OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*

DecOrExp ::= Dec | Exp

Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp

Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'

Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Extended BNF (EBNF) grammars

- BNF is extended with the regular expressions operators `*`, `+`, `?`
- Example: `Pat+` means `Pat` concatenated one or more times
- Remark: `+` (reg-exp operator) is different from `'+'` (terminal symbol)

OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*
DecOrExp ::= Dec | Exp
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Quick comments

- `ID` variable identifiers `(_[\w']|[a-zA-Z])[\w']*`
- `NUM` natural numbers
  `0[bB][01][01_]*|0[oO][0-7][0-7_]*|0[xX][\da-fA-F][\da-fA-F_]*|\d[\d_]*`
- `UOP` unary arithmetic operators `[+-]`
- `BOP` binary arithmetic operators `[+-*/]|mod`
- `Pat` patterns: very simple for now, a more complete definition will be considered later on

OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*
DecOrExp ::= Dec | Exp
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Functions and application

- examples of functions

```
let inc = fun x -> x+1 (* the increment function *)
let inc2 x = x+1 (* a more compact syntax *)
```

- function application (= function call)

```
inc 3 (* syntax inc(3) optional, evaluation returns 4 *)
inc2 3 (* syntax inc2(3) optional, evaluation returns 4 *)
```

OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*
DecOrExp ::= Dec | Exp
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Functions and application

- examples of anonymous function

```
fun x -> x+1 (* the increment function *)
```

- function application (= function call)

```
(fun x -> x+1) 3 (* evaluation returns 4 *)
```

OCaml core

EBNF grammar defining a simplified syntax

```
Prog ::= (DecOrExp ';;')*
DecOrExp ::= Dec | Exp
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
Exp ::= ID | NUM | Exp Exp | 'fun' Pat+ '->' Exp | UOP Exp |
    Exp BOP Exp | '(' Exp ')'
Pat ::= ID | '_' |  '(' Pat ')' // simplified pattern
```

Semantics of function application

`exp1 exp2`

- `exp1` is evaluated in a function *f*
- `exp2` is evaluated in the argument *a* of *f*
- `exp1 exp2` is evaluated in $f(a)$ (*f* applied to *a*)

OCaml core

Precedence and associativity rules

- standard rules for arithmetic expressions
- application has higher precedence then binary operators

```
inc 1+2 (* equivalent to (inc 1)+2 *)
1+inc 2 (* equivalent to 1+(inc 2) *)
```

- anonymous functions have lower precedence

```
fun x->x+1 (* equivalent to fun x->(x+1) *)
fun f->f 2 (* equivalent to fun f->(f 2), not (fun f->f) 2 *)
```

- more critical cases: application and unary operators

```
inc + 3 (* addition *)       inc (+3) (* application *)
inc - 3 (* subtraction *)    inc (-3) (* application *)
+ inc 3 (* is +(inc 3) *)    - inc 3    (* is -(inc 3) *)
```

OCaml type inference

A simple interpreter session (Read Eval Print Loop)

Types can be automatically deduced (=inferred) by the interpreter!

```
# 42;;
- : int = 42
# let inc x = x+1;;
val inc : int -> int = <fun>
# inc 2;;
- : int = 3
```

Simplified syntax of OCaml core type expressions

BNF Grammar

```
Type ::= 'int' | Type '->' Type | '(' Type ')'
```

OCaml core types

Terminology

- `int` is a built-in simple type: the type of integers
- `int -> int` is a built-in composite type
- `->` is a type constructor: it is used for building composite types from simpler types
- types built with the `->` (arrow) constructor are called *arrow types* or *function types*

Meaning of arrow types

t_1 `->` t_2 is the type of functions from t_1 to t_2 that

- can only be applied to a single argument of type t_1
- always returns values of type t_2

OCaml core types

Remarks

- the arrow type constructor is right-associative

  `int->int->int` $=$ `int->(int->int)`

- a type constructor always builds a type different from its type components

  $t_1 \neq t_1$`->`t_2 and $t_2 \neq t_1$`->`t_2

- two arrow types are equal if they are built with the same type components

  t_1`->`$t_2 = t$ if and only if $t = t_3$`->`t_4, $t_3 = t_1$, $t_4 = t_2$

- Remark: from the items above

  `int->(int->int)` $\neq$ `(int->int)->int`

Types and type expressions

Remarks

- `int->int->int` is a type expressions, but is also called a type, because it represents a specific type
- `int->int->int` and `int->(int->int)` are different type expressions which represent the same type

Higher order functions in OCaml

A useful syntactic abbreviation

fun $pat_1\ pat_2 \ldots pat_n$ -> *exp*

is an abbreviation for

fun pat_1 -> **fun** pat_2 -> ... **fun** pat_n -> *exp*

Examples

```
# fun x y->x+y
- : int -> int -> int = <fun>
# fun x->fun y->x+y
- : int -> int -> int = <fun>
# fun x y z->x*y*z
- : int -> int -> int -> int = <fun>
# fun x->fun y->fun z->x*y*z
- : int -> int -> int -> int = <fun>
```

Higher order functions in OCaml

Examples

- `(int->int)->int`: a function that takes a function `int->int` and returns an integer `int`
- `int->(int->int)`: a function that takes an integer `int` and returns a function `int->int`
- `(int->int)->(int->int)`: a function that

  takes a function `int->int` and returns a function `int->int`

Recall

`->` is right associative, therefore

- `int->(int->int)=int->int->int`
- `(int->int)->(int->int)=(int->int)->int->int`

Higher order functions in OCaml

Examples

```
# let apply_f_to_0_and_inc = fun f -> 1+f 0;;
val apply_f_to_0_and_inc : (int -> int) -> int = <fun>

# let add x y = x+y ;; (* let add = fun x->fun y->x+y *)
val add : int -> int -> int = <fun>

# add 3 4;;
- : int = 7

# let apply_f_to_x_square_and_inc = fun f -> fun x -> 1+f (x*x) ;;
val apply_f_to_x_square_and_inc : (int -> int) -> int -> int = <fun>

# let mul x y z = x*y*z ;; (* let mul = fun x->fun y->fun z->x*y*z *)
val mul : int -> int -> int -> int = <fun>

# mul 2 3 4 ;;
- : int = 24
```

Tuples in OCaml

Syntax

New productions for `Exp` and `Pat`

```
Exp ::= '(' ')' | Exp (',' Exp)+
Pat ::= '(' ')' | Pat (',' Pat)+
```

New production for `Type`

```
Type ::= 'unit' | Type ('*' Type)+
```

Precedence and associativity rules

- the tuple constructor `,` has higher precedence than anonymous functions
- the tuple constructor `,` has lower precedence than the other operators
- the tuple constructor `,` is neither left nor right associative
- the `*` type constructor has higher precedence than the `->` constructor
- the `*` type constructor is neither left nor right associative

Tuples in OCaml

, is a value constructor with arity ≥ 2

- $(v_1, \ldots, v_n) \neq v_i$ for all $i = 1 \ldots n$
- $(v_1, \ldots, v_n) = v$ if and only if $v = (v'_1, \ldots, v'_n)$ and $v_i = v'_i$ for all $i = 1 \ldots n$

$*$ is a type constructor with arity ≥ 2

- $t_1 * \ldots * t_n \neq t_i$ for all $i = 1 \ldots n$
- $t_1 * \ldots * t_n = t$ if and only if $t = t'_1 * \ldots * t'_n$ and $t_i = t'_i$ for all $i = 1 \ldots n$

Tuples in OCaml

Examples

```
# ()
- : unit = () (* this is the void value *)
# print_int;; (* predefined, prints an integer on stdout *)
- : int -> unit = <fun>
# 1,2,3
- : int * int * int = (1, 2, 3)
# (1,2),3
- : (int * int) * int = ((1, 2), 3)
# 1,(2,3)
- : int * (int * int) = (1, (2, 3))
```

Functions with tuple arguments

Examples

```
# let const3() = 3;; (* let const3 = fun()->3 *)
val const3 : unit -> int = <fun>

# let add(x,y) = x+y;; (* let add = fun(x,y) -> x+y *)
val add : int * int -> int = <fun>

# add (3,4);;
- : int = 7

# let mul(x,y,z) = x*y*z;; (* let mul = fun (x,y,z)->x*y*z *)
val mul : int * int * int -> int = <fun>

# mul (2,3,5);;
- : int = 30
```

Declarations of global “variables”

Grammar

```
Dec ::= 'let' Pat '=' Exp | 'let' ID Pat+ '=' Exp
```

A simple example

```
# let x=2;;
val x : int = 2
# let y=x+40;;
val y : int = 42
# x+y;;
- : int = 44
```

Remarks

- variables are global because they are declared at the top level
- local variables can be declared as well at inner levels (see later)
- the content of variables cannot be changed, variables are constant
- there is no variable assignment

Declarations of global “variables”

More elaborate examples

```
# let x=2;;
val x : int = 2
# let y=x+40;;
val y : int = 42
# x+y;;
- : int = 44
# let x=3;; (* previous declaration is shadowed *)
val x : int = 3
# x+y;;
- : int = 45
# let x,y = 3+2,3*2;;
val x : int = 5
val y : int = 6
```

Remark

a new declaration with the same name shadows the previous declaration

Declarations of global “variables”

Examples of global function declarations

```
# let inc x = x+1;;
val inc : int -> int = <fun>
# let add (x,y) = x+y;;
val add : int * int -> int = <fun>
# let add2 x y = x+y;;
val add2 : int -> int -> int = <fun>
```

A useful syntactic abbreviation

let *id* $pat_1\ pat_2 \ldots pat_n$ = *exp*

is an abbreviation for

let *id* = **fun** $pat_1\ pat_2 \ldots pat_n$ -> *exp*

which is an abbreviation for

let *id* = **fun** pat_1 -> **fun** pat_2 -> ... **fun** pat_n -> *exp*

Curried/uncurried functions

Multiple arguments can be handled in two different ways

- Curried function (from Haskell Curry) with n arguments:
  a higher-order function returning a “chain” of (higher order) functions

  $$\texttt{fun}\ \mathit{pat}_1\ \texttt{->}\ \texttt{fun}\ \mathit{pat}_2\ \texttt{->}\ \ldots\ \texttt{fun}\ \mathit{pat}_n\ \texttt{->}\ \mathit{exp}$$

- Uncurried function with n arguments:
  a function taking as argument a tuple of size n

  $$\texttt{fun}\ (\mathit{pat}_1, \mathit{pat}_2, \ldots, \mathit{pat}_n)\ \texttt{->}\ \mathit{exp}$$

Correspondence between curried and uncurried function

- an uncurried function can be transformed in the equivalent curried version
- a curried function can be transformed in the equivalent uncurried version
- isomorphism between type $t_1\ \texttt{->}\ t_2\ \texttt{->}\ \ldots\ \texttt{->}\ t_n\ \texttt{->}\ t$ and type $t_1\ \star\ t_2\ \star\ \ldots\ \star\ t_n\ \texttt{->}\ t$

Curried/uncurried functions

Examples

```
(* addition of two integers *)
# fun x y->x+y;; (* curried version *)
- : int -> int -> int = <fun>

# fun (x,y)->x+y;; (* uncurried version *)
- : int * int -> int = <fun>

(* multiplication of three integers *)
# fun x y z->x*y*z;; (* curried version *)
- : int -> int -> int -> int = <fun>

# fun (x,y,z)->x*y*z;; (* uncurried version *)
- : int * int * int -> int = <fun>
```

Partial application

Curried functions and partial application

- curried functions allow partial application:

  arguments can be passed once at time

- uncurried functions do not allow partial application:

  arguments must be passed altogether

Partial application

Example

```
# let uncurried_add(x,y)=x+y;;
val uncurried_add : int * int -> int = <fun>

# uncurried_add(1,2);; (* both arguments must be passed *)
- : int = 3

# let curried_add x y=x+y;;
val curried_add : int -> int -> int = <fun>

# let inc=curried_add 1;; (* only argument 1 is passed *)
val inc : int -> int = <fun>

# inc 2;; (* argument 2 is passed to compute the final result *)
- : int = 3
```

Remark: the result of the partial evaluation is saved in `inc` as a useful by-product, 1+2 can always be computed with the single expression `curried_add 1 2`

Partial application

Partial application promotes generic programming

Partial application allows function specialization: from a generic function it is possible to generate more specific ones with no code duplication.

- software reuse and maintenance are favored
- interesting examples will be shown later on