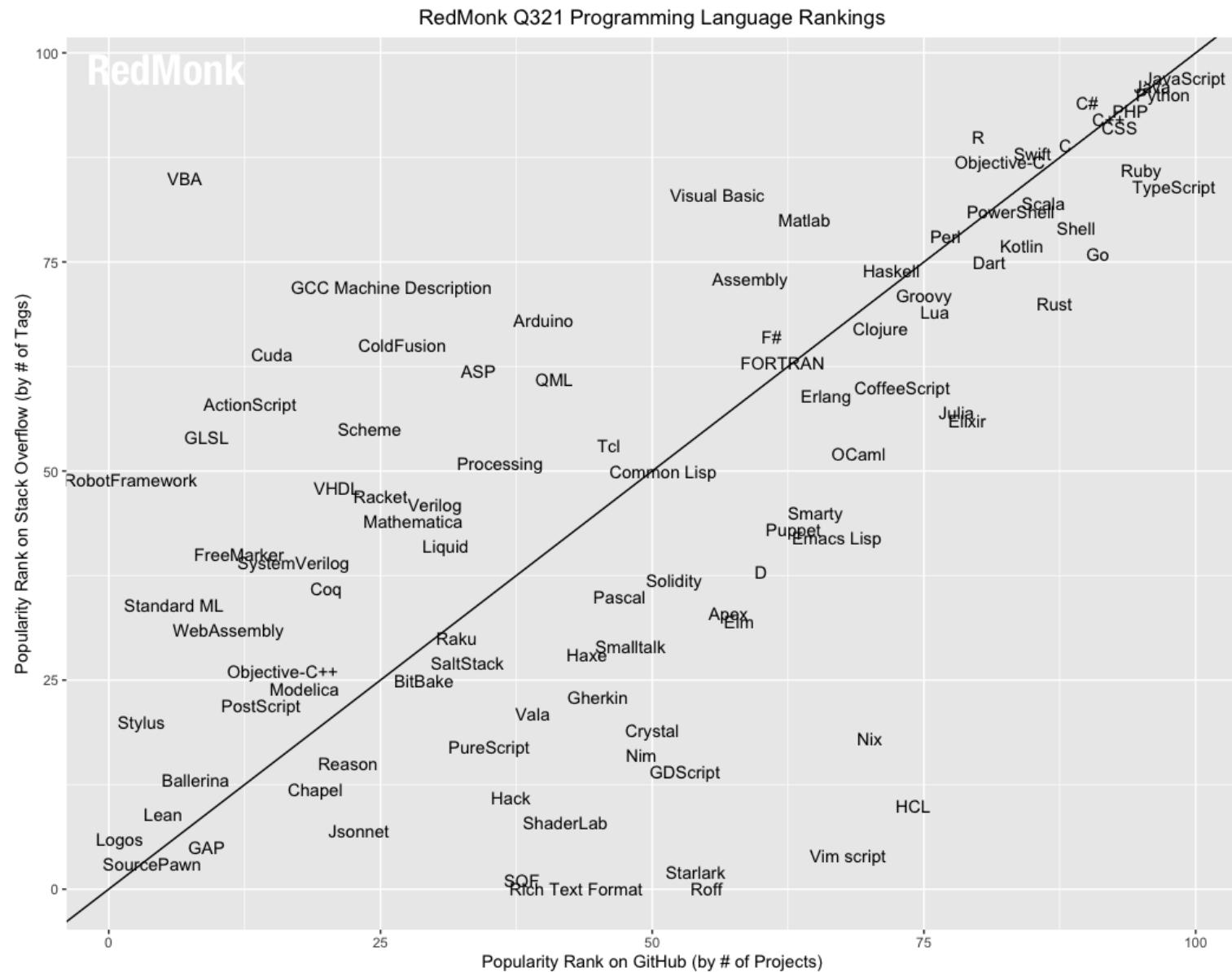
- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Become more productive

Genealogy of Common Languages



Popularity

RedMonk has a language ranking scheme that combines pull requests on GitHub and questions on StackOverflow. June 2021.



Programming Domains

- Scientific applications
 - Large numbers of floating point computations; use of arrays
 - Fortran
- Business applications
 - Produce reports, use decimal numbers and characters
 - COBOL
- Artificial intelligence
 - Symbols rather than numbers manipulated; use of linked lists
 - LISP
- Systems programming
 - Need efficiency because of continuous use
 - C
- Web Software
 - Eclectic collection of languages: markup (e.g., HTML), scripting (e.g., PHP), general-purpose (e.g., Java)

Language Evaluation Criteria

- **Readability:** the ease with which programs can be read and understood
- **Writability:** the ease with which a language can be used to create programs
- **Reliability:** conformance to specifications (i.e., performs to its specifications)

Evaluation Criteria: Readability

- Overall simplicity
 - A manageable set of features and constructs
 - Minimal feature multiplicity
 - Minimal operator overloading
- Orthogonality
 - A relatively small set of primitive constructs can be combined in a relatively small number of ways
 - Every possible combination is legal
- Data types
 - Adequate predefined data types
- Syntax considerations
 - Identifier forms: flexible composition
 - Special words and methods of forming compound statements
 - Form and meaning: self-descriptive constructs, meaningful keywords

Evaluation Criteria: Writability

- Simplicity and orthogonality
 - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
 - The ability to define and use complex structures or operations in ways that allow details to be ignored
- Expressivity
 - A set of relatively convenient ways of specifying operations
 - Strength and number of operators and predefined functions

Evaluation Criteria: Reliability

- Type checking
 - Testing for type errors
- Exception handling
 - Intercept run-time errors and take corrective measures
- Aliasing
 - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability
 - A language that does not support “natural” ways of expressing an algorithm will require the use of “unnatural” approaches, and hence reduced reliability

Evaluation Criteria: Cost

- Training programmers to use the language
- Writing programs (closeness to particular applications)
- Executing programs
- Reliability: poor reliability leads to high costs
- Maintaining programs

Evaluation Criteria: Others

- Portability
 - The ease with which programs can be moved from one implementation to another
- Generality
 - The applicability to a wide range of applications
- Well-definedness
 - The completeness and precision of the language's official definition

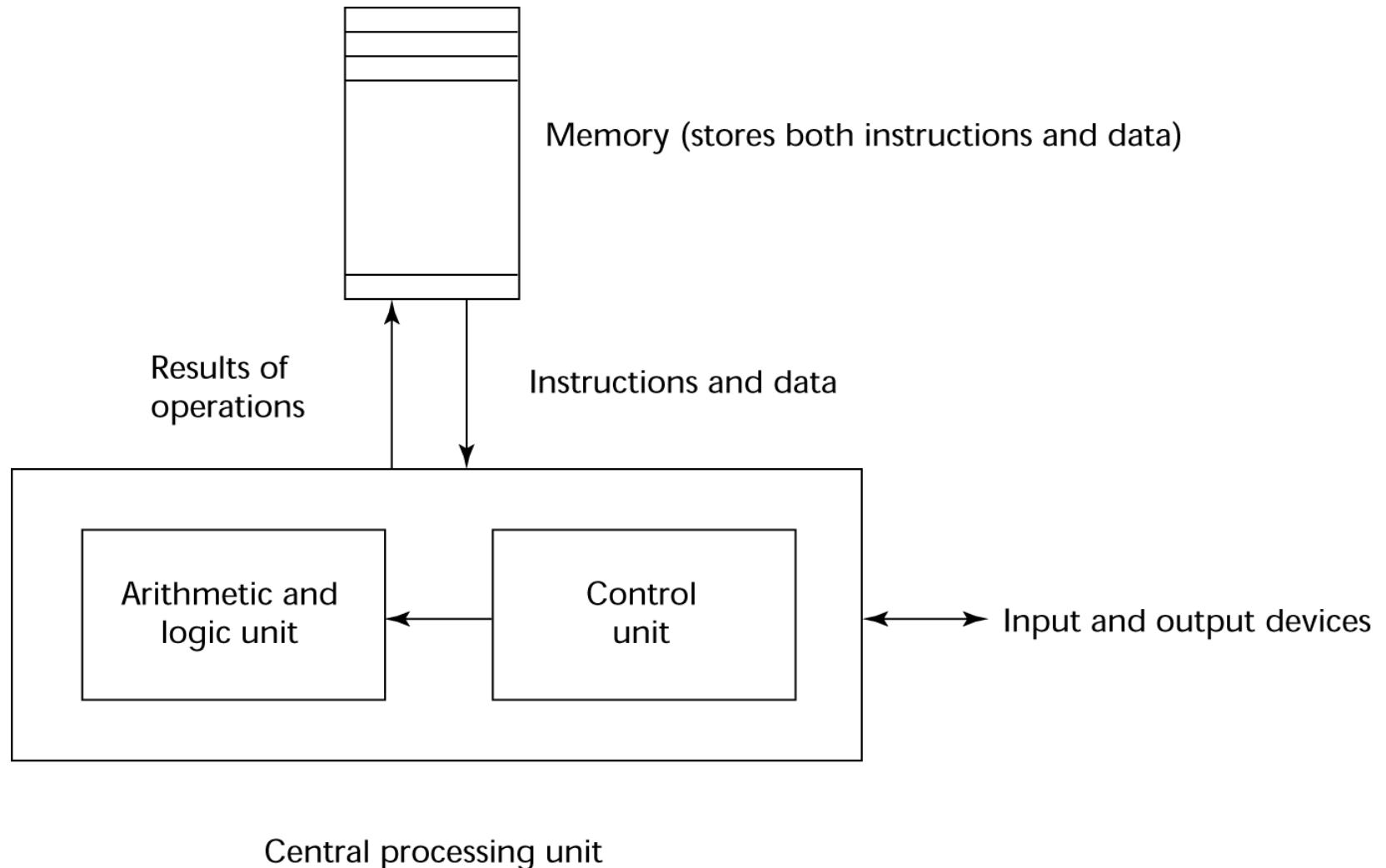
Influences on Language Design

- Computer Architecture
 - Languages are developed around the prevalent computer architecture, known as the *von Neumann* architecture
- Program Design Methodologies
 - New software development methodologies (e.g., object-oriented software development) led to new programming paradigms and by extension, new programming languages

Computer Architecture Influence

- Well-known computer architecture: Von Neumann
- Imperative languages, most dominant, because of von Neumann computers
 - Data and programs stored in memory
 - Memory is separate from CPU
 - Instructions and data are piped from memory to CPU
 - Basis for imperative languages
 - Variables model memory cells
 - Assignment statements model piping
 - Iteration is efficient

The von Neumann Architecture



The von Neumann Architecture

- Fetch-execute-cycle (on a von Neumann architecture computer)

initialize the program counter

repeat forever

 fetch the instruction pointed by the counter

 increment the counter

 decode the instruction

 execute the instruction

end repeat

Language Paradigms

- Imperative
 - Central features are variables, assignment statements, and iteration
 - Include languages that support object-oriented programming
 - Include scripting languages
 - Include the visual languages
 - Examples: C, Java, Perl, JavaScript, Visual BASIC .NET, C++
- Functional
 - Main means of making computations is by applying functions to given parameters
 - Examples: Haskell, LISP, Scheme, ML, F#
- Logic
 - Rule-based (rules are specified in no particular order)
 - Example: Prolog
- *Markup/programming hybrid*
 - Markup languages extended to support some programming
 - Examples: JSTL, XSLT

Language Design Trade-Offs

- Reliability vs. cost of execution
 - Example: Java demands all references to array elements be checked for proper indexing, which leads to increased execution costs
- Readability vs. writability

Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability
- Writability (flexibility) vs. reliability
 - Example: C++ pointers are powerful and very flexible but are unreliable

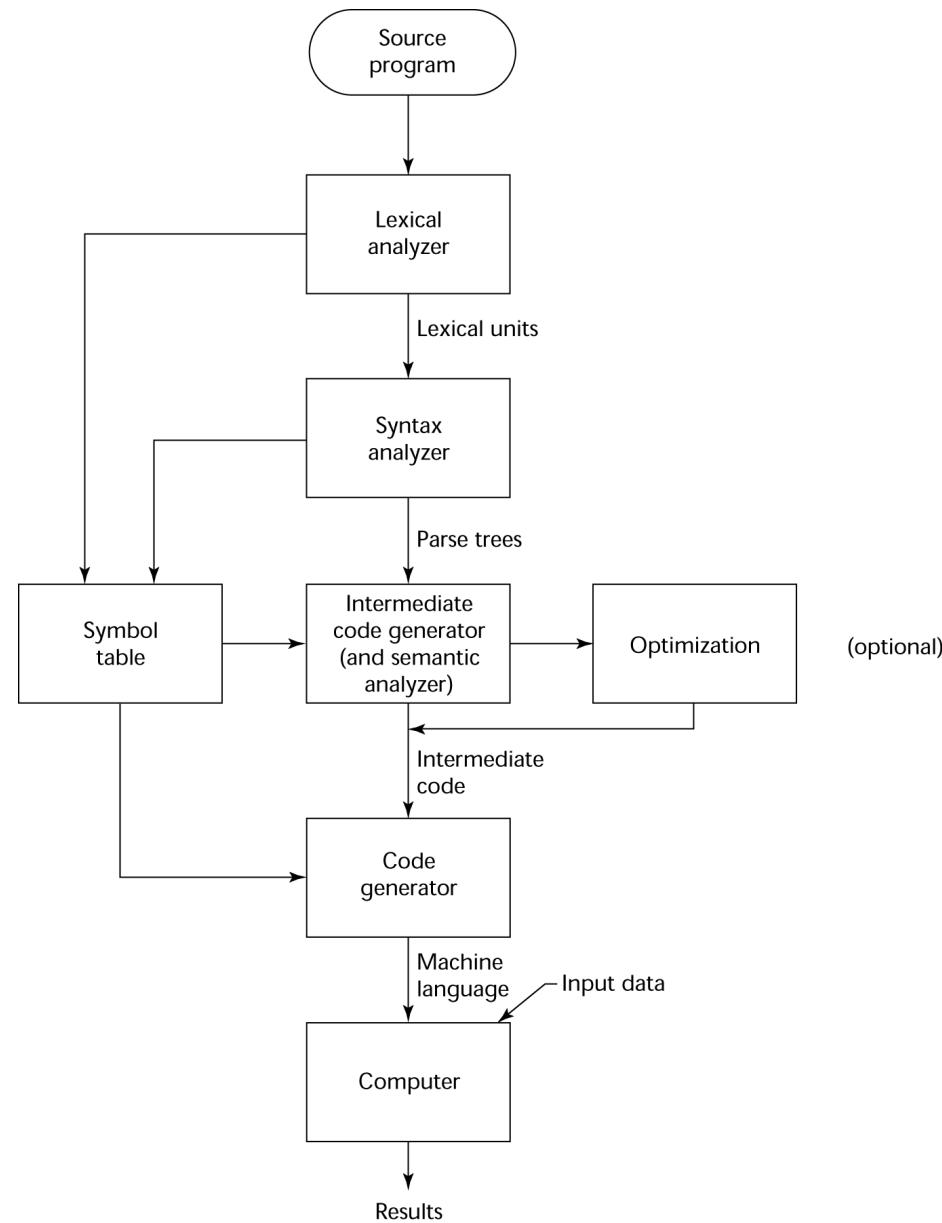
Implementation Methods

- Compilation
 - Programs are translated into machine language; includes JIT systems
 - Use: Large commercial applications
- Pure Interpretation
 - Programs are interpreted by another program known as an interpreter
 - Use: Small programs or when efficiency is not an issue
- Hybrid Implementation Systems
 - A compromise between compilers and pure interpreters
 - Use: Small and medium systems when efficiency is not the first concern

Compilation

- Translate high-level program (source language) into machine code (machine language)
- Slow translation, fast execution
- Compilation process has several phases:
 - lexical analysis: converts characters in the source program into lexical units
 - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
 - Semantics analysis: generate intermediate code
 - code generation: machine code is generated

The Compilation Process (CS 480)



Additional Compilation Terminologies

- **Load module** (executable image): the user and system code together
- **Linking and loading**: the process of collecting system program units and linking them to a user program

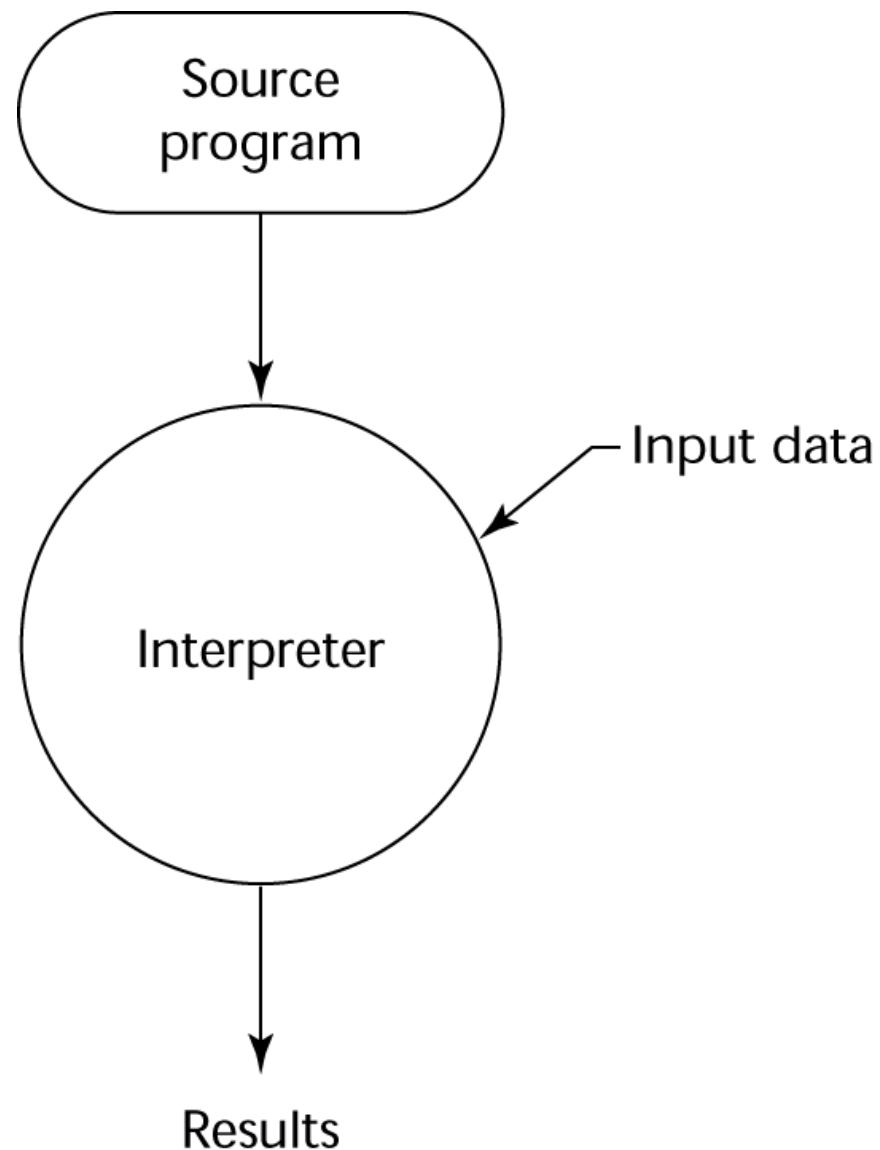
Von Neumann Bottleneck

- Connection speed between a computer's memory and its processor determines the speed of a computer
- Program instructions often can be executed much faster than the speed of the connection; the connection speed thus results in a *bottleneck*
- Known as the *von Neumann bottleneck*; it is the primary limiting factor in the speed of computers

Pure Interpretation

- No translation
- Easier implementation of programs (run-time errors can easily and immediately be displayed)
- Slower execution (10 to 100 times slower than compiled programs)
- Often requires more space
- Now rare for traditional high-level languages
- Significant comeback with some Web scripting languages (e.g., JavaScript, PHP)

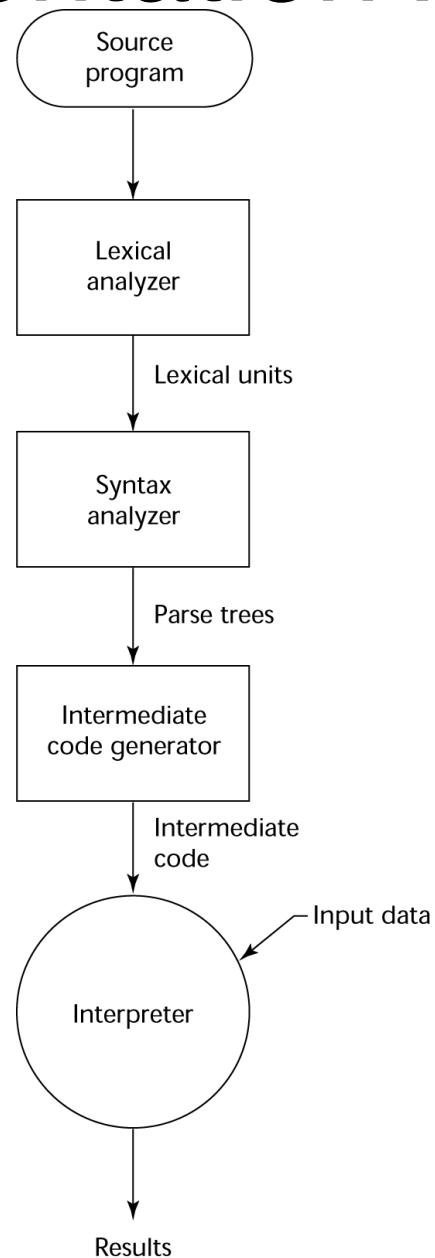
Pure Interpretation Process



Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
- Examples
 - Perl programs are partially compiled to detect errors before interpretation
 - Initial implementations of Java were hybrid; the intermediate form, *byte code*, provides portability to any machine that has a byte code interpreter and a run-time system (together, these are called *Java Virtual Machine*)

Hybrid Implementation Process



Languages

IBM 704 and Fortran

- Fortran 0: 1954 - not implemented
- Fortran I: 1957
 - Designed for the new IBM 704, which had index registers and floating point hardware
 - This led to the idea of compiled programming languages, because there was no place to hide the cost of interpretation (no floating-point software)
 - Environment of development
 - Computers were small and unreliable
 - Applications were scientific
 - No programming methodology or tools
 - Machine efficiency was the most important concern

Fortran I Overview

- First implemented version of Fortran
 - Names could have up to six characters
 - Post-test counting loop (**DO**)
 - Formatted I/O
 - User-defined subprograms
 - Three-way selection statement (arithmetic **IF**)
 - No data typing statements

Fortran 77

- Became the new standard in 1978
 - Character string handling
 - Logical loop control statement
 - **IF-THEN-ELSE** statement

Fortran 90

- Most significant changes from Fortran 77
 - Modules
 - Dynamic arrays
 - Pointers
 - Recursion
 - **CASE** statement
 - Parameter type checking

Latest versions of Fortran

- Fortran 95 – relatively minor additions, plus some deletions
- Fortran 2003 – support for OOP, procedure pointers, interoperability with C
- Fortran 2008 – blocks for local scopes, co-arrays, Do Concurrent

Fortran Evaluation

- Highly optimizing compilers (all versions before 90)
 - Types and storage of all variables are fixed before run time
- Dramatically changed forever the way computers are used

Functional Programming: Lisp

- LISt Processing language
 - Designed at MIT by McCarthy
- AI research needed a language to
 - Process data in lists (rather than arrays)
 - Symbolic computation (rather than numeric)
- Only two data types: atoms and lists
- Syntax is based on *lambda calculus*

Lisp Evaluation

- Pioneered functional programming
 - No need for variables or assignment
 - Control via recursion and conditional expressions
- Still the dominant language for AI
- Common Lisp and Scheme are contemporary dialects of Lisp
- ML, Haskell, and F# are also functional programming languages, but use very different syntax

Scheme

- Developed at MIT in mid 1970s
- Small
- Extensive use of static scoping
- Functions as first-class entities
- Simple syntax (and small size) make it ideal for educational applications

ALGOL 60 Overview

- Modified ALGOL 58 at 6-day meeting in Paris
- New features
 - Block structure (local scope)
 - Two parameter passing methods
 - Subprogram recursion
 - Stack-dynamic arrays
 - Still no I/O and no string handling

Computerizing Business Records: COBOL

- Environment of development
 - UNIVAC was beginning to use FLOW-MATIC
 - USAF was beginning to use AIMACO
 - IBM was developing COMTRAN

COBOL Historical Background

- Based on FLOW-MATIC
- FLOW-MATIC features
 - Names up to 12 characters, with embedded hyphens
 - English names for arithmetic operators (no arithmetic expressions)
 - Data and code were completely separate
 - The first word in every statement was a verb

COBOL Design Process

- First Design Meeting (Pentagon) - May 1959
- Design goals
 - Must look like simple English
 - Must be easy to use, even if that means it will be less powerful
 - Must broaden the base of computer users
 - Must not be biased by current compiler problems
- Design committee members were all from computer manufacturers and DoD branches
- Design Problems: arithmetic expressions? subscripts? Fights among manufacturers

COBOL Evaluation

- Contributions
 - First macro facility in a high-level language
 - Hierarchical data structures (records)
 - Nested selection statements
 - Long names (up to 30 characters), with hyphens
 - Separate data division

COBOL: DoD Influence

- First language required by DoD
 - would have failed without DoD
- Still the most widely used business applications language

SNOBOL

- Designed as a string manipulation language at Bell Labs by Farber, Griswold, and Polensky in 1964
- Powerful operators for string pattern matching
- Slower than alternative languages (and thus no longer used for writing editors)
- Still used for certain text processing tasks

The Beginning of Data Abstraction: SIMULA 67

- Designed primarily for system simulation in Norway by Nygaard and Dahl
- Based on ALGOL 60 and SIMULA I
- Primary Contributions
 - Coroutines - a kind of subprogram
 - Classes, objects, and inheritance

Orthogonal Design: ALGOL 68

- From the continued development of ALGOL 60 but not a superset of that language
- Source of several new ideas (even though the language itself never achieved widespread use)
- Design is based on the concept of orthogonality
 - A few basic concepts, plus a few combining mechanisms

ALGOL 68 Evaluation

- Contributions
 - User-defined data structures
 - Reference types
 - Dynamic arrays (called flex arrays)
- Comments
 - Less usage than ALGOL 60
 - Had strong influence on subsequent languages, especially Pascal, C, and Ada

Pascal - 1971

- Developed by Wirth (a former member of the ALGOL 68 committee)
- Designed for teaching structured programming
- Small, simple, nothing really new
- Largest impact was on teaching programming
 - From mid-1970s until the late 1990s, it was the most widely used language for teaching programming

C - 1972

- Designed for systems programming (at Bell Labs by Dennis Richie)
- Evolved primarily from BCLP and B, but also ALGOL 68
- Powerful set of operators, but poor type checking
- Initially spread through UNIX
- Though designed as a systems language, it has been used in many application areas

Programming Based on Logic: Prolog

- Developed, by Comerauer and Roussel (University of Aix-Marseille), with help from Kowalski (University of Edinburgh)
- Based on formal logic
- Non-procedural
- Can be summarized as being an intelligent database system that uses an inferencing process to infer the truth of given queries
- Comparatively inefficient
- Few application areas

History's Largest Design Effort: Ada

- Huge design effort, involving hundreds of people, much money, and about eight years
- Sequence of requirements (1975-1978)
 - (Strawman, Woodman, Tinman, Ironman, Steelman)
- Named Ada after Augusta Ada Byron, the first programmer

Ada Evaluation

- Contributions
 - Packages - support for data abstraction
 - Exception handling - elaborate
 - Generic program units
 - Concurrency - through the tasking model
- Comments
 - Competitive design
 - Included all that was then known about software engineering and language design
 - First compilers were very difficult; the first really usable compiler came nearly five years after the language design was completed

Ada 95

- Ada 95 (began in 1988)
 - Support for OOP through type derivation
 - Better control mechanisms for shared data
 - New concurrency features
 - More flexible libraries
- Ada 2005
 - Interfaces and synchronizing interfaces
- Popularity suffered because the DoD no longer requires its use but also because of popularity of C++

Object-Oriented Programming: Smalltalk

- Developed at Xerox PARC, initially by Alan Kay, later by Adele Goldberg
- First full implementation of an object-oriented language (data abstraction, inheritance, and dynamic binding)
- Pioneered the graphical user interface design
- Promoted OOP

Combining Imperative and Object-Oriented Programming: C++

- Developed at Bell Labs by Stroustrup in 1980
- Evolved from C and SIMULA 67
- Facilities for object-oriented programming, taken partially from SIMULA 67
- A large and complex language, in part because it supports both procedural and OO programming
- Rapidly grew in popularity, along with OOP
- ANSI standard approved in November 1997
- Microsoft's version: MC++
 - Properties, delegates, interfaces, no multiple inheritance

A Related OOP Language

- Swift – a replacement for Objective-C
 - Released in 2014
 - Two categories of types, classes and struct, like C#
 - Used by Apple for systems programs
- Delphi – another related language
 - A hybrid language, like C++
 - Began as an object-oriented version of Pascal
 - Designed by Anders Hejlsberg, who also designed Turbo Pascal and C#

An Imperative-Based Object-Oriented Language: Java

- Developed at Sun in the early 1990s
 - C and C++ were not satisfactory for embedded electronic devices
- Based on C++
 - Significantly simplified (does not include **struct**, **union**, **enum**, pointer arithmetic, and half of the assignment coercions of C++)
 - Supports *only* OOP
 - Has references, but not pointers
 - Includes support for applets and a form of concurrency

Java Evaluation

- Eliminated many unsafe features of C++
- Supports concurrency
- Libraries for applets, GUIs, database access
- Portable: Java Virtual Machine concept, JIT compilers
- Widely used for Web programming
- Use increased faster than any previous language

Scripting Languages for the Web

- Perl
 - Designed by Larry Wall—first released in 1987
 - Variables are statically typed but implicitly declared
 - Three distinctive namespaces, denoted by the first character of a variable's name
 - Powerful, but somewhat dangerous
 - Gained widespread use for CGI programming on the Web
 - Also used for a replacement for UNIX system administration language
- JavaScript
 - Began at Netscape, but later became a joint venture of Netscape and Sun Microsystems
 - A client-side HTML-embedded scripting language, often used to create dynamic HTML documents
 - Purely interpreted
 - Related to Java only through similar syntax
- PHP
 - PHP: Hypertext Preprocessor, designed by Rasmus Lerdorf
 - A server-side HTML-embedded scripting language, often used for form processing and database access through the Web
 - Purely interpreted

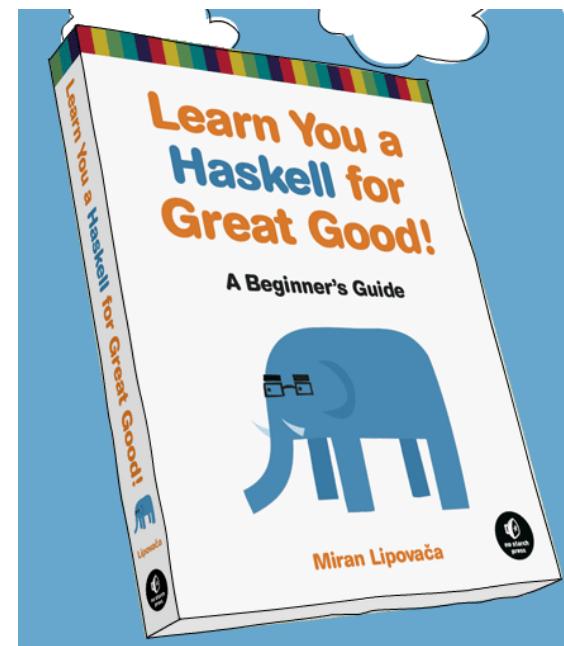
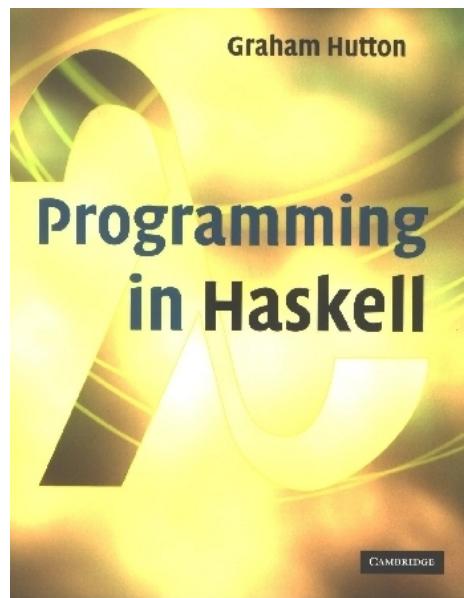
The Flagship .NET Language: C#

- Part of the .NET development platform (2000)
- Based on C++ , Java, and Delphi
- Includes pointers, delegates, properties, enumeration types, a limited kind of dynamic typing, and anonymous types
- Is evolving rapidly

Lesson 2

PROGRAMMING IN HASKELL

An Introduction



Based on lecture notes by Graham Hutton

What is a Functional Language?

Opinions differ, and it is difficult to give a precise definition, but generally speaking:

- Functional programming is style of programming in which the basic method of computation is the application of functions to arguments;
- A functional language is one that supports and encourages the functional style.

Example

Summing the integers 1 to 10 in an imperative language:

```
total = 0;  
for (i = 1; i < 10; ++i)  
    total = total+i;
```

The computation method is variable assignment.

Example

Summing the integers 1 to 10 in Haskell:

$$\text{sum}[1..10] = 1 + \text{sum}[10]$$

sum [1..10]

The computation method is function application.

$$\text{mySum}[a] = a$$

$$\text{sum}[\emptyset, s] = s + \text{sum}[s]$$

$$\text{mySum}(x : xs) = x + \text{mySum}(xs)$$

Historical Background

1930s:



Alonzo Church develops the lambda calculus,
a simple but powerful theory of functions.

Historical Background

1987:



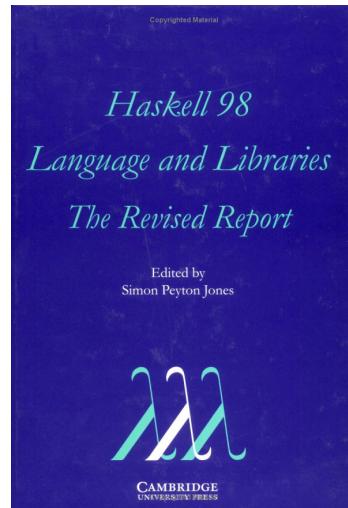
An international committee of researchers initiates the development of Haskell, a standard lazy functional language.

Partially in response to “Can programming be liberated from the Von Neuman style?”, by John Backus.

(Named in honor of logician Haskell B. Curry.)

Historical Background

2003:



The committee publishes the Haskell 98 report, defining a stable version of the language.

A Taste of Haskell

```
f []    = []
```

```
f (x:xs) = f ys ++ [x] ++ f zs
```

where

```
ys = [a | a ≤ x, a ≤ x]
```

```
zs = [b | b ≥ x, b > x]
```

quicksore

?

Basic Structure

- Purely function
- Lazy evaluation
- Statically typed with strong typing
- Uses type inference (like Python)
- VERY concise – small and elegant code
- Types are KEY (like Java or C – but more)

Features we care about:

- On turing
- Website with interface (somewhat limited functionality)
- Free and easy to download locally

Glasgow Haskell Compiler

- GHC is the leading implementation of Haskell, and comprises a compiler and interpreter;
- The interactive nature of the interpreter makes it well suited for teaching and prototyping;
- GHC is available at www.haskell.org/platform

Starting GHC on Flip server

The GHC interpreter can be started from the Unix command prompt by simply typing ghci:

```
flip2 ~ 151% ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> █
```

The GHCi prompt > means that the interpreter is ready to evaluate an expression.

For example:

```
Loading package base ... finished, version 0.0.0.0
Prelude> 5+2
7
Prelude> :set prompt Juli>
Juli>:type 5
5 :: Num a => a
Juli>reverse [1,4,6]
[6,4,1]
Juli>sqrt(10 + 2)
3.4641016151377544
Juli>[1..8]
[1,2,3,4,5,6,7,8]
Juli>
```

The Standard Prelude: List Madness!

Haskell comes with a large number of standard library functions. In addition to the familiar numeric functions such as `+` and `*`, the library also provides many useful functions on lists.

Select the first element of a list:

```
> head [1,2,3,4,5]  
1
```

→ return first element

Calculate the length of a list:

```
> length [1,2,3,4,5]  
5
```

Remove the first element from a list:

→ return a list.

```
> tail [1,2,3,4,5]  
[2,3,4,5]
```

Select the nth element of a list:

```
> [1,2,3,4,5] !! 2  
3
```

Select the first n elements of a list:

```
> take 3 [1,2,3,4,5]  
[1,2,3]
```

Calculate the product of a list of numbers:

```
> product [1,2,3,4,5]  
120
```

Append two lists:

```
> [1,2,3] ++ [4,5]  
[1,2,3,4,5]
```

Reverse a list:

```
> reverse [1,2,3,4,5]  
[5,4,3,2,1]
```

Function Application

In mathematics, function application is denoted using parentheses, and multiplication is often denoted using juxtaposition or space.

$$f(a,b) + c d$$



Apply the function f to a and b , and add the result to the product of c and d .

In Haskell, function application is denoted using space, and multiplication is denoted using *.

f a b + c*d

If I wrote $f(a+b) + c*d$
↓
treated as product

As previously, but in Haskell syntax.

Moreover, function application is assumed to have higher priority than all other operators.

$f a + b$



Means $(f a) + b$, rather than $f (a + b)$.

Examples

Mathematics

 $f(x)$ $f(x,y)$ $f(g(x))$ $f(x,g(y))$ $f(x)g(y)$

Haskell

f xf x yf (g x) $f\;x\;(g\;y)$ $f\;x\;*\;g\;y$

Types

All computation in Haskell is done via evaluation of expressions to get values. Every value has an associated type, and is a first class object.

Examples:

- { Integer ✓
- { Int ✓
- Float
- Char
- Integer->Integer ↗
- [a] -> Integer

e :: t

If evaluating an expression e would produce a value of type t, then e has type t, written

All type errors are found at compile time, which makes programs safer and faster by removing the need for type checks at run time.

In GHCi, the :type command calculates the type of an expression, without evaluating it:

```
> not False  
True  
  
> :type not False  
not False :: Bool  
  
> :t head  
head :: [a] -> a
```

My First Script

When developing a Haskell script, it is useful to keep two windows open, one running an editor for the script, and the other running GHCi.

Start an editor, type in the following two function definitions, and save the script as test.hs:

```
myDouble :: Num a => a -> a  
myDouble x = x + x
```

```
intDouble :: Int -> Int  
intDouble x = x + x
```

Leaving the editor open, in another window start up GHCi with the new script:

```
% ghci test.hs
```

Useful GHCi Commands

Command

Meaning

:load name load script *name*

:reload reload current script

:edit name edit script *name*

:edit edit current script

:type expr show type of *expr*

:? show all commands

:quit quit GHCi

Leaving GHCi open, return to the editor, add the following two definitions, and resave:

```
myFactorial n = product [1..n]
```

```
myAverage ns = sum ns `div` length ns
```

Note:

- div is enclosed in back quotes, not forward;
- x `f` y is just syntactic for f x y.
- So this is just saying div (sum ns) (length ns)

GHCi does not automatically detect that the script has been changed, so a reload command must be executed before the new definitions can be used:

```
> :reload  
Reading file "test.hs"  
  
> factorial 10  
3628800  
  
> average [1,2,3,4,5]  
3
```

Functions are often done using pattern matching,
as well as a declaration of type (more later):

```
head      :: [a] -> a  
head (x:xs) = x
```

Now you try one...

Exercise

Write a program to find the number of elements in a list. Here's how to start:

```
myLength      :: [a] -> Int
```

```
myLength []    = 0
```

```
myLength (x:xs) = 1 + myLength xs
```

How do you test this to be sure it works?

Naming Requirements

- Function and argument names must begin with a lower-case letter.
For example:

myFun

fun1

arg_2

x'

- By convention, list arguments usually have an s suffix on their name. For example:

xs

ns

nss

The Layout Rule

In a sequence of definitions, each definition must begin in precisely the same column:

a = 10

b = 20

c = 30

a = 10

b = 20

c = 30

a = 10

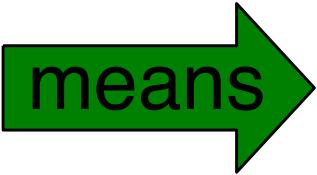
b = 20

c = 30



The layout rule avoids the need for explicit syntax to indicate the grouping of definitions.

```
a = b + c  
where  
  b = 1  
  c = 2  
d = a * 2
```

means 

```
a = b + c  
where  
  {b = 1;  
   c = 2}  
d = a * 2
```

implicit grouping

explicit grouping

Exercise

Fix the syntax errors in the program below, and test your solution using GHCi.

```
N = a `div` length xs
```

where

```
  a = 10
```

```
  xs = [1,2,3,4,5]
```

(circled up)

What is a Type?

A type is a name for a collection of related values.
For example, in Haskell the basic type

Bool

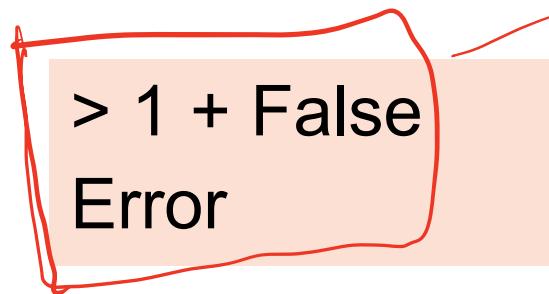
contains the two logical values:

False

True

Type Errors

Applying a function to one or more arguments of the wrong type is called a type error.



```
> 1 + False  
Error
```

worrs on c & JavaScript

1 is a number and False is a logical value, but + requires two numbers.

Basic Types

Haskell has a number of basic types, including:

Bool

- logical values

Char

- single characters

String

- strings of characters

Int

- fixed-precision integers

Integer

- arbitrary-precision integers

Float

- floating-point numbers

List Types

A list is sequence of values of the same type:

[False,True,False] :: [Bool]

['a','b','c'] :: [Char]

[['a'],['b', 'c']] :: [[Char]]

In general:

[t] is the type of lists with elements of type t.

Tuple Types

A tuple is a sequence of values of different types:

(False,True) :: (Bool,Bool)

(False,'a',True) :: (Bool,Char,Bool)

In general:

(t_1, t_2, \dots, t_n) is the type of n-tuples whose ith components have type t_i for any i in $1 \dots n$.

Function Types

A function is a mapping from values of one type to values of another type:

```
not    :: Bool -> Bool
```

```
isDigit :: Char -> Bool
```

In general:

$t_1 \rightarrow t_2$ is the type of functions that map values of type t_1 to values to type t_2 .

The arrow  is typed at the keyboard as `->`.

The argument and result types are unrestricted.
For example, functions with multiple arguments or
results are possible using lists or tuples:

```
addXY    :: (Int,Int) -> Int
addXY (x,y) = x+y
```

```
zeroto   :: Int -> [Int]
zeroto n = [0..n]
```

$$(myAdd x) y = x+y$$

★ Curried Functions

Functions with multiple arguments are also possible by returning functions as results:

another function

myAdd :: $\boxed{\text{Int} \rightarrow \text{Int}}$ \rightarrow Int

myAdd x y = x+y

map (myAdd 6) [1,2,3,4] \rightarrow [7, 8, 9, 10]

myAdd2 = myAdd 2

\hookrightarrow 2

$\text{Int} \rightarrow \text{Int}$

myAdd takes one argument at a time

map myAdd2 [1..10]

[3, 4 ... 12]

As a consequence, it is then natural for function application to associate to the left.

mult x y z

Means $((\text{mult } x) y) z.$

Unless tupling is explicitly required, all functions in Haskell are normally defined in curried form.

Polymorphic Functions

A function is called polymorphic ("of many forms") if its type contains one or more type variables.

length :: [a]  Int

[a] \rightarrow Int

myLength :: [a] \rightarrow Int
myLength [] = 0
myLength (x : xs) = 1 + myLength xs

for any type a, length takes a list of values of type a and returns an integer.

Ranges in Haskell

As already discussed, Haskell has extraordinary range capability on lists:

```
ghci> [1..15]
```

```
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
```

```
ghci> ['a'..'z']
```

```
"abcdefghijklmnopqrstuvwxyz"
```

```
ghci> ['K'..'Z']
```

```
"JKLMNOPQRSTUVWXYZ"
```

```
ghci> [2,4..20]
```

```
[2,4,6,8,10,12,14,16,18,20]
```

```
ghci> [3,6..20]
```

```
[3,6,9,12,15,18]
```

Infinite Lists

A few useful infinite list functions:

```
ghci> take 10 (cycle [1,2,3])  
[1,2,3,1,2,3,1,2,3,1]  
ghci> take 12 (cycle "LOL ") ??.?  
"LOL LOL LOL "
```

```
ghci> take 10 (repeat 5)  
[5,5,5,5,5,5,5,5,5,5]
```

List Comprehensions

→ it always returns a list.

Very similar to standard set theory list notation:

ghci> [x*2 | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]

Annotations: 'output' points to the expression `x*2`, 'load to x' points to the variable `x` in the list comprehension, and 'input as a list' points to the list `[1..10]`.

Can even add ~~predicates~~ to the comprehension:

ghci> [x*2 | x <- [1..10], x*2 >= 12]
[12,14,16,18,20]

ghci> [x | x <- [50..100], x `mod` 7 == 3]
[52,59,66,73,80,87,94]

Annotations: 'output' points to the expression `x*2`, 'load to x' points to the variable `x`, and 'input' points to the list `[1..10]`. In the second example, 'output' points to `x`, 'load to x' points to `x`, and 'input' points to the list `[50..100]`.

→ filter.
Predicates, a predicate is a statement can be true or false.

multiple Generators:

twoFour = [(x,y) | x <- [1..2], y <- [1..4]]

Annotations: 'two input' points to the list `[1..2]` under the first generator, and 'two input' also points to the list `[1..4]` under the second generator.

$\rightarrow [(1,1), (1,2), (1,3), (1,4)]$
 $[(2,1), (2,2), (2,3), (2,4)]$

Dependent Generators :

array a b = [(x,y) | x <= [1..a], y <= [x..b]]

Output: array 3 3

$[(1,1), (1,2), (1,3), (2,2), (2,3), (3,3)]$

```
GHCi, version 8.0.1: http://www.haskell.org/ghc/ :? for help
Prelude> [2..20]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [y^2 | y<-[1..10]]
[2,4,6,8,10,12,14,16,18,20]
Prelude> [y^2 | y<-[1..10], y^2 >= 14]
[14,16,18,20]
Prelude> [y | y <-[10..30], mod y 7 == 3]
[10,17,24]
Prelude> [y | y <-[10..30], mod y 7 == 0]
[14,21,28]
Prelude> [y | y <-[10..30], y `mod` 7 == 0]
[14,21,28]
Prelude> happySad xs = [ if x < 10 then "happy" else "sad" | x <- xs, odd x]
Prelude> happySad [7..13]
["happy","happy","sad","sad"]
Prelude> [7..13]
[7,8,9,10,11,12,13]
Prelude> [x | x <-[10..30], x/=20, x /=10, x/=30]
[11,12,13,14,15,16,17,18,19,21,22,23,24,25,26,27,28,29]
Prelude> [ x*y | x<- [2..4..6], y<-[3..5..7] ]
[6,10,14,12,20,28,18,30,42]
Prelude> [ x*y | x<- [2..4..6], y<-[3..5..7], x*y >20]
[28,30,42]
Prelude> lengthN xs = sum [1| x<-xs] → sum [1,1,1]
Prelude> lengthN [1,2,3] → then we get length
3
Prelude> lengthN [1,2,3, 6, 8]
5
Prelude>
```

$\downarrow \text{lengthN } xs = \sum [1 | x \in xs]$ input a list

pattern match
output 1

$\rightarrow \text{then } \sum [1, 1 \dots x \dots]$

Pattern Matching

Functions can often be defined in many different ways using pattern matching. For example

```
(&&)      :: Bool  Bool  Bool  
True && True = True  
True && False = False  
False && True = False  
False && False = False
```

can be defined more compactly by

```
True && True = True  
_ && _ = False
```

However, the following definition is more efficient, because it avoids evaluating the second argument if the first argument is False:

```
True && b = b  
False && _ = False
```

Note: The underscore symbol `_` is a wildcard pattern that matches any argument value.

Patterns are matched in order. For example, the following definition always returns False:

```
_ && _ = False
```

```
True && True = True
```

List Patterns

Internally, every non-empty list is constructed by repeated use of an operator (:) called “cons” that adds an element to the start of a list.

[1,2,3,4]

Means 1:(2:(3:(4:[]))).

return []

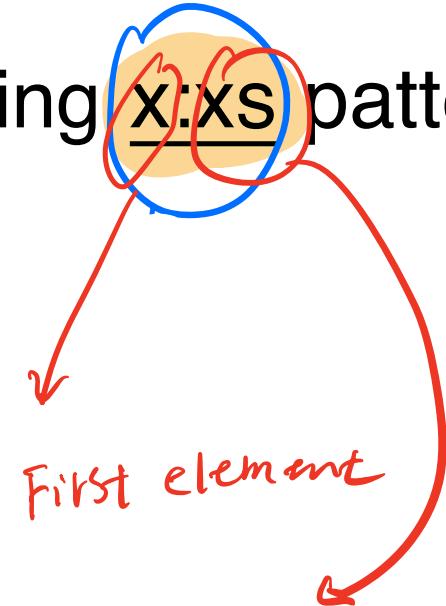
Functions on lists can be defined using $x:xs$ patterns.

head :: [a] \rightarrow a

head ($x:_$) = x

tail :: [a] \rightarrow [a]

tail ($_:xs$) = xs



head and tail map any non-empty list to its first and remaining elements.



Type of functions

It's good practice (and REQUIRED in this class) to also give functions types in your definitions.

removeNonUppercase :: [Char] -> [Char]

removeNonUppercase st =

[c | c <- st, c `elem` ['A'..'Z']]

list comprehension

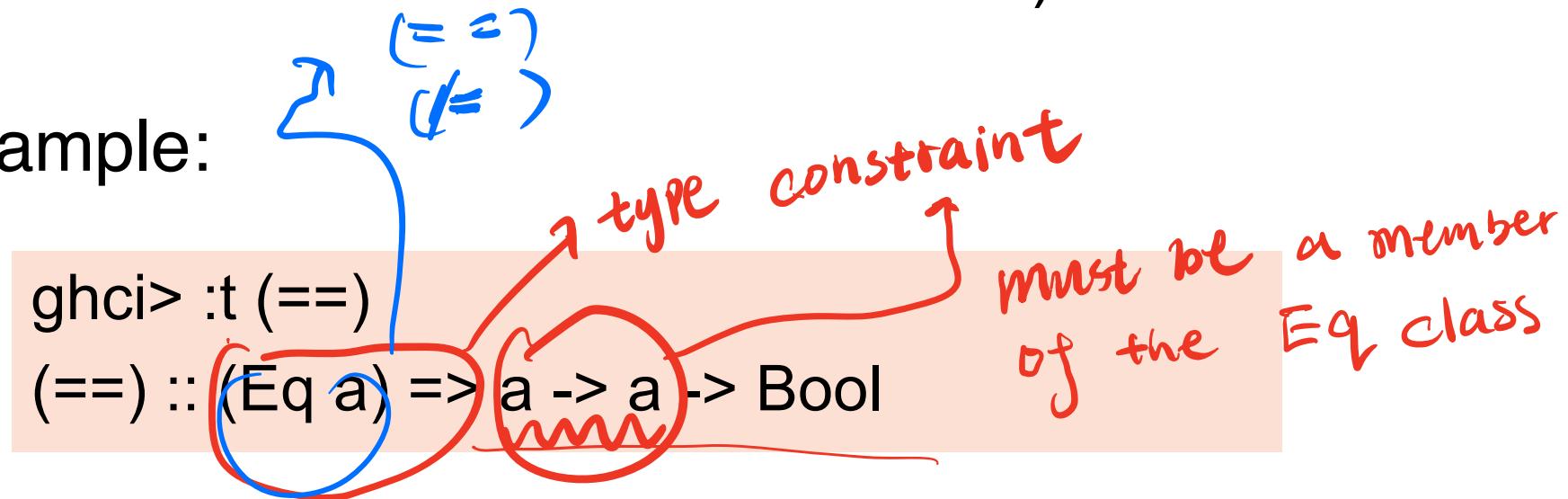
addThree :: Int -> Int -> Int -> Int

addThree x y z = x + y + z

Type Classes

In a typeclass, we group types by what behaviors are supported. (These are NOT object oriented classes – closer to Java interfaces.)

Example:



Everything before the \Rightarrow is called a type constraint, so the two inputs must be of a type that is a member of the Eq class.

Type Classes

Other useful typeclasses:

$\leftarrow \leftarrow = \rightarrow \rightarrow = \max, \min$

- **Ord** is anything that has an ordering.
- **Show** are things that can be presented as strings.
- **Enum** is anything that can be sequentially ordered.
- **Bounded** means has a lower and upper bound.
- **Num** is a numeric typeclass – so things have to “act” like numbers. $(+)$ $(-)$ $(*)$ Negate
- **Integral** and **Floating** what they seem.

deriving (show, Eq)

Overloaded Functions

A polymorphic function is called overloaded if its type contains one or more class constraints.

sum :: Num a \rightarrow [a] \rightarrow a

for any numeric type a, sum takes a list of values of type a and returns a value of type a.

Overloaded functions

Constrained type variables can be instantiated to any types that satisfy the constraints:

```
> sum [1,2,3]
```

```
6
```

```
> sum [1.1,2.2,3.3]
```

```
6.6
```

```
> sum ['a','b','c']
```

```
ERROR
```

a = Int

a = Float

Char is not a
numeric type



Useful functions: map

applies a function

→ : applies a function to each element.

The function map applies a function across a list:

Function list elem
map :: (a -> b) -> [a] -> [b] → return list elem
map [] = []
map f (x:xs) = f x : map f xs

```
ghci> map (+3) [1,5,3,1,6]  
[4,8,6,4,9]
```

```
ghci> map (replicate 3) [3..6]  
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
```

```
ghci> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]  
[[1,4],[9,16,25,36],[49,64]]
```

Useful functions: filter

The function filter: → Filter selects all items from L that satisfy Property P

filter :: $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a]$

filter [] = []

filter p (x:xs)

| p x = x : filter p xs

| otherwise = filter p xs

extracts elements that match a property

```
ghci> filter (>3) [1,5,3,2,1,6,4,3,2,1]
```

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

```
ghci> filter (==3) [1,2,3,4,5]
```

```
[3]
```

```
ghci> filter even [1..10]
```

```
[2,4,6,8,10]
```

combining map & filter

sqr-even :: $\text{[Int]} \rightarrow \text{[Int]}$

sqr-even l = map sqrt filter is-even l

e.g. [1, 2, 6, 9, 11, 14]

filter = [2, 6, 14]

map = [4, 36, 196]

Conditional Expressions

As in most programming languages, functions can be defined using conditional expressions.

```
abs :: Int → Int  
abs n = if n > 0 then n else -n
```

Note: In Haskell, conditional expressions must always have an else branch, which avoids any possible ambiguity issues.

Guarded Equations

As an alternative to conditionals, functions can also be defined using guarded equations.

abs n

| $n \neq 0$ = n

| otherwise = -n

As previously, but using guarded equations.

Exercise

Create a function allX that returns a list of the x's in a list of ordered pairs (x,y)

```
allX    :: [(Int, Int)] -> [Int]
```

```
allX :: [(Int, Int)] -> [Int]
allX xs = [x | (x,y) <- xs]
```

Exercise

Create a function evens that returns a list of even numbers from a list.

```
evens :: [Int] -> [Int]
```

```
evens :: [Int] -> [Int]
evens []      = []
 $\exists$  evens (x:xs)
  | mod x 2 == 0 = x: evens xs
  | otherwise     = evens xs
```

Exercise

Create a function myElem that returns True if an element is in a given list and returns False otherwise

```
myElem    :: (Eq a) -> a -> [a] -> Bool
```

myElem :: (Eq a) => a -> [a] -> Bool
myElem [] = False
myElem e (x:xs) = (e == x) || (myElem e xs)

$$e = 6 \quad [1, 3, 6, 9]$$

$$e = 6 \quad l: [3, 6, 9] \\ x = 1 \quad || \quad e = 6 \quad [3, 6, 9] \quad || \quad e = 6 \quad [6, 9]$$

F OR F T

Exercise

True

Create a function `isAsc` that returns True if the list given is in ascending order

`isAsc :: [Int] -> Bool`

```
isAsc :: [Int] -> Bool  
isAsc [] = True  
isAsc [x] = True  
isAsc (x:y:xs) = (x <= y) && isAsc (y:xs) ✓
```

first ever second the rest

Exercise

Create a list of the factors of an integer

```
factors :: Int -> [Int]
```

```
factors :: Int -> [Int]
```

```
factors n =
```

```
| [x | x <- [1..n], n `mod` x == 0]
```

Factors 3 :

[x | x <- [1, 2, 3], 3 `mod` x == 0]

3 `mod` 1 == 0

Output

[1, 3]

Lambda Expressions

→ use for:
o rearranging data
o the order of parameters

Functions can be constructed without naming the functions by using lambda expressions.

$$\lambda x \rightarrow x+x$$

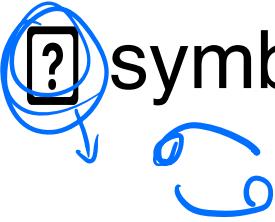
$$\lambda x \rightarrow x+x$$

the nameless function that takes a
number x and returns the result x+x.

$$(\lambda x \rightarrow x+1)$$
$$(\lambda x y z \rightarrow x+y+z)$$

$$\text{eg: } (\lambda x \rightarrow x+1) 1 \Rightarrow 2$$

Note:

-  The symbol λ is the Greek letter lambda, and is typed at the keyboard as a backslash \.
-  In mathematics, nameless functions are usually denoted using the λ symbol, as in $x \lambda x+x$.

-  In Haskell, the use of the λ symbol for nameless functions comes from the lambda calculus, the theory of functions on which Haskell is based.

Lambda expressions are useful when defining functions that return functions as results.

For example:

```
const :: a → b → a  
const x_ = x
```

is more naturally defined by

```
const :: a → (b → a)  
const x = \_ → x
```

Lambda expressions can be used to avoid naming functions that are only referenced once.

For example:

odds n = map f [0..n-1]

where

$$f x = \underline{x^2 + 1}$$

can be simplified to

odds n = map (\x  x² + 1) [0..n-1]

Sections

An operator written between its two arguments can be converted into a curried function written before its two arguments by using parentheses.

For example:

```
> 1+2
```

```
3
```

```
> (+) 1 2
```

```
3
```

This convention also allows one of the arguments of the operator to be included in the parentheses.

For example:

```
> (1+) 2
```

```
3
```

```
> (+2) 1
```

```
3
```

In general, if \oplus is an operator then functions of the form (\oplus) , $(x\oplus)$ and $(\oplus y)$ are called sections.

Why Are Sections Useful?

Useful functions can sometimes be constructed in a simple way using sections. For example:

(1+) - successor function

(1/) - reciprocation function

(*2) - doubling function

(/2) - halving function

Higher-order Functions

A function is higher-order if it takes a function as argument or returns a function as a result.

twice :: $(a \rightarrow a) \rightarrow a \rightarrow a$
twice $f x = f(f(x))$

app :: $(a \rightarrow b) \rightarrow a \rightarrow b$
app :: $f x = fx$
app $(\lambda x \rightarrow x+1) 1$

Twice is higher-order because it takes a function as its first argument.

map :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$
map $(\lambda x \rightarrow x+1) [1..5]$
 $\Rightarrow [2, 3, 4, 5, 6]$

map :: $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$
map $(\lambda (x,y) \rightarrow x+y) [(1,2), (2,3), (3,4)]$
 $\Rightarrow [3, 5, 7]$

The Foldr Function

$\text{Foldr} :: (\alpha \rightarrow \beta \rightarrow \beta) \dashv \beta \rightarrow [\alpha] \rightarrow \beta$

A number of functions on lists can be defined using the following simple pattern of recursion:

$$f [] = v$$

$$f (x:xs) = x \oplus f xs$$

f maps the empty list to some value v, and any non-empty list to some function ~~⊕~~ applied to its head and f of its tail.

For example:

sum [] = 0

sum (x:xs) = x + sum xs

v = 0
⊕ = +

product [] = 1

product (x:xs) = x * product xs

v = 1
⊗ = *

and [] = True

and (x:xs) = x && and xs

v = True
⊕ = &&

The higher-order library function foldr (fold right) encapsulates this simple pattern of recursion, with the function \oplus and the value v as arguments.

For example:

sum = foldr (+) 0

product = foldr (*) 1

or = foldr (||) False

and = foldr (&&) True

Foldr itself can be defined using recursion:

foldr :: (a → b → b) → b → [a] → b

foldr f v [] = v

foldr f v (x:xs) = f x (foldr f v xs)

However, it is best to think of foldr non-recursively, as simultaneously replacing each (:) in a list by a given function, and [] by a given value.

For example:

sum [1,2,3]

=

foldr (+) 0 [1,2,3]

=

foldr (+) 0 (1:(2:(3:[])))

=

1+(2+(3+0))

=

6

Replace each (:) by (+) and [] by 0.

For example:

product [1,2,3]

=

foldr (*) 1 [1,2,3]

=

foldr (*) 1 (1:(2:(3:[])))

=

1*(2*(3*1))

=

6

Replace each (:) by (*) and [] by 1.

Now recall the reverse function:

```
reverse [] = []
```

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

For example:

```
reverse [1,2,3]
```

=

```
reverse (1:(2:(3:[])))
```

=

```
(([] ++ [3]) ++ [2]) ++ [1]
```

=

```
[3,2,1]
```

Replace each (:) by
x xs ~~()~~ xs ++ [x]
and [] by [].

The library function all decides if every element of a list satisfies a given predicate.

```
all    :: (a → Bool) → [a] → Bool  
all p xs = and [p x | x ← xs]
```

For example:

```
> all even [2,4,6,8,10]
```

```
True
```

Type Declarations

In Haskell, a new name for an existing type can be defined using a type declaration.

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

String is a synonym for the type [Char].

Type declarations can be used to make other types easier to read. For example, given

```
type Point = (Int,Int)
```

we can define:

```
origin :: Point  
origin = (0,0)
```

```
left :: Point → Point  
left (x,y) = (x-1,y)
```

Like function definitions, type declarations can also have parameters. For example, given

```
type Pair a = (a,a)
```

we can define:

```
mult    :: Pair Int → Int  
mult (m,n) = m*n
```

```
copy    :: a → Pair a  
copy x  = (x,x)
```

Data Declarations

A completely new type can be defined by specifying its values using a data declaration.

```
data Pet = Cat | Dog
```

Pet is a new type, with two new values
Cat and Dog.

- The two values Cat and Dog are called the constructors for the type Pet.
- Type and constructor names must begin with an upper-case letter.
- Data declarations are similar to context free grammars. The former specifies the values of a type, the latter the sentences of a language.

Values of new types can be used in the same ways as those of built in types. For example, given

```
data Answer = Yes | No | Unknown
```

we can define:

```
answers    :: [Answer]  
answers    = [Yes, No, Unknown]
```

```
flip      :: Answer → Answer
```

```
flip Yes   = No
```

```
flip No    = Yes
```

```
flip Unknown = Unknown
```

The constructors in a data declaration can also have parameters. For example, given

```
data Shape = Circle Float | Rect Float Float
```

we can define:

```
square      :: Float -> Shape  
square n    = Rect n n
```

```
area        :: Shape -> Float  
area (Circle r) = pi * r^2  
area (Rect x y) = x * y
```

Note:

- Shape has values of the form Circle r where r is a float, and Rect x y where x and y are floats.
- Circle and Rect can be viewed as functions that construct values of type Shape:

Circle :: Float  Shape

Rect :: Float  Float  Shape

Not surprisingly, data declarations themselves can also have parameters. For example, given

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

we can define:

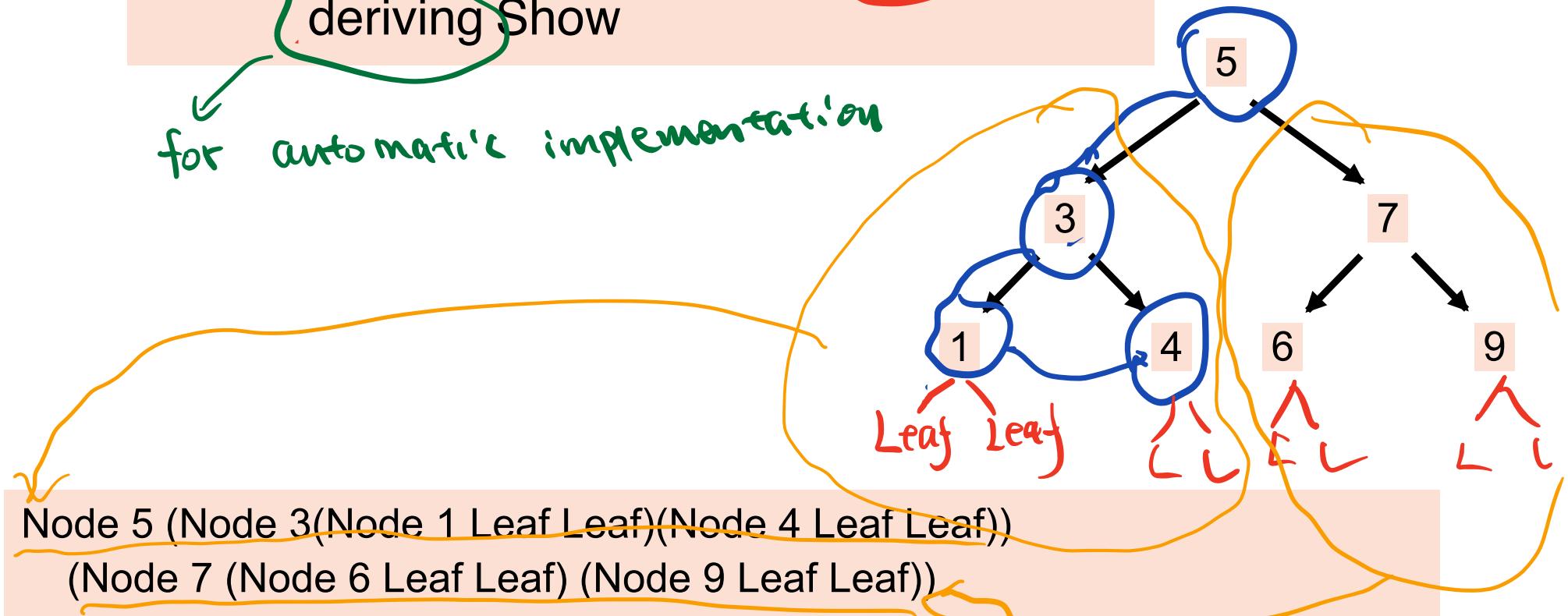
```
safediv :: Int ? Int ? Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)
```

Binary Trees HW 2

In computing, it is often useful to store data in a two-way branching structure or binary tree.

```
data Tree = Node Int Tree Tree | Leaf  
deriving Show
```

for automatic implementation



We can now define a function for a traversal of a binary tree:

L , R - R

inorder :: Tree -> [Int]

inorder Leaf = []

inorder (Node x l r) = inorder l ++ [x] ++ inorder r

↓ [1, 3, 4, 5, 6, 7, 9]

Let t5 = Node 1 (Node 2 Leaf Leaf) (Node1 Leaf Leaf)

= [2, 1, 1)

Search trees have the important property that when trying to find a value in a tree we can always decide which of the two sub-trees it may occur in:

occurs m (Leaf) = False

occurs m (Node n l r)

| m==n = True

| $m < n$ = occurs m l

| $m > n$ = occurs m r

This definition is efficient, because it only traverses one path down the tree.

Modules

So far, we've been using built-in functions provided in the Haskell prelude. This is a subset of a larger library that is provided with any installation of Haskell.

Examples of other modules:

- lists
- concurrent programming
- complex numbers
- char
- sets
- ...

Example: Data.List

To load a module, we need to import it:

```
import Data.List
```

All the functions in this module are immediately available:

```
numUniques :: (Eq a) => [a] -> Int  
numUniques = length . nub
```

function
concatenation

This is a function in Data.List
that removes duplicates from a
list.

File I/O

So far, we've worked mainly at the prompt, and done very little true input or output. This is logical in a functional language, since nothing has side effects!

However, this is a problem with I/O, since the whole point is to take input (and hence change some value) and then output something (which requires changing the state of the screen or other I/O device).

Luckily, Haskell offers work-arounds that separate the more imperative I/O.

A simple example: save the following file as helloworld.hs

```
main = putStrLn "hello, world"
```

Now we actually compile a program:

```
$ ghc --make helloworld
[1 of 1] Compiling Main
      ( helloworld.hs, helloworld.o )
Linking helloworld ...
$ ./helloworld
hello, world
```

What are these functions?

```
ghci> :t putStrLn  
putStrLn :: String -> IO ()  
ghci> :t putStrLn "hello, world"  
putStrLn "hello, world" :: IO ()
```

So `putStrLn` takes a string and returns an I/O action (which has a result type of `()`, the empty tuple).

In Haskell, an I/O action is one with a side effect - usually either reading or printing. Usually some kind of a return value, where `()` is a dummy value for no return.

More on getLine:

```
ghci> :t getLine  
getLine :: IO String
```

This is the first I/O we've seen that doesn't have an empty tuple type - it has a String.

Once the string is returned, we use the <- to bind the result to the specified identifier.

Notice this is the first non-functional action we've seen, since this function will NOT have the same value every time it is run! This is called "impure" code, and the value name is "tainted".

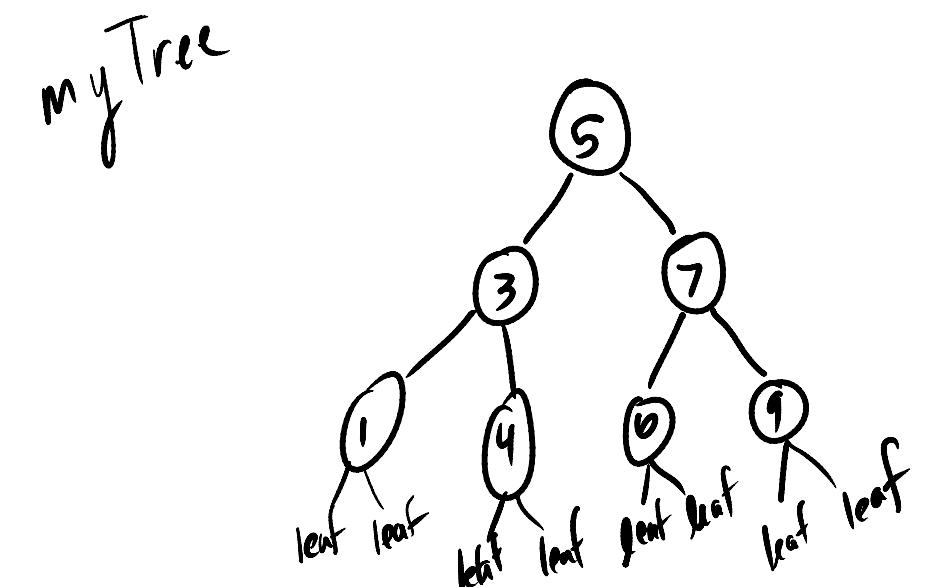
-- Trees

hw 2

```
data Tree = Node Int Tree Tree | Leaf  
deriving Show
```

```
myTree :: Tree
```

```
myTree = Node 5 (Node 3(Node 1 Leaf Leaf)(Node 4 Leaf Leaf))  
          (Node 7 (Node 6 Leaf Leaf) (Node 9 Leaf Leaf))
```



```
singleton :: Int -> Tree
singleton x = Node x Leaf Leaf
```

```
Ok, two modules loaded.
*Main> singleton 5
Node 5 Leaf Leaf
```

```
treeInsert :: Int -> Tree -> Tree
treeInsert x Leaf = singleton x ✓
treeInsert x (Node n left right)
| x == n = Node n left right
| x < n  = Node n (treeInsert x left) right
| x > n  = Node n left (treeInsert x right)
```

```
*Main> let myTree = singleton 5
*Main> myTree
Node 5 Leaf Leaf
*Main> treeInsert 4 myTree
Node 5 (Node 4 Leaf Leaf) Leaf
```

* - Use a foldr to build a tree from a List

```
numbers = [5, 7, 2, 9, 11, 6]  
numberTree = foldr treeInsert Leaf numbers
```

!
Leaf
base case
bottom

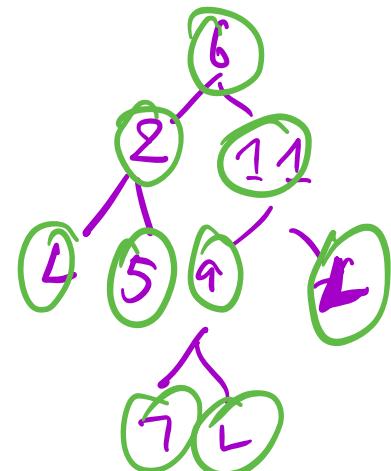
```
-- Use a foldr to build a tree from a List  
numbers = [5, 7, 2, 9, 11, 6]  
numberTree = foldr treeInsert Leaf numbers
```

```
*Main> numbers  
[5,7,2,9,11,6]  
*Main> numberTree = foldr treeInsert Leaf numbers  
*Main> numberTree  
Node 6 (Node 2 Leaf (Node 5 Leaf Leaf)) (Node 11 (Node 9 (Node 7 Leaf Leaf) Leaf) Leaf)
```

[5,7,2,9,11,6]

5 Insert 7 (tree2) tree9 tree11,

tree 6
Node 6
Leaf Leaf



```
|  
| -- Use a foldr to build a tree from a List  
numbers = [5, 7, 2, 9, 11, 6]  
numberTree = foldr treeInsert Leaf numbers
```

```
buildTree :: [Int] -> Tree  
buildTree xs = foldr treeInsert Leaf xs
```

```
|  
| -- Use a foldr to build a tree from a List  
numbers = [5, 7, 2, 9, 11, 6]  
numberTree = foldr treeInsert Leaf numbers  
  
buildTree :: [Int] -> Tree  
buildTree xs = foldr treeInsert Leaf xs
```

*Main>/buildTree [2,6,8,5]

Node 5 (Node 2 Leaf Leaf) (Node 8 (Node 6 Leaf Leaf) Leaf)

```
mergeTrees :: Tree -> Tree -> Tree
mergeTrees xs ys = let zs = inorder (ys) in
    foldr treeInsert xs zs
```



```
*Main> t1
Node 5 (Node 2 Leaf Leaf) (Node 8 (Node 6 Leaf Leaf) Leaf)
*Main> t2
Node 4 (Node 3 Leaf Leaf) (Node 7 Leaf (Node 10 (Node 9 Leaf Leaf) Leaf))
*Main> let t3 = mergeTrees t1 t2
*Main> t3
Node 5 (Node 2 Leaf (Node 4 (Node 3 Leaf Leaf) Leaf)) (Node 8 (Node 6 Leaf (Node 7 Leaf Leaf))
(Node 10 (Node 9 Leaf Leaf) Leaf))
```

```
inorder :: Tree -> [Int]
inorder Leaf = []
inorder (Node x l r) = inorder l ++ [x] ++ inorder r
```

```
*Main> inorder t1
[2,5,6,8]
*Main> inorder t2
[3,4,7,9,10]
*Main> inorder t3
[2,3,4,5,6,7,8,9,10]
```

```
occurs :: Int -> Tree -> Bool
occurs m (Leaf) = False
occurs m (Node n l r)
| m==n = True
| m<n = occurs m l
| m>n = occurs m r
```

```
myGraph :: Graph  
myGraph = [(1,2), (2,3), (1,4), (4,1), (5, 6), (2,6)]
```

-- hasPath directed graph

```
hasPath :: Graph -> Node -> Node -> Bool
hasPath [] x y = x == y
hasPath xs x y
| x == y = True
| otherwise =
  let xs' = [ (n,m) | (n,m) <- xs, n /= x ] in
  or [ hasPath xs' m y | (n,m) <- xs, n == x ]
```

```
| completeG :: Int -> Graph  
completeG n = [ (x,y) | x <- [1..n], y<-[1..n] ]
```

```
*Main> completeG 3  
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,1),(3,2),(3,3)]  
*Main> █
```

```
vertexList :: Graph -> [Node]
vertexList [] = []
vertexList xs = nub ([ x | (x,y) <- xs] `union` [ y | (x,y) <- xs])
```

```
-- Strongly Connected
stcon :: Graph -> Bool
stcon [] = False
stcon xs = let ys = vertexList xs in
| | | | | and [ hasPath xs x y | x <- ys, y <- ys, x /= y ]
```

CS 381 - Activity 1

Name your file Act1Functions.hs with the first line: module Act1Functions where

You can test your functions using the act1Verifier.hs file. ghc act1Verifier.

1. Write a Haskell function called “range” that produces a consecutive list of numbers that starts with the number n given as an argument and ends at 1 [n...1] . You can assume that $n \geq 0$. For example:

```
*Main> range 0
[]
*Main> range 1
[1]
*Main> range 4
[4,3,2,1]
*Main> range 5
[5,4,3,2,1]
```

2. Write a Haskell function called “copies” that produces a list of n copies of a list of integers. You can assume that $n \geq 0$. For examples:

```
*Main> copies 0 []
[]
*Main> copies 1 []
[[]]
*Main> copies 3 [1,2,3,4]
[[1,2,3,4],[1,2,3,4],[1,2,3,4]]
*Main> copies 1 [2,3]
[[2,3]]
*Main> copies 0 [1,2,3]
[]
```

3. Write a Haskell function called “greaterList” that produces a list of elements from a list (second argument) that are greater than the first argument nFor example:

```
*Main> greaterList 5 [3,8,1,9,2]
[8,9]
*Main> greaterList 8 []
[]
*Main> greaterList 'g' "Hello"
"llo"
*Main> greaterList 2.3 [6.1, 1.2, 7.5, 0]
[6.1,7.5]
*Main> greaterList 8 [1,2,4,7]
[]
```

Hint: Use (Ord a) => a → [a] → [a]

4. Write a Haskell function called “allSame” that returns True if the characters in a list (or String) are all the same. If the characters differ or the list is empty the function evaluates to False.

CS 381 - Activity 1

```
*Main> allSame "aaa"
True
*Main> allSame ['a','b', 'c']
False
*Main> allSame []
False
*Main> allSame ['x']
True
*Main> allSame "Hello"
False
*Main> allSame ['a','a']
True
*Main> allSame ""
False
```

5. Write a Haskell function called “minmax” that returns as an pair the minimum and maximum value from a list of integers. Assume that the minimum and maximum of an empty list is (0,0). See the examples below.

```
*Main> minmax [1]
(1,1)
*Main> minmax [2,2,2]
(2,2)
*Main> minmax []
(0,0)
*Main> minmax [3,5,1,9, 10]
(1,10)
```

HW 2

- Submit solutions in HW2sol.hs file without a main function.
 - Module HW2sol where
 - Sname = “Your name”
- Comment code
 - Header – name, class, date
 - Comments before solutions to each question
- You can use hw2verifier.hs to test your functions.
 - ghc hw2verifier.hs // to create an executable
 - Ghci hw2verifier.hs // and then call main

week 3

Syntax

- Grammars & Derivations
 - Natural Languages
 - Formal Languages
 - Programming Languages
- Syntax Tree / Parse Tree
- Abstract vs Concrete Syntax
- Representing Grammars by Haskell DataTypes

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*).

LHS A ::= B C ... *RHS*

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

How are sentences generated by rules?

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

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

Example Grammar

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

nonterminal symbols

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

terminal symbols

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

Derivation

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

sentence

noun verb noun

dogs verb noun

dogs have noun

dogs have teeth

apply rule (R1)

apply rule (R2)

apply rule (R4)

apply rule (R3)

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

Derivation Order

The order of rule application is *not* fixed.

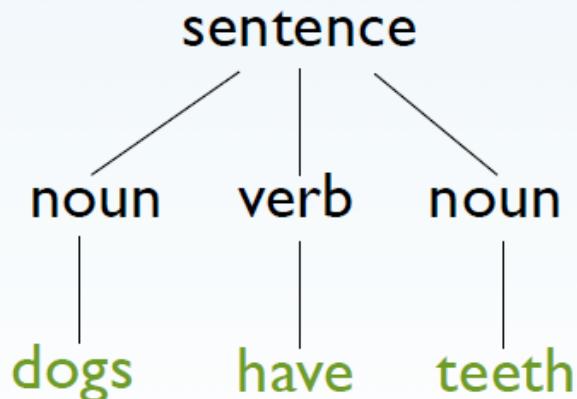
sentence	::=	noun verb noun	(R1)
noun	::=	dogs	(R2)
noun	::=	teeth	(R3)
verb	::=	have	(R4)

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)	

Syntax Tree

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

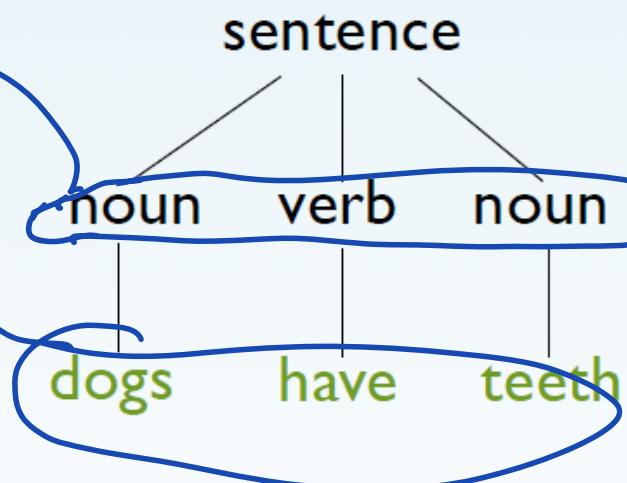


Derivation is a process of producing a sentence according to the rules of a grammar.

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

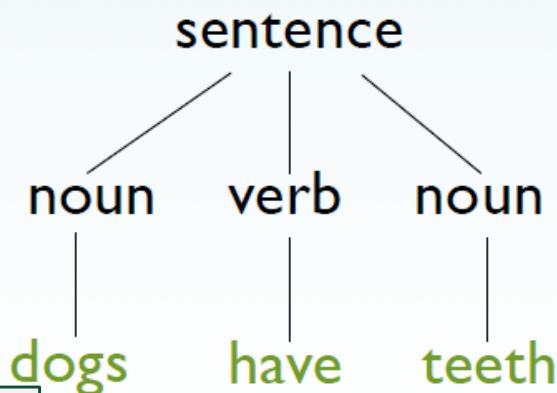
Observations About Syntax Trees

- (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



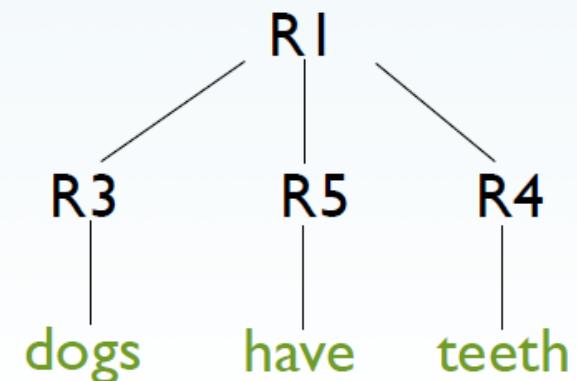
Alternative Representation

- (1) Leaves contain *terminal symbols*
(2) Internal nodes contain
nonterminal symbols

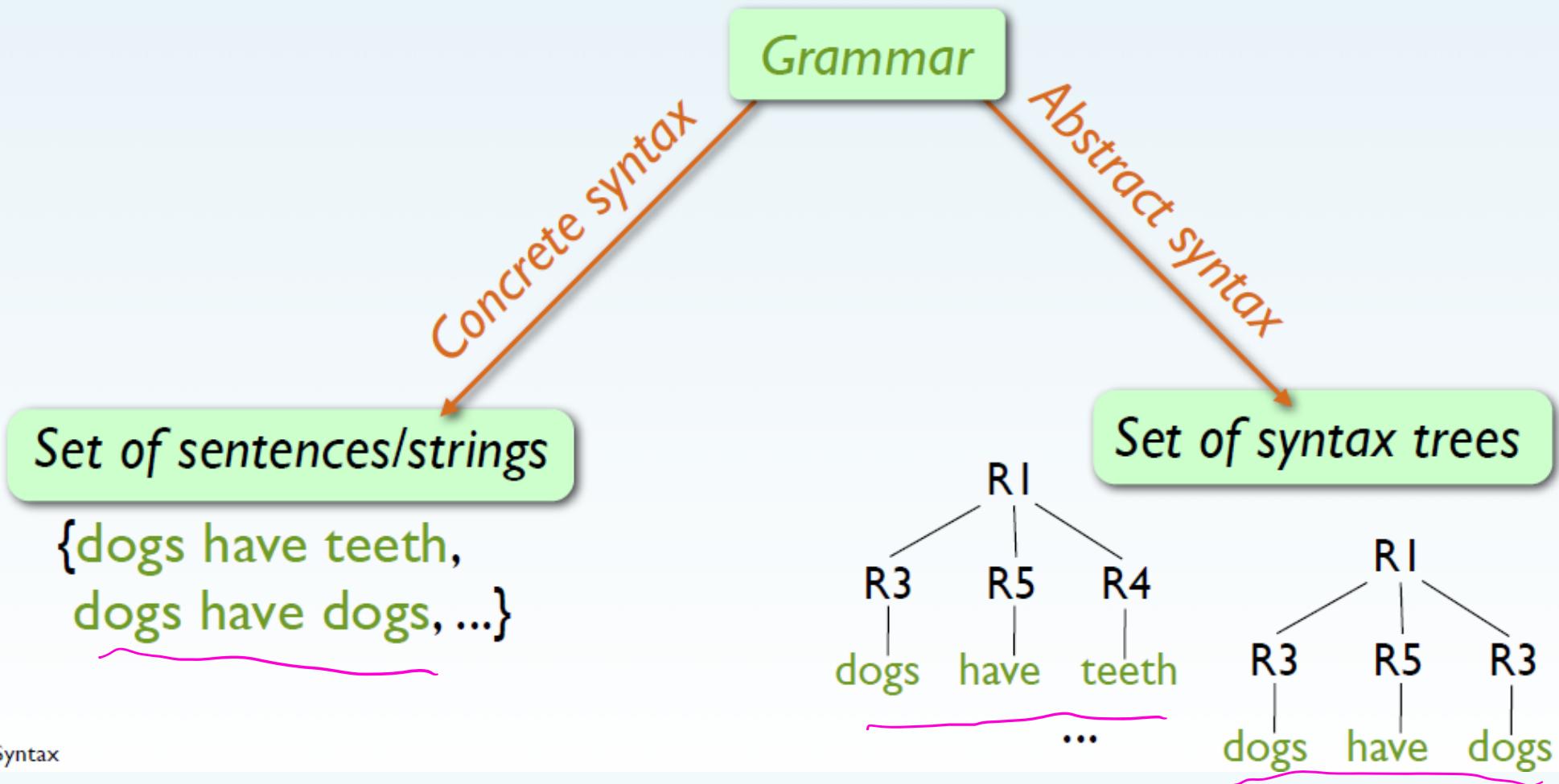


```
sentence ::= noun verb noun (R1)
sentence ::= sentence and sentence (R2)
noun     ::= dogs (R3)
noun     ::= teeth (R4)
verb     ::= have (R5)
```

- (1) Leaves contain *terminal symbols*
(2) Internal nodes contain *rule names*



Concrete vs. Abstract Syntax



(I) 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 **and** sentence (R1)
sentence ::= noun verb noun (R2)
noun ::= dogs (R3)
noun ::= teeth (R4)
noun ::= cats (R5)
noun ::= claws (R6)
verb ::= have (R7)

```
sentence  
cats have claws and dogs have teeth
```

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.

(I) 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 and sentence (R1)

sentence :: noun verb noun (R2)

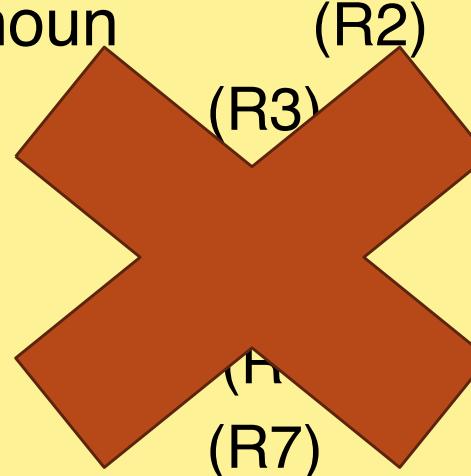
noun :: dogs

noun :: teeth

noun :: cats

noun :: claws

verb :: have



What if I wanted ??????
cats and dogs have teeth

cats and dogs have teeth

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

sentence ::= nounphrase verb noun (R1)

nounphrase ::= noun (R2a)

nounphrase ::= noun and noun (R2b)

noun ::= dogs (R3)

noun ::= teeth (R4)

noun ::= cats (R5)

noun ::= claws (R6)

verb ::= have (R7)

sentence	$::=$	nounphrase verb noun	(R1)
nounphrase	$::=$	noun	(R2a)
nounphrase	$::=$	noun and noun	(R2b)
noun	$::=$	dogs	(R3)
noun	$::=$	teeth	(R4)
noun	$::=$	cats	(R5)
noun	$::=$	claws	(R6)
verb	$::=$	have	(R7)

cats and dogs have teeth

sentence \Rightarrow nounphrase verb noun
 \Rightarrow noun and noun verb noun
 \Rightarrow cats and noun verb noun

 \Rightarrow cats and dogs have teeth

cats and dogs have teeth and claws

sentence ::= nounphrase verb nounphrase (R1)

nounphrase ::= noun (R2a)

nounphrase ::= noun and noun (R2b)

noun ::= dogs (R3)

noun ::= teeth (R4)

noun ::= cats (R5)

noun ::= claws (R6)

verb ::= have (R7)

sentence	::= nounphrase verb noun	(R1)
nounphrase	::= noun	(R2a)
nounphrase	::= multinoun and noun	(R2b)
multinoun	::= multinoun noun	(R2c)
multinoun	::= noun	(R2d)
noun	::= dogs cats bears	(R3)
noun	::= teeth claws	(R4)
verb	::= have	(R7)

cats bears and dogs have teeth

sentence \Rightarrow nounphrase verb noun
 \Rightarrow multinoun and noun verb noun
 \Rightarrow multinoun noun and noun verb noun
 \Rightarrow noun noun and noun verb noun

\Rightarrow cats bears and dogs have teeth



Formal Definition of a Context-Free Grammar

$$G = (N, \Sigma, P, S)$$

Formally, a grammar is defined as a four-tuple (N, Σ, P, S) where

- (1) N is a set of *nonterminal symbols*,
- (2) Σ is a set of *terminal symbols* with $N \cap \Sigma = \emptyset$,
- (3) $P \subseteq N \times (N \cup \Sigma)^*$ is a set of *productions*, and
- (4) $S \in N$ is the *start symbol*.

Definitions:

- A sequence of terminals is a *sentence*. A *sentence* $\boxed{?}$
- An intermediate sequence in the derivation that contains nonterminals is a *sentential form*. A *sentential form* $\boxed{?}(N \boxed{?})^*$
- A production (A, B) can also be written as
 - $A \rightarrow B$ where $A \in N$ and $B \in (N \cup \Sigma)^*$

Example

Let $G = (N, \Sigma, P, S)$ where $\Sigma = \{ 1, 2, 3 \}$ and $N = \{ S, A, B \}$

- $\Sigma^* = \{ \epsilon, S, A, B, 1, 1A, 2A, B3, A1B1, 12, 13, 21, \dots \}$
- $(N \cup \Sigma)^* = \{ S, A, B, 1, 1A, 2A, B3, A1B1, 12, 13, 21, \dots \}$
- $N \times (N \cup \Sigma)^* = \{ (S, A), (A, 111), (S, A1), (S, S1), (A, 2A), (B, 3), (A, 2B) \dots \}$
- Let $P = \{ (S, A1), (S, S1), (A, 2A), (A, B), (B, 33) \}$
 - $S \rightarrow A1 \mid S1$
 - $A \rightarrow 2A \mid B$
 - $B \rightarrow 33$

Note: ϵ is the empty string

Example

Let $G = (N, T, P, S)$ where $T = \{ 1, 2, 3 \}$ and $N = \{ S, A, B \}$

- Let $P = \{ (S, A1), (S, S1), (A, 2A), (A, B), (B, 33) \}$

- $S \rightarrow A1 \mid S1$
- $A \rightarrow 2A \mid B$
- $B \rightarrow 33$

$S \Rightarrow S1$

$\Rightarrow S11$

$\Rightarrow A111$

$\Rightarrow 2A111$ *sentential*

$\Rightarrow 22A111$

$\Rightarrow 22B111$

$\Rightarrow 2233111$ *sentence*

Formal Definition of a Grammar

$$G = (N, \Sigma, P, S)$$

Formally, a grammar is defined as a four-tuple (N, Σ, P, S) where

- (1) N is a set of *nonterminal symbols*,
- (2) Σ is a set of *terminal symbols* with $N \cap \Sigma = \emptyset$,
- (3) P $\subseteq N \times (N \cup \Sigma)^*$ is a set of *productions*, and
- (4) $S \in N$ is the *start symbol*.

The language $L(G)$ defined by the grammar G .

$$L(G) = \{\sigma \in \Sigma^* \mid S \Rightarrow^* \sigma\}$$

Example

Let $G = (N, \Sigma, P, S)$ where $\Sigma = \{1, 2, 3\}$ and $N = \{S, A, B\}$

- Let $P = \{(S, A1), (S, S1), (A, 2A), (A, B), (B, 33)\}$
 - $S \rightarrow A1 \mid S1$
 - $A \rightarrow 2A \mid B$
 - $B \rightarrow 33$

$S \Rightarrow A1$ *sentential*

$\Rightarrow B1$

$\Rightarrow 331$ *sentence*

$$L(G) = 2^*3311^*$$

Language starts (0 or more 2's) middle (33) ends (1 or more 1's)

Example

Let $G' = (N, \Sigma, P, S)$ where $\Sigma = \{1, 2, 3\}$ and $N = \{S, A, B\}$

- Let $P = \{(S, ABC), (A, ?), (A, 2A), (B, 33), (C, C1), (C, 1)\}$

- $S \rightarrow ABC$
- $A \rightarrow 2A \mid ?$? production
- $B \rightarrow 33$
- $C \rightarrow C1 \mid 1$

$S \Rightarrow ABC$
 $\quad \quad \quad ?BC$
 $\Rightarrow BC$
 $\Rightarrow 33C$
 $\Rightarrow 331$

$L(G) = L(G')$

Example : Grammar G for “even” binary numbers

$L(G) = \{ \text{binary numbers the represent even integers} \}$

$$N = \{S, D\}$$

$$= \{0, 1\}$$

$$P = \{(S, DS), (S, 0), (D, 1), (D, 0)\}$$

$$P: \quad S \rightarrow DS \mid 0$$

$$D \rightarrow 0 \mid 1$$

$$S \Rightarrow DS \Rightarrow DDS \Rightarrow 0DS \Rightarrow 01S \Rightarrow 010$$

S 010

010 $\boxed{\text{?}}$ $\boxed{\text{?}}$

$\boxed{\text{?}}$

$$L(G) = \{ \dots, 010, \dots \}$$

Example : Grammar G for “even” binary numbers

$L(G) = \{ \text{binary numbers the represent even integers} \}$

$$N = \{S, D\}$$

$$= \{0, 1\}$$

$$P = \{(S, DS), (S, 0), (D, 1), (D, 0)\}$$

$$P: \quad S \rightarrow DS \mid 0$$

$$D \rightarrow 0 \mid 1$$

$$S \Rightarrow DS \Rightarrow DDS \Rightarrow 0DS \Rightarrow 01S \Rightarrow 010$$

S 010

010 ? ?

?

$$L(G) = \{ \dots, 010, \dots \}$$

Formal Definition of a Grammar

$$G = (N, \Sigma, P, S)$$

Formally, a grammar is defined as a four-tuple (N, Σ, P, S) where

- (1) N is a set of *nonterminal symbols*,
- (2) Σ is a set of *terminal symbols* with $N \cap \Sigma = \emptyset$,
- (3) P $\subseteq N \times (N \cup \Sigma)^*$ is a set of *productions*, and
- (4) $S \in N$ is the *start symbol*.

The language $L(G)$ defined by the grammar G .

$$L(G) = \{\sigma \in \Sigma^* \mid S \Rightarrow^* \sigma\}$$

Example : Grammar G for all combinations of the letters “a”, “c”, “t” that start with a “c” and end with a “t”

$$\begin{aligned}N &= \{S, M, E\} \\&= \{a, c, t\}\end{aligned}$$

$$P = \{(S, cM), (M, cM), (M, aM), (M, tM), (M, E), (E, t)\}$$

$$\begin{aligned}P: \quad S &\rightarrow cM \\M &\rightarrow aM \mid cM \mid tM \mid E \\E &\rightarrow t\end{aligned}$$

$$S \Rightarrow cM \Rightarrow caM \Rightarrow caaM \Rightarrow caaE \Rightarrow caat$$

S caat

caat ? { a, c, t } *

$$L(G) = \{ \dots, caat, \dots \}$$

Syntax – Programming languages

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
 - Users of a language definition
 - Other language designers
 - Implementers
 - Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., *, sum, begin)
- A *token* is a category of lexemes (e.g., identifier)

Formal Definition of Languages

- **Recognizers**

- A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
- Example: syntax analysis part of a compiler
 - Detailed discussion of syntax analysis appears in Chapter 4

- **Generators**

- A device that generates sentences of a language
- One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

BNF and Context-Free Grammars

- Context-Free Grammars
 - Developed by Noam Chomsky in the mid-1950s
 - Language generators, meant to describe the syntax of natural languages
 - Define a class of languages called context-free languages
- Backus-Naur Form (1959)
 - Invented by John Backus to describe the syntax of Algol 58
 - BNF is equivalent to context-free grammars

BNF Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures--they act like **syntactic variables** (also called *nonterminal symbols*, or just *terminals*)
- *Terminals* are lexemes or tokens
- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

BNF Fundamentals (continued)

- Nonterminals are often enclosed in angle brackets

- Examples of BNF rules:

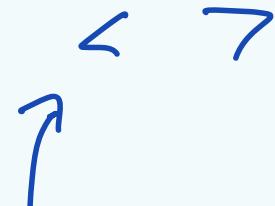
```
<ident_list> → identifier | identifier, <ident_list>  
<if_stmt> → if <logic_expr> then <stmt>
```

- Grammar: a finite non-empty set of rules
- A *start symbol* is a special element of the nonterminals of a grammar

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

<stmt>  ? <single_stmt>
| begin <stmt_list> end



Describing Lists

- Syntactic lists are described using recursion

$\text{<ident_list>} \xrightarrow{\quad} \text{ident} \rightarrow \text{terminal}$

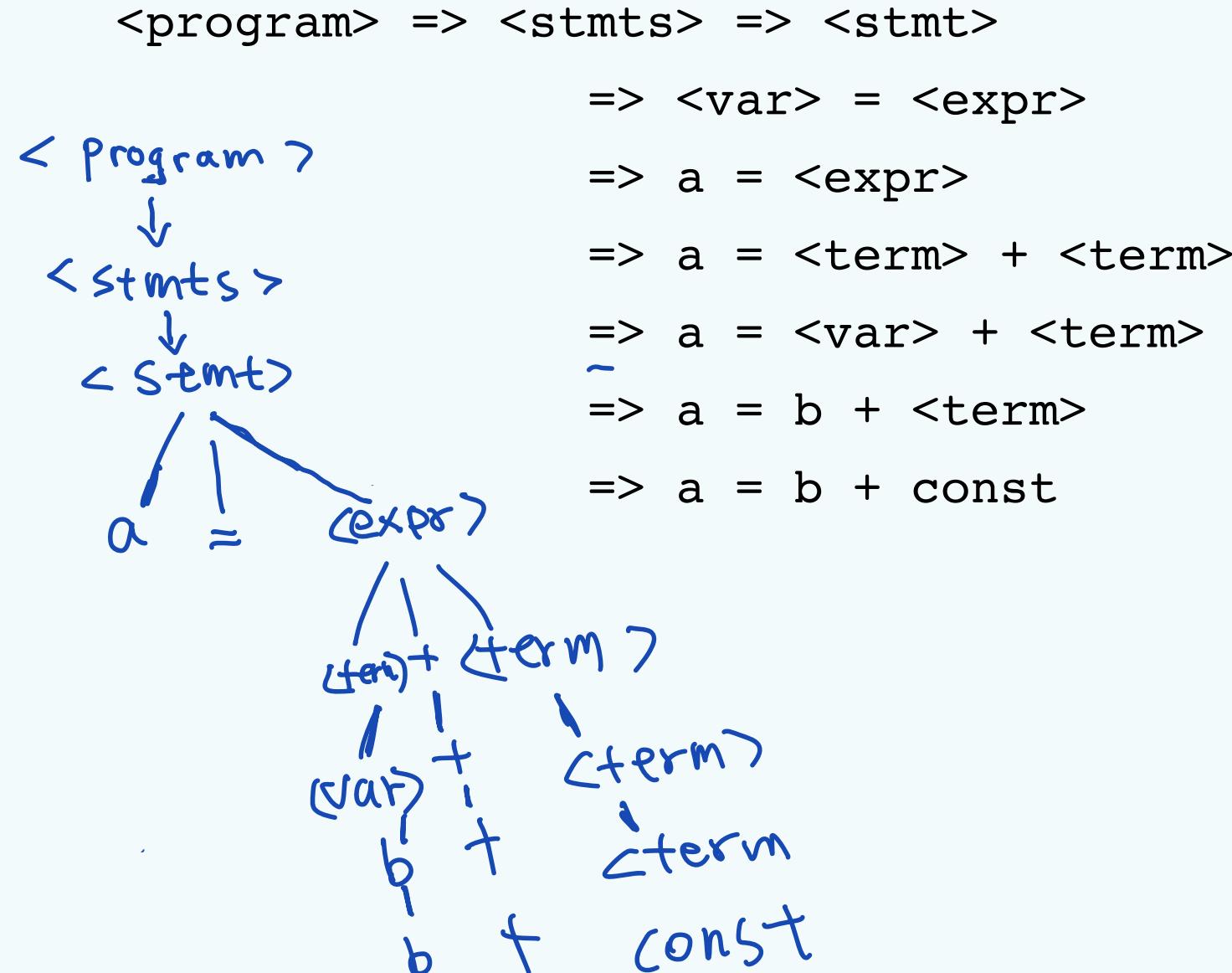
| ident, <ident_list>
 Non-terminal

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar

```
<program> → <stmts>  
<stmts> → <stmt> | <stmt> ; <stmts>  
<stmt> → <var> = <expr>  
<var> → a | b | c | d  
<expr> → <term> + <term> | <term> - <term>  
<term> → <var> | const
```

An Example Derivation

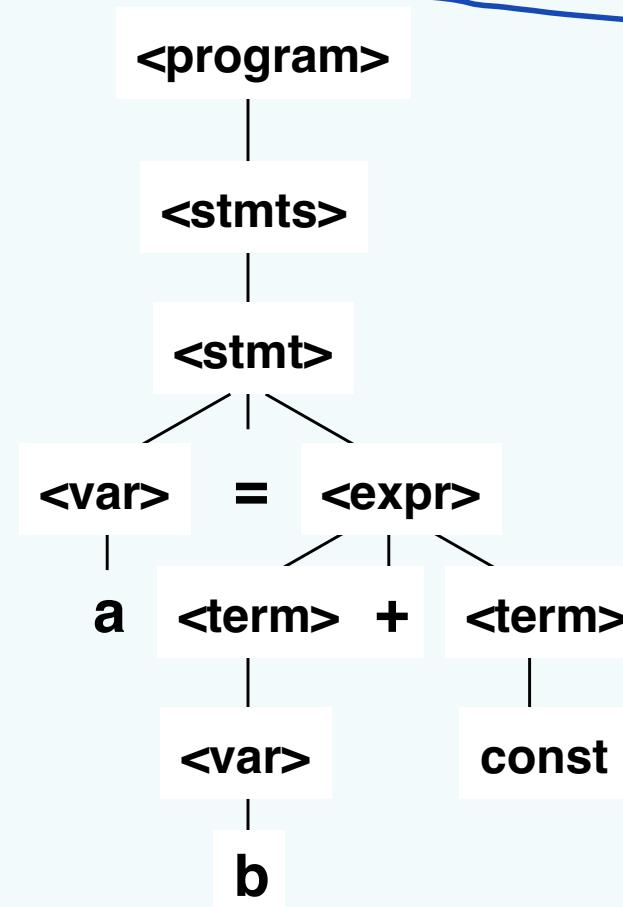


Derivations

- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Parse Tree

- A hierarchical representation of a derivation



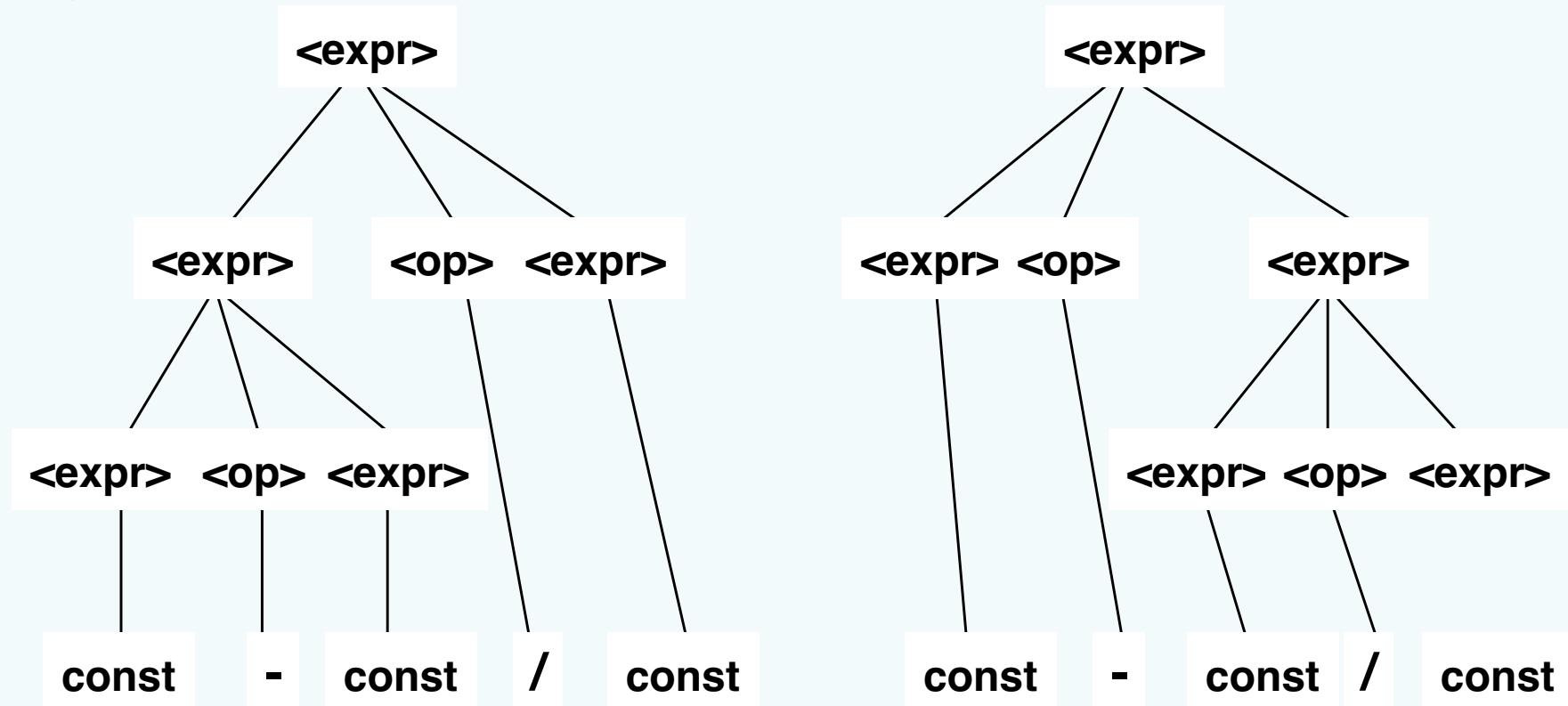
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

$\text{<expr>} \rightarrow \text{<expr>} \text{ <op>} \text{ <expr>} \quad | \quad \text{const}$

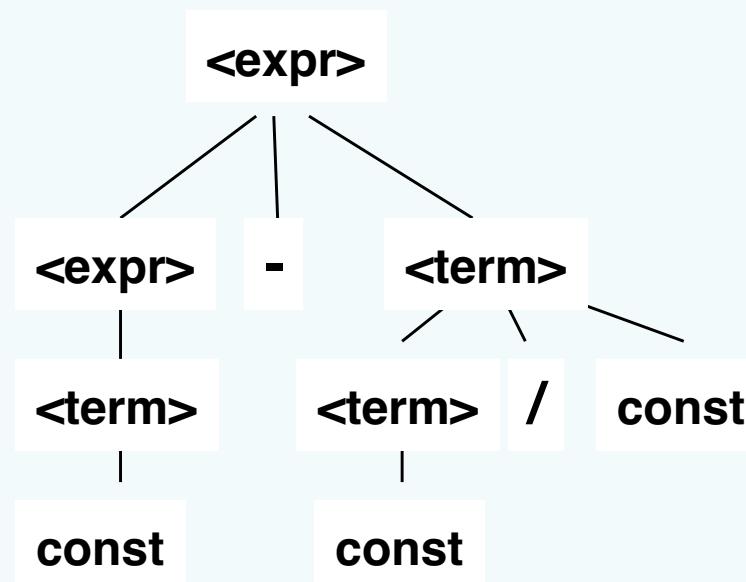
$\text{<op>} \rightarrow / \quad | \quad -$



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle - \langle \text{term} \rangle \quad | \quad \langle \text{term} \rangle$
 $\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle / \text{const} \quad | \quad \text{const}$

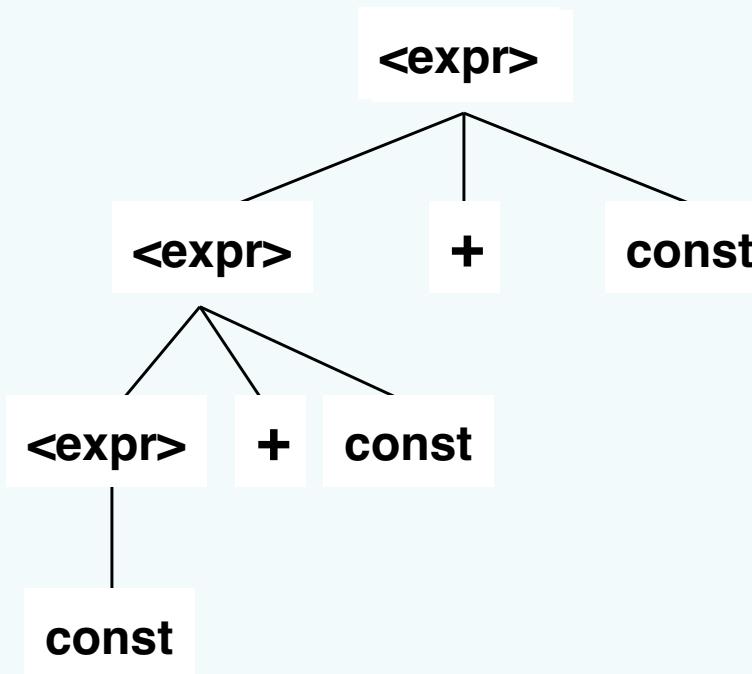


Associativity of Operators

- Operator associativity can also be indicated by a grammar

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \text{const}$ (ambiguous)

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \text{const} \mid \text{const}$ (unambiguous)



Extended BNF

- Optional parts are placed in brackets []
$$\langle \text{proc_call} \rangle \rightarrow \text{ident} [(\langle \text{expr_list} \rangle)]$$
- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars
$$\langle \text{term} \rangle \rightarrow \langle \text{term} \rangle (+ | -) \text{ const}$$
- Repetitions (0 or more) are placed inside braces {}
$$\langle \text{ident} \rangle \rightarrow \text{letter} \{ \text{letter} | \text{digit} \}$$

BNF and EBNF

- BNF

```
<expr> ? <expr> + <term>
      | <expr> - <term>
      | <term>

<term> ? <term> * <factor>
      | <term> / <factor>
      | <factor>
```

- EBNF

```
<expr> ? <term> { (+ | -) <term> }
<term> ? <factor> { (* | /) <factor> }
```

HW 2

- Submit solutions in `homework2.hs` file without a main function.
- Comment code
 - Header – name, class, date
 - Comments before solutions to each question
- You can use `verifier.hs` to test your functions.
 - Place the main after your function definitions.
 - start `ghci`, load your file, call `main`
 - Compare your results to my `verifier.txt` file

Syntax

- Grammars & Derivations
 - Natural Languages
 - Formal Languages
 - Programming Languages
- Syntax Tree / Parse Tree
- Abstract vs Concrete Syntax
- Representing Grammars by Haskell DataTypes

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*).

LHS A ::= B C ... *RHS*

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

How are sentences generated by rules?

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

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

Example Grammar

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

nonterminal symbols

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

terminal symbols

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

Derivation

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

sentence

noun verb noun

dogs verb noun

dogs have noun

dogs have teeth

apply rule (R1)

apply rule (R2)

apply rule (R4)

apply rule (R3)

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

Derivation Order

The order of rule application is *not* fixed.

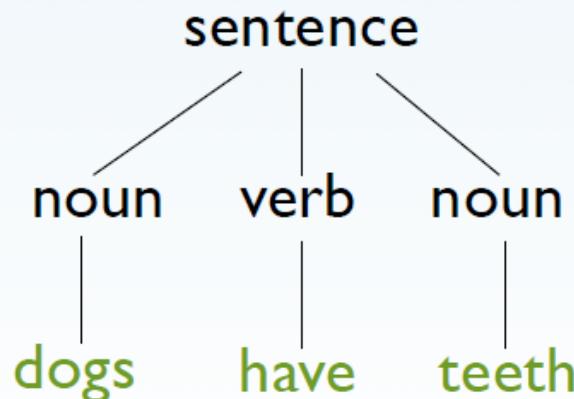
```
sentence ::= noun verb noun (R1)
noun      ::= dogs          (R2)
noun      ::= teeth         (R3)
verb      ::= have         (R4)
```

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)

Syntax Tree

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

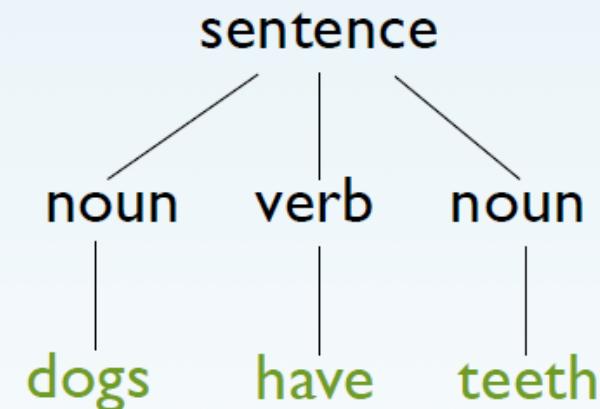


Derivation is a process of producing a sentence according to the rules of a grammar.

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

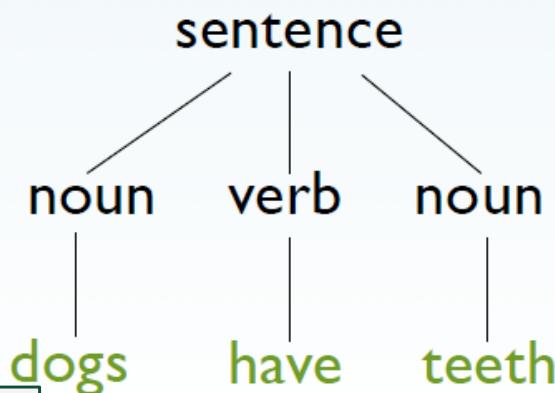
Observations About Syntax Trees

- (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



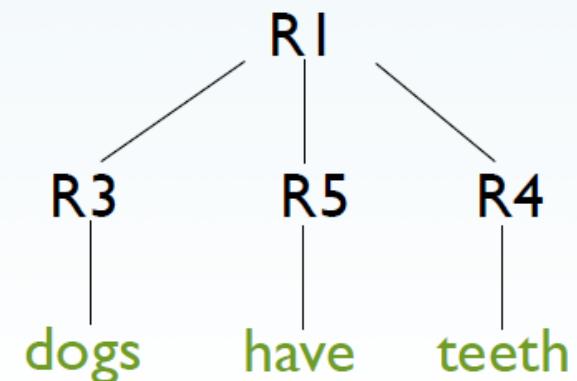
Alternative Representation

- (1) Leaves contain *terminal symbols*
(2) Internal nodes contain
nonterminal symbols

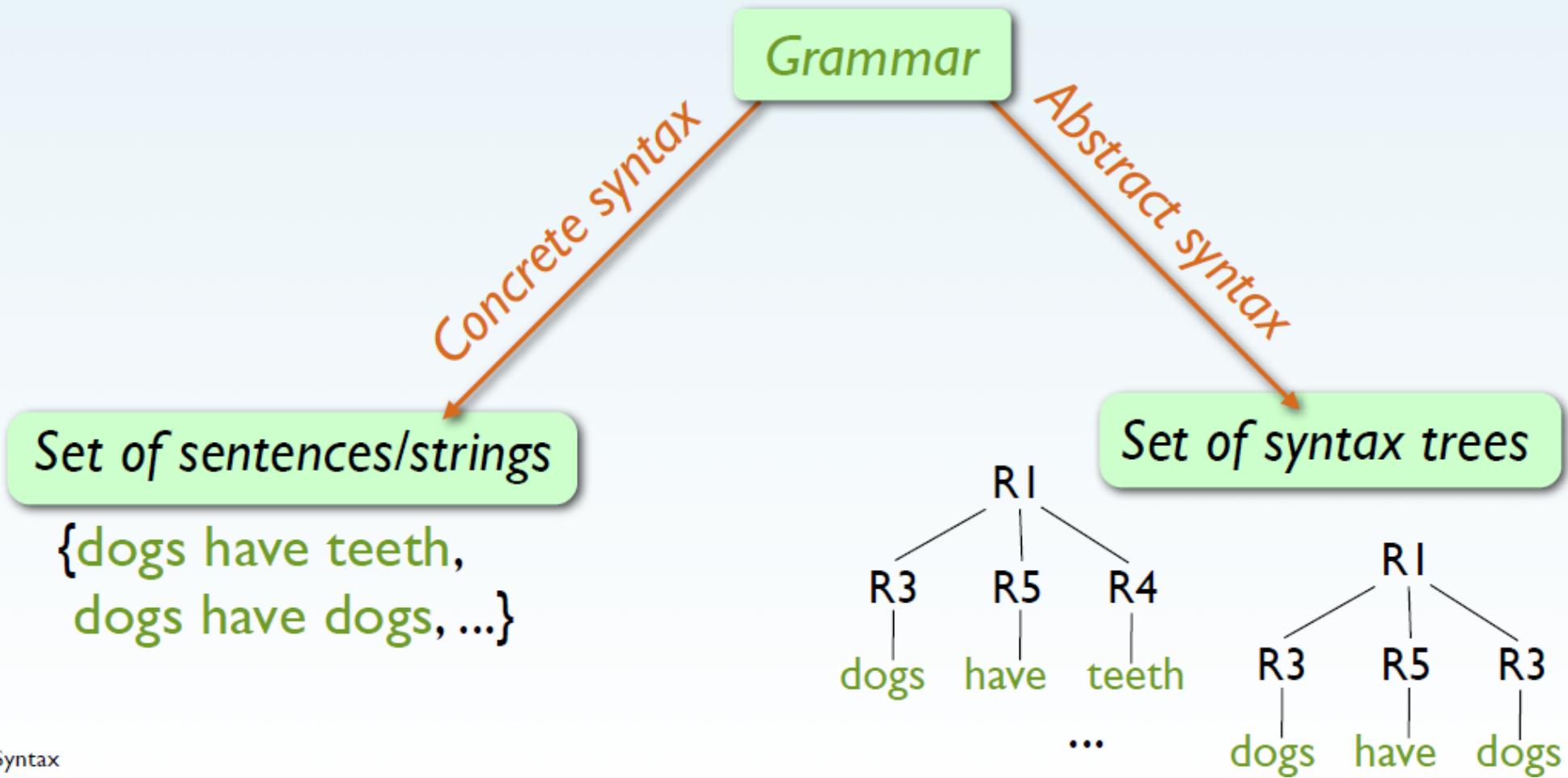


```
sentence ::= noun verb noun (R1)
sentence ::= sentence and sentence (R2)
noun     ::= dogs (R3)
noun     ::= teeth (R4)
verb     ::= have (R5)
```

- (1) Leaves contain *terminal symbols*
(2) Internal nodes contain *rule names*



Concrete vs. Abstract Syntax



(I) 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 and sentence (R1)

sentence ::= noun verb noun (R2)

noun ::= dogs (R3)

noun ::= teeth (R4)

noun ::= cats (R5)

noun ::= claws (R6)

verb ::= have (R7)

sentence

cats have claws and dogs have teeth

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.

(I) 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 and sentence (R1)

sentence :: noun verb noun (R2)

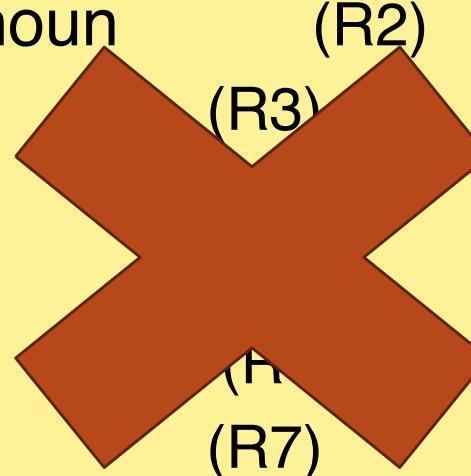
noun :: dogs

noun :: teeth

noun :: cats

noun :: claws

verb :: have



What if I wanted ??????
cats and dogs have teeth

cats and dogs have teeth

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

sentence ::= nounphrase verb noun (R1)

nounphrase ::= noun (R2a)

nounphrase ::= noun and noun (R2b)

noun ::= dogs (R3)

noun ::= teeth (R4)

noun ::= cats (R5)

noun ::= claws (R6)

verb ::= have (R7)

sentence	$::=$	nounphrase verb noun	(R1)
nounphrase	$::=$	noun	(R2a)
nounphrase	$::=$	noun and noun	(R2b)
noun	$::=$	dogs	(R3)
noun	$::=$	teeth	(R4)
noun	$::=$	cats	(R5)
noun	$::=$	claws	(R6)
verb	$::=$	have	(R7)

cats and dogs have teeth

sentence \Rightarrow nounphrase verb noun
 \Rightarrow noun and noun verb noun
 \Rightarrow cats and noun verb noun

 \Rightarrow cats and dogs have teeth

cats and dogs have teeth and claws

sentence ::= nounphrase verb nounphrase (R1)

nounphrase ::= noun (R2a)

nounphrase ::= noun and noun (R2b)

noun ::= dogs (R3)

noun ::= teeth (R4)

noun ::= cats (R5)

noun ::= claws (R6)

verb ::= have (R7)

sentence	::= nounphrase verb noun	(R1)
nounphrase	::= noun	(R2a)
nounphrase	::= multinoun and noun	(R2b)
multinoun	::= multinoun noun	(R2c)
multinoun	::= noun	(R2d)
noun	::= dogs cats bears	(R3)
noun	::= teeth claws	(R4)
verb	::= have	(R7)

cats bears and dogs have teeth

sentence \Rightarrow nounphrase verb noun
 \Rightarrow multinoun and noun verb noun
 \Rightarrow multinoun noun and noun verb noun
 \Rightarrow noun noun and noun verb noun
....
 \Rightarrow cats bears and dogs have teeth

Formal Definition of a Context-Free Grammar

$$G = (N, \Sigma, P, S)$$

Formally, a grammar is defined as a four-tuple (N, Σ, P, S) where

- (1) N is a set of *nonterminal symbols*,
- (2) Σ is a set of *terminal symbols* with $N \cap \Sigma = \emptyset$,
- (3) $P \subseteq N \times (N \cup \Sigma)^*$ is a set of *productions*, and
- (4) $S \in N$ is the *start symbol*.

Definitions:

- A sequence of terminals is a *sentence*. A *sentence* \vdash
- An intermediate sequence in the derivation that contains nonterminals is a *sentential form*. A *sentential form* $\vdash (N \vdash$
- A production (A, B) can also be written as
 - $A \rightarrow B$ where $A \in N$ and $B \in (N \vdash$

Example

Let $G = (N, \Sigma, P, S)$ where $\Sigma = \{ 1, 2, 3 \}$ and $N = \{ S, A, B \}$

- $\Sigma^* = \{ \epsilon, S, A, B, 1, 1A, 2A, B3, A1B1, 12, 13, 21, \dots \}$
- $(N \cup \Sigma)^* = \{ S, A, B, 1, 1A, 2A, B3, A1B1, 12, 13, 21, \dots \}$
- $N \times (N \cup \Sigma)^* = \{ (S, A), (A, 111), (S, A1), (S, S1), (A, 2A), (B, 3), (A, 2B) \dots \}$
- Let $P = \{ (S, A1), (S, S1), (A, 2A), (A, B), (B, 33) \}$
 - $S \rightarrow A1 \mid S1$
 - $A \rightarrow 2A \mid B$
 - $B \rightarrow 33$

Note: ϵ is the empty string

Example

Let $G = (N, T, P, S)$ where $T = \{ 1, 2, 3 \}$ and $N = \{ S, A, B \}$

- Let $P = \{ (S, A1), (S, S1), (A, 2A), (A, B), (B, 33) \}$

- $S \rightarrow A1 \mid S1$
- $A \rightarrow 2A \mid B$
- $B \rightarrow 33$

$S \Rightarrow S1$

$\Rightarrow S11$

$\Rightarrow A111$

$\Rightarrow 2A111$ *sentential*

$\Rightarrow 22A111$

$\Rightarrow 22B111$

$\Rightarrow 2233111$ *sentence*

Formal Definition of a Grammar

$$G = (N, \Sigma, P, S)$$

Formally, a grammar is defined as a four-tuple (N, Σ, P, S) where

- (1) N is a set of *nonterminal symbols*,
- (2) Σ is a set of *terminal symbols* with $N \cap \Sigma = \emptyset$,
- (3) $P \subset N \times (N \cup \Sigma)^*$ is a set of *productions*, and
- (4) $S \in N$ is the *start symbol*.

The language $L(G)$ defined by the grammar G .

$$L(G) = \{\sigma \in \Sigma^* \mid S \Rightarrow^* \sigma\}$$

Example

Let $G = (N, \Sigma, P, S)$ where $\Sigma = \{1, 2, 3\}$ and $N = \{S, A, B\}$

- Let $P = \{(S, A1), (S, S1), (A, 2A), (A, B), (B, 33)\}$
 - $S \rightarrow A1 \mid S1$
 - $A \rightarrow 2A \mid B$
 - $B \rightarrow 33$

$S \Rightarrow A1$ *sentential*

$\Rightarrow B1$

$\Rightarrow 331$ *sentence*

$$L(G) = 2^*3311^*$$

Language starts (0 or more 2's) middle (33) ends (1 or more 1's)

Example

Let $G' = (N, \Sigma, P, S)$ where $\Sigma = \{1, 2, 3\}$ and $N = \{S, A, B\}$

- Let $P = \{(S, ABC), (A, ?), (A, 2A), (B, 33), (C, C1), (C, 1)\}$

- $S \rightarrow ABC$
- $A \rightarrow 2A \mid ?$? production
- $B \rightarrow 33$
- $C \rightarrow C1 \mid 1$

$S \Rightarrow ABC$
 $\quad \quad \quad ?BC$
 $\Rightarrow BC$
 $\Rightarrow 33C$
 $\Rightarrow 331$

$L(G) = L(G')$

Example : Grammar G for “even” binary numbers

$L(G) = \{ \text{binary numbers the represent even integers} \}$

$$N = \{S, D\}$$

$$= \{0, 1\}$$

$$P = \{(S, DS), (S, 0), (D, 1), (D, 0)\}$$

$$P: \quad S \rightarrow DS \mid 0$$

$$D \rightarrow 0 \mid 1$$

$$S \Rightarrow DS \Rightarrow DDS \Rightarrow 0DS \Rightarrow 01S \Rightarrow 010$$

S 010

010 $\boxed{\text{?}}$ $\boxed{\text{?}}$

$\boxed{\text{?}}$

$$L(G) = \{ \dots, 010, \dots \}$$

Example : Grammar G for “even” binary numbers

$L(G) = \{ \text{binary numbers the represent even integers} \}$

$$N = \{S, D\}$$

$$= \{0, 1\}$$

$$P = \{(S, DS), (S, 0), (D, 1), (D, 0)\}$$

$$P: \quad S \rightarrow DS \mid 0$$

$$D \rightarrow 0 \mid 1$$

$$S \Rightarrow DS \Rightarrow DDS \Rightarrow 0DS \Rightarrow 01S \Rightarrow 010$$

S 010

010 $\boxed{\text{?}}$ $\boxed{\text{?}}$

$\boxed{\text{?}}$

$$L(G) = \{ \dots, 010, \dots \}$$

Formal Definition of a Grammar

$$G = (N, \Sigma, P, S)$$

Formally, a grammar is defined as a four-tuple (N, Σ, P, S) where

- (1) N is a set of *nonterminal symbols*,
- (2) Σ is a set of *terminal symbols* with $N \cap \Sigma = \emptyset$,
- (3) P $\subseteq N \times (N \cup \Sigma)^*$ is a set of *productions*, and
- (4) $S \in N$ is the *start symbol*.

The language $L(G)$ defined by the grammar G .

$$L(G) = \{\sigma \in \Sigma^* \mid S \Rightarrow^* \sigma\}$$

Example : Grammar G for all combinations of the letters “a”, “c”, “t” that start with a “c” and end with a “t”

$$\begin{aligned}N &= \{S, M, E\} \\&= \{a, c, t\}\end{aligned}$$

$$P = \{(S, cM), (M, cM), (M, aM), (M, tM), (M, E), (E, t)\}$$

$$\begin{aligned}P: \quad S &\rightarrow cM \\M &\rightarrow aM \mid cM \mid tM \mid E \\E &\rightarrow t\end{aligned}$$

$$S \Rightarrow cM \Rightarrow caM \Rightarrow caaM \Rightarrow caaE \Rightarrow caat$$

S caat

caat ? { a, c, t } *

$$L(G) = \{ \dots, caat, \dots \}$$

Syntax – Programming languages

- **Syntax:** the form or structure of the expressions, statements, and program units
- **Semantics:** the meaning of the expressions, statements, and program units
- Syntax and semantics provide a language's definition
 - Users of a language definition
 - Other language designers
 - Implementers
 - Programmers (the users of the language)

The General Problem of Describing Syntax: Terminology

- A *sentence* is a string of characters over some alphabet
- A *language* is a set of sentences
- A *lexeme* is the lowest level syntactic unit of a language (e.g., `*`, `sum`, `begin`). Described by regular expressions.
- A *token* is a category of lexemes (e.g., identifier)

```
index = 2 * count + 17;
```

<i>Lexemes</i>	<i>Tokens</i>
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

Formal Definition of Languages

- **Recognizers**

- A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
- Syntax analysis part of a compiler
- In formal languages Pushdown Automata (PDA) or Finite Automata

- **Generators**

- A device that generates sentences of a language
- One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator
- Context-free Grammar generates context-free languages
- Regular grammar or regular expression can generate regular languages

BNF and Context-Free Grammars

- Context-Free Grammars
 - Developed by Noam Chomsky in the mid-1950s
 - Language generators, meant to describe the syntax of natural languages
 - Define a class of languages called context-free languages
- Backus-Naur Form (1959)
 - Invented by John Backus to describe the syntax of Algol 58
 - BNF is equivalent to context-free grammars

BNF Fundamentals

- BNF is a metalanguage
- In BNF, abstractions are used to represent classes of syntactic structures--they act like syntactic variables (also called *nonterminal symbols*, or just *terminals*)
- *Terminals* are lexemes or tokens
- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

BNF Fundamentals (continued)

- Nonterminals are often enclosed in angle brackets
 - Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```
- Grammar: a finite non-empty set of rules
- A *start symbol* is a special element of the nonterminals of a grammar

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> ? <single_stmt>
      | begin <stmt_list> end
```

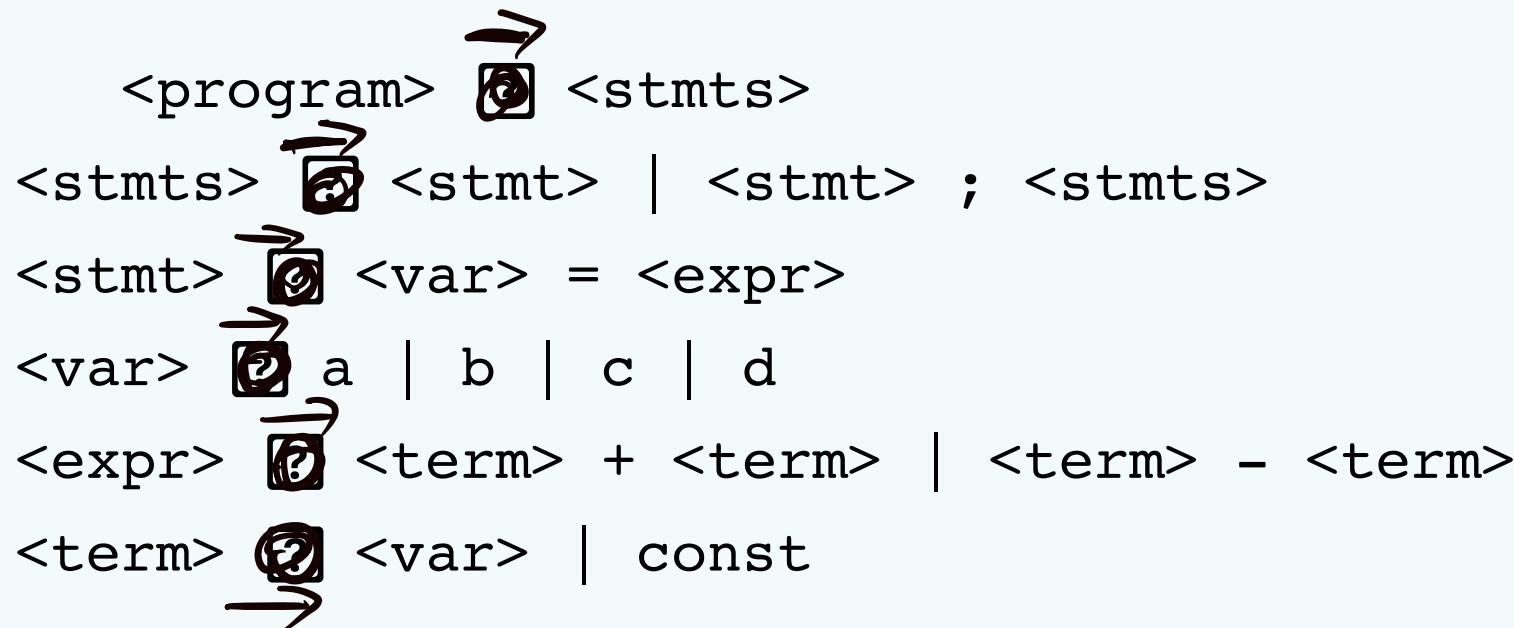
Describing Lists

- Syntactic lists are described using recursion

```
<ident_list> ::= ident  
                  | ident, <ident_list>
```

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

An Example Grammar



An Example Derivation

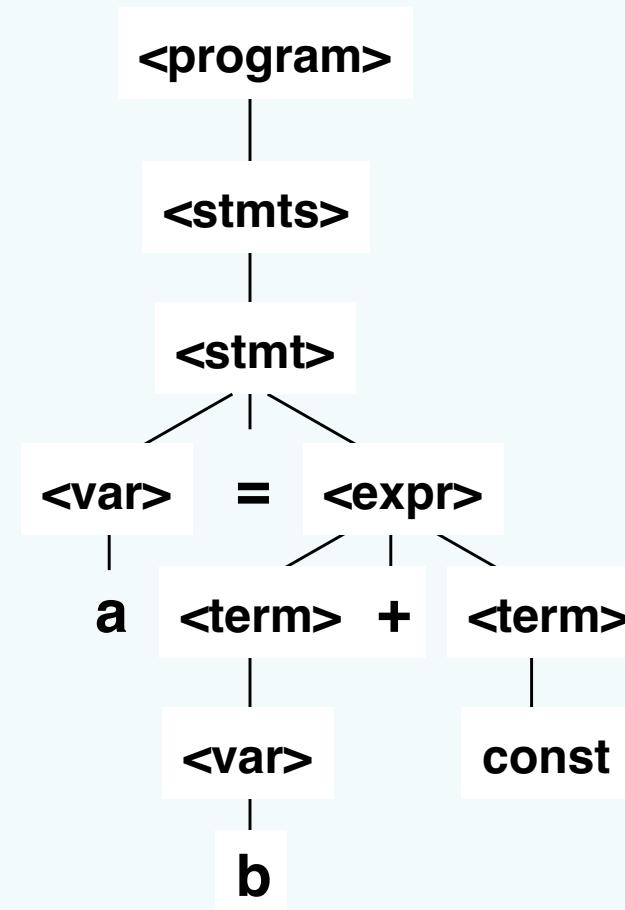
```
<program> => <stmts> => <stmt>
                  => <var> = <expr>
                  => a = <expr>
                  => a = <term> + <term>
                  => a = <var> + <term>
                  => a = b + <term>
                  => a = b + const
```

Derivations

- Every string of symbols in a derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols
- A *leftmost derivation* is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Parse Tree

- A hierarchical representation of a derivation



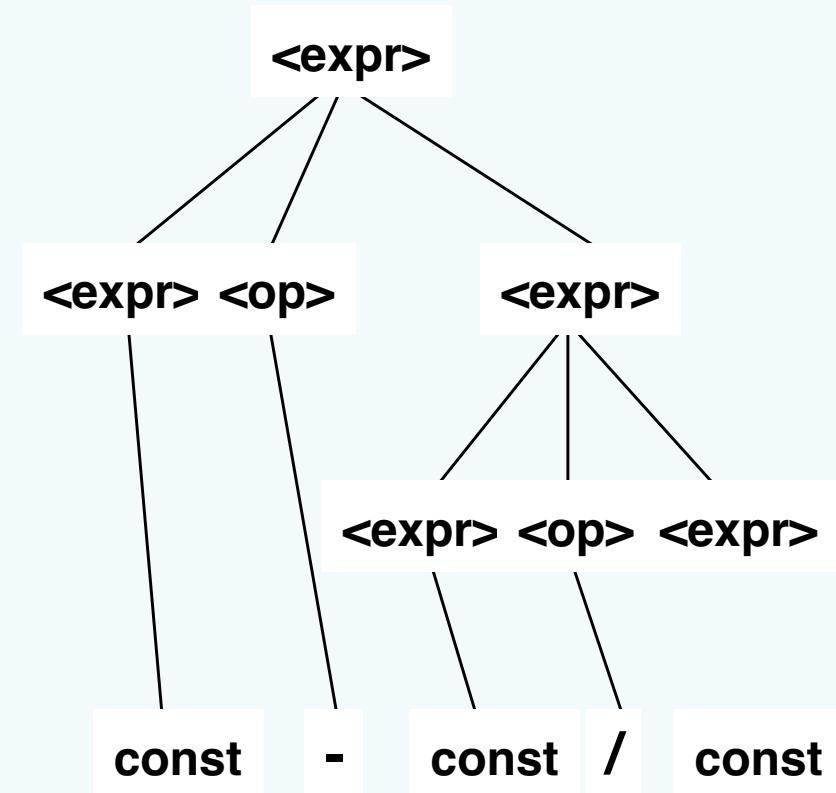
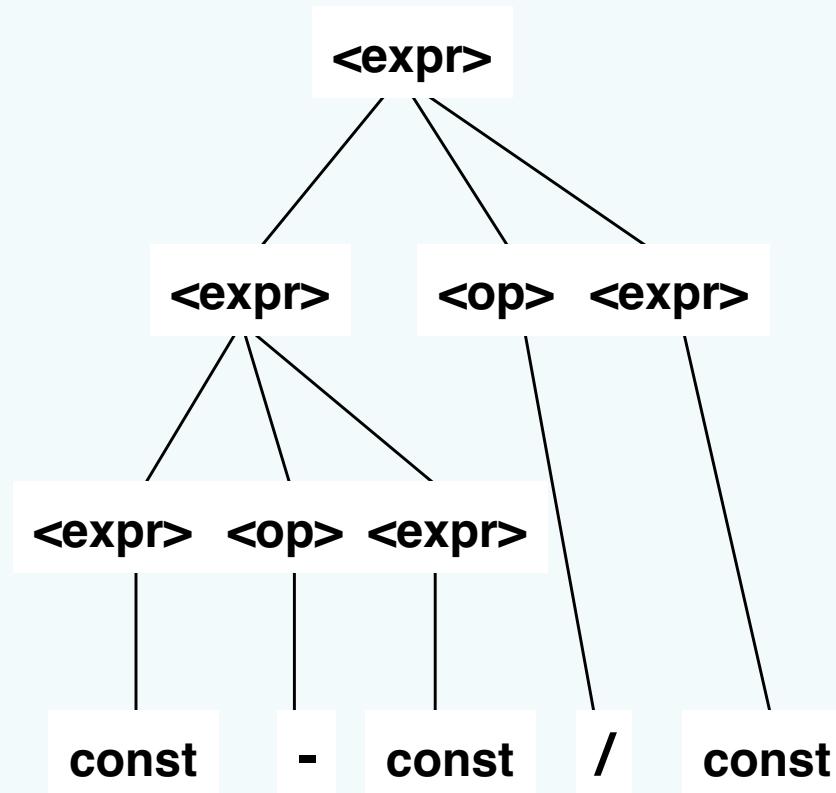
Ambiguity in Grammars

- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees

An Ambiguous Expression Grammar

$\langle \text{expr} \rangle \xrightarrow{\quad} \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \quad | \quad \text{const}$

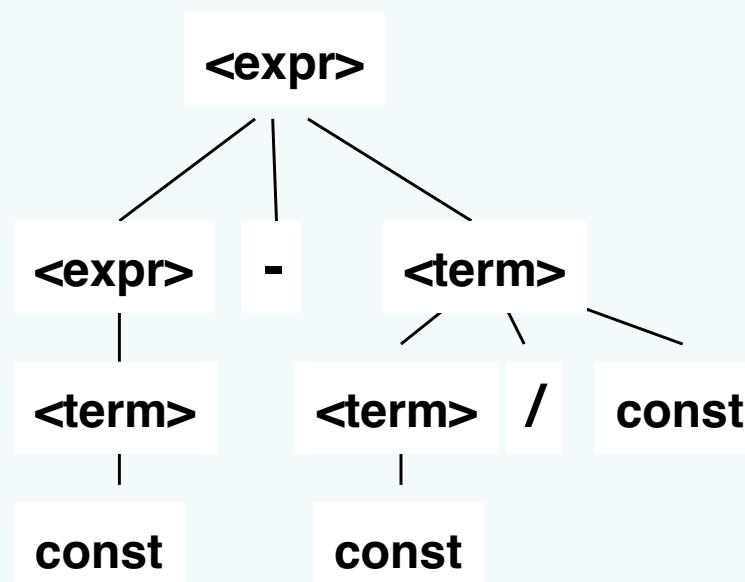
$\langle \text{op} \rangle \xrightarrow{\quad} / \quad | \quad -$



An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

$\begin{array}{l} \text{<expr>} \xrightarrow{\quad} \text{<expr>} - \text{ <term>} \quad | \quad \text{<term>} \\ \text{<term>} \xrightarrow{\quad} \text{<term>} / \text{ const} \quad | \quad \text{const} \end{array}$

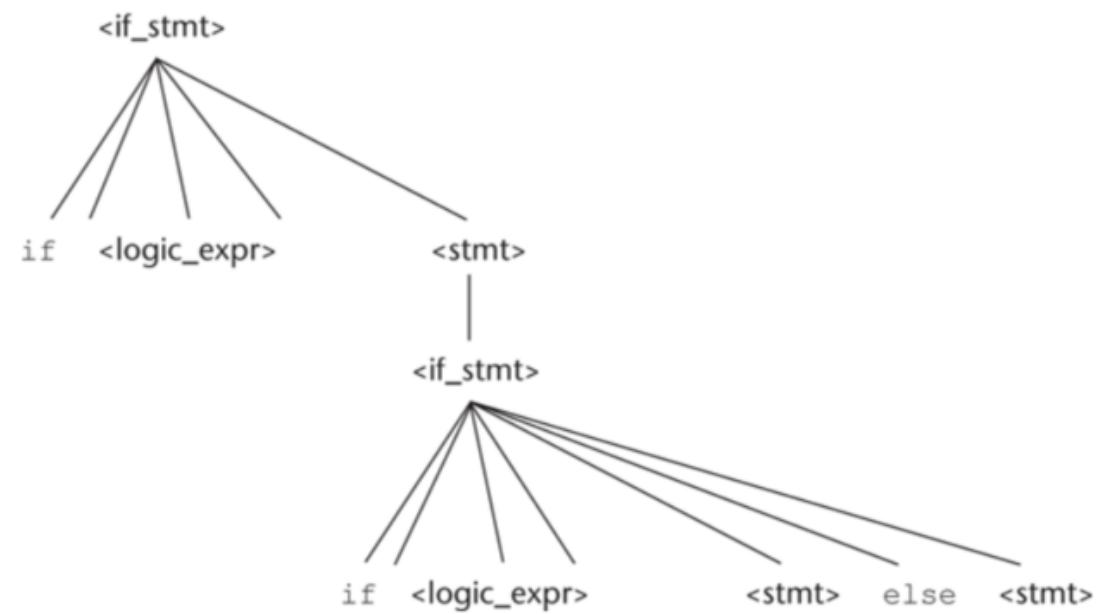
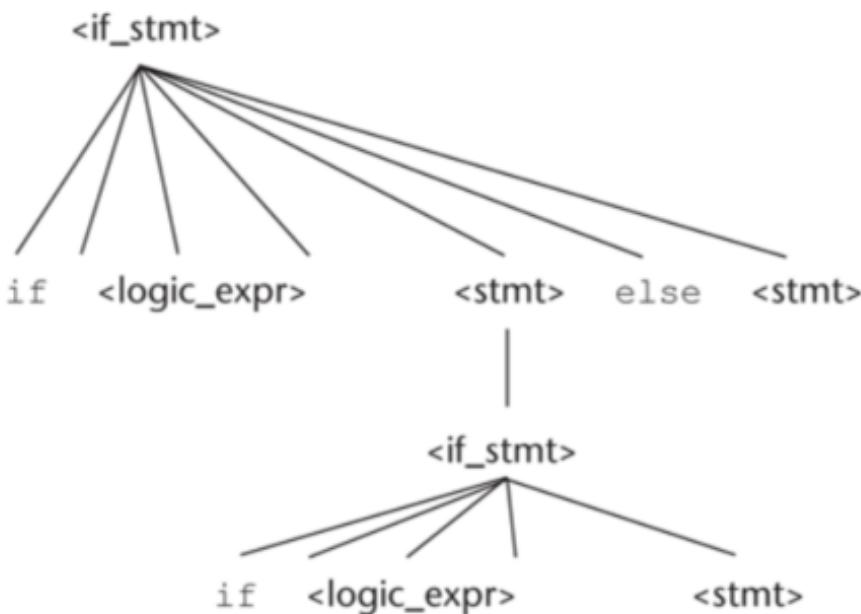


If-else Statement

$\langle \text{if_stmt} \rangle \rightarrow \text{if } (\langle \text{logic_expr} \rangle) \langle \text{stmt} \rangle \mid$

$\text{if } (\langle \text{logic_expr} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

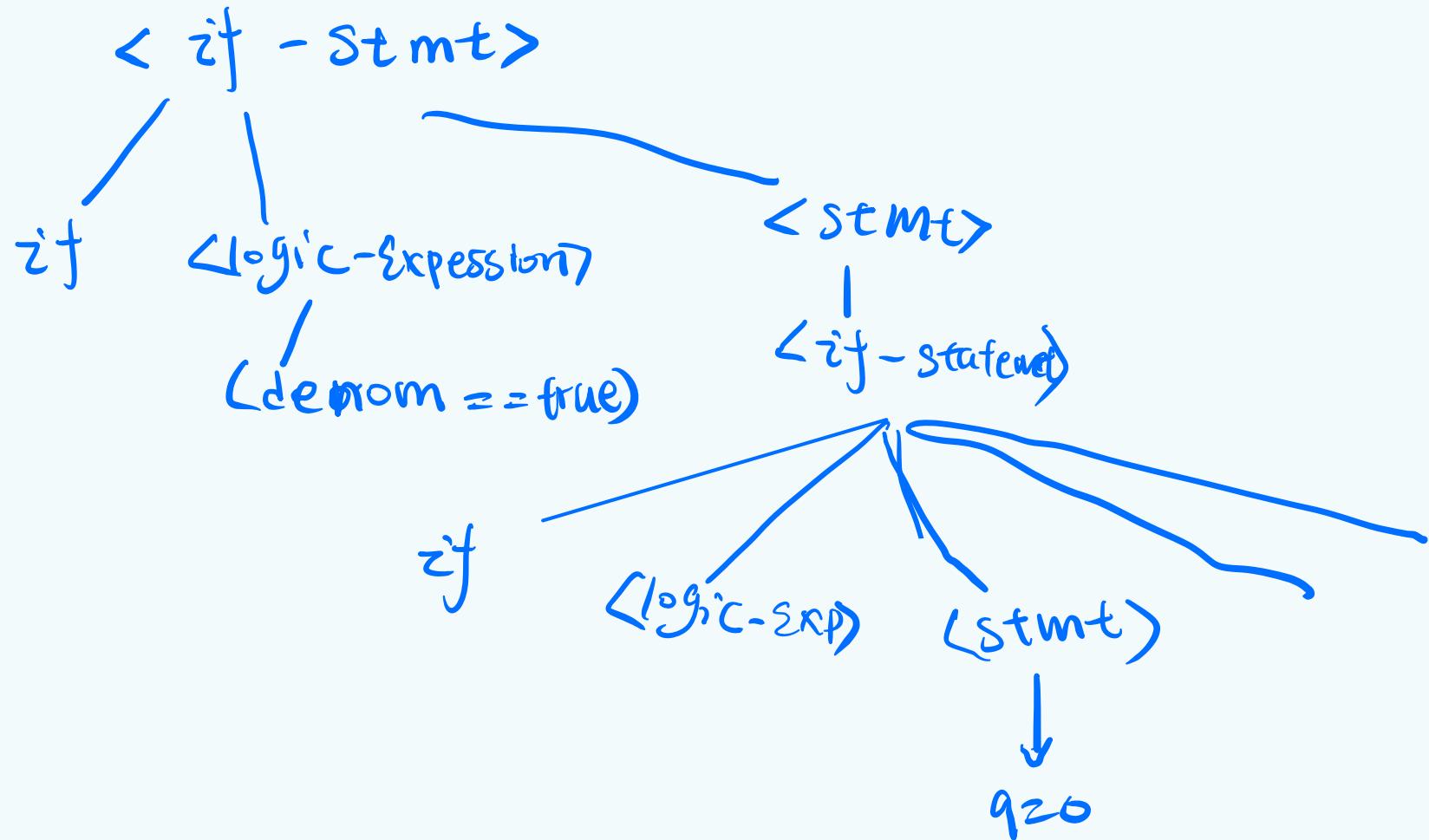
$\langle \text{stmt} \rangle \rightarrow \langle \text{if_stmt} \rangle,$



$\text{if } (\langle \text{logic_expr} \rangle) \text{ if } (\langle \text{logic_expr} \rangle) \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle$

```
if (done == true)  
  
if (denom == 0)  
  
    quotient = 0;  
  
else quotient = num / denom;
```

Draw trees



`<stmt> → <matched> | <unmatched>`

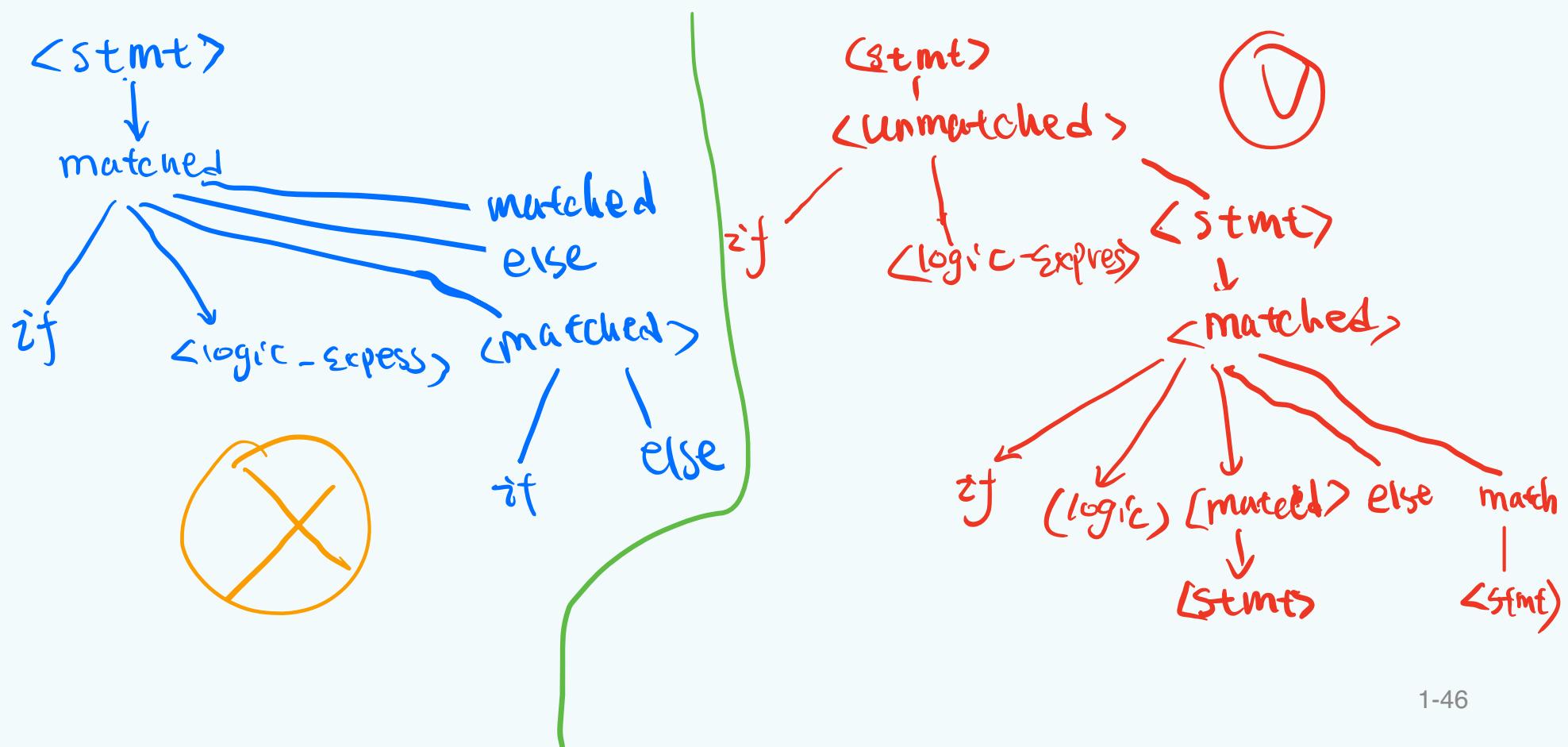
`<matched> → if (<logic_expr>) <matched> else <matched>`

| any non-if statement

`<unmatched> → if (<logic_expr>) <stmt>`

| if (<logic_expr>) <matched> else <unmatched>

→ `if (<logic_expr>) if (<logic_expr>) <stmt> else <stmt>`



```
if (done == true)  
  
if (denom == 0)  
  
    quotient = 0;  
  
else quotient = num / denom;
```

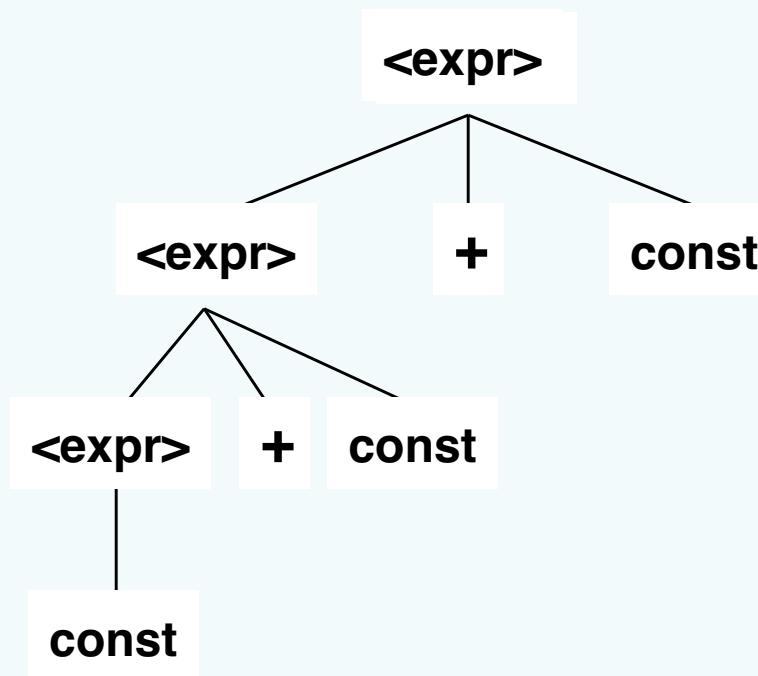
<stmt> → <matched> | <unmatched>
<matched> → if (<logic_expr>) <matched> else <matched>
| any non-if statement
<unmatched> → if (<logic_expr>) <stmt>
| if (<logic_expr>) <matched> else <unmatched>

Associativity of Operators

- Operator associativity can also be indicated by a grammar

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle \mid \text{const}$ (ambiguous)

$\langle \text{expr} \rangle \rightarrow \langle \text{expr} \rangle + \text{const} \mid \text{const}$ (unambiguous)



Extended BNF

- Optional parts are placed in brackets []

Optional

$$<\text{proc_call}> \rightarrow \text{ident} [(<\text{expr_list}>)]$$

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

$$<\text{term}> \rightarrow <\text{term}> (+ | -) \text{ const}$$

- Repetitions (0 or more) are placed inside braces { }

$$<\text{ident}> \rightarrow \text{letter} \{ \text{letter} | \text{digit} \} ^*$$

Repetitions

BNF and EBNF

- BNF

```
<expr> = <expr> + <term>
      | <expr> - <term>
      | <term>
<term> = <term> * <factor>
      | <term> / <factor>
      | <factor>
```

- EBNF

```
<expr> = {<term> { (+ | -) <term>} }
<term> = <factor> { (* | /) <factor>}  
→
```

2 Syntax

Grammars & Derivation

Syntax Tree

Abstract vs. Concrete Syntax

Representing Grammars by Haskell Data Types

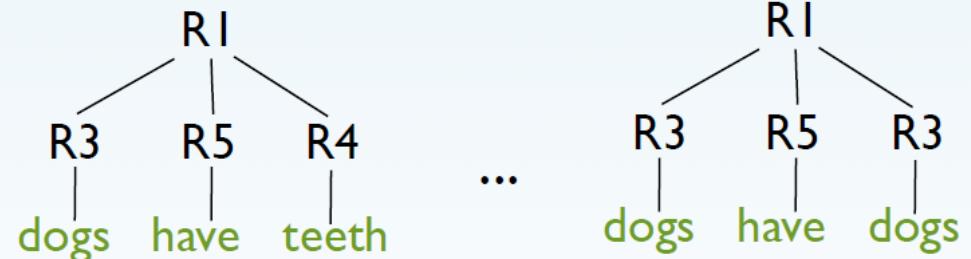
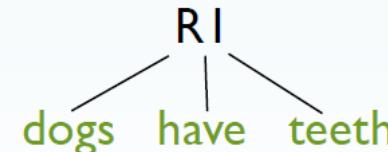
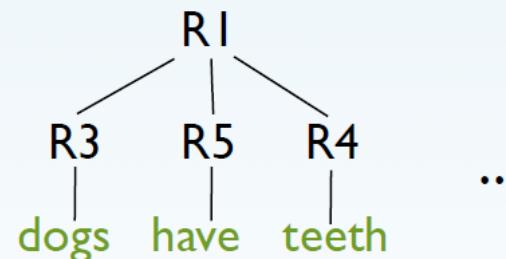
Abstract Syntax

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

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
```

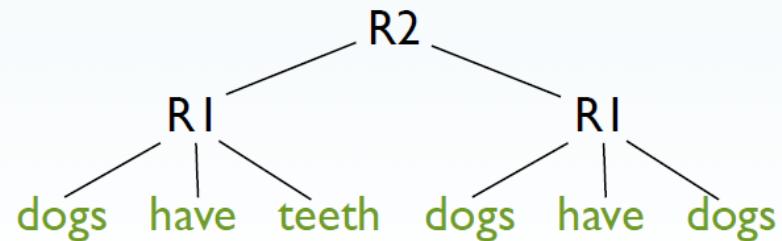
A syntax tree is represented by a Haskell value (built by data constructors)

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
```



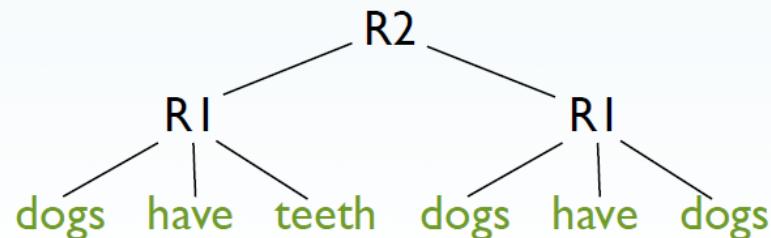
R1 Dogs Have Teeth

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

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
=

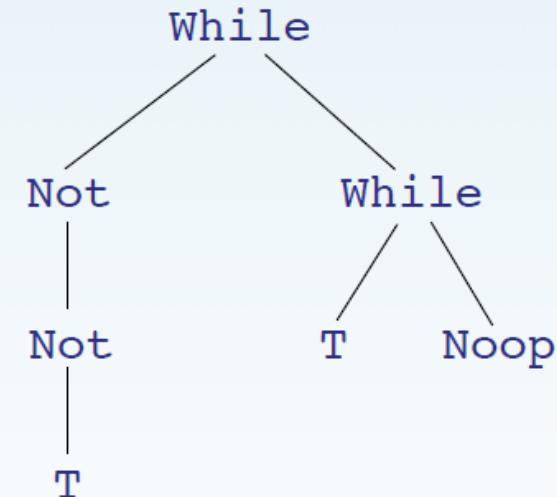
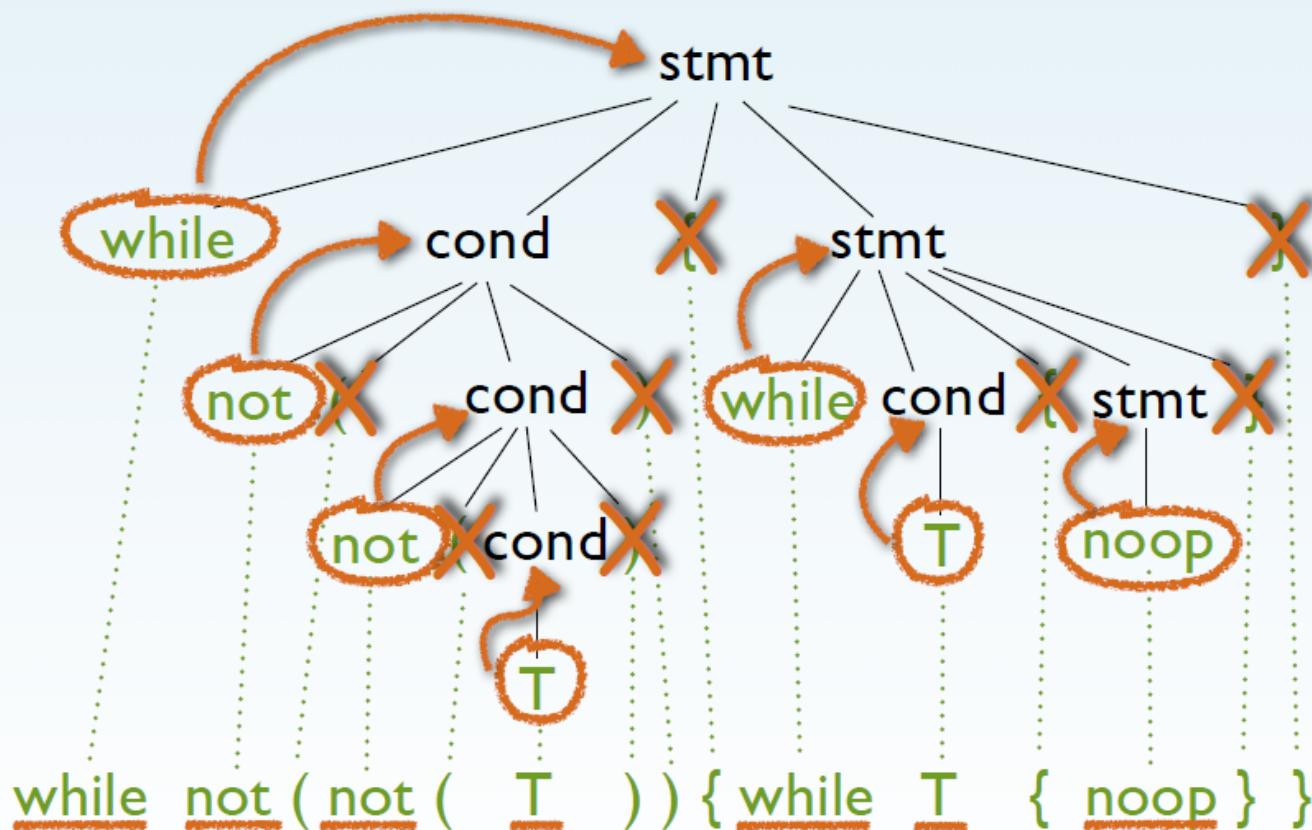
Abstract grammar

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

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

Abstract Syntax Tree

From Concrete To Abstract Syntax



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

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)
```



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

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

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

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

pretty printer

parser

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*

```
② exp ::= ① num | exp + exp | (exp)
stmt ::= ③ while exp { stmt }
       | noop
```

```
data ② Exp = N ① Int | Plus Exp Exp
data Stmt = ③ While Exp Stmt
           | Noop
```

Note Carefully!

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

Constructor is
indispensable!

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

A perfectly valid
alternative!
Plus (Exp, Exp)

→ 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).

```
type EPair = (Exp, Exp)

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

Exercise III

- (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 ?

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

Semantics

Week 4

- The **syntax** of a programming language is the form of its expressions, statements and programming units.
- Its **semantics** is the meaning of the expressions, statements and programming units.

Why Semantics?

Swap the values of two variables

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

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



Effect? y: 1
 x: 1

Effect? y: 1
 x: 8

Why Semantics? scoping

Access to non-local variables

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

Output ? 5 or 7?

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

Boolean Expression Example

- BoolSyn.hs
- BoolSem.hs

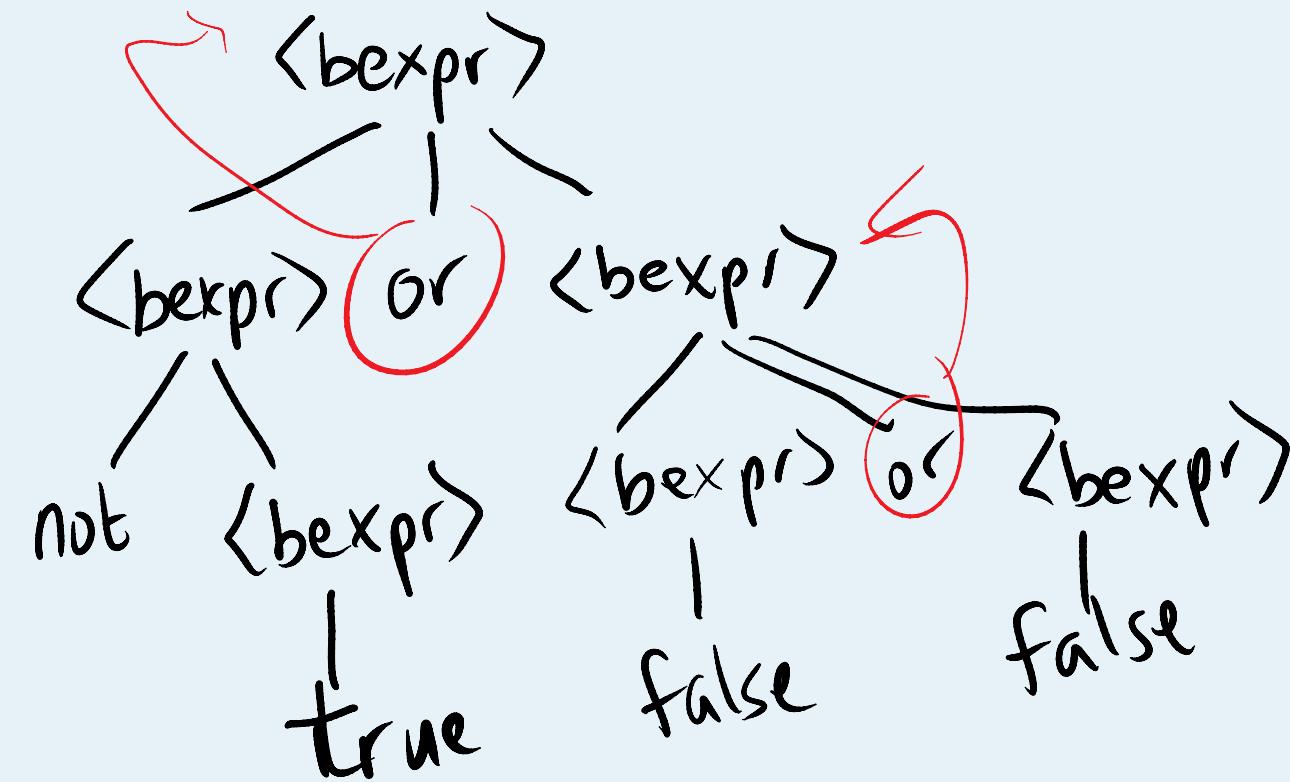
BNF grammar

<bexpr> -> not <bexpr> | <bexpr> or <bexpr>
| true | false

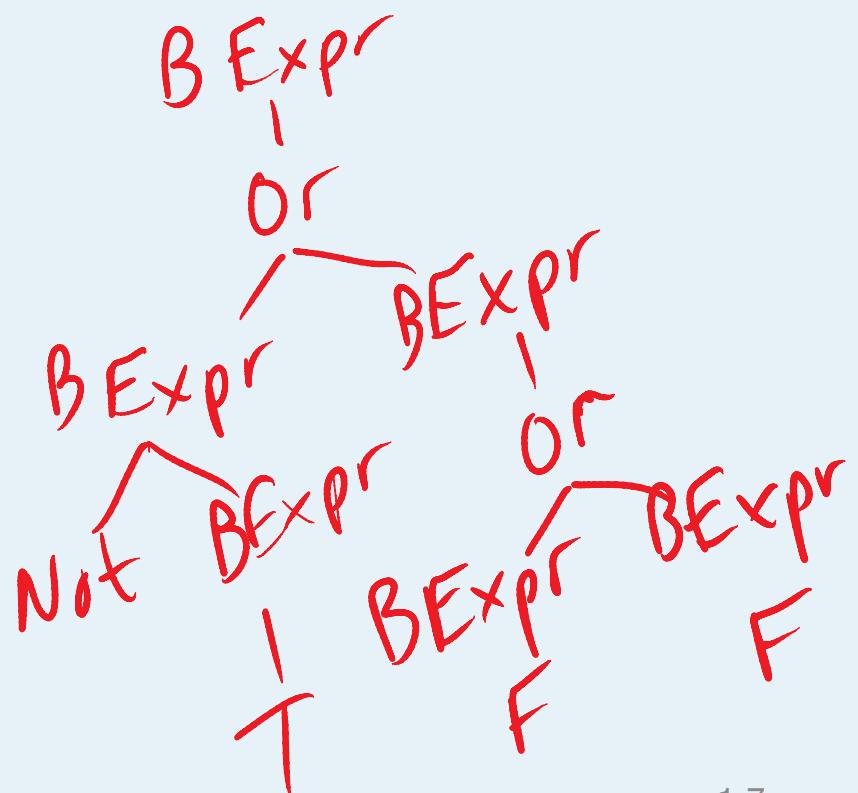
Haskell Data type

```
data BExpr = T | F  
           | Not BExpr  
           | Or BExpr BExpr  
deriving Show
```

```
sem :: BExpr -> Bool  
sem T          = True  
sem F          = False  
sem (Not b)    = not (sem b)  
sem (Or b b') = sem b || sem b'
```



Or (Not T) (Or F F)



Bool Expression

- Add pretty print and pretty evaluation

Boolean Expressions

- Add an “And” operation to the BNF, abstract syntax and semantics
- Define a Haskell function to apply DeMorgan’s laws to boolean expressions.
 - $\text{Not } (x \text{ and } y) \Rightarrow (\text{not } x) \text{ or } (\text{not } y)$
 - $\text{Not } (x \text{ or } y) \Rightarrow (\text{not } x) \text{ and } (\text{not } y)$

Simple Expression Example

- ExprSyn.hs
- ExprSem.hs

BNF grammar

```
<expr> -> <bexpr> + <bexpr> | - <bexpr>
                                | <int>
```

Haskell Data type

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

Simple Expressions

- Add multiple
- Add division
- Check divide by zero

Defining Semantics in 3 Steps

Example Language
“Arithmetic expressions”

- (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* $\llbracket \cdot \rrbracket : S \rightarrow D$
that maps trees to semantic values

$S: \text{Expr}$
 $D: \text{Int}$
 $\llbracket \cdot \rrbracket: \text{sem} :: \text{Expr} \rightarrow \text{Int}$

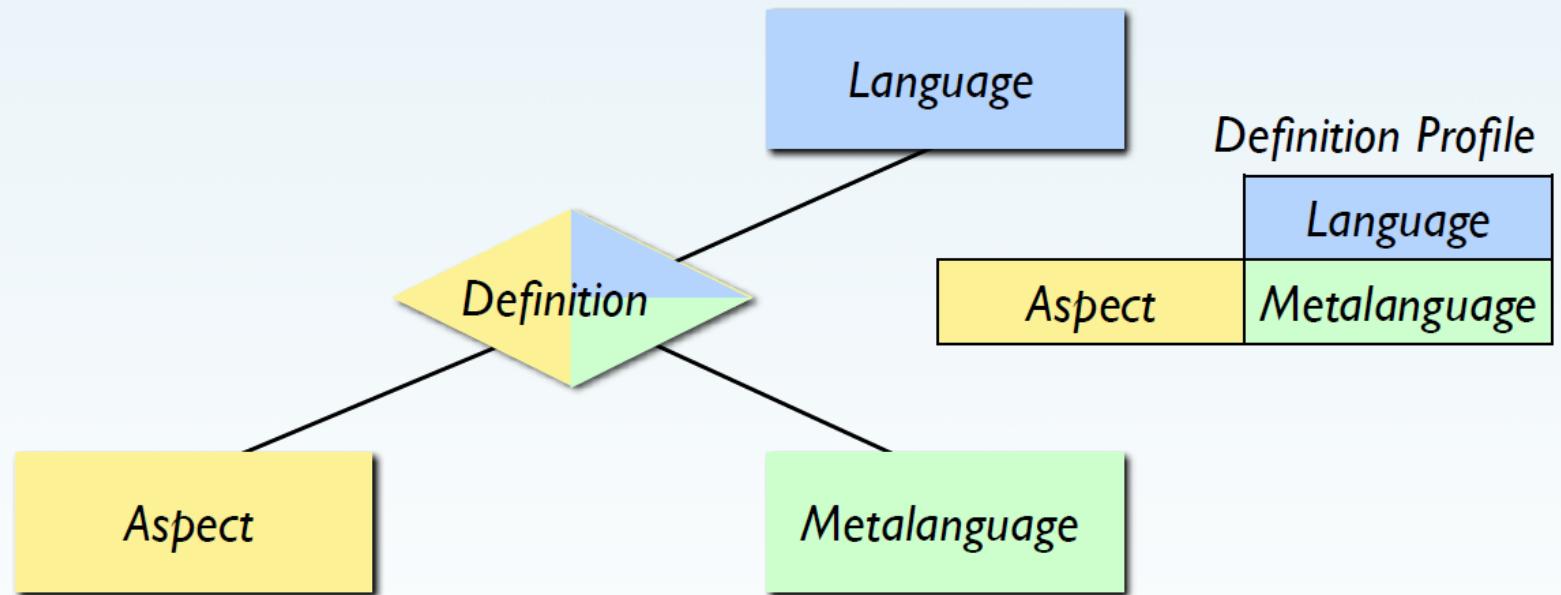
Translating Grammars Into Data Types

- (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*

```
② exp ::= ① num | exp + exp | (exp)
stmt ::= ③ while exp { stmt }
       | noop
```

```
data ② Exp = N ① Int | Plus Exp Exp
data Stmt = ③ While Exp Stmt
           | Noop
```

Language Definitions



Example Expression Language

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

	<i>Expr</i>
Syntax	Haskell

```
Expr ::= Num | Expr+Expr | -Expr
```

	<i>Expr</i>
Syntax	Grammar

Semantic Domain

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

Semantic Function

	<i>Expr</i>
Semantics	Haskell

Syntactic Symbol

```
[[·]] : Expr → Int
[[n]]   = n
[[e+e']] = [[e]] + [[e']]
[[−e]]   = −[[e]]
```

Semantic Operation

	<i>Expr</i>
Semantics	Math

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

The diagram shows the Haskell type definition for `Maybe`. It is enclosed in a yellow box. Three arrows point from green text labels to specific parts of the code:

- An arrow points from *regular value* to the `Just a` constructor.
- An arrow points from *error value* to the `Nothing` constructor.
- An arrow points from *type of regular values* to the type variable `a`.

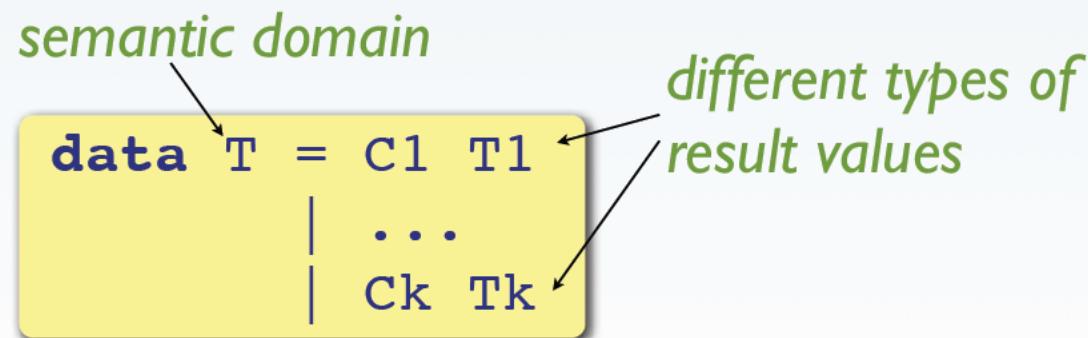
```
data Maybe a = Just a | Nothing
```

Example

ExprErr.hs

Union Domains

If $T_1 \dots T_k$ are types representing different semantic values for different nonterminals, define the semantic domain as a data type with k constructors.



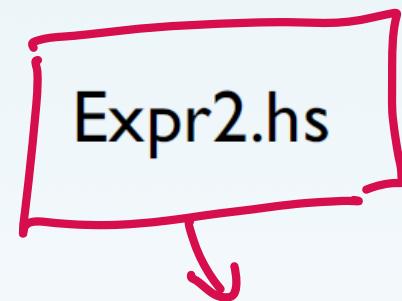
Special Case: Binary Union Domains

If T_1 and T_2 are types representing different semantic values for different nonterminals, define the semantic domain as a data type with 2 constructors.

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

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

Example



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 $T \rightarrow T$

type $D = T \rightarrow T$

sem :: $S \rightarrow D$

=

sem :: $S \rightarrow (T \rightarrow T)$

=

sem :: $S \rightarrow T \rightarrow T$

*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
```

```
data Op = LD Reg Int  
        | INC Reg  
        | DUP Reg  
  
data Reg = A | B
```

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

```
type RegCont = Int  
  
type D = RegCont -> RegCont
```

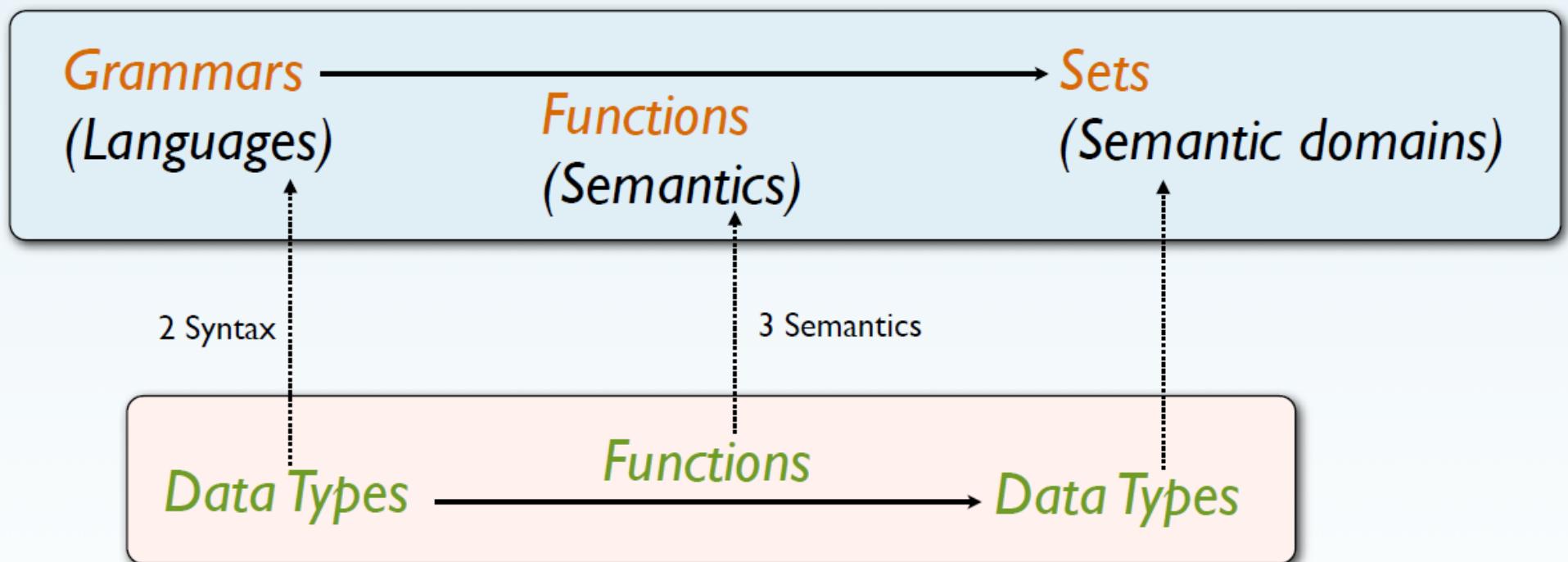
```
type RegCont = (Int, Int)
```

- (3) Define the semantics functions for the extended language

RegMachine2.hs

Haskell as a Mathematical Metalanguage

Math World



Haskell World
= Executable Math World

Types

Week 5

- Types, type Errors, Type Systems
- Why Type Checking
- Static vs Dynamic Typing
- Polymorphism
- Parametric Polymorphism

Why Types are a good thing

- Summarize a program on an abstract level
- Provide a precise documentation
- Type checkers deliver partial correctness
- Type systems can prevent runtime errors
- Type information can be exploited by compilers to generate efficient code.

JavaScript Madness

The image shows a terminal window with two panes. The left pane contains a file named 'index.js' with the following code:

```
js index.js
1 ~ if (5 > true)  {
2     x = "10";
3     console.log(x);
4 }
5 ~ else {
6     x = 17;
7     console.log(x);
8 }
9 //x = true
10 ~ if (x >= 1) {
11     x = "goodbye";
12     console.log(x);
13 }
14
15 x = x && false
16 console.log("line 16",x);
17 z = "10"
18
19 y = x/0
20 console.log(y);
21 console.log("add hello"+6);
22 console.log("times hello" * 6);
23 console.log("bool g hello"+x);
24 console.log("z hello"+z);
```

The right pane shows the output of the code execution:

```
10
goodbye
line 16 false
NaN
add hello6
NaN
bool g hellofalse
z hello10
Hint: hit control+c anytime to enter REPL.
> []
```

Type System

- Defines the basic programming constructs such as values and functions
- Provides rules of what composition of those constructs are valid and the resulting types.
- Formal system to characterize the types of PL elements and their interactions
- Purpose – to ensure meaningful interaction of different program parts – ie ensure absence of errors

Types

- A type is a collection of PL elements that share a common behavior.
- Types are typically associated with values, functions/procedures/methods and control structures.
- Types often have an atomic name for example **Int** but some don't such as a pair of Int.
- Type definition
Type IntPair = (Int, Int)

Note: Differs from data definition with constructors

Polymorphic type

type Pair a,b = (a,b)

type Property a = a -> Bool

- Parametric Polymorphism – type parameters

type Pos = Pair Int Int

type ListProperty a = Property [a]

- Typing Rules
 - Express judgement about types of expressions.
 - “assign” types to expressions
- Type Checking
 - An algorithm that used type declarations of values and functions in a program to ensure that all functions and values are used according to typing rules.

Type Errors

- Illegal combination of PL elements
- Lead to program crashes
- Cause incorrect computations

Expression Language with 2 Types

- Expr2.hs

Type Safety

- A programming language is called type safe if all type errors are detected.
- Type Safe – Lisp, Haskell, Java, Expr + eval, Expr + evalDynTC
- Unsafe Languages (type casts, pointers) - C, C++, Javascript (NaN)

UnSafe Language

- Unsafe eval function
 - Use Int as semantic domain
 - Map Boolean values to 0 and 1
- 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. For example “5” 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.

A Type Checker for Expression Language

- Expr2.hs
- TypeCheck.hs

Dynamic vs Static Typing

- Static Typing – Types and type errors are found before a program is run.
- Dynamic Typing – Types are checked while a program runs.

	(Mostly) Type Safe	Unsafe
Statically Typed	Haskell, Java	C, C++
Dynamically Typed	Lisp, Python	Javascript

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

What types are determined for each of the following expressions under static and dynamic typing? What is the type of x?

a) If $x < 2$ then even x else x

Static: Type Error

Dynamic: Bool if $x < 2$, otherwise Int

Type of x: Int

b) If head x then x else tail

Static: [Bool]

Dynamic: [Bool]

Type of x: [Bool]

c) If x then $x+1$ else $x-1$

Static: Type Error

Dynamic: Type Error

Type of x: Bool

What types are determined for each of the following expressions under static and dynamic typing? What is the type of x?

d) If False then “Hello” else x

Static: String
Dynamic: String
Type of x: String

e) If fst x == 1 then x + 1 else fst x

Static: Type Error
Dynamic: Int if $\text{fst } x = 1$, otherwise Type Error
Type of x: (Int, a)

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

	<i>Advantage</i>	<i>Disadvantage</i>
<i>Static Typing</i>	prevents type errors smaller & faster code early error detection (saves debugging)	rejects some o.k. programs
<i>Dynamic Typing</i>	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

HW 5

- Sample Input – Stack
- Sample Output – RankError, TypeError, A Stack
 - run p1 s1
 - rankP p1 0
- Questions

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

Names

- Design issues for names:
 - Are names case sensitive?
 - Are special words reserved words or keywords?
- Length
 - If too short, they cannot be connotative
 - Language examples:
 - C99: no limit but only the first 63 are significant; also, external names are limited to a maximum of 31
 - C# and Java: no limit, and all are significant
 - C++: no limit, but implementers often impose one

Names (continued)

- Special characters

- PHP: all variable names must begin with dollar signs
- Perl: all variable names begin with special characters, which specify the variable's type
- Ruby: variable names that begin with @ are instance variables; those that begin with @@ are class variables

Variables

- A variable is an abstraction of a memory cell
- Variables can be characterized as a sextuple of attributes:
 - Name
 - Address
 - Value
 - Type
 - Lifetime
 - Scope

Variables Attributes

- Name - not all variables have them
- Address - the memory address with which it is associated
 - A variable may have different addresses at different times during execution
 - A variable may have different addresses at different places in a program
 - If two variable names can be used to access the same memory location, they are called aliases
 - Aliases are created via pointers, reference variables, C and C++ unions
 - Aliases are harmful to readability (program readers must remember all of them)

Variables Attributes (continued)

- *Type* - determines the range of values of variables and the set of operations that are defined for values of that type; in the case of floating point, type also determines the precision
- *Value* - the contents of the location with which the variable is associated
 - The l-value of a variable is its address
 - The r-value of a variable is its value
- *Abstract memory cell* - the physical cell or collection of cells associated with a variable

Variable Attributes: Scope

- The *scope* of a variable is the range of statements over which it is visible
- The *local variables* of a program unit/block are those that are declared in that unit/block
- The *nonlocal variables* of a program unit/block are those that are visible in the unit/block but not declared there
- *Global variables* are a special category of nonlocal variables
- The scope rules of a language determine how references to names are associated with variables
- Blocks

Blocks

```
{ int x;①
  int y;②
③x := 1;
{ int x;③
  ④x := 5;
②y := ③x;⑤
};
{ int z;④
  ②y := ①x;
}
}
```

A **block** consists of a group of declarations and

- (a) a sequence of statements (in imperative languages)
- (b) an expression (in functional languages)

```
let ①x=1
②y=x①
in
let ③x=5
④z=x③
in (y②,z④)
```

Observe references to
local and non-local variables

Nested Blocks: Shadowing

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

hides

Declarations in inner blocks can temporarily hide declarations in enclosing blocks

```
let x=1  
  y=x  
in let x=5  
  z=x  
in (y,z)
```

hides

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

push

push

pop

push

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
```

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

Example ...

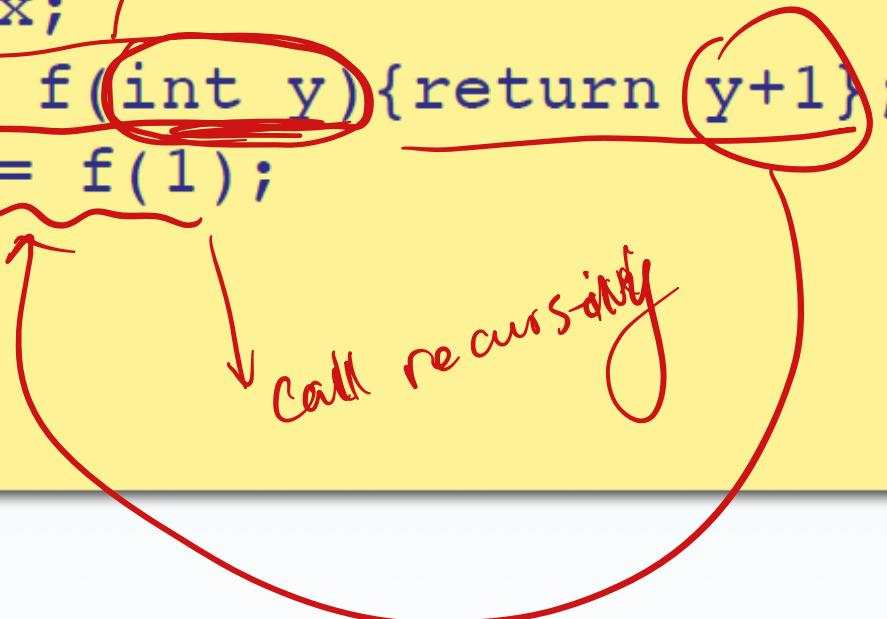
Var.hs
(Variables and Definitions)

Scope of Functions and Parameters

integer type function

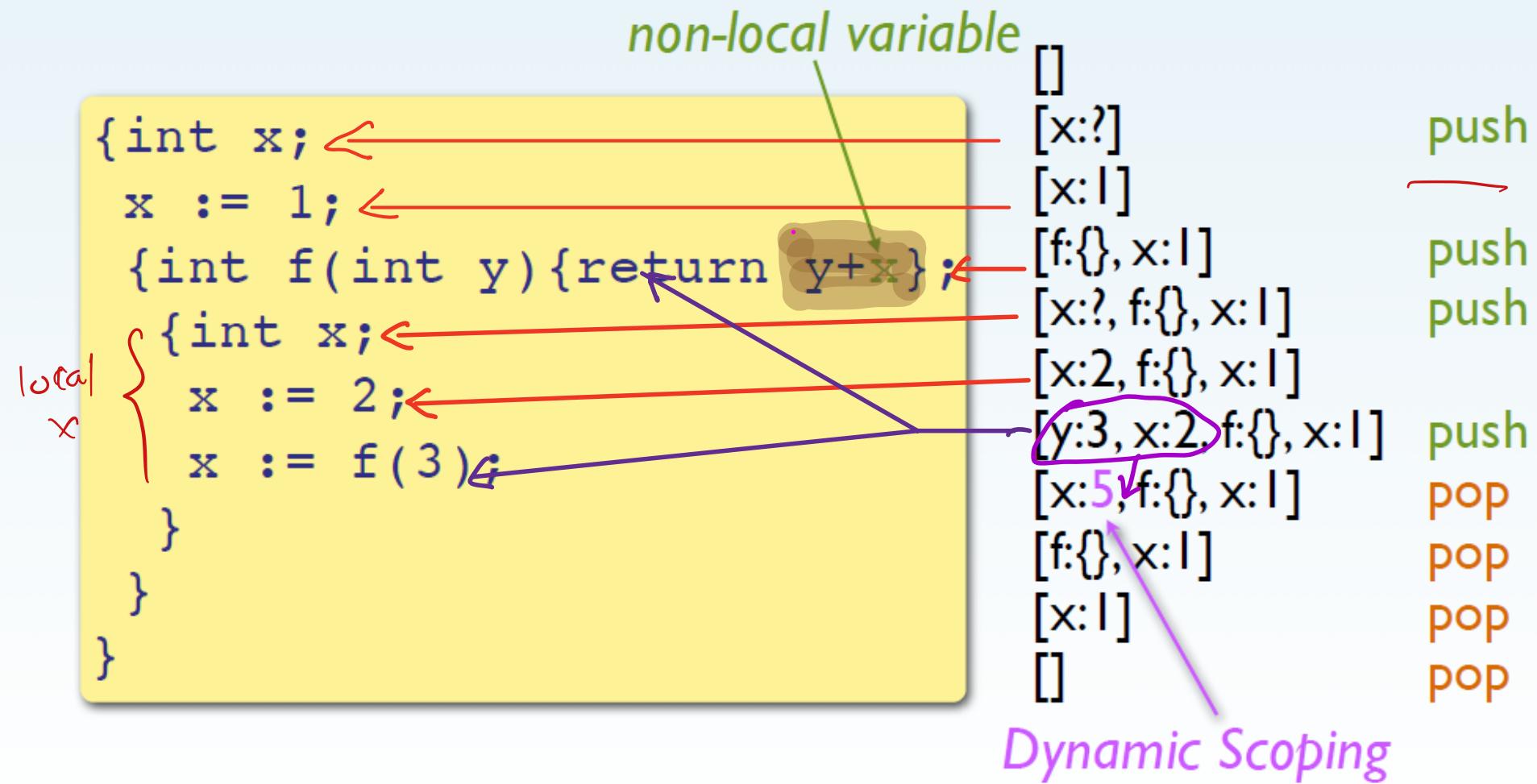
```
{int x;  
{int f(int y){return y+1};  
x := f(1);  
}  
}
```

call recursively



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

Dynamic Scoping



Example

FunDynScope.hs
(Functions)

Static vs. Dynamic Scoping

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

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);  
}  
}  
}
```

dynamic

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

Static Scoping

Implementation of Static Scoping

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

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);  
    }  
  }  
}
```

3+1
return 4

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

• Scope

Two Interpretations of Access Links

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

$[f:\{\}, x:1]$ definition of f
...
 $[y:3, x:2, f:\{\}, x:1]$ call of f
...

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

$[f:\{\}, x:1]$ definition of f
 $[x:2, f:\{\}, x:1]$
...
 $[[y:3, x:1], [x:2, ...]]$ temporary stack
...

(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

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 1

Draw the runtime stacks under dynamic scoping that result *immediately after* the statements on lines 8, 4, and 9 have been executed.

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

Annotations:

- Line 4: A red circle highlights the line number, and a red bracket encloses the entire function body. A red arrow points from this bracket to the text "not execute".
- Line 8: A red circle highlights the line number, and a red bracket encloses the assignment statement `x := 4;`. A red arrow points from this bracket to the text "call function".
- Line 9: A red circle highlights the line number, and a red bracket encloses the assignment statement `x := f(x-1);`.

Runtime stacks:

- 1: [x ?]
- 2: [x:2]
- 6: [f:{}, x:2]
- after 8: [x:4, f:{}, x:2]
- 3: [y:3, x:4, f:{}, x:2]
- after 4: [y:3, x:12, f:{}, x:2]
- after 9: [x:13, f:{}, x:2]

Dynamic:

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

1 [x: ?]

2 [x: 2]

3 [f:{}, x: 2]

4 [x: 4, f:{}, x: 2]

>>

5 [y: 3, x: 4, f:{}, x: 2]

6 [y: 3, x: 12, f:{}, x: 2]

7 [res: 13,
y: 3, x: 12, f:{}, x: 2]

8 <<

[x: 13, f:{}, x: 2]

Exercise 2

Draw the runtime stacks under static scoping that result *immediately after* the statements on lines 8, 4, and 9 have been executed.

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

1: [x: ?]
2: [x: 2]
6: [f:{}, x: 2]
after 8: [x: 4, f:{}, x: 2]

2: [x: 2]
6: [f:{}, x: 2]
after 8: [x: 4, f:{}, x: 2]

after 9: [x: 7, f:{}, x: 6]

Static:

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

1 [x: ?]

2 [x: 2]

3 [f:{}, x: 2]

6 >> 7 [x: ? f:{}, x: 2]

8 [x: 4, f:{}, x: 2]

>>

3 [y: 3, ~~x: 4, f:{}, x: 4~~]

4 [y: 3, ~~x: 6, f:{}, x: 6~~]

5 [y: 3, ~~x: 7, f:{}, x: 7~~]

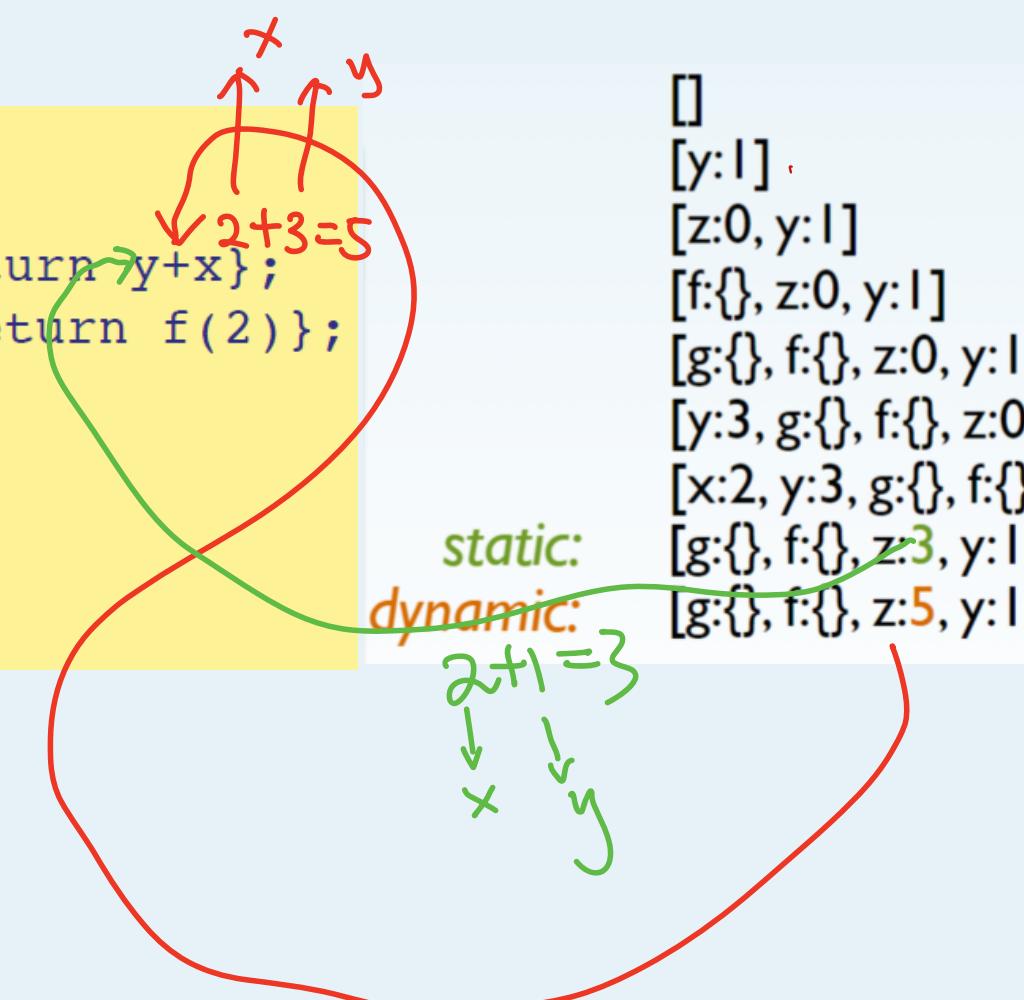
9 <<

[x: 7, f:{}, x: 2]

Exercise 3

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);  
}  
...  
}
```



```

1 {int y := 1;
2 {int z := 0;
3 {int f(int x){return y+x;};
4 {int g(int y){return f(2)};
5 z := g(3);
6 }
...

```

dynamic 1 [y: 1]

2 [z: 0, y: 1]

3 [f:{}, z: 0, y: 1]

5 >>

4 [y: 3, g:{} , f:{}, z: 0, y: 1]

3 >>

[x: 2,

y: 3, g:{} , f:{}, z: 0, y: 1]

3 [res: 5 ...]

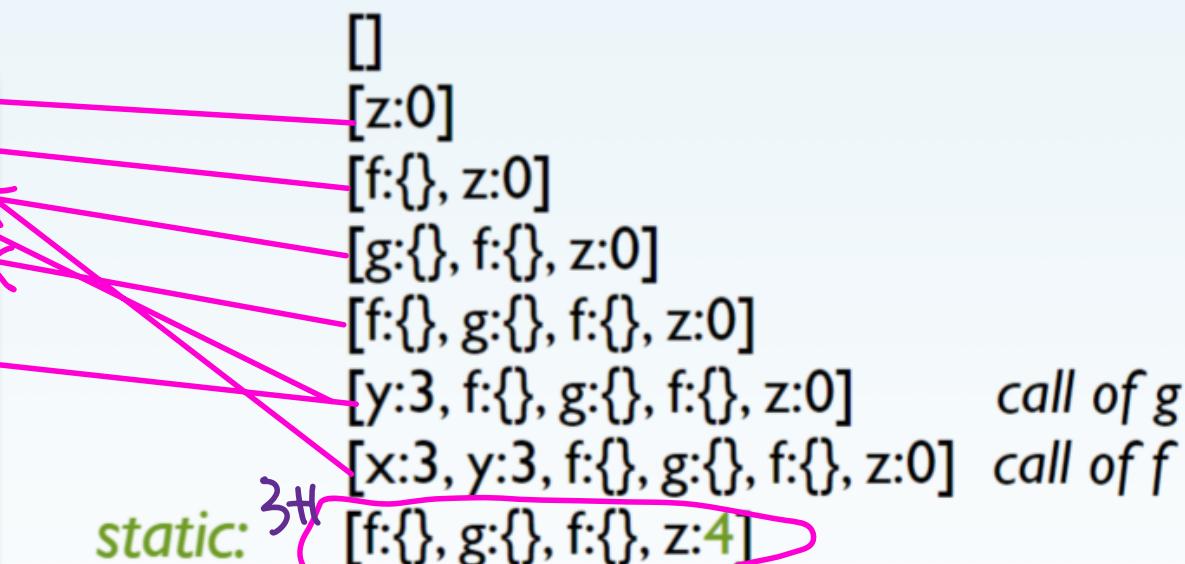
4 [res: 5 ...]

5 [z=5 ...]

Exercise 4

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);  
 }  
 ...
```



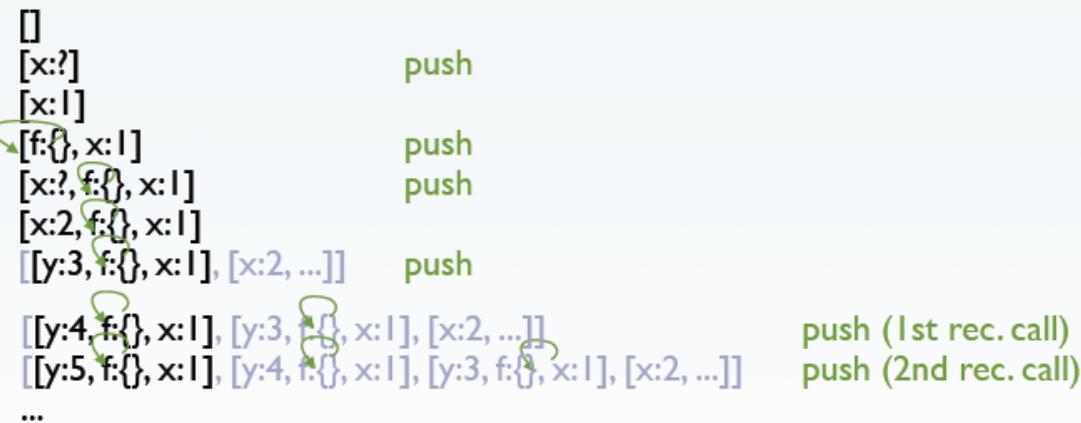
Implementation of Recursion

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

```

1 { int x;
2   int z;
3   z := 3;
4   { int f(int x) {
5     if x=0 then {
6       z := 1 }
7     else {
8       z := f(x-1)*2 };
9     return z;
10   };
11   x := f(3);
12 }
13 }
```

```

3 [z=3, x=?]
4 [f={}, z=3, x=?]
11 >>
5 [x=3, f={}, z=3, x=?]
8 >>
5 [x=2, x=3, f={}, z=3, x=?]
8 >>
5 [x=1, x=2, x=3, f={}, z=3, x=?]
8 >>
5 [x=0, x=1, x=2, x=3, f{}, z=3, x=?]
6 [x=0, x=1, x=2, x=3, f={}, z=1, x=?]
9 [res=1 x=0, x=1, x=2, x=3, f={}, z=1, x=?]
<<
8 [x=1, x=2, x=3, f={}, z=2, x=?]
9 [res=4 x=1, x=2, x=3, f={}, z=2, x=?]
<< 2
8 [x=2, x=3, f={}, z=4, x=?]
9 [res=8 x=2, x=3, f={}, z=4, x=?]
<< 4
8 [x=3, f={}, z=8, x=?]
9 [res=8 x=3, f={}, z=8, x=?]
<<
11 [f={}, z=8, x=8]
12 [z=8, x=8]
```

return value

Parameter Passing

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

(Formal) Parameter

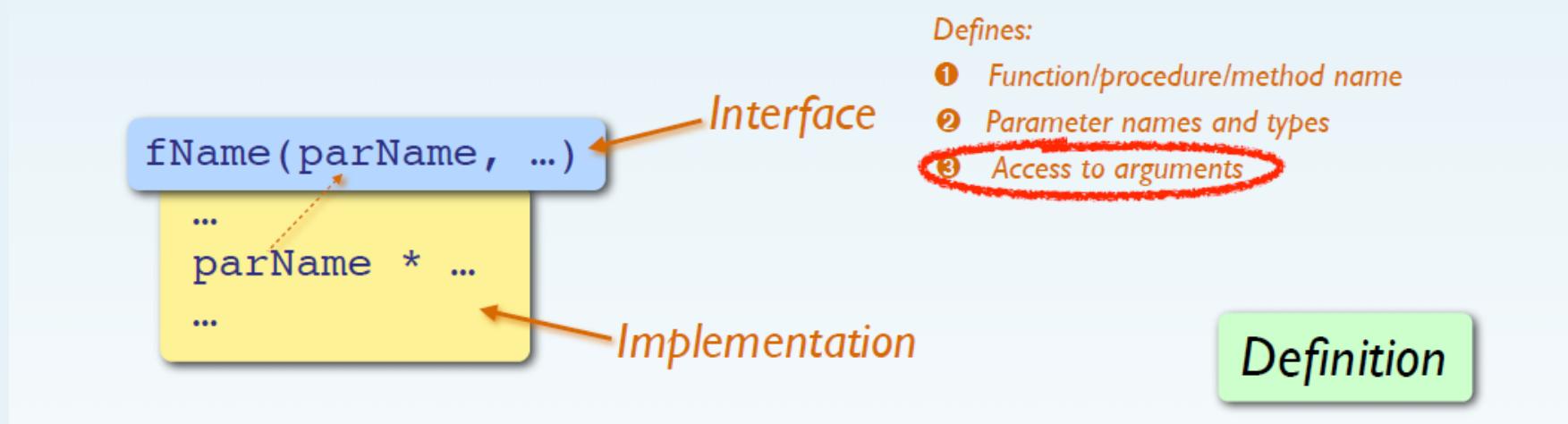
```
x := 3;  
z := 1;  
f(2*x,z);
```

Actual Parameter, or:
Argument

What is the value of z?

... it depends on the
parameter passing mechanism.

Function Definition & Use



Function Call

fName(3, x+1, x, ...)

6

[... ⟨x: 5⟩ ...]

- ① Evaluate argument expressions

[⟨a: ?, b: ?, c: ?⟩, ... ⟨x: 5⟩ ...]

- ② Create activation record with parameter names on runtime stack

[⟨a: 3, b: 6, c: x⟩, ... ⟨x: 5⟩ ...]

- ③ Store values and locations in activation record

- ④ Transfer control to function and run code

[⟨a: 7, b: 6, c: x⟩, ... ⟨x: 5⟩ ...]

[⟨a: 7, b: 6, c: x⟩, ... ⟨x: 1⟩ ...]

fName(a, b, c, ...)

...
a := b+1
c := a-b

N/A
[... ⟨x: 1⟩ ...]

- ⑤ Pass values back to call site

- ⑥ Remove activation record from runtime stack

fName(a, b, c, ...)

...
a := b+1
c := a-b

Call-By-Value

$fName(3, x+1, x, \dots)$

$\langle a: ?, b: ?, c: ? \rangle, \dots \langle x: 5 \rangle \dots$

$\langle a: 3, b: 6, c: 5 \rangle, \dots \langle x: 5 \rangle \dots$

\downarrow

$x+1 = 5+1$

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

$\langle a: 7, b: 6, c: 5 \rangle, \dots \langle x: 5 \rangle \dots$

$\langle a: 7, b: 6, c: 1 \rangle, \dots \langle x: 5 \rangle \dots$

$fName(a, b, c, \dots)$

...

$a := b+1$

$c := a-b$

⑤ ~~Pass values back to call site~~

⑥ Remove activation record from runtime stack

Call-By-Value

fName(p,...)

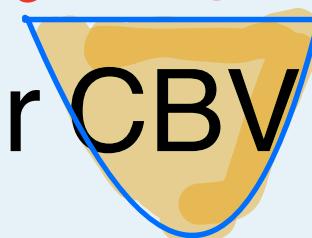
p := ...
... := p

p

Synopsis

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

call-by-value



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 >>  
4: [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*

$[(c: \bullet), \dots (x: 3) \dots]$

$[\dots (x: 5) \dots]$

$[(c: ?), \dots (x: 5) \dots]$

$[(c: \bullet), \dots (x: 5) \dots]$

~~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, ...)

...
 $c := c - 2$

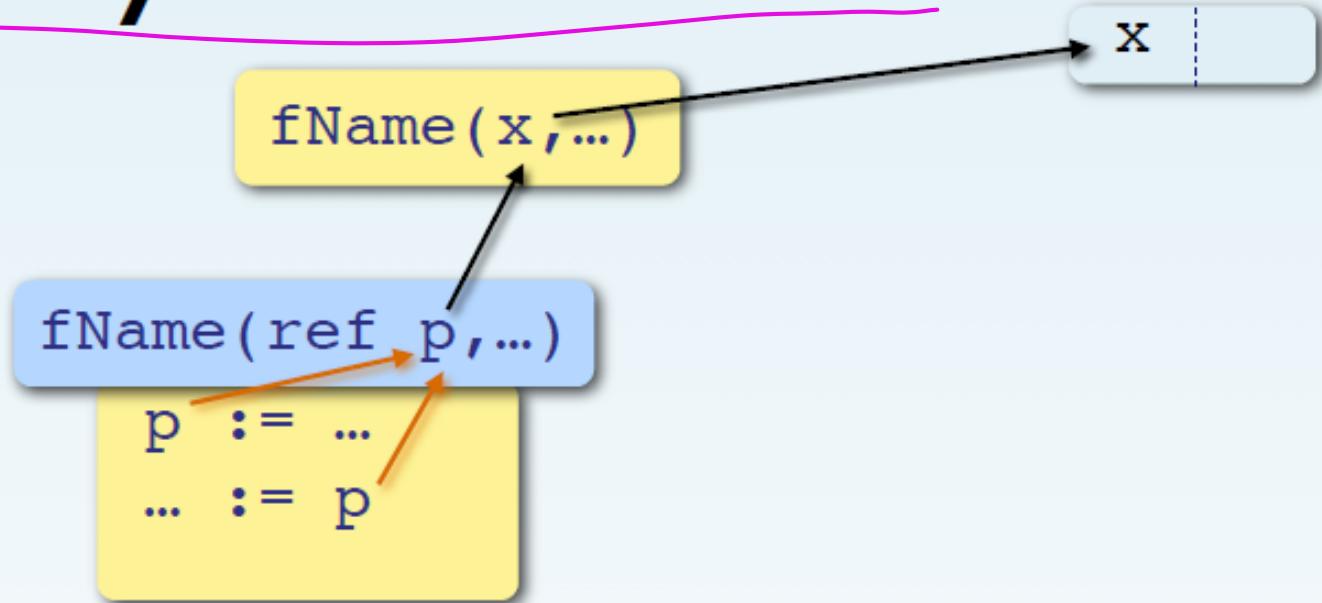
*Indicating
parameter
passing schema*

~~⑤ Pass values back to call site~~

$[\dots (x: 3) \dots]$

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

Compute: 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]  
...
```

3.

Call-By-Value-Result

~~fName(3, x+1, x, ...)~~

Only variables can be arguments

[... ⟨x: 5⟩ ...]

➊ Evaluate argument expressions

[⟨c: ?⟩, ... ⟨x: 5⟩ ...]

➋ Create activation record with parameter names on runtime stack

[⟨c: 5⟩, ... ⟨x: 5⟩ ...]

➌ Store values ~~and locations~~ in activation record

➍ Transfer control to function and run code

fName(~~valres~~ c, ...)

...
c := c-2

Indicating
parameter
passing schema

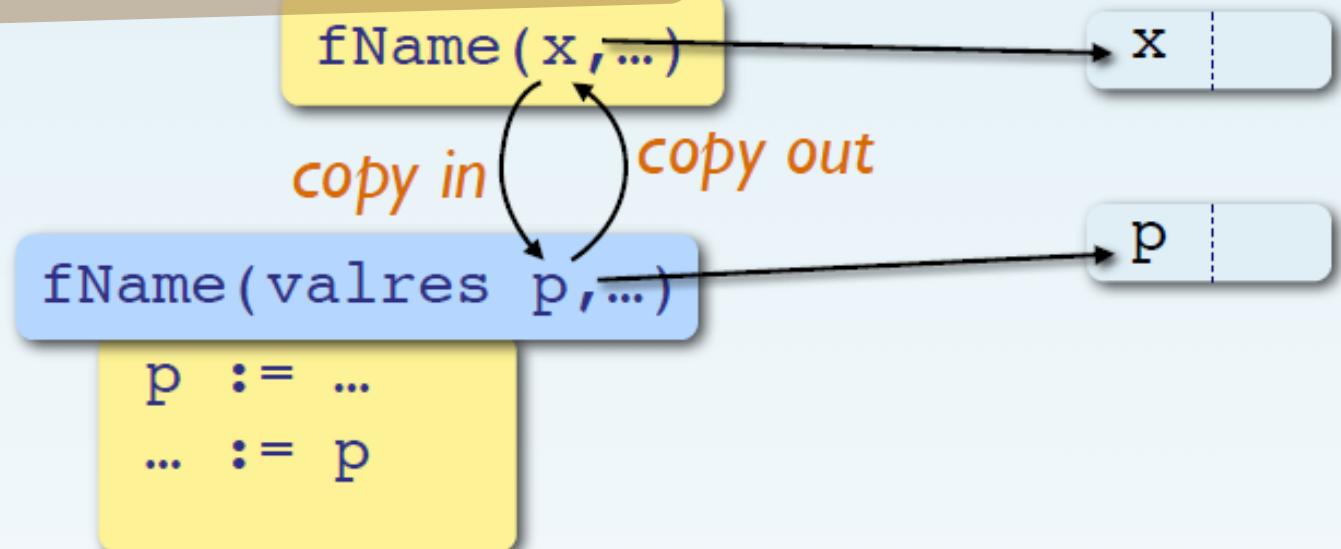
[⟨c: 3⟩, ... ⟨x: 5⟩ ...]

➎ Pass values back to call site

[... ⟨x: 3⟩ ...]

➏ Remove activation record from runtime stack

4 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⟩ ...]

x := f(x);

“before” “after”
[⟨x: 5⟩ ...] [⟨x: 7⟩ ...]

Does it matter?

Yes!

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

```
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]

call by reference

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] → x = y + 5
4 [y:6, f:{}, x:1]
5 [res:6, y:6, f:{}, x:1]
<< return from f
7 [z:11, f:{}, x:6]
```

y: y + 1
5

Call-By-Name

fName(x+1 , ...)

fName(name a,...)
local variable
z := a-2
x := 2
z := 2*a

Indicating parameter passing schema

[... ⟨x: 5⟩ ...]

① Evaluate argument expressions

[⟨a: ?⟩, ... ⟨x: 5⟩ ...]

② Create activation record with parameter names on runtime stack

[⟨a: x+1⟩, ... ⟨x: 5⟩ ...]

expressions

③ Store values and locations in activation record

④ Transfer control to function and run code

fName(name a,...)

local variable
z := a-2
x := 2
z := 2*a

Indicating parameter passing schema

[⟨z: 4, a: x+1⟩, ... ⟨x: 5⟩ ...]

⑤ Pass values back to call site

[⟨z: 4, a: x+1⟩, ... ⟨x: 2⟩ ...]

⑥ Remove activation record from runtime stack

[⟨z: 6, a: x+1⟩, ... ⟨x: 2⟩ ...]

[... ⟨x: 2⟩ ...]

Call-By-Name

fName(e, ...)

fName(name p, ...)

... := ... p

... := ... (e)

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 , ...)

fName(name a,...)

local variable

```
z := a-2
x := 2
z := 2*a
```

Indicating parameter passing schema

[... ⟨x: 5⟩ ...]

① Evaluate argument expressions

[⟨a: ?⟩, ... ⟨x: 5⟩ ...]

② Create activation record with parameter names on runtime stack

[⟨a: x+1⟩, ... ⟨x: 5⟩ ...]

expressions, replaced by values

③ Store values and locations in activation record

④ Transfer control to function and run code

[⟨z: ?, a: x+1⟩, ... ⟨x: 5⟩ ...]

[⟨z: 4, a: 6⟩, ... ⟨x: 5⟩ ...]

[⟨z: 4, a: 6⟩, ... ⟨x: 2⟩ ...]

[⟨z: 12, a: 6⟩, ... ⟨x: 2⟩ ...]

fName(need a,...)

local variable

```
z := a-2
x := 2
z := 2*a
```

Indicating parameter passing schema

⑤ Pass values back to call site

[... ⟨x: 2⟩ ...]

⑥ 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 };
```

```
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		

Pass the Orange Juice ...



Call By:	Value	Reference	Value/Result	Name	Need
----------	-------	-----------	--------------	------	------

```

1 { int x;
2   int y;
3   y := 2;
4   { int f(int x) {
5     if x=1 then {
6       y := 0 }
7     else { 2 } y := f(x-1)+y+2 ;
8     return y;
9   };
10  }; x := f(3);
11  };
12  };
13  }

```

$$f(1) = 0$$

$$f(n) = f(n-1) + f(n-1) + 2$$

$$f(n) = 2 f(n-1) + 2$$

n	$f(n)$
1	0
2	2
3	6
4	14
5	30

```

[] 
2 [y=?, x=?]
3 [y=2, x=?]
4 [f={}, y=2, x=?]

```

```

11 >>
5 [x=3, f={}, y=2, x=?]
8 >>

```

```

5 [x=2, x=3, f={}, y=2, x=?]
8 >>

```

```

5 [x=1, x=2, x=3, f={}, y=2, x=?]
6 [x=1, x=2, x=3, f={}, y=0, x=?]
9 [res=0 x=1, x=2, x=3, f={}, y=0, x=?]
<<

```

```

8 [x=2, x=3, f={}, y=2, x=?]
9 [res=2, x=2, x=3, f={}, y=2, x=?]
<<

```

```

8 [x=3, f={}, y=6, x=?]
9 [res=6, x=3, f={}, y=6, x=?]
<<

```

```

11 [f={}, y=6, x=6]
12 [y=6, x=6]

```

Example Languages

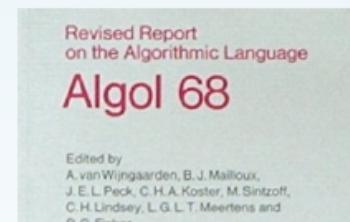
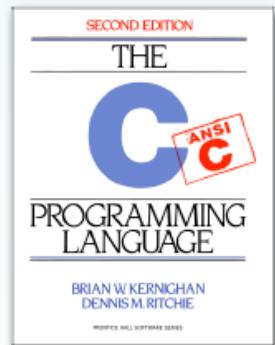
Call-by-Value

Call-by-Reference

Call-by-Value-Result

Call-by-Name

Call-by-Need



Haskell

Scala – Call-by-Name

- Multi-paradigm: concurrent, functional, imperative, object-oriented
- OOP like Java but more concise with many functional features like Haskell

```

1  object Main {
2    // Call-by-Value
3    def byValuePrint(x: Long) = {
4      println(x);
5      println(x);
6    }
7    // Call-by-Name
8    def byNamePrint(x: => Long) = {
9      println(x);
10     println(x);
11   }
12
13   def main(args: Array[String]): Unit = {
14     println("By Name Print")
15     byNamePrint(System.nanoTime())
16     println("By Value Print")
17     byValuePrint(System.nanoTime())
18   }
19 }
```

```

> scalac -classpath . -d . main.scala
> scala -classpath . Main
By Name Print
4088420852756
4088420995606
By Value Print
4088421110686
4088421110686
```

Scala – Call-by-Name

- Multi-paradigm: concurrent, functional, imperative, object-oriented
- OOP like Java but more concise with many functional features like Haskell

```
def infinite(): Int = 1 + infinite()
def printFirst(x: Int, y: => Int) = println(x)
```

```
> scalac -classpath . -d . main.scala
> scala -classpath . Main
50
```

```
printFirst(50, infinite())
printFirst(infinite(), 50)
```



```
at Main$.infinite(main.scala:13)
exit status 1
```



Ada: Call-by-Value-Result

- Ada is a structured, statically typed, imperative, and object-oriented high-level programming language, extended from Pascal and other languages. It has built-in language support for design by contract (DbC), extremely strong typing, explicit concurrency, tasks, synchronous message passing, protected objects, and non-determinism. Ada improves code safety and maintainability by using the compiler to find errors in favor of runtime errors.
- [https://en.wikipedia.org/wiki/Ada_\(programming_language\)](https://en.wikipedia.org/wiki/Ada_(programming_language))
- https://www.tutorialspoint.com/compile_ada_online.php
- <https://onecompiler.com/ada/3xu835bf9>

Ada: Call-by-Value-Result

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 procedure CallParameters is
3     procedure PassModes (A : in Integer;
4                           B : out Integer;
5                           C : in out Integer) is
6 begin
7     Put_Line("Enter Procedure-----");
8     Put_Line("A :"&Integer'Image(A));
9     -- Should not use out parameters until assigned
10    -- Put_Line(Integer'Image(B));
11    Put_Line("C :"&Integer'Image(C));
12    -- Assignments can not be made to in
13    -- A := 2;
14    B := 100;
15    C := C * 2;

16    Put_Line("A :"&Integer'Image(A));
17    Put_Line("B :"&Integer'Image(B));
18    Put_Line("C :"&Integer'Image(C));
19    Put_Line("Leaving Procedure-----");
20 end PassModes;

21
22 X : Integer := 12;
23 Y : Integer := 44;
24 Z : Integer := 30;
25
26 -- Main
27 begin
28     Put_Line("Calling Methods in, out, in out");
29     Put_Line("Before Call-----");
30     Put_Line("X :" &Integer'Image(X));
31     Put_Line("Y :" & Integer'Image(Y));
32     Put_Line("Z :" & Integer'Image(Z));
33     PassModes(X,Y,Z);
34     Put_Line("After Call-----");
35     Put_Line("X :" &Integer'Image(X));
36     Put_Line("Y :" & Integer'Image(Y));
37     Put_Line("Z :" & Integer'Image(Z));
38
39 end CallParameters;
```

```
Calling Methods in, out, in out
Before Call-----
X : 12
Y : 44
Z : 30
Enter Procedure-----
A : 12
C : 30
A : 12
B : 100
C : 60
Leaving Procedure-----
After Call-----
X : 12
Y : 100
Z : 60
```



```
1 using System;
2 class Program {
3
4     public static void Main (string[] args) {
5         int x = 7;
6         int y = 10;
7         Console.WriteLine ("Start main x = "+x+", y = "+y);
8         PassModes(x, out y);
9         Console.WriteLine ("End main x = "+x+", y = "+y);
10    }
11
12    static void PassModes( int x, out int y)
13    {
14        // Console.WriteLine ("x = "+x+", y = "+y);
15        x = x*2;
16        y = x;
17        Console.WriteLine ("End Passmodes x = "+x+", y = "+y);
18    }
19 }
```

```
Start main x = 7, y = 10
End Passmodes x = 14, y = 14
End main x = 7, y = 14
```

C++

```
1 #include <stdio.h>
2 v void callExamples(int x, int &y) {
3     printf("top fun: x %d y %d \n",x,y);
4     x = x * 2;
5     y = y + 100;
6     printf("end fun: x %d y %d \n",x,y);
7 }
8 v int main(void) {
9     int x = 7;
10    int y = 10;
11    printf("top main: x %d y %d \n",x,y);
12    callExamples(x,y);
13    printf("end main: x %d y %d \n",x,y);
14    return 0;
15 }
```

```
> make -s
> ./main
top main: x 7 y 10
top fun: x 7 y 10
end fun: x 14 y 110
end main: x 7 y 110
> █
```

7. Prolog

- Introduction
- Logical Programming
- Predicates & Goals
- Rules
- Satisfying Goals, Recursion & Backtracking
- Equality
- Arithmetic
- Pitfalls

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

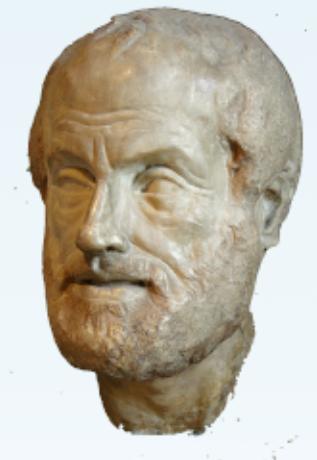
Logic: A Tool for Reasoning

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

E.g.: *All humans are mortal.*

Socrates is human.

Therefore: Socrates is mortal.

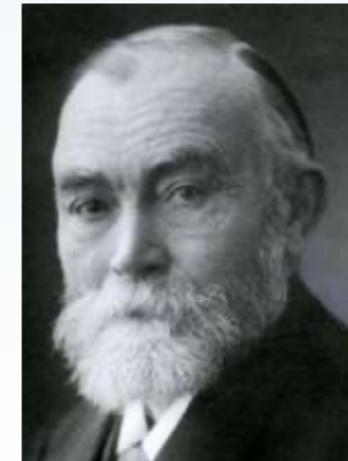


First-Order Logic (*Gottlob Frege, 1879 “Begriffsschrift”*)

E.g.: $\forall x. \text{Human}(x) \rightarrow \text{Mortal}(x)$

$\text{Human}(\text{Socrates})$

$\therefore \text{Mortal}(\text{Socrates})$



Comparing Logical, Functional and Imperative Languages

Concept	Prolog	Haskell	C, Java, etc.
Algorithm	Set of rules	Function	Program
Instruction	Term	Expression	Statement
Input	Value Argument	Function Argument	<code>read</code> Statement
Output	Variable Binding	Function Result	<code>print</code> Statement
Iteration	Recursion	Recursion	Loop
Step	Term unification	Expression simplification	State update
Computation	Tree of terms		Sequence of steps

Logic and Programming

Prolog program

Fact

$$\forall x. \text{Human}(x) \rightarrow \text{Mortal}(x)$$

Human(Socrates)

Rule

$\therefore \text{Mortal}(\text{Socrates})$

Goal / Query

Program execution

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

Predicates, Functions & Sets

Nullary Predicate \cong Set A

Term = {**true**, **false**, **not true**, **not false**, ...}

\approx SQL query over table
(with schema A, AxB, ...)

Unary Predicate (over A) $\cong A \rightarrow \text{Bool} \cong$ Subset of A

Even : $\mathbb{N} \rightarrow \mathbb{B} = \{(0, \text{true}), (1, \text{false}), (2, \text{true}), \dots\} \supseteq \{0, 2, 4, \dots\} \subseteq \mathbb{N}$

Binary Predicate (over A and B) $\cong A \times B \rightarrow \text{Bool} \cong$ Subset of A \times B

$<: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B} = \{((0, 0), \text{false}), ((0, 1), \text{true}), \dots, ((5, 3), \text{false}), \dots, ((5, 7), \text{true}), \dots\}$
 $\cong \{(0, 1), \dots, (5, 7), \dots\} \subseteq \mathbb{N} \times \mathbb{N}$

Predicates & Goals

The basic entities of Prolog are *predicates*
(Note: predicate \cong relation \cong 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)

Exercise

Define the successor predicate
on the first 5 natural numbers

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

Define the less than predicate
on the first 3 natural numbers

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

Goals & Queries

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

Prolog prompt → `?- color(red).` ← *Goal*
`true.` ← *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):

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

`likes.pl`

Prolog program →

Variables

Goals may contain variables

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

```
?- color(X).  
X = red ↲
```

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

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

satisfied with answer

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

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

want more evidence

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

enumerating a predicate

Exercise

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

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

Which goal finds the successor of 3?

```
?- successor(3,X).  
X = 4.
```

Which goal finds the predecessor of 2?

```
?- successor(X,2).  
X = 1.
```

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

```
?- lt(1,X).  
X = 2 ;  
X = 3.
```

Variables are “1st order”

Variables cannot be used for predicates:



```
?- X(red).  
?- X(john,mary).
```

- ⇒ We can only ask whether a particular predicate holds.
- ⇒ We cannot ask which predicate(s) hold for objects.

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

Conjunctions

Do John and Mary like each other?

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

Comma denotes logical and

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

Which colors does John like?

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

No other solutions found

Rules

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

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

`:-` denotes \Leftarrow (read “if”)

Rules can contain *free variables*

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

Are there any other solutions? Which?

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

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

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

```
?-
```

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

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

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

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

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

```
?- inside(mouse,house).  
true.
```

(2)

Why?

```
contains(house,kitchen), inside(mouse,kitchen).
```

Τ

(2)

Τ

```
contains(kitchen,fridge), inside(mouse,fridge).
```

Τ

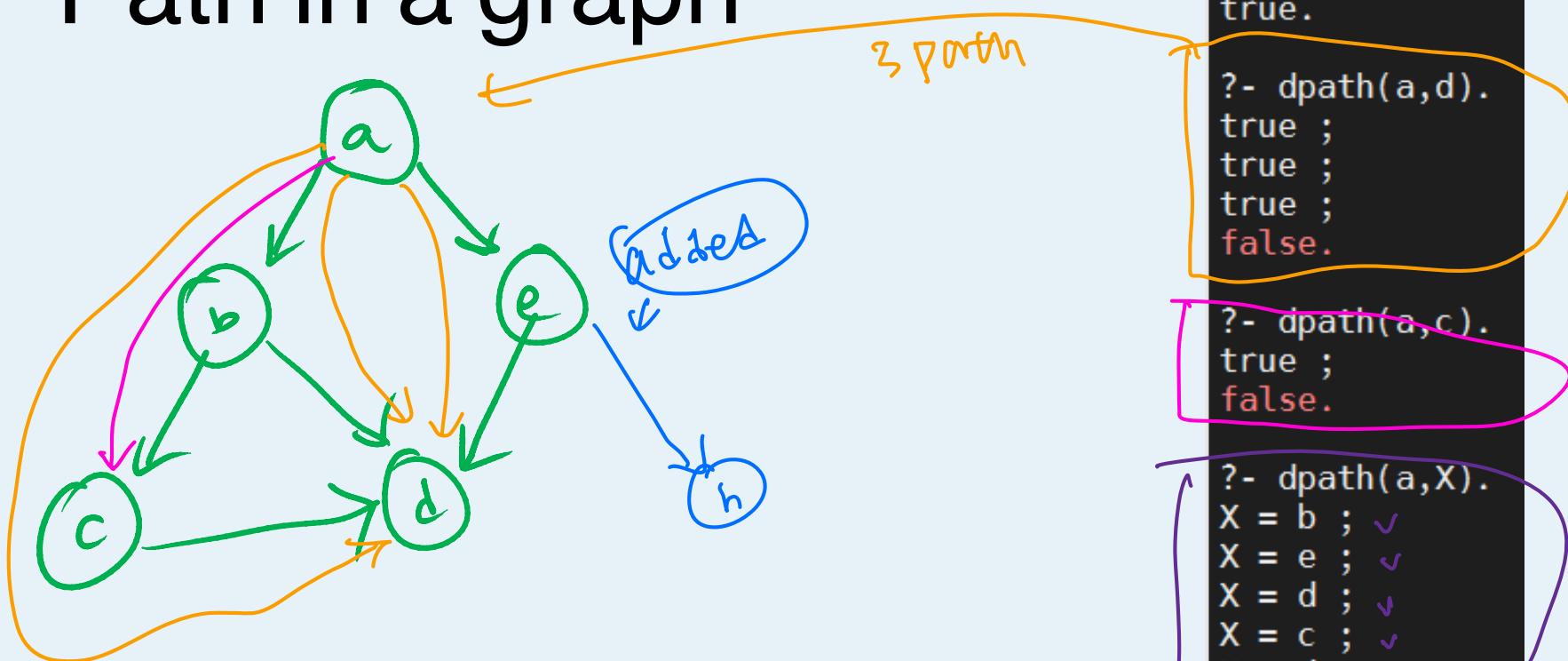
(1)

Τ

```
contains(fridge,mouse).
```

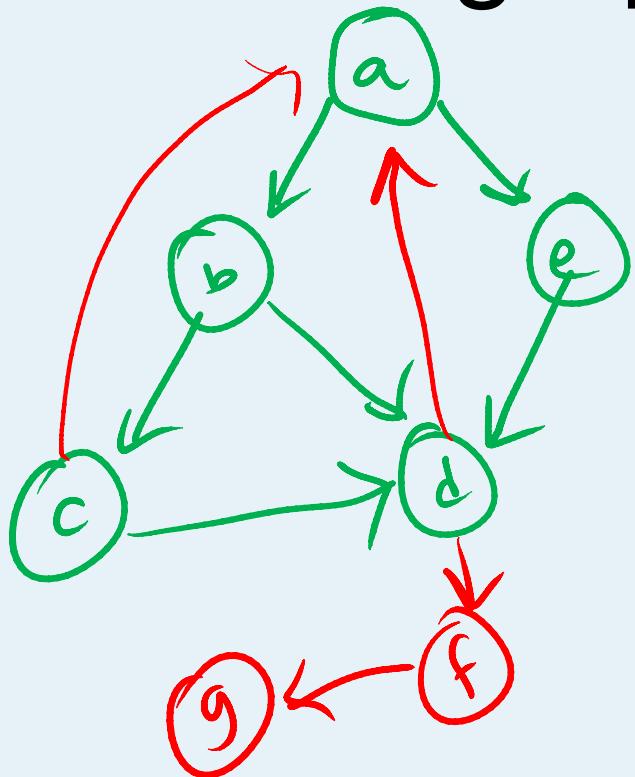
Τ

Path in a graph



```
edge(a,b).  
edge(a,e).  
edge(b,d).  
edge(b,c).  
edge(c,d).  
edge(e,d).  
dpath(X,Y):-edge(X,Y).  
dpath(X,Y) :- edge(X,Z),dpath(Z,Y).
```

Path in a graph

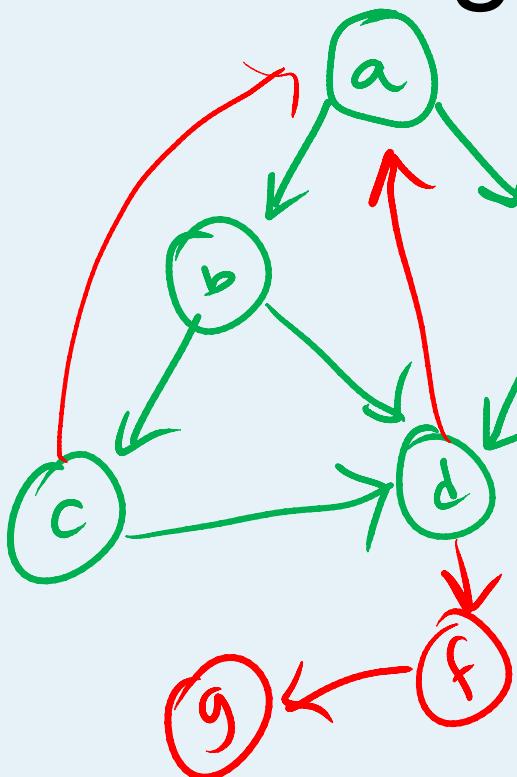


```
edge(a,b).  
edge(a,e).  
edge(b,d).  
edge(b,c).  
edge(c,a).  
edge(c,d).  
edge(e,d).  
edge(d,a).  
edge(d,f).  
edge(f,g).
```

```
dpath(X,Y):-edge(X,Y).  
dpath(X,Y) :- edge(X,Z),dpath(Z,Y).
```

```
?- dpath(a,g).  
ERROR: Stack limit (1.0Gb) exceeded  
ERROR: Stack sizes: local: 1.0Gb, global: 33Kb, trail: 33.0Mb  
ERROR: Stack depth: 4,329,134, last-call: 0%, Choice points: 4,329,127  
ERROR: Possible non-terminating recursion:  
ERROR: [4,329,133] user:dpath(a, g)  
ERROR: [4,329,132] user:dpath(d, g)  
?- █
```

Path in a graph

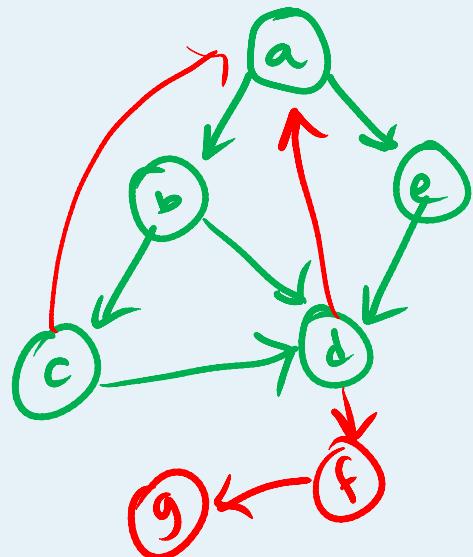


```
edge(a,b).  
edge(a,e).  
edge(b,d).  
edge(b,c).  
edge(c,a).  
edge(c,d).  
edge(e,d).  
edge(d,a).  
edge(d,f).  
edge(f,g).  
  
dpath(X,Y):-edge(X,Y).  
dpath(X,Y) :- edge(X,Z), dpath(Z,Y).
```

```
?- dpath(a,X).  
X = b ;  
X = e ;  
X = d ;  
X = c ;  
X = a ;  
X = f ;  
X = b ;  
X = e ;  
X = d ;  
X = c ;  
X = a ;  
X = f ;  
X = b ;  
X = e ;  
X = d ;  
X = c ;  
X = a ;  
X = f ;  
X = b ;  
X = e ;  
X = d ;  
X = c ;  
X = a ;  
X = f ;  
X = b ;  
X = e ;  
X = d ;  
X = c ;  
X = a ;  
X = f ;
```

DFS -Path in a graph

```
edge(a,b).  
edge(a,e).  
edge(b,d).  
edge(b,c).  
edge(c,a).  
edge(c,d).  
edge(e,d).  
edge(d,a).  
edge(d,f).  
edge(f,g).  
  
member(X,[X|_]).  
member(X,[_|Tail]) :- member(X,Tail).  
  
solve(S,F,Solution) :- dfs([],S,F,Solution).  
  
dfs(Path, X,F, [X| Path]) :- X==F.  
  
dfs(Path, X,F, Sol) :- edge(X,Y), not(member(Y,Path)), dfs([X| Path],Y,F,Sol).  
  
dpath(X,Y) :- edge(X,Y).  
dpath(X,Y) :- edge(X,Z), dpath(Z,Y).
```



```
?- solve(a,g,X).  
X = [g, f, d, b, a] ;  
X = [g, f, d, c, b, a] ;  
X = [g, f, d, e, a] ;  
false.
```

```
?- █
```

Prolog & Databases

Sailor database

sailor(sid, sname, rating, age)

boat(bid, bname, color)

reserves(sid, bid, date)

```
boat(101, interlake, pink).  
boat(102, interlake, red).  
boat(103, clipper, green).  
boat(104, marine, red).  
reserves(21, 101, feb0122).  
reserves(21, 102, feb0522).  
reserves(21, 103, feb1622).  
reserves(21, 104, jan1222).  
reserves(21, 104, feb1122).  
reserves(31, 101, jan0722).  
reserves(31, 103, feb0822).  
reserves(31, 104, feb2122).  
reserves(64, 104, jan1222).  
reserves(64, 102, dec1221).  
reserves(74, 103, dec2221).  
sailor(21, Dustin, 7, 45).  
sailor(29, brutus, 1, 31).  
sailor(31, lubber, 8, 56).  
sailor(32, andy, 8, 25).  
sailor(58, rusty, 10, 35).  
sailor(64, horatio, 7, 35).  
sailor(74, horatio, 9, 38).  
sailor(99, art, 3, 25).
```

sailor(sid, sname, rating, age)

boat(bid, bname, color)

reserves(sid, bid, date)

- Find the names and ages of all sailors

```
?- sailor(_,Name,_,Age).  
Name = dustin,  
Age = 45 ;  
Name = brutus,  
Age = 31 ;  
Name = lubber,  
Age = 56 ;  
Name = andy,  
Age = 25 ;  
Name = rusty,  
Age = 35 ;  
Name = horatio,  
Age = 35 ;  
Name = horatio,  
Age = 38 ;  
Name = art,  
Age = 25.
```

`sailor(sid, sname, rating, age)`

`boat(bid, bname, color)`

`reserves(sid, bid, date)`

- Find all sailors with a rating over 7

```
?- sailor(_,Name,Rating,_), Rating>7.  
Name = lubber,  
Rating = 8 ;  
Name = andy,  
Rating = 8 ;  
Name = rusty,  
Rating = 10 ;  
Name = horatio,  
Rating = 9 ;  
false.  
  
?- █
```

sailor(sid, sname, rating, age)

boat(bid, bname, color)

reserves(sid, bid, date)

Find the names of the sailors who have reserved boat 103.

```
?- sailor(Sid,Name,_,_), reserves(Sid,103,_).  
Sid = 21,  
Name = dustin ;  
Sid = 31,  
Name = lubber ;  
Sid = 74,  
Name = horatio ;  
false.
```

```
?- █
```

sailor(sid, sname, rating, age)

boat (bid, bname, color)

reserves(sid, bid, date)

List the names of the sailors and the names of the boats they reserve.

```
?- sailor(Sid,Name,_,_),reserves(Sid,Bid,_),boat(Bid,BName,_).  
Sid = 21,  
Name = dustin,  
Bid = 101,  
BName = interlake ;  
Sid = 21,  
Name = dustin,  
Bid = 102,  
BName = interlake ;  
Sid = 21,  
Name = dustin,  
Bid = 103,  
BName = clipper ;  
Sid = 21,  
Name = dustin,  
Bid = 104,  
BName = marine ;  
Sid = 21,  
Name = dustin,  
Bid = 104,  
BName = marine ;  
Sid = 31,  
Name = lubber,  
Bid = 101,  
BName = interlake ;  
Sid = 31,  
Name = lubber,  
Bid = 103,  
BName = clipper ;  
Sid = 31,  
Name = lubber,  
Bid = 104,  
BName = marine ;  
Sid = 64,  
Name = horatio,  
Bid = 104,  
BName = marine .
```

sailor(sid, sname, rating, age)

boat(bid, bname, color)

reserves(sid, bid, date)

Find the Names of the sailors who have reserved a red boat

```
red(Name) :- sailor(Sid,Name,_,_), reserves(Sid,Bid,_), boat(Bid,_,red).
```

```
?- red(Name).
Name = dustin ;
Name = dustin ;
Name = dustin ;
Name = lubber ;
Name = horatio ;
Name = horatio ;
false.
```

```
?- 
```

```
sailor(sid, sname, rating, age)
```

```
boat(bid, bname, color)
```

```
reserves(sid, bid, date)
```

Find the Names of the sailors who have reserved a red boat or a pink boat

```
redorpink(Name) :- sailor(Sid, Name, _, _), reserves(Sid, Bid, _), boat(Bid, _, red).  
redorpink(Name) :- sailor(Sid, Name, _, _), reserves(Sid, Bid, _), boat(Bid, _, pink).
```

```
?- redorpink(Name).  
Name = dustin ;  
Name = dustin ;  
Name = dustin ;  
Name = lubber ;  
Name = horatio ;  
Name = horatio ;  
Name = dustin ;  
Name = lubber ;  
false.
```

```
?- █
```

sailor(sid, sname, rating, age)

boat(bid, bname, color)

reserves(sid, bid, date)

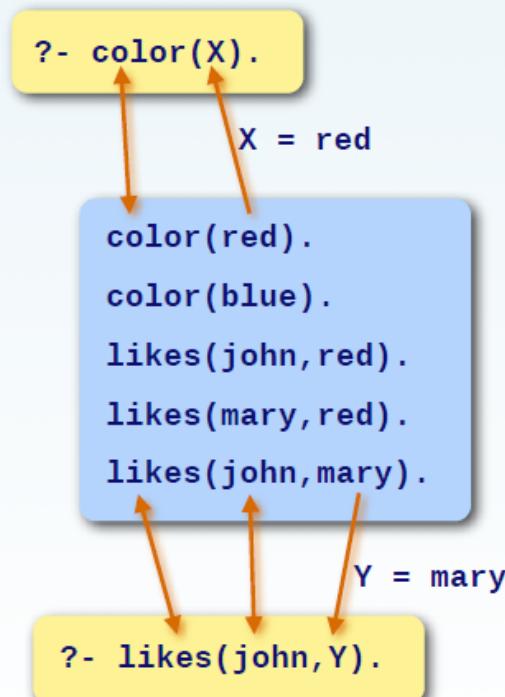
Find the Names of the sailors who have reserved a red boat and a pink boat.

```
redandpink(Name) :- sailor(Sid,Name,_,_), reserves(Sid,Bid1,_), boat(Bid1,_,pink), reserves(Sid,Bid2,_), boat(Bid2,_,red).
```

“Fact” Queries

Simplified view:

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



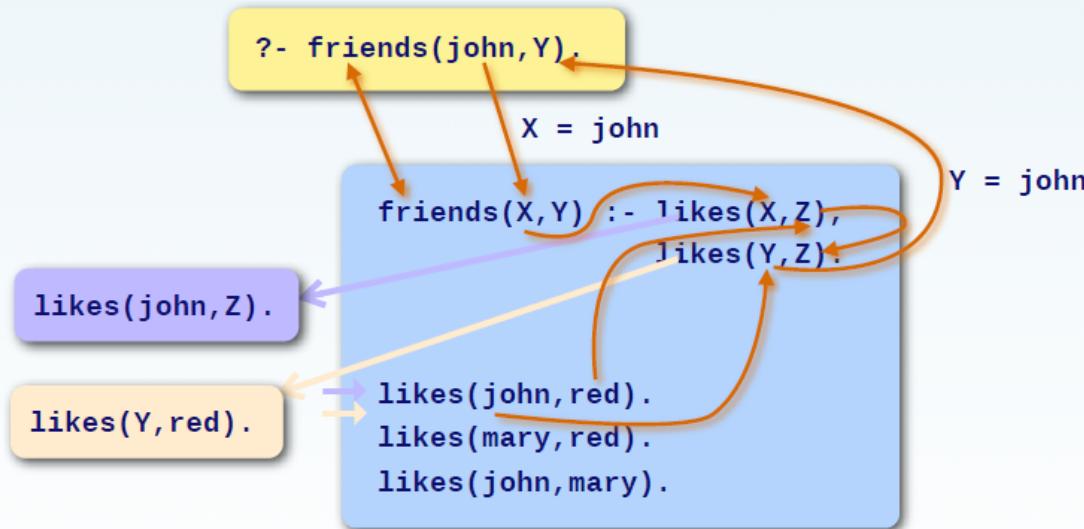
Variables in goals:
Result parameters

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 “ \rightarrow ” in the Prolog database.

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

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

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

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

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

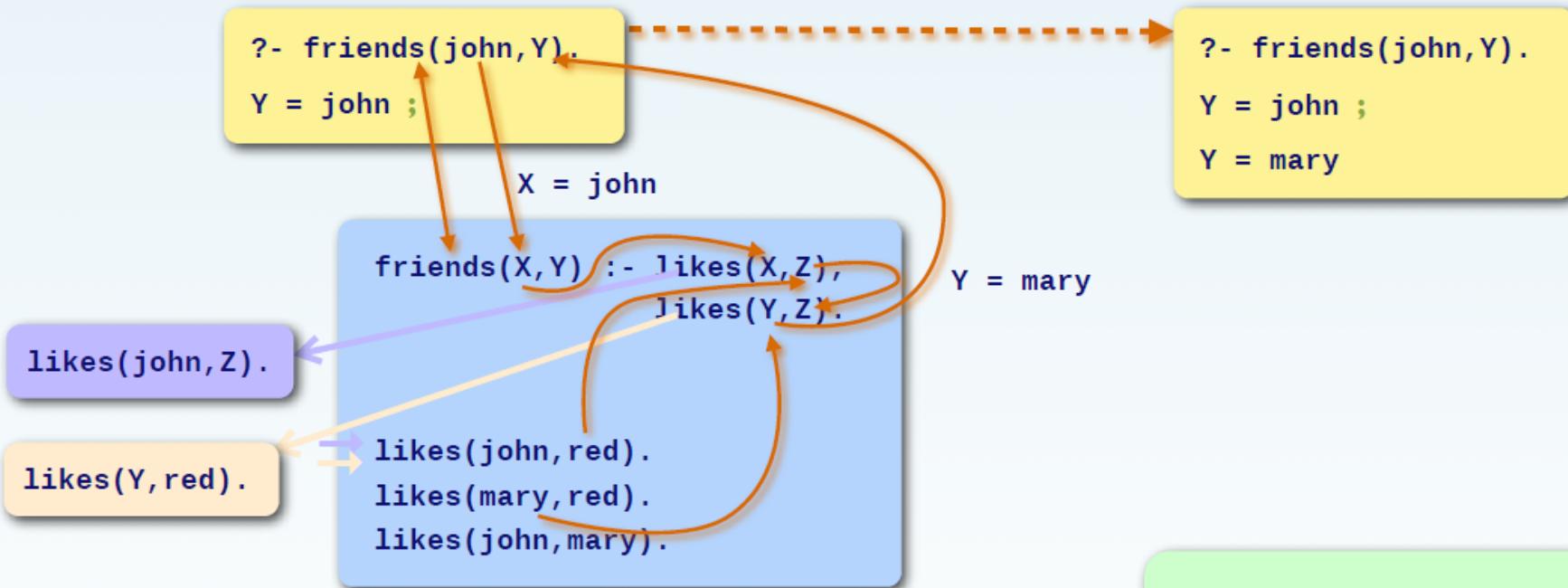
```
?- color(X).  
X = red ;  
X = blue ;  
false.
```

Disregard solution, i.e.
assume failure
 \Rightarrow advance pointer

No more facts

Not always printed. A more complex
goal might be needed, e.g.
`likes(john,X), color(X).`

Multiple Predicate Instances

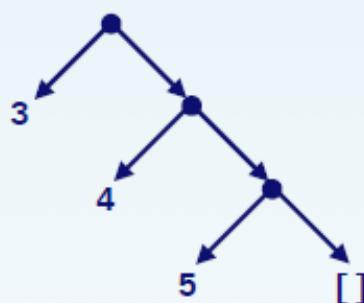


*Advanced pointer
leads to new binding(s)*

Lists

Lists are terms with special syntax

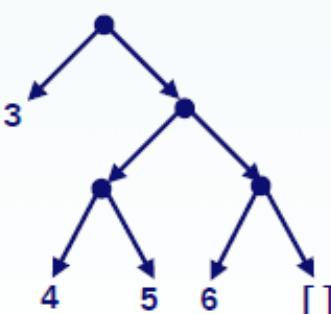
Empty list
[]



[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*

```
[X,Y|Z]
```

*Pattern
in Haskell*

```
x:y:z
```

```
[X,Y,Z]
```

```
[x,y,z]
```

List Predicates

Wildcard, matches anything

```
member(X, [X|_]).
```

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

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

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

Test

```
?- 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

The append/3 predicate

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

Constructing result in argument

*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.

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

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

Common Mistake 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.

```
del( _,  [],  [] ) .  
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`.



```
sublist(S, L) :- append(L1, L2, L), append(S, L3, L2).
```

Delete All Occurrences

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

```
delAll(_, [], []).  
delAll(X, [X|L], M) :- delAll(X, L, M). ;  
delAll(X, [Y|L], [Y|M]) :- X \= Y, delAll(X, L, M).
```

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**.

```
length([], 0).
```

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

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.
```

```
?- lengths([1,2,3,4],Len).  
Len = 0+1+1+1+1.
```

```
?- 
```

Meta-predicates

- `findall/3`, `setof/3`, and `bagof/3` are all *meta-predicates*

`findall(X,P,L)`

`setof(X,P,L)`

`bagof(X,P,L)`

All produce a list L of all the objects X such
that goal P is satisfied

- They all repeatedly call the goal P, instantiating the variable X within P and adding it to the list L
- They succeed when there are no more solutions

findall/3

- **findall/3** is the most straightforward of the three, and the most commonly used:

```
?- findall(X, member(X, [1,2,3,4]), Results).  
Results = [1,2,3,4]  
yes
```

- Solutions are listed in the result in the same order in which Prolog finds them
- If there are duplicated solutions, all are included. If there are infinitely-many solutions, it will never terminate!

findall/3

- The findall/3 can be used in more sophisticated ways
- The second argument, which is the goal, might be a compound goal:

```
?- findall(X, (member(X, [1,2,3,4]), X > 2), R).  
R = [3,4]?
```

Yes

- The first argument can be a term of any complexity:

```
?- findall(X/Y, (member(X, [1,2,3,4]), Y is X*X), R).  
R = [1/1, 2/4, 3/9, 4/16]?
```

yes

setof/3

- **setof/3** works very much like **findall/3**, except that:
 - It produces the **set** of all results, with any duplicates and the results **sorted**
 - If any variables are used in the goal, which do not appear in the first argument, **setof/3** will return a separate result for each possible instantiation of that variable:

```
age(peter, 7).  
age(ann, 5).  
age(pat, 8).  
age(tom, 5).  
age(ann, 5).  
..
```

```
?- setof(Child, age(Child, Age) , R).  
Age = 5,  
R = [ann, tom] ;  
Age = 7,  
R = [peter] ;  
Age = 8,  
R = [pat] ;
```

setof/3

- A *nested* call to setof/3 collects together the individual results:

```
?- setof(Age/Children, setof(Child, age(Child,Age) ,  
Children) , AllRes) .  
AllRes = [5/[ann,tom] ,7/[peter] ,8/[pat]] .
```

bagof/3

- **bagof/3** is very much like **setof/3** except:
 - that the list of results might contain duplicates
 - and isn't sorted

```
?- bagof(Child, age(Child,Age) ,Results) .  
Age = 5, Results = [tom,ann,ann]  
Age = 7, Results = [peter]  
Age = 8, Results = [pat] .
```

Prolog & Databases

Sailor database

```
boat(101, interlake, pink).  
boat(102, interlake, red).  
boat(103, clipper, green).  
boat(104, marine, red).  
reserves(21, 101, feb0122).  
reserves(21, 102, feb0522).  
reserves(21, 103, feb1622).  
reserves(21, 104, jan1222).  
reserves(21, 104, feb1122).  
reserves(31, 101, jan0722).  
reserves(31, 103, feb0822).  
reserves(31, 104, feb2122).  
reserves(64, 104, jan1222).  
reserves(64, 102, dec1221).  
reserves(74, 103, dec2221).  
sailor(21, dustin, 7, 45).  
sailor(29, brutus, 1, 31).  
sailor(31, lubber, 8, 56).  
sailor(32, andy, 8, 25).  
sailor(58, rusty, 10, 35).  
sailor(64, horatio, 7, 35).  
sailor(74, horatio, 9, 38).  
sailor(99, art, 3, 25).
```

`sailor(sid, sname, rating, age)`

`boat(bid, bname, color)`

`reserves(sid, bid, date)`

- Count the number of sailors 35 and older.

```
countOld(List,N) :- setof(Sname, oldguys(Sname), List), length(List,N).  
  
oldguys(Name) :- sailor(Name,Age), Age >= 35.
```

```
?- countOld(L,N).  
L = [dustin, horatio, lubber, rusty],  
N = 4.
```

```
countOldB(List,N) :- bagof(Sname, oldguys(Sname), List), length(List,N).
```

```
?- countOldB(L,N).  
L = [dustin, lubber, rusty, horatio, horatio],  
N = 5.
```

sailor(sid, sname, rating, age)

boat(bid, bname, color)

reserves(sid, bid, date)

- Reservations by Sailor

```
reserveBySailor(Sname, List, N) :- setof(Bname, nameboat(Sname, Bname), List),
                                         , length(List, N).
```

```
?- reserveBySailor(lubber,L,N).
L = [clipper, interlake, marine],
N = 3.
```

`sailor(sid, sname, rating, age)`

`boat(bid, bname, color)`

`reserves(sid, bid, date)`

- Reservations by all Sailors

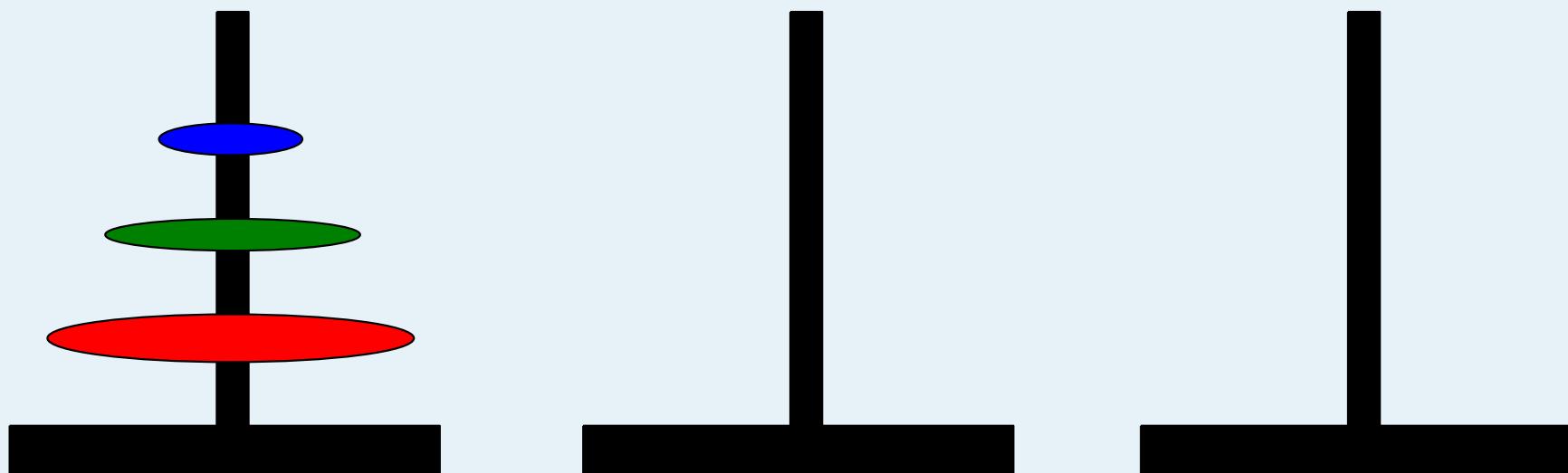
```
reservesByAll(List,N) :- setof(Sname/BList, setof(Boat, nameboat(Sname,Boat), BList), List),
                           length(List, N).
```

```
?- reservesByAll(List,N).
List = [art/[interlake], dustin/[clipper, interlake, marine], horatio/[clipper, interlake, marine], lubber/[clipper, interlake, marine]], N = 4.
```

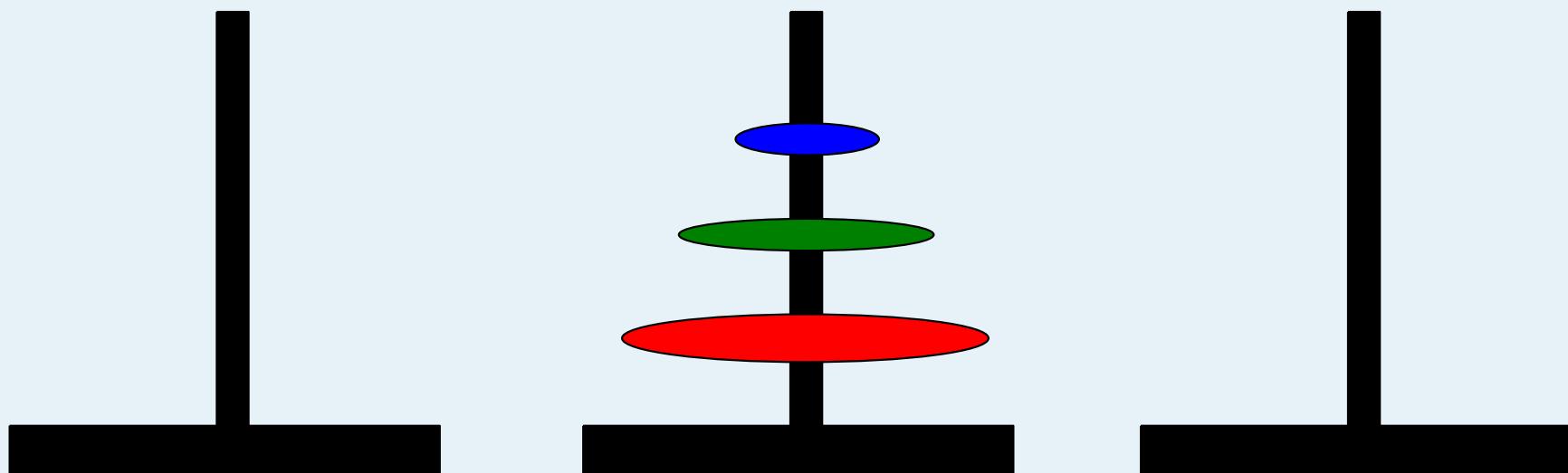
Tower of Hanoi

- There are three towers
- N gold disks, with decreasing sizes, placed on the first tower
- You need to move all of the disks from the first tower to the last tower
- Larger disks can not be placed on top of smaller disks
- The third tower can be used to temporarily hold disks

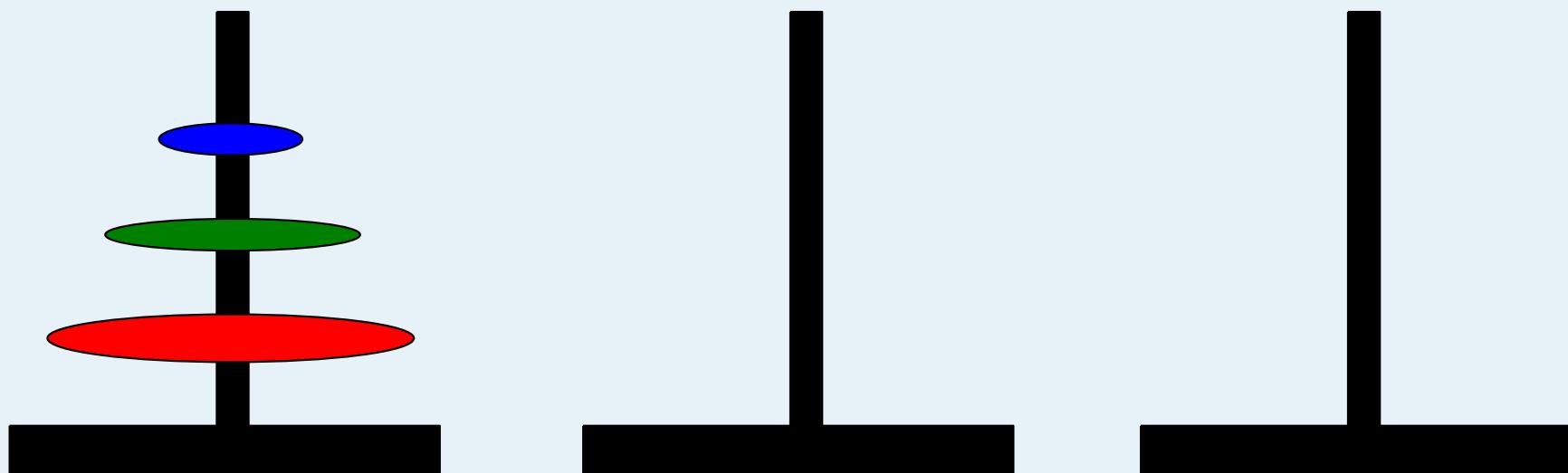
Tower of Hanoi - Start



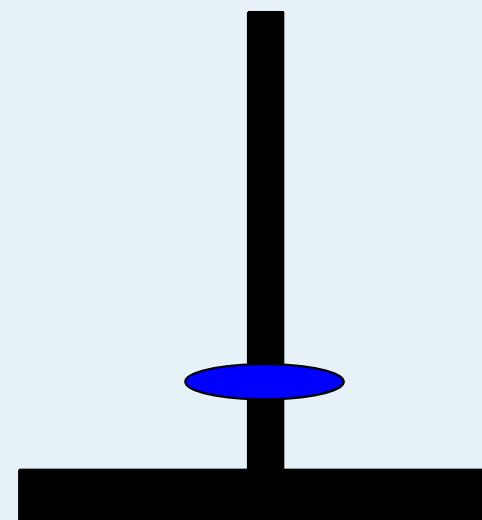
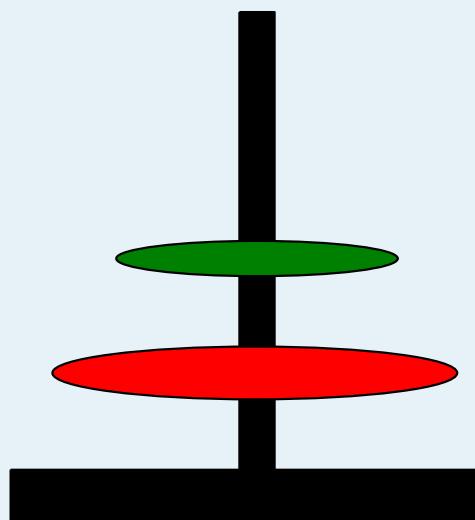
Recursive Solution



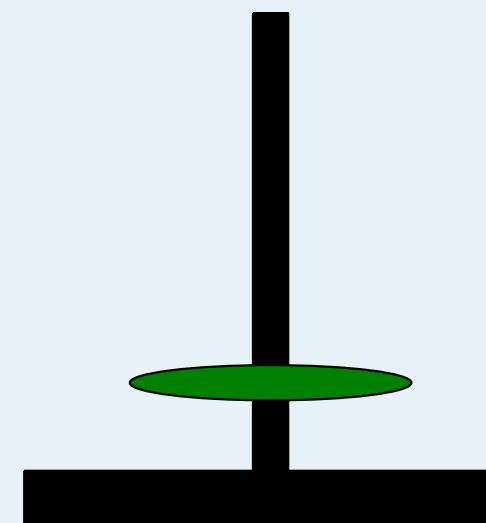
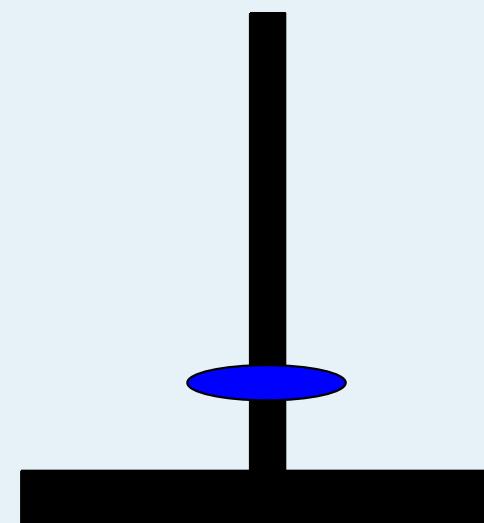
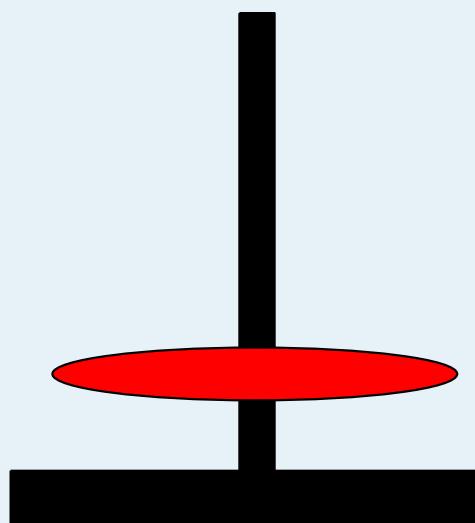
Tower of Hanoi



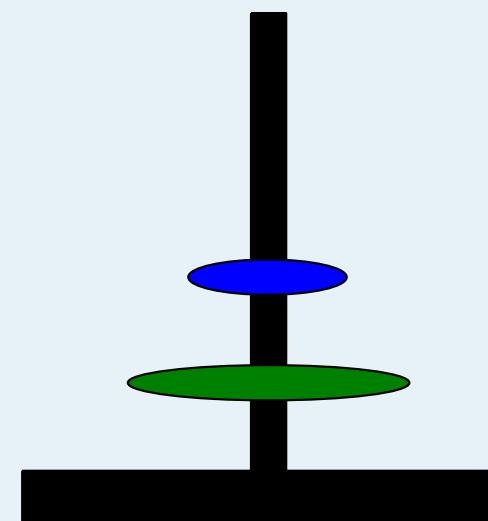
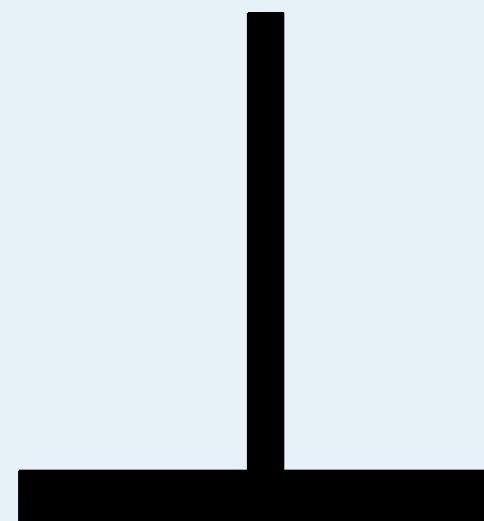
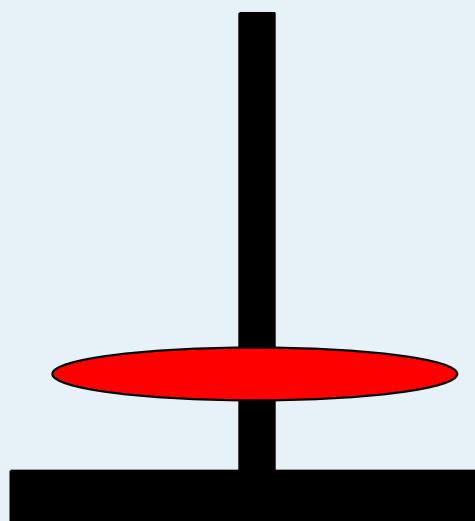
Tower of Hanoi



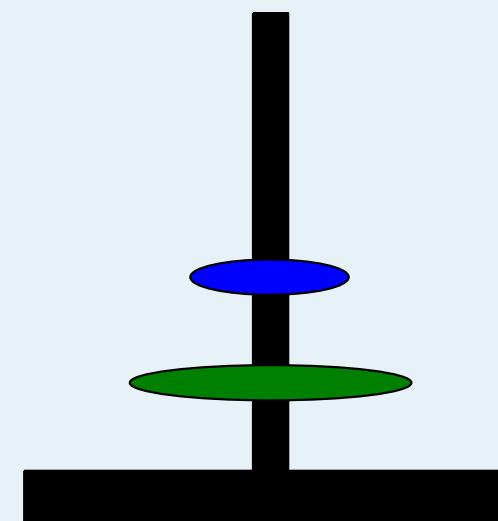
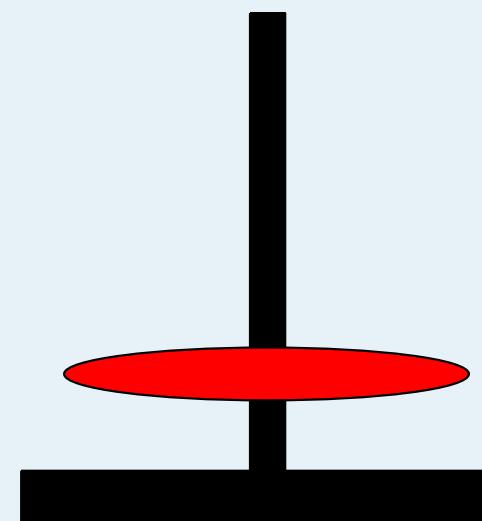
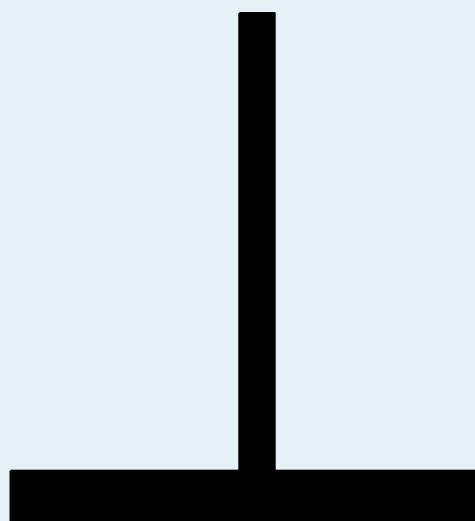
Tower of Hanoi



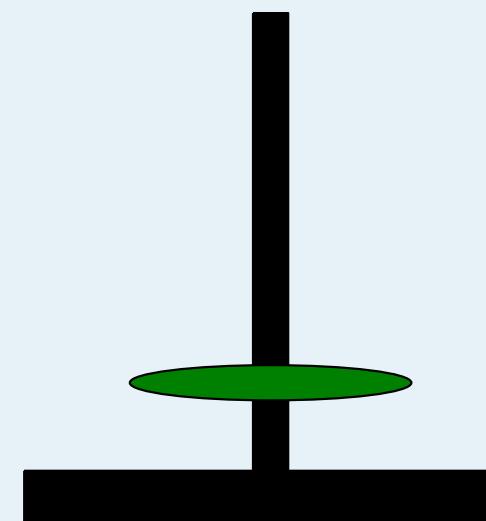
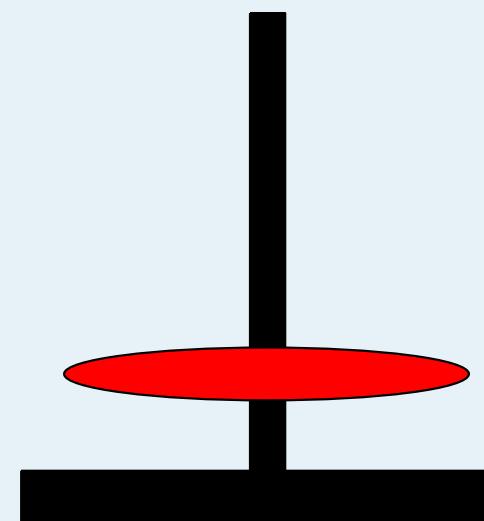
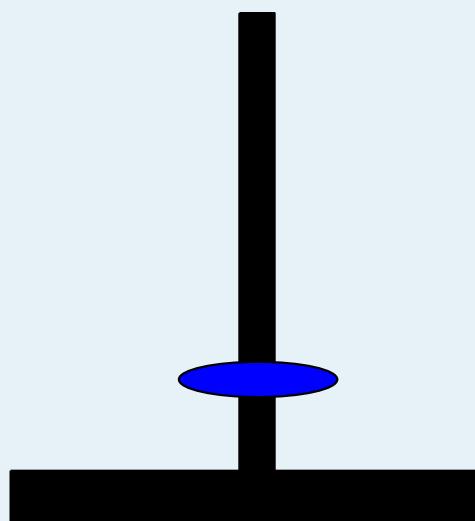
Tower of Hanoi



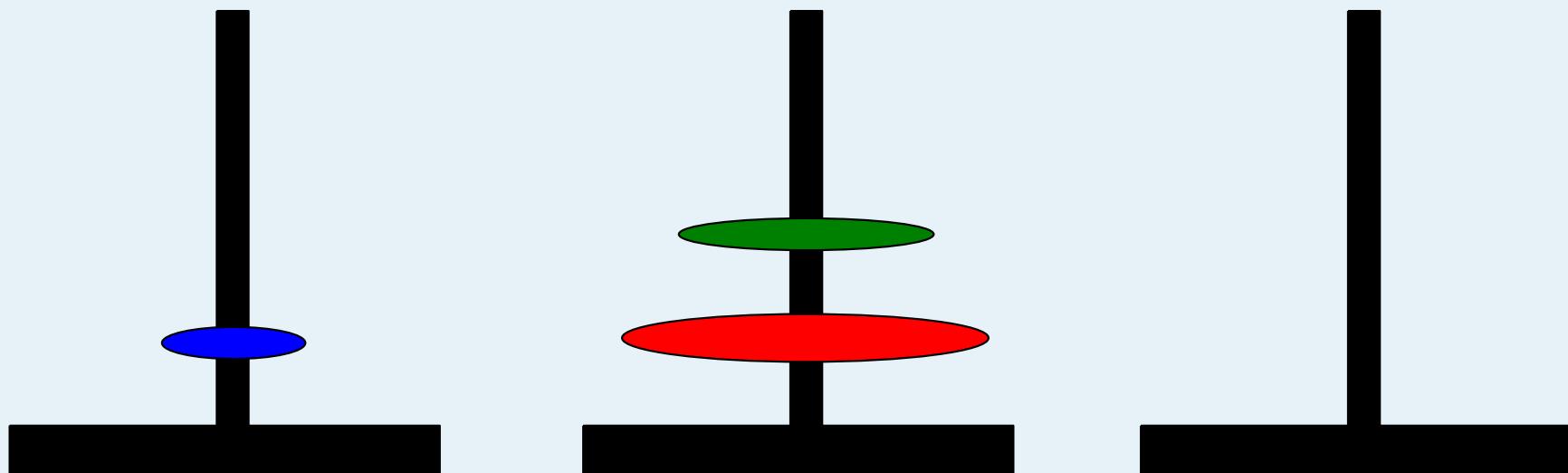
Tower of Hanoi



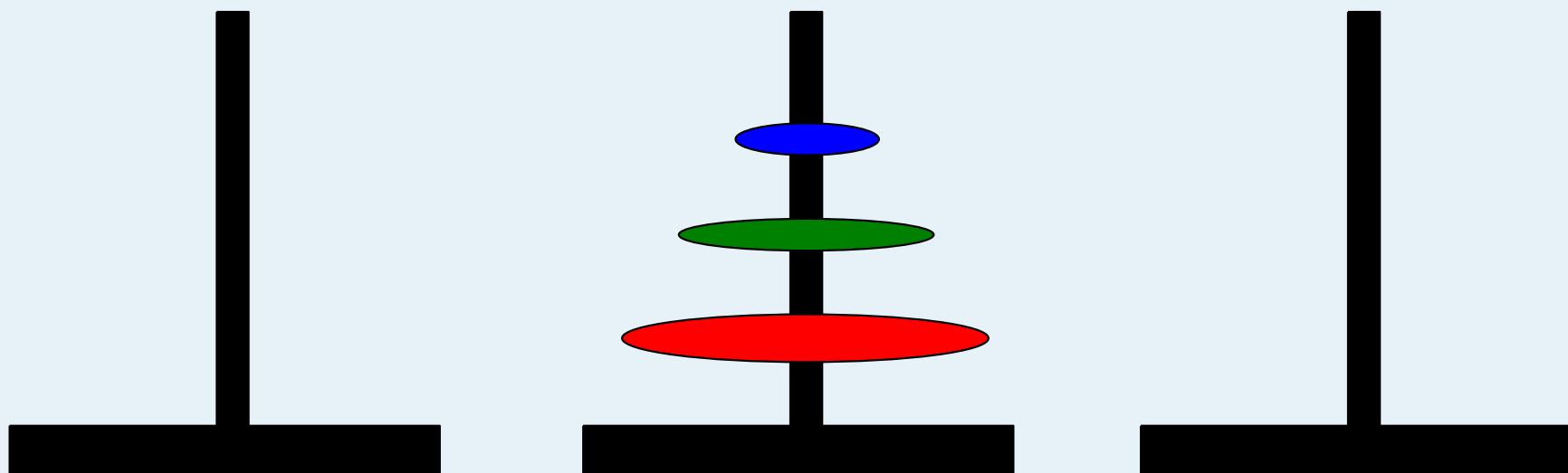
Tower of Hanoi



Tower of Hanoi



Tower of Hanoi



Towers of Hanoi - Prolog

```
hanoi(N) :- move(N, left, center, right).  
move(0, _, _, _) :- !.  
move(N, A, B, C) :- M is N-1,  
    move(M, A, C, B), inform(A, B), move(M, C, B, A).  
inform(X, Y) :- write([move, disk, from, X, to, Y]), nl.
```

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].
```

Negation Predicate

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

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

If P is true, then $\text{not}(P)$ is false

Otherwise, $\text{not}(P)$ is true

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

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

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

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

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

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`)

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 $\backslash=$

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

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

Terms & Lists

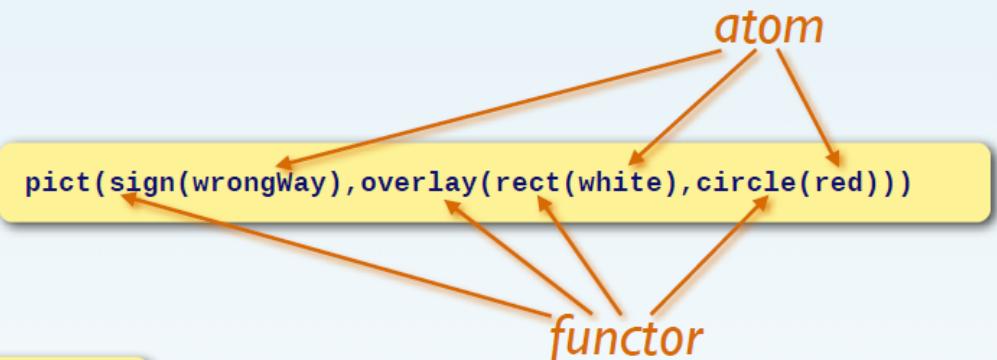
Terms denote structured objects

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

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

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

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



Variables **cannot** be used
to match functors

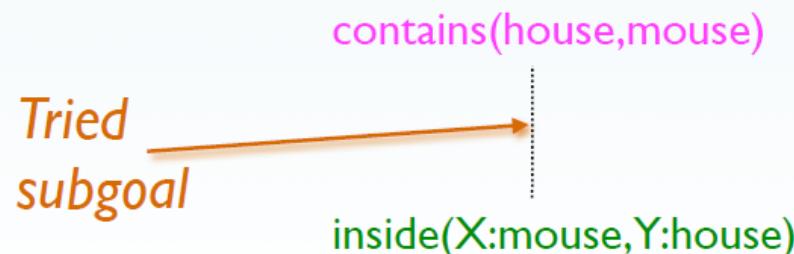
?- owns(john,X(blue)).

Tree Model

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

(I) 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 " \rightarrow " 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)
 \Rightarrow goal fails and is revoked.

`contains(house,mouse)`

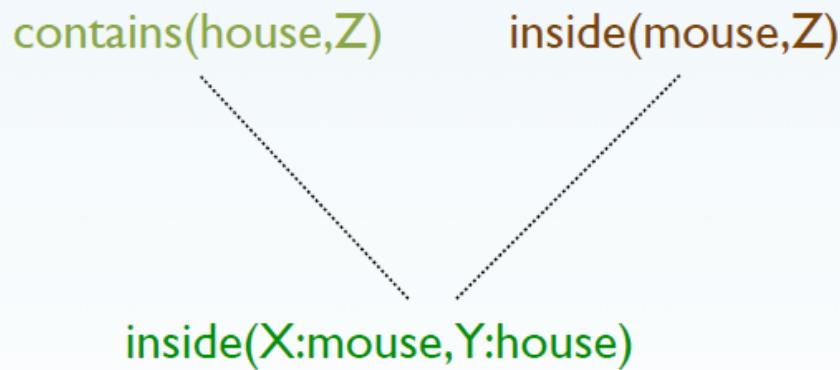


`inside(X:mouse,Y:house)`

```
→ 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

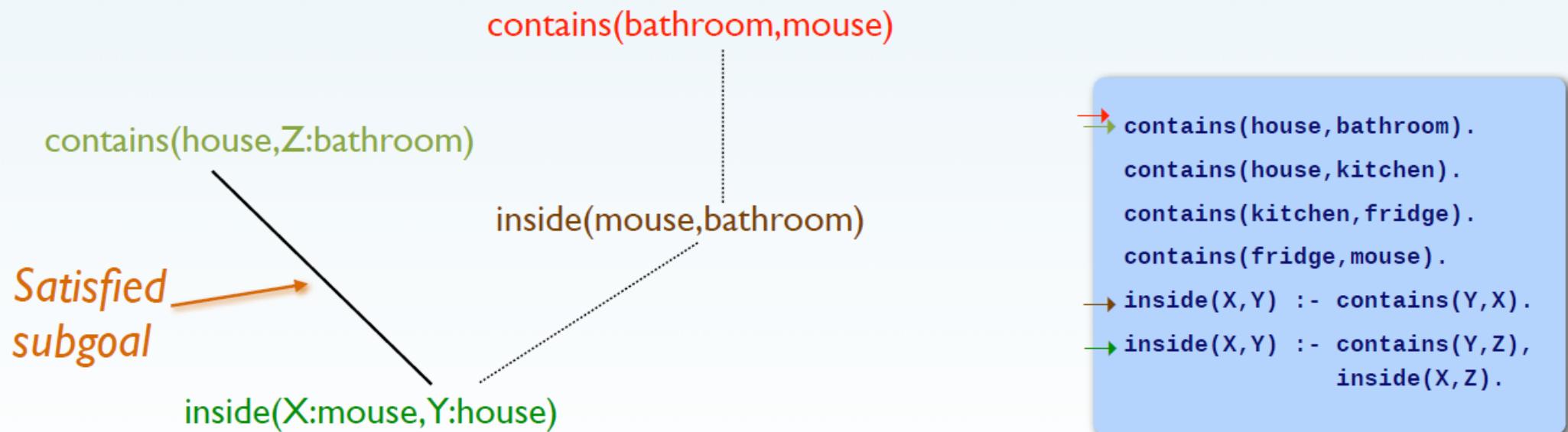
(3) Marker " \rightarrow " for goal `inside(mouse,house)` is advanced
⇒ create new subgoals `contains(house,Z)` and `inside(mouse,Z)`
together with two new markers.



```
→ 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).
```

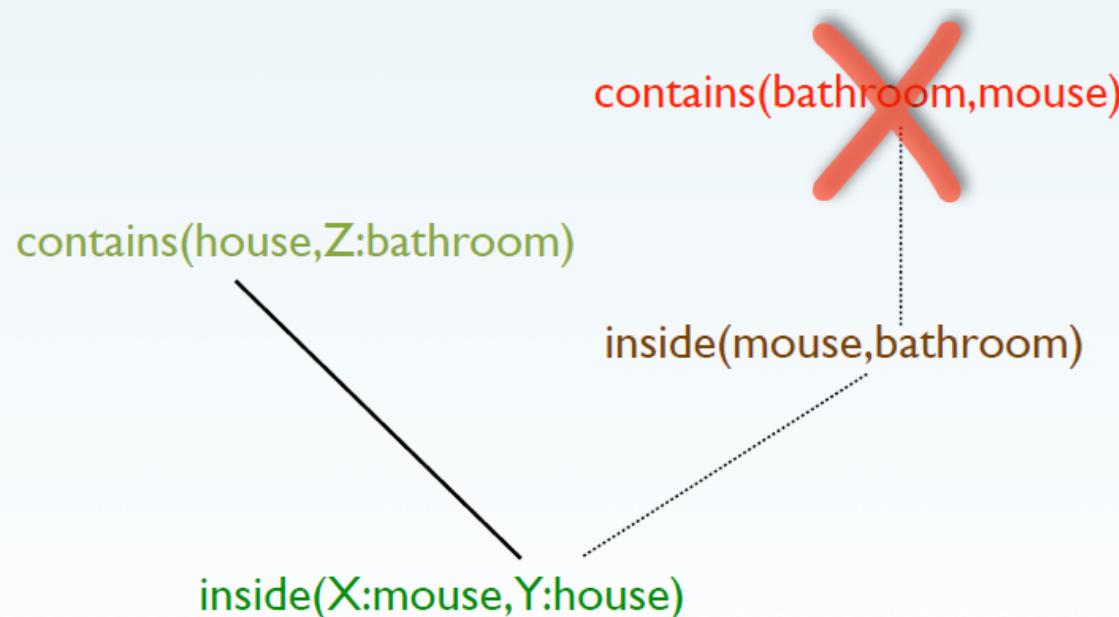
Lots of Markers ...

(4) Z is bound to bathroom, which causes the "contains" goal to succeed.
The goal $\text{inside}(\text{mouse}, \text{bathroom})$ is expanded to
 $\text{contains}(\text{bathroom}, \text{mouse})$ (plus new marker " \rightarrow ").



Lots of Markers ...

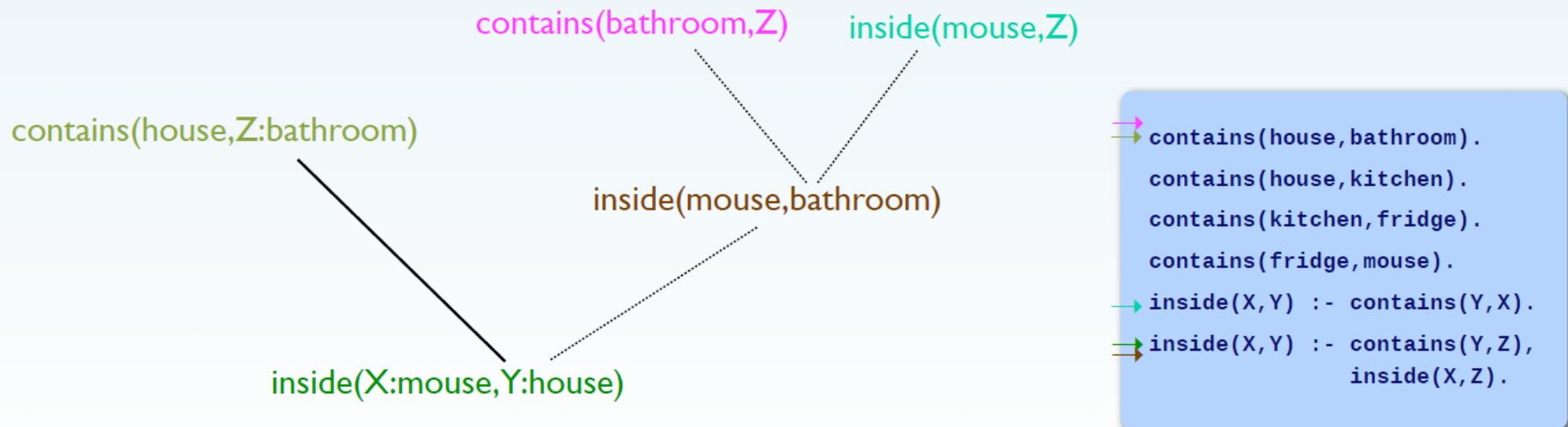
(5) $\text{contains}(\text{bathroom}, \text{mouse})$ cannot be satisfied and must be revoked;
marker " \rightarrow " for goal $\text{inside}(\text{mouse}, \text{bathroom})$ is advanced.



```
→ 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).
```

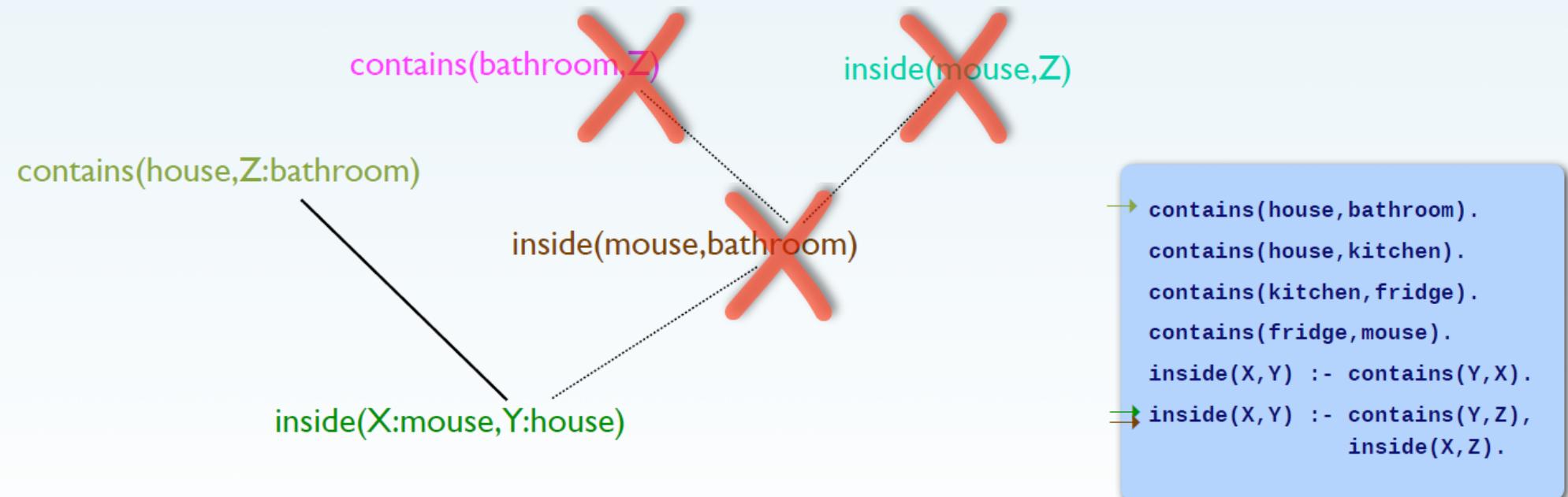
Even more markers ...

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



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 " \rightarrow " pointer.



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.

