Las estructuras de flujo de control de flujo son case, cond e if.

Case

Case nos permite comparar un valor con muchos patrones hasta que encontremos uno coincidente:

```
iex> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} ->
...>     "This clause will match and bind x to 2 in this clause"
...>   _ ->
...>     "This clause would match any value"
...> end
"This clause will match and bind x to 2 in this clause"
```

Si desea comparar patrones con una variable existente, debe usar el operador ^:

```
iex> x = 1
1
iex> case 10 do
...>   ^x -> "Won't match"
...>   _ -> "Will match"
...> end
"Will match"
```

Las cláusulas también permiten que se especifiquen condiciones adicionales a través de

guards:

```
iex> case {1, 2, 3} do
...>   {1, x, 3} when x > 0 ->
...>     "Will match"
...>   _ ->
...>     "Would match, if guard condition were not satisfied"
...> end
"Will match"
```

La primera cláusula anterior solo coincidirá cuando x sea positivo.

Tenga en cuenta que los errores en los guardias no tienen fugas, sino que simplemente hacen que el guardia falle:

```
iex> hd(1)
** (ArgumentError) argument error
iex> case 1 do
...>   x when hd(x) -> "Won't match"
...>   x -> "Got #{x}"
...> end
"Got 1"
```

Si ninguna de las cláusulas coincide, se genera un error:

```
iex> case :ok do
...>   :error -> "Won't match"
...> end
```

```
** (CaseClauseError) no case clause matching: :ok
```

Consulte la documentación completa de los guardias para obtener más información sobre los guardias, cómo se usan y qué expresiones están permitidas en ellos.

Tenga en cuenta que las funciones anónimas también pueden tener múltiples cláusulas y guardias:

```
iex> f = fn
...>   x, y when x > 0 -> x + y
...>   x, y -> x * y
...> end
#Function<12.71889879/2 in :erl_eval.expr/5>
iex> f.(1, 3)
4
iex> f.(-1, 3)
-3
```

El número de argumentos en cada cláusula de función anónima debe ser el mismo, de lo contrario se genera un error.

```
iex> f2 = fn
...>   x, y when x > 0 -> x + y
...>   x, y, z -> x * y + z
...> end
** (CompileError) iex:1: cannot mix clauses with different arities
in anonymous functions
```

Cond

Case es útil cuando necesita hacer coincidir valores diferentes. Sin embargo, en muchas circunstancias, queremos verificar diferentes condiciones y encontrar la primera que no se evalúa como nula o falsa. En tales casos, uno puede usar cond:

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This will not be true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   1 + 1 == 2 ->
...>     "But this will"
...> end
"But this will"
```

Esto es equivalente a las cláusulas else if en muchos idiomas imperativos (aunque aquí se usan con menos frecuencia).

Si todas las condiciones devuelven nulo o falso, se genera un error (CondClauseError). Por esta razón, puede ser necesario agregar una condición final, igual a verdadera, que siempre coincidirá con:

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This is never true"
...>   2 * 2 == 3 ->
...>     "Nor this"
```

```
...>   true ->
...>     "This is always true (equivalent to else)"
...> end
"This is always true (equivalent to else)"
```

Finalmente, note cond considera que cualquier valor además de nil y false es verdadero:

```
iex> cond do
...>   hd([1, 2, 3]) ->
...>     "1 is considered as true"
...> end
"1 is considered as true"
```

if y unless

Además de case y cond, también tenemos las macros if/2 y unless/2, que son útiles cuando necesita verificar una sola condición:

```
iex> if true do
...>   "This works!"
...> end
"This works!"
iex> unless true do
...>   "This will never be seen"
...> end
nil
```

Si la condición if/2 devuelve false o nil, el cuerpo dado entre do/end no se ejecuta y en su lugar devuelve nil. Lo contrario sucede con unless/2.

También admiten bloques más:

```
iex> if nil do
...>   "This won't be seen"
...> else
...>   "This will"
...> end
"This will"
```

Bloques do/end

Como hemos visto en las cuatro estructuras de control: case, cond, if y unless. En todas estaban envueltas en bloques do/end. Sucede que también podríamos escribir si es así:

```
iex> if true, do: 1 + 2
3
```

Los bloques do/end son una conveniencia sintáctica construida sobre las palabras clave. Es por eso que los bloques do/end no requieren una coma entre el argumento anterior y el bloque. Son útiles exactamente porque eliminan la verbosidad al escribir bloques de código. Estos son equivalentes:

```
iex> if true do
...>   a = 1 + 2
```

```
...>   a + 10
...> end
13
iex> if true, do: (
...>   a = 1 + 2
...>   a + 10
...> )
13
```

Las listas de palabras clave juegan un papel importante en el lenguaje y son bastante comunes en muchas funciones y macros.