U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEPARTMENT OF INFORMATICS ENGINEERING

U.PORTO
FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

DEPARTMENT OF COMPUTER SCIENCE

Functional and Logic Programming

Bachelor in Informatics and Computing Engineering
2021/2022 – 1st Semester

Prolog

Applications and Libraries

Daniel Castro Silva
dcs@fe.up.pt

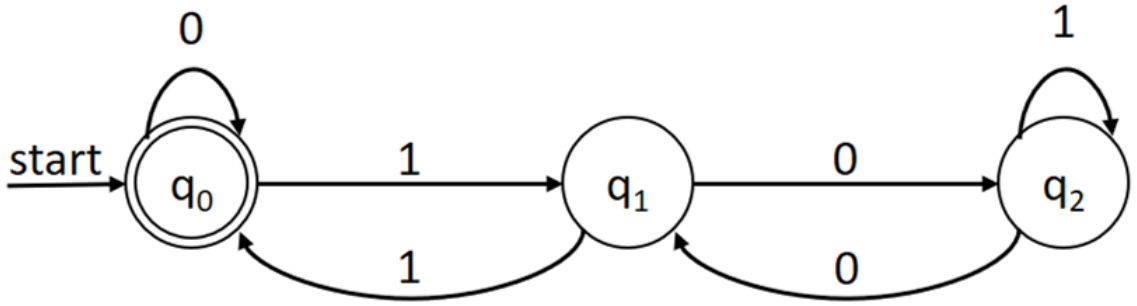
Agenda

- Computational Models
- Syntactic sugar
- Statistics
- SICStus Libraries

Computational Models

- We can easily create a Prolog program to emulate DFAs / NFAs
  - Generic solver uses graph search

DFA = ⟨ Q, ∑, δ, I, F ⟩



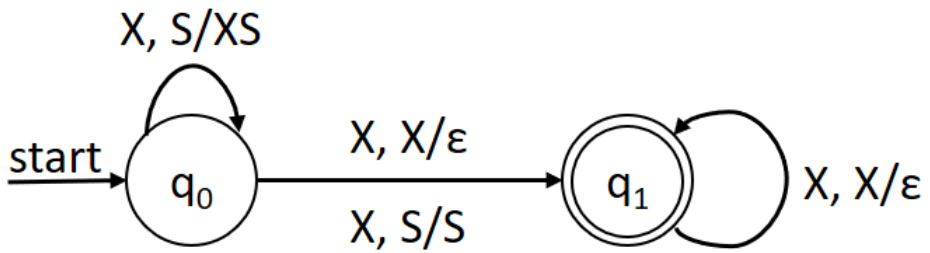

```
accept(Str):-
      initial(State),
      accept(Str, State).

accept([], State):-
      final(State).
accept([S|Ss], State):-
      delta(State, S, NState),
      accept(Ss, NState).
```

```
initial(q0).
final(q0).
delta(q0, 0, q0).
delta(q0, 1, q1).
delta(q1, 0, q2).
delta(q1, 1, q0).
delta(q2, 0, q1).
delta(q2, 1, q2).
```

Computational Models

- The same kind of logic can be used to emulate PDAs

PDA = $\langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$



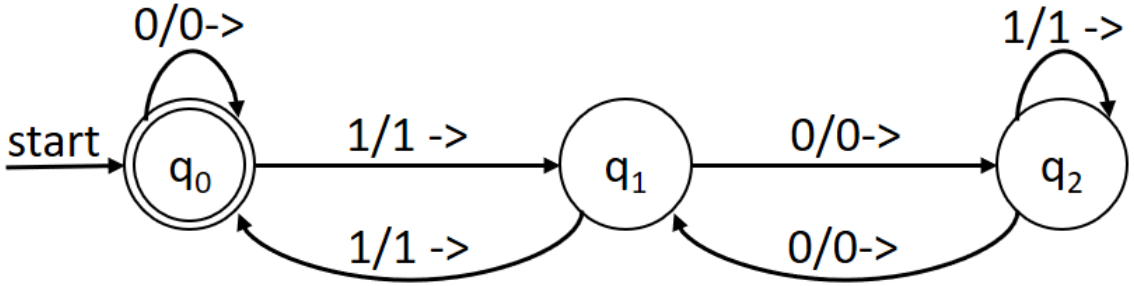

```
initial(q0).
final(q1).
delta(q0, X, Stack, q0, [X|Stack]).
delta(q0, X, Stack, q1, Stack).
delta(q0, X, [X|Stack], q1, Stack).
delta(q1, X, [X|Stack], q1, Stack).
```

```
accept(Str):- initial(State), accept(Str, State, []).

accept([], State, []):- final(State).
accept([S|Ss], State, Stack):-
      delta(State, S, Stack, NewState, NewStack),
      accept(Ss, NewState, NewStack).
```

Computational Models

- The same kind of logic can also be used to emulate TMs

TM = ⟨ Q, Σ, Γ, δ, q_0, B, F ⟩




```
initial(q0).
final(q0).
delta(q0,L,[0|R],q0,[0|L],R).
delta(q0,L,[1|R],q1,[1|L],R).
delta(q1,L,[0|R],q2,[0|L],R).
delta(q1,L,[1|R],q0,[1|L],R).
delta(q2,L,[0|R],q1,[0|L],R).
delta(q2,L,[1|R],q2,[1|L],R).
```

```
tm(Str):- initial(State),
      append(Str, [empty], StrEmpty),
      tm([empty], StrEmpty, State).
tm(Left, [S|Right], State):-
      delta(State, Left, [S|Right], NewState, NewLeft, NewRight),!,
      tm(NewLeft, NewRight, NewState).
tm(_, _, State):- final(State).
```

Computational Models

- We can also easily emulate CFGs

CFG = ⟨ V, T, P, S ⟩

S → ε
S → X
S → XSX

```
accept(Str):- s(Str).

s([]).
s([X]).
s([X|SX]):- append(S, [X], SX), s(S).
```

Computational Models

- The definition of CFGs can be simplified using DCGs (Definite Clause Grammars)
  - It uses a syntax similar to the specification of grammar rules
  - It can be used both to recognize and to generate strings

```
pal --> [].
pal --> [_].
pal --> [S], pal, [S].

palb --> [].
palb --> [S], {[S] = "0"; [S]="1"}.
palb --> [S], palb, [S], {[S] = "0"; [S]="1"}.
```

```
| ?- phrase(pal, "not a palindrome").
no
| ?- phrase(pal, "abba").
yes
| ?- phrase(pal, "madam").
yes
| ?- phrase(pal, X).
X = [] ? ;
X = [_A] ? ;
X = [_A,_A] ? ;
X = [_A,_B,_A] ? ;
X = [_A,_B,_B,_A] ? ;
X = [_A,_B,_C,_B,_A] ?
```

Syntactic Sugar

- Conditionals can be attained with the If-Then-Else construct

  `(If -> Then ; Else)`

  - The Else part may be omitted (fails if the If condition fails)

```
| ?- ( 0 is 4 mod 2 -> write(even) ; write(odd) ).
even
yes
| ?- ( 0 is 3 mod 2 -> write(even) ; write(odd) ).
odd
yes
| ?- ( 0 is 4 mod 2 -> write(even) ).
even
yes
| ?- ( 0 is 3 mod 2 -> write(even) ).
no
```

Syntactic Sugar

- Do loops can be used instead of iteration predicates

  `(Iterators do Body)`

  - There are several iterators available (see section 4.2.3.5)
    - *fromto(First, In, Out, Last)*
    - *foreach(Elem, List)*
    - *for(Iterator, MinExpression, MaxExpression)*
    - *count(Iterator, MinExpression, Max)*
    - ...
  - More than one Iterator can be used
    - They are iterated synchronously (first element of each, second of each, ...)

Syntactic Sugar

```
| ?- foreach(X, [1,2,3,4]) do Y is X*2, write(X-Y),nl.
1-2
2-4
3-6
4-8
yes
| ?- Min is 3, (for(X, Min, 2*Min) do Y is X*2, write(X-Y),nl).
3-6
4-8
5-10
6-12
Min = 3 ?
yes
| ?- Min is 3, (count(X, 2*Min, 9) do Y is X*2, write(X-Y),nl).
6-12
7-14
8-16
9-18
Min = 3 ?
yes
```

Syntactic Sugar

```
| ?- M is 3, ( for(X, 2*M, 3*M), count(Iter,1,Iterations) do Y is X*2,write(X-Y),nl ).
6-12
7-14
8-16
9-18
M = 3,
Iterations = 4 ?
yes
| ?- ( foreach(Elem, [2,4,6,8]), foreach(Res, List) do Res is Elem*3 ).
List = [6,12,18,24] ?
yes
```

```
| ?- M is 3, ( for(X, 2*M, 3*M), count(Iter,1,Iterations), foreach(Y,List) do Y is X*2 ).
M = 3,
Iterations = 4,
List = [12,14,16,18] ?
yes
```

Statistics

- Execution statistics can be obtained using the *statistics/0* or *statistics/2* predicates
  - *statistics/0* prints statistics related to memory usage, execution time, garbage collection and others (counting from session start)
  - *statistics(?Keyword, ?Value)* obtains values (or lists of values) for several available statistics
    - See section 4.10.1.2 for a full list of available keywords and respective details

```
| ?- statistics(runtime, [Before|_]), fib(30,F), statistics(runtime, [After|_]),
Time is After-Before.
Before = 16849,
F = 832040,
After = 19207,
Time = 2358 ?
yes
```

Statistics

```
| ?- statistics.
memory (total)     787611008 bytes
   global stack    443817856 bytes:       7112 in use, 443810744 free
   local stack     183558176 bytes:        368 in use, 183557808 free
   trail stack      73793272 bytes:         64 in use,  73793208 free
   choice stack     73793768 bytes:        560 in use,  73793208 free
   program space    12647872 bytes:   11145360 in use,   1502512 free
   program space breakdown:
            compiled code              3376112 bytes
            JIT code                   2590352 bytes
            sw_on_key                  1535184 bytes
            try_node                    869248 bytes
            predicate                   818400 bytes
            aatree                      656208 bytes
            atom                        515072 bytes
            interpreted code            333376 bytes
            incore_info                 252464 bytes
            atom table                   98336 bytes
            miscellaneous                46752 bytes
            SP_malloc                    32064 bytes
            int_info                      9936 bytes
            FLI stack                     5456 bytes
            BDD hash table                3168 bytes
            module                        1840 bytes
            numstack                      1056 bytes
            source info                    176 bytes
            foreign resource               160 bytes
    7279 atoms (343192 bytes) in use, 33547152 free
    No memory resource errors

       0.656 sec. for 13 global, 43 local, and 11 choice stack overflows
      15.370 sec. for 79 garbage collections which collected 3866624680 bytes
       0.000 sec. for 0 atom garbage collections which collected 0 atoms (0 bytes)
       0.000 sec. for 0 defragmentations
       0.000 sec. for 412 dead clause reclamations
       0.000 sec. for 0 dead predicate reclamations
       0.485 sec. for JIT-compiling 1097 predicates
      29.365 sec. runtime
    ========
      45.391 sec. total runtime
  734307.664 sec. elapsed time
yes
```

Statistics

```
measure_time(Keyword, Goal, Before, After, Diff):-
        statistics(Keyword, [Before|_]),
        Goal,
        statistics(Keyword, [After|_]),
        Diff is After-Before.
```

```
| ?- measure_time(runtime, fib(30,F), Before, After, Time).
F = 832040,
Before = 21973,
After = 24333,
Time = 2360 ?
yes
| ?- measure_time(total_runtime, fib(30,F), Before, After, Time).
F = 832040,
Before = 37110,
After = 41141,
Time = 4031 ?
yes
```

Libraries

- SICStus has several (54) libraries, with different purposes
  - Providing common data structures, such as sets and ordered sets, bags, queues, association lists, trees, or graphs, among others
  - Promoting interoperability, with functionalities such as
    - Parsing and writing information in csv, json or xml format
    - Connecting with databases
    - Connecting with Java or .Net applications
    - Sockets and web programming
  - Providing an Object-Oriented abstraction

Libraries

- The *aggregate* library provides operators for SQL-like queries
  - Can aggregate results using sum, count, max, min, ...

```
| ?- aggregate(count, Child^parent(Person, Child), NChildren), NChildren >1.
Person = cameron,
NChildren = 2 ? ;
Person = claire,
NChildren = 3 ?
yes
```

```
| ?- aggregate( sum(Dur), O^D^C^T^flight(O, D, Company, C, T, Dur), _TotalDur),
     aggregate( count,  O^D^C^T^Dur^flight(O,D,Company, C, T, Dur), _Count),
     AvgDuration is _TotalDur/_Count.
Company = iberia,
AvgDuration = 108.33333333333333 ? ;
Company = lufthansa,
AvgDuration = 165.0 ? ;
Company = tap,
AvgDuration = 122.0 ? ;
no
```

Libraries

- The *clpfd* library provides one of the best constraint programming solvers and library for integers
  - Very good for puzzles, and several optimization problems
  - Example: Schedule seven resource-consuming tasks so they finish as quickly as possible and such that no more than a maximum is consumed at any given time

Task	Duration	Energy Consumption
1	16	2
2	6	9
3	13	3
4	7	7
5	5	10
6	18	1
7	4	11

Constraint Programming

```
schedule(Ss, End):-
        length(Ss, 7), domain(Ss, 1, 30),
        length(Es, 7), domain(Es, 1, 50),
        buildTasks(Ss, [16,6,13,7,5,18,4], Es, [2,9,3,7,10,1,11], Tasks),
        maximum(End, Es),
        cumulative(Tasks, [limit(13)]),
        labeling([minimize(End)], [End|Ss]).

buildTasks([], [], [], [], []).
buildTasks([S|Ss], [D|Ds], [E|Es], [C|Cs], [task(S, D, E, C, 0)|Ts]):-
        buildTasks(Ss, Ds, Es, Cs, Ts).
```

```
| ?- schedule(Starts, End).
Starts = [1,17,10,10,5,5,1],
End = 23 ?
yes
```

Q & A