TUTORIAL BÁSICO DE MERCURY: ENFOCADO A LA PROGRAMACIÓN LÓGICA

INTRODUCCIÓN

Mercury es un lenguaje de programación lógico y funcional multipropósito que combina la programación declarativa con análisis estáticos avanzados.
Dentro de sus características se destacan:

- Basado en prolog (recomendamos el tutorial disponible aquí)
- Soporta modos
- Tiene un fuerte sistema de determinismo
- Tiene garbage collector
- Es modularizado
- Su compilador facilita el análisis estático y la optimización de código
- Variedad de lenguajes de destino

INSTALACIÓN

- Unix **(recomendado)**: Se requiere tener GNU C (gcc) y GNU make.
  - Para sistemas basados en debian, basta con seguir el proceso de instalación recomendado en este enlace: http://dl.mercurylang.org/deb/
  - Los archivos tar de las versiones de Mercury para UNIX se encuentran en la siguiente ruta: http://dl.mercurylang.org/
  - Para más información, haga click aquí

- Para trabajar el lenguaje online, visite: https://glot.io/new/mercury

COMPILACIÓN

Inicialmente se crea un archivo ***nombre.m*** (nombre es definido por el programador) dónde irá el código del programa. Para compilarlo en la consola basta con escribir: "***mmc nombre.m"***. mmc hace referencia al compilador de Mercury.

ESTRUCTURA DEL CÓDIGO

Por ejemplo “Hola Mundo!” en Mercury: El nombre del archivo es “hello.m”

```
:- module hello.
```
Modulo llamado hello, en un archivo llamado hello.m

```
:- interface.
```
Todo lo que va después de interface será visto por quienes importen el módulo que estamos escribiendo

```
:- import_module io.
```
Importa módulos que serán utilizados, pueden ser de la librería estándar de Mercury o creados por el programador

```
:- pred main(io::di, io::uo) is det.
```
Se crea un predicado en este caso determinista main de dos argumentos, uno en modo entrada y otro en modo salida

```
:- implementation.
```
Todo lo que va despues de implementación es de tipo privado y solo está visible para el módulo actual

```
main(!IO) :-
	io.write_string("Hello, World!\n", !IO).
```
Se crea una cláusula main con cabeza y cuerpo separados por el símbolo :- donde el cuerpo se compone de un solo “goal”

TIPOS

Mercury contiene los tipos clásicos int, float, char, tuple, etc. Sin embargo el usuario también puede definir nuevos tipos fácilmente utilizando la palabra reservada type.

```
:- type suit
	---> heart
	; diamond
	; spade
	; club.

:- type playing_card
	---> normal_card(
```

```
        c_suit :: suit,
        c_num :: int )
;
joker.
```

PREDICADOS

Los predicados en Mercury son similares a los procedimientos en otros lenguajes, no tienen un valor de retorno por sí solos. En cambio tienen valores de verdad de acuerdo con las entradas y salidas que reciben como argumento.

La *declaración* de predicados tiene la siguiente sintaxis:

```
:- pred nombre_predicado (arg1 :: modo, arg2 :: modo, ...)
determinismo
```

Inician con la palabra reservada pred seguido del nombre del predicado y entre paréntesis sus argumentos, que pueden ser tipos o variables seguidos cada uno del *modo* en el que serán interpretados. Luego, se escribe el *determinismo* del predicado que puede ser opcional. Si no se escribe, por defecto es determinista.

Como se mostró anteriormente en el ejemplo de la estructura de código, la definición del predicado main era:

```
:- pred main(io::di, io::uo) is det.
```

La *definición* de predicados tiene la siguiente sintaxis:

```
nombre_predicado (arg1, arg2, ... ) :- cuerpo
```

Se compone de dos partes, cabeza y cuerpo. En la cabeza se determina el nombre del predicado y los parámetros con los que va a trabajar. En el cuerpo se especifica cómo va a trabajar. Por ejemplo:

Cabeza Cuerpo

```
fibs(N, F) :-
    ( N < 2 ->
        F = 1
    ;
        fibs(N - 1, FA),
```

```
        fibs(N - 2, FB),
        F = FA + FB
).
```

En el cuerpo se calcula el fibonacci de un número N

- **Hechos**

```
nombre_predicado (arg1, arg2, ...)
```

Los hechos son predicados sin cuerpo y representan sucesos o información real. Cuando se necesitan varios argumentos en el hecho, se dice es una relación. Por ejemplo:

```
femenino(ada)
```

quiere decir que ada es de sexo femenino mientras que

```
pariente(daniel, ada)
```

representa una relación donde daniel y ada son parientes

- **Reglas**

```
pred1(args, ...) :-
    pred2(args, ...),
    pred3(args, ...), ...
    predN(args, ...).
```

Las reglas son predicados con cuerpo dónde para que la cabeza sea verdad, el cuerpo también lo debe ser. Las comas que separan los predicados del cuerpo significan AND, mientras que los punto y coma OR. Para hacer una analogía, el símbolo **:-** representa un ← (“entonces” hacia la derecha), es decir, si pred2 y pred3 hasta predN son verdaderos, entonces, pred1 es verdadero también. Por ejemplo:

```
madre(P, C) :-
    pariente(P, C),
    femenino(P).
```

Significa que para que P sea madre de C, es necesario que P sea pariente de C y P sea de sexo femenino.

MODOS

Los modos especifican la forma en que se toman los argumentos al declarar predicados. Son en total 4 tipos de modos. Se pueden escribir de dos maneras:

Implícitamente en el predicado luego del símbolo `::` :

```
:- pred main(io::di, io::uo) is det
```

O mediante la palabra reservada mode y dentro de los paréntesis el modo de cada argumento:

```
:- pred main(io, io).
:- mode main(di, uo) is det
```

La forma en la que trabajan los modos se especifica así,

in y out:

```
:- mode in == ground >> ground.
:- mode out == free >> ground.
```

Para in, el parámetro que se introduce al predicado inicia con algún valor (ground) y acaba con algún valor (ground). Para out, el argumento que ingresa al predicado viene vacío (free) y sale con algún valor (ground).

di y uo:

```
:- mode di == unique >> clobbered.
:- mode uo == free >> unique.
```

Unique, referencia un valor único. Esta io está optimizada y no consume mucha memoria. Los argumentos en modo di, son únicos al entrar al predicado y destruidos antes de salir (no pueden volver a ser usados) clobbered significa que la memoria que solía contener el valor a sido sobreescrita: ningún programa puede leer este valor. El modo uo recibe una variable vacía y la mantiene única.

Ejemplo:

```
:- type persona ---> rafael; laura; james.

:- pred mother(person, person).
:- mode mother(in, out).
:- mode mother(out, in).
```

```
mother(rafael, laura).
mother(james, laura).

Predicado con dos modos:

Quien es la madre de rafael?
mother(rafael, M)

De quien es madre laura?
mother(H, laura)
```

En el código anterior se declara primero un tipo *persona* con los nombres que se van a usar posteriormente. A continuación se declara un predicado que relaciona dos personas con una madre, luego se especifica que ambos argumentos de mother pueden ser tanto entradas como salidas (útil al hacer consultas). Después se definen dos hechos que muestran que laura es madre de rafael y laura es madre de james. Ahora para saber quien es la madre de rafael, mercury verifica el hecho de que `mother(rafael, laura).`

El programa completo es el siguiente:

```
:- module mother.

:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is det.

:- implementation.
:- type persona --->  laura; rafael; james.
:- pred mother(persona, persona).
:- mode mother(in, out).
:- mode mother(out, in).

mother(rafael, laura).
mother(james, laura).

main(!IO) :-
	(
	if mother(rafael, X)
	then write_string("La madre de rafael es ",
!IO),
	write(X, !IO),
	write_string("\n", !IO)
```

```
        else write_string("Rafael no tiene madre",
!IO)
        ).
```

Y retorna: “La madre de rafael es laura”.

DETERMINISMO

Para cada modo de un predicado hay una categoría de determinismo de acuerdo al número de soluciones que puede producir o si puede llegar a fallar sin producir ninguna solución.

Si todas las posibles llamadas a un modo de un predicado tienen:

exactamente una solución, el modo es **det**
ninguna o sólo una solución, el modo es **semidet**
tiene una o más soluciones, el modo es **multi**
cero o más soluciones, el modo es **nondet**
falla sin producir una solución, el modo tiene determinismo **failure**
Si no se retorna solución a quien lo llama, (e.g.por una excepción), el modo tiene determinismo **erroneous**

Por ejemplo:

```
:- pred pariente(persona, persona) .
:- mode pariente(in, in) is semidet.
:- mode pariente(in, out) is nondet.
:- mode pariente(out, in) is nondet.
:- mode pariente(out, out) is multi.
```

nondet es más general que las otras categorías de determinismo, entonces, ¿por qué molestarse en declarar determinismos más estrictos? Porque permiten al compilador generar código más eficiente; respaldar el rastreo tiene un costo. Si el compilador infiere un determinismo diferente de lo que se declara, a menudo es una buena señal de que hay un error en alguna parte.

Hay otras dos categorías de determinismo de "elección comprometida". **cc_multi** y **cc_nondet** son multidireccionales y no indecisos, respectivamente, pero se comprometerán de forma arbitraria con una única solución y evitarán el retroceso para encontrar otras soluciones. A veces, una solución es tan buena como cualquier otra.

EJEMPLOS:

1. Factorial de un número

```
:- module factorial.
:- interface.

:- import_module io.

:- pred main(io::di, io::uo) is det.

:- implementation.

:- import_module int.

:- func factorial(int) = int.

factorial(N) =
    ( if N = 0 then
        1
    else
        N * factorial(N - 1)
    ).

main(!IO) :-
    F = factorial(5),
    write_int(F, !IO),
    nl(!IO).
```

SALIDA: “120”

2. Lógica de los marcianos

```
/*
PROBLEMA:

ngtrks es pequeño y verde
pgvdrk es un marciano saltarin
todas la criaturas saltarinas son verdes
todas las criaturas pequeñas saltarinas son marcianos
todas las criaturas verdes y marcianas son inteligentes.

¿Cual de los dos es inteligente?
```

*/

```
:- module martians.
:- interface.
:- import_module io.
:- pred main(io::di, io::uo) is cc_multi.
:- implementation.
:- type martian ---> ngtrks; pgvdrk.
:- pred small(martian::out) is det.
:- pred green(martian::out) is multi.
:- pred martian(martian::out) is multi.
:- pred jumping(martian::out) is det.
:- pred intelligent(martian::out) is nondet.

small(ngtrks).
green(ngtrks).
martian(pgvdrk).
jumping(pgvdrk).

green(X) :- jumping(X).
martian(X) :- small(X), jumping(X).

intelligent(X) :- green(X), martian(X).

main(!IO) :-
      io.write_string("Que marciano es inteligente?: ", !IO),
      (
      if intelligent(X)
      then io.write(X, !IO),
           io.write_string(" es inteligente\n", !IO)
      else io.write_string(" no se puede determinar\n", !IO)
      ).
```

SALIDA: Que marciano es inteligente?: pgvdrk es inteligente

3. Árbol genealógico

```
:- module family.
:- interface.

:- import_module io.
```

```
:- pred main(io::di, io::uo) is det.

:- implementation.

:- import_module list.
:- import_module solutions.

:- type person
    --->    ada
    ;       bob
    ;       dan
    ;       ema
    ;       fay
    ;       joe.

:- pred female(person).
:- mode female(in) is semidet.
:- mode female(out) is multi.

female(ada).
female(ema).
female(fay).

:- pred male(person).
:- mode male(in) is semidet.
:- mode male(out) is multi.

male(bob).
male(dan).
male(joe).

:- pred parent(person, person).
:- mode parent(in, in) is semidet.
:- mode parent(in, out) is nondet.
:- mode parent(out, in) is nondet.
:- mode parent(out, out) is multi.

parent(ada, dan).
parent(bob, dan).
parent(dan, fay).
parent(ema, fay).
```

```
parent(dan, joe).
parent(ema, joe).

:- pred father(person, person).
:- mode father(in, in) is semidet.
:- mode father(in, out) is nondet.
:- mode father(out, in) is nondet.
:- mode father(out, out) is nondet.

father(P, C) :-
    parent(P, C),
    male(P).

:- pred mother(person, person).
:- mode mother(in, in) is semidet.
:- mode mother(in, out) is nondet.
:- mode mother(out, in) is nondet.
:- mode mother(out, out) is nondet.

mother(P, C) :-
    parent(P, C),
    female(P).

:- pred grandparent(person, person).
:- mode grandparent(in, in) is semidet.
:- mode grandparent(in, out) is nondet.
:- mode grandparent(out, in) is nondet.
:- mode grandparent(out, out) is nondet.

grandparent(PP, C) :-
    parent(PP, P),
    parent(P, C).

main(!IO) :-
    % The first argument to solutions is a closure.
    % solutions/2 returns all the solutions produced by the closure
    % as a list.
    solutions(
        (pred(PP::out) is nondet :-
            grandparent(PP, fay)
        ),
        PPs),
```

```
    (
        PPs = [],
        write_string("fay has no known grandparents\n", !IO)
    ;
        PPs = [_ | _],
        write_string("fay has these known grandparents: ", !IO),
        write(PPs, !IO),
        nl(!IO)
    ).
```

SALIDA: fay has these known grandparents: [ada, bob]