Erlang

Tipi di dato

Come in tutti i linguaggi di programmazione, i dati si dividono in **predefiniti** o **definiti dall'utente**, ed **atomici** o **non atomici**.

In generale, ogni linguaggio di programmazione mette a disposizione alcuni tipi di dato predefiniti e, a seconda del linguaggio, offre la possibilità di definirne di nuovi.

I nuovi tipi possono essere semplici alias per tipi esistenti, oppure strutture che estendono tipi già presenti (non più semplici alias in questo caso).

Esistono anche tipi di dato più complessi, come i tipi algebrici, che descrivono le possibili forme che un dato può assumere.

Essendo Erlang un linguaggio dinamicamente tipato, non esistono dichiarazioni esplicite di nuovi tipi di dato. Non è presente un costrutto sintattico dedicato alla definizione di tipi. L'utente crea nuove tipologie di dato semplicemente utilizzando i valori in modo coerente.

Un esempio sono i valori booleani, che in Erlang sono rappresentati dagli atomi `true` e `false`.

Per quanto riguarda la distinzione tra dati atomici e non atomici, i tipi atomici sono quelli che non contengono altri dati al loro interno.

Un esempio di dato atomico è un numero, mentre un esempio di dato non atomico è una lista, che contiene al suo interno altri elementi.

Tra i dati **atomici**, in Erlang troviamo:

- **Numeri interi**, sui quali è possibile eseguire le comuni operazioni matematiche.
  È importante notare che gli operatori di confronto hanno alcune particolarità sintattiche: mentre l'operatore maggiore o uguale mantiene la forma standard (`>=`), l'operatore minore o uguale diventa `=<` per distinguerlo dalla forma di una freccia (essendo Erlang simile al Prolog, le frecce hanno un significato particolare).
  Altri operatori di confronto importanti sono l'uguaglianza stretta (`=:=`) e la disuguaglianza stretta (`=/=`).
- **Numeri in virgola mobile** (floating point), che quando combinati con numeri interi provocano la conversione implicita del risultato in floating point. È importante notare che il confronto tra un numero intero e uno floating point con `=:=` restituisce `false` (ad esempio, `5.0 =:= 5` è `false`), poiché rappresentano sequenze di bit differenti. Per un test di uguaglianza meno rigoroso, che considera equivalenti valori numericamente uguali indipendentemente dal tipo, si possono usare gli operatori `==` o `/=`.
- **PID** (Process IDentifier), ottenibili chiamando la funzione `self()`, che identificano univocamente i processi.
- **Reference**, ottenibili chiamando la funzione `make_ref()`. Una reference è un valore probabilisticamente unico, progettato per essere diverso da tutte le reference generate in precedenza. Non dovrebbe esistere un algoritmo in grado di prevedere la prossima reference che verrà generata.

- **Porte**. Nel modello ad attori di Erlang, quando è necessario interagire con entità esterne che non sono attori, è possibile avvolgerle in una specie di attore intermediario che permette di comunicare con esse utilizzando i meccanismi di invio e ricezione di messaggi tipici del linguaggio.
  Questi attori speciali, che fanno da wrapper a entità esterne, non possiedono tutte le caratteristiche degli attori normali. Ad esempio, non seguono il principio "Let it fail" di Erlang, che normalmente termina tutti gli attori associati a un attore che fallisce.
  Quando viene creato questo tipo di attore speciale, gli viene assegnata una porta anziché un PID.
- **Atomi**, che si scrivono normalmente con lettere minuscole. L'idea è che un programma utilizzi un numero limitato di atomi, che verranno rappresentati in memoria come sequenze di bit efficienti.
  È possibile racchiudere un atomo contenente spazi tra apici singoli (ad esempio, `'hello world'`). Da notare che `'ciao' =:= ciao` restituisce `true`, poiché sono considerati lo stesso atomo.

Passando ai dati **non atomici**, Erlang offre:

- **Tuple**. Si definiscono tra parentesi graffe, con elementi separati da virgole. Un esempio di tupla è `{4, {ciao, 2.0}, true}`. Esiste anche la tupla vuota `{}`, utilizzabile quando non si desidera restituire alcun valore significativo.
- **Liste**. Una lista può essere vuota (`[]`), oppure ha una *testa* (primo elemento) e una *coda* (una lista contenente tutti gli altri elementi).
  La testa può essere un valore qualsiasi, mentre la coda è a sua volta una lista.
  Un esempio di lista può essere scritto come `[ 2 | [ 3 | [ 4 | [ ] ] ] ]`, che rappresenta la struttura fondamentale. Per comodità è possibile utilizzare la sintassi semplificata `[2, 3, 4]`, ma concettualmente la lista è sempre composta da una testa e una coda.

  Essendo Erlang un linguaggio dinamicamente tipato, non ci sono garanzie che la coda sia effettivamente una lista. Quando la coda non è una lista, si parla di *lista impropria*, sulla quale non è possibile applicare le normali operazioni previste per le liste.

  Le operazioni predefinite sulle liste includono il calcolo della lunghezza, la concatenazione (`[2, 3] ++ [4, 5]` restituisce `[2, 3, 4, 5]`) e la sottrazione (`[2, 3, 4, 5] - [4, 2]` restituisce `[3, 5]`). La sottrazione segue una logica insiemistica, quindi in casi come `[2, 3, 4, 5] - [4, 2] - [4]`, il risultato sarà `[3, 4, 5]` e non `[3, 5]`.

  Un'altra potente caratteristica delle liste è la **list comprehension**. Un esempio:
  `[ {X, Y + 1} || X <- [1, 2, 3], {Y, _} <- [{4, 5}, {6, 7}] ].`
  Questa espressione restituisce:
  `[ {1, 5}, {1, 7}, {2, 5}, {2, 7}, {3, 5}, {3, 7} ].`

  Concettualmente, è come se ci fossero dei cicli for annidati che estraggono valori per X e Y. È anche possibile aggiungere filtri, ad esempio:
  `[ {X, Y + 1} || X <- [1, 2, 3], {Y, _} <- [{4, 5}, {6, 7}], X + Y < 6 ].`
  Questa espressione restituisce solamente `[ {1, 5} ]`, poiché solo la coppia $\{1, 4\}$ soddisfa la condizione $X + Y < 6$.

- **Bit strings**. Erlang offre la possibilità di accedere alla rappresentazione binaria di qualsiasi dato, permettendo di manipolare e analizzare sequenze di bit tramite pattern matching.
  Un esempio: `N = 16#7A5.` definisce un numero in base 16.
  Per accedere alla sua rappresentazione in bit, possiamo usare la sintassi:
  `« R:4, G:4, B:4 » = « N:12 ».`
  A questo punto, accedendo a R, G o B otterremo le rispettive sequenze di bit (nell'ordine: 7, 10 e 5).

  Questa funzionalità è particolarmente utile quando si lavora con pacchetti di rete, file binari o per interagire con dispositivi a basso livello, consentendo un controllo preciso sulle sequenze di bit.

Rimangono infine le **funzioni**, note anche come **chiusure**. Una caratteristica fondamentale dei linguaggi funzionali è che le funzioni sono oggetti di prima classe, manipolabili come qualsiasi altro valore. Una funzione può essere passata come argomento, restituita come risultato, inserita in strutture dati e così via.

In Erlang esistono diverse sintassi per definire funzioni. La prima forma ha la struttura: `nome_funzione(lista_argomenti) -> corpo ... end.` Questa sintassi può essere utilizzata nei file da compilare, ma non direttamente nella shell interattiva.

Una sintassi utilizzabile ovunque impiega la parola chiave **fun**, ad esempio:
`fun (lista_argomenti) -> ... end.` Questa è la sintassi per creare una funzione anonima.

È possibile definire funzioni annidate all'interno di altre funzioni. Le funzioni interne hanno accesso alle variabili definite nello scope più esterno (chiusura lessicale).

Un esempio: `G = fun (X) -> fun (Y) -> X + Y end end.`

Eseguendo `H = G(2).` e poi `H(3).`, otterremo il valore `5`. La variabile X, con valore 2, è stata "catturata" nella chiusura restituita da G.

Per dichiarare una funzione ricorsiva, si può usare la sintassi: `fun G(N) -> N * G(N) end.`
Il nome G è visibile solo all'interno della funzione stessa e non può essere richiamato dall'esterno.

In generale, le funzioni in Erlang utilizzano il pattern matching per selezionare diverse implementazioni in base all'input ricevuto, come accade anche con il costrutto *receive* per la gestione dei messaggi.

Tutti i linguaggi funzionali moderni permettono di definire funzioni per **casi**, consentendo di scrivere algoritmi in modo conciso e comprensibile, riducendo significativamente la quantità di codice.

Un esempio di funzione definita per casi può essere:
`fun ({N, 2}) -> N; ({ciao, N, M}) -> N + M; ([_, _, {X, Y}]) -> X + Y end.`
Qui il simbolo `_` indica un pattern che corrisponde a qualsiasi valore, il quale viene ignorato (non gli viene assegnato un nome).

Questa è una funzione definita tramite *pattern matching*. In base all'input fornito, verrà eseguito il ramo corrispondente al pattern che corrisponde. Se viene fornito un input che non corrisponde a nessuno dei pattern definiti, verrà sollevata un'eccezione.

È inoltre possibile utilizzare delle **guardie** per aggiungere condizioni aggiuntive. Dopo il pattern, attraverso la parola chiave `when`, si possono specificare condizioni che devono essere soddisfatte. Ad esempio:
`fun ({N, 2}) when N > 2 -> N; ({ciao, N, M}) -> N + M; ([_, _, {X, Y}]) -> X + Y end.`

Quando più pattern possono corrispondere all'input, l'ordine di valutazione è **sequenziale** dall'alto verso il basso.

Un aspetto importante delle guardie in Erlang è che il linguaggio si assicura che la loro valutazione non produca effetti collaterali, come l'invio di messaggi o la creazione di nuovi processi. Molti linguaggi moderni non effettuano questo controllo.

In Erlang, le guardie possono contenere solo combinazioni di funzioni predefinite chiamate **BIF** (Built-In Functions).
Questa restrizione rende il linguaggio delle guardie meno espressivo, il che può diventare problematico in casi complessi, come quando si desidera impedire l'attivazione di uno specifico caso in base a condizioni elaborate.