EBook Gratis

APRENDIZAJE OCaml

Free unaffiliated eBook created from
Stack Overflow contributors.

#ocaml

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: ocaml

It is an unofficial and free OCaml ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official OCaml.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con OCaml

Observaciones

Esta sección proporciona una descripción general de qué es ocaml y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema grande dentro de ocaml, y vincular a los temas relacionados. Dado que la Documentación para ocaml es nueva, es posible que deba crear versiones iniciales de los temas relacionados.

Examples

Instalación o configuración

Instalando OPAM

OPAM es un gestor de paquetes para OCaml. Construye y administra las versiones del compilador y las bibliotecas OCaml para usted fácilmente.

La forma más fácil de instalar OPAM en su sistema operativo es usar un administrador de paquetes para su sistema. por ejemplo, apt-get, yum o homebrew.

Instrucciones de instalación de Mac OSX

Actualice las fórmulas caseras e instale OPAM.

```
brew update
brew install opam
```

Instrucciones de instalación de Ubuntu

```
add-apt-repository ppa:avsm/ppa
apt-get update
apt-get install ocaml opam
```

Compilando desde la fuente

```
wget http://caml.inria.fr/pub/distrib/ocaml-4.03/ocaml-4.03.0.tar.gz
tar xf ocaml-4.03.0.tar.gz
cd ocaml-4.03.0
./configure -prefix installation_path
make world.opt
```

```
make install
```

Inicializando OPAM

Una vez que haya instalado OPAM, ejecute `opam init` y siga las instrucciones.

Una vez hecho esto, debería poder ejecutar el intérprete OCaml desde su shell.

```
$ ocaml
        OCaml version 4.03.0

#
```

Tu primer programa en OCaml

Ahora que la distribución de OCaml está disponible en su sistema operativo favorito, podemos crear su primer programa en OCaml: ¡Hello World!

Tenemos diferentes formas de lanzar un programa OCaml.

El REPL (nivel superior)

Puede ejecutar su código de forma *interactiva* con el nivel *superior* . Con el nivel *básico de* OCaml, puede escribir y ejecutar código OCaml, como un shell de UNIX. Luego, el nivel *superior* verifica el tipo de su código inmediatamente. Por lo tanto, puede probar rápida y fácilmente algunas partes del código sin compilación ni ejecución.

Puede iniciar el *nivel* `ocaml` con el comando `ocaml` . Luego, puedes escribir una *oración* OCaml terminada por `;;` que se evalúa de inmediato. El nivel *superior* muestra el tipo y el valor de su expresión justo después de:

```
# "Hello Worlds!";;
- : string = "Hello Worlds!"
```

También es posible iniciar el nivel *superior* en su archivo. Puedes ver esta explicación sobre eso.

Para facilitar su entrada en el nivel *superior* , puede usar una herramienta como `ledit` o `rlwrap` que proporciona algunas características (como el historial de entrada):

```
$ ledit ocaml

$ rlwrap ocaml
```

Recopilación al *bytecode*

Tenemos dos compiladores diferentes, uno que compila a *bytecode* y el otro que compila a código nativo. El primero es el mismo que el *código* de *bytes* de la máquina virtual de Java. Entonces, el *bytecode* es menos eficiente pero más portátil.

Tenemos algunos archivos de extensiones utilizados por los compiladores OCaml:

extensión	definición
`.ml`	El código fuente (como `.c` en C)
`.mli`	La interfaz (como `.h` en C)
`.cmo`	Código fuente compilado por `ocamlc` en *bytecode*
`.cmi`	Código de interfaz compilado por `ocamlc`
`.cmx` y `.o`	Código fuente compilado por `ocamlopt` en código nativo
`.cma`	Biblioteca (cubo de algunos `*.cmo`) en *bytecode*
`.cmxa` y `.a`	Biblioteca en código nativo
`.cmxs`	Biblioteca en código nativo (para cargar dinámicamente)

El compilador de *bytecode* es `ocamlc` .

Tienes diferentes opciones comunes:

- `-c` : para compilar un archivo fuente sin el proceso de vinculación (para producir un ejecutable). Entonces, el comando `ocaml -c foo.ml` produce un archivo `.cmo` . A diferencia de C, en el que no es necesario compilar el archivo de `.mli` en OCaml es necesario compilar el archivo `ocaml -c foo.mli` : `ocaml -c foo.mli` .

Necesitas compilar la interfaz primero. Después de compilar el archivo fuente, OCaml intenta verificar que la implementación coincida con la interfaz.

El archivo `.mli` no es obligatorio. Si compila un archivo `.ml` sin un archivo `.mli` , OCaml producirá un archivo `.cmi` automáticamente.

- `-o` : para compilar algunos archivos `.cmo` a un ejecutable. Por ejemplo: `ocamlc -o program foo.cmo bar.cmo` . Estos archivos deben estar ordenados por las dependencias para las cuales el primer archivo no tiene dependencia.

- `-I` : para indicar otro directorio donde el compilador puede encontrar los archivos necesarios para la compilación (como la interfaz o el código fuente). Es lo mismo que el `-I` de un compilador de C.

Tenemos muchas otras opciones. Puedes ver el manual para más información.

Entonces, puede escribir el `hello.ml` ahora, y compilar este archivo con `ocamlc -o hello hello.ml`

para producir un programa de *bytecode* :

```
let () = print_endline "Hello World!"
```

El `let () = ...` es la primera entrada de su programa (como el `main` en C). Después, usamos la función `print_endline` (proporcionada por la biblioteca estándar) con el argumento `"Hello World!"` para imprimir `Hello Worlds` con una nueva línea en la salida estándar.

Después de la compilación, tiene el archivo `.cmo` y el archivo `.cmi` producidos automáticamente por el compilador y su programa `hello` . Puedes abrir tu programa, y en la parte superior de este archivo, puedes ver:

```
#!/usr/local/bin/ocamlrun
```

Eso significa que su programa necesita el programa `ocamlrun` (proporcionado por la distribución) para ejecutar el *bytecode* (como la JVM).

Compilación al código nativo.

Tenemos otro compilador que produce código nativo. El compilador es: `ocamlopt` . Sin embargo, el ejecutable resultante no puede funcionar en la mayoría de las otras arquitecturas.

`ocamlopt` usa las mismas opciones que `ocamlc` para que pueda ejecutar `ocamlopt -o hello hello.ml` Después, puedes ver un `.cmx` y un archivo `.o` .

Finalmente, desde su programa de código de *bytes* / código nativo, puede ejecutar:

```
$ ./hello
Hello World!
$
```

Instalación en Windows (nativo)

Premisa

Esta instrucción muestra un procedimiento para instalar binarios nativos de OCaml en Windows. Si su sistema operativo es `Windows 10 (Insider Preview) build 14316` o posterior, también puede instalar OCaml a través de Bash en Ubuntu en Windows . En este caso, siga las instrucciones para instalar OCaml en Ubuntu.

Instala OCaml y Opam

Descargar la distribución oficial de OCaml . Contiene tanto compiladores OCaml como gestor de paquetes Opam. Supongamos que ha instalado el software en `C:/OCaml` . Para asegurarse de haber instalado correctamente OCaml, abra `cmd.exe` y escriba `ocaml` .

Si ve el mensaje, `'ocaml' is not recognized as an internal or external command, operable program or batch file` , debe agregar `C:/OCaml/bin` a su ruta (variable de entorno).

Añadir binarios OCaml a la ruta

en `Control Panel > System and Security > System > Advanced system settings (on the left) > Environment Variables` y luego seleccione `Path` en la pestaña `System Variable` , luego `Edit` .

Añadir `C:/OCaml/bin;` a la lista.

Instalar cygwin

Sin Cygwin no puedes usar Opam. De hecho, si intenta abrir Opam escribiendo `opam` en `cmd.exe` se mostrará un mensaje: `Fatal error: exception Unix.Unix_error(20, "create_process", "cygcheck")` .

Descarga Cygwin e inicia el instalador. Asegúrese de revisar los siguientes paquetes:

- automake
- difutiles
- libreadline
- hacer
- m4
- mingw64-x86_64-gcc-core
- mingw64-x86_64-gmp
- mingw64-x86_64-openssl
- mingw64-x86_64-pkg-config
- mingw64-x86_64-sqlite3
- parche
- envoltura
- abrir la cremallera
- wget

Supongamos que ha instalado el software en `C:/cygwin` (`C:/cygwin64` para la versión de 64 bits). Abra `cmd` y escriba `wget` (o uno de los ejecutables presentes en `C:/cygwin/bin`) para verificar si puede usar los ejecutables de Cygwin. Si el archivo ejecutable no se abre, agregue `C:/cygwin/bin` a su ruta (Variable de entorno).

Configurar Opam

Abra `cmd.exe` y escriba `opam init` para configurar Opam.

Luego instale `ocamlfind` (parte del compilador OCaml) con

```
opam install ocamlfind
opam config env
```

Compruebe si `ocamlfind` está instalado escribiéndolo en `cmd.exe` .

El comando `opam config env` se usa para agregar el directorio de ejecutables de `opam` a la ruta del entorno. Si después de cerrar la sesión ya no puede alcanzar `ocamlfind` , puede agregarlo manualmente agregando a la ruta la siguiente línea: `C:/Users/<your user>/Documents/.opam/system/bin/` .

Instalando paquetes

Los paquetes se instalan a través de Opam con el comando `opam install xyz` donde `xyz` es el nombre del paquete.

Instalar UTop

Intente ejecutar el comando `opam install utop` . Si no tiene errores, al escribir `utop` se abrirá el ejecutable.

Si ves el mensaje

```
[ERROR] The compilation of zed failed at "ocaml setup.ml -build".
```

Tienes que instalar manualmente los paquetes individuales. Intente de nuevo escribiendo:

```
opam install zed
opam install lambda-term
opam install utop
```

Tanto `lambda-term` como `utop` pueden no instalarse. Ver la sección de resolución de problemas.

Instalación de Core

Puede instalar el paquete `core` con `opam install core` . En la versión de Windows de 64 bits (y 64 bits de Cygwin) verá el siguiente error:

```
[ERROR] core is not available because your system doesn't comply with os != "win32" & ocaml-version = "4.02.3".
```

Solucionar problemas: no se puede crear un archivo normal

Si el paquete con el nombre `xyz.10.1` no se instala (donde xyz es el nombre del paquete y 10.1 su versión) con el siguiente mensaje:

```
install: cannot create regular file '/cygdrive/c/Users/<your user>/Documents/.opam/system/bin/<something>': File exists
```

Tienes que ir en este directorio:

```
C:\Users\<your user>\Documents\.opam\repo\default\packages\<xyz>\<xyz.10.1>\files
```

y elimine el archivo `xyz.10.1.install` .

Solución de problemas: no se puede cargar la biblioteca compartida

Si intenta abrir algún paquete de Opam (por ejemplo: `utop`) y ve este error:

```
Fatal error: cannot load shared library dlllwt-unix_stubs
Reason: The specified module could not be found.
```

Ejecute `opam config env` nuevamente e intente volver a abrir el ejecutable.

Lea Empezando con OCaml en línea: https://riptutorial.com/es/ocaml/topic/1826/empezando-con-ocaml

Capítulo 2: Campos de registro mutables

Introducción

Como la mayoría de los valores de OCaml, los registros son inmutables por defecto. Sin embargo, dado que OCaml también maneja la programación imperativa, proporciona una manera de hacer que los campos individuales sean *mutables* . Los campos mutables se pueden modificar in situ por asignación, en lugar de tener que recurrir a las técnicas funcionales habituales, como la actualización funcional.

Al introducir efectos secundarios, los campos mutables pueden dar como resultado un mejor rendimiento cuando se usan correctamente.

Examples

Declarar un registro con campos mutables.

A continuación, el `weight` se declara como un campo mutable.

```
type person = {
  name: string;
  mutable weight: int
};;
```

Observación : En lo que se refiere al diseño aquí, uno podría considerar el hecho de que el nombre de una `person` probablemente no cambie, pero su peso sí lo es.

Inicializando un registro con campos mutables

La inicialización de un registro con campos mutables no es diferente de la inicialización de un registro regular.

```
let john = { name = "John"; weight = 115 };;
```

Estableciendo el valor en un campo mutable

Para asignar un nuevo valor a un campo de registro mutable, use el operador `<-` .

```
john.weight <- 120;;
```

Nota : La expresión anterior tiene un tipo de `unit` .

Lea Campos de registro mutables en línea: https://riptutorial.com/es/ocaml/topic/9367/campos-de-registro-mutables

Capítulo 3: Errores comunes

Examples

Usando el operador equivocado

En OCaml, hay diferentes operadores aritméticos para flotadores y enteros. Además, estos operadores solo se pueden usar en 2 flotantes o 2 enteros. Aquí hay expresiones inválidas en OCaml

```
1.0 + 2.0
1 + 2.0
1 +. 2
1 +. 2.0
```

La expresión correcta para cada uno de estos respectivamente son

```
1. +. 2.
float_of_int 1 +. 2.
1 + 2
float_of_int 1 +. 2.
```

No hay conversión automática de enteros a flotadores o viceversa en OCaml. Todo es explícito. Aquí hay una lista de los operadores de enteros y flotadores.

Operación	Operador entero	Operador de flotador
Adición	`a + b`	`c +. d`
Sustracción	`a - b`	`c -. d`
Multiplicación	`a * b`	`c *. c`
División	`a / b`	`c /. d`
Módulo	`a mod b`	`modfloat cd`
Exposicióncián	N / A	`c ** d`

Donde `a` y `b` son enteros y `c` y `d` son flotadores.

Olvidando paréntesis alrededor de argumentos de función

Un error común es olvidar los argumentos circundantes de funciones compuestas entre paréntesis, lo que lleva a errores de tipo.

```
# string_of_int 1+1;;
```

```
Error: This expression has type string but an expression was expected of type int
```

Esto se debe a la precedencia. De hecho, lo anterior evalúa a

```
# (string_of_int 1) + 1;;
```

Cuál está mal. Una sintaxis correcta sería

```
# string_of_int (1+1);;

- : string = "2"
```

Lea Errores comunes en línea: https://riptutorial.com/es/ocaml/topic/8146/errores-comunes

Capítulo 4: Escribe tu primer Script OCaml

Examples

Hola Mundo

Este ejemplo asume que has instalado OCaml .

Compilando el código OCaml

Cree un nuevo archivo llamado `hello.ml` , con el siguiente contenido:

```
print_string "Hello world!\n"
```

`ocamlc` es el compilador de OCaml. Para compilar y ejecutar este script, ejecute

```
$ ocamlc -o hello hello.ml
```

y luego ejecutar el binario resultante

```
$ ./hello
Hello world!
```

Ejecutando código OCaml

También puede ejecutar este script sin compilarlo en un binario. Puede hacerlo utilizando `ocaml` , el sistema de nivel básico de ocaml que permite el uso interactivo de OCaml. En tu shell, simplemente ejecuta

```
$ ocaml hello.ml
Hello world!
```

En la réplica

Abra un nuevo shell y escriba `ocaml` para abrir el sistema de nivel superior. Una vez en la sesión, puede escribir el mismo programa:

```
        OCaml version 4.02.1

# print_string "hello world!\n";;
```

pulse Intro para evaluar la expresión y desencadenar la impresión.

```
hello world!
- : unit = ()
```

¡Éxito! Lo vemos impreso `hello world!` , pero de que se trata el `- : unit = ()` ? OCaml no tiene declaraciones, todo es una expresión que se evalúa a algún valor escrito. En este caso, `print_string` es una función que toma una `string` como entrada y devuelve una `unit` . Piense en la `unit` como un tipo que solo puede tomar un valor, `()` (también denominado unidad), y representa un cálculo final que no devuelve ningún valor significativo.

En este caso, `print_string` también tiene el efecto secundario de poner los caracteres que recibió como entrada en la pantalla, por lo que vemos la primera línea.

Para salir del REPL, presione `ctrl+D`

Como un script de Unix

Tenemos dos formas de crear un script OCaml. El primero usa el nivel superior del sistema (proporcionado por su administrador de paquetes como `apt-get`) y el segundo usa el nivel superior proporcionado por OPAM .

Usa el sistema de nivel superior.

Abre tu editor favorito y escribe:

```
#!/usr/bin/ocaml

print_string "hello worlds!\n";;
```

Después, puedes usar `chmod +x your_file.ml` y puedes ejecutar tu script con `./your_file.ml` .

Utilice el nivel superior proporcionado por OPAM

```
#!/usr/bin/env ocaml

print_string "hello worlds!\n";;
```

La gran diferencia es sobre la versión de tu nivel superior. De hecho, si configuró su OPAM con un switch específico (como `opam switch 4.03.0`), el script usará OCaml 4.03.0. De la primera forma, en Debian Sid, por ejemplo, el script usará OCaml 4.02.3.

Puede reemplazar el *shebang* por `#!/usr/bin/env utop` para usar `utop` lugar del nivel de vainilla.

utop

`utop` es otro nivel de ocaml fuera de la distribución, es decir, necesita descargar e instalar `utop` (la

forma más sencilla es usar OPAM: `opam install utop`). `utop` tiene muchas características como el histórico, la finalización y la edición de la línea interactiva.

Entonces, si quieres una manera fácil de probar algunos códigos de `utop` , `utop` es la mejor.

¿Por qué `utop` y no `ocaml` ?

`utop` y `ocaml` no tienen una gran diferencia si quieres un script ocaml como el de arriba. Pero lo común en la comunidad de OCaml es usar `utop` lugar de `ocaml` .

De hecho, el `ocaml` REPL es proporcionado por la distribución de ocaml. Por lo tanto, este REPL sigue el ciclo de lanzamiento del compilador y si desea algunas características adicionales, debe esperar el próximo lanzamiento del compilador. `utop` , como explicamos, está fuera de la distribución, por lo que el ciclo de lanzamiento no está restringido por el compilador y, si desea una característica adicional, es más probable que intente presionar esta característica dentro de `utop` que `ocaml` :).

Para este punto (y para la característica histórica) la mayoría de las personas en la comunidad de ocaml prefieren usar `utop` que `ocaml` .

Lea Escribe tu primer Script OCaml en línea: https://riptutorial.com/es/ocaml/topic/2168/escribe-tu-primer-script-ocaml

Capítulo 5: Funciones

Examples

Definiendo una función con un enlace let

A los valores se les puede dar nombres usando `let` :

```
# let a = 1;;
val a : int = 1
```

Puedes usar una sintaxis similar para definir una función. Simplemente proporcione parámetros adicionales para los argumentos.

```
# let add arg1 arg2 = arg1 + arg2;;
val add : int -> int -> int = <fun>
```

Podemos llamarlo así:

```
# add 1 2;;
- : int = 3
```

Podemos pasar valores directamente de esa manera, o podemos pasar valores vinculados a nombres:

```
# add a 2;;
- : int = 3
```

La línea que nos da el intérprete después de definir algo es el valor del objeto con su tipo de firma. Cuando le dimos un valor simple vinculado a `a` , regresó con:

```
val a : int = 1
```

Lo que significa que `a` es un `int` , y su valor es `1` .

La firma tipo de nuestra función es un poco más complicada:

```
val add : int -> int -> int = <fun>
```

El tipo de firma de `add` ve como un manojo de flechas y flechas. Esto se debe a que una función que toma dos argumentos es en realidad una función que solo toma un argumento, pero devuelve otra función que toma el siguiente argumento. En su lugar, podrías leerlo así:

```
val add : int -> (int -> int) = <fun>
```

Esto es útil cuando queremos crear diferentes tipos de funciones sobre la marcha. Por ejemplo,

una función que agrega 5 a todo:

```
# let add_five = add 5;;
val add_five : int -> int = <fun>
# add_five 5;;
- : int = 10
# add_five 10;;
- : int = 15
```

Usando la palabra clave de la función

La palabra clave de `function` tiene una coincidencia de patrón automáticamente cuando define el cuerpo de su función. Obsérvalo a continuación:

```
# let foo = function
0 -> "zero"
| 1 -> "one"
| 2 -> "couple"
| 3 -> "few"
| _ -> "many";;
val foo : int -> bytes = <fun>

# foo 0;;
- : bytes = "zero"

# foo 3;;
- : bytes = "few"

# foo 10;;
- : bytes = "many"

# let bar = function
"a" | "i" | "e" | "o" | "u" -> "vowel"
| _ -> "consonant";;
val bar : bytes -> bytes = <fun>

# bar "a";;
- : bytes = "vowel"

# bar "k";;
- : bytes = "consonant"
```

Funciones anonimas

Dado que las funciones son valores ordinarios, hay una sintaxis conveniente para crear funciones sin nombres:

```
List.map (fun x -> x * x) [1; 2; 3; 4]
(* - : int list = [1; 4; 9; 16] *)
```

Esto es útil, ya que de otro modo tendríamos que nombrar la función primero (ver let) para poder usarla:

```
let square x = x * x
```

```
(* val square : int -> int = <fun> *)

List.map square [1; 2; 3; 4]
(* - : int list = [1; 4; 9; 16] *)
```

Funciones recursivas y recursivas mutuas

Puede definir una función para que sea recursiva con la palabra clave `rec` , para que pueda llamarse a sí misma.

```
# let rec fact n = match n with
    | 0 -> 1
    | n -> n * fact (n - 1);;

val fact : int -> int = <fun>

# fact 0;;
- : int = 1
# fact 4;;
- : int = 24
```

También puede definir funciones recursivas entre sí con la palabra clave `and` , para que puedan llamarse entre sí.

```
# let rec first x = match x with
    | 1 -> 1
    | x -> second (x mod 10)

  and second x = first (x + 1);;

val first : int -> int = <fun>
val second : int -> int = <fun>

# first 20;;
- : int = 1
# first 12345;;
- : int = 1
```

Observe que la segunda función no tiene la palabra clave `req` .

Lea Funciones en línea: https://riptutorial.com/es/ocaml/topic/2793/funciones

Capítulo 6: Funciones de orden superior

Sintaxis

- `val (|>) : 'a -> ('a -> 'b) -> 'b`
- `val (@@) : ('a -> 'b) -> 'a -> 'b`

Examples

Algoritmos genéricos

Se pueden usar funciones de orden superior para implementar algoritmos genéricos, renunciando a la responsabilidad de proporcionar detalles finales al usuario. Por ejemplo, `List.sort` espera una función de comparación, que permite implementar varias formas de clasificación. Aquí implementamos la clasificación de cadenas insensibles a mayúsculas y minúsculas:

```
let string_case_insensitive_sort lst =
  let case_insensitive_compare a b =
    String.compare (String.lowercase a) (String.lowercase b)
  in
  List.sort case_insensitive_compare lst
```

Hay una lista rica de funciones de orden superior en la biblioteca estándar, especialmente en el módulo *Lista* , vea `List.fold_left` y `List.sort` por ejemplo. Se pueden encontrar ejemplos más avanzados en bibliotecas de terceros. Un buen ejemplo es el recocido simulado implementado en **ocaml-gsl** . El recocido simulado es un procedimiento de optimización genérico que está parametrizado por una función utilizada para explorar el conjunto de estados del problema y una función de error (denominada aquí función de energía).

Los usuarios familiarizados con C ++ pueden comparar esto con el patrón de *Estrategia* .

Disponer los recursos del sistema incluso cuando se produce una excepción

Se pueden usar funciones de orden superior para garantizar que los recursos del sistema se eliminan, incluso cuando un tratamiento genera una excepción. El patrón utilizado por `with_output_file` permite una clara separación de preocupaciones: las funciones de orden superior `with_output_file` se encargan de administrar los recursos del sistema vinculados a la manipulación de archivos, mientras que el tratamiento `f` solo consume el canal de salida.

```
let with_output_file path f =
  let c = open_out path in
  try
    let answer = f c in
    (close_out c; answer)
  with exn -> (close_out c; raise exn)
```

Usemos esta función de orden superior para implementar una función escribiendo una cadena en

un archivo:

```
let save_string path s =
  (with_output_file path) (fun c -> output_string c s)
```

Usando funciones más avanzadas que `fun c -> output_string cs` es posible guardar valores más complejos. Vea, por ejemplo, el módulo *Marshal* en la biblioteca estándar o la biblioteca *Yojson* de Martin Jambon.

Operadores de composicion

Dos funciones útiles de orden superior son los operadores de *aplicación* binaria (`@@`) y *de aplicación inversa* o "pipe" (`|>`). Aunque desde la versión 4.01 están disponibles como primitivas, podría ser instructivo definirlas aquí:

```
let (|>) x f = f x
let (@@) f x = f x
```

Considere el problema de incrementar el cuadrado de 3. Una forma de expresar ese cálculo es la siguiente:

```
(* 1 -- Using parentheses *)
succ (square 3)
(* - : int = 10 *)

(* where `square` is defined as: *)
let square x = x * x
```

Tenga en cuenta que no podríamos simplemente hacer `succ square 3` porque (debido a la asociatividad a la izquierda) se reduciría al significado sin sentido `(succ square) 3` . Usando la aplicación (`@@`) podemos expresar eso sin los paréntesis:

```
(* 2 -- Using the application operator *)
succ @@ square 3
(* - : int = 10 *)
```

¿Observe cómo la última operación que se realiza (es decir, `succ`) ocurre primero en la expresión? El operador de *aplicación inversa* (`|>`) nos permite, bueno, revertir esto:

```
(* 3 -- Using the reverse-application operator *)
3 |> square |> succ
(* - : int = 10 *)
```

El número 3 ahora está "canalizado" a través del `square` y luego `succ` , en lugar de aplicarse al `square` para obtener un resultado al que se aplica `succ` .

Lea Funciones de orden superior en línea: https://riptutorial.com/es/ocaml/topic/2729/funciones-de-orden-superior

Capítulo 7: La coincidencia de patrones

Examples

Función factorial utilizando el patrón de coincidencia

```
let rec factorial n = match n with
| 0 | 1 -> 1
| n -> n * (factorial (n - 1))
```

Esta función hace coincidir los valores 0 y 1 y los asigna al caso base de nuestra definición recursiva. Luego, todos los otros números se asignan a la llamada recursiva de esta función.

Evaluación de expresiones booleanas.

Definimos el tipo de expresiones booleanas cuyos átomos son identificados por cadenas como

```
type expr =
| Atom of string
| Not of expr
| And of expr * expr
| Or of expr * expr
```

y puede evaluar estas expresiones usando un `oracle : string -> bool` da los valores de los *átomos* que encontramos como sigue:

```
let rec eval oracle = function
| Atom(name) -> oracle name
| Not(expr) -> not(eval oracle expr)
| And(expr1, expr2) -> (eval oracle expr1) && (eval oracle expr2)
| Or(expr1, expr2)  -> (eval oracle expr1) || (eval oracle expr2)
```

Vea cómo la función es clara y fácil de leer. Gracias al uso correcto de la coincidencia de patrones, un programador que lea esta función necesita poco tiempo para asegurarse de que se implementa correctamente.

Negación de forma normal: emparejamiento profundo patrón

La coincidencia de patrones permite deconstruir valores complejos y de ninguna manera se limita al nivel "más externo" de la representación de un valor. Para ilustrar esto, implementamos la función que transforma una expresión booleana en una expresión booleana donde todas las negaciones son solo en los átomos, la llamada *forma normal de negación* y un predicado que reconoce las expresiones de esta forma:

Definimos el tipo de expresiones booleanas cuyos átomos son identificados por cadenas como

```
type expr =
```

```
| Atom of string
| Not of expr
| And of expr * expr
| Or of expr * expr
```

Primero definamos un predicado que reconozca expresiones en *forma normal de negación* :

```
let rec is_nnf = function
| (Atom(_) | Not(Atom(_))) -> true
| Not(_) -> false
| (And(expr1, expr2) | Or(expr1, expr2)) -> is_nnf expr1 && is_nnf expr2
```

Como ve, es posible hacer coincidencias con patrones anidados como `Not(Atom(_))` . Ahora implementamos una función que asigna una expresión booleana a una expresión booleana equivalente en forma normal de negación:

```
let rec nnf = function
| (Atom(_) | Not(Atom(_))) as expr -> expr
| Not(And(expr1, expr2)) -> Or(nnf(Not(expr1)),nnf(Not(expr2)))
| Not(Or(expr1, expr2)) -> And(nnf(Not(expr1)),nnf(Not(expr2)))
| And(expr1, expr2) -> And(nnf expr1, nnf expr2)
| Or(expr1, expr2) -> Or(nnf expr1, nnf expr2)
| Not(Not(expr)) -> nnf expr
```

Esta segunda función hace aún más usos de patrones anidados. Finalmente podemos probar nuestro código en el nivel superior en la negación de una implicación:

```
# let impl a b =
Or(Not(a), b);;
  val impl : expr -> expr -> expr = <fun>
# let expr = Not(impl (Atom "A") (Atom "B"));;
val expr : expr = Not (Or (Not (Atom "A"), Atom "B"))
# nnf expr;;
- : expr = And (Atom "A", Not (Atom "B"))
# is_nnf (nnf expr);;
- : bool = true
```

El sistema de tipo OCaml es capaz de verificar la exhaustividad de una coincidencia de patrones. Por ejemplo, si omitimos el caso `Not(Or(expr1, expr2))` en la función `nnf` , el compilador emite una advertencia:

```
# let rec non_exhaustive_nnf = function
| (Atom(_) | Not(Atom(_))) as expr -> expr
| Not(And(expr1, expr2)) -> Or(nnf(Not(expr1)),nnf(Not(expr2)))
| And(expr1, expr2) -> And(nnf expr1, nnf expr2)
| Or(expr1, expr2) -> Or(nnf expr1, nnf expr2)
| Not(Not(expr)) -> nnf expr;;
         Characters 14-254:
  ..............function
  | (Atom(_) | Not(Atom(_))) as expr -> expr
  | Not(And(expr1, expr2)) -> Or(nnf(Not(expr1)),nnf(Not(expr2)))
  | And(expr1, expr2) -> And(nnf expr1, nnf expr2)
  | Or(expr1, expr2) -> Or(nnf expr1, nnf expr2)
  | Not(Not(expr)) -> nnf expr..
```

```
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
Not (Or (_, _))
val non_exhaustive_nnf : expr -> expr = <fun>
```

(Esta advertencia se puede tratar como un error con la opción `-w @8` cuando se invoca al compilador o al intérprete).

Esta característica proporciona un mayor nivel de seguridad y corrección en los programas que son aceptados por el compilador. Sin embargo, tiene otros usos y puede, por ejemplo, usarse en programación exploratoria. Es muy divertido escribir una conversión a una forma normal, comenzando con versiones básicas de la función que manejan los casos fáciles y utilizando ejemplos de casos no coincidentes proporcionados por el compilador para refinar el tratamiento.

Campos de registro coincidentes

La coincidencia de patrones se puede utilizar para deconstruir registros. Ilustramos esto con un tipo de registro que representa ubicaciones en un archivo de texto, *por ejemplo* , el código fuente de un programa.

```
type location = {
  filename : string;
  line: int;
  column: int;
  offset: int;
}
```

Un valor `x` de tipo de ubicación se puede deconstruir así:

```
let { filename; line; column; offset; } = x
```

Se puede utilizar una sintaxis similar para definir funciones, por ejemplo, una función para imprimir ubicaciones:

```
let print_location { filename; line; column; offset; } =
  Printf.printf "%s: %d: %d" filename line column
```

o alternativamente

```
let print_location = function { filename; line; column; offset; } ->
  Printf.printf "%s: %d: %d" filename line column
```

Los patrones que coinciden con los registros no necesitan mencionar todos los campos de un registro. Como la función no utiliza el campo de `offset` , podemos omitirla:

```
let print_location { filename; line; column; } =
  Printf.printf "%s: %d: %d" filename line column
```

Cuando el registro se define en un módulo, es suficiente para calificar el primer campo que se

produce en el patrón:

```
module Location =
struct
  type t = {
      filename : string;
      line: int;
      column: int;
      offset: int;
    }
end

let print_location { Location.filename; line; column; } =
  Printf.printf "%s: %d: %d" filename line column
```

Procesamiento de listas recursivas con coincidencia de patrones

Aquí mostramos cómo procesar listas recursivamente utilizando la sintaxis de coincidencia de patrones de OCaml.

```
let rec map f lst =
  match lst with
  | [] -> []
  | hd::tl -> (f hd)::(map f tl)
```

En este caso, el patrón `[]` coincide con la lista vacía, mientras que `hd::tl` coincide con cualquier lista que tenga *al menos un* elemento, y asignará el primer elemento de la lista a `hd` y el resto de la lista (podría estar vacío) a `tl` .

Tenga en cuenta que `hd::tl` es un patrón muy general y coincidirá con cualquier lista que no esté vacía. También podemos escribir patrones que coincidan en listas con un número específico de elementos:

```
(* Return the last element of a list. Fails if the list is empty. *)
let rec last lst =
  match lst with
  | [] -> failwith "Empty list"
  | [x] -> x (* Equivalent to x::[], [x] matches a list with only one element *)
  | hd::tl -> last tl

(* The second to last element of a list. *)
let rec second_to_last lst =
  match lst with
  | [] -> failwith "Empty list"
  | x::[] -> failwith "Singleton list"
  | fst::snd::[] -> snd
  | hd::tl -> second_to_last tl
```

Además, OCaml admite la coincidencia de patrones en los elementos de las listas. Podemos ser más específicos sobre la estructura de los elementos dentro de una lista, y OCaml inferirá el tipo de función correcto:

```
(* Assuming a list of tuples, return a list with first element of each tuple. *)
```

```
let rec first_elements lst =
  match lst with
  | [] -> []
  | (a, b)::tl -> a::(first_elements tl)
(* val first_elements : ('a * 'b) list -> 'a list = <fun> *)
```

Al combinar estos patrones juntos, podemos procesar cualquier lista arbitrariamente compleja.

Definiendo una función usando el patrón de coincidencia.

La `function` palabra clave se puede utilizar para iniciar la coincidencia de patrones en el último argumento de una función. Por ejemplo, podemos escribir una función llamada `sum` , que calcula la suma de una lista de enteros, de esta manera

```
let rec sum = function
  | [] -> 0
  | h::t -> h + sum t
;;

val sum : int list -> int = <fun>
```

Lea La coincidencia de patrones en línea: https://riptutorial.com/es/ocaml/topic/2656/la-coincidencia-de-patrones

Capítulo 8: Ocamlbuild

Examples

Proyecto en función de bibliotecas externas.

Si su proyecto depende de las bibliotecas externas, primero debe instalarlas con opam. Asumiendo que sus dependencias son `foo` y `bar` y que el punto de entrada principal de su proyecto es `foobar.ml` , puede construir un ejecutable de `foobar.ml` con

```
ocamlbuild -use-ocamlfind -pkgs 'foo,bar' foobar.byte
```

Advertencia: los nombres `foo` y `bar` deben ser los nombres de los paquetes ocamlfind, pueden diferir de los nombres de los paquetes opam.

En lugar de especificar los paquetes en la línea de comandos, puede crear un archivo de configuración llamado `_tags` con el siguiente contenido

```
true: package(foo), package(bar)
```

Ejemplo básico sin dependencia externa.

Si su proyecto no tiene una dependencia externa y tiene `foo.ml` como su punto de entrada principal, puede compilar una versión de bytecode con

```
ocamlbuild foo.byte
```

Para obtener un ejecutable nativo, ejecute

```
ocamlbuild foo.native
```

Lea Ocamlbuild en línea: https://riptutorial.com/es/ocaml/topic/2183/ocamlbuild

Capítulo 9: Procesamiento de listas

Examples

List.Map

`List.map` tiene la firma `('a -> 'b) -> 'a list -> 'b list` que en inglés es una función que toma una función (llamaremos a esto la función de mapeo) de un tipo (a saber, `'a`) a otro tipo (a saber, `'b`) y una lista del primer tipo. La función devuelve una lista del segundo tipo donde cada elemento es el resultado de llamar a la función de mapeo en un elemento de la primera lista.

```
List.map string_of_int [ 1; 2; 3; 4 ]
#- [ "1"; "2"; "3"; "4" ] : string list
```

Los tipos `'a` y `'b` no tienen que ser diferentes. Por ejemplo, podemos asignar números a sus cuadrados con la misma facilidad.

```
let square x = x * x in
List.map square [ 1; 2; 3; 4 ]
#- [ 1; 4; 9; 16 ] : int list
```

Datos agregados en una lista

Las funciones `List.fold_left` y `List.fold_right` son funciones de orden superior que implementan la lógica externa de la agregación de listas. La agregación de una lista, a veces también denominada reducción de una lista, significa calcular un valor derivado de la inspección secuencial de todos los elementos de esa lista.

La documentación del módulo de lista indica que

- `List.fold_left fa [b1; ...; bn]` es `f (... (f (fa b1) b2) ...) bn` .
- `List.fold_right f [a1; ...; an] b` es `f a1 (f a2 (... (f an b) ...))` . (Esta última función no es recursiva de cola.)

En lenguaje sencillo en inglés, `List.fold_left fa [b1; ...; bn]` equivale a correr a través de la lista `[b1; ...; bn]` hacer el seguimiento de un *acumulador* establecido inicialmente en `a` : cada vez que vemos un elemento de la lista, usamos `f` para actualizar el valor del acumulador, y cuando hayamos terminado, el acumulador es el valor final de nuestro cálculo. La función `List.fold_right` es similar.

Aquí hay algunos ejemplos prácticos:

Calcular la suma total de una lista de números

```
List.fold_left ( + ) 0 lst
```

Calcular el promedio de una lista de flotadores.

```
let average lst =
  let (sum, n) =
    List.fold_left (fun (sum, n) x -> (sum +. x, n + 1)) (0.0, 0) lst
  in
  sum /. (float_of_int n)
```

Reimplementar el procesamiento básico de listas

Las funciones `List.fold_left` y `List.fold_right` son tan generales que se pueden usar para implementar casi todas las demás funciones del módulo de lista:

```
let list_length lst = (* Alternative implementation to List.length *)
  List.fold_left ( + ) 0 lst

let list_filter predicate lst = (* Alternative implementation to List.filter *)
  List.fold_right (fun a b -> if predicate a then a :: b else b) lst []
```

Incluso es posible `List.iter` función `List.iter` , recuerde que `()` es el estado global del programa para interpretar este código como un ejemplo adicional de *agregación* de *lista* :

```
let list_iter f lst = (* Alternation implementation to List.iter *)
  List.fold_left (fun () b -> f b) () lst
```

Estos ejemplos pretenden ser material de aprendizaje, estas implementaciones no tienen ninguna virtud sobre las funciones correspondientes de la biblioteca estándar.

Lea Procesamiento de listas en línea: https://riptutorial.com/es/ocaml/topic/2730/procesamiento-de-listas

Capítulo 10: Recursion de cola

Introducción

Los lenguajes funcionales como OCaml dependen en gran medida de las funciones recursivas . Sin embargo, tales funciones pueden llevar a la memoria a un consumo excesivo o, cuando se manejan grandes conjuntos de datos, a apilar desbordamientos .

La recursión de la cola es una fuente importante de optimización en tales casos. Permite que un programa descarte el contexto de la persona que llama **cuando la llamada recursiva es la última de la función** .

Examples

Función de suma

A continuación se muestra una función no recursiva de cola para calcular la suma de una lista de enteros.

```
let rec sum = function
  | [] -> 0
  | h::t -> h + (sum t)
```

La última operación que realiza la función es la suma. Por lo tanto, la función no es recursiva de cola.

A continuación se muestra una versión recursiva de la cola de la misma función.

```
let sum l =
  let rec aux acc = function
    | [] -> acc
    | h::t -> aux (acc+h) t
  in
  aux 0 l
```

Aquí, la función `aux` es recursiva de cola: la última operación que realiza se llama a sí misma. Como consecuencia, la última versión de `sum` puede usarse con listas de cualquier longitud.

Lea Recursion de cola en línea: https://riptutorial.com/es/ocaml/topic/9650/recursion-de-cola

Capítulo 11: Tuberías, archivos y flujos

Examples

Leer desde Entrada estándar e Imprimir en Salida estándar

Preparamos un archivo llamado `reverser.ml` con los siguientes contenidos:

```
let acc = ref [] in
    try
        while true do
            acc := read_line () :: !acc;
        done
    with
        End_of_file -> print_string (String.concat "\n" !acc)
```

Luego compilamos nuestro programa usando el siguiente comando:

```
$ ocamlc -o reverser.byte reverser.ml
```

Lo probamos canalizando los datos a nuestro nuevo ejecutable:

```
$ cat data.txt
one
two
three
$ ./reverser.byte < data.txt
three
two
one
```

El programa `reserver.ml` está escrito en un estilo imperativo. Si bien el estilo imperativo está bien, es interesante comparar esto con la traducción funcional:

```
let maybe_read_line () =
  try Some(read_line())
  with End_of_file -> None

let rec loop acc =
  match maybe_read_line () with
  | Some(line) -> loop (line :: acc)
  | None -> List.iter print_endline acc

let () = loop []
```

Gracias a la introducción de la función `maybe_read_line` el flujo de control es mucho más simple en esta segunda versión que en la primera.

Lea Tuberías, archivos y flujos en línea: https://riptutorial.com/es/ocaml/topic/3252/tuberias--

archivos-y-flujos

Creditos

S. No	Capítulos	Contributors
1	Empezando con OCaml	chucksys, Community, incud, Jason Yeo, Pierre Chambart, Romain Calascibetta, Thomash
2	Campos de registro mutables	RichouHunter
3	Errores comunes	Eli Sadoff, RichouHunter
4	Escribe tu primer Script OCaml	Conrad.Dean, Jason Yeo, Kevin Chavez, Romain Calascibetta, Uncle Pa
5	Funciones	Conrad.Dean, fileyfood500, Hunan Rostomyan, Jason Yeo
6	Funciones de orden superior	chucksys, Hunan Rostomyan, Michael Le Barbier Grünewald
7	La coincidencia de patrones	Conrad.Dean, jayelm, Michael Le Barbier Grünewald, RichouHunter
8	Ocamlbuild	Thomash
9	Procesamiento de listas	Hunan Rostomyan, Kyle, Michael Le Barbier Grünewald
10	Recursion de cola	RichouHunter
11	Tuberías, archivos y flujos	Conrad.Dean, ivg, Marco Predari, Michael Le Barbier Grünewald