Pocaml Language Reference Manual

Feitong Qiao, Yiming Fang, Yunlan Li, Peter Choi

***Abstract*—This document describes the Pocaml language's syntax. The Pocaml language is a functional language that implements the core subset of the OCaml language, with type inference and many features from OCaml's standard library.**

I. Lexical Aspects

A. Blanks

Characters including space, tab, carriage return (\r), line feed (\n), and form feed are considered blanks in Pocaml. They serve to separate the program into tokens.

B. Comments

Comments begin with the 2-character sequence (* and end with the 2-character sequence *). Comments do not occur within a string or character literals. In nested comments, all opening (* should be closed with a corresponding *).

```
(* this is a comment *)
(* this is a
multi-line
comment *)
(* this is a (* nested *) comment *)
(* this is not (* a valid comment *)
```

C. Identifier

Identifiers are sequences of letters, digits, _ (the underscore character), and ' (the single quote), starting with a letter or an underscore. Letters contain the lowercase and uppercase alphabets from ASCII. In many places, Pocaml distinguishes between capitalized and non-capitalized identifiers. Underscore is considered a lowercase letter for this purpose.

ident ::= (*letter*|_){*letter*|*0...9*|_|'}
uppercase-ident ::= (*A...Z*){*letter*|*0...9*|_|'}
lowercase-ident ::= (*a...z*|_){*letter*|*0...9*|_|'}
letter ::= *A...Z*|*a...z*

D. Integer literals

An integer literal is a sequence of one or more digits, optionally preceded by a minus sign. Integer literals are in decimal.

integer-literal `::=` `[-]` (*0...9*) { *0...9* }

E. Boolean literals

A boolean literal is either $true$ or $false$. They have the type $Bool$.

bool :

$true$
$false$

F. Character literals

Characters include the regular set of characters and the escape sequence, which serve to delimit characters.

char-literal ::= *regular-char* | *escape-sequence*
escape-sequence ::= \(" |'| n | t | b | r | *space*)

II. Expressions

A. Lvalues

An *lvalue* represents a storage location that can be assigned a value: variables and parameters.

$lvalues$:

id

B. Return values

The return value of a `let-in` expression is the value after the in. `if-then-else` and other functions, including operators, have the return value equal to the result of the corresponding computation.

C. List Literals

Array expressions can be defined as `[e1; e2;...;en]` and must be explicitly typed. For example, one may say `let lst: int list = [1;2;3]`. Pocaml supports the efficient appending of the head element `e1` to the tail list `[e2; ...; en]`, using the operator ::, as well as the less efficient concatenation between two lists using the operator @.
Furthermore, pattern matching is possible with lists as follows

D. Lambda Functions

The lambda functions are used in Pocaml using the keyword `fun` by specifying the operations on the function input. They can be used as expressions and passed as argument into other functions.

E. Function Calls

A function application is a prefix expression `id arg1 arg2 ...` with zero or more blank-separated expression parameters. Functions applications are curried. The values of the parameters are strictly evaluated from left to right and bound to the function's formal parameters using conventional static scoping rules.

Partial function applications are supported and a function that takes in the remaining arguments is returned.

F. Operators

The binary operators are $+, -, *, /, =, <>, <, >, <=, >=$ $, \&\&, ||$.

A leading minus sign negates an integer expression.

Parentheses group expressions in the usual way.

The binary operators $+, -, *, /$ require integer operands and return an integer result.

The binary operators $=, <>, >, <, >=, <=$ compare the operands, which may be either both integer or both string and produce $true$ if the comparison holds and $false$ otherwise. String comparison is done using normal ASCII lexicographic order.

The binary operators $\&\&, ||$ do the usual logical AND and OR on two boolean values.

Unary minus has the highest precedence followed by $*, /$, then $+, -$, then $=, <>, >, <, >=, <=$, then $\&\&$, then $||$.

G. Flow Control

The brancing expression **if** *expr*$_1$ **then** *expr*$_2$ **else** *expr*$_3$ evaluates to *expr*$_2$ if *expr*$_1$ evaluates to **true**. Otherwise, it evaluates to *expr*$_3$. *expr*$_i$ is an *expr* and is used here simply for ease of referring to different expressions that appear in the branching expression.

H. Let

The expression **let** *declaration* **in** *expr* produces a set of name to value bindings that are accessible within *expr-list*. The let expression evaluates to the value of the last expression in *expr-list*.

I. Pattern Matching

A pattern matching expression is in the form of *match expr1 with pattern-matching*, where *pattern-matching* is a sequence of clauses in the form of $pattern \rightarrow exprValue$, separated by pipes |. The value of the entire pattern matching expression is the $exprValue$ of the first $pattern$ that $expr1$ matches.

III. Declarations

A Pocaml program is a sequence of declarations.

$declaration$:

$lvalue = expr$

function-declaration

type-declaration

A. Let declaration

The declaration **let** *declaration* is used only at the top level. It produces a name to value binding that can be accessed globally within the same file.

B. Types

Pocaml has predefined types including `int`, `bool`, `char` New types can be defined using the following context free grammar rules.

type-declaration :

`type` *type-id* = *type*

type :

type-id

`array` *of* *type-id*

C. Functions

function-declaration :

let id *args*$_{opt}$ = *expr*

let rec id *args*$_{opt}$ = *expr*

let id *args*$_{opt}$: *type* = *expr*

let rec id *args*$_{opt}$: *type* = *expr*

$args$:

id

(id : *type*)

$param \quad param$

The last two forms is a function declaration of the first two with return type annotation. The first two form declares a function named id that takes in zero or more parameters defined by param; *expr* is the body of the function. The scope of the function arguments is *expr*. The **rec** keyword defines a recursive function whose id is available in the scope of *expr*.

The following function declarations are equivalent and both functions have type $int \rightarrow int \rightarrow int$.

let fun1 (a: int) (b: int) = a + b
let fun2 (a: int) = fun (b:int) → a + b

IV. Standard Library

$print(s : string)$
Print the string to the standard output.

map : (‘$a \rightarrow$ ‘b) $\rightarrow$ ‘a *list* $\rightarrow$ ‘b *list*
Apply a function to each element of a list to return a new list with the original type.

$iter$: (‘$a \rightarrow unit$) $\rightarrow$ ‘a *list* $\rightarrow unit$
Call a function with each element of a list.

$append$: ‘a *list* $\rightarrow$ ‘a *list* $\rightarrow$ ‘a *list*
Return a new array containing the concatenation of two arrays

$fold\_left$: (‘$a \rightarrow$ ‘$b \rightarrow$ ‘a) $\rightarrow$ ‘$a \rightarrow$ ‘b *list* $\rightarrow$ ‘a
fold_left f lst init applies function f on the current accumulator (initially $init$) and each element in lst,

going from left to right. It returns the current accumulator after going through the whole list.

$fold\_right : (`a \to `b \to `a) \to `a \to `b\ list \to `a$
fold_right f lst init applies function f on the current accumulator (initially $init$) and each element in lst, going from right to left. It returns the current accumulator after going through the whole list.

V. Example

This example demonstrates how to implement Euclid's algorithm for finding the Greatest Common Denominator (GCD), printing the result after finding the answer. This code snippet showcases many features of our ls, such as `let-in` declaration, recursive function call, type specification, and control flow statements.

```
let rec gcd (a : int) (b : int) : int =
    if b = 0 then a
    else gcd b (a mod b)
let print_gcd (a : int) (b : int) : () =
    print_endline (string_of_int (gcd a b))
```