OCaml: The Basics

Concepts of Programming Languages
Lecture 2

CAS CS 320

Announcements

- You must attend **both** the in-class quizzes
- We will follow the online book: *OCaml Programming: Correct + Efficient + Beautiful*
- Hope you all have installed OCaml

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**
- Write basic OCaml programs on primitive types

Recall: Expressions

In OCaml and (functional languages in general), **everything is an expression**

Functions, variables, arguments, assignments, etc.

All expressions have a value

$$2 + (2 * 3)$$

if x = 3 then 4 else 5

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

An OCaml Program is an Expression

Therefore, it has a type
And a value

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 is a build tool for OCaml

It allows us to specify project-level dependencies and configurations

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 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 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 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 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 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 is a build tool for OCaml

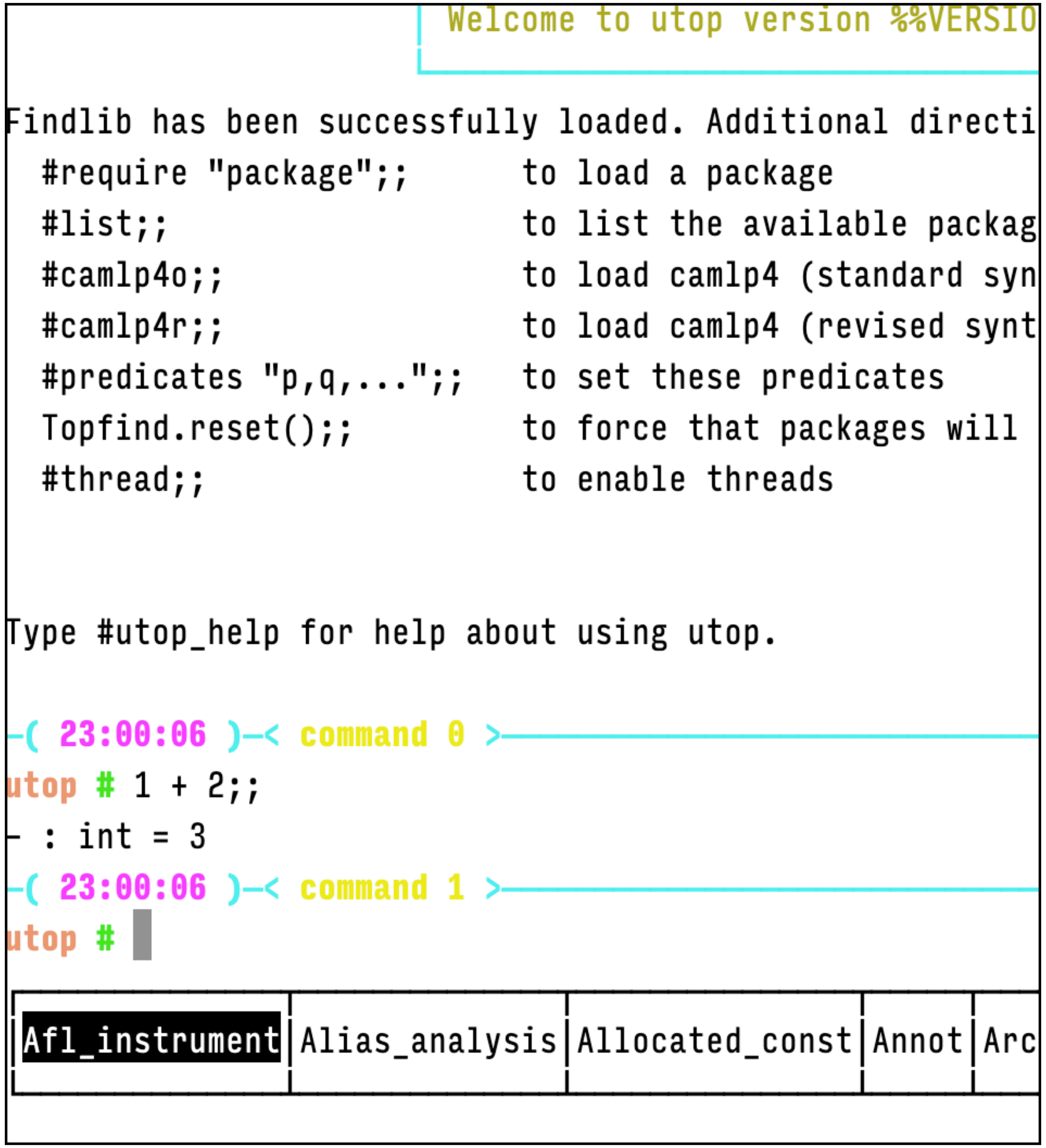
It allows us to specify project-level dependencies and configurations

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)

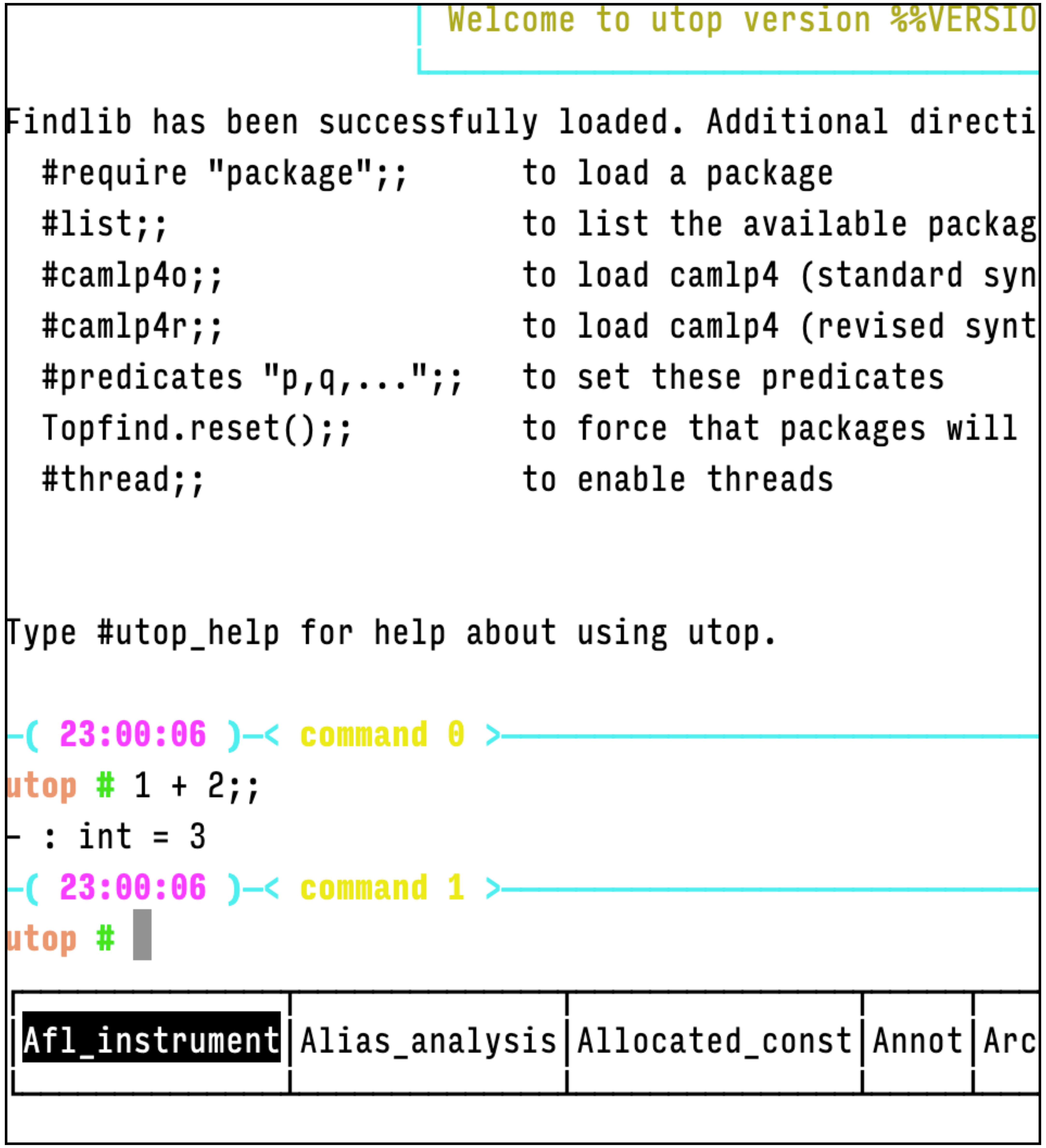
UTop



Welcome to utop version %%VERSIO
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

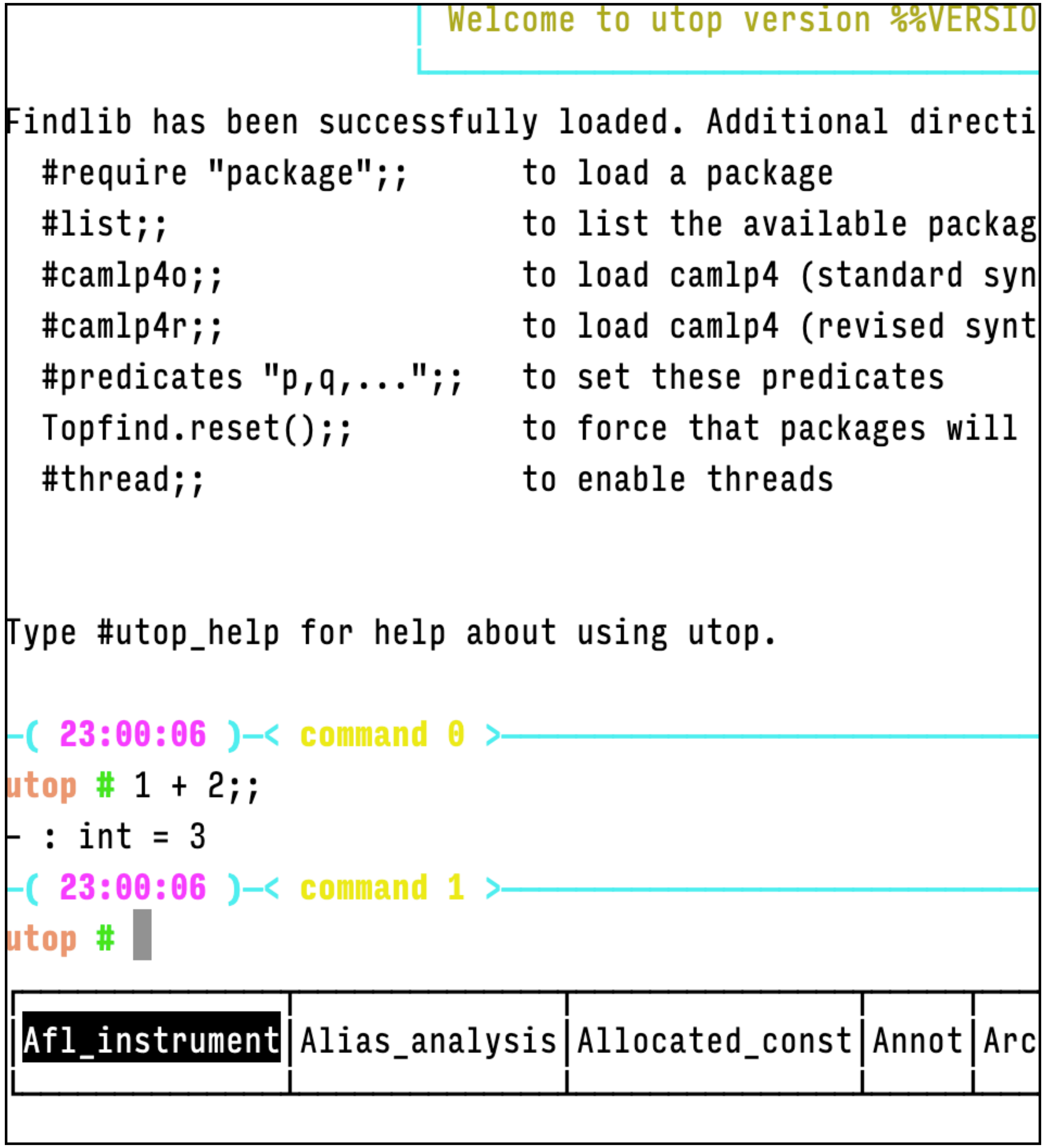
UTop is an interface for the OCaml toplevel (use utop in terminal)



UTop

UTop is an interface for the OCaml toplevel (use utop in terminal)

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

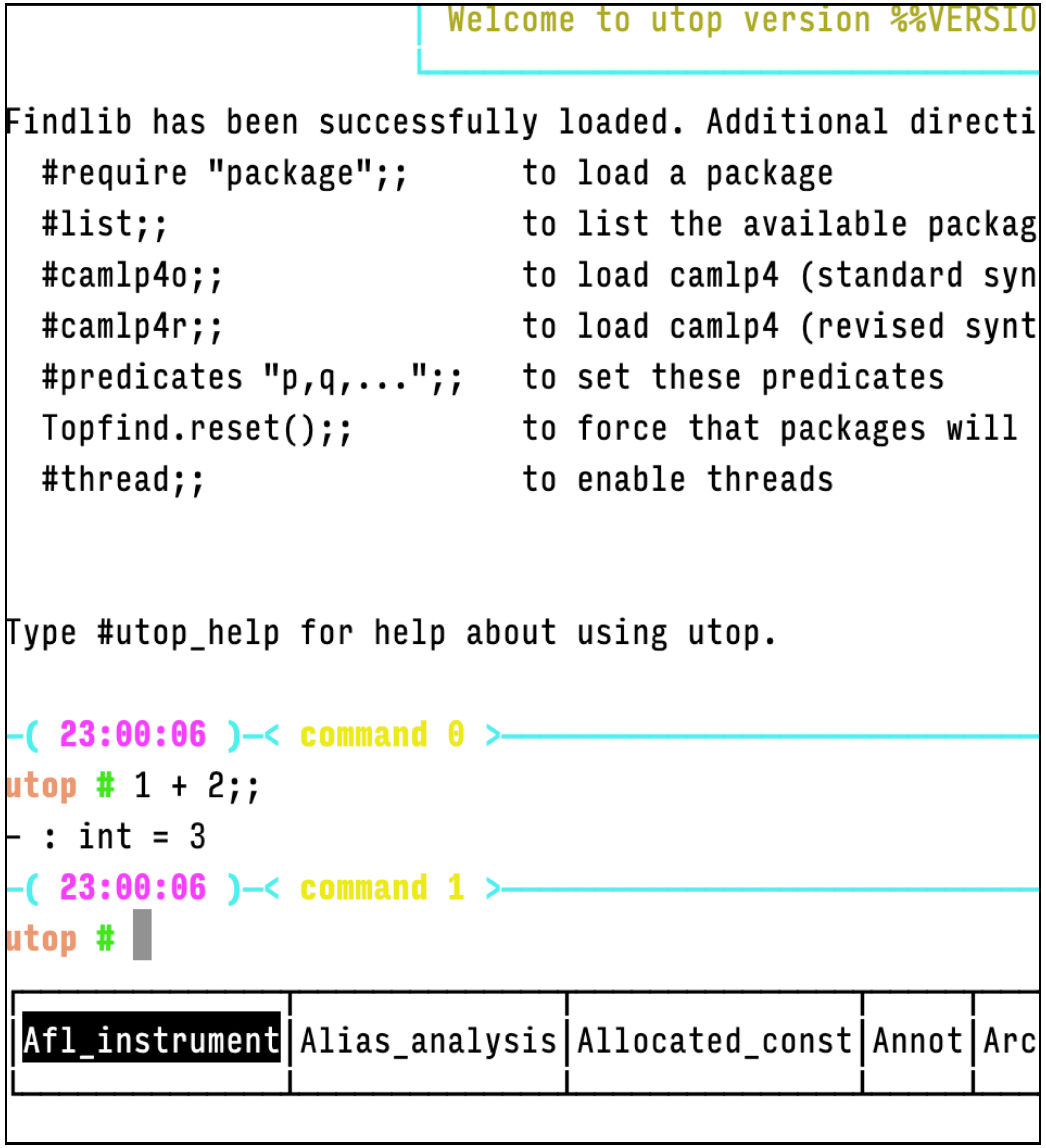


UTop

UTop is an interface for the OCaml toplevel (use utop in terminal)

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

Cheatsheet:



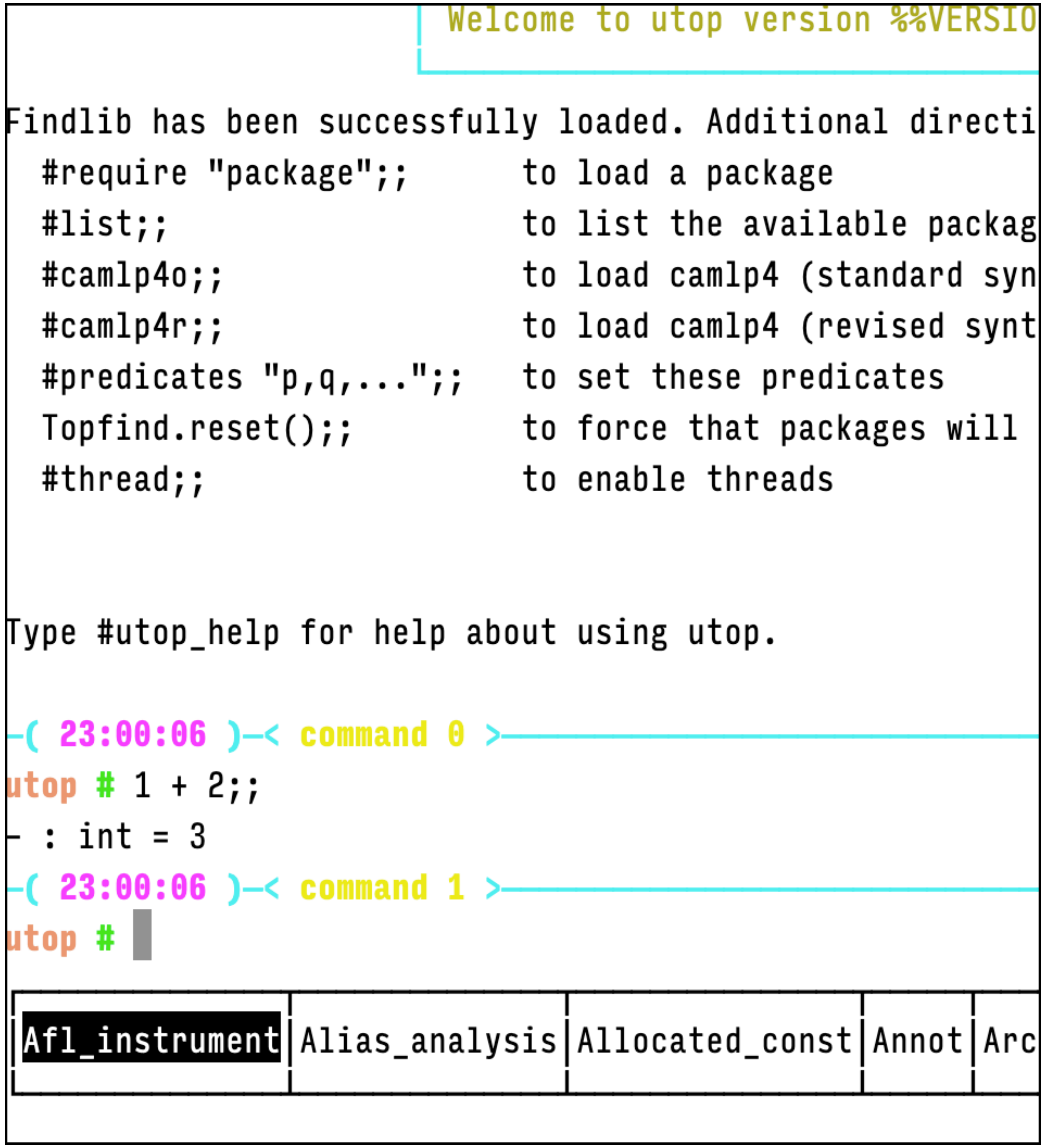
UTop

UTop is an interface for the OCaml toplevel (use `utop` in terminal)

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

Cheatsheet:

- expressions must be followed with two semicolons



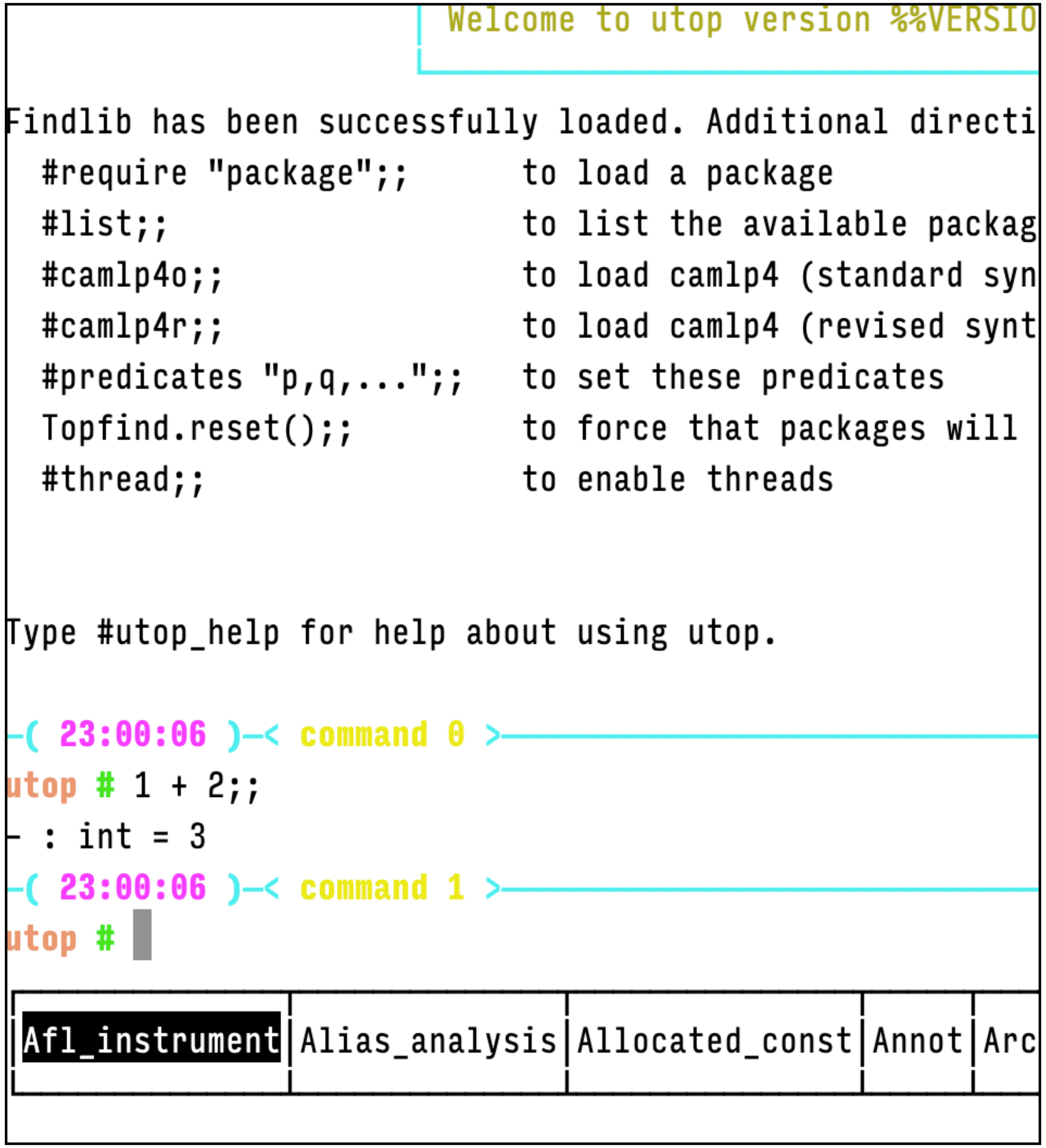
UTop

UTop is an interface for the OCaml toplevel (use utop in terminal)

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



Expressions (Informally)

High Level View

`Each expression has a` **form**
`(e.g. 2 + 7, true, “true”)`
`[defined by` **syntax rules**`]`

`Each expression has a` **unique type**
`(e.g. 2 + 7 : int, true : bool, “true” : string)`
`[defined by typing rules]`

`Each expression has a` **unique value**
(e.g. 2 + 7 $\Downarrow$ 9, true $\Downarrow$ true, “true” $\Downarrow$ “true”)
`[defined by semantics rules]`

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	&&, ll, not
char	'b', 'c'	
string	"word", "@*&#"	^

demo

(simple use of primitive types)

A Couple Notes on Operators

A Couple Notes on Operators

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

A Couple Notes on Operators

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

OCaml has **no operator overloading (bad feature)**

A Couple Notes on Operators

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

OCaml has **no operator overloading (bad feature)**

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

A Couple Notes on Operators

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

OCaml has **no operator overloading (bad feature)**

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 `!=`)

A Note on Type Annotations

```
let x : int = 2 + 7
let y : bool = true
let z : string = “true"
```

A Note on Type Annotations

```
let x : int = 2 + 7
let y : bool = true
let z : string = “true"
```

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

A Note on Type Annotations

```
let x : int = 2 + 7
let y : bool = true
let z : string = “true"
```

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

That said, you **should** include type annotations, especially at the beginning, because they're useful for *documentation* and for *code clarity*

First Abstraction of the Day

Let Expressions and Variables

Local Variables

body

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

Local Variables

body

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

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

Local Variables

body

```
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

Local Variables

body

```
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

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

Python

```ocaml
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.

Local Variables (Informal)

body

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

Local Variables (Informal)

body

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

syntax: `let VARIABLE = EXPRESSION in BODY`

Local Variables (Informal)

body

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

syntax: `let VARIABLE = EXPRESSION in BODY`

typing: *compute type of* EXPRESSION*; assume* VARIABLE *has that type; compute type of* BODY

Local Variables (Informal)

body

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

syntax: `let VARIABLE = EXPRESSION in BODY`

typing: *compute type of* `EXPRESSION`*; assume* `VARIABLE` *has that type; compute type of* `BODY`

semantics: *compute value of* `EXPRESSION`*; substitute that value for* `VARIABLE` *in* `BODY`

Syntax Examples

Syntax Examples

body

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

Syntax Examples

body

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

body

```
let x = true in if x then 3 else 4
```

Syntax Examples

body

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

body

```
let x = true in if x then 3 else 4
```

body

```
let x = 3.5 in 2. *. x
```

Syntax Examples

body

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

body

```
let x = true in if x then 3 else 4
```

body

```
let x = 3.5 in 2. *. x
```

body body

```
let y = (let x = 2 in x + x) in 4 * y
```

Example: Ill-Typed Let-Expression

```
let x = 2. in 3 + x
```

An ill-typed expression will throw a type error when you type it into `utop`

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

Semantics Examples

Semantics Examples

`let x = 2 in x + x` $\longrightarrow$ `2 + 2`

Semantics Examples

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

→

```
2 + 2
```

```
let x = true in
if x then 3 else 4
```

→

```
if true then 3 else 4
```

Semantics Examples

`let x = 2 in x + x` ⟶ `2 + 2`

```
let x = true in
if x then 3 else 4
```
⟶ `if true then 3 else 4`

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

Semantics Examples

`let x = 2 in x + x` ⟶ `2 + 2`

```
let x = true in
if x then 3 else 4
```
⟶ `if true then 3 else 4`

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**

Semantics Examples

`let x = 2 in x + x` ⟶ `2 + 2`

```
let x = true in
if x then 3 else 4
```

⟶ `if true then 3 else 4`

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

(simple use of lets)

Second Abstraction of the Day

If Expressions

If-Expressions

```
if x > 0 then x else -x
```

```
if x <> y then x+y else x-y
```

If-Expressions

```
if x > 0 then x else -x
```

```
if x <> y then x+y else x-y
```

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

If-Expressions

```
if x > 0 then x else -x
```

```
if x <> y then x+y else x-y
```

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

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)

If-Expressions (Informal)

```
if x > 0 then x else -x
```

If-Expressions (Informal)

```
if x > 0 then x else -x
```

Syntax: `if CONDITION then TRUE–CASE else FALSE–CASE`

If-Expressions (Informal)

```
if x > 0 then x else -x
```

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

Typing: CONDITION *must be a Boolean; compute the types of* TRUE–CASE *and* FALSE–CASE*; must be the same type; expression type is same as that of* TRUE–CASE *and* FALSE–CASE

If-Expressions (Informal)

```
if x > 0 then x else -x
```

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

Typing: CONDITION *must be a Boolean; compute the types of* TRUE–CASE *and* FALSE–CASE*; must be the same type; expression type is same as that of* TRUE–CASE *and* FALSE–CASE

Semantics: *If* CONDITION *evaluates to true; evaluate* TRUE–CASE*, else evaluate* FALSE–CASE

demo

(simple use of ifs)

Third (and Most Important) Abstraction of the Day

Functions

Functions (Informal)

body

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

Syntax: `fun VARIABLE -> EXPRESSION`

Typing: assume VARIABLE has type **T1;** compute the type of EXPRESSION; suppose it is **T2;** type of function is **T1 -> T2**

Semantics: A function is a value; evaluates to itself

Functions are also Expressions

body

```
fun x -> 3 + x
```

body

```
fun y -> 2. *. x
```

body

```
fun x y z -> if x then y else z
```

body

```
fun x -> if x > 0 then x else -x
```

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

We Can Give them Names using Let

body

```
let f = fun x -> 3 + x
```

body

```
let g = fun y -> 2. *. x
```

body

```
let h = fun x y z -> if x then y else z
```

body

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

Another Way to Define Functions

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

```
let h = fun x y z -> if x then y else z
```

Another Way to Define Functions

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

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

```
let h = fun x y z -> if x then y else z
```

Another Way to Define Functions

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

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

```
let h = fun x y z -> if x then y else z
```

```
let h x y z = if x then y else z
```

Another Way: Curried Functions

```
let h = fun x y z -> if x then y else z
```

```
let h = fun x -> fun y -> fun z -> if x then y else z
```

Another way of thinking about functions:

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

This seems restrictive, but ultimately it doesn't affect us <u>at all</u>

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

Curried Functions Return 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)

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

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: *compute type of* FUNCTION–EXPR*; say it is* **T1 –> T2***; compute type of* ARG–EXPR*; it must be* **T1***; then the type of expression is* **T2**

Application (Informally)

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

Syntax: FUNCTION–EXPR ARG–EXPR

Typing: *compute type of* FUNCTION–EXPR*; say it is* **T1 –> T2***; compute type of* ARG–EXPR*; it must be* **T1***; then the type of expression is* **T2**

Semantics: Evaluate ARG–EXPR; substitute its value into the body of FUNCTION–EXPR; evaluate the result

Application (Example)

Application (Example)

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

Application (Example)

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

is the same as

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

Application (Example)

(fun x -> fun y -> x + y + 1) 3 2 *is the same as*

(fun x -> (fun y -> x + y + 1)) 3 2 *is the same as*

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

Application (Example)

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

is the same as

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

is the same as

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

Application (Example)

```
(fun x ->  fun y -> x + y  + 1)  3  2
```
is the same as

```
(fun x -> (fun y -> x + y  + 1)) 3  2
```
is the same as

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

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

Application (Example)

(fun x -> fun y -> x + y + 1) 3 2 *is the same as*

(fun x -> (fun y -> x + y + 1)) 3 2 *is the same as*

((fun x -> (fun y -> x + y + 1)) 3) 2 *evaluates to*

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

Application (Example)

(fun x -> fun y -> x + y + 1) 3 2 *is the same as*

(fun x -> (fun y -> x + y + 1)) 3 2 *is the same as*

((fun x -> (fun y -> x + y + 1)) 3) 2 *evaluates to*

(fun y -> 3 + y + 1) 2 *evaluates to*

3 + 2 + 1

Application (Example)

(fun x -> fun y -> x + y + 1) 3 2 *is the same as*

(fun x -> (fun y -> x + y + 1)) 3 2 *is the same as*

((fun x -> (fun y -> x + y + 1)) 3) 2 *evaluates to*

(fun y -> 3 + y + 1) 2 *evaluates to*

3 + 2 + 1 *is the same as*

(3 + 2) + 1

Application (Example)

(fun x -> fun y -> x + y + 1) 3 2 *is the same as*

(fun x -> (fun y -> x + y + 1)) 3 2 *is the same as*

((fun x -> (fun y -> x + y + 1)) 3) 2 *evaluates to*

(fun y -> 3 + y + 1) 2 *evaluates to*

3 + 2 + 1 *is the same as*

(3 + 2) + 1 *evaluates to*

5 + 1

Application (Example)

(fun x -> fun y -> x + y + 1) 3 2	*is the same as*
(fun x -> (fun y -> x + y + 1)) 3 2	*is the same as*
((fun x -> (fun y -> x + y + 1)) 3) 2	*evaluates to*
(fun y -> 3 + y + 1) 2	*evaluates to*
3 + 2 + 1	*is the same as*
(3 + 2) + 1	*evaluates to*
5 + 1	*evaluates to*
6	

demo

(anonymous and curried functions)

Summary

OCaml is a language of **expressions**, everything is an expression

OCaml has everything we need to do basic programming