1 Variant Types

In Racket, we often wrote procedures that consumed mixed data. In OCaml, we can do that too, but we need to make it perfectly clear to the compiler (specifically, the type checker) exactly which types of data are consumed and produced by every procedure. To facilitate this process, there is a mechanism called **variant types** (or just **variants**).
Variant types are also sometimes called or types, or disjunctive types, since values of some variant type can be constructed in one way or another.

**Example 1** Here is our first example of a variant type definition:

```ocaml
type season =
  | Fall
  | Winter
  | Spring
  | Summer
```

**Question:** What is the vertical line in the definition of `season`?

**Answer:** It is a delimiter that separates values. It is called a *pipe*, and pronounced “or.” The first pipe is included for neatness.

The word `season` is the name of this variant type. The four words that follow are constructors. This is an apt name for them, since they are used to construct values of the `season` type.

The name of a type (like any other OCaml identifier) must begin with a lowercase letter. In contrast, every constructor name must begin with a *capital letter*. (That’s how OCaml distinguishes constructors from identifiers.)

Once this variant type is defined, the constructors can be used to construct values of type `season`. For example, the value `Fall` has type `season`.

**Question:** How can we tell whether a value of type `season` is Fall, Winter, Spring, or Summer?

**Answer:** Using pattern-matching, which you will be learning about very soon.

**Example 2** You can define more interesting variants using constructors that take an argument. To do so, you use the keyword `of`. For example,

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

You can then supply an argument of the specified type to one of the constructors, producing a value of the variant type. For example, `Hearts 13` is of type `card`.

Note that the keyword `of` appears only in the type definition, not in values.

Now, let’s talk about writing a procedure that determines the rank of a card. By the way, the rank of a card is the number on the face of the card. Here’s how we would do it:

```ocaml
let rank (c:card) : int =
  match c with
  | Clubs n -> n
  | Diamonds n -> n
  | Hearts n -> n
  | Spades n -> n;;
```
The code is sort of surprising. \( c \) is the card, and we say if \( c \) was constructed with the \texttt{Clubs} constructor and argument \( n \), the answer is \( n \). If \( c \) was constructed with the \texttt{Diamonds} constructor and argument \( n \), the answer is \( n \), and so on.

So, let’s apply our procedure to some card, and see what happens!

\begin{verbatim}
rank (Hearts 4);;
- : int = 4
\end{verbatim}

However, we do not need to restrict our variant types to just one base type. If we find the need for a variant type that’s adaptable to different base types, we can create a \textbf{parameterized variant type}. For example, in defining a tree data type, we may want to be able to create a tree of ints, a tree of strings, or a tree of floats. How can we get around this, when OCaml requires that we specify a single type? Well as we saw in some of our Racket procedures, we can notate a general type, such as ‘a:

\begin{verbatim}
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
\end{verbatim}

## 2 Records

OCaml tuples allow us to combine a group of values of various types into a single entity. Analogous to Racket structures, OCaml \textbf{records} enable us to name these values.

Tuples and records are sometimes called \textbf{and types}, or \textbf{conjunctive types}, since values of these types compound multiple other types.

Here is an example of a record type definition, followed by an annotated instance of that type:

\begin{verbatim}
(* A point in 3 dimensions is characterized by x, y, and z coordinates *)
type point_3d = {x : int; y : int; z : int}

(* Examples of point_3d *)
((x = 1; y = 2; z = 3) : point_3d)
let origin3 = ((x = 0; y = 0; z = 0) : point_3d)
\end{verbatim}

Here, \texttt{point_3d} represents a point in three-dimensional space. We could have also represented such points as a tuple of type \texttt{int * int * int}, but using a record facilitates selection. That is, we can access the various fields by name. The syntax for selection is \texttt{a .} (a dot). For example:

\begin{verbatim}
origin.z
=> - : int = 0
\end{verbatim}

Although you can select field in a record by name, pattern matching is generally preferred to selection in OCaml. Pattern matching is the subject of our next lecture.

It is straightforward to combine record types with variant types. Here is another record type:
And here is a variant type of points in either 2d or 3d space:

```ocaml
(* A point is either a 2d or a 3d point *)
type point =
  | Point2d of point_2d
  | Point3d of point_3d

(* Examples of point *)
Point2d origin2
Point3d origin3
```

Typically, in CS 17, it is sufficient to use tuples, rather than records. This is because there are usually only two or three distinct fields in the data, and what each field represents is usually clear from context. However, when the size of a tuple is much more than 3, or if it would improve readability if fields were named, you should consider storing your data as records.

## 3 Pattern Matching

OCaml does not have a direct equivalent of `cond`. Instead, it has a keyword called `match`.

The basic shape of a `match` expression is:

```ocaml
match ⟨expr⟩ with
| ⟨pattern-1⟩ -> ⟨expr-1⟩
| ⟨pattern-2⟩ -> ⟨expr-2⟩
| ...
| ⟨pattern-n⟩ -> ⟨expr-n⟩
```

In certain special cases, this corresponds to the following Racket code:

```racket
(cond
[([equal? expr pattern-1] expr-1]
[([equal? expr pattern-2] expr-2]
...
[([equal? expr pattern-k] expr-k])
```

Like `cond`, a `match` expression consists of a series of clauses, each with a question (called a `pattern`) and a result.

## 4 Built-in Patterns

OCaml supports both built-in and user-defined patterns. We describe the former first, namely constant patterns, the wild card, identifier patterns, tuple patterns, and list patterns.
4.1 Constant Patterns

A constant pattern is the simplest of all atomic patterns. It matches only itself.

Here is a simple example of a match expression with constant (integer) patterns:

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

This expression evaluates exactly as you’d expect. The identifier `num` matches 1, 2, or 3, respectively, only when it is bound to one of these values, and it then produces the corresponding string.

Here is a sample evaluation:

```ocaml
let guess (num : int) : string =
  match num with
  | 1 -> "Too small."
  | 2 -> "Right on!"
  | 3 -> "Too big."

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

(* 2 doesn't match with the first case, so the next one gets checked... *)

=> match 2 with
  | 2 -> "Right on!"
  | 3 -> "Too big."

(* We've hit a match, so the program outputs the associated result! *)

=> "Right on!"
```

**Question:** What happens when you apply this match expression to the input 4?

**Answer:** 4 does not match any of the patterns, so OCaml generates an “Exception: Match_Failure”:

```ocaml
match 4 with
| 1 -> "Too small."
| 2 -> "Right on!"
| 3 -> "Too big."
=> Exception: Match_Failure
```

Similarly, here is a simple example of a match expression with string patterns:
match str with  
| "clubs" -> "You lose"  
| "hearts" -> "You win"  
| "spades" -> "Tie"  
| "diamonds" -> "Tie"

This expression also evaluates exactly as you’d expect:

```ocaml
let suit (str : string) : string =  
    match str with  
    | "clubs" -> "You lose"  
    | "hearts" -> "You win"  
    | "spades" -> "Tie"  
    | "diamonds" -> "Tie"

suit "hearts"  
=> match "hearts" with  
    | "clubs" -> "You lose"  
    | "hearts" -> "You win"  
    | "spades" -> "Tie"  
    | "diamonds" -> "Tie"

(* "hearts" doesn't match the first case, so the next one gets checked... *)  
=> match "hearts" with  
    | "hearts" -> "You win"  
    | "spades" -> "Tie"  
    | "diamonds" -> "Tie"

(* We've hit a match, so the program outputs the associated result! *)  
=> "You win"
```

In addition to integers and strings, other constant patterns include floating point numbers, booleans, and the empty list, `[ ]`.

### 4.2 The Wild Card

To achieve the effect of an `else` clause, OCaml has a special pattern, spelled `_` and pronounced **wild card**, which matches any expression. For example,

```
match 4 with  
| 1 -> "Too small."  
| 2 -> "Right on!"  
| 3 -> "Too big."  
| _ -> "Pick a number between 1 and 3"  
=> "Pick a number between 1 and 3"
```

Here, because 4 did not match with 1, 2, nor 3, it then matched with the wild card and the program output the associated sentence.
But beware! Just as \texttt{else} was a dangerous thing to use in Racket, \_ is just as dangerous when used in this way in OCaml.

If the only possible patterns were constants and the wild card, then pattern matching would not be very powerful. However, patterns can be much more interesting, as we’ll see next.

### 4.3 Identifier Patterns

Like constants, another kind of atomic pattern is an \texttt{identifier}. Like a wild card, an identifier matches \textit{any value}. Moreover, if an identifier pattern matches some value, then the identifier is \textit{bound} to the value it matches, and that value is substituted for the identifier everywhere it appears in the answer to that clause. Because of this, \textit{it’s a bad idea to use as a pattern-matching identifier any identifier that already has a meaning, such as one of the arguments to your procedure, or the procedure name}.

For example:

```ocaml
let suit (str : string) =
  match str with
  | "clubs" -> "You lose"
  | "hearts" -> "You win"
  | "spades" -> "Tie"
  | "diamonds" -> "Tie"
  | other -> "You said " ^ other ^ " but that is not a suit."
```

`suit "stars"
=> match "stars" with
  | "clubs" -> "You lose"
  | "hearts" -> "You win"
  | "spades" -> "Tie"
  | "diamonds" -> "Tie"
  | other -> "You said " ^ other ^ " but that is not a suit."
=> ...
=> match "stars" with
  | other -> "You said " ^ other ^ " but that is not a suit."
=> "You said stars but that is not a suit."

**Question:** Could you have written "You said " ^ str ^ "but that is not a suit." in the final clause of this match expression?

**Answer:** Yes. But it would have been poor style to do so. Once you match a value to an identifier, you should refer to that value using its matched identifier thereafter.

Here is a more complicated example that sheds some light on OCaml’s scoping rules as they pertain to pattern matching:
let test myInt myOtherInt =
  match myInt with
  | 1 -> 1
  | 2 -> 2
  | myOtherInt -> myOtherInt

Here’s our procedure being applied:

test 17 18
=> 17

When test is applied to the actual arguments 17 and 18, the formal arguments myInt and myOtherInt are bound to 17 and 18, respectively. But after the first two questions fail, myOtherInt is matched with the value of myInt, so myOtherInt is rebound to 17. The inner binding of myOtherInt to 17 trumps the outer binding of myOtherInt to 18.

4.4 Tuple, Cons, and List Patterns

But patterns can be more interesting still. There are tuple patterns, cons patterns, and list patterns. These patterns, intentionally, look a lot like the constructors for tuples and lists. However, these patterns are themselves composed of patterns, including constants, identifiers, and the wild card, as well as other tuple, cons, and list patterns.

A tuple pattern looks like a tuple, but is composed of patterns, like this:

( ⟨pat-1⟩ , ⟨pat-2⟩ , ... , ⟨pat-k⟩ )

Here are some examples of tuple patterns and corresponding matches:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>(true, 17)</td>
<td>(true, 17)</td>
</tr>
<tr>
<td>(foo, bar, &quot;hearts&quot;)</td>
<td>(&quot;ace&quot;, &quot;of&quot;, &quot;hearts&quot;), (13, &quot;broken&quot;, &quot;hearts&quot;)</td>
</tr>
<tr>
<td>(3.14, (e, u))</td>
<td>(3.14, (&quot;E&quot;, &quot;U&quot;)), (3.14, (2.71, &quot;aces&quot;))</td>
</tr>
</tbody>
</table>

The first pattern matches only the tuple (true, 17). The second pattern matches any three-element tuple whose third element is the string "hearts", and binds foo to the first element and bar to the second element. The third pattern matches a two-element tuple whose first element is 3.14, and whose second element is a two-element tuple, whose first element is bound to e, and whose second element is bound to u.

**Question:** Can we write something like (3.14, (e, e)) to match things like (3.14, ("Hi", "There "))?  

**Answer:** Unfortunately using the same identifier twice in a pattern is not allowed. You may only use it once in a single pattern. However, as you will see later, using the same identifier over different patterns in the same match-statement is totally allowed!

Similarly, a cons pattern looks like a cons expression but is composed of patterns:

⟨pat-1⟩ :: ⟨pat-2⟩
And a list pattern looks like a list expression but is also composed of patterns:

\[
\{ \text{pat-1} \}; \{ \text{pat-2} \}; \ldots ; \{ \text{pat-k} \}
\]

Here are some examples of cons and list patterns, and their corresponding matches:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>[] (empty list)</td>
</tr>
<tr>
<td>head::tail</td>
<td>17::[], 17::18::[], [17;18;19]</td>
</tr>
<tr>
<td>(_, 4)::[second]</td>
<td>(2, 4)::[(6,8)], [&quot;two&quot;4;(&quot;six&quot;,8)]</td>
</tr>
<tr>
<td>[first;2;_]</td>
<td>[1;2;3],[-1;2;-3]</td>
</tr>
</tbody>
</table>

The first pattern matches only the empty list. The second pattern matches any nonempty list, and binds head to the “head” (i.e., first element) of the list, and binds tail to the “tail” (i.e., rest) of the list. The third pattern (i) matches any two-element list of tuples for which 4 is the second entry of the first tuple in the list, and (ii) binds second to the second tuple in the list. The fourth pattern matches a three-element list whose second element is 2, and binds first to the first element.

**Question:** In the third pattern above, what would happen if [second] did not have brackets around it?

**Answer:** In that case, second would become a list of arbitrary length, as opposed to a list of length 1 from before.

**Question:** In OCaml, do we use the identifiers head and tail to indicate the first and the rest of a list?

**Answer:** Yes! Or you can use hd and tl to save on typing. Or simply h and t, if space is a premium.

Different languages have different conventions, and it is a good idea to adopt those conventions as quickly as possible, so that you can more easily read others’ code and share your own.

### 5 User-Defined Patterns

The final kind of OCaml patterns are those that arise out of user-defined type definitions. Like the built-in tuple and list constructors, user-defined constructors can be used as patterns. Recall that there are two kinds of user-defined types: variants and records.

#### 5.1 Pattern Matching on Variants

For example, we can define the type season to have four values:

```ocaml
type season =
    | Fall
    | Winter
    | Spring
    | Summer
```
And then we can pattern match values of this type:

```ocaml
match Fall with
  | Fall -> 55
  | Winter -> 25
  | Spring -> 65
  | Summer -> 90
=> 55
```

```ocaml
match Fall with
  | Winter -> "Brrr!"
  | _ -> "Ah..."
=> "Ah...."
```

In these examples, pattern matching is guaranteed to succeed. In the first, we listed all possible values of the variant type. In the second, one of the patterns is a wild card. The following match, however, is not guaranteed to succeed:

```ocaml
match Fall with
  | Fall -> 17
  | Spring -> 18
=> 17
```

Although OCaml evaluates this expression correctly, the compiler warns you that “this pattern-matching is not exhaustive.” The implication of this warning is: Beware, in other situations this expression may not evaluate properly. When you encounter this warning, treat it as a bug.

One way to avoid such bugs is to use a trusty template. In fact, with every variant type comes with such a template. For example:

```ocaml
let proc (s : season) : 'a =
  match s with
    | Fall -> ...
    | Winter -> ...
    | Spring -> ...
    | Summer -> ...
```

Given this template, it is straightforward to write any procedure whose input type is `season`, such as `avg_temp` or `which_course`:

```ocaml
let avg_temp (s : season) : int =
  match s with
    | Fall -> 55
    | Winter -> 25
    | Spring -> 65
    | Summer -> 90

let which_course (s : season) : int =
  match s with
    | Fall -> 17
    | Winter -> failwith "No course offered. Sorry."
    | Spring -> 18
    | Summer -> failwith "No course offered. Sorry."
```
**Notice**: `failwith str` is the OCaml equivalent of Racket’s `make-error str`.

Here is another example of defining a user-defined type, and then creating a template for a procedure that pattern matches on values of that type:

**Variant Type**:

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

**Template**:

```ocaml
let proc (c : card) : int =
  match c with
  | Clubs n -> ... n ...
  | Spades n -> ... n ...
  | Diamonds n -> ... n ...
  | Hearts n -> ... n ...
```

**Sample Procedure**:

```ocaml
let card_value (c : card) : int =
  match c with
  | Clubs n -> n
  | Spades n -> n * 17
  | Diamonds n -> n + 17
  | Hearts n -> n * n
```

Applying the `card_value` procedure to cards works as follows:

- `card_value (Spades 1)`
  => 17

- `card_value (Diamonds 1)`
  => 18

And one last example:

**Variant Type**:

```ocaml
type 'a lunch =
| Soup of 'a
| Salad of 'a
| Sandwich of 'a
```

**Template**:

```ocaml
let proc (l : 'a lunch) : 'b =
  match l with
  | Soup f -> ... f ...
  | Salad f -> ... f ...
  | Sandwich f -> ... f ...
```
Sample Procedure:

```ocaml
let discount (l : float lunch) : float =  
  match l with  
  | Soup f -> f *. 0.75  
  | Salad f -> f *. 0.5  
  | Sandwich f -> f *. 0.25
```

Applying `discount` procedure to lunches works as follows:

```ocaml
discount (Soup 1.0)  
=> 0.75
discount (Salad 1.0)  
=> 0.5
discount (Sandwich 1.0)  
=> 0.25
```

In all the procedures we wrote in this section, we pattern matched on the procedure’s formal argument, but there was no other use of the formal argument in the body of the procedure. As this can happen often, OCaml provides an alternative syntax in which it is not necessary to name the formal argument. Here is an example, based on the `traffic_light` data type:

```ocaml
type traffic_light =  
  | Red  
  | Yellow  
  | Green

let string_of_traffic_light (light : traffic_light) : string =  
  match light with  
  | Red -> "Red"  
  | Yellow -> "Yellow"  
  | Green -> "Green"

let modified_string_of_traffic_light : traffic_light -> string = function  
  | Red -> "Red"  
  | Yellow -> "Yellow"  
  | Green -> "Green"
```

The two key differences between the first and second of these conversion procedures are the type signature and the use of the `function` keyword in place of `match`. Following this idiom, we can rewrite all the procedures defined in this section as follows:

```ocaml
let avg_temp : season -> int = function  
  | Fall -> 55  
  | Winter -> 25 (* no longer *)  
  | Spring -> 65  
  | Summer -> 90 (* and rising *)

let which_course : season -> int = function
```
let card_value : card -> int = function
| Clubs n -> n
| Spades n -> n * 17
| Diamonds n -> n + 17
| Hearts n -> n * n

let discount : float lunch -> float = function
| Soup f -> f *. 0.75
| Salad f -> f *. 0.5
| Sandwich f -> f *. 0.25

5.2 Pattern Matching on Records

Recall the selection operator ".", which can be used to access fields in a record. While in Racket using selectors to access fields is the norm, in OCaml, you can instead access fields using pattern matching. Indeed, pattern matching—which is more concise—is preferred!

Recall the definition of point from last lecture:

\[
\text{(* A point in 2 dimensions is characterized by } x \text{ and } y \text{ coordinates *)}
\]

| type point_2d = {x : int; y : int} |

\[
\text{(* A point in 3 dimensions is characterized by } x, y, \text{ and } z \text{ coordinates *)}
\]

| type point_3d = {x : int; y : int; z : int} |

\[
\text{(* A point is either a 2d or a 3d point *)}
\]

| type point = |
| Point2d of point_2d |
| Point3d of point_3d |

The get\_z\_pm, which uses pattern matching, is preferred to get\_z\_sel, which uses selection:

let get\_z\_pm ({x; y; z} : point_3d) : int = z

let get\_z\_sel (p : point_3d) : int = p.z

You can also combine pattern matching on records (and-types) with pattern matching on variants (or-types):

let get\_x : point -> int = function
| Point2d {x; y} -> x |
| Point3d {x; y; z} -> x

let get\_y : point -> int = function

| Fall -> 17 |
| Winter -> failwith "No course offered. Sorry."
| Spring -> 18 |
| Summer -> failwith "No course offered. Sorry." |
6 Templates in OCaml

The beauty of OCaml is just this: given any user-defined type (recursive or otherwise), pattern matching can easily be used to write a template for procedures that operate on data of that type. And from there, well, it’s all cake and ice cream.

6.1 List Recursion

Recall our Racket template for list recursion:

```racket
(define (proc alod)
  (cond
   [(empty? alod) ...]
   [(cons? alod) ... (first alod) ... (proc (rest alod)) ... ]))
```

We can easily translate this template into OCaml. We use a `match` expression in place of `cond`, and pattern matching in place of selectors and type predicates. In addition, we type annotate `alod`.

```ocaml
let rec proc (alod : 'a list) : 'b =
  match alod with
  | [] -> ...
  | head :: tail -> ... head ... proc tail ...
```

Like `let` in Racket, `let` in OCaml also cannot be used to define a recursive procedure. Instead, we use `let rec`, which allows a procedure to refer to itself.

Using this template, we can write the `length` procedure in OCaml (complete with type annotations):

```ocaml
let rec length (alod : 'a list) : int =
  match alod with
  | [] -> 0
  | _ :: tail -> 1 + length tail
```

Alternatively, we can write the `length` procedure in OCaml (complete with type annotations) like this:

```ocaml
let rec length (alod : 'a list) : int =
  match alod with
  | [] -> 0
  | _ :: tail -> 1 + length tail
```
Here, instead of using the identifier `head`, we use the wildcard `_`. Indeed, it is better style to use `_` because it improves the clarity of our code. Specifically, it signifies that we don’t care about the value of that portion of the pattern we’re matching against.

Having finished writing the `length` procedure, we notice that the formal argument `alod` appears only once in the procedure’s body, as the expression being matched. Consequently, we can simplify our procedure using the `function` keyword, as follows:

```ocaml
let rec length : 'a list -> int = function
    | [] -> 0
    | _ :: tail -> 1 + length tail
```

### 6.2 Natural Number Recursion

When it comes to translating our basic template for natural number recursion from Racket into OCaml, things are not so straightforward. The type “natural number” is not built into OCaml; the only number types are `int` and `float`. We could, if we were feeling inspired, define our own recursive variant type `nat_num`, and then define the necessary operations on natural numbers. But that could be a massive undertaking, with needlessly inefficient consequences.

Instead, here’s the translation. From:

```racket
(define (proc num)
  (cond
    [(zero? num) ...
    [(succ? num) ... num ... (proc (sub1 num)) ... ]
    
```

To:

```ocaml
let rec proc (num : int) =
  match num with
    | 0 -> ...
    | n when (n > 0) -> ... n ... proc (n - 1) ...
    | _ -> failwith "proc: Input is not a natural number."
```

For example:

```ocaml
let rec factorial (num : int) : int =
  match num with
    | 0 -> 1
    | n when (n > 0) -> n * factorial (n - 1)
    | _ -> failwith "factorial: Input is not a natural number."
```

Or, more concisely,

```ocaml
let rec factorial : int -> int = function
    | 0 -> 1
    | n when (n > 0) -> n * factorial (n - 1)
    | _ -> failwith "factorial: Input is not a natural number."
```
6.3 Two Argument Recursion

We close this section with an example of a template for list and natural number recursion combined. The procedure takes two arguments, and then the pattern that is matched is a tuple consisting of those arguments. This is a common implementation strategy in OCaml.

```
let rec proc (n : int) (alod : 'a list) =
  match n, alod with
  | 0, [] -> ...
  | 0, head :: tail -> ...
  | n, [] when (n > 0) -> ...
  | n, head :: tail when (n > 0) -> ...
  | n, _ when (n > 0) -> ...
  | _ , _ -> failwith "proc: Input is not a natural number."
```

For example:

```
let rec take (n : int) (alod : 'a list) : 'a list =
  match n, alod with
  | 0, _ -> []
  | n, [] when (n > 0) -> failwith "too short"
  | n, head :: tail when (n > 0) -> head :: (take (n - 1) tail)
  | _, _ -> failwith "take: Input is not a natural number."

let rec drop (n : int) (alod : 'a list) : 'a list =
  match n, alod with
  | 0, _ -> alod
  | n, [] when (n > 0) -> failwith "too short"
  | n, _ :: tail when (n > 0) -> drop (n - 1) tail
  | _, _ -> failwith "drop: Input is not a natural number."
```

One major benefit of pattern matching, which all of the above code demonstrates, is that it takes the place of using selectors, like `first` and `rest`. Instead, every component of the matched value that you care about is bound to an identifier of your choice. When the identifier names are sensible and follow established conventions, this makes your code eminently readable.

7 Procedure Definitions

In OCaml, there are two kinds of procedures: primitive, which are often written using infix notation, and user-defined, which are always written using prefix notation.

**Note:** A prefix procedure in OCaml can be applied to one argument only!

In particular, all user-defined procedures can be applied to one argument only. This may seem limiting, but it is not, because of currying, as you will learn shortly.

**Procedure Definitions** The shape of a (one-argument) procedure definition is:

```
let <id> <arg> = <body>
```

For example,
let add17 num = num + 17

When applied to an argument (an `int`, specifically), this procedure evaluates as expected:

```
add17 1
=> 1 + 17
=> 18
```

The analog of this code in Racket is:

```
(define (add17 num) (+ num 17))
(add17 1) => (+ 1 17) => 18
```

**Anonymous Procedures** We all think that lambda is fun, don’t we? With that in mind, the shape of an anonymous procedure in OCaml is quite reasonable:

```
fun <arg> -> <body>
```

In fact, `fun` is an abbreviation for `function`. But why bother typing `function` when it is so much fun to type `fun`.

Here is an example:

```
fun num -> num + 17
```

Anonymous procedure application works just as it does in Racket (except that it is even more fun):

```
(fun num -> num + 17) 1
=> 1 + 17
=> 18
```

The analog of this procedure application in Racket is:

```
((lambda (num) (+ num 17)) 1) => (+ 1 17) => 18
```

**Syntactic Sugar** Now, can you guess what was really going on when we defined the `add17` procedure? If you guessed that the name `add17` was associated with the anonymous procedure `(fun num -> num + 17)`, then you guessed correctly.

In other words,

```
let add17 num = num + 17
```

is actually syntactic sugar for

```
let add17 = fun num -> num + 17
```

So, in the above evaluation of `add17 1`, we glossed over what was going on under the hood. Here is the whole story, with nothing swept under the rug:
add17 1
=> (fun num -> num + 17) 1
=> 1 + 17
=> 18

The analog of this derivation in Racket is:

(let (add17 1)
  => ((lambda (num) (+ num 17)) 1)
  => (+ 1 17)
  => 18)

More generally, just like in Racket

let <id> <arg> = <body>

is syntactic sugar for

let <id> = fun <arg> -> <body>

Type Annotations If a procedure takes an argument of type ⟨arg-type⟩ and returns a value of type ⟨val-type⟩, then its type signature is written as ⟨arg-type⟩ -> ⟨val-type⟩. For example, both sample OCaml procedures above (add17 and the anonymous procedure) are of type int -> int.

When defining procedures, it is good practice to annotate the procedures’ inputs with their types. For example,

let add17 (num : int) = num + 17

fun (num : int) -> num + 17

You should also type annotate a procedure’s output, like this:

let add17 (num : int) : int = num + 17

(fun (num : int) -> num + 17 : int -> int)

Question: How do you type in the arrows?
Answer: You type a “minus” sign followed by a “greater than” sign.

That completes the story for one-argument procedures, and modulo syntax, things work as they did in Racket. When we come to two argument procedures, however, we will find that OCaml differs from Racket, because OCaml does not technically support multi-argument procedures.
8 Summary

Ideas

- Variant types are an extremely useful new OCaml tool — they allow you to define your own types.
- Pattern matching with the keyword `match` is our OCaml analogue to Racket’s `cond`.
- Pattern matching supports built-in patterns — such as constant, identifier, tuple, cons, and list patterns — and user-defined patterns.
- Technically speaking, OCaml doesn’t have multi-parameter procedures. It uses a technique called currying to bind multiple arguments to one function body, so you can define a multi-argument procedure in practice.

Skills

- To make debugging OCaml code easier, add type annotations to all relevant variables and procedures. Doing this makes it much easier to track down OCaml type inference bugs later on.
- Make sure to check out the Racket to OCaml guide for more info on exact OCaml syntax, and strategies for putting Racket ideas into OCaml.
- We can implement pattern matching with built-in and user-defined patterns.
- We know how to create a template for one-argument and two-argument list recursion in OCaml.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs017/feedback](http://cs.brown.edu/courses/cs017/feedback)