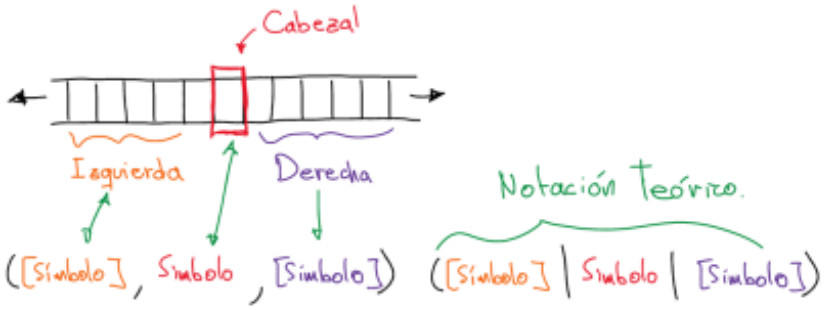
Nicolas De Lucca

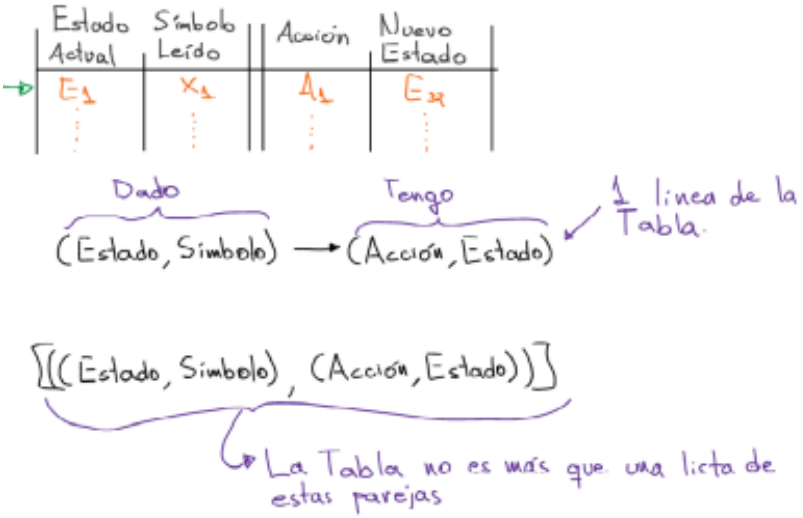
Turing Machine Programming

Semántica

- Cinta



- Tabla



- Ejecución

- Arranca con un estado inicial i y con una cinta ya dada.
- Una pareja (Estado, Cinta) determina la Configuración actual de la Máquina de Turing
- La Tabla determina la siguiente Configuración, hasta que llegamos al estado final h

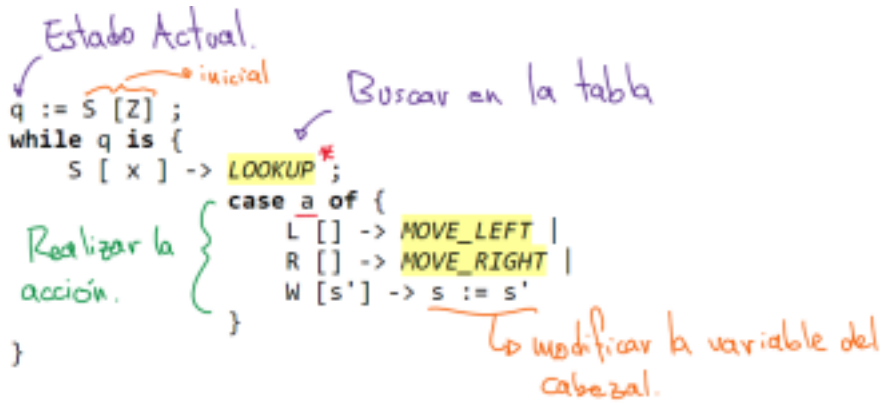
Nicolas De Lucca

Componentes

- Ejecución

Acciones:
L[]
R[]
W[símbolo]

Mientras no lleguemos a Z (estado final) nos movemos a la siguiente configuración



LOOKUP: (q, s) → (a,q')

- Tabla

[(Estado,símbolo) → (Acción, Estado Nuevo)]

[Pair[Pair[Estado, símbolo], Pair[Acción,Estado]]]

- Cinta

Usamos variables:

left → [simbolo]
right → [simbolo]
s → simbolo

- Estados (Naturales)

Z = O | S natural
Z → final
S[Z] → Inicial

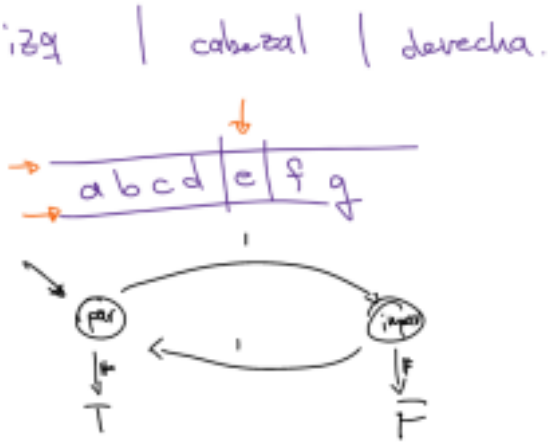
Nicolas De Lucca

- Alfabeto

[O[], 1[], Blanco[]]

Ejemplo: movámonos 3 veces a la izquierda

- ([a], b, [c, ...])
- ([ ], a, [b, c, ...])
- ([ ], #, [a, b, c, ...])
- ([ ], #, [#, a, b, c, ...])
- ([ ], #, [#, #, a, b, c, ...])
- ([#], #, [#, a, b, c, ...])



Aproximación

pasoSimple :: Tabla → Configuracion → Configuracion

ejecutarMaquina :: Tabla (comportamiento) → Cinta (cinta inicial) → Cinta (cinta final)

1. Arrancar con el estado inicial “S”
2. Llamar a pasoSimple hasta que llegue al estado final “h”

Nicolas De Lucca

Código en Haskell

```haskell
{-#OPTIONS_GHC -fno-warn-tabs #-}

module MT where
import Prelude
import Data.List

-- cada máquina emplea solo un número finito de símbolos diferentes

-- Símbolo
data Symbol = Sym String | B
 deriving (Show,Eq)

-- Cinta
data Tape = T ([Symbol], Symbol, [Symbol])
instance Show Tape where
 show (T (l,c,r)) = show l ++ " | " ++ show c ++ " | " ++ show r

-- Estados (Naturales)
-- inicial S[O]
-- final O

data Z = O | S Z
 deriving (Show,Eq)

-- Acción
data Action = L | R | W [Symbol]
 deriving Show

-- Tabla de control o código
type ElemTable = ((Z,Symbol),(Action,Z))
type Table = [ElemTable]

-- Configuración
type Config = (Z, Tape)

-- FUNCIONES:

-- ([..a], b, [c,..]) -> ([..], a, [b, c,..])
moveLeft :: Tape -> Tape
moveLeft (T (ls, s, rs)) = case ls of {
        [] -> (T (ls, B, [s] ++ rs));
        _ -> (T (init ls, last ls, [s] ++ rs))
}

-- ([..a], b, [c,..]) -> ([..,a,b], c, [..])
moveRight :: Tape -> Tape
moveRight (T (ls, s, rs)) = case rs of {
       [] -> (T (ls ++ [s], B, rs));
       _ -> (T (ls ++ [s], head rs, tail rs))
}
```

```
-- q := S [Z]
-- while q is {
--   S [x] -> case lookup (q, s) t of {
--         (a, q') -> case a of {
--                L [] -> moveLeft tp
--                R [] -> moveRight tp
--                W [s'] -> s := s'
--         }
--   }
--}

-- Paso simple
simpleStep :: Table -> Config -> Config
simpleStep t (qi, (T (ls, s, rs))) = case lookup (qi, s) t of {
      Just (a, qf) -> case a of {
          L -> (qf, moveLeft (T (ls, s, rs)));
          R -> (qf, moveRight (T (ls, s, rs)));
          W [s2] -> (qf, T (ls, s2, rs))
      };
      Nothing -> error "Inconsistent control table"
}

-- Ejecución completa
completeExecution :: Table -> Tape -> Tape
completeExecution [] tp = error "No elements in control table"
completeExecution t tp = case elem (S O) (fst (unzip (fst (unzip t)))) of {
      True -> case elem O (snd (unzip (snd (unzip t)))) of {
         False -> error "No final state in control table";
         True -> execution t (S O, tp)}; -- ejecución con estado inicial
      False -> error "No initial state in control table"
}

execution :: Table -> Config -> Tape
execution t cf = case fst cf of {
            O -> snd cf; -- estado final
            S x -> execution t (simpleStep t cf) -- llamada recursiva
}
```

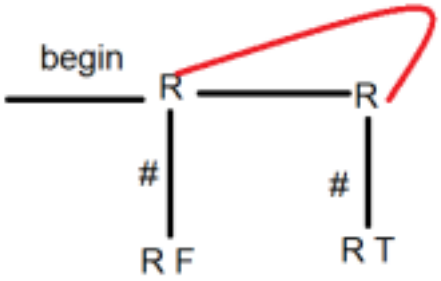
Ejemplos de máquinas Pseudocódigo y diagrama

- **Paridad** (números unarios)

```
while (cabezal != #){
      if (cabezal.R == #){
            break;
            cabezal.R;
            cabezal.W(F);
      };
      cabezal.R;
}
```

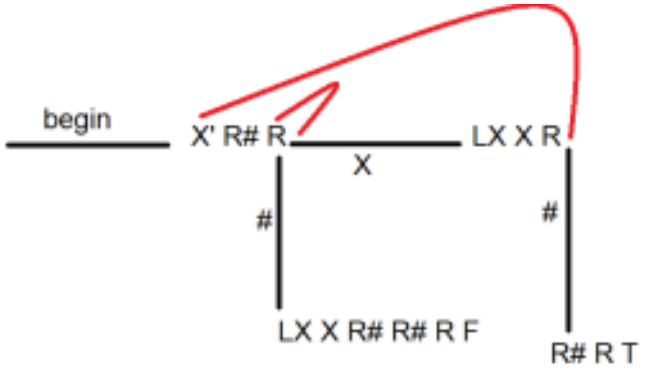
Nicolas De Lucca

```
cabezal.R;
cabezal.W(T);
```



- Desorden (sin repetidos)

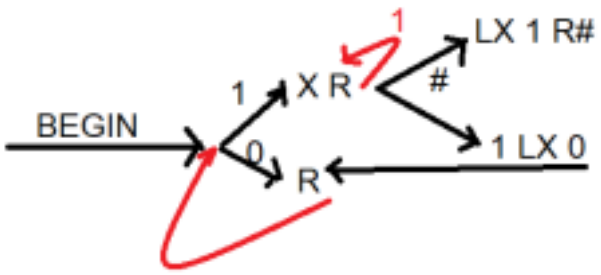
```
while (cabezal != #){
      cabezal.H(X);
      cabezal.R#;
      while (cabezal.R != #, X){
            cabezal.R;
      }
      if (cabezal == #){
            cabezal.LX;
            cabezal.W(X);
            cabezal.R#R#;
            cabezal.R;
            cabezal.W(F);
            return;
      }else{
            cabezal.LX;
            cabezal.W(X);
            cabezal.R;
      }
}
cabezal.R#;
cabezal.R;
cabezal.W(T);
```



Nicolas De Lucca

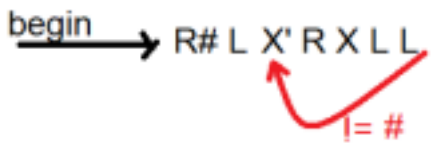
- Ordenar (números binarios)

```
while (cabezal != #){
      if (cabezal == 1){
            cabezal.W(X);
            while (cabezal.R == 1);
            if (cabezal == #){
                  cabezal.LX;
                  cabezal.W(1);
                  cabezal.R#;
                  return;
            }else{
                  cabezal.W(1);
                  cabezal.LX;
                  cabezal.W(0);
            }
      }
      cabezal.R;
}
```



- Shift Right

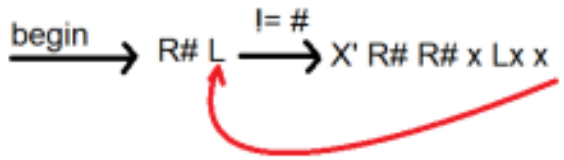
```
cabezal.R#;
cabezal.L;
while (cabezal != #){
      cabezal.H(X);
      cabezal.R;
      cabezal.W(X);
      cabezal.L;
      cabezal.L;
 }
```



Nicolas De Lucca

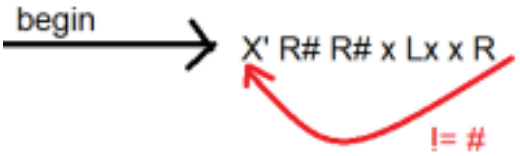
- Reverso

```
cabezal.R#;
while (cabezal.L != #){
      cabezal.H(X);
      cabezal.R#;
      cabezal.R#;
      cabezal.W(x);
      cabezal.LX;
      cabezal.W(x);
}
```



- Clon

```
while (cabezal != #){
      cabezal.H(X);
      cabezal.R#;
      cabezal.R#;
      cabezal.W(x);
      cabezal.LX;
      cabezal.W(x);
      cabezal.R;
}
```



- Producto (números unarios)

#111#1111#
#111#1111#111111111111**#**

```
while (cabezal.R != #){
      cabezal.W(X);
      cabezal.R#;
      while (cabezal.R != #){
```

```
            cabezal.W(Y);
            cabezal.R#;
            cabezal.R#;
            cabezal.W(1);
            cabezal.LY;
            cabezal.W(1);
      }
      cabezal.LX;
      cabezal.W(1);
}
cabezal.R#;
cabezal.R#;
```