Caml tail

Caml Light, but smaller and less useful

Language Reference Manual

Jennifer Lam
Fiona Rowan
Sandra Shaefer
Serena Shah-Simpson

March 2, 2016
Table of Contents

1. Lexical Elements
   1.1. Separators
   1.2. Phrases
   1.3. Comments
   1.4. Identifiers
   1.5. Integer literals
   1.6. Floating point literals
   1.7. Character literals
   1.8. String literals
   1.9. Keywords and special symbols

2. Values
   2.1. Lists
   2.2. Constants
   2.3. Functions

3. Type expressions
   3.1. Type variables
   3.2. Parenthesized types
   3.3. Function types

4. Patterns
   4.1. Variable patterns
   4.2. Alias patterns
   4.3. Parenthesized patterns
   4.4. “Or” patterns
   4.5. Constant patterns

5. Expressions
   5.1. Simple expressions
   5.2. Control constructs
   5.3. Operators

6. Standard Library
   6.1. General
   6.2. List operations

7. Sample Code
   7.1. Rec function
   7.2. Pattern matching
   7.3. Nested functions and anonymous functions
   7.4. Standard library use
Introduction

*Caml tail* is a functional programming language that is based on OCaml -- without the O. *Caml tail* will be strongly typed (with a smaller set of primitive types). It will support basic arithmetic, pattern matching, function definitions both imperative and recursive, anonymous functions, and nested functions. Our language will compile to LLVM.

1. Lexical Elements

This part of the manual describes the lexical elements of *tail*.

1.1 Separators

Horizontal tabs, newlines, carriage returns, and whitespace separate tokens. They are otherwise ignored. Double semicolons delineate phrases. A single semicolon is used as a sequencer of 1) multiple expressions, or 2) list elements.

1.2 Phrases

```
phrase:
    expr
    | value-definition
    | type-definition
```

A phrase is defined as an expression, a value definition, and a type definition. It is always terminated by double semicolons.

1.3 Comments

Comments are ignored, and are enclosed by (* and *).

1.4 Identifiers

```
ident:
    letter {letter | 0...9 | _}* 

letter:
    A...Z | a...z 
```
Identifiers are a sequence of characters beginning with a letter.

1.5 Integer literals

\texttt{int\_literal:}
\[\begin{align*}
[-] & \{0...9\}^+ \\
\end{align*}\]

Integer literals are a sequence of digits in the decimal system, optionally beginning with a minus sign to signify negative numbers.

1.6 Floating point literals

\texttt{float\_literal:}
\[\begin{align*}
[-] & \{0...9\}^+ \cdot \{0...9\}^+ \\
\end{align*}\]

Floating point literals are a sequence of digits with a decimal point at the beginning, end, or any other intermediary position within the sequence. An optional minus sign at the beginning designates a negative number.

1.7 Character literals

\texttt{char\_literal:}
\[\begin{align*}
(a...zA...Z) \\
\mid \` \( \mid \` \) \mid n \mid t \mid b \mid r) ` \\
\mid \` \{0...9\} \{0...9\} \{0...9\} ` \\
\end{align*}\]

Character literals are single letters, digits, or symbols enclosed by two single quotes.

1.8 String literals

String literals are sequences of letters, digits, or symbols enclosed by two double quotes. Strings can be concatenated via a carat: “O”^ “Caml, guys!” evaluates to “OCaml, guys!”

1.9 Keywords and special symbols

Keywords are special identifiers that are predefined by the language, and cannot clash with user-defined function or variable names. Below is the list of keywords in our language:

\begin{verbatim}
int char string float unit bool if then else let match with
\end{verbatim}
fun     func     rec     true     false     type
in      as       abst     open

The following sequence of characters are special symbols:

| : ) ( _ -> , = [ ] == != < <= > >= && | |
not + - * / mod +. -. *. / . :: ^ ‘ “ ; ; ; >. >= .< <=.

2. Values

2.1 Lists

typedcl list :

    [ item; item; item;...] | [ item ] | []

item:

    int
    | float
    | char
    | string

A list is designated by opening and closing square brackets. All items in a list must be literals. The literals must be of the same type (no type mixing or tuples).

2.1.1 List Operators

Prepend to a list is denoted by double colons. For example, (new_item :: existing_list) prepends new item to an existing list.

2.2 Constants

The following table shows constant values.
false the boolean false
true the boolean true
unit the void value
[] the empty list

2.3 Functions
Functional values are mappings from values to values.

3. Type expressions

3.1 Type variables
These are variables that are bound to types, which may be user defined. They take the following format: `typedef identifier = value_1 | ... | value_n`.

3.2 Parenthesized Types
A parenthesized type represents the same type as are inside the parentheses.

3.3 Function Types
The expression type `type_1 -> type_2` represents the type mapping of a function, which takes input of type `type_1` and outputs type `type_2`.

4. Patterns

4.1 Variable patterns
These patterns consist of identifiers and match any value, thus binding that variable to said value. The wildcard symbol `_` also matches any value, but there is no binding involved.

4.2 Alias patterns
These are patterns bound to identifiers, and are created in the following format: `pattern as identifier`. If a value matches this pattern successfully, the value is bound to the identifier.
4.3 Parenthesized patterns
A pattern enclosed in parentheses matches the same value as just that pattern would. Type constraints can be placed on patterns in this way as well, in the following manner: (pattern : type).

4.4 “Or” patterns
Or patterns are represented by two patterns separated by the symbol |. A value matches an “or” pattern if it either of these patterns.

4.5 Constant patterns
These patterns consist of constants, and only match values that are equal to those constants.

5. Expressions

5.1 Simple expressions

5.1.1 Variables
An expression consisting of a variable always evaluate to the value bounded to that variable.

5.1.2 Parenthesized expressions
An expressions inside parentheses evaluate to the value of that expression.

5.1.3 Function abstraction
Here, we use the keyword fun to match a set of values to a set of patterns. If the set of values matches the $i$th row of patterns, then the $i$th expression is evaluated -- that is, if the value $v_j$ matches the $pattern_j$ in row $i$, $expr_i$ is evaluated. The first row of patterns matched is the one whose expression is evaluated. These statements occur in the following format:

```
fun abst
  pattern\_1 \ldots pattern\_m -> expr\_1,
  \mid pattern\_1 \ldots pattern\_m -> expr\_2,
  \ldots
  \mid pattern\_1 \ldots pattern\_m -> expr\_n,
  \mid _ -> expr\_default
```
All rows must have the same number of patterns. If each row has only one pattern, use the keyword \texttt{match} instead of \texttt{fun} (see section 4.23). A default expression must exist in the case that the set of values does not match any of the given rows of patterns.

5.1.4 Function application

The expression \texttt{expr₁ expr₂ … exprₙ} evaluates the expression \texttt{exprₙ} to \texttt{expr₁}. \texttt{exprₙ} must evaluate to a functional value, which is applied to the values that follow.

5.1.5 Local definitions

We bind variables locally using the following formats:

\[
\text{let pattern₁ = expr₁ and \ldots and patternₙ = exprₙ in expr}
\]

Each of the indexed expressions are evaluated, and if their values match their corresponding patterns, then \texttt{expr} is evaluated as the value of the entire \texttt{let} statement. If matchings succeed, \texttt{expr} is evaluated in environment enriched by bindings performed during matching(s), and the value of \texttt{expr} is returned as value of whole \texttt{let} expression. Local variables defined in the preceding pattern matches can be used in the evaluation of \texttt{expr}.

Alternatively, we may bind recursive definitions of variables locally with the following format:

\[
\text{let rec pattern₁ = expr₁ and \ldots and patternₙ = exprₙ in expr}
\]

5.1.6 Function definitions

Anonymous functions have variable input specified within parentheses. The last expression evaluated is the functional return value of the function, and is immediately applied to the parameter values specified after the function body in parentheses. The type of value, must match the type of identifier. Anonymous functions are defined in the following way:

\[
type_{\text{return}} \text{ fun (type₁ identifier₁, \ldots, typeₙ identifierₙ) ->}
\]

\[
\text{expr₁;}
\]

\[
\text{\ldots
\}
\]

\[
\text{exprₘ (* the last expr returns a value of type}_{\text{return}} *) (value₁, \ldots, valueₙ)
\]

Declarative functions can be bound to an identifier. The last expression evaluates to the return type, and the functions have the following format:

\[
\text{let type_{\text{return}} function name (type₁ formal_arg₁, \ldots, typeₙ formal_argₙ) =}
\]

\[
\text{expr₁;}
\]

\[
\text{\ldots
\}
\]

\[
\text{exprₘ (* the last expr returns a value of type}_{\text{return}} *)
\]
Recursive functions use the `rec` keyword, and have the following format:

\[
\text{let rec type}_{\text{return}} \text{function\_name (type}_{1} \text{, identifier}_{1}, \ldots , \text{type}_{n} \text{, identifier}_{n}) =
\]

\[
\text{expr}_{1};
\]

\[
\ldots
\]

\[
\text{function\_name( ident}_{1}, \text{ident}_{2}, \ldots , \text{ident}_{n})
\]

5.2 Control constructs

5.2.1 Sequence

The expression \( \text{expr}_{1}; \text{expr}_{2} \) evaluates \( \text{expr}_{1} \) first and then returns the value of \( \text{expr}_{2} \).

5.2.2 Conditional

The expression \( \text{if expr}_{1} \text{ then expr}_{2} \text{ else expr}_{3} \) evaluates the first function, and if that function evaluates to true, it evaluates the second function and returns its value. Otherwise, the third function is evaluated and its value is returned. The else part may be omitted.

5.2.3 Case Expression

The following expression matches \( \text{expr} \) with the following sequence of patterns. If it matches one of the patterns, its corresponding expression is evaluated, and the entire match expression evaluates to its value. If \( \text{expr} \) matches multiple patterns, the first pattern matched is considered the successful matching.

\[
\text{match expr}
\]

\[
\text{with pattern}_{1} \rightarrow \text{expr}_{1}
\]

\[
\text{| pattern}_{2} \rightarrow \text{expr}_{2}
\]

\[
\ldots
\]

\[
\text{| pattern}_{n} \rightarrow \text{expr}_{n}
\]

\[
\text{| _ \rightarrow expr}_{\text{default}}
\]

5.3 Operators

The following are operators within boolean expressions:

\[
\text{==} \quad \text{infix} \quad \text{Equality test.}
\]

\[
\text{!=} \quad \text{infix} \quad \text{Inequality test.}
\]

\[
\text{<} \quad \text{infix} \quad \text{“Less than” integer test.}
\]

\[
\text{<=} \quad \text{infix} \quad \text{“Less than or equal to” integer test.}
\]

\[
\text{>} \quad \text{infix} \quad \text{“Greater than” integer test.}
\]
The following are numerical operators:

- `+`  infix  Integer addition.
- `-`  infix  Integer subtraction.
- `-`  prefix  Integer negation.
- `*`  infix  Integer multiplication.
- `/`  infix  Integer division.
- `mod`  infix  Integer or float modulus.

The following are string and list operators:

- `^`  prefix  String concatenation.

The prepend operator is a list operator, and consists of `::`. The expression `expr_1 :: expr_2` evaluates to the list with `expr_1` as its head and `expr_2` as its tail.

6. Standard library

The standard library can be included using the `open` keyword.

6.1 General

<table>
<thead>
<tr>
<th>function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print_char</code></td>
<td>Prints a character to standard output.</td>
</tr>
<tr>
<td><code>print_string</code></td>
<td>Prints a string to standard output.</td>
</tr>
<tr>
<td><code>print_int</code></td>
<td>Prints an integer to standard output.</td>
</tr>
<tr>
<td><code>print_float</code></td>
<td>Prints a float to standard output.</td>
</tr>
<tr>
<td><code>random()</code></td>
<td>Generates a random float between 0 and 1.</td>
</tr>
<tr>
<td><code>int_to_string</code></td>
<td>Converts integers to strings.</td>
</tr>
<tr>
<td><code>char_to_string</code></td>
<td>Converts characters to strings.</td>
</tr>
<tr>
<td><code>float_to_string</code></td>
<td>Converts float to strings.</td>
</tr>
</tbody>
</table>
6.2 List operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>len</td>
<td><code>List.len</code> returns the length of a list.</td>
</tr>
<tr>
<td>rev</td>
<td><code>List.rev</code> returns a new list with the items of the original list reversed.</td>
</tr>
<tr>
<td>hd</td>
<td><code>List.hd</code> returns the first element of the list.</td>
</tr>
<tr>
<td>tl</td>
<td><code>List.tl</code> returns a list consisting of all elements of the original list except the head.</td>
</tr>
<tr>
<td>nth</td>
<td><code>List.nth</code> returns the nth item.</td>
</tr>
<tr>
<td>iter</td>
<td><code>List.iter</code> applies function f in turn to a1; ...; an.</td>
</tr>
<tr>
<td>map</td>
<td><code>List.map</code> applies function f to a1, ..., an, and builds the list [f a1; ...; f an] with the results returned by f.</td>
</tr>
<tr>
<td>fold_left</td>
<td><code>List.fold_left</code> function accumulator [item1; item2; ...; itemn] is equivalent to function (accumulator item1) item2 ... itemn)</td>
</tr>
<tr>
<td>fold_right</td>
<td><code>List.fold_right</code> folds the opposite way as <code>List.fold_left</code>.</td>
</tr>
<tr>
<td>concat</td>
<td><code>List.concat</code> concatenates a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result.</td>
</tr>
</tbody>
</table>

7. Sample code

7.1 Rec function

```haskell
let rec int fib (int n) =
    if n=1 then 1;
    else
        if n=2 then 1;
        else fib(n-1) + fib(n-2);
```

7.2 Pattern matching

```haskell
let rec int fib (int n) = match n with
  1 -> 1
| 2 -> 1
| _ -> fib(n-1) + fib(n-2);
```

```haskell
let int gcd (int a, int b) =
  let rec int gcd_helper (int c, int d, int r) =
    match r with
      0 -> d
    | _ -> let c = c % d in let gcd_helper(d, c, d % c)
     in gcd_helper(a, b, a % b);
```

7.3 Nested and anonymous functions

```haskell
let int x = 5 in
  (int fun (int y) -> y + x) x;;
```

(* This line evaluates to (((function x -> x+5) 5), which returns 10 *)

7.4 Standard library use

(* Demonstrating a hello world program *)
let unit hello_world() =
  let string yo = "Hello ";
  let string dude = "World!";
  print_string (yo ^ dude);
hello_world(); (* Prints Hello World! to the screen *)

(* Demonstrating list operations *)
let unit hello() =
  let char gniteerg = ['o'; 'l'; 'l'; 'e'; 'H'];
  let char greeting = List.rev gniteerg;
  List.iter print_char greeting;,)
hello(); (* Prints Hello to the screen *)