Az Erlang programozási nyelv

Lövei László
lovei@elte.hu

Irodalom

- `http://www.erlang.org` - Dokumentáció Erlang/OTP Rxxx dokumentáció
  - Getting Started
  - Reference Manual
  - Basic Applications – kernel
  - Basic Applications – stdlib
  - Working with OTP – Design Principles
- Online manual: `erl -man erlang`, `erl -man lists`, `...`

Termek, változók

- *Term:* valamilyen adat a programban.
- Minden termnek van típusa.
- *Változó:* nagybetűvel kezdődő alfanumerikus azonosító. Lehet kötött vagy szabad.
- A szabad változóknak nincs típusa (gyengén típusos a nyelv).
- A kötött változóknak egy konkrét term az értékük.

Számok

- Egészek: `10, -33, 0`
- Nem decimális alak: `alap#szám`, pl. `16#1AF`
- Lebegőpontos számok: `3.141, 6.0e-23`
- Karakterek ASCII kódja: `$a, $\n`

Atomok

- Szöveges konstansok, de nem stringek.
- Kisbetűvel kezdődő, alfanumerikus atomok önmagukat jelölik: `alma, egy_szo`
- Nem kisbetűvel kezdődő, vagy nem alfanumerikus karaktert is tartalmazó atomot aposztrófok közé kell zárni: `'Alma', 'ket szo'`
- Egy atom tetszőleges hosszú lehet, és tetszőleges karaktert tartalmazhat (pl. `'ket\nsor'`).

Rendezett n-esek

- Fix számú, tetszőleges termet tartalmazhat, akár további *n*-eseket is.
- Szintaxis: **{**`123, alma`**}**
- Null tuple: **{}**
- Az egy elemű *n*-es nem egyezik meg magával az elemmel

Listák

- A lista is tetszőleges termet tartalmazhat, akár különböző típusúakat is. A mérete nem fix.
- Konstrukció fejből és farokból: **[]**, **[**`a`**|[]]**, **[**`a`**|[**`1`**|[]]]**, **[**`a`**|[**`1`**|[{**`x,y`**}|[]]]]**
- Rövidítés: **[**`a,1,{x,y}`**]**
- A kettőt keverni is lehet: **[**`a,1`**|[**`b,2`**]]**
- A konkatenációnak van önálló jelölése is: `[a,1]`**++**`[b,2]`

Stringek

- Önálló string típus nincs, a karakterek kódjainak listája ábrázolja
- Rövidítés: `"alma" = [$a,$l,$m,$a]`
- A C nyelvhez hasolóan az egymás mellett álló string literálokat összevonja a fordító: `"alma" "korte" = "almakorte"`

Mintaillesztés

- A minták szintaktikailag ugyanolyanok, mint a termek, de bármelyik részterm helyén állhat szabad változó.
- Ha az illesztés sikeres, a szabad változók kötötté válnak, a megfelelő részterm lesz az értékük. Egy kötött változóhoz nem lehet más értéket kötni!
- Mintaillesztő operátor:

```
A = 2
[H|T] = [1,2,3]
{X,Y} = {alma,korte}
```


Kifejezések

- A termek, a változók és a minták kifejezések.
- Mintaillesztő kifejezést készít az = operátor.
- Függvényhívás:
  - `fv(arg1, arg2, ..., argn)`
  - `modul:fv(arg1, ..., argn)`
- Blokk kifejezés:
  `begin` `kif1, kif2, ... kifn` **`end`**
  - Az eredmény az utolsó kifejezés eredménye.

Beépített függvények

- Beépített függvények (BIF-ek) azok a függvények, amiket a futtató környezet implementál.
- A típusok alapműveletei általában ilyenek:
  - Számok: `abs(Num), trunc(Num), round(Num)`
  - Lista: `length(List),hd(List),tl(List)`
  - Tuple: `size(Tuple), element(Ind,Tuple)` `setelement(Ind, Tuple, Value)`
  - Egyéb: `date(), time(), exit(Reason)`

Típusok vizsgálata, konverziója

- Típusvizsgálat: `is_integer, is_float, is_number, is_atom, is_tuple, is_list`
- Konverzió: általában az adatok string alakját lehet előállítani:
  `integer_to_list, list_to_integer`
  `atom_to_list, list_to_atom`
  `tuple_to_list, list_to_tuple`

Aritmetikai kifejezések

- Matematikai műveletek:
  - `+, -, *, /`
  - `div, rem`
- Bitaritmetika:
  - `bnot, band, bor, bxor`
  - eltolás: `bsl, bsr`

Logikai kifejezések

- Összehasonlítás: `==, /=, <, =<, >, >=`
  - Lista rendezése: elemenként, *n*-es rendezése: méret szerint, majd elemenként
- Típuskonverzió nélkül: `=:=, =/=`
- Logikai típus nincs, helyette a `true` vagy a `false` atom lehet az eredmény
- Logikai kifejezések: `not, and, or, xor`
- Rövidzáras logikai kif.: `orelse, andalso`

Listakifejezések

- Konkatenáció: ++
- Listagenerátor:
  `[Kif || Minta<-Lista,FeltKif,...]`
  - `[2*X || X<-[1,2,3]]` ⇒ `[2,4,6]`
  - `[X/2 || X<-L, X rem 2 == 0]` ⇒ az `L` lista páros elemeinek a fele
- Kivonás: a második lista minden elemének az első előfordulását hagyja el, pl.:
  `[1,2,3,2,1]--[2,1,2]=>[3,1]`

Őrfeltételek

- Konstansok, összehasonlító, logikai és aritmetikai kifejezések
- Néhány függvény (nincs mellékhatásuk):

```
abs(N)           trunc(N)     round(N)
length(List)     hd(List)     tl(List)
size(Tuple)      element(N, Tuple)
is_atom(Term)    is_number(Term)   ...
```

- Feltételek sorozatai:
  $F_1, \ldots, F_n; \ldots; F_m, \ldots, F_{m+k} \Leftrightarrow$
  $(F_1 \wedge \ldots \wedge F_n) \vee \ldots \vee (F_m \wedge \ldots \wedge F_{m+k})$

Elágazások

- Őrfeltételek szerint (ha nincs igaz ág, az hiba!):

```
if
  Felt1 -> Kif1 ; ...;
  Felt2 -> Kif2
end
```

- Mintaillesztés alapján (kell illeszkedő minta):

```
case Kif of
  Minta1 when Felt1 -> Kif1 ; ...;
  Minta2 -> Kif2
end
```


Függvények

- Függvénydeklaráció alakja (a név egy atom):

```
fvnév(M1, ..., Mk) [when Felt1] ->
  FvTörzs1 ;
... ;
fvnév(M1, ..., Mk) [when Feltn] ->
  FvTörzsn .
```

- Például:

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


Függvénykifejezések

- Az Erlang funkcionális nyelv, a függvények önálló értékek lehetnek és létezik függvény típus.
- Névtelen függvény létrehozása:

```
fun
 (M1, ..., Mk) [when Felt1] -> Törzs1 ;
  ... ;
 (M1, ..., Mk) [when Feltn] -> Törzsn
end.
```

- Már deklarált függvény, mint érték:
  `fun fv/aritás` (pl. `fun fact/1`)

Modulok

- Egy Erlang modul attribútumok és függvénydeklarációk sorozata, mint ponttal lezárva:

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

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

Modul attribútumok

- Modul neve (a név egy atom):
  `-module(név).`
- Kívülről elérhető függvények:
  `-export([fv/arit, ..., fv/arit]).`
- Más modulokból minősítés nélkül használható függvények:
  `-import(modul,[fv/arit,...,fv/arit]).`
- Nem attribútumok, de hasonló alakúak:
  `-include("file").`
  `-define(makró, helyettesítés).`

Rekordok használata

- Nincs igazi rekord típus, a rekordokat egy jelölt tuple-ben tárolja a rendszer: `{nev,mezo1,...}`.
- Rekord bevezetése:
  `-record(rnev, {`m_1`, ..., `m_{1n}`}).`
- Rekord létrehozása (a rekord minta is ilyen):
  `#rnev{`m_1`=`v_1`, ..., `m_n`=`v_n`}.`
- Mező lekérdezése:
  `kif#rnev.`m_i

Hibakezelés

- Kivételek, hibák elkapása: `catch Expr`
  Ha a kifejezés kiértékelése közben hiba lép fel, akkor a hibát adja eredményül, különben a kifejezés eredményét.
- Lehetséges hibák alakjai:
  - Futási hiba: `{'EXIT', {Reason, Stack}}`
  - Processz megszakítás: `{'EXIT', Reason}`
- Kivételek dobása: `throw(Term)`
  A `catch` ilyenkor a `Term`-et adja vissza.

Szövegkezelő függvények

- A `string` modul néhány függvénye:
  - `len(Str)`
  - `equal(Str1, Str2)`
  - `concat(Str1, Str2)`
  - `chr(Str, Ch)`, `str(Str, Sub)`: részszöveg keresés
  - `substr(Str, Start, Length)`
  - `tokens(Str, ChList)`: a listaelemek mint lehetséges elválasztójelek mentén tördeli a szöveget

Listafüggvények

- A `lists` modul néhány függvénye:
  - `nth(N, Lst)`
  - `member(El, Lst)`
  - `delete(El, Lst)`
  - `flatten(Lst)`
  - `seq(From, To), seq(From, To, Incr)`
  - `sort(Lst)`
  - `foldl(Fun, Zero, Lst), foldr(...)`
  - `map(Fun, Lst)`

További könyvtári függvények

- A `math` modul néhány függvénye:
  - `sin, acos, exp, pow, log, sqrt`
- Az `io:format(Format, ArgLst)` formázójelei:
  - `~c`: karakter
  - `~b`: integer
  - `~f`: float
  - `~s`: string
  - `~p`: tetszőleges term
  - `~n`: sorvége

Az Erlang shell

- A shell az `erl` paranccsal indítható. Ponttal lezárt Erlang kifejezéseket értékel ki.
- Fordítás és betöltés: `c(file)`.
- Betöltés fordítás nélkül: `l(file)`.
- Help: `help()`.
- Kilépés: `halt()`.
- Ha nincs prompt (pl. mert fut egy program), a Ctrl+C-vel lehet előhozni egy egyszerű menüt.

Párhuzamos programozás

- Processz: azonosítóval és üzenetsorral rendelkező végrehajtási szál.
- Van egy önálló *pid* típus, ez reprezentálja a processzek egyedi azonosítóit. Saját *pid*: `self()`
- A processzek aszinkron módon üzeneteket küldhetnek egymásnak. Tetszőleges term lehet üzenet.
- A processzeket névvel is el lehet látni (regisztráció), így a *pid* ismerete nélkül is lehet üzenetet küldeni.

Processz indítása

- `spawn(Fun)`
  A paraméterként kapott függvényt értékeli ki új processzben (paraméter nélkül), eredményként az új *pid*et adja vissza.
- `spawn(Modul,FvNév,[Arg,...,Arg])`
  Az adott modulból exportált függvény kiértékelését indítja el egy új processzben (eredmény: *pid*).
- Az első esetben a függvény egy függvény típusú érték, a második esetben pedig egy exportált függvénynév!

Üzenetküldés

- Egy processznek üzenetet lehet küldeni a processz *pid*jének ismeretében. Szintaxis:
  ```
  pid ! message
  ```
  ahol a message tetszőleges term lehet.
- Az üzenetküldés aszinkron, a küldő processz nem várja meg, hogy a fogadó oldal átvegye az üzenetet.

Üzenetfogadás

- Mintaillesztésen alapuló szelektív üzenetfogadás haszálható Erlangban. A fogadó kifejezés:
  ```
  receive
    Minta1 [when Felt1] -> Törzs1 ;
    ... ;
    Mintan [when Feltn] -> Törzsn
  end
  ```
- Ez a kifejezés megvárja az első olyan bejövő üzenetet, amelyik megfelel az egyik mintának, majd kiértékeli a megfelelő ágat.

Példa

```
-module(print).
-export([print/1]).

print(T) ->
  Chld = spawn(fun child/0),
  Chld ! T.

child() ->
  receive
    Term -> io:format("~p~n", Term)
  end.
```

Üzenet forrásának vizsgálata

- Egy *pid* is egy term, ezért azt is el lehet küldeni egy üzenetben, és mintaillesztést lehet rá alkalmazni. Például:

```
-module(cp).
-export([parent/1, child/1]).

parent(Msg) ->
  Child = spawn(cp, child, [self()]),
  Child ! {self(), Msg}.

child(Parent) ->
  receive {Parent, T} -> ... end.
```


Alternatív fogadás

- Több ággal rendelkező fogadó utasítás is pontosan egy üzenetet vesz át, csak az üzenet alakjától függően másképp reagál.

```
server() ->
  receive
    {From, Request} ->
      From ! Reply,
      server();
    stop -> ok
  end
```


Időzítés

- Külön ágat lehet készíteni arra az esetre, ha adott időn belül nem kap üzenetet a processz.

```
receive
  ...
after
  Idő -> ...
end
```

- `Idő` ezredmásodperc után lép érvénybe az ág.

```
sleep(T) -> receive
  after T -> true end.
```


Nevesített processzek

- Egy processzhez nevet lehet regisztrálni. A név tetszőleges atom lehet, és üzenetküldés során a *pid* helyett lehet használni:

```
start() ->
  register(cnt, spawn(c,count,[0])),
  cnt ! {get,self()},
  receive {cnt,K} -> K end.
count(N) ->
  receive
   {get,P} -> P!{cnt,N}, count(N+1)
  end.
```

Rejtett kliens-szerver struktúra

```
-module(srv).
-export([start/0, store/1, retr/0]).

start() ->
  register(storage, spawn(fun srv/0)).

srv() -> srv({}).
srv(T) -> …
store(T) -> …
retr(T) -> …
```



```
srv(T) ->
  receive
    {retr, Pid} ->
      Pid ! {storage, T}, srv(T);
    {store, Pid, N} ->
      Pid ! {storage, ok}, srv(N)
  end.

store(T) ->
  storage ! {store, self(), T},
  receive {storage, ok} -> ok end.

retr() ->
  storage ! {retr, self()},
  receive {storage, T} -> T end.
```


Távoli hibák kezelése

- Ha egy több lépéses kommunikáció során valamelyik partnernél hiba történik, akkor a másik partner nehezen tud tovább működni.
- Ha egy kliensben hiba történik, az a szervert is megállíthatja, például mert az vár egy üzenetre, amit sosem fog megkapni.
- Két processzt össze lehet kötni úgy, hogy ha az egyikben hiba történik és ezért leáll, akkor erről a másik kapjon értesítést.

Exit signal

- Amikor egy processz megáll, generál egy *exit signal*-t, és elküldi a hozzákötött processzeknek.
- Kötés létrehozása:
- Ha a `receive` kifejezés szignált kap:
  - alapesetben a fogadó processz megáll
  - a `process_flag(trap_exit, true)` hívás hatására a szignálok sima üzenetekké alakulnak át

Szignálfajták

- Explicit processz-megszakítás:
  `exit(Reason)`
- Ennek hatására a következő szignál jön létre:
  `{'EXIT', Pid, Reason}`
- A programhibák is ilyen szignálokat küldenek.
- Ha hiba nélkül ér véget egy processz, akkor a `Reason` értéke `normal`. Az ilyen szignálokat csak akkor veszi figyelembe egy processz, ha üzenetté kell alakítania.