Introducción a la programación lógica con Scryer Prolog

Adrián Arroyo Calle

Mayo 2024

Outline

$whoami

- Adrián Arroyo Calle
- aarroyoc@adrianistan.eu
- @aarroyoc@castilla.social
- https://adrianistan.eu
- Desarrollador backend en Telefónica Innovación Digital
- Colaborador en Scryer Prolog

¿No es todo tipo de programación un pensamiento lógico?
¿Cuál es la diferencia entonces entre la programación lógica y el resto de tipos?

Paradigmas de programación

En programación existen diferentes paradigmas. En cada paradigma tenemos que pensar de forma diferente.

- Programación imperativa
  - Fundamento teórico: la máquina de Turing.
  - Lenguajes: C, Pascal, Go, ...
- Programación funcional
  - Fundamento teórico: el cálculo lambda de Alonzo Church.
  - Lenguajes: Haskell, Scheme, Idris, ...
- Programación lógica
  - Fundamento teórico: la lógica de predicados.
  - Lenguajes: Prolog, Mercury, ...

Adicionalmente tendríamos otros paradigmas como concatenativo, arrays y paradigmas *complementarios* como orientado a objetos y reactivo.

Lógica de Aristóteles

El primer tratamiento sofisticado de la lógica lo encontramos en Aristóteles, con su Órganon. La idea fundamental es que existen proposiciones, que son sentencias que pueden ser ciertas o falsas. Se dan tres axiomas:

- Principio de no contradicción
- Principio de identidad
- Principio del tercero excluido

Mediante el silogismo podemos obtener conocimiento deductivo (razonando). En contraposición existe el conocimiento inductivo (mediante la experiencia, típico de la ciencia).

Lógica de predicados

La lógica de Aristóteles fue muy estudiada durante la Edad Media por árabes y cristianos.
Pero posteriormente fueron encontrándose limitaciones. Un estudio más sistemático fue necesario.
La lógica de predicados añade expresividad, al introducir el concepto de variables.

Ejemplo

- Sócrates es humano.
- Todos los humanos son mortales.

Definimos:

- H(x) : X es humano
- M(x) : X es mortal
- soc : Sócrates

Lógica:

$$H(soc) \tag{1}$$

$$\forall x\, H(x) \implies M(x) \tag{2}$$

Deducimos $M(soc)$ por modus ponens.

Sócrates en Prolog

Código

```
human(socrates).
mortal(X) :- human(X).
```

Terminal

```
scryer-prolog socrates.pl
?- mortal(socrates).
   true.
?- mortal(X).
   X = socrates.
```

Prolog

- Prolog nació en 1972 en Marsella (Francia) de la mano de Alain Colmerauer, Philippe Roussel y Bob Kowalski.
- Estandarizado como ISO Prolog en 1995.
- Existen multitud de implementaciones, académicas, software libre, privativas, etc
  - Actualmente la implementación más balanceada en cuanto a funcionalidad, rendimiento y soporte a estándares es SICStus Prolog, software privativo.

- La idea fundamental es usar la lógica de predicados, aunque simplificada en ciertos puntos.
- Los programas son bases de conocimiento que realizan un cómputo al razonar sobre algo que pidamos.
- Prolog es *backward-chaining*, hay que indicarle primero lo queremos probar. Y Prolog lo intentará probar.
- Prolog es Turing-completo. Es decir, misma potencia expresiva que lenguajes como C, JavaScript o Haskell.

Scryer Prolog

- Scryer Prolog es un sistema ISO Prolog de software libre y comunitario.
- Escrito en Rust y Prolog.
- Todavía en versión *beta*, pero compatible con multitud de librerías.
- El objetivo es ser una implementación correcta y eficiente, en ese orden.
- Windows/macOS/Linux/WASM
- `https://www.scryer.pl`

¿Cómo usar Scryer Prolog?

- Scryer Prolog está disponible en los repos de algunas distros: NixOS, Alpine Linux, Arch Linux (AUR), Void Linux, ...
- Versiones precompiladas en `https://www.scryer.pl`
- Playground online en `https://play.scryer.pl` (*beta* y demasiado lento todavía)

Sintaxis de Prolog

- Los átomos empiezan por letra minúscula. Solo son iguales a ellos mismos.

```
?- valencia = valencia.
   true.
?- madrid = valencia.
   false.
```

Sintaxis de Prolog

- Las variables lógicas empiezan por letra mayúscula. Pueden adoptar cualquier valor, incluso otra variable. Pero solamente una vez. Se parecen más a las variables de álgebra que a las de un lenguaje imperativo.

```
?- X = valencia.
   X = valencia.
?- X = valencia, X = madrid.
   false.
```

Sintaxis de Prolog

- Los términos compuestos se basan en un átomo y a continuación entre paréntesis, N términos más.
- Variables, átomos y compuestos son términos.

```
?- X = persona(adrian, valladolid).
   X = persona(adrian, valladolid).
?- persona(N, C) = persona(adrian, valladolid).
   N = adrian, C = valladolid.
?- persona(N, valladolid) = persona(adrian, C).
   N = adrian, C = valladolid.
```

Sintaxis de Prolog

- En Prolog cada definición acaba con un punto.
- Si solo hay un término, esa definición es un *fact* o hecho.

```
?- [user].
ciudad(valencia).
ciudad(madrid).
?- ciudad(valencia).
   true.
?- ciudad(X).
   X = valencia
; ... .
```

Sintaxis de Prolog

- Si existe una implicación, esa definición es una regla.
- El conjunto de definiciones con el mismo átomo inicial y mismo número de argumentos (aridad) se llama predicado.

```
?- [user].
dark_mode.
background_color(black) :- dark_mode.

?- background_color(black).
   true.
```

Sintaxis de Prolog

- Dentro de una regla, podemos combinar predicados con , (AND) y ; (OR).

```
?- [user].
autorizado(DNI) :-
    mayor_de_edad(DNI) ;
    (menor_de_edad(DNI), autorizacion_paterna(DNI)).
```

Semántica de Prolog

- La semántica de Prolog se basa en dos operaciones que ya hemos intuido:
  - La unificación
  - El no determinismo

Unificación

- Se trata de conseguir que dos términos sean iguales.
- Se aplican comprobaciones de igualdad, pero también sustitución de variables.
- Se aplica tanto con = como al buscar predicados en el programa.

```
?- 10 = 10.
   true.
?- X = 10.
   X = 10.
?- X = Y.
   X = Y.
?- f(A, 10) = f(A, A).
   A = 10.
```

```
?- [user].
   print_msg(X) :- write(X).
?- print_msg(valladolid).
valladolid   true.
```

No determinismo

- Tanto en los OR (;) como en predicados con diferentes reglas/hechos, Prolog tiene que tomar una decisión de cuál es el siguiente paso.
- Estos puntos se llaman *choicepoints*. Prolog los almacena. Si queremos otra solución o si el camino que hemos tomado falla, Prolog vuelve atrás.

```
?- [user].
a(10).
a(20).
a(30).

?- a(X).
   X = 10
;  X = 20
```

Grafo de viajes

Tenemos un listado de precios de vuelos entre aeropuertos.
¿Cómo lo podemos expresar en Prolog?

```
vuelo(mad,bcn,200).
vuelo(bcn,mad,100).
vuelo(bcn,vll,30).
vuelo(vll,bcn,30).
vuelo(mad,vlc,50).
vuelo(vlc,mad,50).
```

Si el vuelo que buscamos es justo el trayecto que tenemos registrado, funcionará perfectamente. Pero no funcionará si necesitamos hacer escala.

```
vuelo(X, Y, Precio) :-
    vuelo(X, Z, P1),
    vuelo(Z, Y, P2),
    Precio #= P1+P2.
```

Añadimos una regla al predicado, en el cual, de nuestro origen vamos a un destino Z y desde ese destino Z buscamos vuelo hasta el destino real Y.
¡Puede ser recursivo!

¿Y si quiero calcular la ida y vuelta?

```
idavuelta(X, Y, Precio) :-
    vuelo(X, Y, P1),
    vuelo(Y, X, P2),
    Precio #= P1+P2.
```

Listas

- Prolog tiene un gran soporte a listas, aunque no son un tipo de dato fundamental.
- Entre corchetes. Podemos acceder al primer elemento con `[H|T]`

```
?- [1,2,3,4] = [H|T].
   H = 1, T = [2,3,4].
```

member/2

- `member(X, L)` es verdadero si X es un elemento de L.

```
?- member(X, [1,2,3]).
   X = 1
;  ... .
?- member(2, [1,2,3]).
   true
;  ... .
```

length/2

- `length/2` es el predicado que nos permite obtener el número de elementos de una lista
- ¡Y también construir listas de N elementos! Es bidireccional.

```
?- length([1,2,3], N).
   N = 3.
?- length([1,2,3], 3).
   true.
?- length(X, 3).
   X = [_A,_B,_C].
```

append/3

- append(A, B, AB) nos dice que AB es A seguido de B.
- ¡Vuelve la magia de Prolog!

```
?- append("abc", "def", X).
   X = "abcdef".
?- append("abc", X, "abcdef").
   X = "def".
?- append(X, XY, "abcdef").
   X = [], XY = "abcdef"
;  X = "a", XY = "bcdef"
;  X = "ab", XY = "cdef"
;  X = "abc", XY = "def"
;  X = "abcd", XY = "ef"
;  X = "abcde", XY = "f"
;  X = "abcdef", XY = [].
```

Aritmética con clpz

- La aritmética habitual no respeta las propiedades lógicas de Prolog.
- Existen librerías que nos permiten tenerlo, aunque con algunas limitaciones.

```
?- #X #= 12 + 30.
   X = 42.
?- 42 #= 12 + #X.
   X = 30.
```

Zebra

- Cinco hombres de cinco nacionalidades diferentes viven en las primeras cinco casas de una calle. Tienen cinco profesiones diferentes y cada uno tiene un animal y una bebida favorita diferentes. Cada casa está pintada de un color diferente.
- El inglés vive en la casa roja.
- El español tiene un perro.
- El japonés es pintor.
- El italiano bebe té.
- El noruego vive en la primera casa por la izquierda.
- El dueño de la casa verde bebe café.
- La casa verde está a la derecha de la casa blanca.

Zebra 2

- El escultor cultiva caracoles.
- El diplomático vive en la casa amarilla.
- La leche se bebe en la casa de enmedio.
- La casa del noruego está al lado de la casa azul.
- El violinista bebe zumo de frutas.
- El zorro está en una casa al lado de la del doctor.
- El caballo está en una casa al lado de la del diplomático.

Zebra 3

- ¿Quién tiene una cebra?
- ¿Quién bebe agua?

Zebra

```
zebra(Sol) :-
    Sol = [
        % casa(Color, Nacio, Prof, Animal, Bebida),
        casa(C1, N1, P1, A1, B1),
        casa(C2, N2, P2, A2, B2),
        casa(C3, N3, P3, A3, B3),
        casa(C4, N4, P4, A4, B4),
        casa(C5, N5, P5, A5, B5)
    ],
```

Zebra

- Primer tipo de condiciones

```
member(casa(roja, ingles, _, _, _), Sol),
member(casa(_, español, _, perro, _), Sol),
member(casa(_, japones, pintor, _, _), Sol),
member(casa(_, italiano, _, _, te), Sol),
```

Zebra

- Noruego al principio de las casas

```
Sol = [casa(_, noruego, _, _, _)|_],
```

Zebra

- Verde a la derecha de blanca

```
append(_, [
           casa(blanca, _, _, _, _),
           casa(verde, _, _, _, _)|_], Sol),
```

Zebra

- Leche en la casa de enmedio

```
append(X, [casa(_, _, _, _, leche)|Y], Sol),
length(X, 2),
length(Y, 2),
```

Zebra

- X al lado de Y

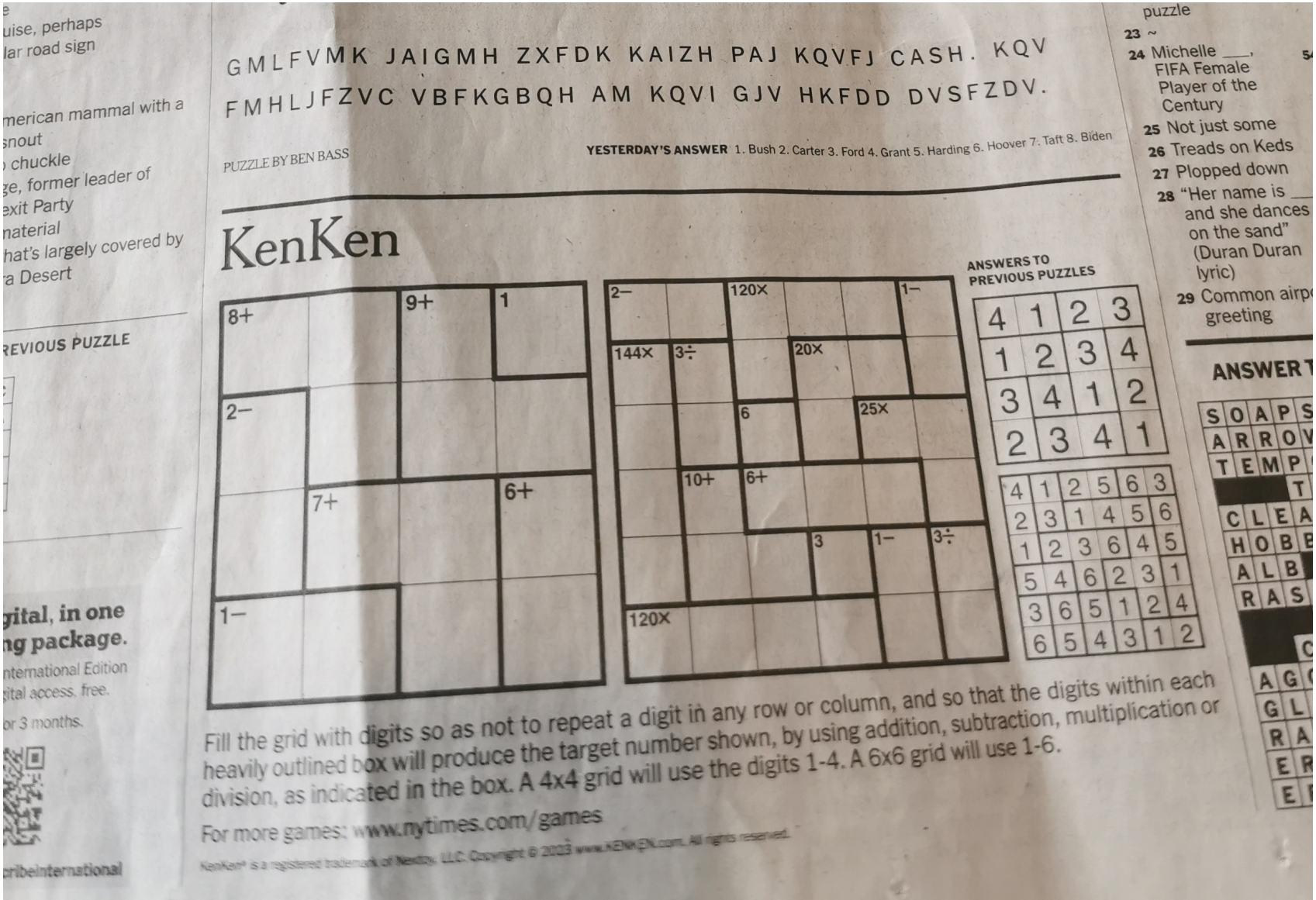
```
next_to(X, Y, Sol) :-
    append(_, [X,Y|_], Sol).
next_to(X, Y, Sol) :-
    append(_, [Y,X|_], Sol).

next_to(
    casa(_, noruego, _, _, _),
    casa(azul, _, _, _, _), Sol),
```

Zebra solución

```
?- zebra(Sol), member(casa(_, X, _, cebra, _), Sol).
   X = japones.
```

KenKen



GMLFVMK JAIGMH ZXFDK KAIZH PAJ KQVFJ CASH. KQV FMHLJFZVC VBFKGBQH AM KQVI GJV HKFDD DVSFZDV.

PUZZLE BY BEN BASS

YESTERDAY'S ANSWER 1. Bush 2. Carter 3. Ford 4. Grant 5. Harding 6. Hoover 7. Taft 8. Biden

KenKen

ANSWERS TO PREVIOUS PUZZLES

4	1	2	3
1	2	3	4
3	4	1	2
2	3	4	1

4	1	2	5	6	3
2	3	1	4	5	6
1	2	3	6	4	5
5	4	6	2	3	1
3	6	5	1	2	4
6	5	4	3	1	2

Fill the grid with digits so as not to repeat a digit in any row or column, and so that the digits within each heavily outlined box will produce the target number shown, by using addition, subtraction, multiplication or division, as indicated in the box. A 4x4 grid will use the digits 1-4. A 6x6 grid will use 1-6.

For more games: www.nytimes.com/games

KenKen

- Rellenar con dígitos de 1 a 4.
- No se puede repetir el dígito en una fila/columna.
- Los dígitos que forman parte de una caja, deben dar el número indicado, tras hacer la operación indicada entre ellos.

KenKen

- Aritmética en Prolog. Usaremos `clpz`.
- Lo primero es definir la estructura de datos del tablero.
- Una elección que tendrá mucha influencia en el código posterior.
- Debe incluir al menos una variable para cada posición del tablero.

KenKen

```
:- use_module(library(clpz)).
:- use_module(library(lists)).

clpz:monotonic.
```

KenKen

```
gen_kenken_board(Board) :-
    length(Rows, 4),
    maplist(same_length(Rows), Rows),
    Board = kenken(4, Rows,
                   [
                       box(8, sum, [1-1, 2-1, 2-2]),
                       box(9, sum, [3-1, 3-2, 4-2]),
                       box(1, [4-1]),
                       box(2, sub, [1-2, 1-3]),
                       box(7, sum, [2-3, 3-3, 3-4]),
                       box(6, sum, [4-3, 4-4]),
                       box(1, sub, [1-4, 2-4])
                   ]).
```

KenKen

- Ahora necesitamos el solver, al que le pasamos un tablero.

KenKen

```
kenken(Board) :-
    Board = kenken(Size, Rows, Boxes),
    append(Rows, Vs),
    Vs ins 1..Size,
```

KenKen

```
maplist(all_different, Rows),
```

KenKen

```
transpose(Rows, Columns),
maplist(all_different, Columns),
```

KenKen

```
maplist(box_restriction(Vars), Boxes),
```

KenKen

```
box_restriction(Vars, box(Target, [X-Y])) :-
    cell(Vars, X-Y, Cell),
    #Cell #= #Target.
```

```
box_restriction(Vars, box(Target, sum, Nums)) :-
    maplist(cell(Vars), Nums, Cells),
    sum(Cells, #=, Target).
```

```
box_restriction(Vars, box(Target, sub, Nums)) :-
    maplist(cell(Vars), Nums, Cells),
    Cells = [A, B],
    #\(#A - #B #= #Target, #B - #A #= #Target).
```

KenKen

```
cell(Vars, X-Y, Cell) :-
    nth1(Y, Vars, Row),
    nth1(X, Row, Cell).
```

KenKen

```
labeling([ffc], Vs).
```

KenKen

```
aarroyoc@adrianistan ~/d/scryer-prolog-eslibre-2024-workshop (main)> ~/dev/scryer-prolog/target/release/scryer
-prolog kenken.pl
?- use_module(library(time)).
   true.
?- gen_kenken_board_complex(X), time(kenken(X)).
   % CPU time: 0.126s, 464_255 inferences
   X = kenken(6,[[1,3,5,2,6,4],[4,6,2,5,1,3],[3,2,6,4,5,1],[6,4,3,1,2,5],[2,5,1,3,4,6],[5,1,4,6,3,2]],[box(2,s
ub,[1-1,2-1]),box(120,mul,[3-1,4-1,5-1,3-2]),box(1,sub,[6-1,6-2]),box(144,mul,[1-2,1-3,1-4,1-5]),box(3,div,[2-
2,2-3]),box(6,[3-3]),box(20,mul,[4-2,4-3,5-2]),box(25,mul,[5-3,6-3,6-4]),box(10,sum,[2-4,2-5,3-5]),box(6,sum,[
3-4,4-4,5-4]),box(3,[4-5]),box(1,sub,[5-5,5-6]),box(3,div,[6-5,6-6]),box(120,mul,[1-6,2-6,3-6,4-6])])
;  ... .
?-
```

Sudoku

- ¿Podrías implementar un solver de Sudoku de forma parecida?

FIN

¿Fin?