Erlang – wprowadzenie

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

Erlang

- Erlang – **Er**icsson **Lang**uage / nawiązanie do A. K. Erlanga.
- Zaprojektowany z myślą o zastosowaniach współbieżnych i tworzeniu rozproszonych systemów.
- Umożliwia aktualizacje kodu bez zatrzymywania aplikacji.
- Dynamicznie typowany (inaczej, niż w Haskellu).

(źródło: Wikipedia:Erlang (język programowania)).

Przykładowy program

```
-module (prog1).
-export ([binom/2]).
% Komentarz: funkcja obliczajaca silnie
silnia(0) -> 1;
silnia (N) -> N*silnia (N-1).
binom(N,K)-> silnia(N)/(silnia(K)*silnia(N-K)).
```

Przykładowy program

- Nazwa modułu (`module`) musi pokrywać się z nazwą pliku, w powyższym przykładzie plik powinien nazywać się `prog1.erl`.
- W pliku możemy definiować wiele funkcji (w tym pomocnicze), na zewnątrz są dostępne jednak jedynie funkcje, które wyeksportujemy (`export`).
- W odróżnieniu od Haskella, wszystkie argumenty funkcji mają nazwę z wielkiej litery.
- Wszystkie definicje są zakończone znakiem kropki.
- W powyższym programie ma miejsce dopasowanie do wzorca. Wzorce oddzielane są średnikami.

Kompilacja i korzystanie z funkcji

Po zapisaniu programu w pliku schemat interakcji z nim wygląda następująco:

```
erl
1>c(prog1).
{ok,prog1}
2>prog1:binom(20,4).
4845.0
3>prog1:binom(20,10).
184756.0
```

Również tutaj każde polecenie musimy zakończyć kropką.

erl

- `c(nazwa_modułu).` – kompilacja.
- `nazwa_modułu:funkcja(...).` – wywołanie funkcji eksportowanej w module.
- `pwd().` – wyświetlenie katalogu, w którym się znajdujemy.
- `cd(...).` – zmiana katalogu, w którym się znajdujemy.
- `q().` – zakończenie działania.

Każde polecenie musimy zakończyć kropką.

Znaki interpunkcyjne

- Przecinek – oddziela linijki kodu w ramach bloku. Możemy np. wprowadzić pomocnicze oznaczenie (jak `let` w Haskellu) i użyć go po przecinku.
- Średnik – oddziela bloki kodu.
- Kropka – kończy definicje.

Atomy

Atom to stała, której nadaliśmy nazwę. Jeśli nazwa ta nie zaczyna się od małej litery lub zawiera inne znaki niż alfanumeryczne, podkreślenie (_) i @, to musimy ją zamknąć w znakach '. Przykładowo

```
-module (zeros).
-export ([isZero/1]).

isZero (0) -> yes;
isZero (_) -> no.
```

eksportuje funkcję `isZero/1` zwracającą atom `yes`, jeśli podany argument jest zerem oraz atom `no` w przeciwnym przypadku.

Krotki

Krotki zapisujemy w nawiasach klamrowych (w Haskellu były to nawiasy okrągłe). Przykładowo:

- `{1,2}` – para liczb.
- `{"Ala",ma,4,'KOTY'}` - czwórka złożona kolejno z łańcucha znaków, atomu, liczby całkowitej i atomu.

"Zmienne"

W Erlangu "zmienne" po zainicjowaniu nie mogą zostać później zmienione. Nazywamy je zawsze z dużej litery.

Listy

- Lista w Erlangu nie musi zawierać tylko elementów jednego typu (jak w Haskellu).
- Listy oznaczamy nawiasami kwadratowymi, przykładowo `L = ["Ala",ma,4,'KOTY']`.
- Dopasowanie do wzorca dla list (na przykładzie powyższego `L`) wygląda następująco:
  - `[A|Reszta]` dopasuje `A = "Ala"` oraz `Reszta=[ma,4,'KOTY']`.
  - `[A,B|Reszta]` dopasuje `A = "Ala"`, `B = ma` oraz `Reszta=[4,'KOTY']`.
- Operator konkatenacji list jest taki sam, jak w Haskellu, ++.

Przykład – iloczyn listy

Poniższy program liczy iloczyn liczb w liście:

```
-module (list).
-export ([product/1]).

product ([]) -> 1;
product ([A|Reszta]) -> A*product(Reszta).
```

Dozory

Dozory (guards) realizujemy za pomocą słowa kluczowego `when`. Przykładowo:

```
-module (silnia).
-export ([fact/1]).

fact (0)->1;
fact (N) when N>0-> N*fact (N-1).
```

Co ważne, w dozorach (w tym w instrukcji `if`) nie możemy używać zdefiniowanych przez siebie / zaimportowanych funkcji. Możemy wywołać pomocniczą funkcję tuż przed, przypisać wynik do zmiennej pomocniczej i to jej użyć w dozorze/ifie, ale nie możemy zrobić tego bezpośrednio, kompilator na to nie pozwala. O wyrażeniach, które
mogą być przydatne w dozorach można przeczytać na przykład tutaj: https://www.erlang.org/doc/reference_manual/expressions.html#guard-expressions

if

Konstrukcja `if` wygląda następująco:

```
if warunek1->
 kod1;
  warunek2->
 kod2;
 ...
  warunekN->
 kodN
end
```

if-else

Aby otrzymać "standardową" instrukcję typu if-else możemy użyć warunku `true->`:

```
if warunek->
 kod1;
  true->
 kod2
end
```

W szczególności, w Haskellu `otherwise` zdefiniowane jest właśnie jako stale równe prawdzie.

lambda-wyrażenia

Lambda wyrażenia w Erlangu realizowane są za pomocą słowa kluczowego `fun`. Zdefiniowanym tak funkcjom można też przypisywać nazwy:

```
KwadratPlusJeden=fun (X)->X*X+1 end.
```

Bardziej złożone operacje na listach

W Erlangu dostępne są też funkcje `lists:map` czy `lists:filter` o działaniu analogicznym to tych znanych z Haskella. Można je znaleźć na przykład tutaj:
https://www.erlang.org/doc/man/lists.html
Możliwe jest również tworzenie list elementów "takich, że" podobnie, jak w Haskellu. Przykładowo listę kwadratów liczb od 1 do 10 można realizować następująco:

```
[X * X || X <- lists:seq(1, 10)]
```

Erlang – procesy

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

Tworzenie procesów

Do tworzenia procesów w Erlangu służy funkcja `spawn/3` o składni

```
spawn(Nazwa_modulu, Eksportowana_funkcja,
  Lista_argumentow).
```

tworząca proces, który wywołuje funkcję `Eksportowana_funkcja` z argumentami `Lista_argumentow`.

Przykładowy program

```
-module (proces1).
-export ([start/0,hello/2]).

hello(Msg,1) -> io:format("~p~n",[Msg]);
hello(Msg,N) -> io:format("~p~n",[Msg]),
                hello(Msg,N-1).

start() -> PID1=spawn(proces1,hello,["Proces 1",5]),
           PID2=spawn(proces1,hello,["Proces 2",5]),
           io:format(
           "PID (Proces 1) = ~p,~nPID (Proces 2) = ~p~n",
           [PID1,PID2]).
```

Przykładowy program – opis

Powyższy program tworzy dwa procesy, każdy z nich wypisuje "Proces " oraz, odpowiednio, numer 1 lub 2, pięć razy. Program wypisuje też na ekran identyfikatory tych procesów. Co ważne, mimo, że po skompilowaniu chcemy użyć tylko funkcji `start`, która rozpoczyna procesy, to żeby program dział poprawnie, **musimy eksportować wszystkie funkcje używane do tworzenia procesów (które podajemy w drugim argumencie `spawn`)**.
Do wypisywania na ekran służy funkcja `io:format`, kolejne wystąpienia "~p" zostaną zastąpione kolejnymi elementami listy, "~n" oznacza znak nowej linii.

Komunikacja między procesami

Do komunikacji między procesami służy konstrukcja

```
PID ! komunikat
```

gdzie `PID` jest PID procesu, do którego chcemy wysłać komunikat, a `komunikat` jest dowolnym termem Erlanga (liczbą, listą, łańcuchem znakowym, atomem itd.).

Komunikacja między procesami

Do odbierania komunikatów wysyłanych w powyższy sposób służy konstrukcja

```
receive
  wzorzec1 ->
    akcja1;
  wzorzec2 ->
    akcja2;
  ...
  wzorzecN ->
    akcjaN
end .
```

receive

Każdy proces przechowuje swoją kolejkę komunikatów, które otrzymuje (nowe są dodawane na koniec). W momencie napotkania `receive` w procesie, próbuje on dopasować pierwszy element kolejki do podanych tam wzorców. Jeśli nastąpiło jakiekolwiek poprawne dopasowanie, wykonywane są akcje dla tego wzorca; jeśli nie, proces bierze kolejny element kolejki i znów próbuje go dopasować; jeśli w wyniku braków dopasowania kolejka się skończy, proces jest blokowany i czeka na dalszy komunikat, aż nie uda mu się jakiegoś dopasować.
Źródło:
`https://erlang.org/doc/getting_started/conc_prog.html`

receive

Jeśli chcemy zaimplementować "typowe" działanie serwera, czyli oczekujemy na kolejne komunikaty na które reagujemy w określony sposób, a następnie czekamy na kolejne, musimy sami o to zadbać w kodzie, który wywołuje się pod danym warunkiem lub już po całym bloku `receive` (inaczej program wykona kod w odpowiednim przypadku `receive` i "pójdzie dalej"). Samo `receive` przypomina więc działaniem pojedynczą instrukcję `if`, ma też wbudowane dopasowanie do wzorca (np. warunek może wyglądać `[X|Xs] ->`, dopasuje się on do dowolnej listy długości dodatniej i w kodzie w tym przypadku możemy używać `X` jako głowy tej listy oraz `Xs` jako pozostałych elementów). Możliwe też jest stosowanie dozorów (`when`).

Przykładowy klient-serwer

```
-module (cs).
-export ([server/0,start/0,client/1]).
recvMsg() -> receive
               %% tutaj obsluga przychodzacych komunikatow,
               %% czyli kod serwera.
             end,
             recvMsg().
server() -> io:format("Server started.~n",[]),
            recvMsg().
client(SPID) -> SPID ! msg
                %% przykladowe dzialanie klienta, w tym
                %% wypadku wyslanie atomu msg do serwera
start() -> Server_PID = spawn(cs,server,[]),
           spawn (cs,client,[Server_PID]).
```

after

Przed zakończeniem klauzuli `receive` możemy dodać klauzulę

```
after Czas_w_milisekundach -> <kod, co robimy>
```

wywołującą się w przypadku, gdy program nie otrzyma przez zadany czas żadnego komunikatu. Jeśli nie chcemy robić nic, tylko przejść dalej, i tak musimy coś napisać (ale może to być trywialne stwierdzenie, na przykład 1=1).

register

W przypadku powyżej najpierw tworzymy serwer, zapisujemy jego PID do zmiennej oraz klienta tworzymy przesyłając mu PID serwera jako argument. Istnieje też inne rozwiązanie tego problemu, za pomocą funkcji

```
register(atom,PID).
```

możemy przypisać do atomu PID procesu (na przykład przy okazji wywoływania `spawn`) i to z niego korzystać w kodzie klienta (czyli wywoływać `atom ! komunikat`). Musimy jednak zadbać o to, żeby przypisanie to nastąpiło przed wywołaniem takiej instrukcji przesłania komunikatu, inaczej dostaniemy błąd. Jeśli chcemy też np. wypisać PID na ekran, jeśli skorzystamy z atomu, zostanie on po prostu wypisany na ekran (więc możemy chcieć w takim przypadku również przypisać PID do jakiejś stałej).