CS 456

Programming Languages
Fall 2024

Week 1

Introduction, Functional Programming, OCaml,Datatypes

Administrivia

Who:

Instructor: Suresh Jagannathan
Office Hours: Tu,Th, 12pm - 1pm (LWSN 3154J)

UTA: Priyam Gupta
gupta751@purdue.edu

Where: BHEE 236

When: August 20 - December 5, 2024

Discussion Board: Piazza
Homeworks: Brightspace and Gradescope

Grading

Quizzes (5%)

- Mostly weekly autograded multiple-choice via Gradescope

Homeworks (35%)

- Approximately 8 over the course of the semester
- Typically 1.5 weeks to complete
- Involves programming (OCaml) and proving (Dafny)

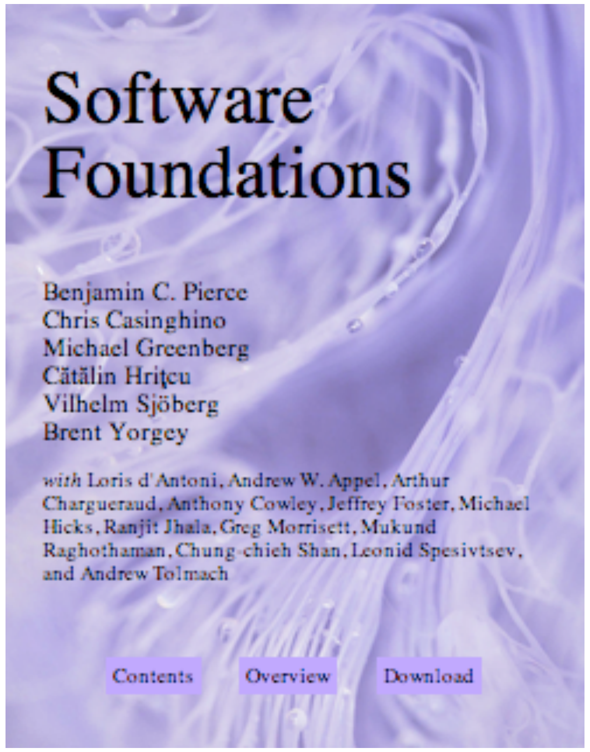
Midterm (25%)

- In-class
- October 17

Final (35%)

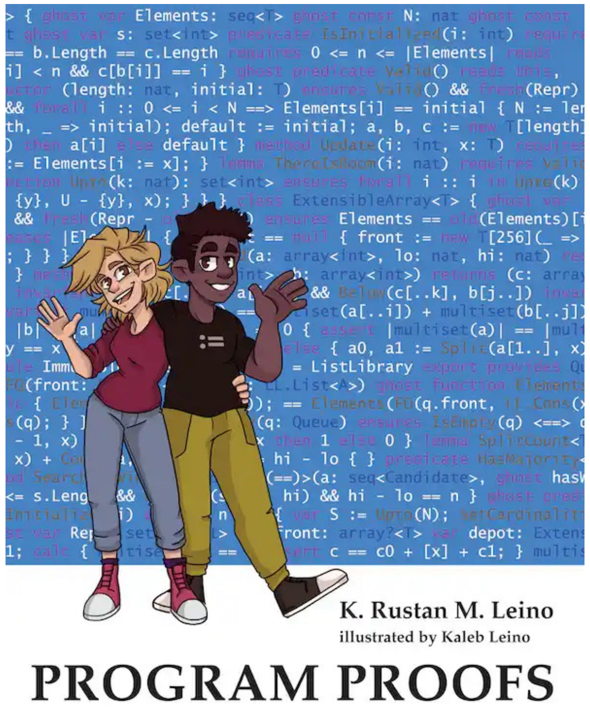
- Cumulative

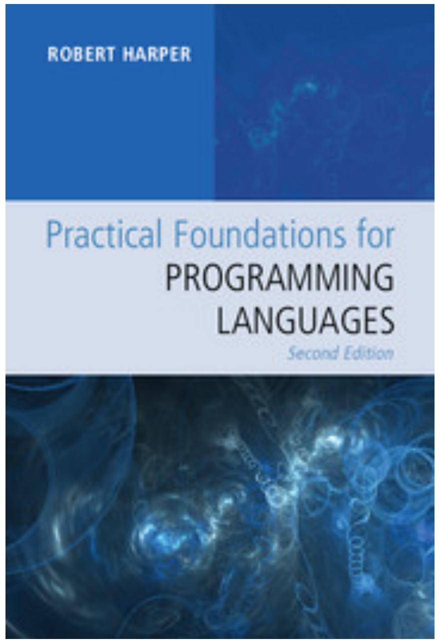
Textbooks (none required)



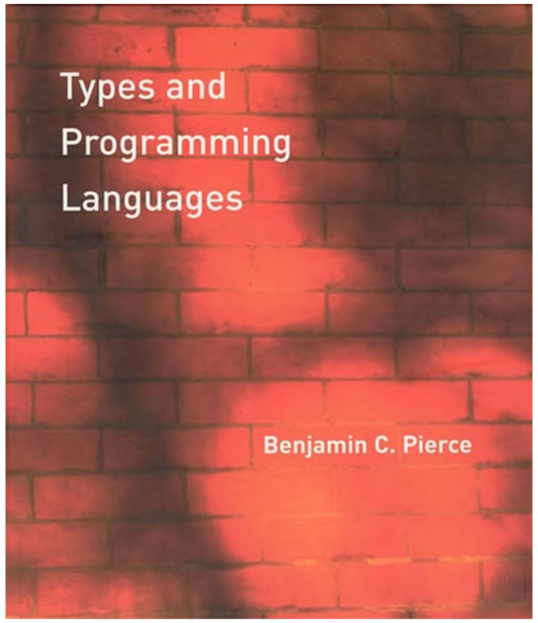
Software Foundations

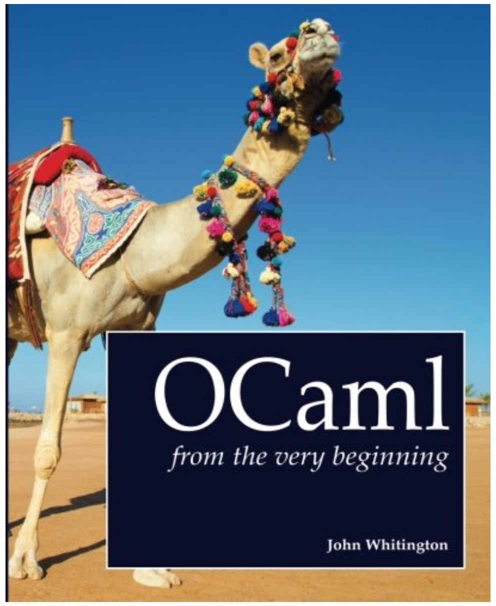
Program Proofs





Practical Foundations of Programming Languages





OCaml from the Very Beginning

Types and Programming Languages

How

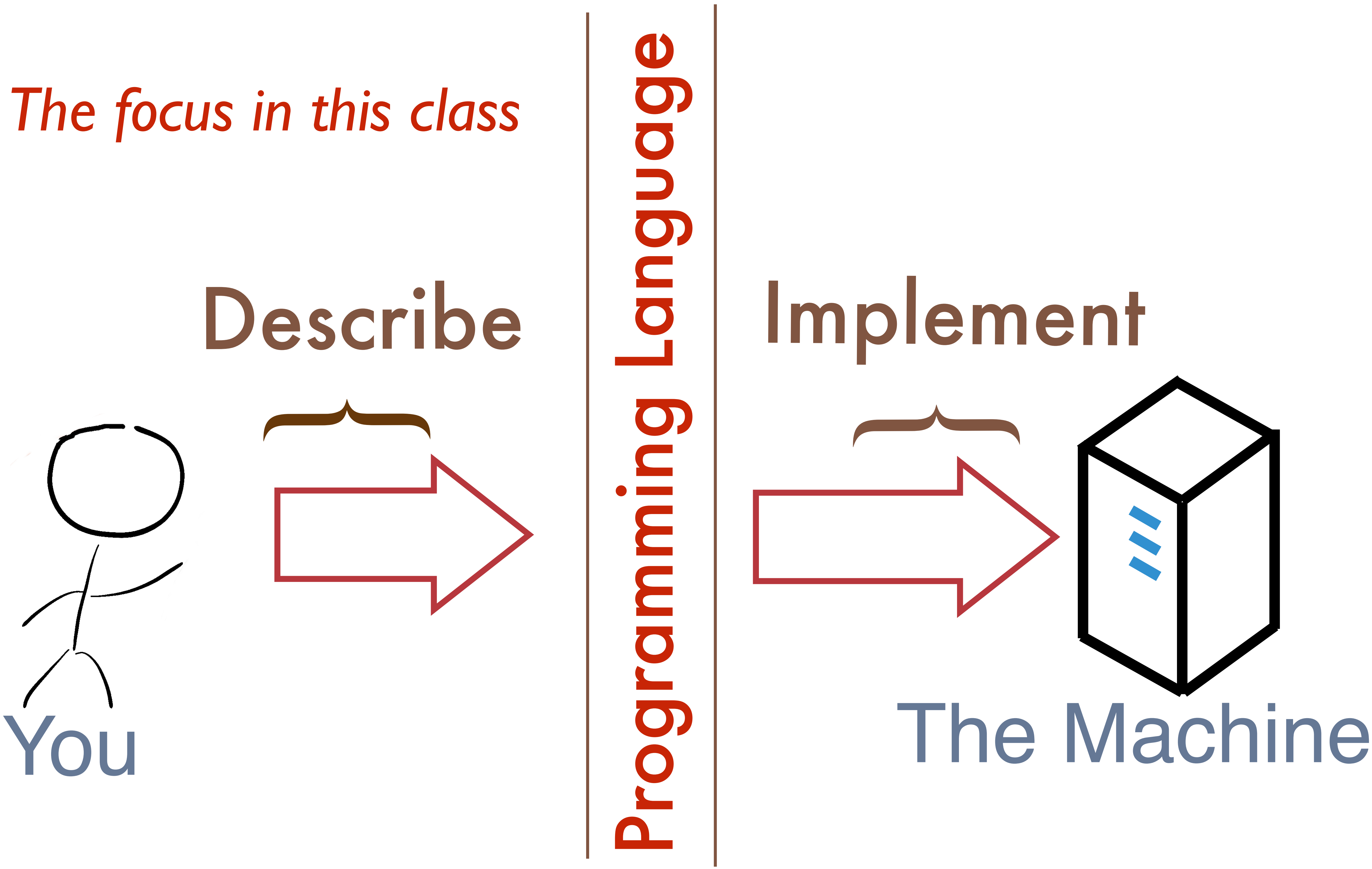
to succeed
in CS 456

Should be familiar with:

- Programming in a high-level language
  (Python, Java, Rust, Haskell, OCaml, ...)
- Basic logic and proofs techniques
  sets, relations, functions, ...
- Basic data structures and algorithms

Participate!

What


The focus in this class
Describe
Programming Language
Implement
You
The Machine

What

Functional Programming Language

- ★ Write interpreters to exercise various PL concepts related to data abstraction, control-flow, and types
- ★ Web Page: ocaml.org

Verifier-Aware Programming Language

- ★ Write programs along with specifications that are automatically verified
- ★ Web Page: dafny.org

Why?

- ★ Develop a more sophisticated appreciation of programs, their structure, and design
  - Judge, distinguish, and relate different language features
  - Define and prove formal claims about a program's (or programming language's) meaning
  - Develop sound intuitions to better judge language properties
  - Devise expressive, interpretable, and useful ways to specify what a program should do without having to say how it does it
- ★ Develop tools to be better programmers, designers, and computer scientists

Why Not?

- ★ An introduction to advanced programming techniques
- ★ Discussion of machine implementations
  - Not motivated from the perspective of a compiler writer
  - Impact of language design decisions on implementation tractability will be considered when appropriate
- ★ Survey of different languages

What

Foundations:

- Functional Programming
- State and Control
- Types

Program Semantics:

- Operational Semantics
- Denotational Semantics

Automated Program Verification

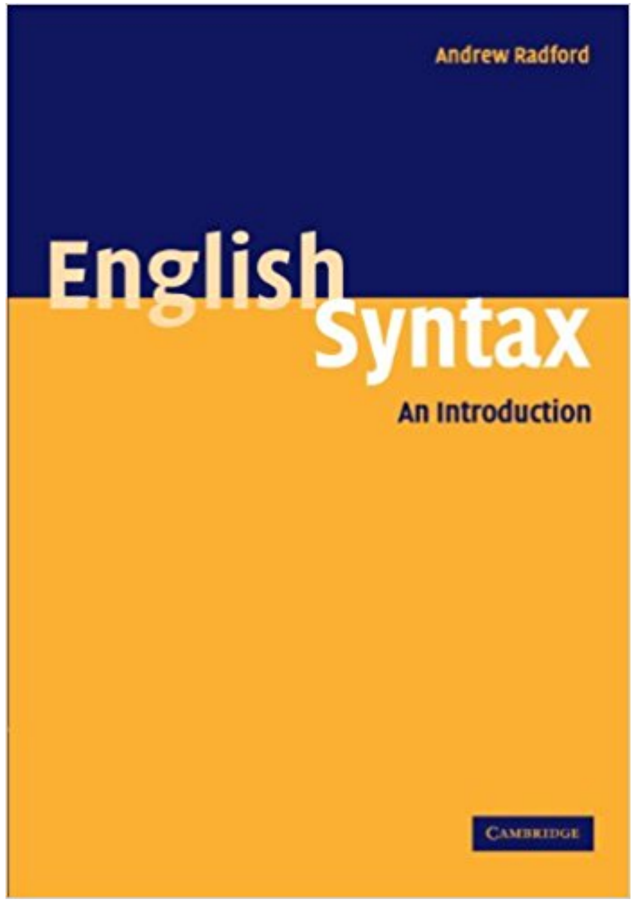
- Hoare Logic and Axiomatic Semantics
- Verification-Aware Languages

Defining a Language

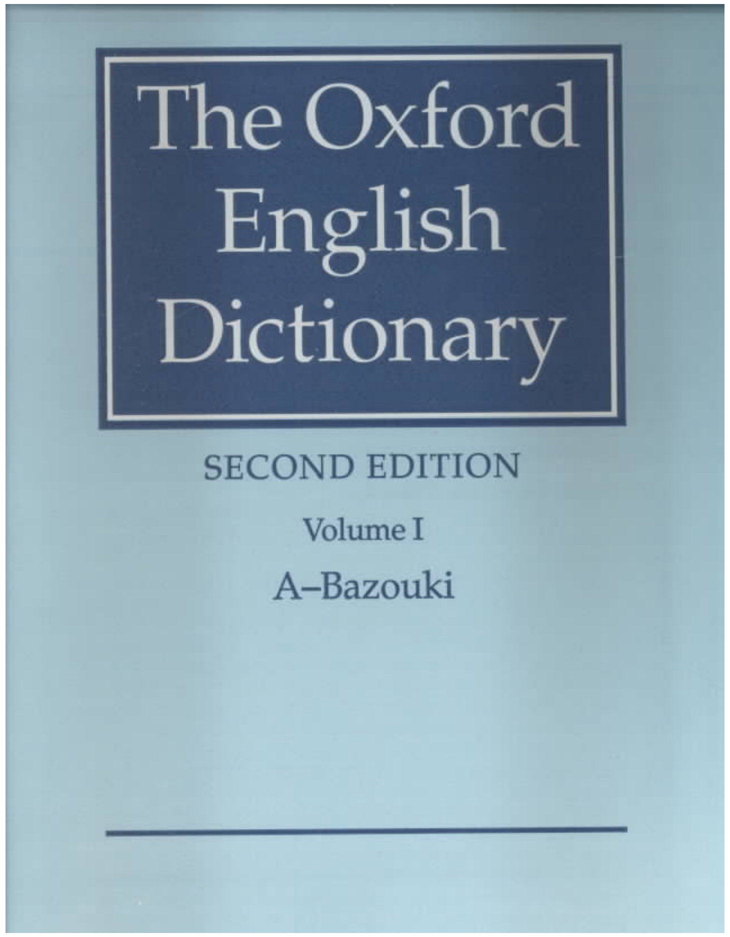
- A "recipe" for defining a language:
  1. Syntax:
     - What are the valid expressions?
  2. Semantics (Dynamic Semantics):
     - What is the meaning of valid expressions?
  3. Sanity Checks (Static Semantics):
     - What expressions have meaningful evaluations?

Defining English

1. Syntax:



2. Semantics:



Defining A Programming Language

1.Syntax

atexp	::=	*scon*	special constant
		$\langle$`op`$\rangle$*longvid*	value identifier
		`{` $\langle$*exprow*$\rangle$ `}`	record
		`let` *dec* `in` *exp* `end`	local declaration
		`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* $\langle$ `,` *exprow*$\rangle$	expression row
exp	::=	*atexp*	atomic
		exp atexp	application (L)
		exp_1 *vid* exp_2	infixed application
		exp `:` *ty*	typed (L)
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`fn` *match*	function
match	::=	*mrule* $\langle$ `|` *match*$\rangle$	
mrule	::=	*pat* `=>` *exp*	
dec	::=	`val` *tyvarseq valbind*	value declaration
		`type` *typbind*	type declaration
		`datatype` *datbind*	datatype declaratic
		`datatype` *tycon* `-=-` `datatype` *longtycon*	datatype replicatio:
		`abstype` *datbind* `with` *dec* `end`	abstype declaration
		`exception` *exbind*	exception declarati
		`local` dec_1 `in` dec_2 `end`	local declaration
		`open` $longstrid_1 \cdots longstrid_n$	open declaration (r
			empty declaration
		dec_1 $\langle$;$\rangle$ dec_2	sequential declarati
		`infix` $\langle d\rangle$ $vid_1 \cdots vid_n$	infix (L) directive
		`infixr` $\langle d\rangle$ $vid_1 \cdots vid_n$	infix (R) directive
		`nonfix` $vid_1 \cdots vid_n$	nonfix directive
valbind	::=	*pat* `=` *exp* $\langle$`and` *valbind*$\rangle$	
		`rec` *valbind*	
typbind	::=	*tyvarseq tycon* `=` *ty* $\langle$`and` *typbind*$\rangle$	
datbind	::=	*tyvarseq tycon* `=` *conbind* $\langle$`and` *datbind*$\rangle$	
conbind	::=	$\langle$`op`$\rangle$*vid* $\langle$`of` *ty*$\rangle$ $\langle$ `|` *conbind*$\rangle$	
exbind	::=	$\langle$`op`$\rangle$*vid* $\langle$`of` *ty*$\rangle$ $\langle$`and` *exbind*$\rangle$	
		$\langle$`op`$\rangle$*vid* `=` $\langle$`op`$\rangle$*longvid* $\langle$`and` *exbind*$\rangle$	

Figure 4: Grammar: Expressions, Matches, Declarations and Bindings

2.Semantics

$$\frac{E \vdash atexp \Rightarrow v}{E \vdash atexp \Rightarrow v} \quad (96)$$

$$\frac{E \vdash exp \Rightarrow vid \qquad vid \neq \texttt{ref} \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (vid, v)} \quad (97)$$

$$\frac{E \vdash exp \Rightarrow en \qquad E \vdash atexp \Rightarrow v}{E \vdash exp\ atexp \Rightarrow (en, v)} \quad (98)$$

$$\frac{s, E \vdash exp \Rightarrow \texttt{ref}\ , s' \qquad s', E \vdash atexp \Rightarrow v, s'' \qquad a \notin \mathrm{Dom}(mem \text{ of } s'')}{s, E \vdash exp\ atexp \Rightarrow a,\ s'' + \{a \mapsto v\}} \quad (99)$$

$$\frac{s, E \vdash exp \Rightarrow\ \texttt{:=}\ , s' \qquad s', E \vdash atexp \Rightarrow \{1 \mapsto a,\ 2 \mapsto v\}, s''}{s, E \vdash exp\ atexp \Rightarrow \{\} \text{ in Val},\ s'' + \{a \mapsto v\}} \quad (100)$$

$$\frac{E \vdash exp \Rightarrow b \qquad E \vdash atexp \Rightarrow v \qquad \mathrm{APPLY}(b, v) = v'/p}{E \vdash exp\ atexp \Rightarrow v'/p} \quad (101)$$

$$\frac{\begin{array}{c} E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \mathrm{Rec}\, VE,\ v \vdash match \Rightarrow v' \end{array}}{E \vdash exp\ atexp \Rightarrow v'} \quad (102)$$

$$\frac{\begin{array}{c} E \vdash exp \Rightarrow (match, E', VE) \qquad E \vdash atexp \Rightarrow v \\ E' + \mathrm{Rec}\, VE,\ v \vdash match \Rightarrow \mathrm{FAIL} \end{array}}{E \vdash exp\ atexp \Rightarrow [\texttt{Match}]} \quad (103)$$

$$\frac{E \vdash exp \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \quad (104)$$

$$\frac{E \vdash exp \Rightarrow [e] \qquad E, e \vdash match \Rightarrow v}{E \vdash exp\ \texttt{handle}\ match \Rightarrow v} \quad (105)$$

$$\frac{E \vdash exp \Rightarrow [e] \qquad E, e \vdash match \Rightarrow \mathrm{FAIL}}{E \vdash exp\ \texttt{handle}\ match \Rightarrow [e]} \quad (106)$$

$$\frac{E \vdash exp \Rightarrow e}{E \vdash \texttt{raise}\ exp \Rightarrow [e]} \quad (107)$$

$$\frac{}{E \vdash \texttt{fn}\ match \Rightarrow (match, E, \{\})} \quad (108)$$

Syntax

(of Arithmetic + Boolean Expressions)

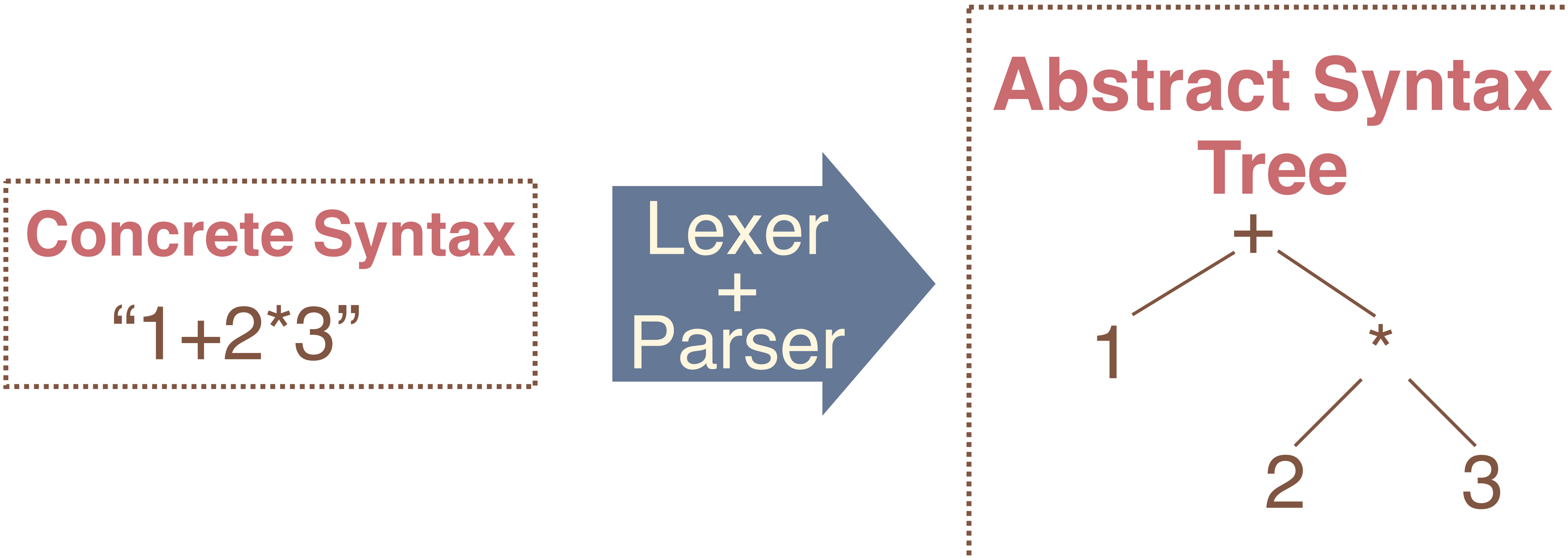
Backus-Naur Form (BNF) Definitions:

```
A ::= ℕ
    | A + A
    | A – A
    | A * A
```

```
B ::= true
    | false
    | A = A
    | A ≤ A
    | not B
    | B and B
```

Abstract Syntax

(of Arithmetic + Boolean Expressions)



Programs as Data

A ::=
  | $\mathbb{N}$
  | A + A
  | A * A
  | - A

```
type aexp =
    Const of int
  | Plus  of (aexp * aexp)
  | Times of (aexp * aexp)
  | Neg of aexp

    (* Can you write down
        (1 + 2) * (- 0)
      as an aexp?  *)

Times (Plus (Const 1)
            (Const 2))
      (Neg (Const 0))
```

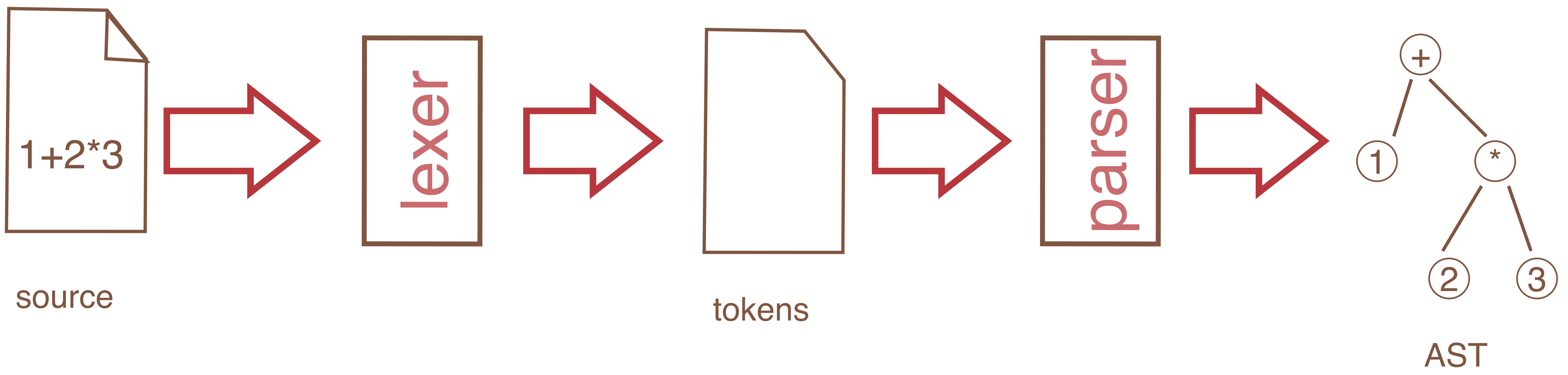
Abstract Syntax

Programs as Data

- This implementation strategy is a **deep embedding** of the source language
  - ASTs are encoded as data types in the host language
  - Programs are values of this type, and can be manipulated and examined within the host language

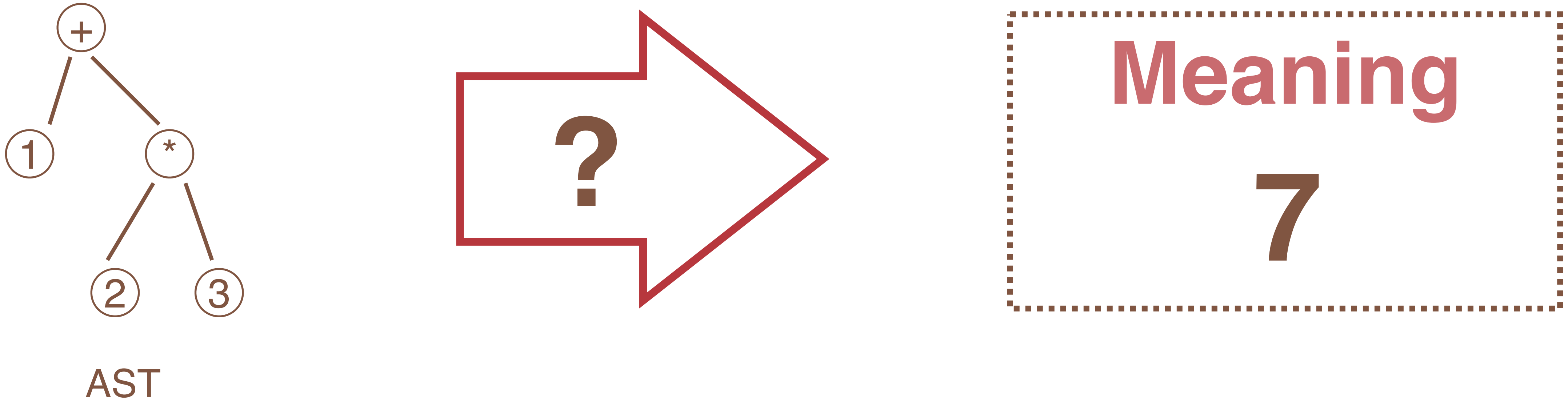
```
type aexp =
   Const of int
 | Plus of (exp * exp)
 | Times of (exp * exp)
 | Neg of exp
```

Semantics



What's the meaning of the expression "1+2*3"?

Semantics



Semantics

- One way to assign meaning is through *evaluation*

```
aeval:  aexp -> int

let aeval = function
  | Const i -> i
  | Plus (a1,a2) ->(aeval a1) + (aeval a2)
  | Times (a1,a2) -> (aeval a1) * (aeval a2)
  | Neg a1 ->  - (aeval a1)
```

Desiderata

Growing a Language

Guy L. Steele Jr.

Sun Microsystems Laboratories
1 Network Drive
Burlington, Massachusetts 01803

guy.steele@sun.com

October 1998

[This is the text of a talk I once gave, but with a few bugs fixed here and there, and a phrase or two changed to make my thoughts more clear. The talk as I first gave it can be had on tape [12].]

I think you know what a man is. A *woman* is more or less like a man, but not of the same sex. (This may seem like a strange thing for me to start with, but soon you will see why.)

Next, I shall say that a *person* is a woman or a man (young or old).

To keep things short, when I say "he" I mean "he or she," and when I say "his" I mean "his or her."

A *machine* is a thing that can do a task with no help, or not much help, from a person.

(As a rule, we can speak of two or more of a thing if we add an "s" or "z" sound to the end of a word that names it.)

⟨noun⟩ ::= ⟨noun that names one thing⟩ "s"
| ⟨noun that names one thing⟩ "es"

These are names of persons: *Alan Turing*, *Alonzo Church*, *Charles Kay Ogden*, *Christopher Alexander*, *Eric Raymond*, *Fred Brooks*, *John Horton Conway*, *James Gosling*, *Bill Joy*, and *Dick Gabriel*.

The word *other* means "not the same." The phrase *other than* means "not the same as."

A *number* may be nought, or may be one more than a number. In this way we have a set of numbers with no bound.

⟨number⟩ ::= `0`
| `1 +` ⟨number⟩

There are other numbers as well, but I shall not speak more of them yet.

These numbers—nought or one more than a number—can be used to count things. We can add two numbers if we count up from the first number while we count down from the number that is not the first till it comes to nought; then the first count is the sum.

$$4 + 2 = 5 + 1 = 6 + 0 = 6$$

Four plus two is the same as five plus one, which is the same as six plus nought, which is six.

We shall take the word *many* to mean "more than two in number."

Functional Programming

We'll start our investigation by considering a small functional language

- These languages tend to have a small core set of features
- Datatypes, functions, and their application
- Written in OCaml

```
> let double (n : int) : int = n + n;
val double : int -> int = <fun>
```

Functions

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume values, produce values

```
> let double (n : int) : int = n + n;
val double : int -> int = <fun>

> double 1;
- : int = 2
```

Functions

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume values, produce values

```
> let rec concat (s : string list) : string =
      match s with
      | [] -> ""
      | s1 :: s2 -> s1 ^ (concat s2);
val concat : string list -> string = <fun>

> concat [“Hello" ;” " ;”World”];
- : string = "Hello World"
```

Functions

- Functional languages tend to have a small core
- Standard libraries tend to have the usual suspects
- Functions are **applied** to arguments
- Functions are **pure**: consume value, produce value
- OCaml can automatically infer many type annotations

```
> let rec concat s  =
      match s with
      | [] -> ""
      | s1 :: s2 -> s1 ^ (concat s2);
val concat : string list -> string = <fun>

> concat [“Hello" ; " “ ; ”World”];
- : string = "Hello World"
```

What about:

```
let rec repeat x n =
match n with
| 0 -> []
| m -> x :: (repeat x (m - 1))
```

Building Blocks

Given the following ingredients:

- bool: a datatype for booleans

Define a Boolean equality function in terms of

- andb: logical and
- orb: logical or
- negb: logical negation

```
> let eqb =
    let andb b1 b2 = if b1 then b2 else false in
    let orb  b1 b2 = if b1 then true else if b2 then true else false in
    let negb b1 = if b1 then false else true in
    fun (b1,b2)  -> orb (andb b1 b2) (andb (negb b1) (negb b2));
val eqb : bool * bool -> bool = <fun>
```

Algebraic Data Types

- Enumerated types are the simplest data types in Coq
- Type annotations can be inferred here
- Constructors describe how to **introduce** a value of a type

```
type mybool = True | False;

type weekdays =
  Monday | Tuesday | Wednesday | Thursday | Friday;
```

Pattern Matching

- Pattern matching lets a program use values of a type
- Patterns are expected to be exhaustive

```
> let negb b =
    match b with
    | True -> False
    | False -> True
val negb : mybool -> mybool = <fun>
```

Pattern Matching

- Pattern matching lets a program use values of a type
- Patterns are expected to be exhaustive
- Use underscore (_) as wildcards

```
let eqb b1 b2 =
  match b1, b2 with
  | true, true -> true
  | false, false -> true
  | false, true -> false
  | true, _ -> false
```

Compound ADTs

- Can build new ADTs from existing ones:
    - A color is either black, white, or a primary color
    - Need to apply primary to something of type rgb:
- ADTs are **algebraic** because they are built from a small set of operators (sums of product).

```
> type rgb = Red | Green | Blue;

> type color =  Black | White | Primary of rgb;

> Primary Red;
- : color = Primary Red
```

Pattern Matching²

- Patterns on compound types need to mention arguments
    - Can be a **variable**

```
let monochrome (c : color) : bool :=
  match c with
  | Black -> true
  | White -> true
  | Primary p -> false  (* could have also used a wildcard *)
```

Concept Check

- Define a type for the 'basic' (h, a, and p) html tags:
  - A header should include a nat indicating its importance
  - The anchor tag should include a string for its destination
  - The paragraph doesn't need anything extra
- Define a pretty printer for opening a tag

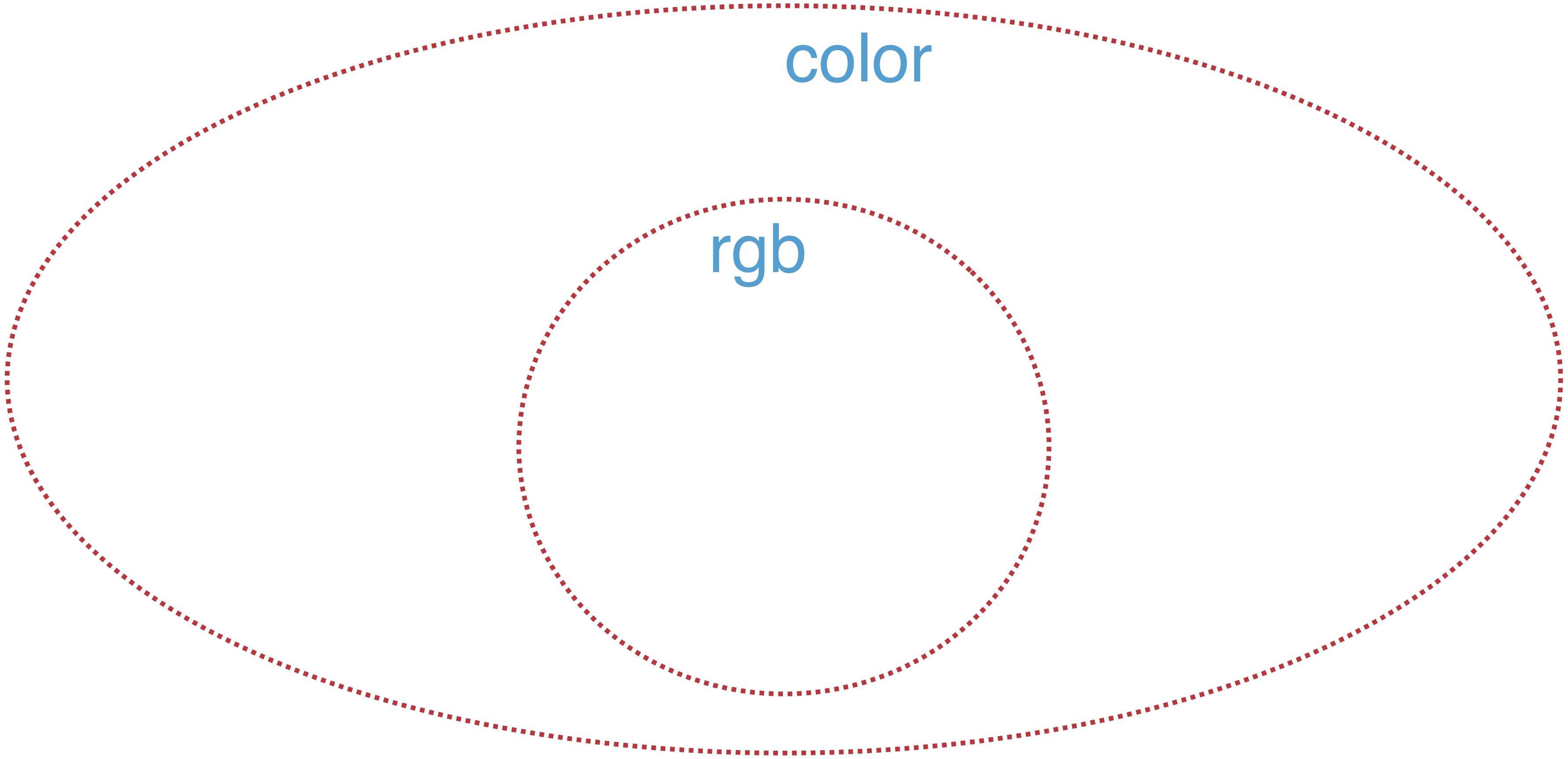
  (* pp (H 3) = "<h3>" *) *)

```
> type tag = H of int | A of string | P;
> let pp t =
    match t with
    | H i ->  "<h" ^ ((string_of_int i) ^ ">")
    | A hr -> "<a href=" ^ (hr ^ ">")
    | _ -> "<p>";
val pp : tag -> string = <fun>
```

So Far:

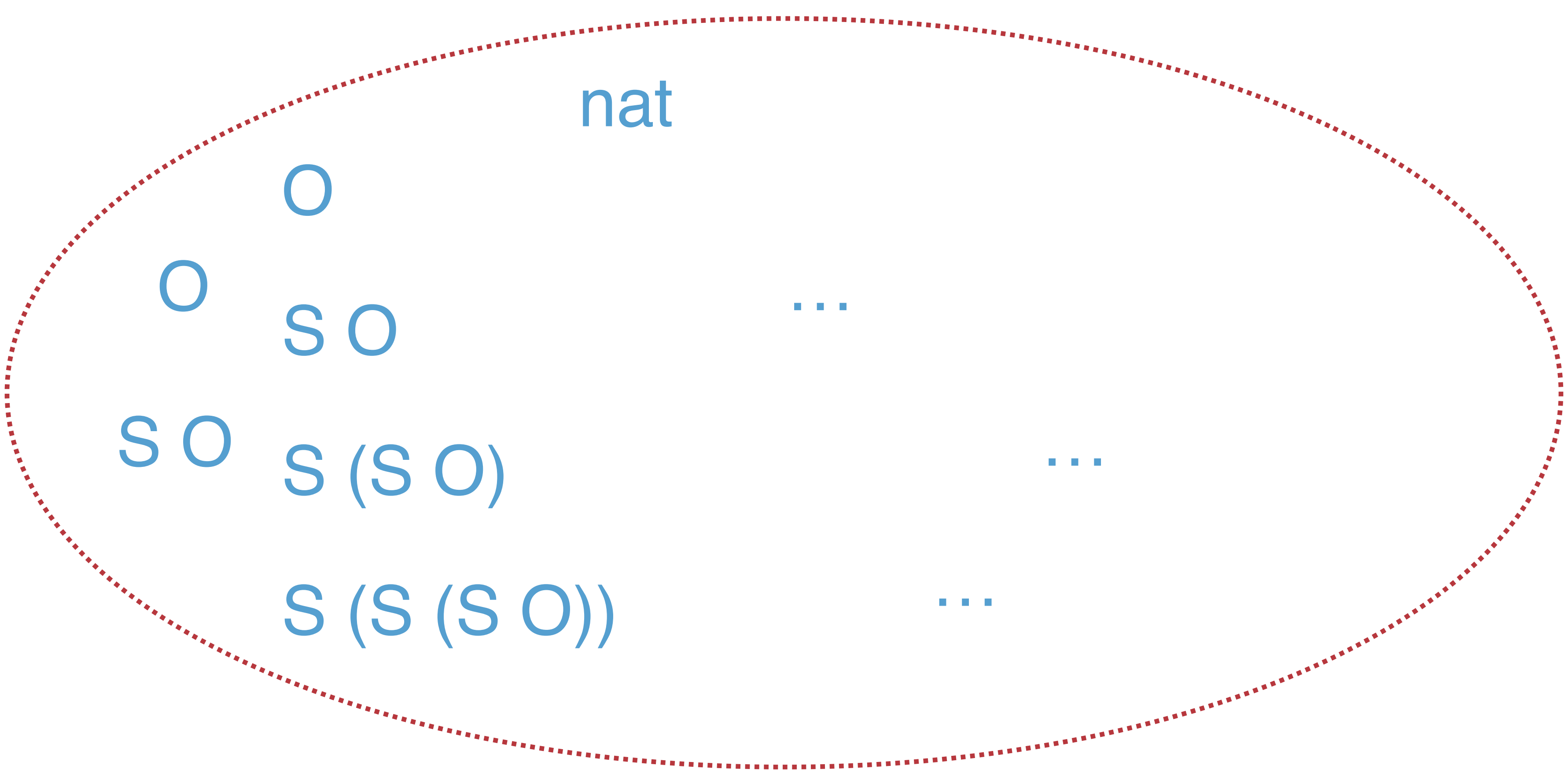

```
type rgb = Red | Green | Blue;

type color = Black | White | Primary of rgb;
```



Natural Numbers


```
type nat =  O  | S of nat
```



Functions

The *interpretation* of these constructors comes from how we use them to compute:

```
type tickNat = stop  | tick of tickNat;;
```

```
> let pred (n : nat) : nat =
    match n with
    | O -> O
    | S m -> m
val pred : nat -> nat = <fun>
```

Recursion

Use recursion to enumerate the elements of an inductive (algebraic) datatype

```
let rec iseven (n : nat) : bool =
  match n with
  | O -> true
  | S 0 -> false
  | S (S m) -> iseven m
```

Recursion

Use recursion to enumerate the elements of an inductive (algebraic) datatype

```
> let rec plus (n,m )=
  match n with
  | O -> m
  | S x -> S (plus x m);
val plus : nat * nat -> nat = <fun>

> plus ((S (S O)), (S (S (S O))));
- : nat = S (S (S (S (S O))))
```

Note that plus (S (S O), (S (S (S O)))) = S (plus ((S O), (S (S (S O)))))

Tuples, Currying

Use a tuple type (a finite collection of heterogeneous elements) to mimic multi-argument functions.

```
> let rec plus (n,m) =
  match n with
  | O -> m
  | S x -> S (plus(x, m));
val plus : nat * nat -> nat = <fun>

> plus ((S (S O)), (S (S (S O))));
- : nat = S (S (S (S (S O))))

> let n = S (S O) in
   let m = S (S (S O)) in
   plus (n,m);
- : nat = S (S (S (S (S O))))
```

Functions abstract values


```
> let rec mapAdd2 (l : int list) =
  match l with
  | [] -> []
  | hd :: tl -> (hd + 2) :: mapAdd2 tl
val mapAdd2 : int list -> int list = <fun>

> let rec mapAdd6 (l : int list) =
  match l with
  | [] -> []
  | hd :: tl -> (hd + 6) :: mapAdd2 tl
val mapAdd2 : int list -> int list = <fun>
```

```
> let rec mapAdd2 (n : int, l : int list) =
  match l with
  | [] -> []
  | hd :: tl -> (hd + n) :: mapAdd2 tl
val mapAdd2 : int * int list -> int list = <fun>
```

Functions abstract computation


```
> let rec mapInc lst = function
   | [] -> []
   | hd :: tl -> (hd + 1) :: (mapInc tl)
val mapInc : int list -> int list

> let rec mapDouble lst = function
   | [] -> []
   | hd :: tl -> (hd * 2) :: (mapDouble tl)
val mapInc : int list -> int list
```

```
> let rec map (f, lst) =
   match lst with
   | [] -> []
   | hd :: tl -> (f hd) :: (map (f,tl))
val map: (int -> int) * int list -> int list

> let inc n = n + 1;
val inc : int -> int

> let double n = n * 2;
val double : int -> int
```

```
> map (inc,[1;2;3]);
- : int list = (::) (2, [3; 4])

> map (double[1;2;3]);
- : int list = (::) (2, [4; 6])
```

map is a “higher-order” function

Functions abstract computation


```
> let rec map (f, lst) =
   match lst with
   | [] -> []
   | hd :: tl -> (f hd) :: (map (f,tl))
 val map: (int -> int) * int list -> int list

> map ((fun n -> n + 1), [1;2;3])
-: int list = (::) (2, [3; 4])

> map ((fun n -> n *  2), [1;2;3])
- : int list = (::) (2, [4; 6])

> map (inc,[1;2;3]);
- : int list = (::) (2, [3; 4])

> map (double[1;2;3]);
- : int list = (::) (2, [4; 6])
```

Functions can be "anonymous" i.e., they can be treated like values (just like values of any other type)

In this example, a function value was supplied as an argument, but function values can also be returned as a result by a function. When is that useful?

Currying


```
> let rec map f lst =
   match lst with
   | [] -> []
   | hd :: tl -> (f hd) :: (map (f,tl))
val map: (int -> int) -> int list -> int list

> let mapDouble = map (fun n -> n * 2)
-: int list -> int list

> mapDouble [1;2;3];
-: int list = (::) ([2; [4; 6])

> mapDouble [2;3;4]
-: int list = (::) ([4; [6; 8])
```

```
> let plus m n =
   match n with
   | O -> m
   | S x -> S (plus x m);

> let plus2 = plus (S (S O))

> plus2 (S (S (S O)))
```