Course Introduction

Concepts of Programming Languages
Lecture 1

CAS CS 320

Outline

- Give an overview of what PL is about
- Take a first look at OCaml

What is a PL?

Fair Question

How would you define a PL?

How would you explain it to your roommate?

How would you answer if you were asked during an interview?

Discuss this with the people around you for 1min

Programmer's view of a PL

```python
def subtraction():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} - {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1 - num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1-num_2}!\n")
        start_game()

def multiplication():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} x {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1*num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1*num_2}!\n")
        start_game()
```

Programmer's view of a PL

» A tool for programming

```python
def subtraction():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} - {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1 - num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1-num_2}!\n")
        start_game()

def multiplication():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} x {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1*num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1*num_2}!\n")
        start_game()
```

Programmer's view of a PL

- » A tool for programming
- » A text-based way of interacting with hardware/a computer

```python
def subtraction():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} - {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1 - num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1-num_2}!\n")
        start_game()

def multiplication():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} x {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1*num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1*num_2}!\n")
        start_game()
```

Programmer's view of a PL

- » A tool for programming
- » A text-based way of interacting with hardware/a computer
- » A way of organizing and working with data

```python
def subtraction():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} - {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1 - num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1-num_2}!\n")
        start_game()

def multiplication():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} x {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1*num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1*num_2}!\n")
        start_game()
```

Programmer's view of a PL

» A tool for programming

» A text-based way of interacting with hardware/a computer

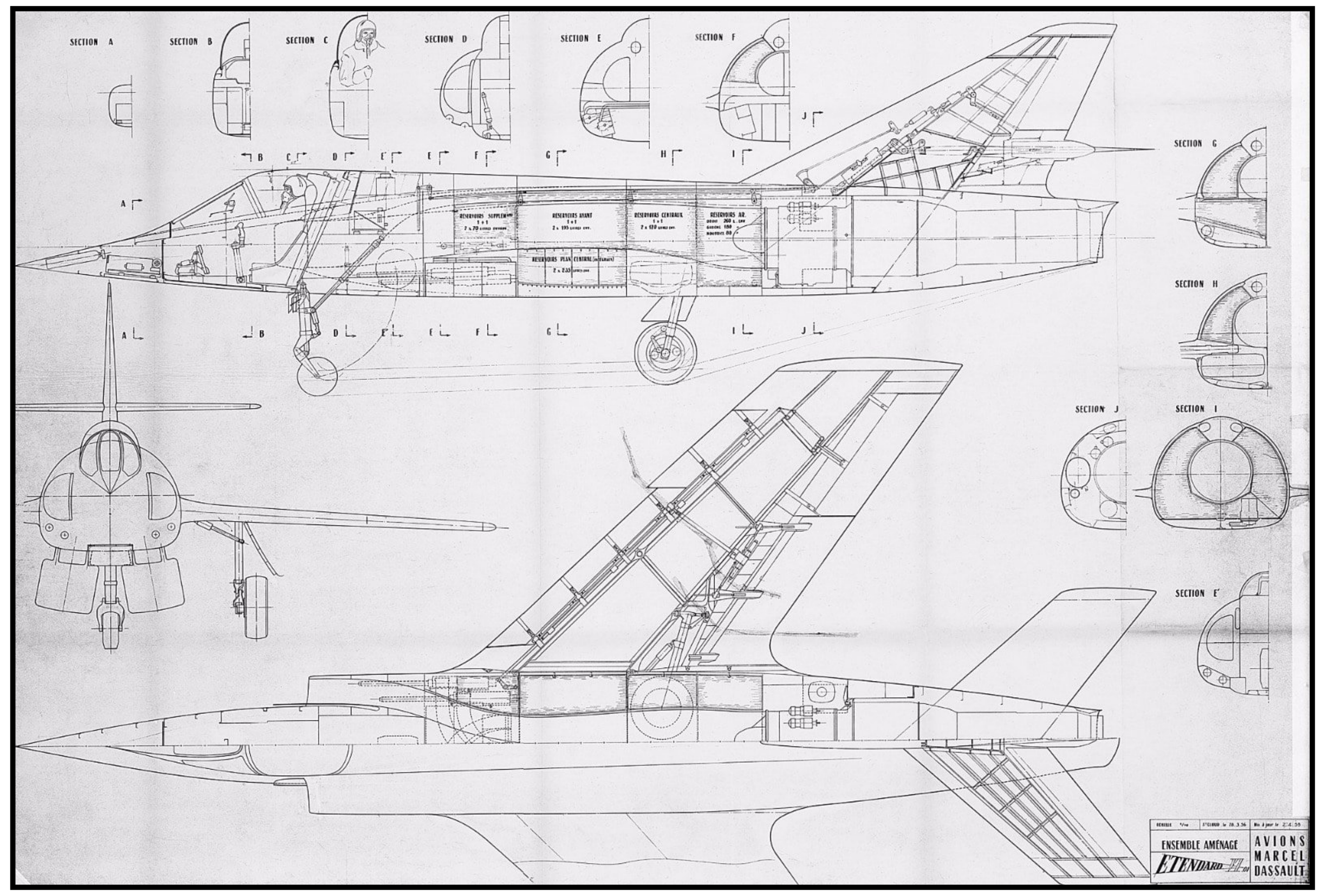
» A way of organizing and working with data

This is not what the course is about

```python
def subtraction():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} - {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1 - num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1-num_2}!\n")
        start_game()

def multiplication():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} x {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1*num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1*num_2}!\n")
        start_game()
```

Users vs. Designers

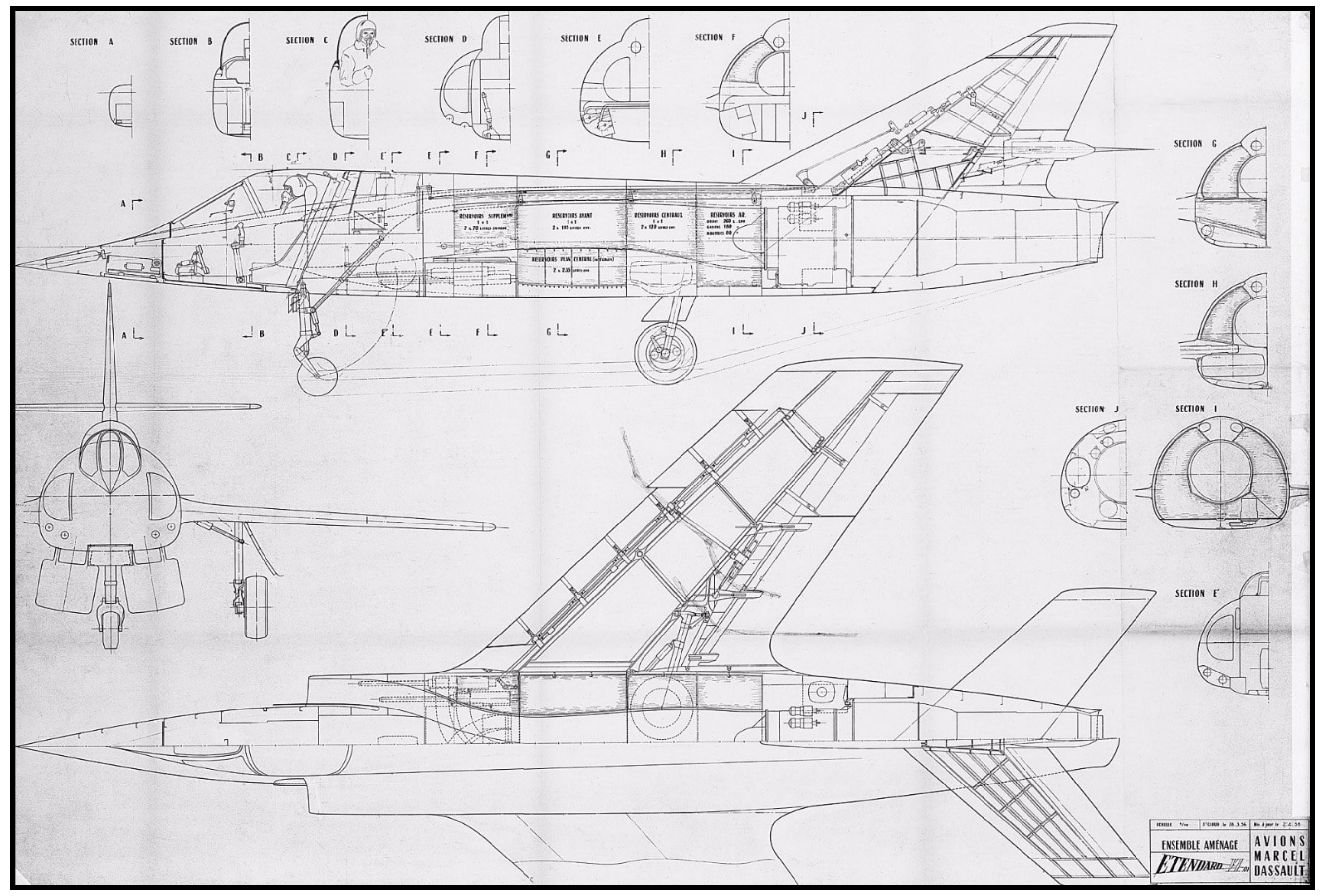
vs.



Users vs. Designers

Programmers *use* PLs. We're interested in **designing** PLs

vs.

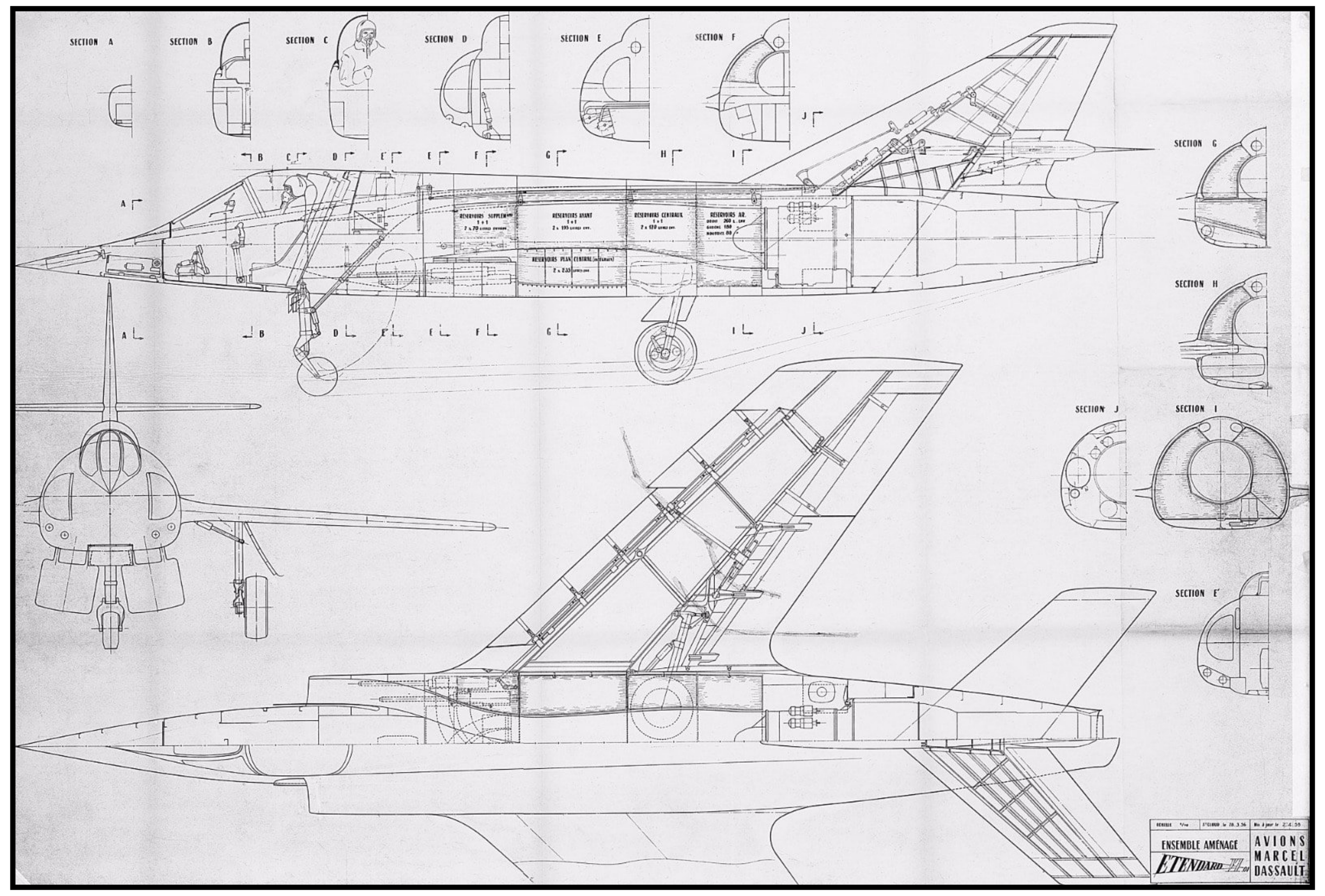


Users vs. Designers

Programmers *use* PLs. We're interested in **designing** PLs

Users are not necessarily the best designers...Who should design PLs?

vs.



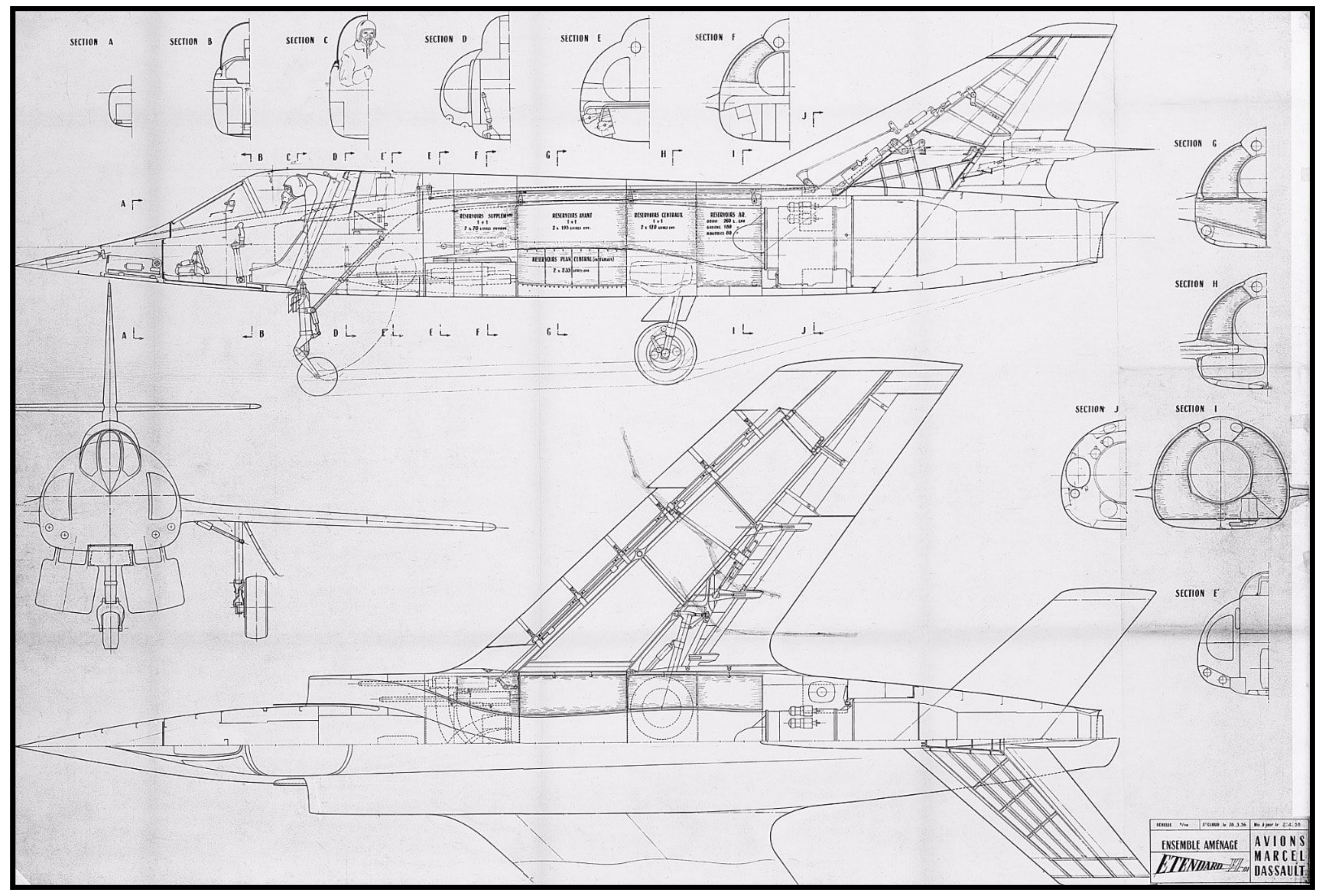
Users vs. Designers

Programmers *use* PLs. We're interested in **designing** PLs

Users are not necessarily the best designers...Who should design PLs?

Answer: Mathematicians

vs.



Users vs. Designers

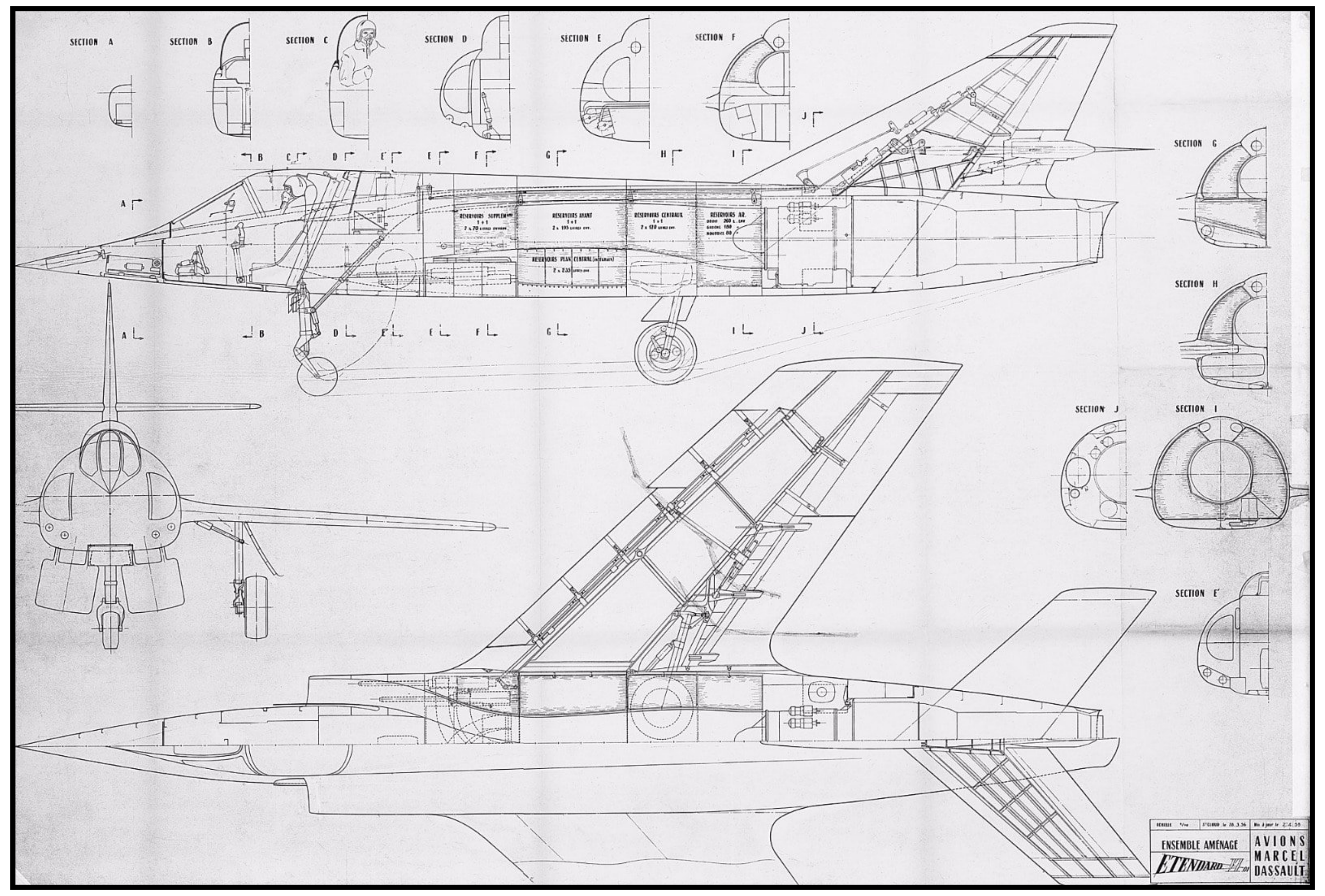
Programmers *use* PLs. We're interested in **designing** PLs

Users are not necessarily the best designers...Who should design PLs?

Answer: **Mathematicians**

(CS320 is a math class, sorry)

vs.



Mathematician's View of PL

```python
def subtraction():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} - {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1 - num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1-num_2}!\n")
        start_game()

def multiplication():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} x {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1*num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1*num_2}!\n")
        start_game()
```

vs.

Syntax

t	::=		*terms:*
		x	*variable*
		$\lambda\mathsf{x}{:}\mathsf{T}.\mathsf{t}$	*abstraction*
		$\mathsf{t}\ \mathsf{t}$	*application*
v	::=		*values:*
		$\lambda\mathsf{x}{:}\mathsf{T}.\mathsf{t}$	*abstraction value*
T	::=		*types:*
		$\mathsf{T}\to\mathsf{T}$	*type of functions*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, \mathsf{x}{:}\mathsf{T}$	*term variable binding*

Evaluation $\boxed{\mathsf{t} \longrightarrow \mathsf{t}'}$

$$\frac{\mathsf{t}_1 \longrightarrow \mathsf{t}_1'}{\mathsf{t}_1\ \mathsf{t}_2 \longrightarrow \mathsf{t}_1'\ \mathsf{t}_2} \quad \text{(E-APP1)}$$

$$\frac{\mathsf{t}_2 \longrightarrow \mathsf{t}_2'}{\mathsf{v}_1\ \mathsf{t}_2 \longrightarrow \mathsf{v}_1\ \mathsf{t}_2'} \quad \text{(E-APP2)}$$

$$(\lambda\mathsf{x}{:}\mathsf{T}_{11}.\mathsf{t}_{12})\ \mathsf{v}_2 \longrightarrow [\mathsf{x} \mapsto \mathsf{v}_2]\mathsf{t}_{12} \quad \text{(E-APPABS)}$$

Typing $\boxed{\Gamma \vdash \mathsf{t} : \mathsf{T}}$

$$\frac{\mathsf{x}{:}\mathsf{T} \in \Gamma}{\Gamma \vdash \mathsf{x} : \mathsf{T}} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathsf{x}{:}\mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \lambda\mathsf{x}{:}\mathsf{T}_1.\mathsf{t}_2 : \mathsf{T}_1\to\mathsf{T}_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_{11}\to\mathsf{T}_{12} \qquad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}_{11}}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{T}_{12}} \quad \text{(T-APP)}$$

(from T&PL by Pierce)

Mathematician's View of PL

» a mathematical object, like a polynomial or a vector

```python
def subtraction():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} - {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1 - num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1-num_2}!\n")
        start_game()

def multiplication():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} x {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1*num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1*num_2}!\n")
        start_game()
```

vs.

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}:\mathtt{T}.\mathtt{t}$	*abstraction*
		$\mathtt{t}\ \mathtt{t}$	*application*
$\mathtt{v}$	::=		*values:*
		$\lambda\mathtt{x}:\mathtt{T}.\mathtt{t}$	*abstraction value*
$\mathtt{T}$	::=		*types:*
		$\mathtt{T}\to\mathtt{T}$	*type of functions*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, \mathtt{x}:\mathtt{T}$	*term variable binding*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \quad \text{(E-APP1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \quad \text{(E-APP2)}$$

$$(\lambda\mathtt{x}:\mathtt{T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \quad \text{(E-APPABS)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}:\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}:\mathtt{T}_1.\mathtt{t}_2 : \mathtt{T}_1\to\mathtt{T}_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \quad \text{(T-APP)}$$

(from T&PL by Pierce)

Mathematician's View of PL

- a mathematical object, like a polynomial or a vector
- a formal specification

```python
def subtraction():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} - {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1 - num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1-num_2}!\n")
        start_game()

def multiplication():
    num_1 = randint(0,9)
    num_2 = randint(0,9)

    print(f"What is {num_1} x {num_2} ?\n")

    choice = input("> ")

    if int(choice) == num_1*num_2:
        print("Correct! Nice job! Keep on playing!\n")
        start_game()
    else:
        print(f"Incorrect...the answer is {num_1*num_2}!\n")
        start_game()
```

vs.

Syntax

t	$::=$		*terms:*
		x	*variable*
		$\lambda x{:}T.t$	*abstraction*
		$t\ t$	*application*
v	$::=$		*values:*
		$\lambda x{:}T.t$	*abstraction value*
T	$::=$		*types:*
		$T \to T$	*type of functions*
Γ	$::=$		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, x{:}T$	*term variable binding*

Evaluation $\boxed{t \longrightarrow t'}$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad \text{(E-APP1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad \text{(E-APP2)}$$

$$(\lambda x{:}T_{11}.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \quad \text{(E-APPABS)}$$

Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{x{:}T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \quad \text{(T-ABS)}$$

$$\frac{\Gamma \vdash t_1 : T_{11} \to T_{12} \qquad \Gamma \vdash t_2 : T_{11}}{\Gamma \vdash t_1\ t_2 : T_{12}} \quad \text{(T-APP)}$$

(from T&PL by Pierce)

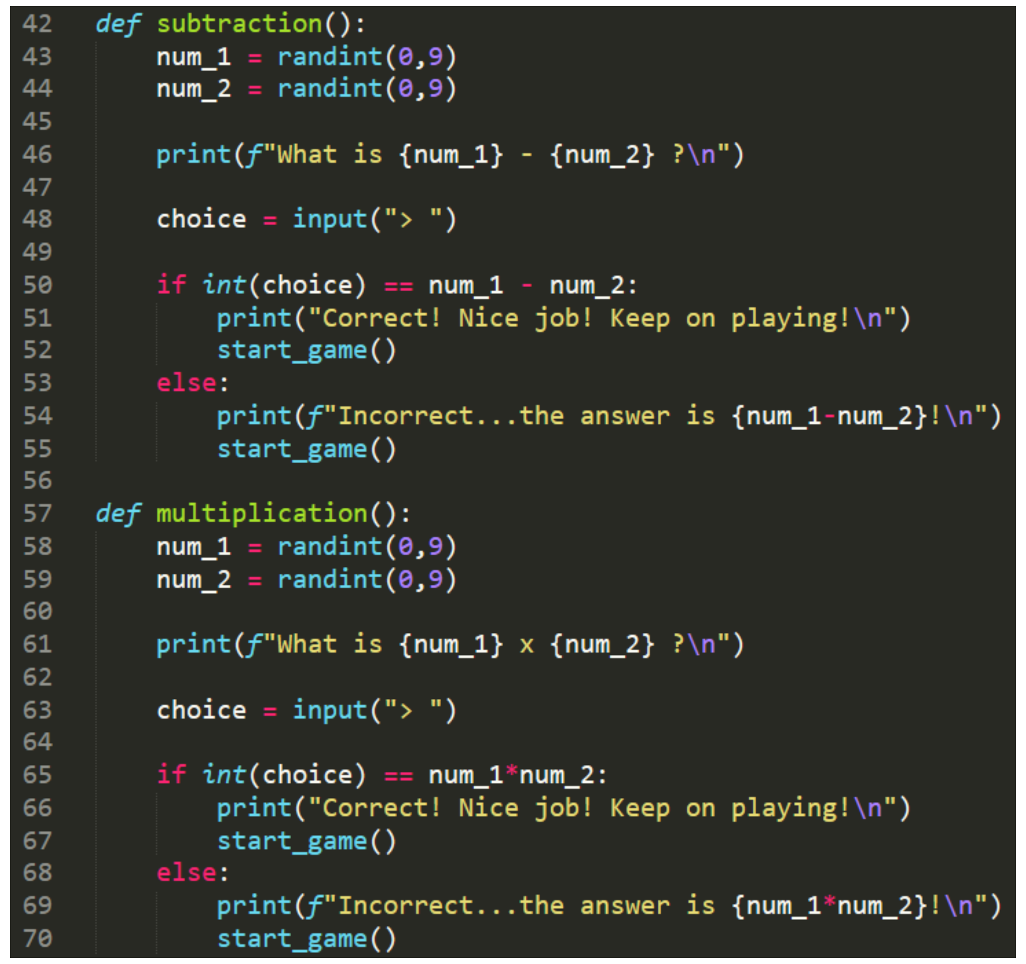
Mathematician's View of PL

» a mathematical object, like a polynomial or a vector

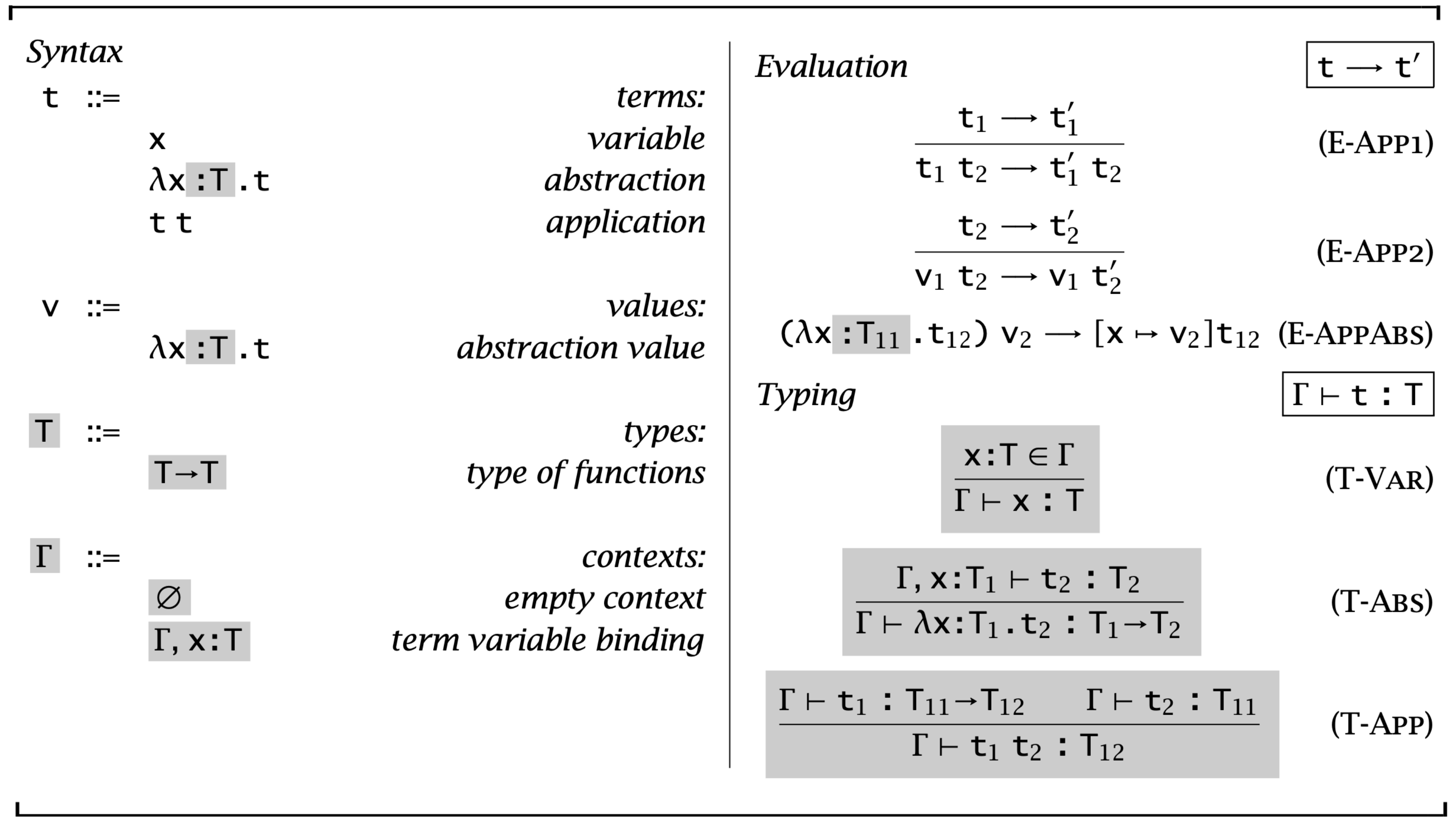
» a formal specification

» composed of exactly three things:

- Syntax
- Type System
- Semantics



vs.



(from T&PL by Pierce)

Why does this matter?

Why does this matter?

There are a lot of *bad* PLs out there

Why does this matter?

There are a lot of *bad* PLs out there

We want *good* PLs

Why does this matter?

There are a lot of *bad* PLs out there

We want *good* PLs

This course is about **what makes a PL good**

Why does this matter?

There are a lot of *bad* PLs out there

We want *good* PLs

This course is about **what makes a PL good**

The punchline: mathematically well-defined syntax, type system, and semantics

Formal PL

The Three Components

The Three Components

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

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

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

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"
```

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 everything we do from now on, we'll define the syntax rules, the typing rules, and the semantic rules

Syntax Rules

Syntax Rules

Syntax rules describe well-formed expressions.
They are independent of meaning

Syntax Rules

<u>Syntax rules</u> describe well-formed expressions.
They are independent of meaning

A syntax rule of the form:

Syntax Rules

<u>Syntax rules</u> describe well-formed expressions.
They are independent of meaning

A syntax rule of the form:

*If **<such-and-such>** is a well-formed expression and **<some-other-things>** are a well-formed expression, then **<some-combination-of-such-and-such-and-some-other-things>** is a well-formed expression*

Example: Integer Addition Syntax

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*

2 + (4-3)

Formal notation:

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

Typing Rules

Typing Rules

Typing rules describe the types of expressions

Typing Rules

Typing rules describe the types of expressions

They are of the form:

Typing Rules

Typing rules describe the types of expressions

They are of the form:

*If **<such-and-such>** is of **<such-and-such-type>** and **<some-other-things>** are of **<some-other-types>**, then **<some-combination-of-such-and-such-and-some-other-things>** is of **<some-different-type>***

Example: Integer Addition Typing

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)

$$\frac{\emptyset \vdash 2 : \text{int} \qquad \emptyset \vdash (4-3) : \text{int}}{\emptyset \vdash 2 + (4-3) : \text{int}}$$

Formal notation:

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

Semantic Rules

Semantic Rules

<u>Semantic rules</u> describe the *value* of an expression

Semantic Rules

Semantic rules describe the *value* of an expression

They are of the form:

Semantic Rules

<u>Semantic rules</u> describe the *value* of an expression

They are of the form:

If **<such-and-such>** evaluates to **<such-and-such-value>** and **<some-other-things>** evaluate to **<some-other-values>** then **<some-combination-of-such-and-such-and-some-other-things>** evaluates to **<some-other-value-computed-based-on-such-and-such-value-and-some-other-values>**

Example: Integer Addition Semantics

If e_1 *evaluates to* v_1 *and* e_2 *evaluates to* v_2*, then* $e_1 + e_2$ *evaluates to* $v_1 + v_2$

Formal Notation:

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 + v_2} \text{ (evalInt)}$$

We'll come back to all this soon...

OCaml: A First Look

What is OCaml?

What is OCaml?

- It is a statically-typed "industrial-strength functional programming language" with powerful type-inference

What is OCaml?

- It is a statically-typed "industrial-strength functional programming language" with powerful type-inference
- It was developed at Inria (smart researchers in France) in the 90s

What is OCaml?

- It is a statically-typed "industrial-strength functional programming language" with powerful type-inference
- It was developed at Inria (smart researchers in France) in the 90s
- It won the ACM SIGPLAN Programming Languages Software Award in 2023

What is OCaml?

- It is a statically-typed "industrial-strength functional programming language" with powerful type-inference
- It was developed at Inria (smart researchers in France) in the 90s
- It won the ACM SIGPLAN Programming Languages Software Award in 2023
- It's used/developed heavily by Jane Street (and too a lesser degree by facebook, Microsoft, docker, Wolfram)

What is OCaml?

What is OCaml?

» **Minimal:** The language is simple, there's very little to it

What is OCaml?

OCaml

- **Minimal:** The language is simple, there's very little to it

- **Functional:** A completely different paradigm. We're not writing procedures via commands/statements, we're defining values via expressions

Functional vs. Imperative

Functional here means:

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

State

```python
def fact(n):
    acc = 1
    for i in range(1, n + 1):
        acc *= i # acc is "stateful"
    return acc
```

In Python, we can define variables that change throughout the evaluation of the program

We can't do this in OCaml. Instead we use **recursion**(!)

If you can write recursive functions in Python, then you can program in OCaml

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

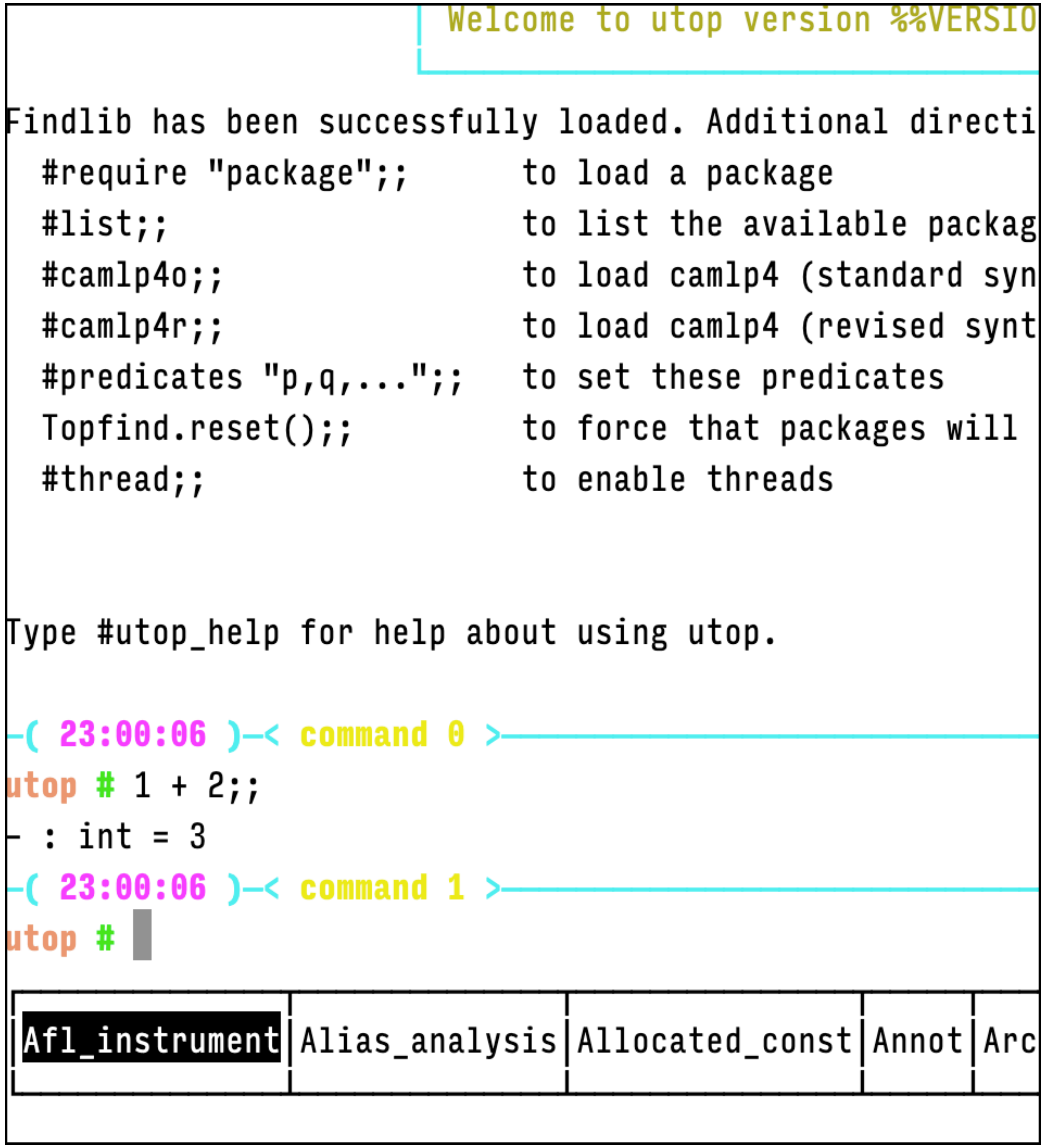
It allows us to specify project-level dependencies and configurations

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

Cheatsheet:

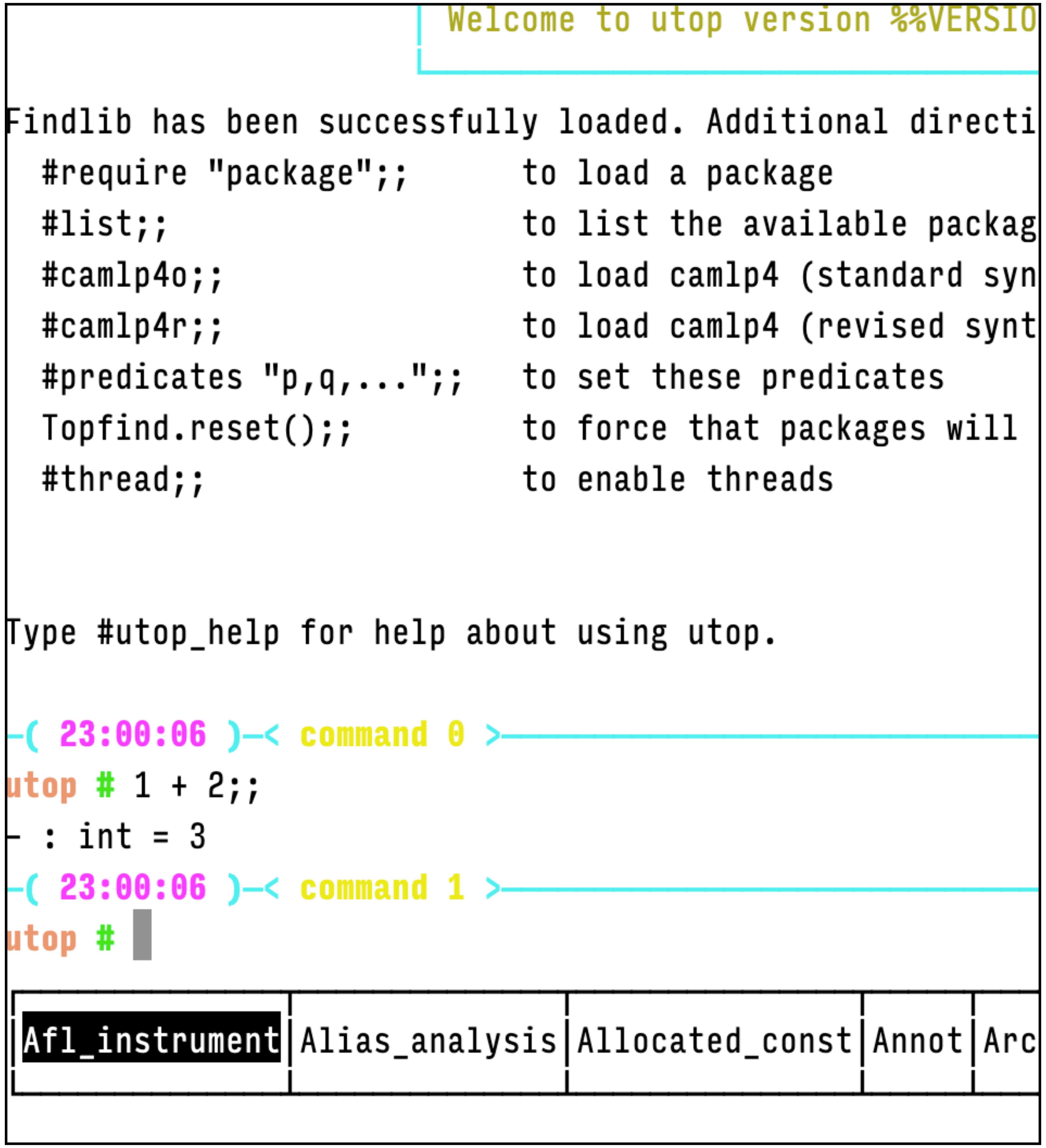
- **dune build:** type check your project
- **dune utop:** open Utop in a project aware way
- **dune test:** run a testing code associated with the project
- **dune exec PROJ_NAME:** run the executable of your project
- **dune clean:** removes files created by dune build (not so important but may come 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

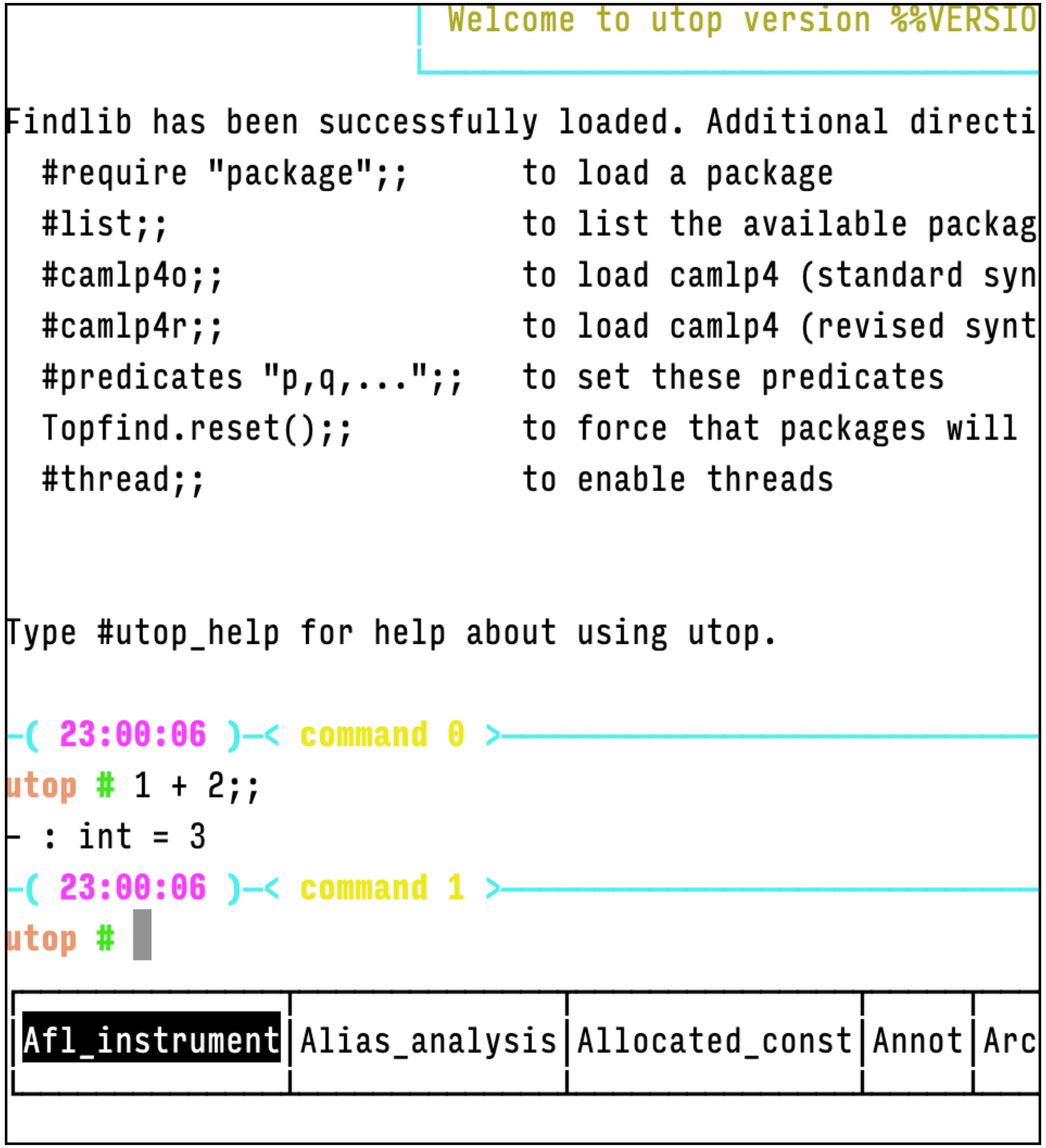
UTop is an interface for the OCaml toplevel



UTop

UTop is an interface for the OCaml toplevel

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

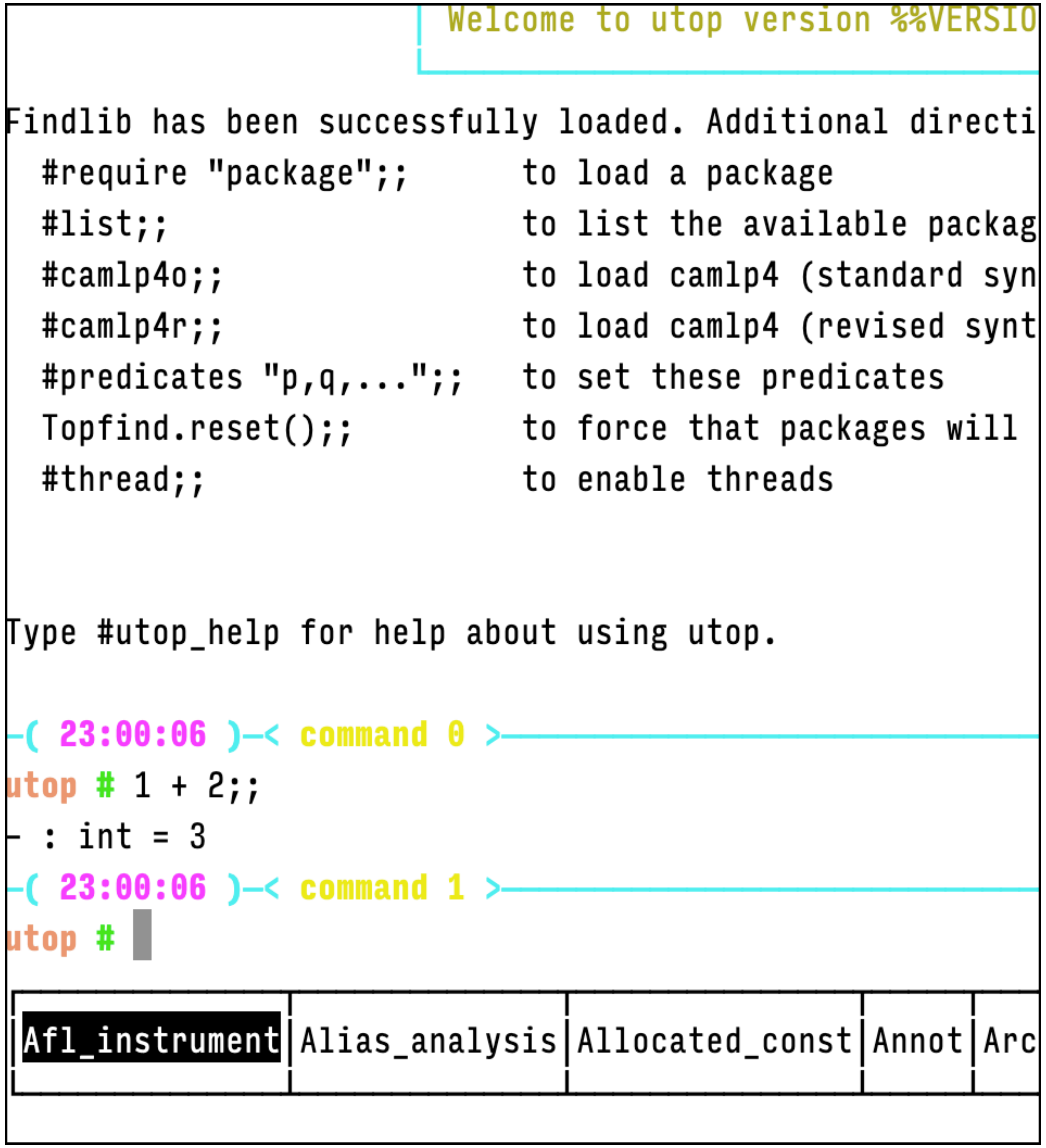


UTop

UTop is an interface for the OCaml toplevel

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

<u>Cheatsheet:</u>



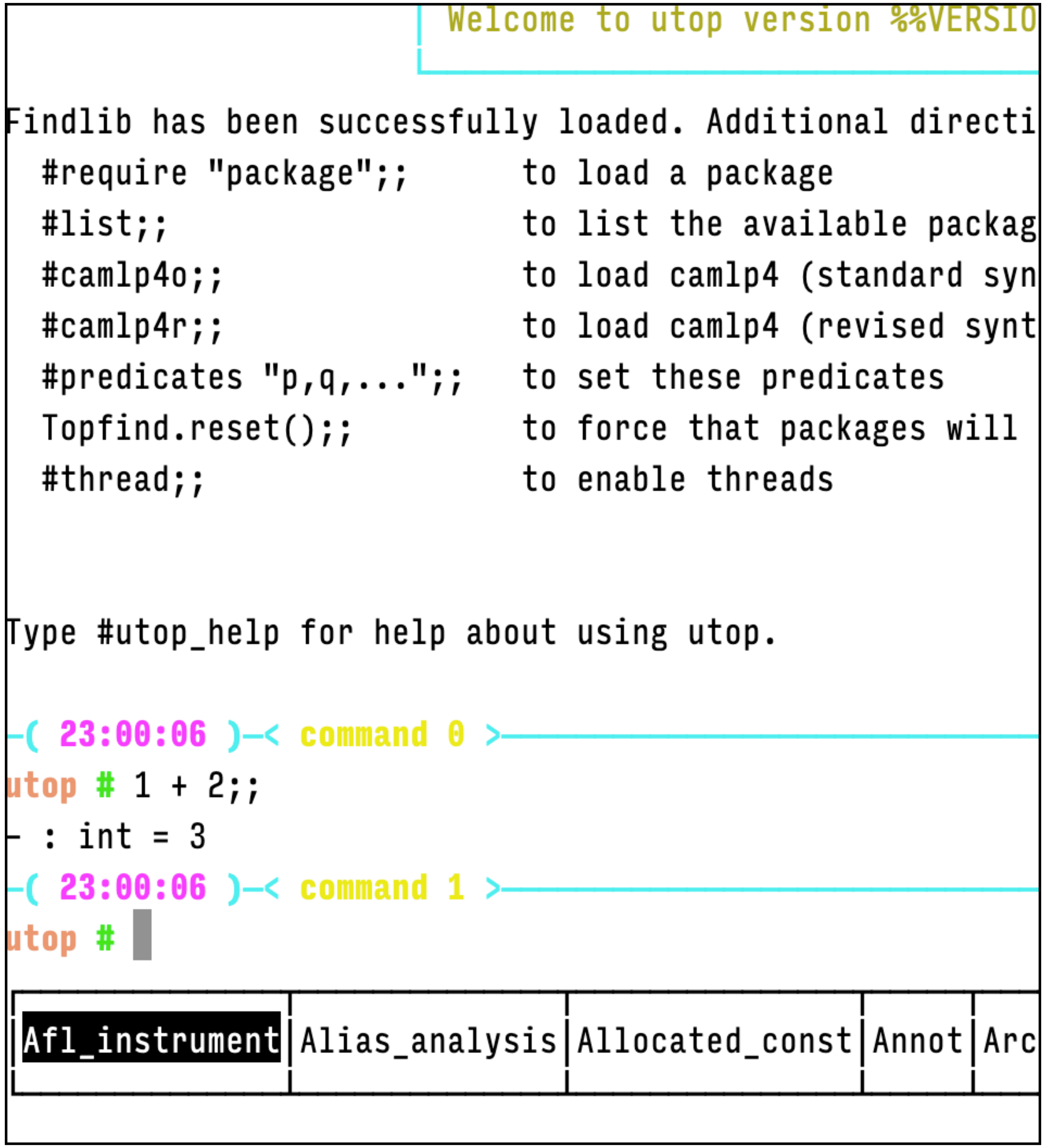
UTop

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



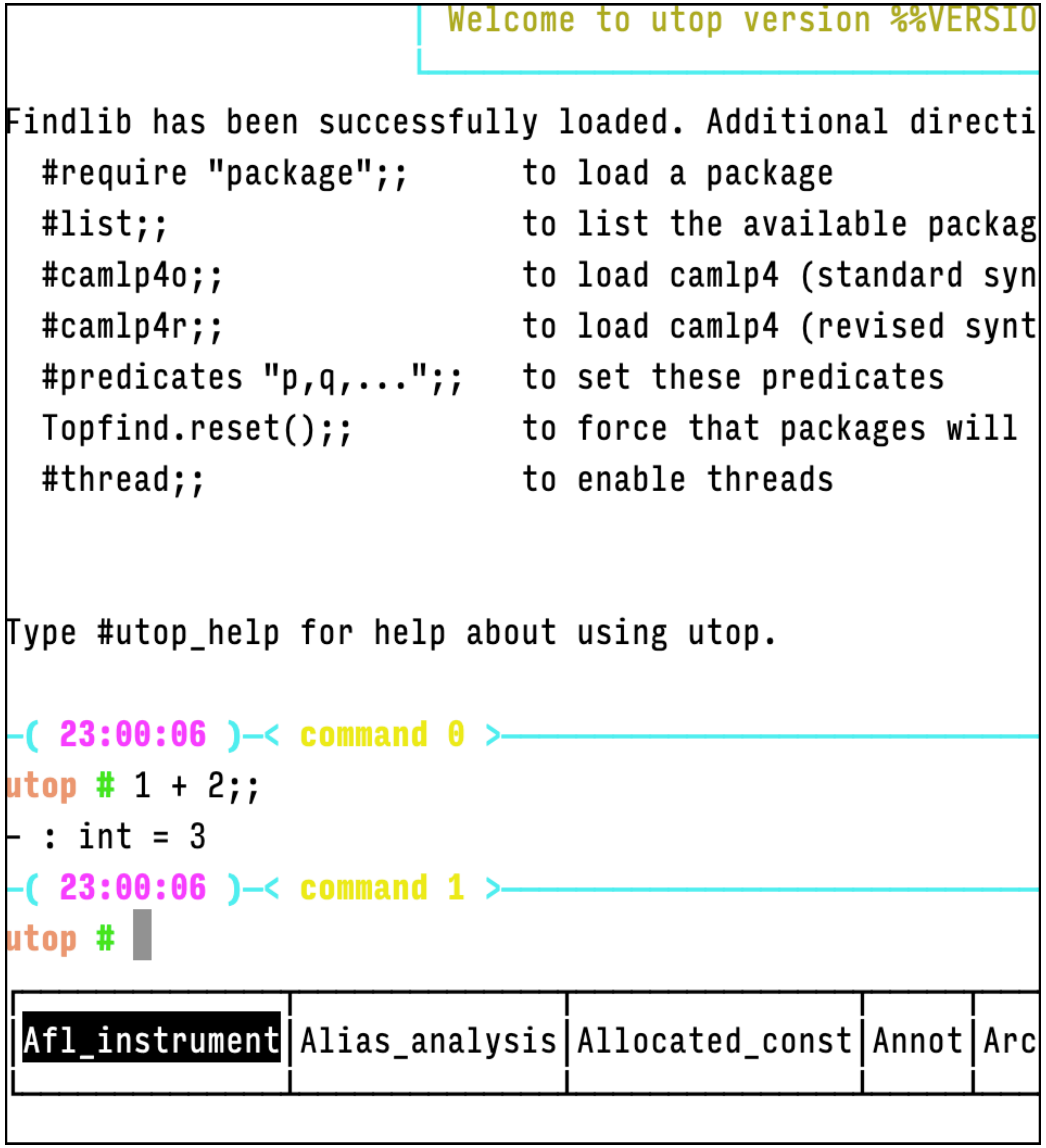
UTop

UTop is an interface for the OCaml toplevel

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

Cheatsheet:

- 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

```
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...

The Basics

Anatomy of an OCaml Program

name, not global var.

expression

```
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*

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
```

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

Values

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

<u>Expressions</u> *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

```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>

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

```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 expressions can be constructed

Basic Expressions

- Literals
- Let-expressions (local variables)
- If-expressions
- Functions
- Applications

Basic Expressions

- » **Literals**
- » Let-expressions (local variables)
- » If-expressions
- » Functions
- » Applications

Primitive Types and Literals

OCaml has a collection of standard literals and types

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

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 `!=`)

Basic Expressions

» Literals

» **Let-expressions (local variables)**

» If-expressions

» Functions

» Applications

Local Variables

body

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

Local Variables

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

We can define local variables in OCaml

Local Variables

body

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

We can define local variables in OCaml

This is useful for writing better abstractions

Local Variables

body

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

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

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

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 ;)

Multiple Local Variables

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

Python

body

```ocaml
let sum_of_squares x y =
  let x_squared = x * x in
  let y_squared = y * y in
  x_squared + y_squared
```

OCaml

body

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: `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
```

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 type it into utop

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

A Note on Substitution

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

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

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 <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*

Basic Expressions

- Literals
- Let-expressions (local variables)
- **If-expressions**
- Functions
- Applications

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!

OCaml has expressions for conditional reasoning

If-Expressions

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

Note: OCaml is whitespace agnostic!

OCaml has expressions for conditional reasoning

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

If-Expressions

```
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 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)

```
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

Basic Expressions

- Literals
- Let-expressions (local variables)
- If-expressions
- **Functions**
- Applications

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 *single argument*

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

Basic Expressions

» Literals

» Let-expressions (local variables)

» If-expressions

» Functions

» **Applications**

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: 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 (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
```

One Last Point: Building Interpreters

One Last Point: Building Interpreters

PL is math but we still like to *use* PLs. The three components of a PL correspond to the three things we need to *implement* in an **interpreter** of a PL.

One Last Point: Building Interpreters

PL is math but we still like to *use* PLs. The three components of a PL correspond to the three things we need to *implement* in an **interpreter** of a PL.

» **Syntax** is implemented by a **parser**

```
parse : string -> expr
```

One Last Point: Building Interpreters

PL is math but we still like to *use* PLs. The three components of a PL correspond to the three things we need to *implement* in an **interpreter** of a PL.

» **Syntax** is implemented by a **parser**

```
parse : string -> expr
```

» **Type system** is implemented by a **type checker**

```
type_check : expr -> bool (* valid or not *)
```

One Last Point: Building Interpreters

PL is math but we still like to *use* PLs. The three components of a PL correspond to the three things we need to *implement* in an **interpreter** of a PL.

» **Syntax** is implemented by a **parser**

```
parse : string -> expr
```

» **Type system** is implemented by a **type checker**

```
type_check : expr -> bool (* valid or not *)
```

» **(Dynamic) semantics** is implemented by an **evaluator**

```
eval : expr -> value
```

Next Steps

- » Make sure you're on Piazza and Gradescope, keep an eye on announcements
- » Bookmark the course webpage and course repo
- » Install opam, VSCode, the course standard library, etc. *(see assignment 1)*
- » **Do the reading listed on the course webpage**

Summary

A PL is a mathematical object given by its **syntax**, **type system** and **semantics**

There is **no state** in functional programming. Programs define the output for a given input

Practice, practice, practice. Functional programming takes time to learn, but once you get it, it's as easy as programming in any other PL