CAML Memo*

Undergraduate S1

Nathalie "Junior" Bouquet

2023

Elements of the Language

Special Characters, Separators and Comments

Some characters and character combinations have special meanings as operators, separators, blanks... They will be seen as we use them. Comments are delimited by `(*` and `*)`.

Identifier Construction Rules

- Identifiers are sequences of letters (`'a'..'z' 'A'..'Z'`), digits (`'0'..'9'`) and '_' (the underscore character).
- CAML is case-sensitive (not only for identifiers).
- Identifiers start with:
  - a lowercase letter[1];
  - an underscore '_' if followed by at least another character.
- Identifiers must differ from keywords.
- Identifiers must be **meaningful**!

Basic Data Types and Operators

Integers: `int`

Range of values	64-bits platforms: $[-2^{62}, 2^{62}-1]$ (32-bits: $[-2^{30}, 2^{30}-1]$)
Values:	1 45 -69
Operators:	`+ - * /` (integer division[2]) `mod = <> < > <= >=`
Some functions:	`succ pred abs`
Special values:	`max_int min_int`

Floating-point numbers: `float`

Double precision (64 bits) numbers:	mantissa on 53 bits and exponent $\in [-1022, 1023]$
Values:	12.5 -0.5 3. -3. 3e2 5.75e-2
Operators:	`+. -. *. /. = <> < > <= >=`
Some functions:	`sqrt ceil floor abs_float log cos` ...
Special values:	`max_float min_float infinity neg_infinity nan`

Booleans: `bool`

Values:	`true false`
Operators:	`not || && = <>`

Characters: `char`

ASCII code:	$[0, 127]$
ISO 8859-1 standard:	$[128, 255]$
Values:	`'a' 'A' '$' '9' '\065' '\n'` [3]
Operators:	`= <> < > <= >=`
Some functions:	`Char.code Char.chr Char.escaped`

*All CAML examples here were evaluated (interpreted) under CAML 4.07.

[1] Except for constructors or modules names (not part of the program).

[2] Only in $\mathbb{N}$ for now.

[3] `'\065'`: character with ASCII code 65. `'\n'`: linefeed.

Character strings: `string`

Finite sequence of characters.

Length	$[0, 2^{57} - 9]$ (32-bit: $2^{24} - 5$)
Values:	"a string" "a" "" (empty string)
Operators:	`^` (concatenation) `=` `<>` `<` `>` `<=` `>=`
Access to a character:	`s.[i]` is the i^{th} $(0 \leq i < \text{length}(s))$ character (type `char`) of `s`.
Some functions:	`String.length String.sub`

Some conversion functions

`float_of_int : int -> float`	integer → floating-point
`int_of_float : float -> int`	floating-point → integer (truncated)
`int_of_char : char -> int` = `Char.code`	character → ASCII code
`char_of_int : int -> char` = `Char.chr`	ASCII code → character
`Char.escaped : char -> string`	character → string
`string_of_int : int -> string`	integer → string
`int_of_string : string -> int`	string → integer
`string_of_float : float -> string`	floating-point → string
`float_of_string : string -> float`	string → floating-point

1 Phrases: Expressions and Definitions

Phrases are either simple **expressions** or **definitions (declarations)**.
When using the interactive mode, phrases must be finished with `;;`.

1.1 Expressions

An expression can be either:

primary :	a simple value	`15`	`x`
structured :	an operation	`a + 15`	`"Hello " ^ "world"`
	between parentheses	`(3*a)`	`(true || false)`
	a function application	`cos x`	`float_of_int 15`

```
# (666 * 42 = 27972) && ("Hello" < "World");;
 - : bool = true
```

An expression may contain local definition(s) (see bellow), alternatives, "pattern matching" (studied later)...

1.2 Definitions

Simple "global" definitions

`let` *ident* = *expression*

Let binding: The name *ident* is **bound** to the value of the *expression*.

```
# let a = 1 + 2 ;;
 val a : int = 3
```

Once bound, a name is bound to the same value (cannot be changed), but it can be hidden by a new binding with the same name...

Multiple global definitions

`let` $ident_1$ = $expression_1$
`and` $ident_2$ = $expression_2$
...
`and` $ident_n$ = $expression_n$

```
# let one = 1 and two = 2. and three = '3' ;;
 val one : int = 1
 val two : float = 2.
 val three : char = '3'
```

Local definitions

"Simple" local definition: `let` *ident* `=` *expression*$_1$ `in` *expression*$_2$

```
# let a = 1 + 2 in a * 3 ;;
 - : int = 9
```

Multiple local definitions:

`let` *ident*$_1$ $=$ *expr*$_1$ `and` ... `and` *ident*$_i$ $=$ *expr*$_i$
`in let` *ident*$_{i+1}$ $=$ *expr*$_{i+1}$ `and` ... `and` *ident*$_j$ $=$ *expr*$_j$
...
`in` *expression*$_n$

```
# let a = 1 and b = 3 in
    let x = a + b and y = a - b in
        x * y ;;
 - : int = -8
```

An expression with a local definition is an expression

2 Functions

2.1 Functions with One Parameter

Define a function: `let` *f x* $=$ *expression*
Type: `f :` *type*$_x$ `->` *type*$_{res}$

```
# let succ x = x + 1 ;;
 val succ : int -> int = <fun>
```

Application of the function *f* to the value *x*: *f x*

```
# succ 3 ;;
 - : int = 4
```

Function applications have higher precedence than other operators.

$f\ x + y \quad \equiv \quad (f\ x) + y$ $\qquad$ $f\ x + g\ y \quad \equiv \quad (f\ x) + (g\ y)$

```
# succ 3*2 ;;
 - : int = 8
```

```
# succ (3 * 2) ;;
 - : int = 7
```

Thus, if the parameter is not a primary expression, it has to be parenthesized.

```
# succ -1 ;;
 Characters 0-4:[4] Error: This expression has type int -> int
                           but an expression was expected of type int
# succ (-1) ;;
 - : int = 0
```

2.2 Functions with Several Parameters

Definition: `let` *f x y* $=$ *expression*
Type: `f :` *type*$_x$ `->` *type*$_y$ `->` *type*$_{res}$

```
# let average a b = float_of_int(a + b) /. 2.;;
 val average : int -> int -> float = <fun>
```

Application to *x* and *y*: $f\ x\ y \quad \equiv \quad (f\ x)\ y$

```
# average 2 3 ;;  (* same as (average 2) 3 *)
 - : float = 2.5
# average ( 2 3) ;;
 Error: This expression has type int
        This is not a function; it cannot be applied.
```

Function applications associate to the left

Therefore: $f\ (g\ x) \quad \neq \quad f\ g\ x$

```
# succ succ 2 ;;
 Error: This function has type int -> int
        It is applied to too many arguments; maybe you forgot a ';'.[5]
# succ (succ 2) ;;
 - : int = 4
```

Watch out: `let` *f (x, y)* $=$ *expression* is a function with 1 parameter (the pair *(x, y)*)
of type *type*$_x$ `*` *type*$_y$ `->` *type*$_{res}$

[4] Indication of the scope of the errors will be omitted on the next examples.

[5] This (';') will be explained later...

3 Case Analysis

3.1 Conditional

`if` *condition* `then` $expression_1$ `else` $expression_2$

- *condition* is a boolean expression ;
- $expression_1$ and $expression_2$ are of the same type ;
- the conditional evaluates to the value of $expression_1$ if *condition* evaluates to the boolean true, and to the value of $expression_2$ if *condition* evaluates to the boolean false.

The conditional is an expression.

```
# if 2 > 1 then
      "higher"
  else
      "lower" ;;
 - : string = "higher"
```

```
# let absolute_value x =
    if x >= 0 then
        x
    else
        -x ;;
val absolute_value : int -> int = <fun>
```