# Programming Elixir

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 6 août 2019

https://www.bortzmeyer.org/programming-elixir.html

Auteur(s) : Dave Thomas
ISBN n°978-1-68050-299-2
Éditeur : The Pragmatic Programmers
Publié en 2018

———————-

Ce livre présente le langage de programmation Elixir, un langage, pour citer la couverture, « fonctionnel, parallèle, pragmatique et amusant ».

Elixir est surtout connu dans le monde des programmeurs Erlang, car il réutilise la machine virtuelle d'Erlang pour son exécution, et reprend certains concepts d'Erlang, notamment le parallélisme massif. Mais sa syntaxe est très différente et il s'agit bien d'un nouveau langage. Les principaux logiciels libres qui l'utilisent sont Pleroma <https://pleroma.social/> (c'est via un travail sur ActivityPub que j'étais venu à Pleroma et donc à Elixir) et CertStream <https://certstream.calidog.io/>.

Comme tous les livres de cet éditeur, l'ouvrage est très concret, avec beaucoup d'exemples. Si vous voulez vous lancer, voici un exemple, avec l'interpréteur iex :

```
% iex
Erlang/OTP 22 [erts-10.4.4] [source] [64-bit] [smp:1:1] [ds:1:1:10] [async-threads:1] [hipe]

Interactive Elixir (1.8.2) - press Ctrl+C to exit (type h() ENTER for help)
iex(1)> "Hello"
"Hello"
iex(2)> 2 + 2
4
```

Mais on peut aussi bien sûr mettre du Elixir dans un fichier et l'exécuter :

```
% cat > test.exs
IO.puts 2+2

% elixir test.exs
4
```

Elixir a évidemment un mode Emacs, elixir-mode <https://melpa.org/#/elixir-mode> (site de référence sur Github <https://github.com/elixir-lang/emacs-elixir>). On peut l'utiliser seul, ou bien intégré dans l'environnement général alchemist <https://melpa.org/#/alchemist>. J'ai utilisé MELPA <https://melpa.org> pour installer elixir-mode. Une fois que c'est fait, on peut se lancer dans les exercices du livre.

Quelles sont les caractéristiques essentielles d'Elixir ? Comme indiqué sur la couverture du livre, Elixir est fonctionnel, parallèle, pragmatique et amusant. Voici quelques exemples, tirés du livre ou inspirés par lui, montrés pour la plupart en utilisant l'interpréteur iex (mais Elixir permet aussi de tout mettre dans un fichier et de compiler, chapitre 1 du livre).

Contrairement à un langage impératif classique, les variables ne sont pas modifiées (mais on peut lier une variable à une nouvelle valeur, donc l'effet est proche de celui d'un langage impératif) :

```
iex(1)> toto = 42
42
iex(2)> toto
42
iex(3)> toto = 1337
1337
iex(4)> ^toto = 1
** (MatchError) no match of right hand side value: 1
    (stdlib) erl_eval.erl:453: :erl_eval.expr/5
    (iex) lib/iex/evaluator.ex:249: IEx.Evaluator.handle_eval/5
    (iex) lib/iex/evaluator.ex:229: IEx.Evaluator.do_eval/3
    (iex) lib/iex/evaluator.ex:207: IEx.Evaluator.eval/3
    (iex) lib/iex/evaluator.ex:94: IEx.Evaluator.loop/1
    (iex) lib/iex/evaluator.ex:24: IEx.Evaluator.init/4
iex(4)> ^toto = 1337
1337
```

Dans les deux derniers tests, le caret avant le nom de la variable indique qu'on ne veut pas que la variable soit redéfinie (chapitre 2 du livre).

Elixir compte sur le "*pattern matching*" (chapitre 2 du livre) et sur les fonctions beaucoup plus que sur des structures de contrôle comme le test ou la boucle. Voici la fonction qui calcule la somme des n premiers nombres entiers. Elle fonctionne par récurrence et, dans un langage classique, on l'aurait programmée avec un test « si N vaut zéro, c'est zéro, sinon c'est N plus la somme des N-1 premiers entiers ». Ici, on utilise le "*pattern matching*" :

```
iex(1)> defmodule Test do
...(1)> def sum(0), do: 0
...(1)> def sum(n), do:  n + sum(n-1)
...(1)> end
iex(2)> Test.sum(3)
6
```

(On ne peut définir des functions nommées que dans un module, d'où le `defmodule`.)

Naturellement, comme dans tout langage fonctionnel, on peut passer des fonctions en paramètres (chapitre 5 du livre). Ici, la traditionnelle fonction `map` (qui est dans le module standard `Enum`) prend en paramètre une fonction anonyme qui multiplie par deux :

```
iex(1)> my_array = [1, 2, 8, 11]
[1, 2, 8, 11]
iex(2)> Enum.map my_array, fn x -> x*2 end
[2, 4, 16, 22]
```

Cela marche aussi si on met la fonction dans une variable :

```
iex(3)> my_func = fn x -> x*2 end
#Function<7.91303403/1 in :erl_eval.expr/5>
iex(4)> Enum.map my_array, my_func
[2, 4, 16, 22]
```

Notez qu'Elixir a une syntaxe courante mais moins claire pour les programmeurs et programmeuses venu·e·s d'autres langages, si on veut définir une courte fonction :

```
iex(5)> Enum.map my_array, &(&1*2)
[2, 4, 16, 22]
```

(Et notez aussi qu'on ne met pas forcément de parenthèses autour des appels de fonction.)

Évidemment, Elixir gère les types de données de base (chapitre 4) comme les entiers, les booléens, les chaînes de caractères…Ces chaînes sont en Unicode, ce qui fait que la longueur n'est en général pas le nombre d'octets :

```
iex(1)>  string = "Café"
"Café"
iex(2)> byte_size(string)
5
iex(3)> String.length(string)
4
```

Il existe également des structures de données, comme le dictionnaire (ici, le dictionnaire a deux élements, `nom` et `ingrédients`, le deuxième ayant pour valeur une liste) :

```
defmodule Creperie do
  def complète do
    %{nom: "Complète", ingrédients: ["Jambon", "Œufs", "Fromage"]}
  end
```

À propos de types, la particularité la plus exaspérante d'Elixir (apparemment héritée d'Erlang) est le fait que les listes de nombres sont imprimées comme s'il s'agissait de chaînes de caractères, si ces nombres sont des codes ASCII plus ou moins imprimables :

```
iex(2)> [78, 79, 78]
'NON'

iex(3)> [78, 79, 178]
[78, 79, 178]

iex(4)> [78, 79, 10]
'NO\n'

iex(5)> [78, 79, 7]
'NO\a'

iex(6)> [78, 79, 6]
[78, 79, 6]
```

Les explications complètes sur cet agaçant problème figurent dans la documentation des charlists <https://elixir-lang.org/getting-started/binaries-strings-and-char-lists.html#charlists>. Pour afficher les listes de nombres normalement, plusieurs solutions :

— Pour l'interpréteur iex, mettre `IEx.configure inspect: [charlists: false]` dans `/.iex.exs`.
— Pour la fonction `IO.inspect`, lui ajouter un second argument, `charlists: :as_lists`.
— Un truc courant est d'ajouter un entier nul à la fin de la liste, `[78, 79, 78] ++ [0]` sera affiché `[78, 79, 78, 0]` et pas `NON`.

À part cette particularité pénible, tout cela est classique dans les langages fonctionnels. Ce livre va nous être plus utile pour aborder un concept central d'Elixir, le parallélisme massif (chapitre 15). Elixir, qui hérite en cela d'Erlang, encourage les programmeuses et les programmeurs à programmer en utilisant un grand nombre d'entités s'exécutant en parallèle, les **processus** (notons tout de suite qu'un processus Elixir, exécuté par la machine virtuelle sous-jacente, ne correspond pas forcément à un processus du système d'exploitation). Commençons par un exemple trivial, avec une machine à café et un client :

```
defmodule CoffeeMachine do

  def make(sugar) do
    IO.puts("Coffee done, sugar is #{sugar}")
  end

end

CoffeeMachine.make(true)
spawn(CoffeeMachine, :make, [false])
IO.puts("Main program done")
```

Le premier appel au sous-programme `CoffeeMachine.make` est classique, exécuté dans le processus principal. Le second appel lance par contre un nouveau processus, qui va exécuter `CoffeeMachine.make` avec la liste d'arguments `[false]`.

Les deux processus (le programme principal et celui qui fait le café) sont ici séparés, aucun message n'est échangé. Dans un vrai programme, on demande en général un minimum de communication et de synchronisation. Ici, le processus parent envoie un message et le processus fils répond (il s'agit de deux messages séparés, il n'y a pas de canal bidirectionnel en Elixir, mais on peut toujours en bâtir, et c'est ce que font certaines bibliothèques) :

```
defmodule CoffeeMachine do

  def make(sugar) do
    pid = self() # PID pour Process IDentifier
    IO.puts("Coffee done by #{inspect pid}, sugar #{sugar}")
  end

  def order do
    pid = self()
    receive do
      {sender, msg} ->
      send sender, {:ok, "Order #{msg} received by #{inspect pid}"}
    end
  end

end

pid = spawn(CoffeeMachine, :order, [])
IO.puts("Writing to #{inspect pid}")
send pid, {self(), "Milk"}

receive do
  {:ok, message} -> IO.puts("Received \"#{message}\"")
  {_, message} -> IO.puts("ERROR #{message}")
after 1000 -> # Timeout after one second
  IO.puts("No response received in time")
end
```

On y notera :

— Que l'identité de l'émetteur n'est pas automatiquement ajoutée, c'est à nous de le faire,

— L'utilisation du "*pattern matching*" pour traiter les différentes types de message (avec clause `after` pour ne pas attendre éternellement),

— Que contrairement à d'autres langages à parallélisme, il n'y a pas de canaux explicites, un processus écrit à un autre processus, il ne passe pas par un canal. Là encore, des bibliothèques de plus haut niveau permettent d'avoir de tels canaux. (C'est le cas avec Phoenix, présenté plus loin.)

`spawn` crée un processus complètement séparé du processus parent. Mais on peut aussi garder un lien avec le processus créé, avec `spawn_link`, qui lie le sort du parent à celui du fils (si une exception se produit dans le fils, elle tue aussi le parent) ou `spawn_monitor`, qui transforme les exceptions ou la terminaison du fils en un message envoyé au parent :

```
defmodule Multiple do

  def newp(p) do
    send p, {self(), "I'm here"}
    IO.puts("Child is over")
    # exit(:boom) # exit() would kill the parent
    # raise "Boom" # An exception would also kill it
  end

end

child = spawn_link(Multiple, :newp, [self()])
```

Et avec `spawn_monitor` :

```
defmodule Multiple do

  def newp(p) do
```

```
    pid = self()
    send p, {pid, "I'm here"}
    IO.puts("Child #{inspect pid} is over")
    # exit(:boom) # Parent receives termination message containing :boom
    # raise "Boom" # Parent receives termination message containing a
                   # tuple, and the runtime displays the exception
  end

end

{child, _} = spawn_monitor(Multiple, :newp, [self()])
```

Si on trouve les processus d'Elixir et leurs messages de trop bas niveau, on peut aussi utiliser le module Task <https://hexdocs.pm/elixir/Task.html>, ici un exemple où la tâche et le programme principal ne font pas grand'chose à part dormir :

```
task = Task.async(fn -> Process.sleep Enum.random(0..2000); IO.puts("Task is over") end)
Process.sleep Enum.random(0..1000)
IO.puts("Main program is over")
IO.inspect(Task.await(task))
```

Vous pouvez trouver un exemple plus réaliste, utilisant le parallélisme pour lancer plusieurs requêtes réseau en évitant qu'une lente ne bloque une rapide qui suivrait, dans cet article <https://www.toptechskills.com/elixir-phoenix-tutorials-courses/clean-concurrent-code-elixir-task-
>.

Les processus peuvent également être lancés sur une autre machine du réseau, lorsqu'une machine virtuelle Erlang y tourne (chapitre 16 du livre). C'est souvent présenté par les zélateurs d'Erlang ou d'Elixir comme un bon moyen de programmer une application répartie sur l'Internet. Mais il y a deux sérieux bémols :

— Ça ne marche que si les deux côtés de l'application (sur la machine locale et sur la distante) sont écrits en Elixir ou en Erlang,
— Et, surtout, surtout, la sécurité est très limitée. La seule protection est une série d'octets qui doit être la même sur les deux machines. Comme elle est parfois fabriquée par l'utilisateur (on trouve sur le Web des exemples de programmes Erlang où cette série d'octets est une chaîne de caractères facile à deviner), elle n'est pas une protection bien solide, d'autant plus que la machine virtuelle la transmet en clair sur le réseau. (Chiffrer la communication semble difficile.)

Bref, cette technique de création de programmes répartis est à réserver aux cas où toutes les machines virtuelles tournent dans un environnement très fermé, complètement isolé de l'Internet public. Autrement, si on veut faire de la programmation répartie, il ne faut pas compter sur ce mécanisme.

Passons maintenant à des exemples où on utilise justement le réseau. D'abord, un serveur echo (je me suis inspiré de cet article <https://elixir-lang.org/getting-started/mix-otp/task-and-gen-tcp.html>). Echo est normalisé dans le RFC 862[1]. Comme le programme est un peu plus compliqué que les "*Hello, world*" faits jusqu'à présent, on va commencer à utiliser l'outil de compilation d'Elixir, mix <https://hexdocs.pm/mix/Mix.html> (chapitre 13 du livre, il existe un court tutoriel en français sur Mix <https://elixirschool.com/fr/lessons/basics/mix/>).

Le plus simple, pour créer un projet qui sera géré avec mix, est :

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc862.txt

```
% mix new echo
```

Le nouveau projet est créé, on peut l'ajouter à son VCS favori, puis aller dans le répertoire du projet et le tester :

```
% cd echo
% mix test
```

Les tests ne sont pas très intéressants pour l'instant, puisqu'il n'y en a qu'un seul, ajouté automatiquement par Mix. Mais ça prouve que notre environnement de développement marche. Maintenant, on va écrire du vrai code, dans `lib/echo.ex` (vous pouvez voir le résultat complet (en ligne sur `https://www.bortzmeyer.org/files/echo.ex`)).

Les points à noter :

— Le livre de Dave Thomas ne parle pas du tout de programmation réseau, ou des prises réseau,
— J'ai utilisé une bibliothèque Erlang.gen_tcp <`http://erlang.org/doc/man/gen_tcp.html`>, puisqu'on peut appeler les bibliothèques Erlang depuis Elixir (rappelez-vous que les deux langages utilisent la même machine virtuelle, Beam.) Les bibliothèques Erlang importées dans Elixir ont leur nom précédé d'un deux-points donc, ici, `:gen_tcp`.
— Une fois un client accepté avec `:gen_tcp.accept`, la fonction `loop_acceptor` lance un processus séparé puis s'appelle elle-même. Comme souvent dans les langages fonctionnels, on utilise la récursion là où, dans beaucoup de langages, on aurait utilisé une boucle. Cette appel récursif ne risque pas de faire déborder la pile, puisque c'est un appel terminal.
— La fonction `serve` utilise l'opérateur `|>`, qui sert à emboiter deux fonctions, le résultat de l'une devenant l'argument de l'autre (comme le tube pour le shell Unix.) Et elle s'appelle elle-même, comme `loop_acceptor`, pour traiter tous les messages envoyés.
— La fonction `write_line` est définie en utilisant le "*pattern matching*" (deux définitions possibles, une pour le cas normal, et une pour le cas où la connexion TCP est fermée.)
— Le code peut sembler peu robuste, plusieurs cas ne sont pas traités (par exemple, que se passe-t-il si `:gen_tcp.recv`, et donc `read_line`, renvoie `{:error, :reset}`? Aucune définition de `write_line` ne prévoit ce cas.) Mais cette négligence provient en partie du d'un style de développement courant en Elixir : on ne cherche pas à faire des serveurs qui résistent à tout, ce qui complique le code (la majorité du programme servant à gérer des cas rares). On préfère faire tourner le programme sous le contrôle d'un superviseur (et l'environnement OTP fournit tout ce qu'il faut pour cela, cf. chapitres 18 et 20 dans le livre), qui redémarrera le programme le cas échéant. Au premier atelier Elixir que j'avais suivi, au BreizhCamp <`https://www.breizhcamp.org/`>, le formateur, Nicolas Savois, m'avait dit que mon code était trop robuste et que je vérifiait trop de choses. Avec Elixir, c'est "*Let it crash*" <`https://www.amberbit.com/blog/2019/7/26/the-misunderstanding-of-let-it-crash/`>, et le superviseur se chargera du reste. (Dans mon serveur echo, je n'ai pas utilisé de superviseur, mais j'aurais dû.)

Et pour lancer le serveur ? Un programme `start-server.exs` contient simplement :

```
import Echo

Echo.accept(7)
```

(7 étant le port standard d'echo) Et on le démarre ainsi :

```
% sudo mix run start-server.exs
18:54:47.410 [info]  Accepting connections on port 7
...
18:55:10.807 [info]  Serving #Port<0.6>
```

La ligne « Serving #Port » est affichée lorsqu'un client apparait. Ici, on peut tester avec telnet :

```
% telnet thule echo
Trying 10.168.234.1...
Connected to thule.
Escape character is '^]'.
toto
toto
^]
telnet> quit
Connection closed.
```

Ou bien avec un client echo spécialisé, comme echoping <http://bortzmeyer.github.io/echoping/> :

```
% echoping  -v thule
...
Trying to send 256 bytes to internet address 10.168.234.1...
Connected...
TCP Latency: 0.000101 seconds
Sent (256 bytes)...
Application Latency: 0.000281 seconds
256 bytes read from server.
Estimated TCP RTT: 0.0001 seconds (std. deviation 0.000)
Checked
Elapsed time: 0.000516 seconds
```

Et si on veut faire un serveur HTTP, parce que c'est quand même plus utile ? On peut utiliser le même gen_tcp <http://erlang.org/doc/man/gen_tcp.html> comme dans l'exemple qui figure au début de cet article <https://blog.appsignal.com/2019/01/22/serving-plug-building-an-elixir-html>. Si vous voulez tester, le code est en (en ligne sur https://www.bortzmeyer.org/files/http-serve.exs), et ça se lance avec :

```
% elixir server.exs
15:56:29.721 [info]  Accepting connections on port 8080
...
```

On peut alors l'utiliser avec un client comme curl, ou bien un navigateur visitant http://localhost:8080/. Mais ce serveur réalisé en faisant presque tout soi-même est très limité (il ne lit pas le chemin de l'URL, il ne traite qu'une requête à la fois, etc) et, en pratique, la plupart des programmeurs vont s'appuyer sur des cadriciels existants comme Phoenix <https://phoenixframework.org/> ou Plug <https://elixirschool.com/en/lessons/specifics/plug/>. Par défaut, tous les deux utilisent le serveur HTTP Cowboy <https://github.com/ninenines/cowboy>, écrit en Erlang (cf. le site Web de l'entreprise qui développe Cowboy <https://ninenines.eu/>, et la documentation <https://ninenines.eu/docs/en/cowboy/2.6/guide/>.) Pour avoir des exemples concrets, regardez cet

article <https://blog.lelonek.me/minimal-elixir-http2-server-64188d0c1f3a> ou bien, avec Plug, celui-ci <https://elixirschool.com/en/lessons/specifics/plug/>.

Mix permet également de gérer les dépendances d'une application (les bibliothèques dont on a besoin), via Hex <https://hex.pm/>, le système de gestion de paquetages commun à Erlang et Elixir (chapitre 13 du livre). Voyons cela avec une bibliothèque DNS, Elixir DNS <https://github.com/tungd/elixir-dns>. On crée le projet :

```
 % mix new test_dns
* creating README.md
* creating .formatter.exs
* creating .gitignore
* creating mix.exs
* creating lib
* creating lib/test_dns.ex
* creating test
* creating test/test_helper.exs
* creating test/test_dns_test.exs
...

% mix test
Compiling 1 file (.ex)
Generated test_dns app
..

Finished in 0.04 seconds
1 doctest, 1 test, 0 failures
```

On modifie le fichier `mix.exs`, créé par Mix, pour y ajouter la bibliothèque qu'on veut, avec son numéro de version minimal :

```
# Run "mix help deps" to learn about dependencies.
defp deps do
    [
      {:dns, "~> 2.1.2"},
    ]
end
```

Et on peut alors installer les dépendances. (Pour utiliser Hex, sur Debian, il faut installer le paquetage `erlang-inets`, sinon on obtient *"(UndefinedFunctionError) function :inets.stop/2 is undefined (module :inets is not available)"*.)

```
% mix deps.get
Could not find Hex, which is needed to build dependency :dns
Shall I install Hex? (if running non-interactively, use "mix local.hex --force") [Yn] y
* creating /home/stephane/.mix/archives/hex-0.20.1
```

Le message *"Could not find Hex [...] Shall I install Hex"* n'apparaitra que la première fois. Après :

```
% mix deps.get
Resolving Hex dependencies...
Dependency resolution completed:
New:
  dns 2.1.2
  socket 0.3.13
* Getting dns (Hex package)
* Getting socket (Hex package)
```

Notez qu'Hex a également installé la dépendance de la dépendance (`dns` dépend de `socket`.) On peut maintenant exécuter nos programmes :

```
% cat example.exs
IO.inspect(DNS.resolve("github.com", :a))

% mix run ./example.exs
{:ok, [{140, 82, 118, 4}]}
```

En conclusion, je trouve le livre utile. Il est en effet très pratique, très orienté vers les programmeuses et programmeurs qui veulent avoir du code qui tourne dès les premiers chapitres. Concernant le langage, je réserve mon opinion tant que je n'ai pas écrit de vrais programmes avec.

Qui, d'ailleurs, écrit des vrais programmes avec Elixir ? Parmi les logiciels connus :

— Une mise en œuvre du fédivers, Pleroma <https://www.bortzmeyer.org/pleroma.html>.
— L'excellent serveur de suivi des journaux "*Certificate Transparency*" (RFC 6962) CertStream <https://certstream.calidog.io/> (journaux qu'il est indispensable de surveiller automatiquement pour détecter des faux certificats utilisant votre nom <https://www.bortzmeyer.org/dnspionage.html>).
— Le système Nerves <https://nerves-project.org/> de développement de code embarqué.
— Toujours pour l'IoT, il y a Astarte <http://astarte-platform.org/>.
— Le système de communication audio/vidéo Discord <https://discordapp.com/> (non libre, contrairement aux trois précédents) l'utilise. Ils publient d'intéressants articles sur les questions de passage à l'échelle, comme celui-ci <https://blog.discordapp.com/scaling-elixir-f9b8e1e7c
ou bien cet autre <https://blog.discordapp.com/using-rust-to-scale-elixir-for-11-milli
qui mentionne les limites d'Elixir.
— Le système de curation de contenu éducatif Moodle <https://moodle.com>.
— Et le futur système de coordination d'actions Mobilizon <https://joinmobilizon.org/> sera en Elixir <https://framagit.org/framasoft/mobilizon>.

Si vous cherchez des ressources supplémentaires sur Elixir, voici quelques idées :

— La bibliothèques standard est documentée en , une adresse à retenir.
— Sur la notion de superviseur, cruciale en Elixir pour écrire des serveurs non-expérimentaux, voir la documentation officielle <https://hexdocs.pm/elixir/Supervisor.html>. (Mais on les utilise typiquement via l'orchestrateur OTP.)
— Pour créer un serveur HTTP supervisé, et avec l'aide de Mix, Cowboy et Plug, voir cet article <https://www.jungledisk.com/blog/2018/03/19/tutorial-a-simple-http-server-in-elix
>.
— Une bonne introduction à Plug <https://www.brianstorti.com/getting-started-with-plug-eli
>.

---

— Pour essayer en pratique un langage de programmation, on utilise souvent l'API CRUD TODO Backend <https://www.todobackend.com/>, qui permet de gérer une liste de choses à faire (ajouter une chose à faire, la voir, la modifier, avoir la liste, la détruire). Parmi les innombrables réalisations de cette API qu'on trouve sur leur site <https://www.todobackend.com/>, il y en a trois en Elixir, une avec Phoenix <https://github.com/jeffweiss/todobackend-phoenix>, une avec Plug <https://github.com/deepredsky/todobackend-plug>, et une avec un autre cadriciel <https://github.com/whitfieldc/maru_todo>, Maru <https://maru.readme.io/>.
— Une bonne ressource pour les débutants, le site Elixir School <https://elixirschool.com/>.
— J'avais mentionné l'atelier de Nicolas Savois au BreizhCamp <https://www.breizhcamp.org/>, les documents de cet atelier sont en ligne <https://github.com/savoisn/elixir-workshop>.