

OCaml: The Basics

Concepts of Programming Languages

Lecture 2

Practice Problem

None today, instead we'll do a demo of the one of the functions from assignment 0 in a moment

Outline

- » Briefly outline the use of **Dune**
- » Cover the **basic expressions** we need to start programming in OCaml, look at some examples
- » Define more carefully the notion of an **inference rule**

Learning Objectives

- » Write basic OCaml programs on primitive types
- » Read inference rules, i.e., translate formal mathematical notation to English and English to mathematical notation

Recap

Recap: Functional vs. Imperative

OCaml is a **functional language**. This means a couple things:

- » No state (which means no loops!)
- » We don't think of a program as **describing a procedure**, but as **defining a value using an expression**

Recap: Expressions

Expressions are syntactic objects which describe values in a program

Mnemonic: *Expressions are
EValuated to Values*

They appear in both functional and imperative PLs, but in functional PLs **we *only* have expressions**

$$2 + (2 * 3)$$

```
if x = 3 then 3 else 4
```

$$H(f(f(f(x, y), 2), g(z)))$$

In fact, we'll often think of an OCaml program as a *single* expression
(more on that in a moment)

Recap: The Three Components

Recap: The Three Components

Syntax: What a *well-formed* program in your PL?

```
def f():  
    return 3
```



```
define f():  
    3 return
```



Recap: The Three Components

Syntax: What a *well-formed* program in your PL?

```
def f():  
    return 3
```

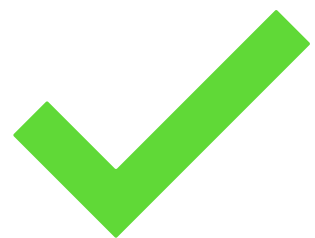


```
define f():  
    3 return
```



Type System (Static Semantics): What is a *valid* program in your PL?

```
x = 2 + 2
```



```
x = 2 + "two"
```



Recap: The Three Components

Syntax: What a *well-formed* program in your PL?

```
def f():  
    return 3
```



```
define f():  
    3 return
```



Type System (Static Semantics): What is a *valid* program in your PL?

```
x = 2 + 2
```



```
x = 2 + "two"
```



Semantics (Dynamic Semantics): What is the *output* of a (valid) program?

```
>>> 2 + 2
```

```
4
```



```
>>> 2 + 2
```

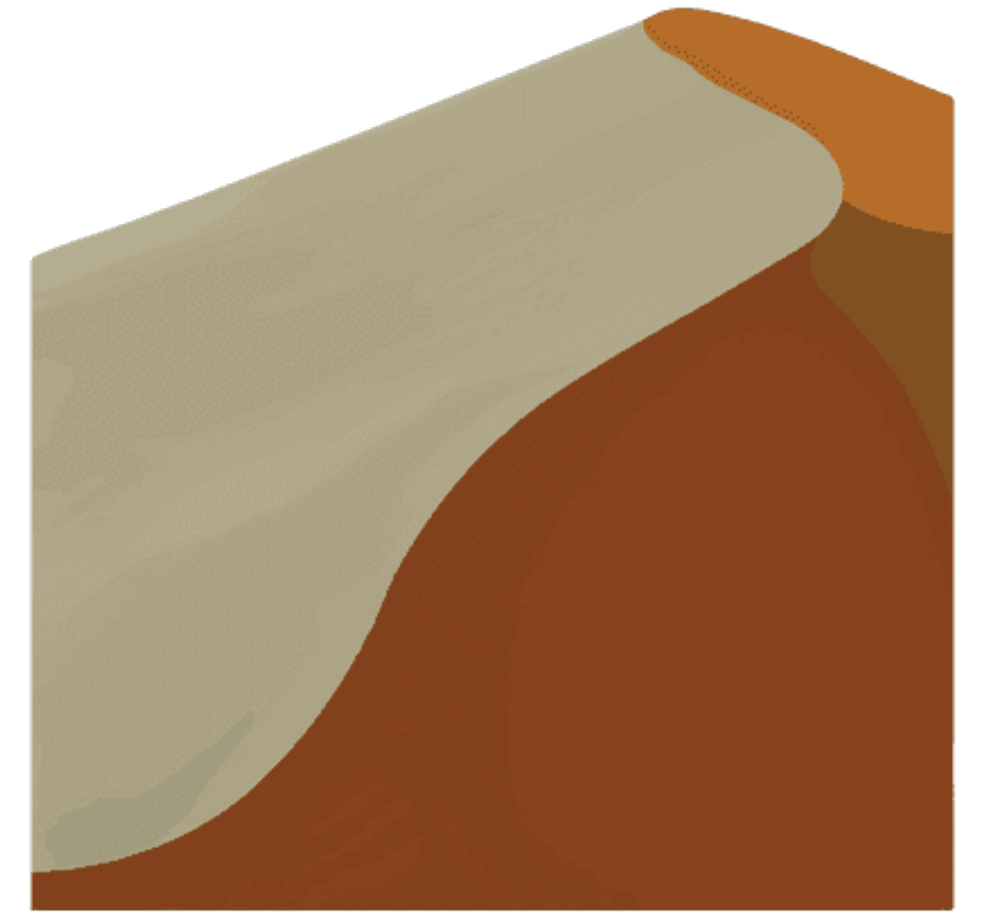
```
False
```



For every possible expression, we'll
define the **syntax rules**, the **typing
rules**, and the **semantic rules**

Working with OCaml

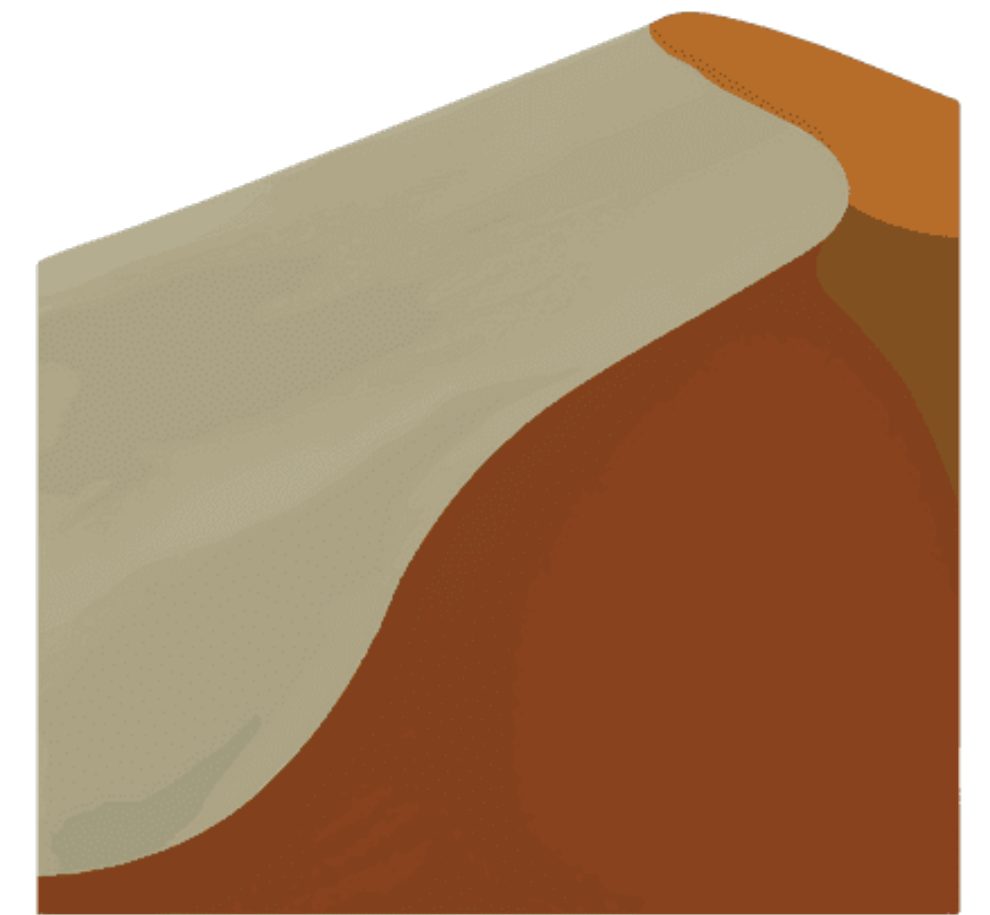
Dune



DUNE

Dune

Dune is a build tool for OCaml

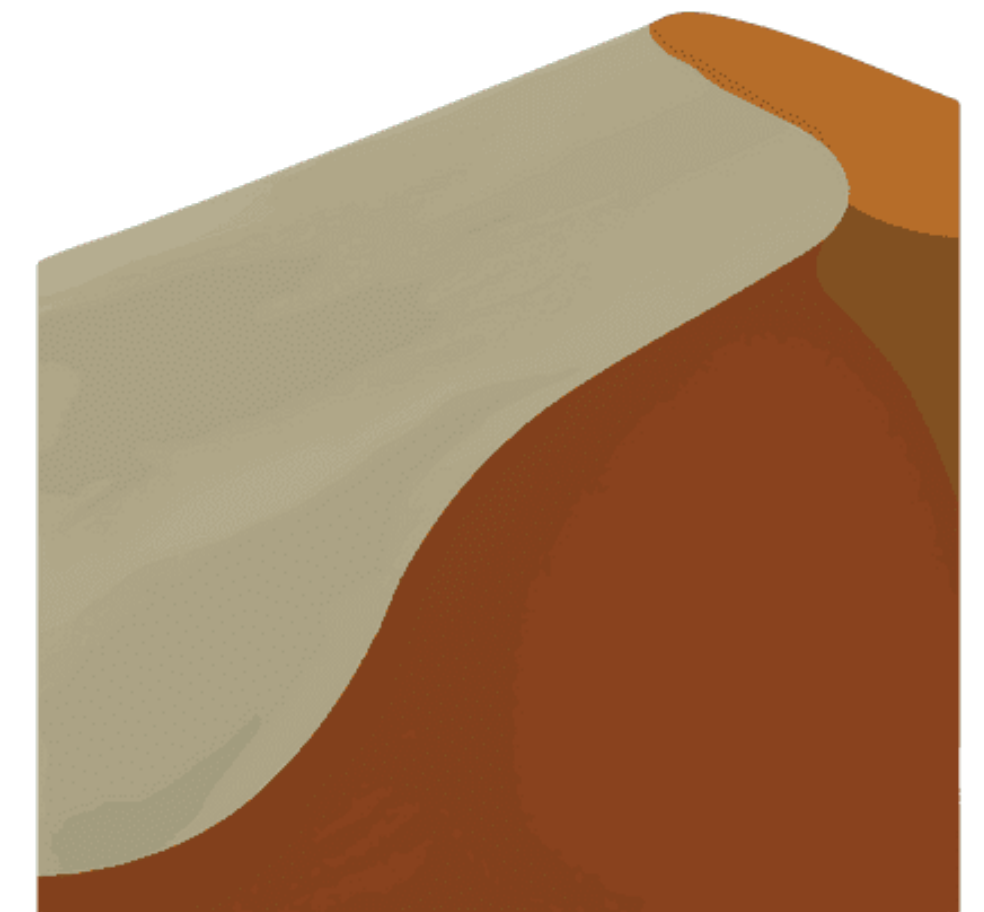


DUNE

Dune

Dune is a build tool for OCaml

It allows us to specify project-level dependencies and configurations



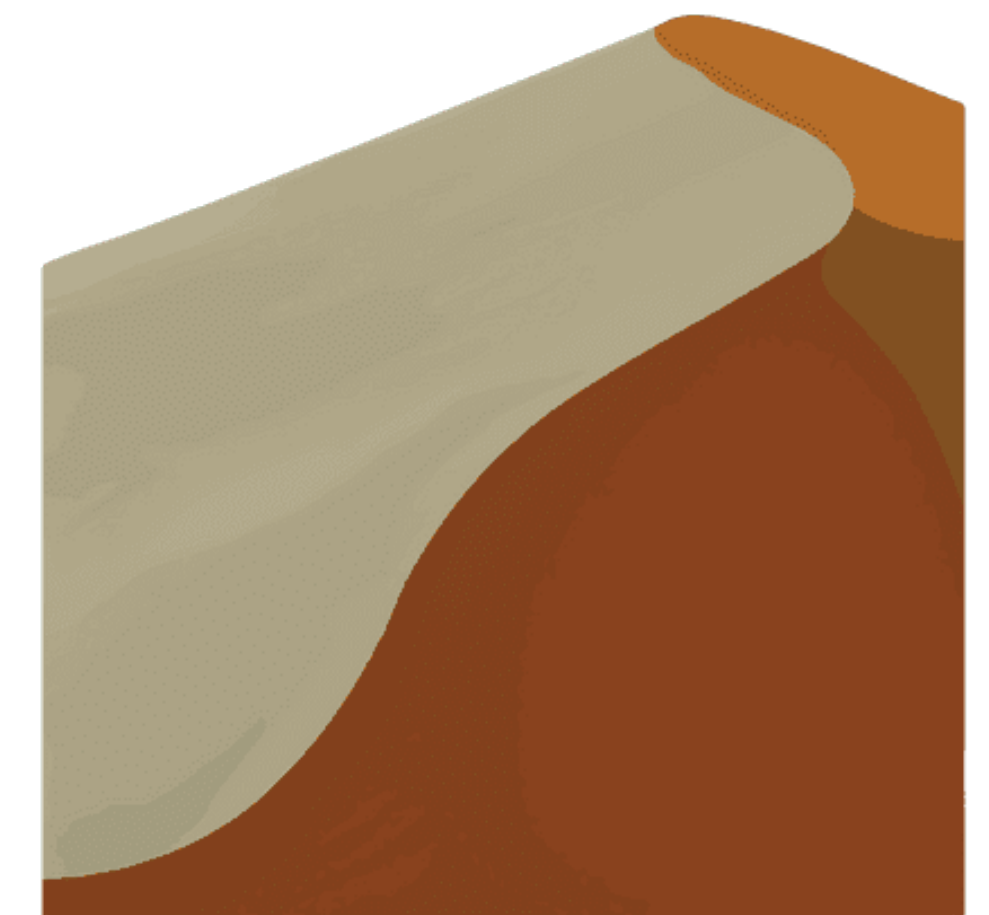
DUNE

Dune

Dune is a build tool for OCaml

It allows us to specify project-level dependencies and configurations

We'll use it throughout the course for all assignments and projects



DUNE

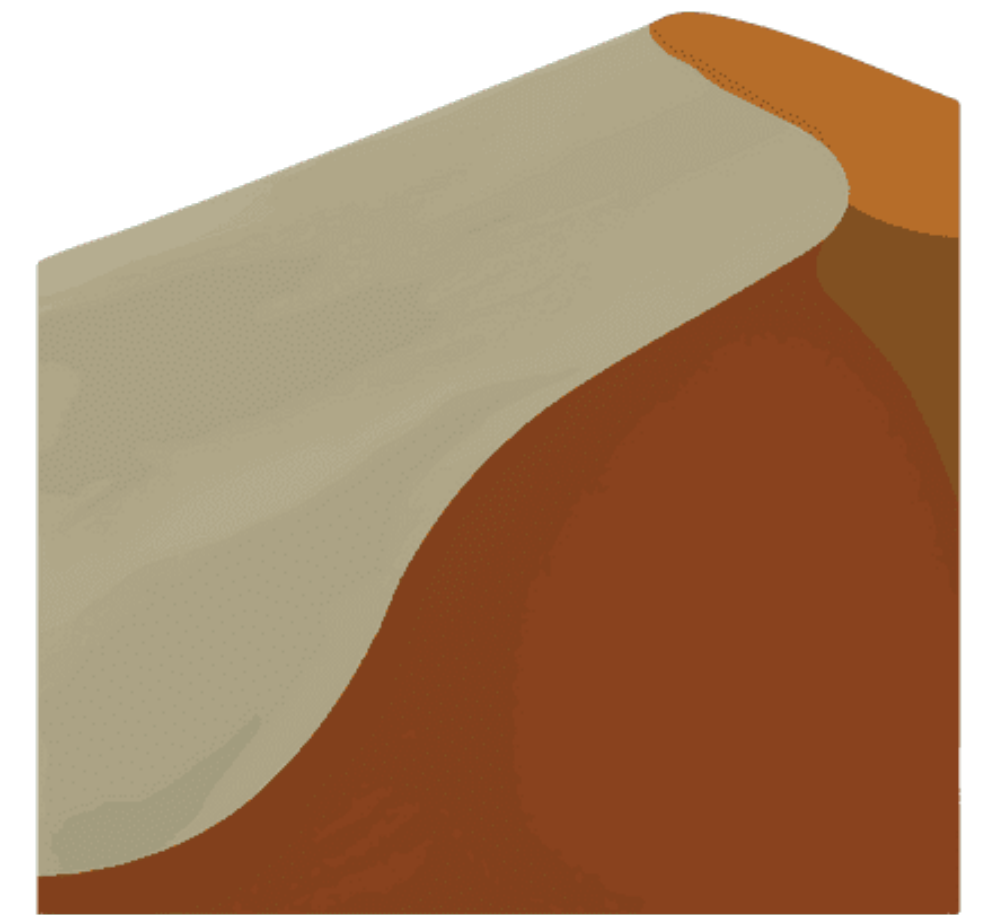
Dune

Dune is a build tool for OCaml

It allows us to specify project-level dependencies and configurations

We'll use it throughout the course for all assignments and projects

Cheatsheet:



DUNE

Dune

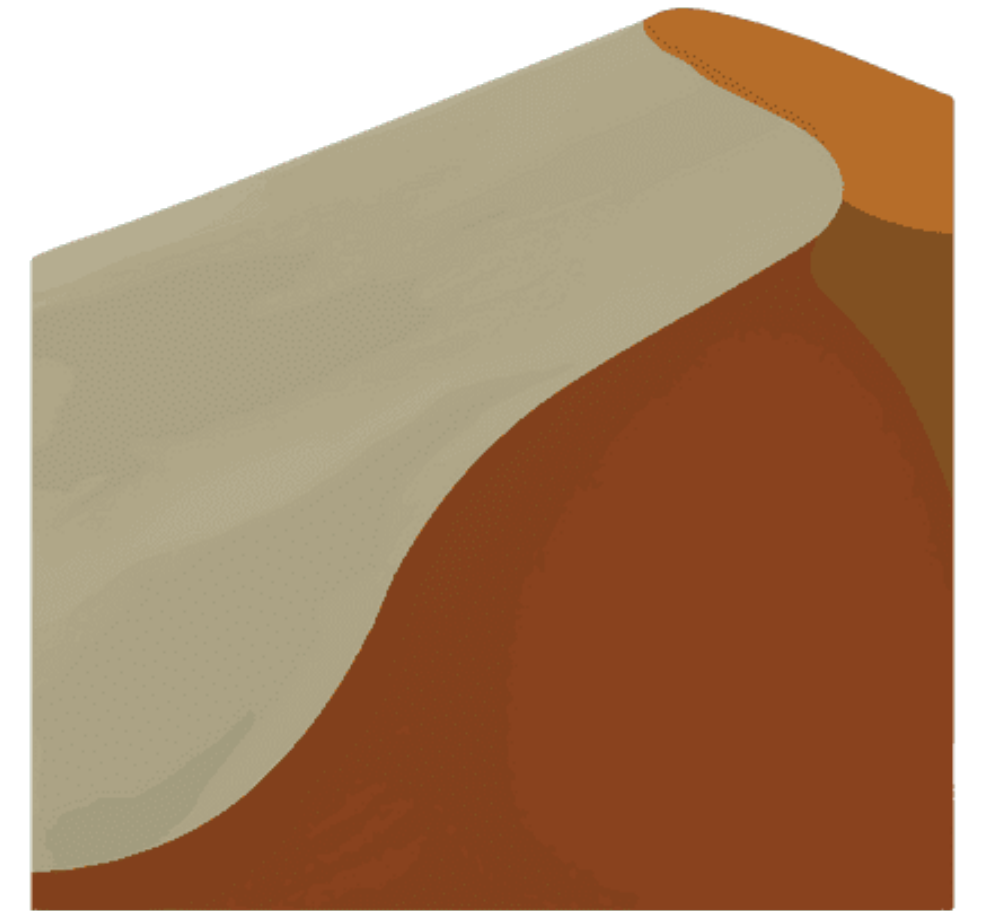
Dune is a build tool for OCaml

It allows us to specify project-level dependencies and configurations

We'll use it throughout the course for all assignments and projects

Cheatsheet:

» **dune build:** type check your project



DUNE

Dune

Dune is a build tool for OCaml

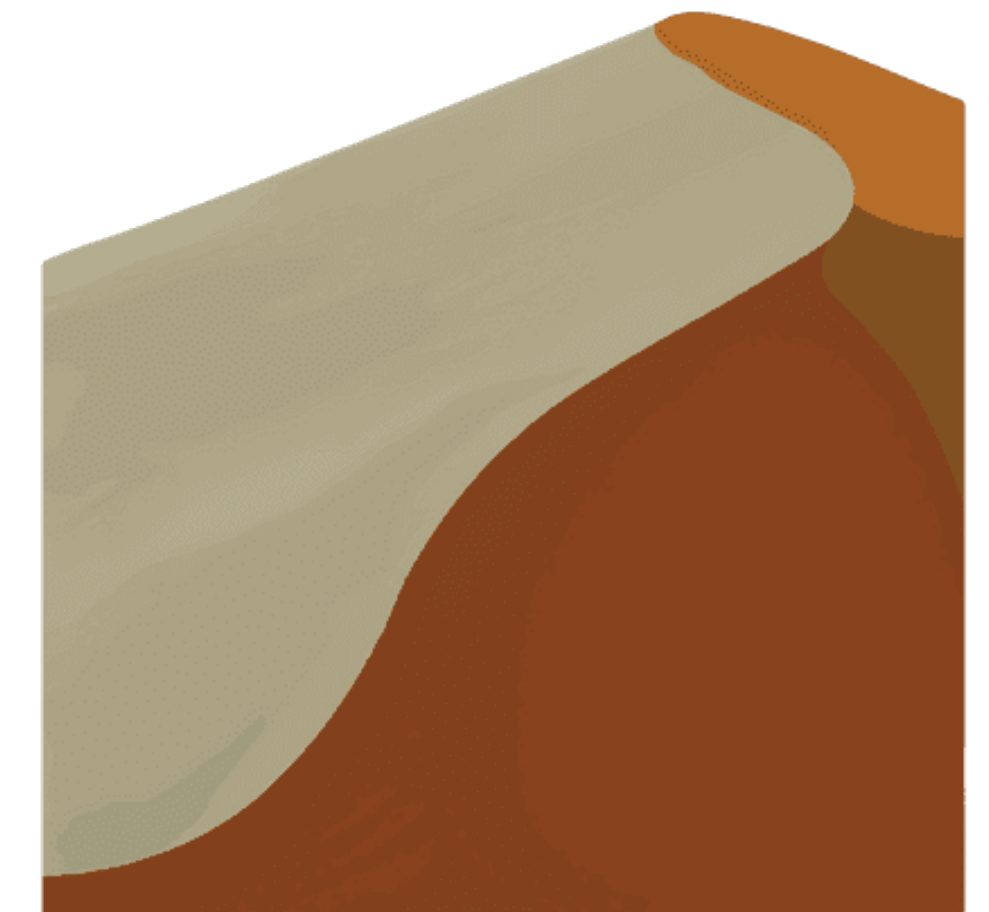
It allows us to specify project-level dependencies and configurations

We'll use it throughout the course for all assignments and projects

Cheatsheet:

» **dune build:** type check your project

» **dune utop:** open Utop in a project aware way



DUNE

Dune

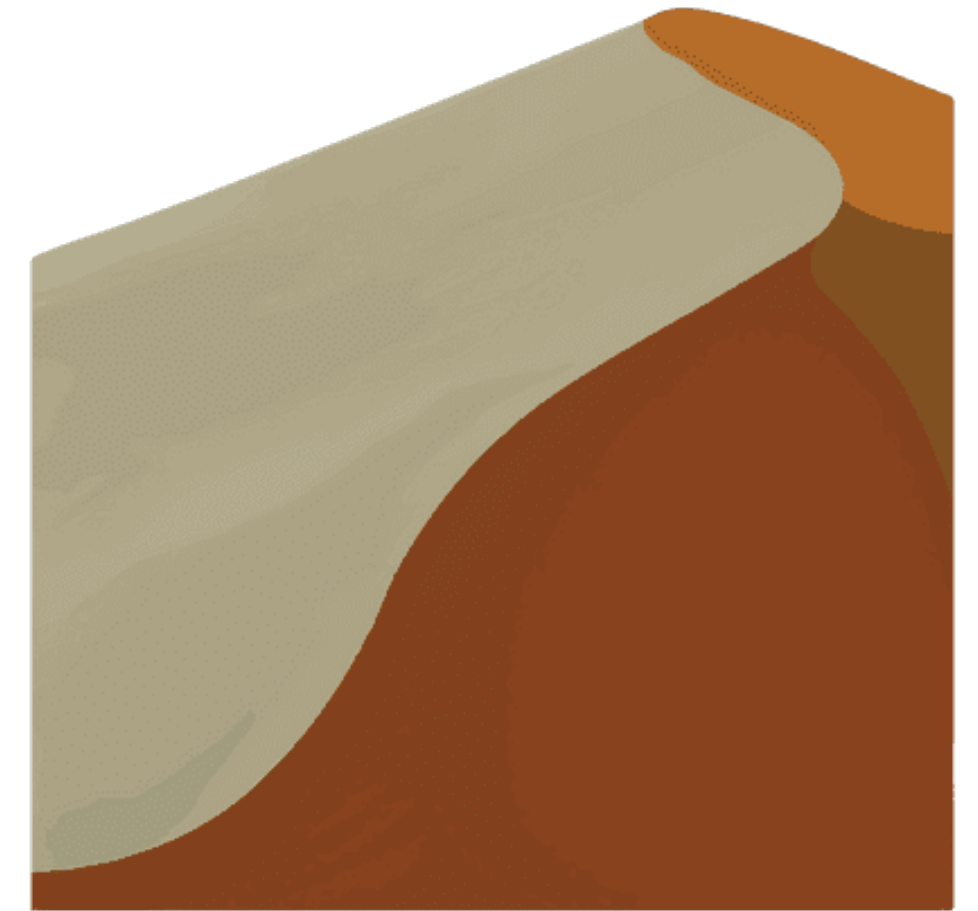
Dune is a build tool for OCaml

It allows us to specify project-level dependencies and configurations

We'll use it throughout the course for all assignments and projects

Cheatsheet:

- » **dune build:** type check your project
- » **dune utop:** open Utop in a project aware way
- » **dune test:** run a testing code associated with the project



DUNE

Dune

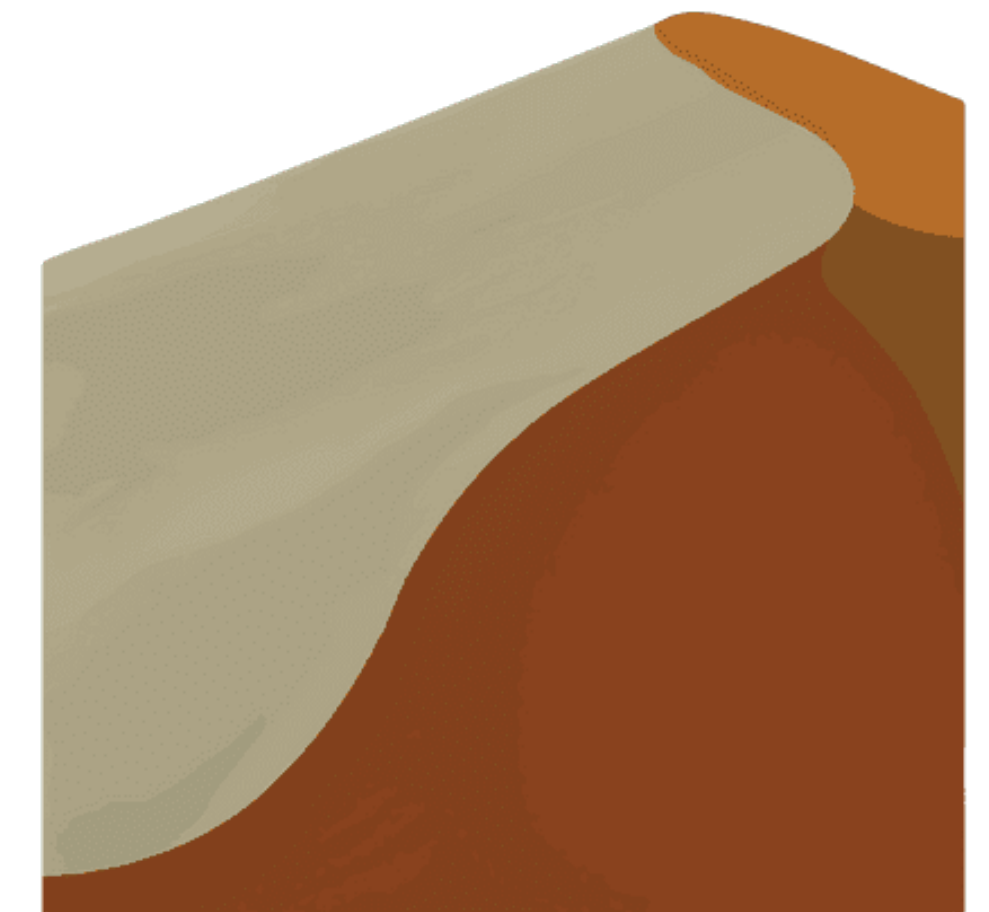
Dune is a build tool for OCaml

It allows us to specify project-level dependencies and configurations

We'll use it throughout the course for all assignments and projects

Cheatsheet:

- » **dune build:** type check your project
- » **dune utop:** open Utop in a project aware way
- » **dune test:** run a testing code associated with the project
- » **dune exec PROJ_NAME:** run the executable of your project



DUNE

Dune

Dune is a build tool for OCaml

It allows us to specify project-level dependencies and configurations

We'll use it throughout the course for all assignments and projects

Cheatsheet:

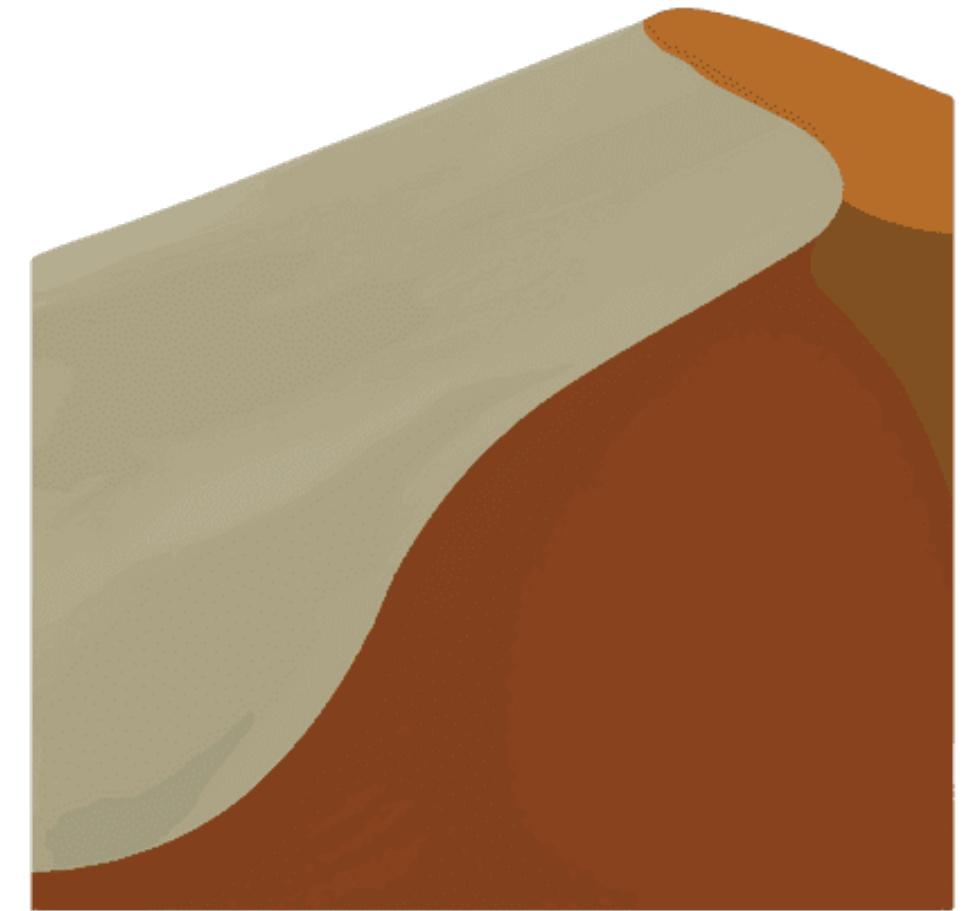
» **dune build:** type check your project

» **dune utop:** open Utop in a project aware way

» **dune test:** run a testing code associated with the project

» **dune exec PROJ_NAME:** run the executable of your project

» **dune clean:** removes files created by dune build (not so important but may come in handy)



DUNE

UTop

```
Welcome to utop version %%VERSION%%

Findlib has been successfully loaded. Additional directions:
#require "package";;      to load a package
#list;;                   to list the available packages
#camlp4o;;                to load camlp4 (standard syntax)
#camlp4r;;                to load camlp4 (revised syntax)
#predicates "p,q,...";;   to set these predicates
Topfind.reset();;         to force that packages will be reloaded
#thread;;                 to enable threads

Type #utop_help for help about using utop.

-( 23:00:06 )-< command 0 >
utop # 1 + 2;;
- : int = 3
-( 23:00:06 )-< command 1 >
utop #
```

Afl_instrument	Alias_analysis	Allocated_const	Annot	Arc
----------------	----------------	-----------------	-------	-----

UTop

UTop is an interface for the OCaml
toplevel

Welcome to utop version %%VERSION%%

Findlib has been successfully loaded. Additional directions:
#require "package";; to load a package
#list;; to list the available packages
#camlp4o;; to load camlp4 (standard syntax)
#camlp4r;; to load camlp4 (revised syntax)
#predicates "p,q,...";; to set these predicates
Topfind.reset();; to force that packages will be reloaded
#thread;; to enable threads

Type #utop_help for help about using utop.

-(23:00:06)-< command 0 >
utop # 1 + 2;;
- : int = 3
-(23:00:06)-< command 1 >
utop #

Afl_instrument	Alias_analysis	Allocated_const	Annot	Arc
----------------	----------------	-----------------	-------	-----

UTop

UTop is an interface for the OCaml toplevel

It's basically an **OCaml calculator**. It can *evaluate* OCaml expressions (useful for debugging)

```
Welcome to utop version %%VERSION%%

Findlib has been successfully loaded. Additional directions:
#require "package";;      to load a package
#list;;                   to list the available packages
#camlp4o;;                to load camlp4 (standard syntax)
#camlp4r;;                to load camlp4 (revised syntax)
#predicates "p,q,...";;  to set these predicates
Topfind.reset();;         to force that packages will be reloaded
#thread;;                 to enable threads

Type #utop_help for help about using utop.

-( 23:00:06 )-< command 0 >
utop # 1 + 2;;
- : int = 3
-( 23:00:06 )-< command 1 >
utop #
```

Afl_instrument	Alias_analysis	Allocated_const	Annot	Arc
----------------	----------------	-----------------	-------	-----

UTop

UTop is an interface for the OCaml
toplevel

It's basically an **OCaml calculator**. It
can *evaluate* OCaml expressions (useful
for debugging)

Cheatsheet:

Welcome to utop version %%VERSION%%

Findlib has been successfully loaded. Additional directions:
#require "package";; to load a package
#list;; to list the available packages
#camlp4o;; to load camlp4 (standard syntax)
#camlp4r;; to load camlp4 (revised syntax)
#predicates "p,q,...";; to set these predicates
Topfind.reset();; to force that packages will be reloaded
#thread;; to enable threads

Type #utop_help for help about using utop.

-(23:00:06)-< command 0 >
utop # 1 + 2;;
- : int = 3
-(23:00:06)-< command 1 >
utop #

Afl_instrument	Alias_analysis	Allocated_const	Annot	Arc
----------------	----------------	-----------------	-------	-----

UTop

UTop is an interface for the OCaml
toplevel

It's basically an **OCaml calculator**. It
can *evaluate* OCaml expressions (useful
for debugging)

Cheatsheet:

>> expressions must be followed with
two semicolons

```
Welcome to utop version %%VERSION%%

Findlib has been successfully loaded. Additional directi
#require "package";;      to load a package
#list;;                  to list the available packag
#camlp4o;;               to load camlp4 (standard syn
#camlp4r;;               to load camlp4 (revised synt
#predicates "p,q,...";;  to set these predicates
Topfind.reset();;        to force that packages will
#thread;;                to enable threads

Type #utop_help for help about using utop.

-( 23:00:06 )-< command 0 >
utop # 1 + 2;;
- : int = 3
-( 23:00:06 )-< command 1 >
utop # █

Afl_instrument | Alias_analysis | Allocated_const | Annot | Arc
```


UTop

UTop is an interface for the OCaml
toplevel

It's basically an **OCaml calculator**. It
can *evaluate* OCaml expressions (useful
for debugging)

Cheatsheet:

» expressions must be followed with
two semicolons

» #quit;; or (Ctl-D) leaves UTop

```
Welcome to utop version %%VERSION%%

Findlib has been successfully loaded. Additional directi
#require "package";;      to load a package
#list;;                  to list the available packag
#camlp4o;;               to load camlp4 (standard syn
#camlp4r;;               to load camlp4 (revised synt
#predicates "p,q,...";;  to set these predicates
Topfind.reset();;        to force that packages will
#thread;;                to enable threads

Type #utop_help for help about using utop.

-( 23:00:06 )-< command 0 >
utop # 1 + 2;;
- : int = 3
-( 23:00:06 )-< command 1 >
utop # █

Afl_instrument | Alias_analysis | Allocated_const | Annot | Arc
```

A Note on Testing

```
let _ = assert (expected = actual)
```

A Note on Testing

```
let _ = assert (expected = actual)
```

OCaml has a useful function called **assert** which can be used to write simple unit tests

A Note on Testing

```
let _ = assert (expected = actual)
```

OCaml has a useful function called **assert** which can be used to write simple unit tests

If you write an assert in the file

A Note on Testing

let _ = assert (expected = actual)

OCaml has a useful function called **assert** which can be used to write simple unit tests

If you write an assert in the file

`test/test_PROJECTNAME.ml`

A Note on Testing

let _ = assert (expected = actual)

OCaml has a useful function called **assert** which can be used to write simple unit tests

If you write an assert in the file

`test/test_PROJECTNAME.ml`

then it will be evaluated when you run **dune test** in the project directory

A Note on Testing

let _ = assert (expected = actual)

OCaml has a useful function called **assert** which can be used to write simple unit tests

If you write an assert in the file

`test/test_PROJECTNAME.ml`

then it will be evaluated when you run **dune test** in the project directory

We'll see how to do this much better later on...

demo

Expressions (Informally)

High Level View

Values are the *things* manipulated and output by programs, e.g., the integer 7 or the string "seven"

Expressions *describe* values (the values to which they evaluate)

Example: The expression $2 + 7$ "describes" the value 9

Types

```
let x : int = 2
let y : string = "two"
let _ = x + y (* THIS IS NOT POSSIBLE *)
```

```
3 | let _ = x + y (* THIS IS NOT POSSIBLE *)
      ^
```

Error: This expression has type string but an expression was expected of type int

Types

```
let x : int = 2
let y : string = "two"
let _ = x + y (* THIS IS NOT POSSIBLE *)
```

```
3 | let _ = x + y (* THIS IS NOT POSSIBLE *)
      ^
```

Error: This expression has type string but an expression was expected of type
int

Every expression in OCaml has a type

Types

```
let x : int = 2
let y : string = "two"
let _ = x + y (* THIS IS NOT POSSIBLE *)
```

```
3 | let _ = x + y (* THIS IS NOT POSSIBLE *)
      ^
```

Error: This expression has type string but an expression was expected of type
int

Every expression in OCaml has a type

The type of an expression describes what *kind* of thing it is

Types

```
let x : int = 2
let y : string = "two"
let _ = x + y (* THIS IS NOT POSSIBLE *)
```

```
3 | let _ = x + y (* THIS IS NOT POSSIBLE *)
      ^
```

Error: This expression has type string but an expression was expected of type
int

Every expression in OCaml has a type

The type of an expression describes what *kind* of thing it is

Types **restrict** how expression can be constructed

Primitive Types and Literals

As with any PL, OCaml has a collection of standard types and literals

Type	Literals	Operators
int	0, -2, 13, -023	+, -, *, /, mod
float	3., -1.01	+, -, *, /.
bool	true, false	&&, , not
char	'b', 'c'	
string	"word", "@*&#"	^

% in Python

and *or*

concat

A Couple Note on Operators

A Couple Note on Operators

Operators int and float are *different*, e.g., `+` (integer addition) and `+.` (float addition)

A Couple Note on Operators

Operators int and float are *different*, e.g., `+` (integer addition) and `+.` (float addition)

OCaml has **no operator overloading**

A Couple Note on Operators

Operators int and float are *different*, e.g., `+` (integer addition) and `+.` (float addition)

OCaml has **no operator overloading**

Comparison operators are standard, e.g., `<`, `<=`, `>`, `>=`, and can be used to compare any expressions of the same type

A Couple Note on Operators

Operators int and float are *different*, e.g., `+` (integer addition) and `+.` (float addition)

OCaml has **no operator overloading**

Comparison operators are standard, e.g., `<`, `<=`, `>`, `>=`, and can be used to compare any expressions of the same type

Note that equality check is just `=` (not `==`) and inequality is `<>` (not `!=`)

demo

A Note on Type Annotations

```
let rec fact (n : int) : int =  
  if n <= 0  
  then 1  
  else n * fact (n - 1)
```

A Note on Type Annotations

```
let rec fact (n : int) : int =  
  if n <= 0  
  then 1  
  else n * fact (n - 1)
```

OCaml has type inference which means we rarely have to *specify* the types of expression in our program

A Note on Type Annotations

```
let rec fact (n : int) : int =  
  if n <= 0  
  then 1  
  else n * fact (n - 1)
```

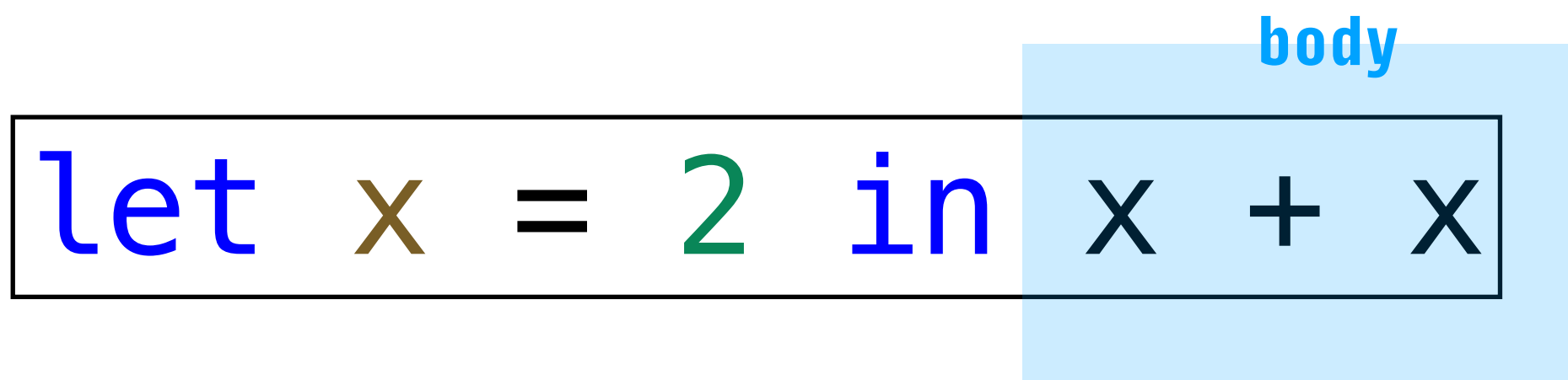
OCaml has type inference which means we rarely have to *specify* the types of expression in our program

Including type annotations can be useful for *documentation* and for *code clarity* (we will often do this to begin with)

Local Variables

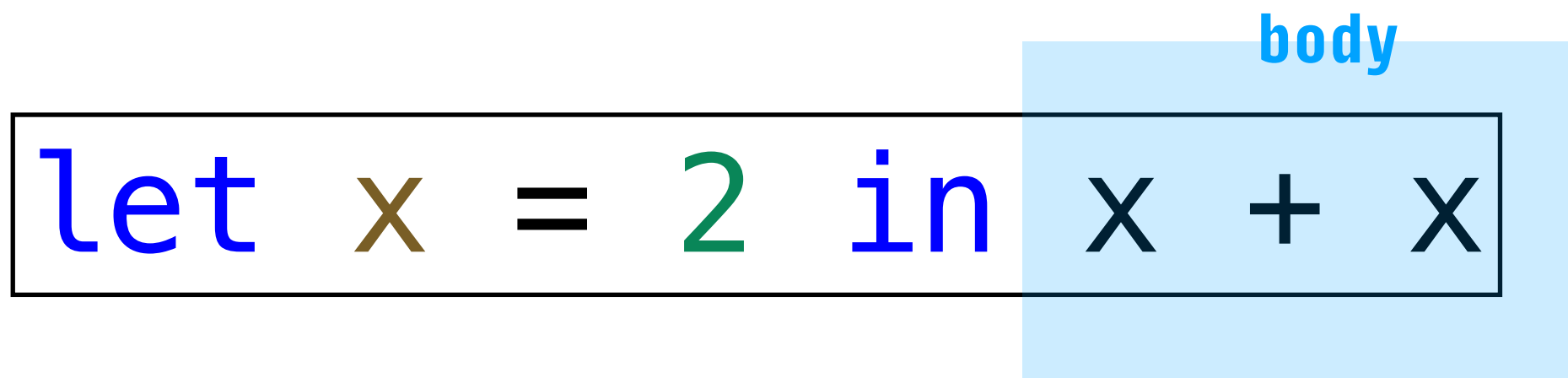
body

```
let x = 2 in x + x
```



The diagram illustrates a `let` expression. The text `let x = 2 in x + x` is enclosed in a black rectangular box. The word `body` is written in blue above the right portion of this box. A light blue rectangular area highlights the right portion of the box, which contains the expression `x + x`, representing the body of the `let` binding.

Local Variables

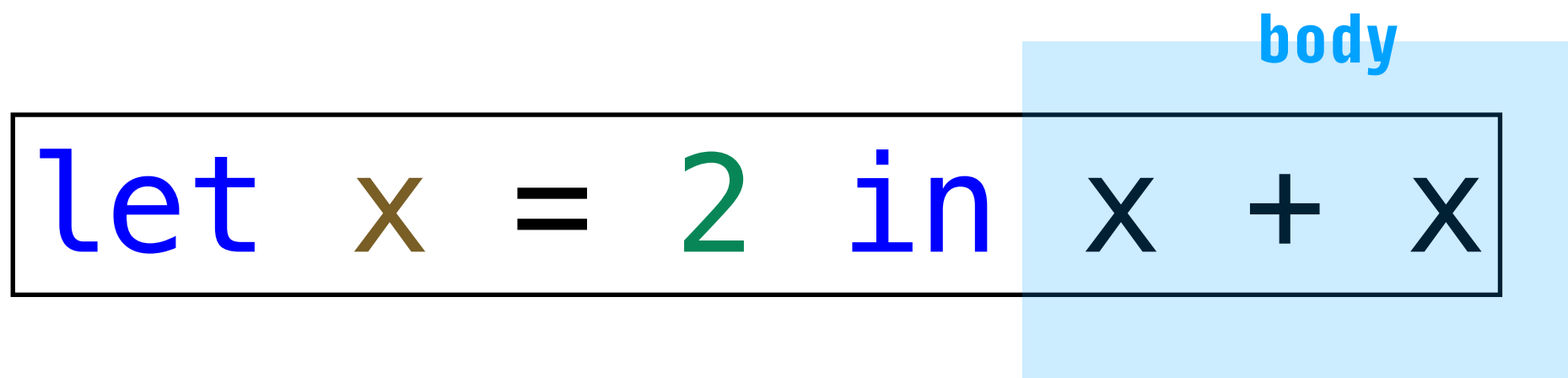


The diagram shows the OCaml expression `let x = 2 in x + x` enclosed in a black rectangular box. The word `let` is blue, `x` is brown, `=` is black, `2` is green, `in` is blue, and `x + x` is black. A light blue rectangular background highlights the entire expression. The word `body` in blue is positioned above the right side of the box, specifically over the `x + x` part.

```
let x = 2 in x + x
```

As with any reasonable PL, we can define local variables in OCaml

Local Variables

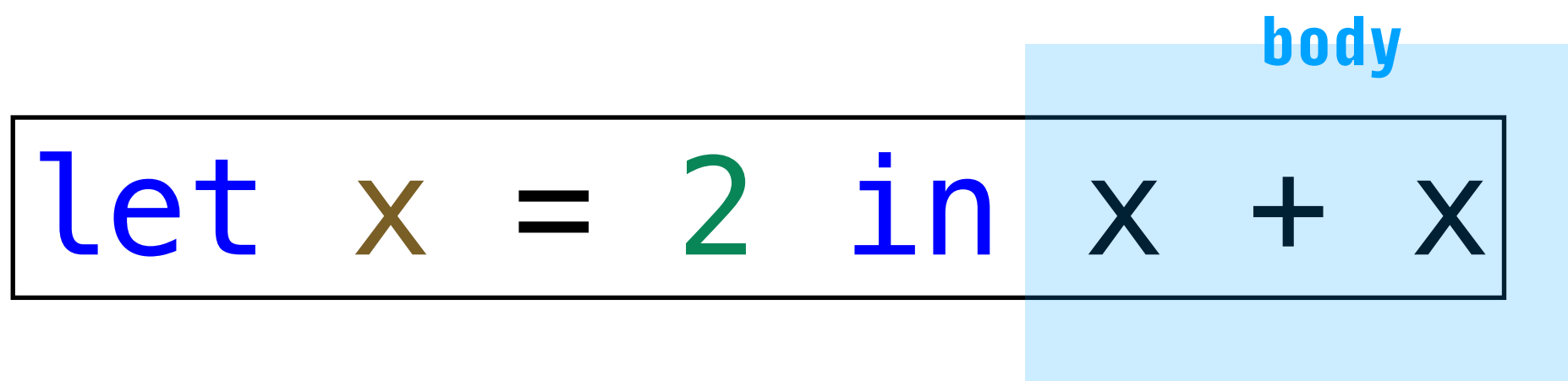


The diagram shows the OCaml code `let x = 2 in x + x`. The text is enclosed in a black rectangular box. The word `let` is blue, `x` is brown, `=` is black, `2` is green, `in` is blue, and `x + x` is black. A light blue rectangular area highlights the `x + x` part of the code. The word `body` is written in blue above the right side of this light blue area.

As with any reasonable PL, we can define local variables in OCaml

This is useful for writing better abstractions

Local Variables



The diagram illustrates the structure of a local variable definition in OCaml. It shows the code `let x = 2 in x + x` enclosed in a black rectangular box. This box is itself contained within a larger, light blue rectangular box. The label `body` is written in blue text above the right side of the light blue box, indicating that the expression `x + x` is the body of the `let` expression.

```
let x = 2 in x + x
```

As with any reasonable PL, we can define local variables in OCaml

This is useful for writing better abstractions

Note that it reads like a sentence: *let x stand for 2 in the expression $x + x$*

Multiple Local Variables

```
def sum_of_squares(x, y):  
    x_squared = x * x  
    y_squared = y * y  
    return x_squared + y_squared
```

Python

```
let sum_of_squares x y =  
    let x_squared = x * x in  
    (let y_squared = y * y in  
     x_squared + y_squared)
```

OCaml

It's very easy to use multiple local variables, we just *nest* local variables

(If it helps, think of *in* as a semicolon *;*)

IMPORTANT: `let x = e1 in e2` is an *expression* so it can be the body of a `let` expression.

Recall: Anatomy of an OCaml Program

```
let x = 3

let y = "string"

(* function definition *)
let square x = x * x

(* recursive function definition *)
let rec f x = if x = 0 then 0 else x + f (x - 1)

(* We can't just print , we assign to wildcard *)
let _ = print_endline("Hello world")
```

An OCaml Program consists of top-level let-expressions,
i.e., it is a **collection of named expressions**

OCaml Programs are Expressions

```
let x = 3 in  
  
let y = "string" in  
  
(* function definition *)  
let square x = x * x in  
  
(* recursive function definition *)  
let rec f x = if x = 0 then 0 else x + f (x - 1) in  
  
(* We can't just print , we assign to wildcard *)  
let _ = print_endline("Hello world") in  
  
0
```

arbitrary



This sequence of top-level let expressions is really shorthand for a **collection of nested local variables**

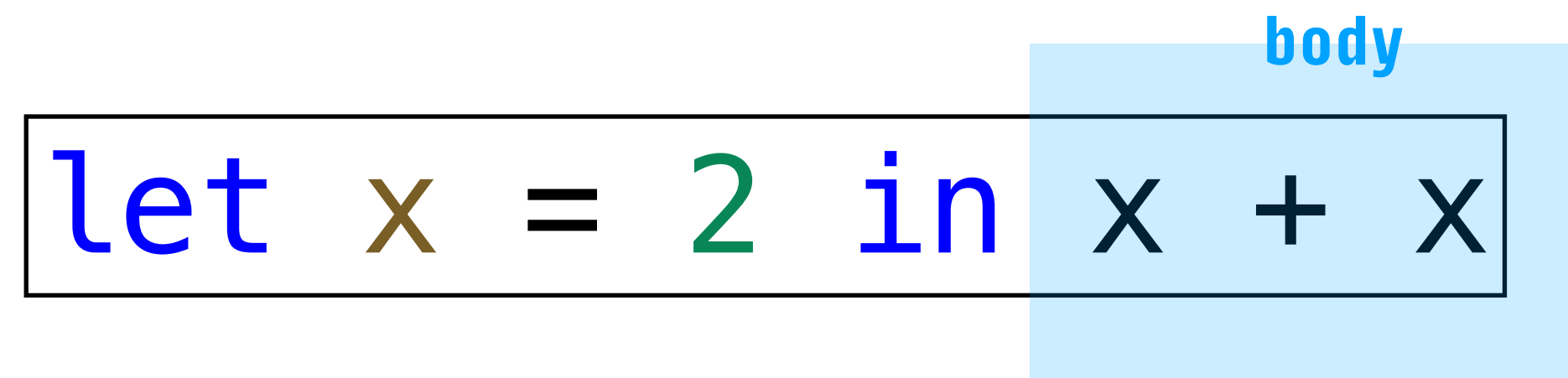
(This is a lie, but its a useful one for now)

Local Variables (Informal)

`let x = 2 in` `x + x`

body

Local Variables (Informal)

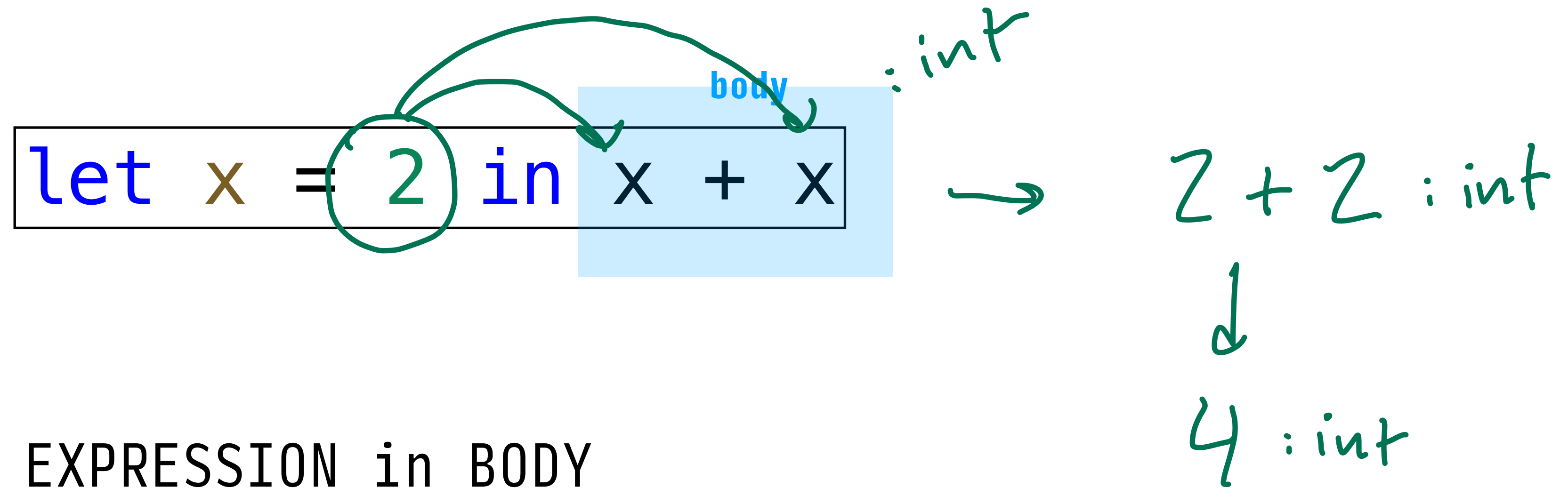


The diagram shows the code snippet `let x = 2 in x + x`. The text is enclosed in a black rectangular border. The words `let` and `in` are blue, `x` is brown, `=` is black, and `2` is green. The expression `x + x` is highlighted by a light blue rectangular background. The word `body` is written in blue above the right side of this background.

```
let x = 2 in x + x
```

syntax: `let VARIABLE = EXPRESSION in BODY`

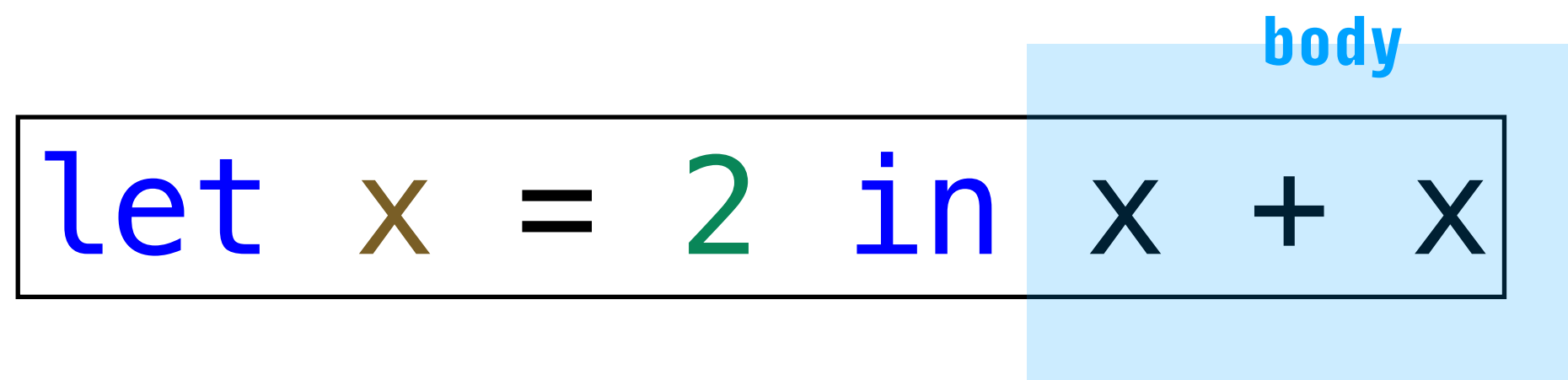
Local Variables (Informal)



syntax: `let VARIABLE = EXPRESSION in BODY`

typing: the type is the same as that of BODY *given BODY is well-typed after substituting the VARIABLE in BODY*

Local Variables (Informal)



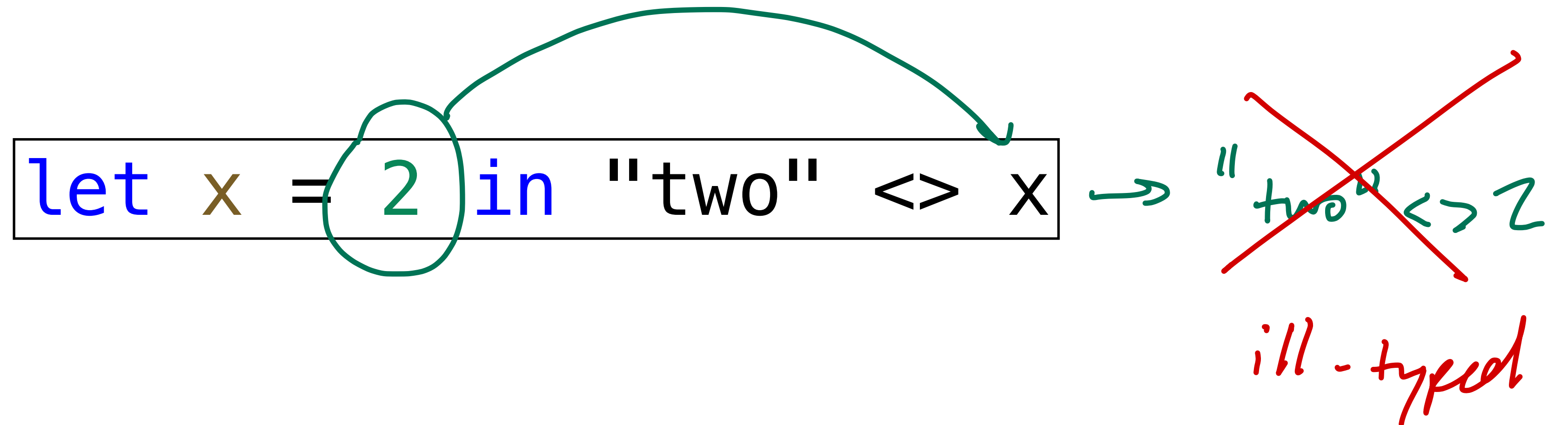
The diagram shows a code snippet `let x = 2 in x + x` enclosed in a black rectangular border. The word `let` is blue, `x` is brown, `=` is black, `2` is green, `in` is blue, and `x + x` is black. A light blue rectangular box highlights the `x + x` portion. The word `body` is written in blue above the right side of this box.

syntax: `let VARIABLE = EXPRESSION in BODY`

typing: the type is the same as that of BODY *given BODY is well-typed after substituting the VARIABLE in BODY*

semantics: the is the same as the value of BODY *after substituting the VARIABLE in BODY*

Example: Ill-Typed Let-Expression



An ill-typed expression will throw a type error when you try to evaluate it

Note that the body of a let-expression may be ill-typed depending on the value assigned to its variable

A Note on Substitution

let $x = 2$ in $x + x$



$2 + 2$

A Note on Substitution

$$\boxed{\text{let } x = 2 \text{ in } x + x} \longrightarrow \boxed{2 + 2}$$

Formally, we write $[v/x]e$ to mean "substitute v for x in e ",
e.g., $[3/x](x + x)$ is the same as $3 + 3$

A Note on Substitution

$$\boxed{\text{let } x = 2 \text{ in } x + x} \longrightarrow \boxed{2 + 2}$$

Formally, we write $[v/x]e$ to mean "substitute v for x in e ",
e.g., $[3/x](x + x)$ is the same as $3 + 3$

Intuitively, substitution is simple: **replace the variable**

A Note on Substitution

$$\boxed{\text{let } x = 2 \text{ in } x + x} \longrightarrow \boxed{2 + 2}$$

Formally, we write $[v/x]e$ to mean "substitute v for x in e ",
e.g., $[3/x](x + x)$ is the same as $3 + 3$

Intuitively, substitution is simple: **replace the variable**

Turns out, this is **very hard** to do correctly, *it's subtle* and
a source of a lot of mistakes in PL implementations

demo

If-Expressions

```
let abs x = if x > 0 then x else -x
```

Note: OCaml is whitespace agnostic!

If-Expressions

```
let abs x = if x > 0 then x else -x
```

Note: OCaml is whitespace agnostic!

As with any reasonable PL, OCaml has expressions for doing conditional reasoning

If-Expressions

```
let abs x = if x > 0 then x else -x
```

Note: OCaml is whitespace agnostic!

As with any reasonable PL, OCaml has expressions for doing conditional reasoning

Note: The **else** case is *required* and the **then** and **else** cases must be the *same type* (why?)

If-Expressions

if false then 0 ^{not valid}

```
let foo x =  
  if x < 0 then  
    "negative"  
  else if x = 0 then  
    "zero"  
  else  
    "positive"
```

if b then 1
else "two"

Answer: Remember, all we have is expressions. So every if-expression must have a value and a type (and therefore, an **else** case of the same type)

We can do **else if** just by nesting if-expressions! (neat)

Aside: If-Expressions in Python

```
if x < 0:  
    return -1  
else:  
    return 1
```

if-stmt (Python)

```
return (-1 if x < 0 else 1)
```

if-expr (Python)

If-*statements* in Python are different from if-expressions, but **both are available**

Statements don't have a value, expressions do

If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE-CASE else FALSE-CASE

If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE-CASE else FALSE-CASE

Typing: CONDITION must be a Boolean and TRUE-CASE and FALSE-CASE must be the same type. The type is then the same as that of TRUE-CASE and FALSE-CASE

If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE-CASE else FALSE-CASE

Typing: CONDITION must be a Boolean and TRUE-CASE and FALSE-CASE must be the same type. The type is then the same as that of TRUE-CASE and FALSE-CASE

Semantics: If CONDITION holds, then we get the TRUE-CASE, otherwise we get the FALSE-CASE

demo

Functions

```
let f x y z = x + y + z
let f (x : int) (y : int) (z : int) : int = x + y + z
```

There are a couple ways of defining functions in OCaml

So far, we've seen that let-expression can take arguments. ***How should we interpret this? If everything is an expression?***

Anonymous Functions

```
let f = fun x -> fun y -> fun z -> x + y + z
```

Answer: It must be that **functions are expressions as well!**

In OCaml, we can define *anonymous* functions, which are just **functions without names**

You should think of:

```
let f x y z = x + y + z
```

as shorthand for the above

Aside: Anonymous Functions in Python

```
lambda x: x + 1
```

Python

```
fun x -> x + 1
```

OCaml

$$\lambda x. x + 1$$

Math

There are also anonymous function in Python!

They're called *lambdas*, based on the **lambda calculus**, a mathematical formulation of a functional PL that dates back to the 1930s, invented by **Alonzo Church**

(You'll find a lot of functional ideas hidden in languages like Python)



Functions (Informal)

`fun x -> x + 1 : int -> int`

body

Syntax: `fun VAR-NAME -> EXPR`

Typing: the type of a function is `T1 -> T2` where T1 is the type of the input and T2 is the type of the output

Semantics: A function will evaluate to special ***function value*** (printed as `<fun>` by UTop)

Important: Curried Functions

```
let f = fun x -> (fun y -> (fun z -> x + y + z))
```

The only kind of function we have is a *single argument function*

This seems restrictive, but ultimately it doesn't affect us at all

We can *simulate* multi-argument functions with nested functions.
This is called **Currying** after Haskell Curry

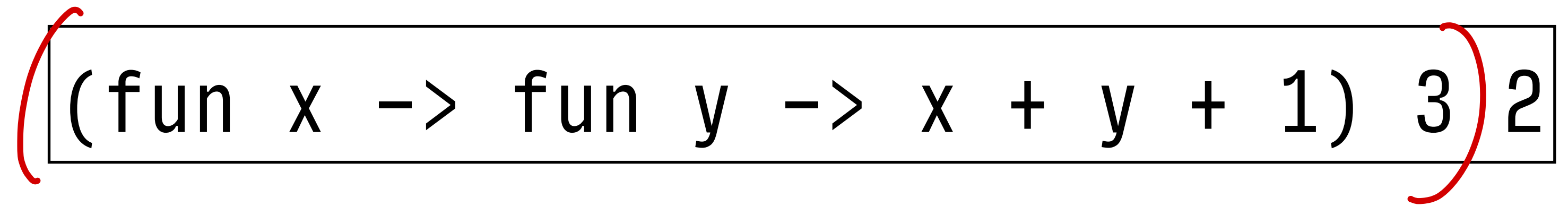
Important: Curried Functions

```
let f = fun x -> fun y -> fun z -> x + y + z
```

We should think of the above function as something which takes an input and returns **another function**

In other words, we *partially apply* the function

Application

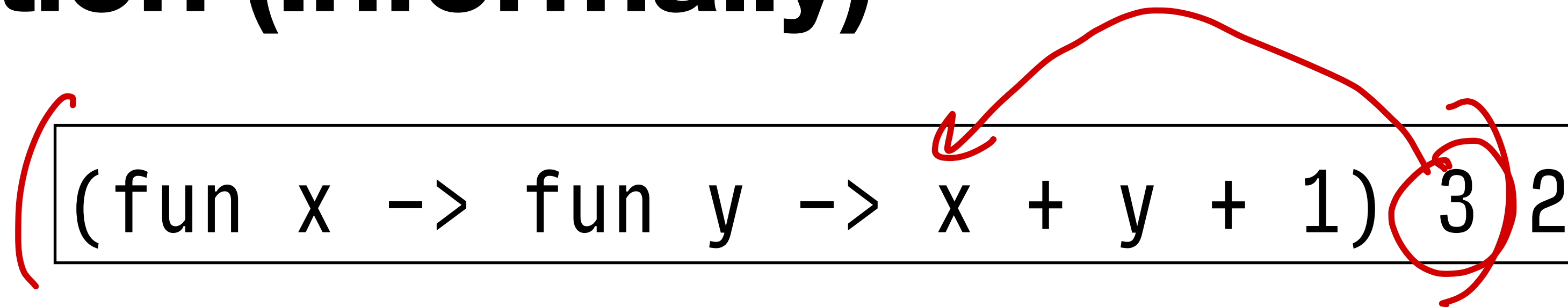


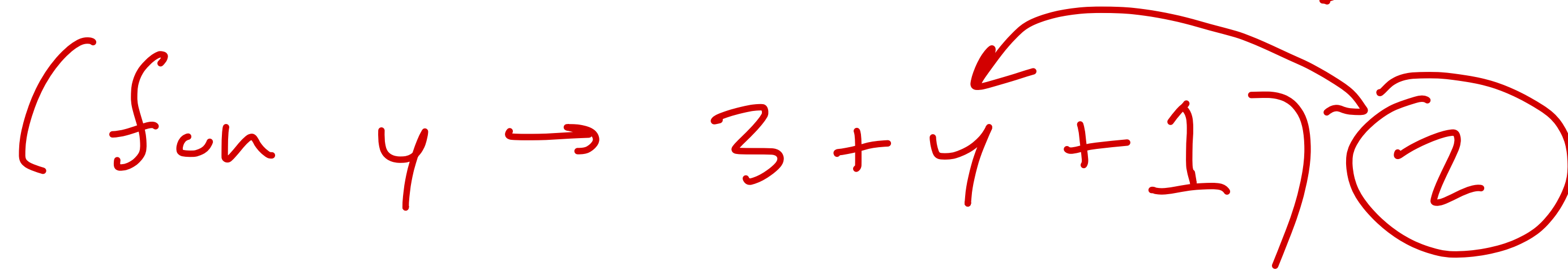
```
(fun x -> fun y -> x + y + 1) 3 2
```

Application is done by *juxtaposition* which means we put the arguments next to the function

Application is *left-associative*, which means we pass arguments from left to right

Application (Informally)


$$(\text{fun } x \rightarrow \text{fun } y \rightarrow x + y + 1) \ 3 \ 2$$


$$(\text{fun } y \rightarrow 3 + y + 1) \ 2$$

$$3 + 2 + 1 = 6$$

Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Typing: If FUNCTION-EXPR is of type $T1 \rightarrow T2$, and ARG-EXPR is of type $T1$, then the type is $T2$

Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Typing: If FUNCTION-EXPR is of type $T1 \rightarrow T2$, and ARG-EXPR is of type $T1$, then the type is $T2$

Semantics: Substitute the value of ARG-EXPR into the body of FUNCTION-EXPR and evaluate that

Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Example:

Practice Problem

Implement a function **first digit** which takes an integer **n** as an input and returns the first digit of **n** (without converting to a string)

Expressions (Formally)

Aside: Production Rules and Syntax

$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$

Aside: Production Rules and Syntax

`<expr> ::= <expr> + <expr>`

Last time, we saw the above notation. This is called a ***production rule*** and is part of a ***BNF grammar***

Aside: Production Rules and Syntax

$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$$

Last time, we saw the above notation. This is called a ***production rule*** and is part of a ***BNF grammar***

Reminder, this reads as: if e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1 + e_2$ is a well-formed expression

Aside: Production Rules and Syntax

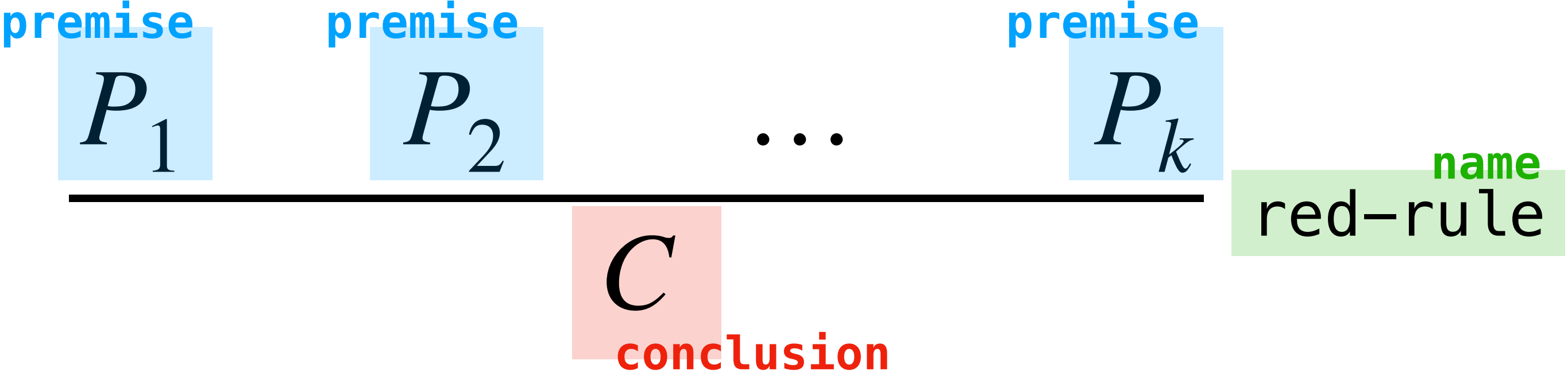
$$\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{expr} \rangle$$

Last time, we saw the above notation. This is called a ***production rule*** and is part of a ***BNF grammar***

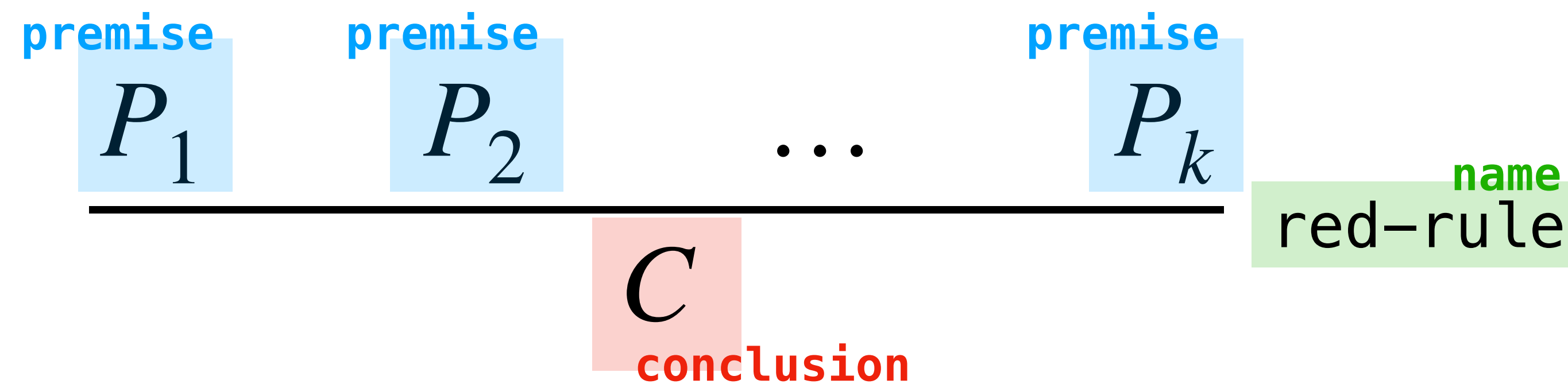
Reminder, this reads as: if e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1 + e_2$ is a well-formed expression

We won't focus on this until the second half of the course but you should start to get comfortable with the syntax

Inference Rules

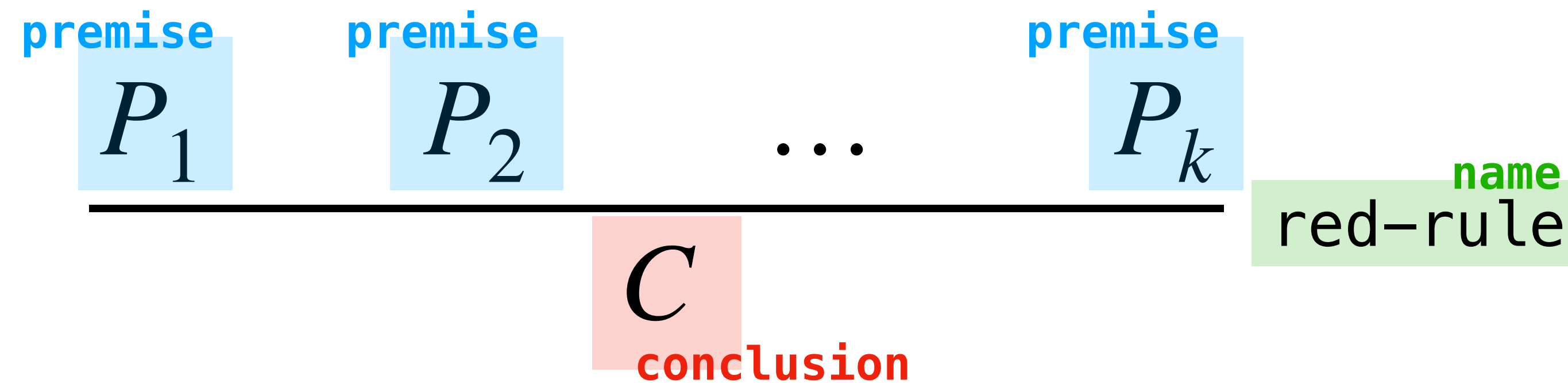


Inference Rules



Then general form of a reduction rule has a collection of **premises** and a **conclusion**

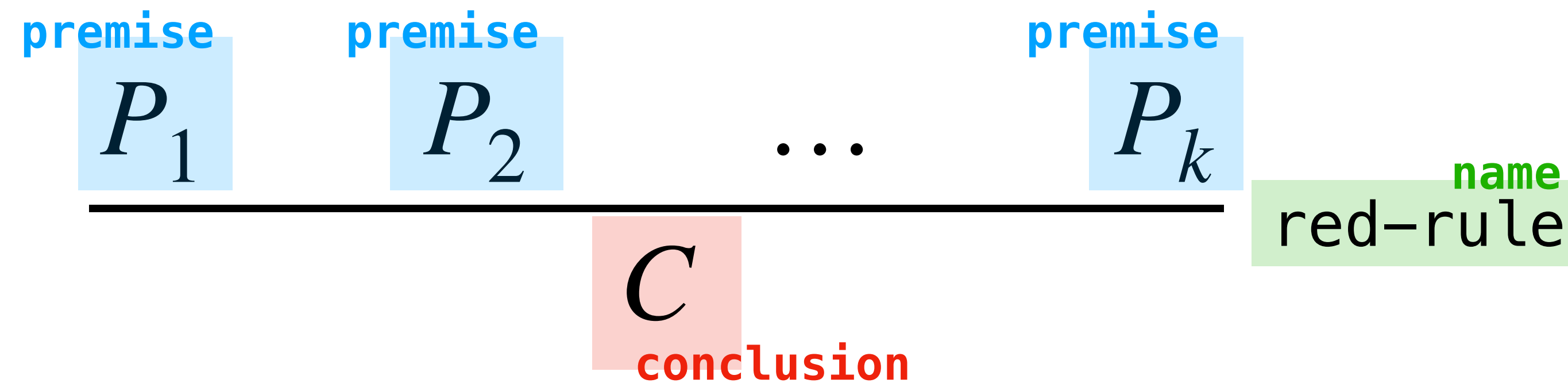
Inference Rules



Then general form of a reduction rule has a collection of **premises** and a **conclusion**

There may be no premises, this is called an **axiom**

Inference Rules

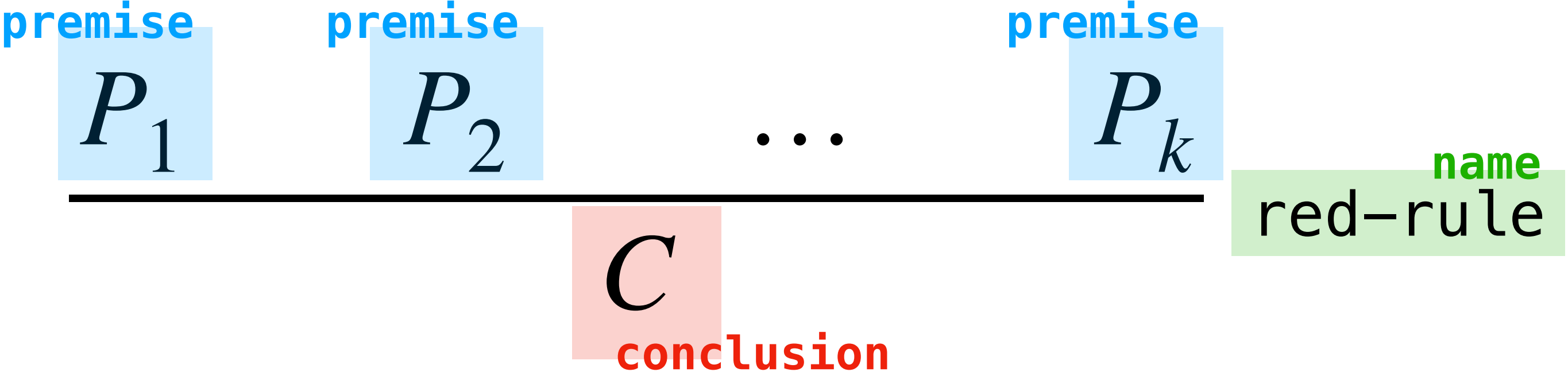


Then general form of a reduction rule has a collection of **premises** and a **conclusion**

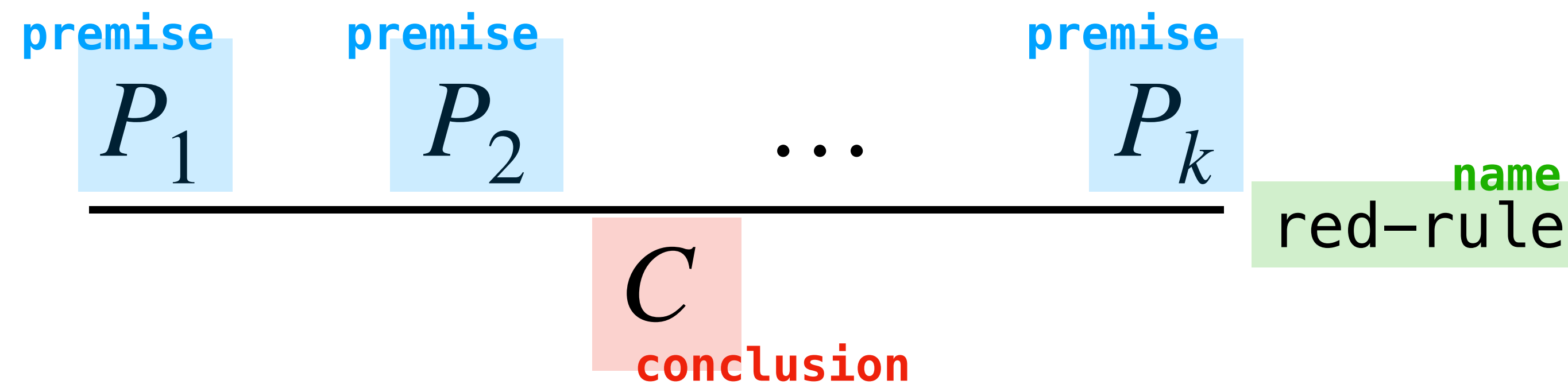
There may be no premises, this is called an **axiom**

Premises which are not of the same form as the conclusion are called **side-conditions**

Inference Rules

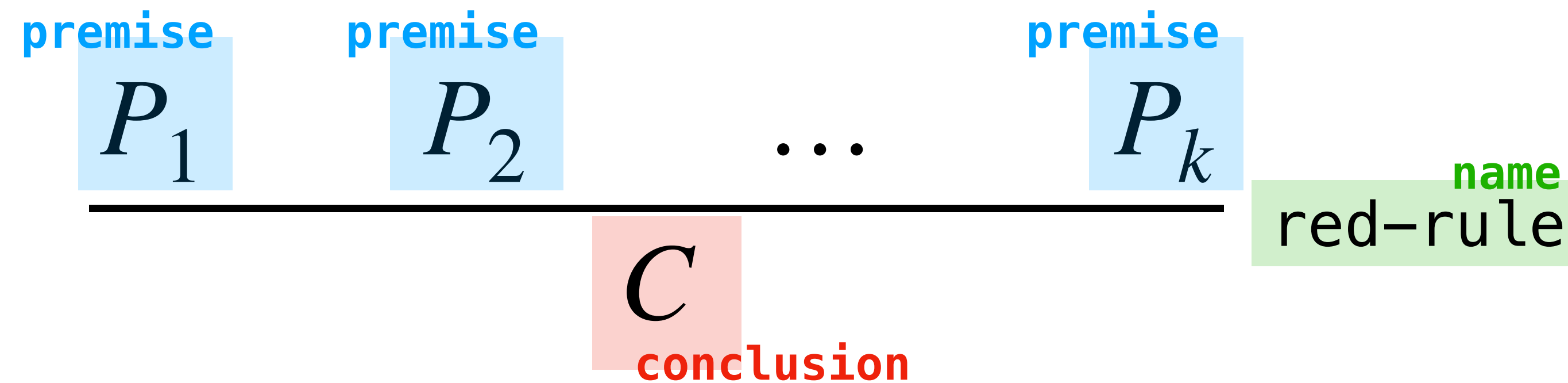


Inference Rules



We can read this as:

Inference Rules



We can read this as:

If P_1 through P_k hold, then C holds (by the **red-rule** rule).

Typing Judgments

$$\text{context } \Gamma \vdash \text{expression } e : \text{type } \tau$$

A typing judgment is a compact way of representing the statement:

e is of type τ in the context Γ

A **typing rule** is an inference rule whose premises and conclusion are typing judgments

Recall: Integer Addition Typing Rule

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \text{ (addInt)}$$

If e_1 is an *int* (in any context Γ) and e_2 is an *int* then (in any context Γ) $e_1 + e_2$ is an *int* (in any context Γ)

Contexts

$\Gamma = \{ x : \text{int}, y : \text{string}, z : \text{int} \rightarrow \text{string} \}$

Contexts

$$\Gamma = \{ x : \text{int}, y : \text{string}, z : \text{int} \rightarrow \text{string} \}$$

A ***context*** is a set of ***variable declarations***

Contexts

$$\Gamma = \{ x : \text{int}, y : \text{string}, z : \text{int} \rightarrow \text{string} \}$$

A **context** is a set of **variable declarations**

A variable declaration $(x : \tau)$ says: "I declare that the variable x is of type τ "

Contexts

$$\Gamma = \{ x : \text{int}, y : \text{string}, z : \text{int} \rightarrow \text{string} \}$$

A **context** is a set of **variable declarations**

A variable declaration $(x : \tau)$ says: "I declare that the variable x is of type τ "

A context is a keeps track of all the types of variables in the "environment"

Example: Reading Typing Judgements

$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$

Example: Reading Typing Judgements

$$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$$

In English: *Given I declare that b is a bool , the expression $\text{if } b \text{ then } 2 \text{ else } 3$ is an int*

Example: Reading Typing Judgements

$$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$$

In English: *Given I declare that b is a bool , the expression $\text{if } b \text{ then } 2 \text{ else } 3$ is an int*

The context tells us what the types of variables need to be in order to be able to determine the type of the whole expression

Judgements are Statements

$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$

Judgements are Statements

$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$

A judgement is a *statement* in the same way that "there are infinitely many twin primes" is a statement

Judgements are Statements

$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$

A judgement is a *statement* in the same way that "there are infinitely many twin primes" is a statement

We haven't **proved** anything by writing down a typing judgment

Judgements are Statements

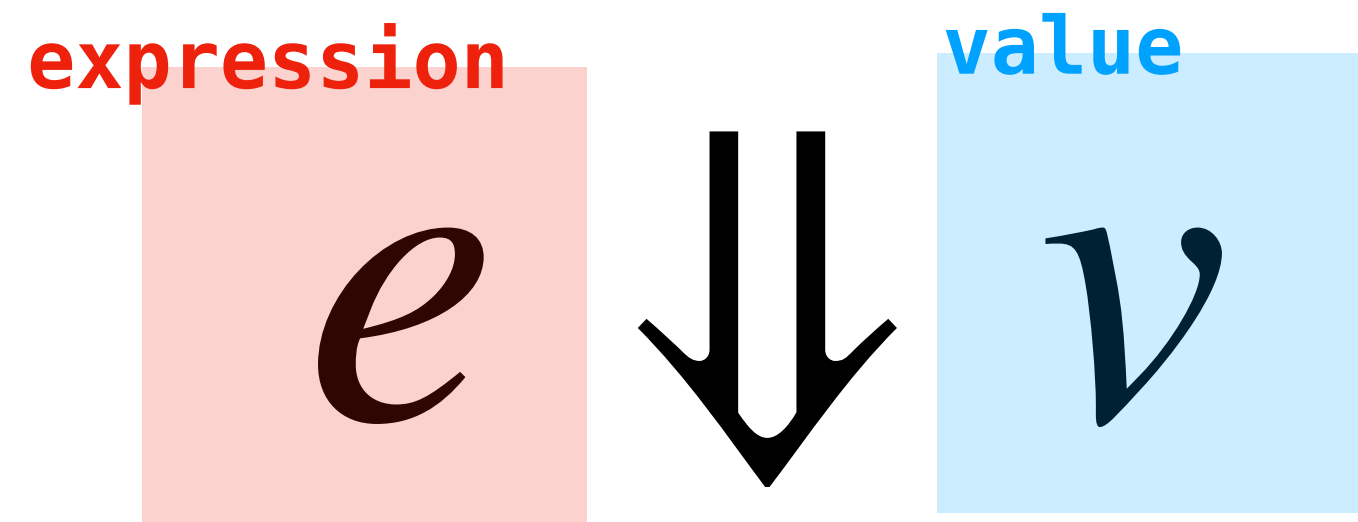
$\{b : \text{bool}\} \vdash \text{if } b \text{ then } 2 \text{ else } 3 : \text{int}$

A judgement is a *statement* in the same way that "there are infinitely many twin primes" is a statement

We haven't **proved** anything by writing down a typing judgment

Next week: We will talk about **typing derivations**, which are used to demonstrate that expressions *actually* have their desired types in our PL

Semantic Judgements



A semantic judgment is a compact way of representing the statement:

The expression e evaluates to the value v

A **semantic rule** is an inference rule with semantic judgments

Example: Reading Semantic Judgments

`if 2 > 3 then 2 + 2 else 3` \Downarrow 3

In English: The expression `if 2 > 3 then 2 + 2 else 3` evaluates to the value 3

Note: we make a *strong distinction between expressions and values*

let's try formally
specifying literals

Literal Syntax

$$\langle \text{expr} \rangle ::= \langle \text{int-lit} \rangle \mid \text{true} \mid \text{false}$$

We'll be a bit less formal about literal syntax (we'll see why this is the case when we talk about *lexing*)

For example, we'll say: *An integer literal is a well-formed expression*

(without saying what an integer literal actually is)

This rule allows to say that **23** is well-formed expression

That said, for booleans we can be formal: **true** and **false** are well-formed expressions

Literal Typing Rules

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{ (trueLit)}$$
$$\frac{}{\Gamma \vdash \text{false} : \text{bool}} \text{ (falseLit)}$$
$$\frac{n \text{ is an integer literal}}{\Gamma \vdash n : \text{int}} \text{ (intLit)}$$

Literal Typing Rules

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{ (trueLit)}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{bool}} \text{ (falseLit)}$$

$$\frac{n \text{ is an integer literal}}{\Gamma \vdash n : \text{int}} \text{ (intLit)}$$

Above are the typing rules for just Boolean and integers. Each rule expresses that a *literal has its expected type in any context*

Literal Typing Rules

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}} \text{ (trueLit)}$$
$$\frac{}{\Gamma \vdash \text{false} : \text{bool}} \text{ (falseLit)}$$
$$\frac{n \text{ is an integer literal}}{\Gamma \vdash n : \text{int}} \text{ (intLit)}$$

Above are the typing rules for just Boolean and integers. Each rule expresses that a *literal has its expected type in any context*

Part in **green** is called a **side condition**. It's something that has to hold for the inference rule to make sense, but *does not* appear in the application of any rule (*this will be more clear when we talk about derivations*)

Literal Semantics Rules

$$\frac{}{\text{true} \Downarrow \top} \text{ (trueLitEval)} \qquad \frac{}{\text{false} \Downarrow \perp} \text{ (falseLitEval)}$$

$$\frac{n \text{ is an integer literal}}{n \Downarrow n} \text{ (intLitEval)}$$

These rules express that *literals evaluate to the values they denote*

This takes some time to get used to but we're going to make a **strong distinction between expressions and values**

`true` does not evaluate to `true` (even though that's what UTop says) it evaluates the true Boolean value, which we will write as \top

and that's it, we've formally
specified (integer and boolean)
literals

We'll see more typing rules and semantic rules, we'll also give a written reference for the specification we'll talk about in class

Summary

- » OCaml is a language of expressions, everything is an expression
- » OCaml has everything we need to do basic programming
- » We use inference rules to formally specify the behavior or typing and evaluation