OCaml: The Basics

Concepts of Programming Languages
Lecture 2

CAS CS 320

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 <u>syntactic</u> objects which describe values in a program

Mneumonic: ***E****xpressions are* ***EV****aluated to* ***V****alues*

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 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

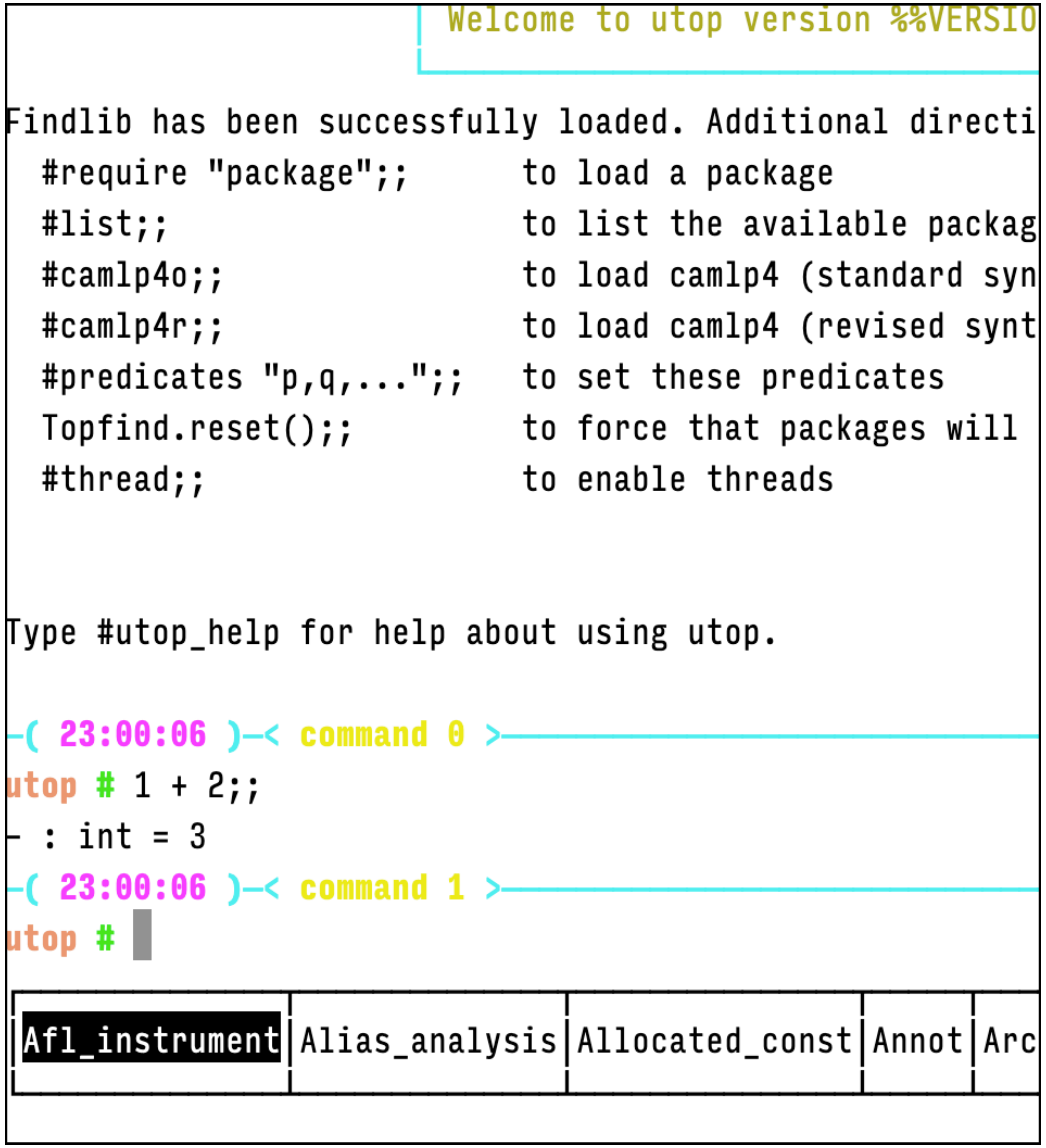
It allows us to specify project-level dependencies and configurations

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)

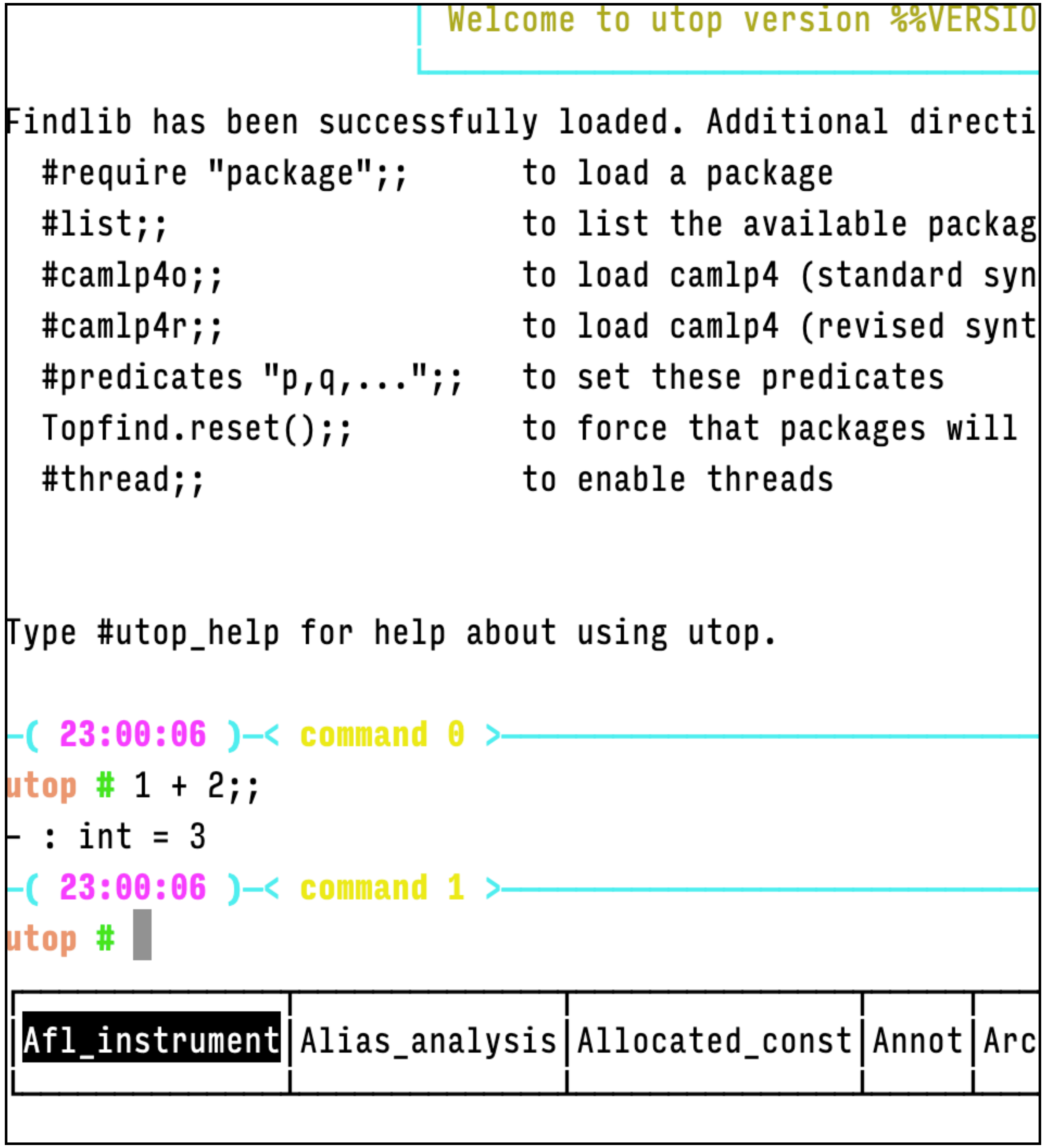
UTop



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

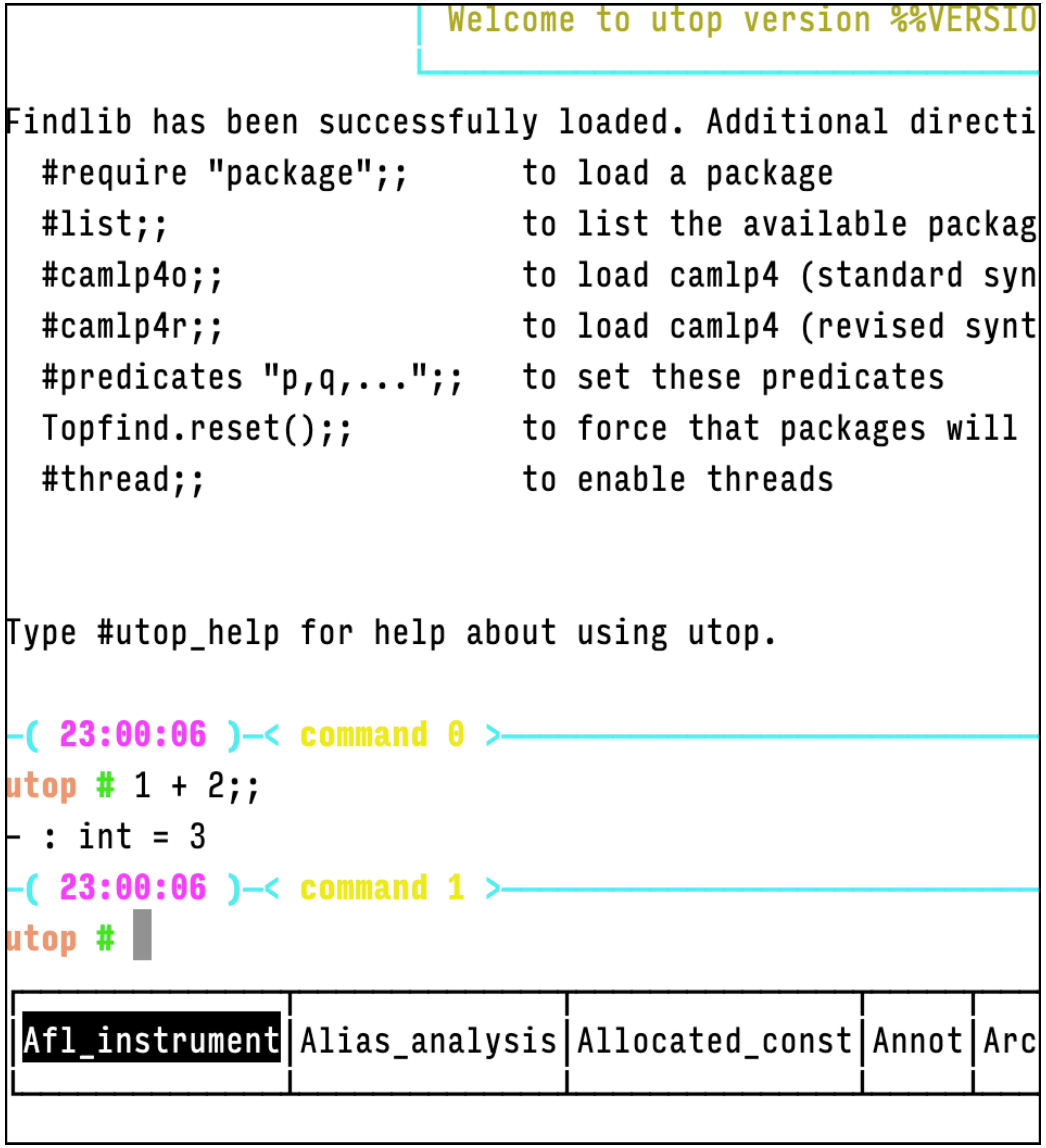
UTop is an interface for the OCaml toplevel



UTop

UTop is an interface for the OCaml toplevel

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

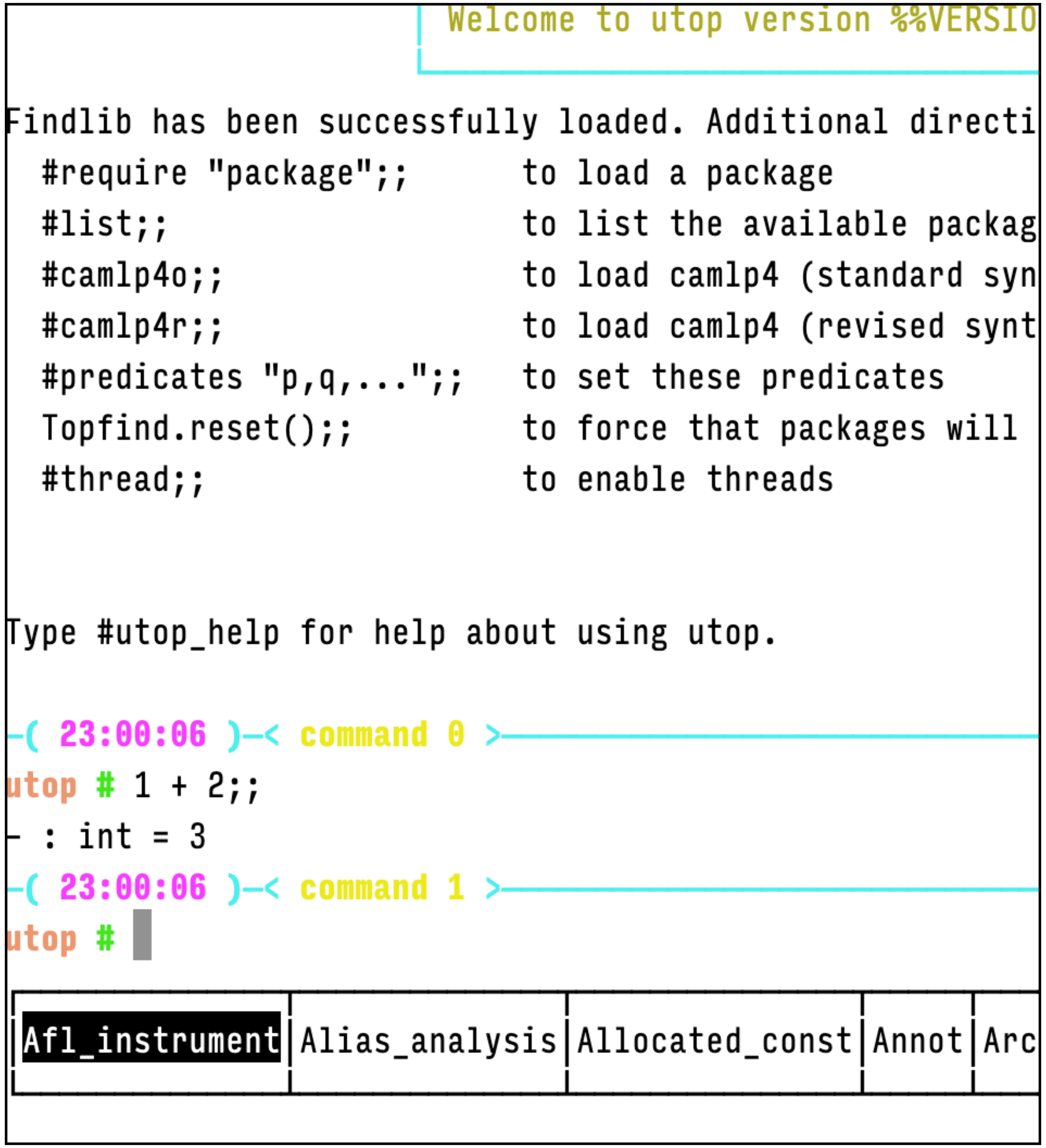


UTop

UTop is an interface for the OCaml toplevel

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

Cheatsheet:



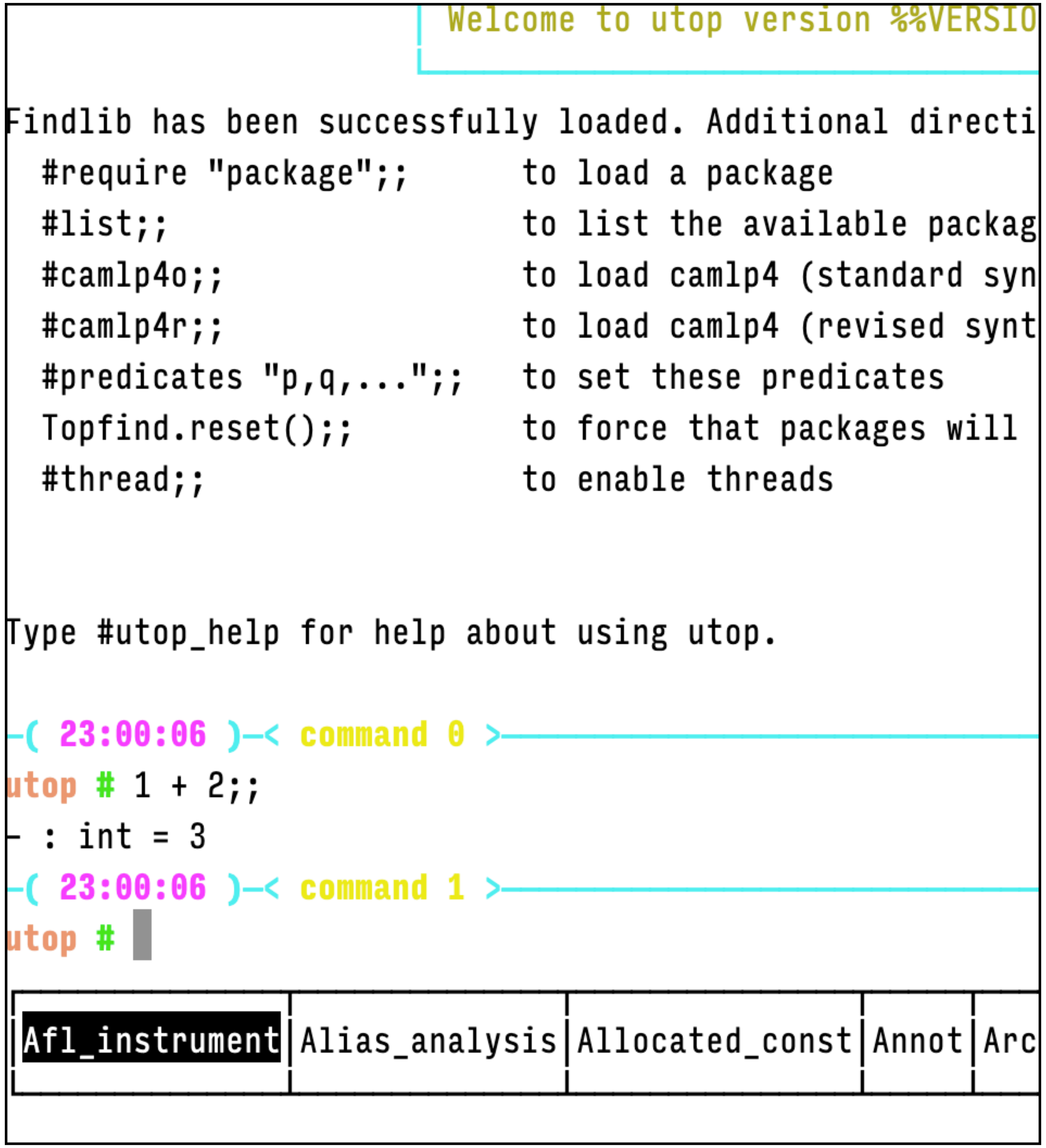
UTop

UTop is an interface for the OCaml toplevel

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

<u>Cheatsheet:</u>

- expressions must be followed with two semicolons



UTop

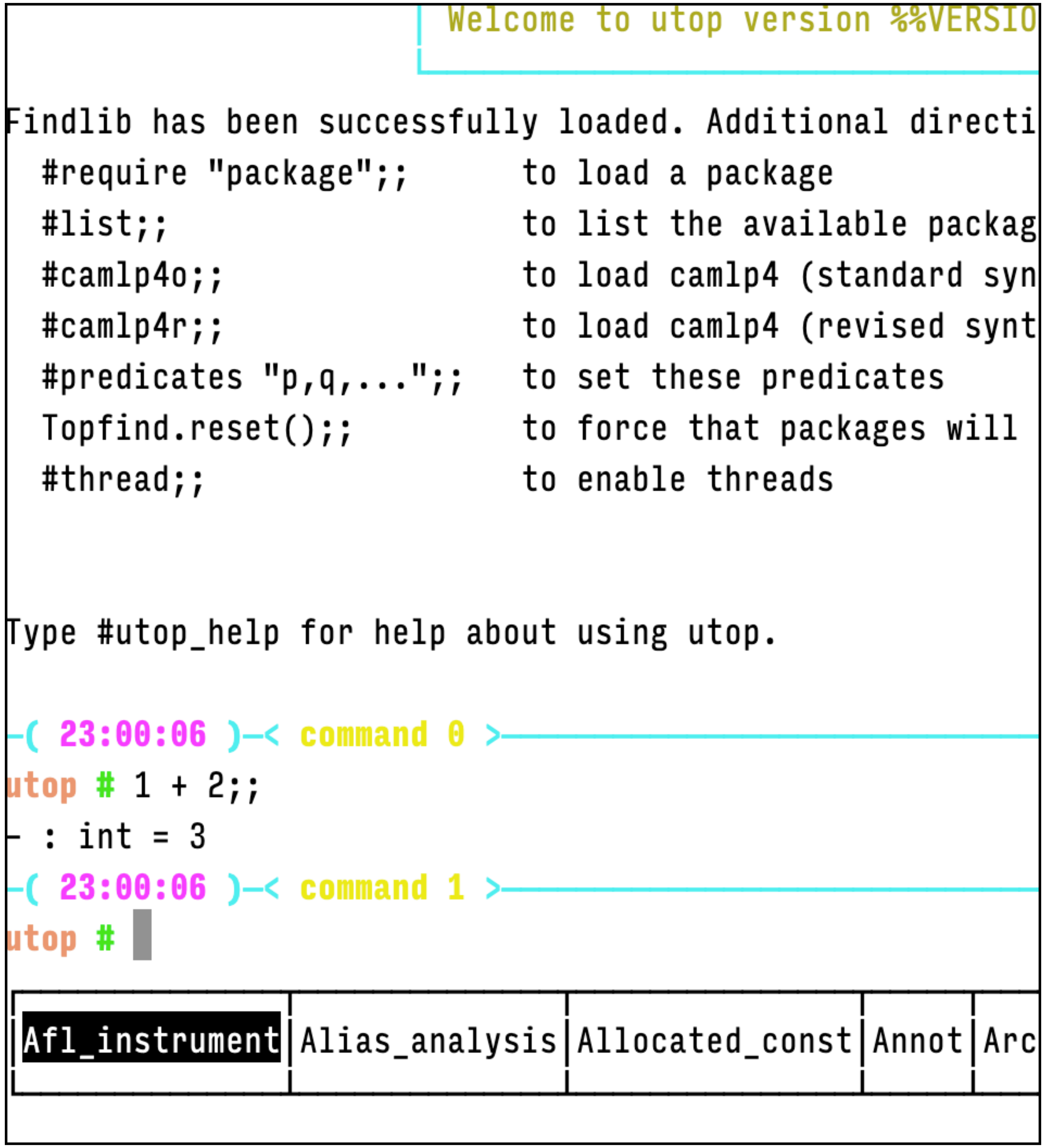
UTop is an interface for the OCaml toplevel

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

<u>Cheatsheet:</u>

» expressions must be followed with two semicolons

» #quit;; or (Ctl-D) leaves UTop



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

```ocaml
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 <u>type</u>

Types

```ocaml
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 <u>type</u>

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

Types

```ocaml
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 <u>type</u>

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

not %

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

Local Variables

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

$\underbrace{\texttt{x + x}}_{\text{body}}$

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

$$\texttt{let x = 2 in } \underbrace{\texttt{x + x}}_{\text{body}}$$

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.

Recall: Anatomy of an OCaml Program

```ocaml
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 <u>top-level let-expressions</u>, 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)

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)

```
let x = 2 in x + x
```
body : int

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)

body

`let x = 2 in x + x` → 2+2

↓

$\boxed{4}$

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

```
let x = 2 in "two" <> x
```

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

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

`let x = 2 in x + x` ⟶ `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

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

$\longrightarrow$

```
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

not valid

if false then 0

if b then 2

else "two"

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

Answer: Remember, all we have is expressions. So every if-expression <u>must have a value and a type</u> (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

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

if-stmt (Python)

```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)

body

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

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)

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

↓

(fun y -> 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 -> 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 -> 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** (<u>without</u> converting to a string)*

Expressions (Formally)

Aside: Production Rules and Syntax

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

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

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

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

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

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

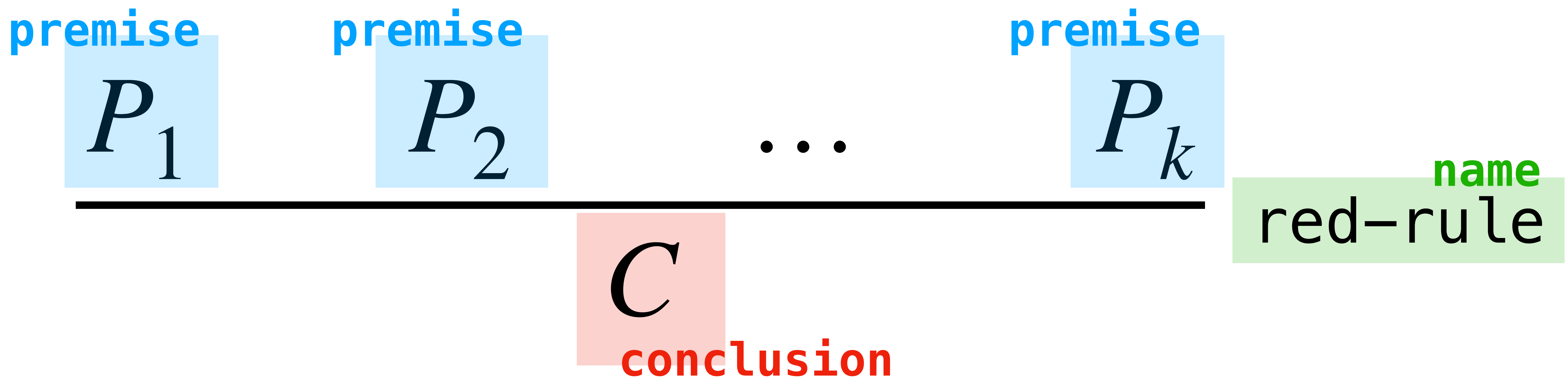
Reminder, this reads as: if e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1 + e_2$ is a well-formed expression

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

$$\frac{\overset{\texttt{premise}}{P_1} \qquad \overset{\texttt{premise}}{P_2} \qquad \cdots \qquad \overset{\texttt{premise}}{P_k}}{\underset{\texttt{conclusion}}{C}}\ \overset{\texttt{name}}{\texttt{red-rule}}$$

Inference Rules



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

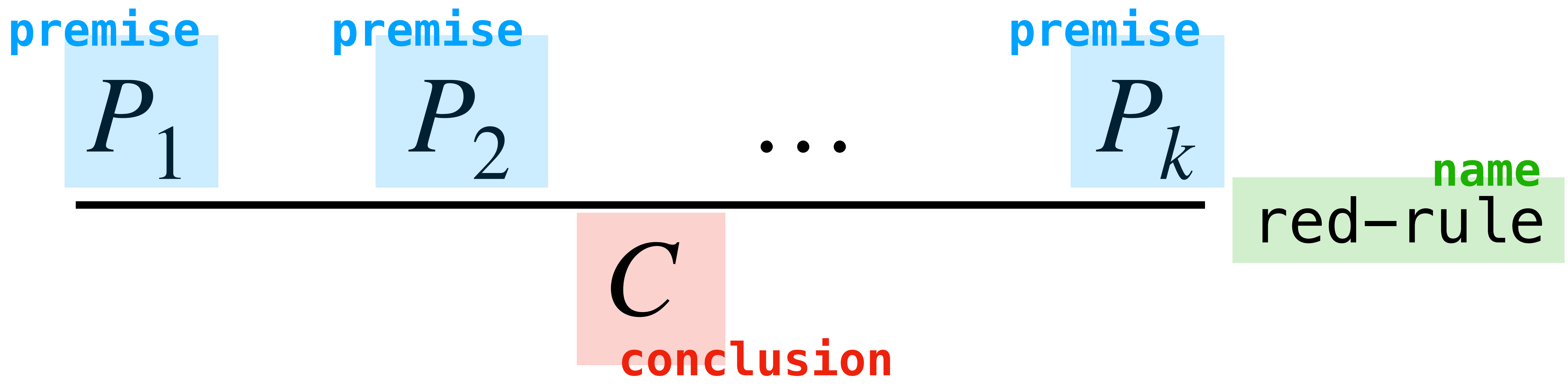
Inference Rules

$$\dfrac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \cdots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}}\;\overset{\text{name}}{\text{red-rule}}$$

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

$$\dfrac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \ldots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}} \;\; \overset{\text{name}}{\texttt{red-rule}}$$

Inference Rules

$$\frac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \cdots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}} \; \overset{\text{name}}{\texttt{red-rule}}$$

We can read this as:

Inference Rules

$$\dfrac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \cdots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}}\ \overset{\text{name}}{\texttt{red-rule}}$$

We can read this as:

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

Typing Judgments

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

A <u>typing judgment</u> 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

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

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

$$\Gamma = \{\ \texttt{x : int, y : string, z : int -> string}\ \}$$

Contexts

$$\Gamma = \{\ \texttt{x : int, y : string, z : int -> string}\ \}$$

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

Contexts

$\Gamma = \{$ `x : int, y : string, z : int -> 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

$$\Gamma = \{\ \texttt{x : int, y : string, z : int -> string}\ \}$$

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

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

Example: Reading Typing Judgements

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

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

Example: Reading Typing Judgements

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

In English: *Given I declare that b is a bool, the expression if b then 2 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

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

Judgements are Statements

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

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

Judgements are Statements

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

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

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

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

$$\underbrace{e}_{\text{expression}} \Downarrow \underbrace{v}_{\text{value}}$$

A <u>semantic judgment</u> 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

$$\texttt{if 2 > 3 then 2 + 2 else 3} \Downarrow 3$$

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

```
<expr> ::= <int-lit> | true | 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

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ (\text{trueLit}) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ (\text{falseLit})$$

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

Literal Typing Rules

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ (\text{trueLit}) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ (\text{falseLit})$$

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

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

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ (\text{trueLit}) \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ (\text{falseLit})$$

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

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

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

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

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