Telefónica

Strand

A twist about parallelism from the 90s

13/12/2023 – Telefónica Innovación Digital

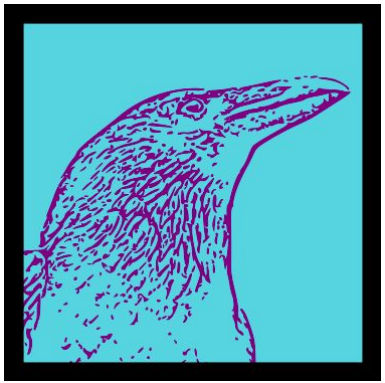
$ whoami

Adrián Arroyo Calle

En Telefónica desde 2019

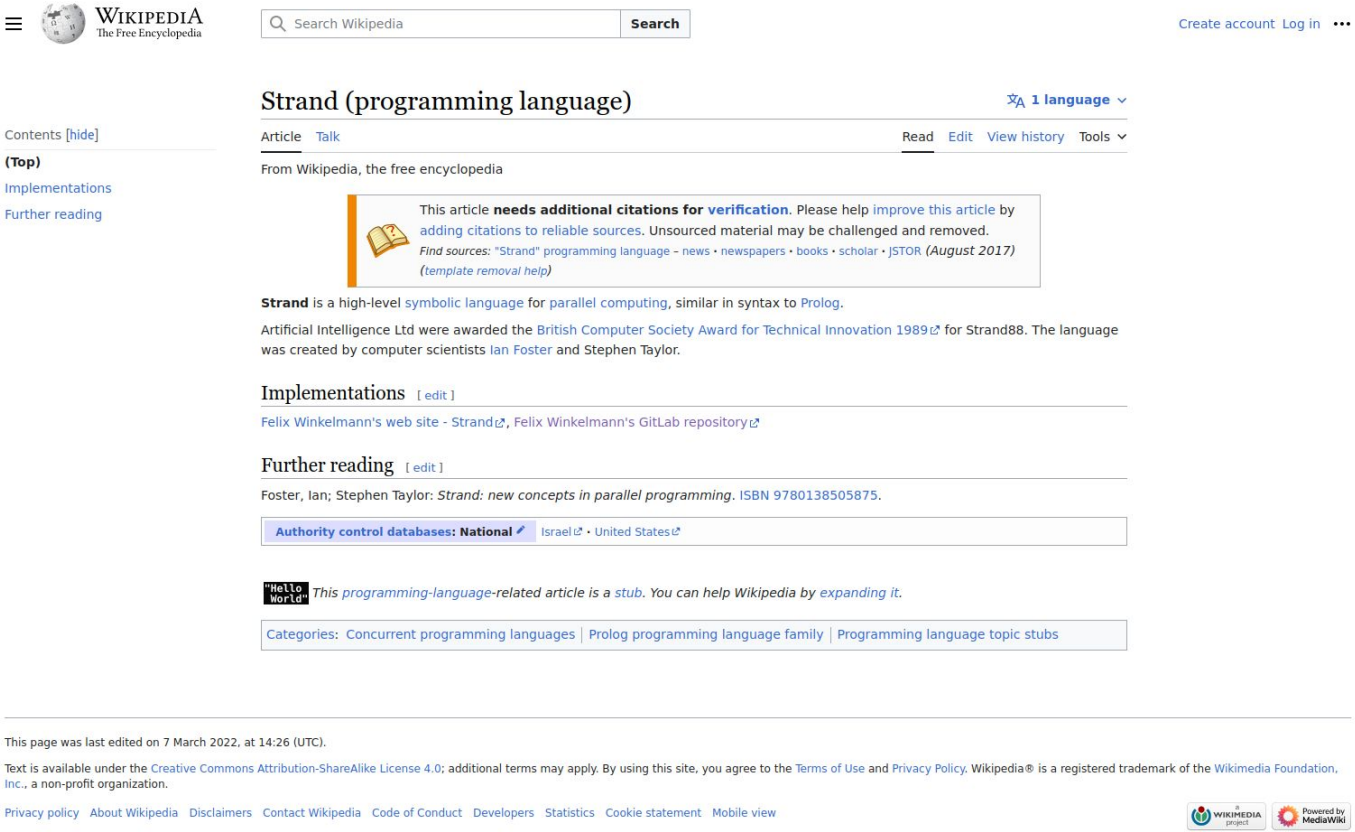
Proyecto HaaC

Prolog aficionado



Quizá no es el lenguaje más conocido del mundo

¿Qué es Strand?



WIKIPEDIA
The Free Encyclopedia

Search Wikipedia | Search

Create account Log in

Strand (programming language)

1 language

Contents [hide]

(Top)
Implementations
Further reading

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

This article **needs additional citations for verification**. Please help improve this article by adding citations to reliable sources. Unsourced material may be challenged and removed.
Find sources: "Strand" programming language – news · newspapers · books · scholar · JSTOR *(August 2017)*
(template removal help)

Strand is a high-level symbolic language for parallel computing, similar in syntax to Prolog.

Artificial Intelligence Ltd were awarded the British Computer Society Award for Technical Innovation 1989 for Strand88. The language was created by computer scientists Ian Foster and Stephen Taylor.

Implementations [edit]

Felix Winkelmann's web site - Strand, Felix Winkelmann's GitLab repository

Further reading [edit]

Foster, Ian; Stephen Taylor: *Strand: new concepts in parallel programming*. ISBN 9780138505875.

Authority control databases: National Israel · United States

"Hello World" *This programming-language-related article is a stub. You can help Wikipedia by expanding it.*

Categories: Concurrent programming languages | Prolog programming language family | Programming language topic stubs

This page was last edited on 7 March 2022, at 14:26 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License 4.0; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.

Privacy policy About Wikipedia Disclaimers Contact Wikipedia Code of Conduct Developers Statistics Cookie statement Mobile view

Powered by MediaWiki

Entonces, ¿qué es Strand?

Repasemos la historia de la familia Prolog

1972 - Prolog 0 (Marseille Prolog)

Anteriormente système OEdipe, Absys

1977 - DEC-10 Prolog

Inaugura la línea Edinburgh Prolog.

Edinburgh será la base de ISO Prolog.

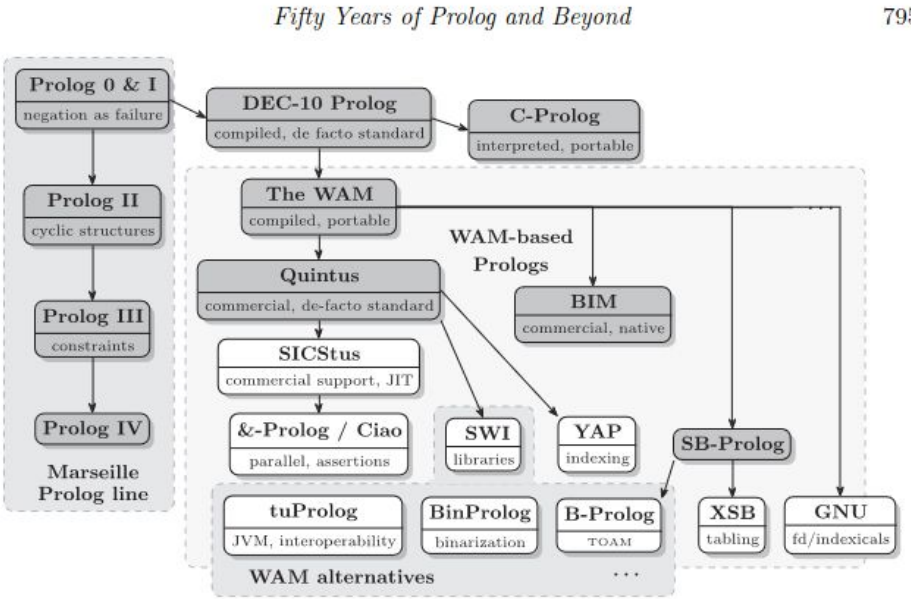
Pero quedan más líneas abiertas:

Marseille, Turbo Prolog, Arity32, microProlog, …

1981 - Fifth Generation Computer System

Japón quiere crear la siguiente generación de computadoras, altamente paralelas.

Como lenguaje en primer lugar partirá de Prolog. Gran revuelo en US y Europa Occidental.



FGCS

Japón se anticipó correctamente a que en el futuro la computación no podía crecer solamente de forma secuencial.

Hay límites físicos:

- De frecuencia, debido a la temperatura
- De miniaturización, debido a que el universo no es infinitamente pequeño

Hoy en día:

- CPUs multinúcleo incluso en móviles
- GPUs
- Computación distribuida

¿Por qué Prolog?

Prolog tiene sus raíces en la lógica de primer orden, a diferencia de la inmensa mayoría de lenguajes de programación.

La lógica de primer orden tiene varias ventajas respecto a la paralelización:

- Propiedad asociativa en la conjunción y en la disyunción
- Propiedad conmutativa en la conjunción y en la disyunción

$$p \wedge q \wedge r \equiv (p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$$

$$p \vee q \vee r \equiv (p \vee q) \vee r \equiv p \vee (q \vee r)$$

$$\frac{P \wedge Q}{\therefore Q \wedge P}$$

and

$$\frac{Q \wedge P}{\therefore P \wedge Q}$$

Es decir, hay muchas posibilidades de reordenación en la ejecución que no alteran el resultado final! Programación declarativa

¿Por qué NO Prolog?

Para 1981, Prolog se había intoxicado.

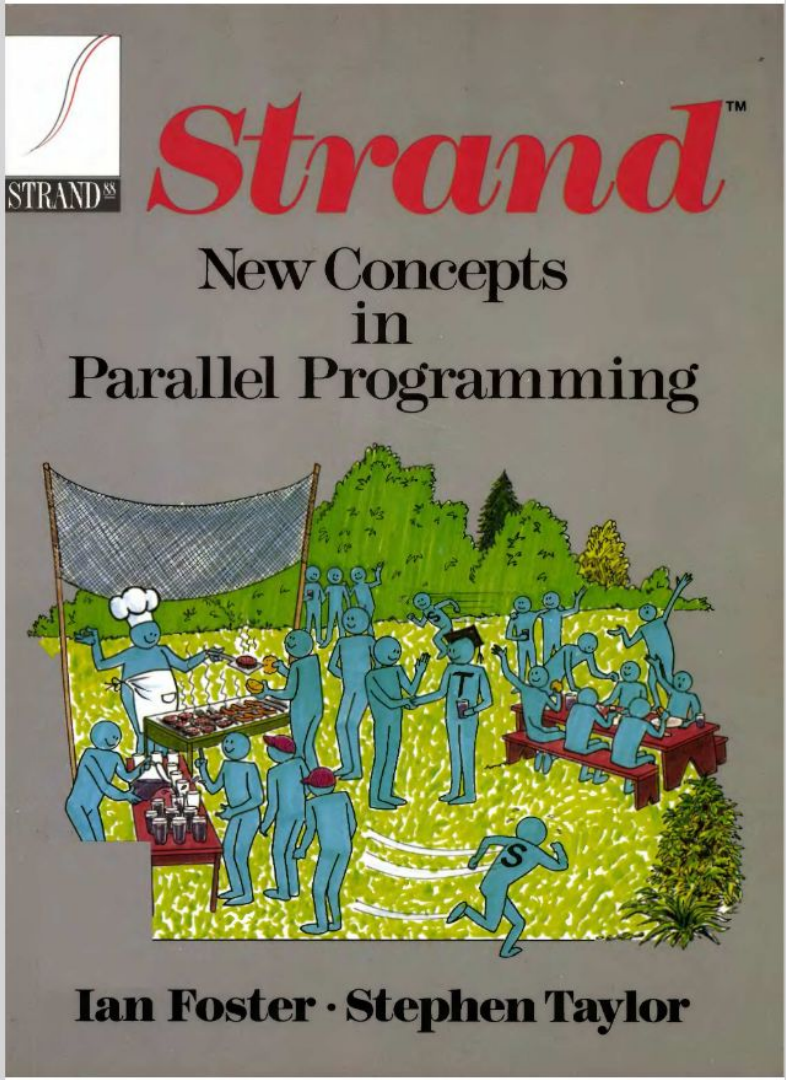
Parecía que había obstáculos insalvables que impedían lograr el ideal, (¿cómo hacer I/O?)

Muchos desarrolladores asumían un orden de ejecución de los programas.

¿Y si en vez de Prolog, que deja demasiada libertad, vamos a un modelo más cerrado donde podamos imponer más restricciones para que esto no pase?

En ese contexto surgen lenguajes como Strand

Strand: New Concepts in Parallel Programming



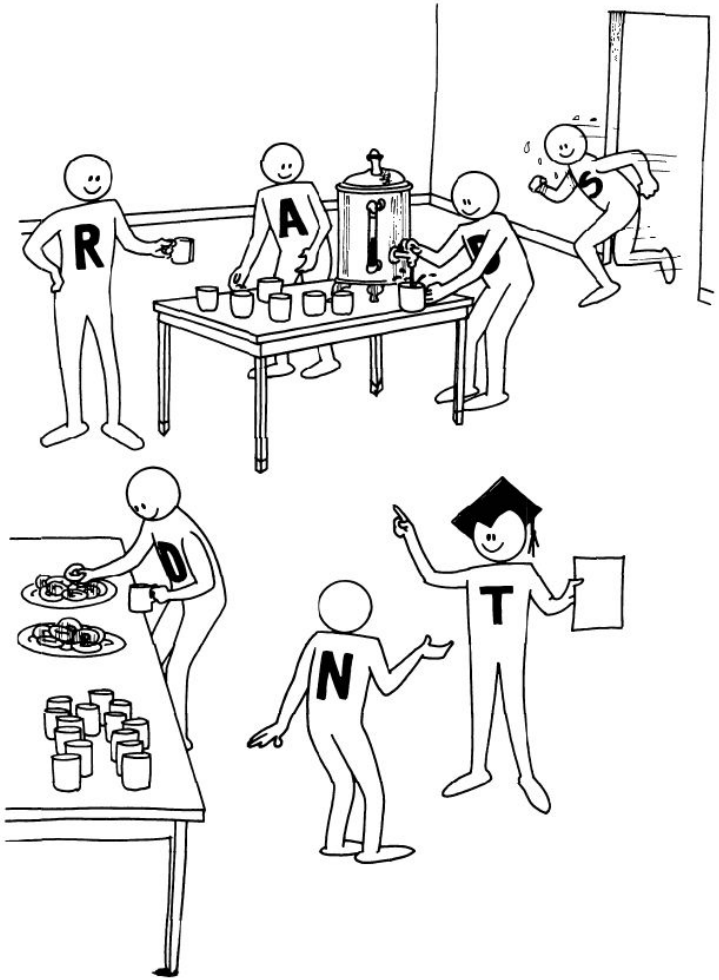
Strand 101

Strand dispone de 4 tipos básicos:

- Números
- Strings
- Variables
- Estructuras

Las variables son **lógicas**, al estilo Prolog, es decir:

- Las variables solo pueden adquirir el valor una única vez y una vez adquirido no lo pierden.
- Sintácticamente se representan con la primera letra en mayúsculas



Procesos

Un programa en Strand es un conjunto de procesos.

Cada proceso puede: terminar, cambiar de estado, forkear.

Los procesos se definen mediante reglas del siguiente tipo:

$$H :- G_1, G_2, \ldots, G_m \mid B_1, B_2, \ldots, B_n. \qquad m,n \geq 0$$

Siendo H la cabeza, G los guards, y B los bodies.

Un conjunto de reglas con mismo nombre y mismo número de argumentos forma parte de la misma definición de proceso.

Cuando un proceso dispara la regla (hace pattern matching con H), y si comprueba que cumple las condiciones G, se cambia el estado a un B arbitrario y el resto se forkean.

¿Hola Mundo?

¿Cómo implementar max en Strand?


```
max(X, Y, Max) :- X > Y | Max := X.
max(X, Y, Max) :- X =< Y | Max := Y.
```

Hola Mundo más complejo

Definición de proceso **fib/2**

Aritmética con **is**

Asignación con **:=** (no hay unificación)

Comprobaciones aritméticas con **=:=**

Al ejecutar un nuevo proceso fib, podemos entrar en cualquiera -> Guards exhaustivos

```
-initialization(main).

fib(N, X) :- N =:= 1 | X := 1.
fib(N, X) :- N =:= 2 | X := 1.
fib(N, X) :- N > 2 |
    N0 is N - 2,
    N1 is N - 1,
    fib(N0, X0),
    fib(N1, X1),
    X is X0 + X1.

main :-
    fib(12, X),
    writeln(X).
```

¡Pero esto también funciona!

El orden dentro del body es indiferente.

Se ejecutan en *paralelo*.

Además podemos hacer pattern matching sobre la Head

```
-initialization(main).

fib(1, X) :- X := 1.
fib(2, X) :- X := 1.
fib(N, X) :- N > 2 |
    X is X0 + X1,
    fib(N0, X0),
    fib(N1, X1),
    N0 is N - 2,
    N1 is N - 1.

main :-
    fib(12, X),
    writeln(X).
```

¿Cómo funciona el pattern matching en Strand?

En Prolog tenemos **unificación**, una operación **bidireccional**.

```
?- person(adrian, X, valladolid) = person(Name, arroyo, City).
   X = arroyo, Name = adrian, City = valladolid.
```

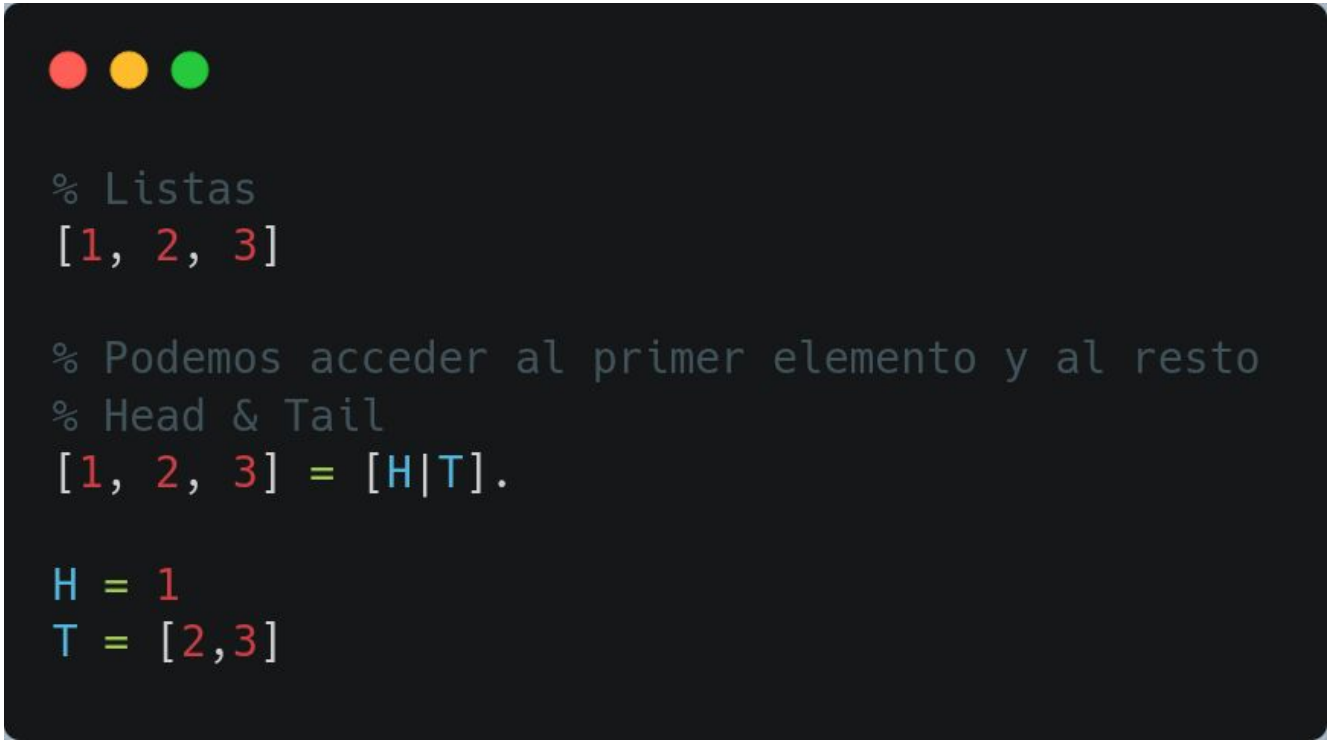
¿Cómo funciona el pattern matching en Strand?

En Strand hay pattern matching, que es unidireccional. Si queremos asignar valores no usaremos la unificación sino el operador de asignación :=

```
?- person(adrian, arroyo, valladolid) = person(Name, Surname, City).
```

```
?- person(adrian, X, valladolid) = person(Name, arroyo, City).
% esperando a X a que tenga valor
```

Listas en Strand



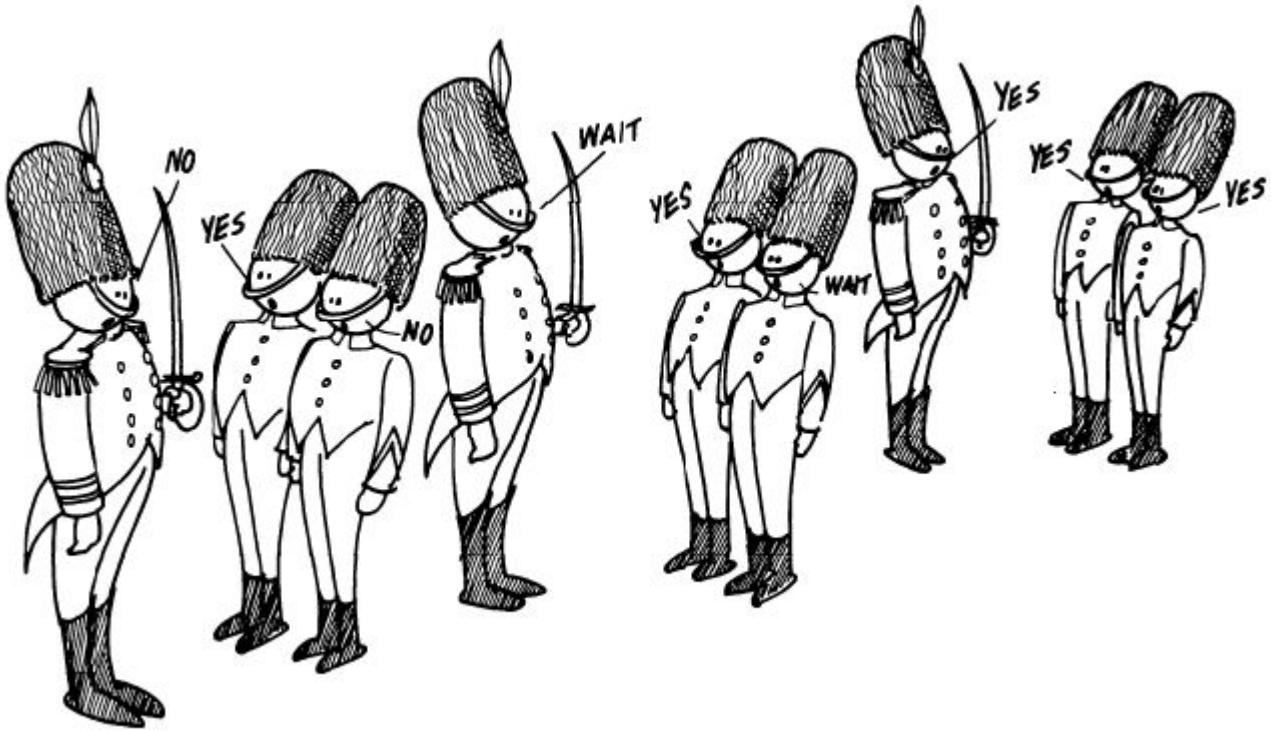
Guards

Se ejecutan secuencialmente

Esperan a tener información para decidir

Dentro de una misma definición de proceso puede haber backtracking.

Una vez se pasan todas las comprobaciones, se hace COMMIT y ya no se vuelve atrás.



- Variable = Communication Channel
- Assignment = Output (message send)
- Match = Input (message receive)

Técnicas paralelismo

En Strand

Productor - Consumidor

Mensajes unidireccionales

```
-initialization(main).

main :-
    producer(Msg),
    consumer(Msg).

producer(Msg) :-
    Msg := 'Hello World'.

consumer(Msg) :-
    writeln(Msg).
```

Productor - Consumidor con listas

Podemos usar listas incompletas, a medio definir, para enviar N elementos.

```
-initialization(main).

generator(N, Stream) :- N > 0 |
    Stream := [N|Stream1],
    N1 is N - 1,
    generator(N1, Stream1).
generator(0, Stream) :- Stream := [].

sum([], Sum) :- Sum := 0.
sum([X|Xs], Sum) :-
    Sum is Sum0 + X,
    sum(Xs, Sum0).

main :-
    generator(10, Stream),
    sum(Stream, Sum),
    writeln(Sum).
```

Mensajes incompletos

Dentro de los mensajes que enviamos en una lista incompleta, podemos dejar variables para recibir respuestas de vuelta

Comunicación bidireccional.

```
-initialization(main).

producer(N, Stream) :- N > 0 |
    Msg := square(N, Result),
    writeln(Result),
    Stream := [Msg|Stream1],
    N1 is N - 1,
    producer(N1, Stream1).
producer(0, Stream) :- Stream := [].

consumer([]).
consumer([Msg|Stream]) :-
    Msg = square(N, Result),
    Result is N * N,
    consumer(Stream).

main :-
    producer(10, Stream),
    consumer(Stream).
```


aarroyoc@adrianistan ~/d/fleng-code [1]> fleng incomplete.fghc -o incomplete
aarroyoc@adrianistan ~/d/fleng-code> ./incomplete
100
81
64
49
36
25
16
9
4
1
aarroyoc@adrianistan ~/d/fleng-code>

Mergers

Podemos juntar varios streams en uno solo mediante **merges**

```
-initialization(main).

producer(N, Stream) :- N > 0 |
    Msg := square(N, Result),
    writeln(Result),
    Stream := [Msg|Stream1],
    N1 is N - 1,
    producer(N1, Stream1).
producer(0, Stream) :- Stream := [].

consumer([]).
consumer([Msg|Stream]) :-
    Msg = square(N, Result),
    Result is N * N,
    consumer(Stream).

main :-
    producer(10, Stream0),
    producer(5, Stream1),
    merger([merge(Stream0), merge(Stream1)], Stream),
    consumer(Stream).
```


aarroyoc@adrianistan ~/d/fleng-code> ./incomplete
100
25
81
16
64
9
49
4
36
1
25
16
9
4
1
aarroyoc@adrianistan ~/d/fleng-code>

Buffers

Los consumidores generan búffers que luego los productores rellenan.

Es el consumidor quién decide cuantos datos se necesitan generar.

Es conveniente mandar un mensaje de STOP para que el productor no espere indefinidamente.

```
-initialization(main).

main :-
    consumer(Buffer),
    producer(Buffer).

producer([stop]).
producer([M|Ms]) :-
    M := 5,
    producer(Ms).

consumer(Buffer) :-
    Buffer := [X1, X2, X3|Buffer1],
    N is X1 * X2 * X3,
    writeln(N),
    stop(Buffer1).

stop(Buffer) :-
    Buffer := [stop].
```

Listas de diferencia

Podemos usar la técnica de listas de diferencia para construir listas en paralelo.

Una lista por diferencia es aquella lista a la que tenemos acceso a la vez tanto a la lista en sí, como a su cola.

```
-initialization(main).

main :-
    add_colors(T, T1),
    add_more_colors(T1, T2),
    add_extra_colors(T2, []),
    writeln(T).

add_colors(T, T1) :-
    T := [red, green, yellow|T1].

add_more_colors(T, T1) :-
    T := [blue, white, orange|T1].

add_extra_colors(T, T1) :-
    T := [black, pink, purple|T1].
```

```
aarroyoc@adrianistan ~/d/fleng-code> ./dif-list
[red, green, yellow, blue, white, orange, black, pink, purple]
```

Cortocircuito

Podemos propagar una constante a través de varios procesos, para poder detectar cuando todos ellos han acabado.

De esta forma podemos secuenciar tareas.

assign nos pueden ayudar a secuenciar tareas

```
-initialization(main).

task1(D1, D2) :-
   writeln(Text),
   assign(Text, 'Task 1', Done),
   link(Done, D1, D2).

link([], D1, D2) :- D2 := D1.

task2(D1, D2) :-
    writeln(Text),
    assign(Text, 'Task 2', Done),
    link(Done, D1, D2).

task3(done) :-
    writeln('Task 3').

main :-
    task1(done, D1),
    task2(D1, D2),
    task3(D2).
```

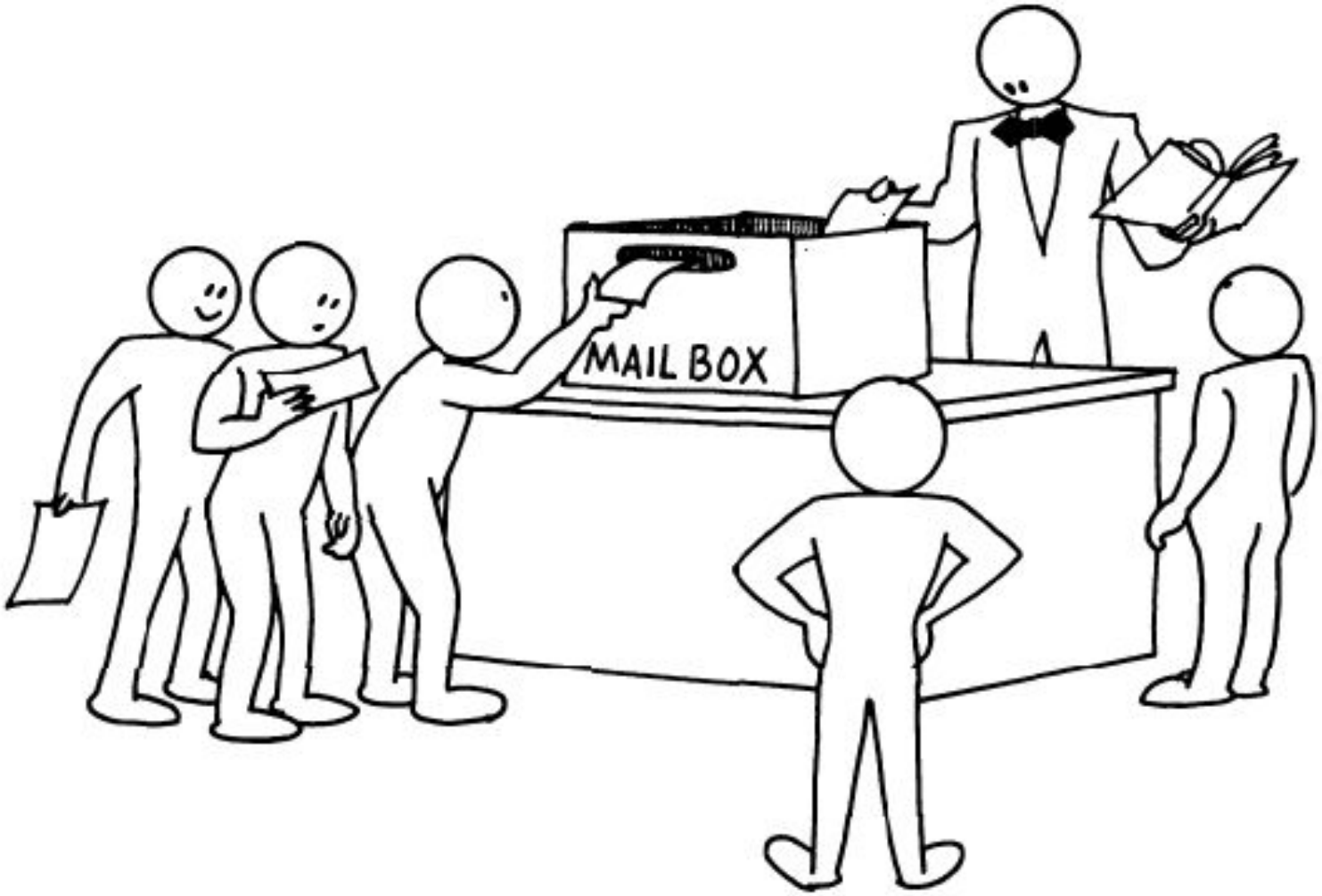
Blackboard

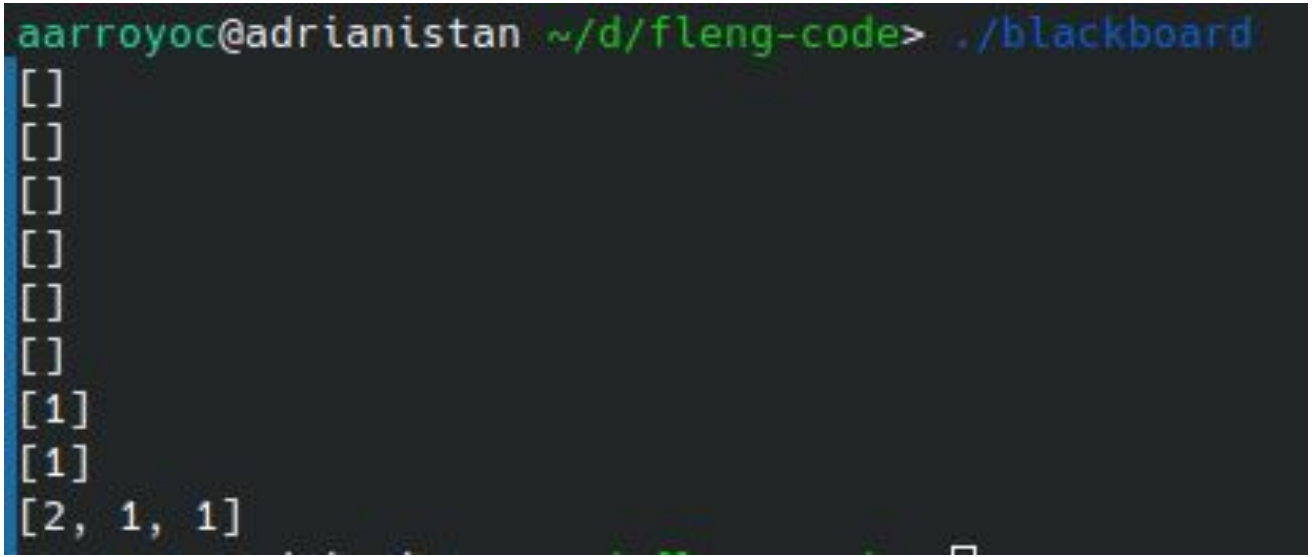
Un proceso se encarga de mantener una estructura de datos. Mediante mensajes varios lectores/escritores pueden acceder a los datos compartidos entre todos.

Mezclamos los streams con **merger** otra vez.

Dentro de un stream, se respeta el orden en la salida, pero entre los streams no hay orden definido.

```
-initialization(main).

reader(R) :-
    R := [read(R1), read(R2), read(R3)],
    writeln(R1),
    writeln(R2),
    writeln(R3).


writer(W) :-
    W := [write(1), write(2), write(3)].

main :-
    reader(R1),
    writer(W1),
    reader(R2),
    writer(W2),
    writer(W3),
    reader(R3),
    merger([merge(R1), merge(R2), merge(R3), merge(W1), merge(W2), merge(W3)], M),
    manager(M).

manager(M) :- manager(M, []).

manager([read(BB)|M], L) :-
    BB := L,
    manager(M, L).

manager([write(N)|M], L) :-
    manager(M, [N|L]).

manager([], _).
```


MAIL BOX


aarroyoc@adrianistan ~/d/fleng-code> ./blackboard
[]
[]
[]
[]
[]
[]
[1]
[1]
[2, 1, 1]

El problema de los filósofos

Problema

Cinco filósofos se sientan en una mesa. Cada uno tiene un plato de espaghetti. El espaghetti es tan escurridizo que un filósofo necesita dos tenedores (en nuestro caso llamados palillos) para comerlo. Entre cada dos platos hay un "palillo", luego existe el mismo númerode filósofos que de palillos.

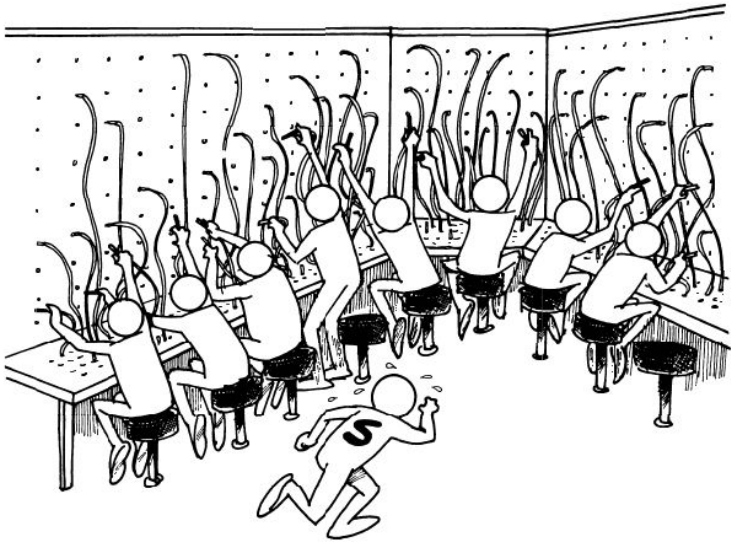
La vida de un filósofo consta de periodos alternados de comer y pensar. Cuando un filósofo siente hambre, intenta coger el palillo de la izquerda y si lo consigue, lo intenta con el de la derecha. Si logra asir dos palillos toma unos bocados y después deja los cubiertos y sigue pensando.

```
-initialization(main).

main :-
    philo(ponder, [fork1, fork2], P1),
    philo(ponder, [fork2, fork3], P2),
    philo(ponder, [fork3, fork4], P3),
    philo(ponder, [fork4, fork5], P4),
    philo(ponder, [fork5, fork1], P5),
    merger([merge(P1), merge(P2), merge(P3), merge(P4), merge(P5)], Ps),
    monitor(Ps).

philo(ponder, Forks, P) :- philo(hungry, Forks, P).
philo(hungry, [ForkLeft, ForkRight], P) :-
    P := [req(ForkLeft, ForkRight, Ok)|P1],
    philo(eat, ForkLeft, ForkRight, Ok, P1).

philo(eat, ForkLeft, ForkRight, [], P) :-
    fmt:format('Comiendo con ~s y ~s~n', [ForkLeft, ForkRight]),
    P := [rel(ForkLeft, ForkRight)|P1],
    philo(ponder, [ForkLeft, ForkRight], P1).
```

```
monitor(In) :-
    monitor(In, [], []).

monitor([req(ForkLeft, ForkRight, Ok)|In], WaitList, CurrentLends) :-
    list:member(ForkLeft, CurrentLends, FLExist),
    list:member(ForkRight, CurrentLends, FRExist),
    monitor_check(FLExist, FRExist, ForkLeft, ForkRight, Ok, In, WaitList, CurrentLends).

monitor([rel(ForkLeft, ForkRight)|In], WaitList, CurrentLends) :-
    list:delete(ForkLeft, CurrentLends, L0),
    list:delete(ForkRight, L0, L1),
    list:append(WaitList, In, In1),
    monitor(In1, [], L1).
```

```
monitor_check(false, false, ForkLeft, ForkRight, Ok, In, WaitList, CurrentLends) :-
    assign(Lends, [ForkLeft, ForkRight|CurrentLends], Ok),
    monitor(In, WaitList, Lends).

monitor_check(F1, _, ForkLeft, ForkRight, Ok, In, WaitList, CurrentLends) :-
    F1 == true |
    W := [req(ForkLeft, ForkRight, Ok)|WaitList],
    monitor(In, W, CurrentLends).

monitor_check(_, F2, ForkLeft, ForkRight, Ok, In, WaitList, CurrentLends) :-
    F2 == true |
    W := [req(ForkLeft, ForkRight, Ok)|WaitList],
    monitor(In, W, CurrentLends).
```

```
Comiendo con fork2 y fork3
Comiendo con fork4 y fork5
Comiendo con fork1 y fork2
Comiendo con fork3 y fork4
Comiendo con fork5 y fork1
Comiendo con fork2 y fork3
Comiendo con fork4 y fork5
Comiendo con fork1 y fork2
Comiendo con fork3 y fork4
Comiendo con fork5 y fork1
Comiendo con fork2 y fork3
Comiendo con fork4 y fork5
Comiendo con fork1 y fork2
Comiendo con fork3 y fork4
Comiendo con fork5 y fork1
Comiendo con fork2 y fork3
Comiendo con fork4 y fork5
Comiendo con fork1 y fork2
Comiendo con fork3 y fork4
Comiendo con fork5 y fork1
Comiendo con fork2 y fork3
Comiendo con fork4 y fork5
Comiendo con fork1 y fork2
^C⏎
aarroyoc@adrianistan ~/d/fleng-code [2]>
```


S

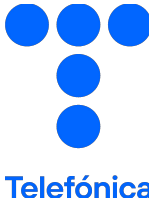
Conclusiones

Strand es un lenguaje histórico dentro del paradigma de **Concurrent Logic Programming**.

En este paradigma realizamos la comunicación entre procesos mediante el uso de **variables lógicas**

No se define explícitamente la paralelización en ningún momento.

El uso de **listas parciales** puede asemejarse a los **canales** y **paso de mensajes** que usan actualmente lenguajes como Go, Erlang, Rust, Kotlin, …


Telefónica

Telefónica

Insert Project Title Max 2 lines

Insert Subheading

dd/mm/aa – Nombre de unidad

Insert Divider Title Max 2 lines

Insert Subheading

02

INSERT SUBHEADING

Index

INSERT SUBHEADING

Insert Title Here
Max 2 lines

Body subhead (second level text)

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam euismod cursus ipsum. Hasellus erat metus, faucibus quis interdum id, viverra nec enim:

- Insert text here
  - Insert text here

INSERT SUBHEADING

Insert Title Here
Max 2 lines

Body subhead (second level text)

Lorem ipsum dolor sit amet, consectetur adispiscing elit. Aliquam euismod cursus ipsum. Hasellus erat metus, faucibus quis interdum id viverra nec enim:

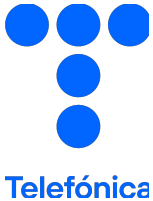
- Insert text here

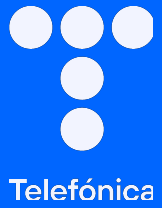
INSERT SUBHEADING

Insert Title Here
Max 2 lines

Body subhead (second level text)

	Valor 1	Valor 2	Valor 3	Valor 4	Valor 5	Valor6
Contenido	00	00	00	00	00	00
Contenido	00	00	00	00	00	00
Contenido	00	00	00	00	00	00
Contenido	00	00	00	00	00	00
Contenido	00	00	00	00	00	00
Contenido	00	00	00	00	00	00
Contenido	00	00	00	00	00	00


Telefónica


Telefónica