A new language: OCaml

• There’s almost a dictionary to translate from Racket into Ocaml
• Less structure in page-layout of programs
• No “DrOCaml” 😞
• Still has an ‘interactive’ mode where you can type things and see them evaluated
• Has a debugger that allows time to run backward.
Key differences

• Parenthesis are used only for grouping.
• whitespace is not significant (as long as Ocaml can parse things)
• Ocaml has a notion of “types” and is very rigid about them.
  • This will annoy you at first, but soon you’ll love it
  • Make debugging easier (after the first few days)
• Ocaml has “cond on steroids”, called “pattern matching”
  • Lets you extract all the parts (like “first” and “rest”) for you and names them all at once.
• Ocaml has two kinds of structs/records
• Ocaml has variant types (like “a season is Fall or Winter or …”) built in
• Ocaml has recursive type (like “lists”) both built in and easy to create
• “lambda” is called “fun”
• All functions take one argument, sort of.
• Expressions end with semicolons (which also get used elsewhere!)
• In interactive session, end expression with double semicolons.
Basic types

- int, float, string, bool
- int constants have no decimal point; float constants have a decimal

# 4;;
- 4: int
# 4. ;;
- 4.0: float
# 7+2*5;;
- : int = 17
Do types matter?

• Yes! You must indicate types when you’re coding…
• …and Ocaml, when it runs, checks that they all make sense or it halts
• You cannot make a list like `(3 4 #true)` in Ocaml, because its type-system allows only monotype lists.
# 3 + 5;;
- : int = 8
# 2.0 +. 3.5;;
- : float = 5.5

+, -, *, / (for ints)
+. -, .*, ./ (for floats)

# 2.0 ** 4.0;;
- : float = 16.
# float_of_int(16);;
- : float = 16.
  ceil, floor, ???...but int_of_float might help.

Think of a type as a set of values.
bool is the set \{true, false\}
int is the set of integer values (actually not all in Ocaml)
float is the set of real numbers (actually not all, of course)
# 4 * 4 ;;
-: int = 16

• The “-” means “the value you just computed doesn’t have a name (or identifier) associated to it.
• The “:” always comes between a thing and its type, so the value we computed is an “int”
• The actual value of that int is 16
# 4 > 2;;
- : bool = true

# 4.1 > 2.6
- : bool = true

# 4 == 5
- : bool = false
Cartesian product

\[ A = \{1, 2, 3\} \quad B = \{♥, ♠, ♣, ♦\} \]

\[ \{(1,♥), (2,♥), (3,♥), (1,♠), (2,♠), (3,♠), (1,♣), (2,♣), (3,♣), (1,♦), (2,♦), (3,♦)\} \]

Note \[ |A \times B| = |A| \cdot |B| \]

But also makes sense for infinite sets

\( \text{int} \times \text{float} \) set of pairs \((i, x)\) where \(i\) is integer, \(x\) is real number

Example element: \((17, 3.14159)\)

Can have three-way (or more) product, e.g. \( \text{int} \times \text{int} \times \text{bool} \)

Example element: \((17, 18, \text{true})\) Pairs, triples, etc. called \textit{tuples}

In Ocaml, \(\times\) is written with *
# (10 + 7, 3.5 * 2.5);;  
- : int * float = (17, 8.75)

Parens only needed for grouping

# 10 + 7, 3.5 * 2.5;;  
- : int * float = (17, 8.75)

**Quiz:** Guess the type

# 2 * 4, 3.5 -. 2.5, false;;  
- : int * float * bool = (8, 1., false)

Compound types can be built from basic or compound types

**Quiz:** Guess the type

# ((1,2), true);;  
- : (int * int) * bool = ((1, 2), true)
Procedures and procedure types

Let $A$ and $B$ be sets.

Mathese notation for the set of all mathematical functions such that
• input is in $A$, and
• output is in $B$
is

$$A \rightarrow B$$

Ocaml procedures ("functions") use same notation for types.

Ocaml type for a procedure such that
• input is in $A$, and
• output is in $B$
is

$$A \rightarrow B$$

Written $A \rightarrow B$
Procedures and procedure types

**Quiz:** Guess the types

```ocaml
# abs;;
- : int -> int = <fun>

# sqrt;;
- : float -> float = <fun>

# abs;;
- : int -> int = <fun>

# abs_float;;
- : float -> float = <fun>

# ceil;;
- : float -> float = <fun>
```

Remember Scheme’s printed representation of a procedure was uninformative.

```ocaml
> (lambda (x) 1)
#<procedure>

> +
#<procedure:+>
```

Ocaml’s printed representation of any procedure is just `<fun>`.

Surprised at output type of `ceil`? How to get an `int` from a `float`?

```ocaml
# float_of_int;;
- : int -> float = <fun>
```
Procedures and procedure types

Note: procedure types are compound types.
A procedure type can be made from other basic/compound types.

Quiz: Guess the types of these procedures:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a procedure that takes an int and produces a float</td>
<td>a bool</td>
</tr>
</tbody>
</table>

Break it down.
Type of the input is int -> float
The type of the output is bool
Therefore the type of the procedure is (int -> float) -> bool

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>an int</td>
<td>a procedure that takes an int and produces an int</td>
</tr>
</tbody>
</table>

Type of the output is int -> int
Type of the input is int
Therefore the type of the procedure is int -> (int -> int)

Ocaml doesn’t print parens in this type.
Ocaml procedures that take multiple arguments

There aren’t any.

Why is that not a problem?

Suppose you think you want a procedure that takes two arguments, an int and a bool, and produces a float as output.

Alternatives
- procedure that takes as input a tuple of type int * bool and outputs a float
- procedure that
  - takes an int as argument, and
  - produces as output a procedure that takes as input a bool and produces a float as output.

Replacing first type with second type is called currying a function/procedure.
Replacing second type with first is decurrying.

Quiz: Guess the types of these procedures:
- The type of the first alternative is (int * bool) -> float
- The type of the second alternative is int -> (bool -> float)
Applying a procedure

To construct an expression in which a procedure is applied to an argument:
• an expression for procedure
• an expression for argument
• next to each other

```scheme
# sqrt 2.0;;
- : float = 1.41421356237309515
```

Parens don’t matter here.

- Scheme style: \((\text{sqrt} \ 2.0)\)
- Java style: \(\text{sqrt}(2.0)\)
- Wacky style: \(\text{sqrt}(((2.0)))\)

\[ f \ a \ b \]
is interpreted as meaning
\[ (f \ a) \ b \]
If you enter an expression whose evaluation would result in the application of a procedure to an argument that is of the wrong type, Ocaml will not even try to evaluate the expression. It will reject it immediately.

What happens if you enter the following?

```
sqrt (1/0);;
```

No division by zero will ever happen because Ocaml will complain about types first.

“You tried to get me to apply a procedure that takes floats but you tried to give it an int.”

Called type-checking. Think of it as Ocaml trying out your code without actually doing any evaluation.
Conditional expressions

Quiz: Guess the types

\[
\text{if } 3 < 5 \text{ then } 11 \text{ else } 2 + 8
\]

\[
\text{if } 3 < 5 \text{ then } "\text{eleven}" \text{ else } 4
\]

fail: expression has a type, just as do its parts. Is this str or int?

Ocaml if-expressions are expressions with values, just as if-expressions are in Scheme. Don’t be fooled by their looking like Java if statements.

Ocaml has another feature that often substitutes for conditionals, pattern-matching.
Strings

Ocaml does not have symbols but it does have type `string`. A string does not act as a variable but it is easy to put strings together:

```ocaml
# "hello" ^ "world";;
- : string = "helloworld"
```

Called *concatenation* of strings.
“lambda” expressions

Instead of lambda, called **fun** or **function**
Difference between **fun** and **function**? Later.

**Quiz:** Guess the type

```ocaml
# fun x -> x+1;;
- : int -> int = <fun>
```

**Syntax:**
- keyword **fun**
- formal argument
- arrow
- body of procedure

How does Ocaml know the type?
Called **type inference**.

You can also tell Ocaml what you think the type is:

```ocaml
# fun x:int -> x+1;;
- : int -> int = <fun>
```

Called **type annotation**.
Often recommended because type inference is mysterious.
More fun

Quiz: Guess the types

fun \( (x,y,z) \) \( \rightarrow \) if \( x \) then \( y+z \) else \( y-z \)

\( \text{bool} \times \text{int} \times \text{int} \) \( \rightarrow \) \( \text{int} \)

fun \( x \) \( \rightarrow \) (fun \( y \) \( \rightarrow \) \( x + \text{int}_{\text{of}} \text{float} y \))

\( \text{int} \) \( \rightarrow \) \( \text{float} \) \( \rightarrow \) \( \text{int} \)
Binding variables to values

The closest thing to Scheme's define in Ocaml is let

```ocaml
# let x = 3+4;;
val x : int = 7
```

The let stuff is a declaration—not an expression. It has no value.

```ocaml
# let add1 = fun n -> n+1;;
val add1 : int -> int = <fun>
```

An important difference between Scheme’s define and Ocaml’s let

Discuss another time.

But one consequence is that you cannot define recursive procedures using let
Must use `let rec`:

```ocaml
# let fac = fun n -> if n = 0 then 1 else n * fac (n-1);;
Error: Unbound value fac
# let rec fac = fun n -> if n = 0 then 1 else n * fac (n-1);
val fac : int -> int = <fun>
```

An important difference between Scheme’s `define` and Ocaml’s `let`.
Discuss another time.
But one consequence is that you cannot define recursive procedures using `let`.
Ocaml provides a nice way to form a list out of a collection of expressions. You use square brackets and separate the expressions using a single semicolon:

```
# [1+2; 3+4; 5+6];;
- : int list = [3; 7; 11]
```

Type is not `list`
Type is `int list`

Every element of an Ocaml list has the same type.

```
# [false; not false];;
- : bool list = [false; true]
```
Lists

Cons is replaced by infix operator consisting of two colons

::

# 3::[2; 1];;
- : int list = [3; 2; 1]

Note that :: is not a procedure. It is called a *constructor*, which matters later.

# 1::[];;
- : int list = [1]
# 2::1::[];;
- : int list = [2; 1]
Let’s write a procedure that, given a nonnegative integer \( n \), produces a list consisting of \( n, n-1, \ldots, 1 \)

```ocaml
# let rec countdown = fun n -> if n = 0 then [] else n::countdown (n-1);;
val countdown : int -> int list = <fun>
# countdown 10;;
- : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1]
```

What is type of the empty list?

```ocaml
# [];;
- : 'a list = []
```

The quote mark followed by the letter \( a \) is a type parameter. This basically says that a list can have any type as its element type.
Boolean operators and expressions

&&, ||, not, true, false

&& is like Scheme’s and

(4 > 2) && (3 == 3)

|| is like Scheme’s or

4 > 2 && 3 == 3

parens not needed because == and < “bind tighter” than &&

(15 < 16) && (17.0 < 18.0)

OK because both subexp's evaluate to bools

15 < 16 && 17.0 < 18.0
type shape = Circle of float | Rectangle of float *
float

• The capital letters are essential: all named
constructors begin with those! Type-names, by
contrast, must start lower-case.

• Simplest form:

type season =
| Fall the “|” is read “or”
| Winter extra “|” at front allowed for neatness
| Spring
| Summer
# type card =
| Clubs of int
| Spades of int
| Diamonds of int
| Hearts of int
;

```
type card = Clubs of int | Spades of int | Diamonds of int | Hearts of int

# let rank (c:card) : int =
  match c with
  | Clubs n -> n
  | Diamonds n -> n
  | Hearts n -> n
  | Spades n -> n;
val rank : card -> int = <fun>
```

```
# Hearts 4;;  Note that “of” appears in type definition, but NOT in use of constructor
-: card = Hearts 4
-# rank (Hearts 4);;
-: int = 4
```
Pattern-matching

match num with
| 1 -> "Too small."
| 2 -> "Right on!"
| 3 -> "Too big."

let guess = fun num ->
    match num with
    | 1 -> "Too small."
    | 2 -> "Right on!"
    | 3 -> "Too big."
    | n -> "The number " ^ string_of_int(n) ^ " is not between 1 and 3"
match Fall with
|    Fall  ->  55
|   Winter  ->  25
|   Spring  ->  65
|  Summer  ->  90