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
2025/2026 – 1st Semester

Prolog

Meta-Programming and Operators

Jácome Cunha
jacome@fe.up.pt

Daniel Castro Silva
dcs@fe.up.pt

Agenda

- Meta-Programming
- Operators
- Computational Models
- Incomplete Data Structures
  - Difference Lists
- Statistics
- SICStus Libraries

Agenda

- Meta-Programming
- Operators
- Computational Models
- Incomplete Data Structures
  - Difference Lists
- Statistics
- SICStus Libraries

Meta-logical Predicates

- Prolog has some meta-logical predicates for type checking
  - *integer(A)* A is an integer
  - *float(A)* A is a floating point number
  - *number(A)* A is a number (integer or float)
  - *atom(A)* A is an atom
  - *atomic(A)* A is an atom or a number
  - *compound(A)* A is a compound term
  - *var(A)* A is a variable (it is not instantiated)
  - *nonvar(A)* A is an atom, a number or a compound term
  - *ground(A)* A is *nonvar*, and all substructures are *nonvar*

Meta-logical Predicates

- These predicates can be useful for graceful fail

```
square1(X, Y):- Y is X*X.
```

```
| ?- square1(a, Y).
! Type error in argument 2 of (is)/2
! expected evaluable, but found a/0
! goal:  _337 is a*a
```

```
square(X, Y):- number(X), !, Y is X*X.
square(_X, _Y):- write('First argument
    should be a number'), nl, fail.
```

```
| ?- square(a, Y).
First argument should be a number
no
```

Meta-logical Predicates

- These predicates can be very useful to implement different versions of predicates depending on variable instantiation

```
grandparent(X, Y):- nonvar(Y), !, parent(Z, Y), parent(X, Z).
grandparent(X, Y):- parent(X, Z), parent(Z, Y).
```

- We can think of an implementation of the *sum/3* predicate that tests for instantiation, using a more appropriate definition in each case

```
sum(A, B, S):- number(A), number(B), !, S is A + B.
sum(A, B, S):- number(A), number(S), !, B is S - A.
sum(A, B, S):- number(B), number(S), !, A is S - B.
```

See section 4.8.1 of the SICStus Manual for more information

Meta-Programming

- Other predicates allow access to terms and their arguments / to construct new terms
  - *functor(+Term, ?Name, ?Arity)* or *functor(?Term, +Name, +Arity)*
    - If *Term* is instantiated, returns the name and arity of the term
    - If *Term* is not instantiated, creates a new term with given name and arity

```
| ?- functor(parent(homer, bart), Name, Arity).
Name = parent,
Arity = 2 ?
yes
| ?- functor(Term, parent, 2).
Term = parent(_A,_B) ?
yes
```

Example

```
process_term(Term, Result) :-
    functor(Term, F, _Arity),
    dispatch(F, Term, Result).

dispatch(sum, sum(A,B), R)      :- R is A + B.
dispatch(diff, diff(A,B), R)    :- R is A - B.
dispatch(neg,  neg(X),   R)     :- R is -X.
dispatch(const, const(C), C).

?- process_term(sum(2,3), R).
R = 5.
```

Meta-Programming

- *arg(+Index, +Term, ?Arg)*
  - Given an index and a term, instantiates *Arg* with the argument in the Nth position (index starts in 1)

```
| ?- arg(2, parent(homer, bart), Arg).
Arg = bart ?
yes
```

Meta-Programming

- +*Term* =.. ?*[Name* | *Args]* or ?*Term* =.. +*[Name* | *Args]*
  - Given a term, returns a list with the name and arguments of the term
  - Given a proper list, creates a new term with name and arguments as specified by the contents of the list

```
| ?- parent(homer, bart) =.. List.
List = [parent,homer,bart] ?
yes
| ?- Term =.. [parent, homer, bart].
Term = parent(homer,bart) ?
yes _
```

- The functionality of *univ* (=..) can be attained using *functor* and *arg* (and vice-versa)

Meta-Programming

- The *call/1* predicate calls (executes) a given goal

```
| ?- C = write('Hello World!'), call(C).
Hello World!
C = write('Hello World!') ?
yes
| ?- C = write('Hello World!'), C.
Hello World!
C = write('Hello World!') ?
yes
| ?- G =.. [write, 'Hi there!'], G.
Hi there!
G = write('Hi there!') ?
yes
```

  - In the example, *C* is a meta-variable - it represents a callable goal
  - *callable/1* verifies if a term is callable

Meta-Programming

- call/1 can be used with up to 255 arguments, in which case the first term is extended with the remaining arguments
  - The first argument must be instantiated
  - This has a similar effect to using univ to construct the term to call

```
| ?- X = square, call(X, 2, Y).
X = square,
Y = 4 ?
yes
| ?- X = square, T =.. [X, 2, Y], T.
X = square,
T = square(2,4),
Y = 4 ?
yes
```

Meta-Programming

- Can be used to implement *higher-order predicates*

```
map(_, []).
map(P, [H|T]):-
        G =.. [P, H],
        G,
        map(P, T).
```

```
| ?- map(write, [1,a,2,b]).
1a2b
yes
| ?- map(number, [1,a,2,b]).
no
| ?- map(atomic, [1,a,2,b]).
yes
```

wooclap

Agenda

- Meta-Programming
- Operators
- Computational Models
- Incomplete Data Structures
    - Difference Lists
- Statistics
- SICStus Libraries

Operators

- Prolog allows for the definition of new operators
  - We can easily change the way we write programs

```
homer likes marge.
marge likes homer.
homer and marge parented bart.
homer and marge parented lisa.
```

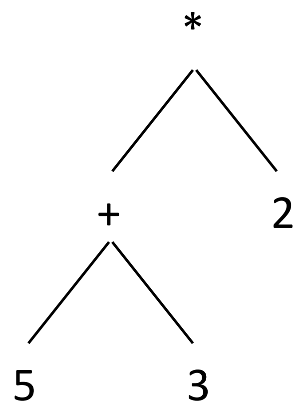
- Operators are characterized by precedence and associativity

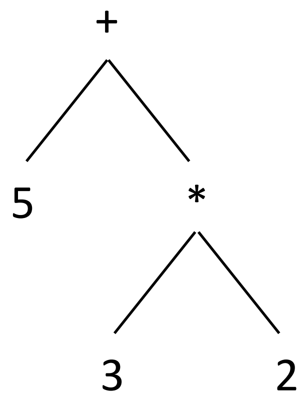
Operators

- Precedence determines which operation is executed first
  - The lower, the more priority the operator has

```
X is 5 + 3 * 2.
```

```
| ?- X is (5 + 3) * 2.
X = 16 ?
yes
| ?- X is 5 + (3 * 2).
X = 11 ?
yes
```



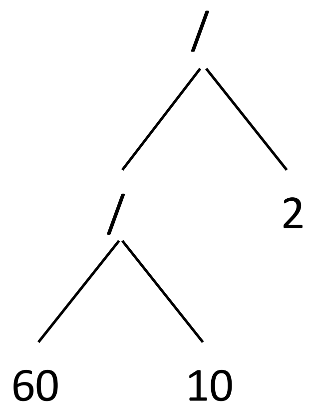


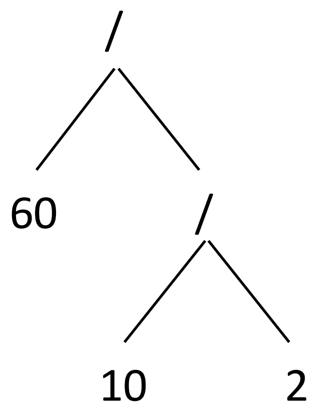
  - Precedence in Prolog is given by a number between 1 and 1200
    - Multiplication has precedence level 400
    - Addition has precedence level 500

Operators

Associativity determines how to associate operations

```
X is 60 / 10 / 2.

| ?- X is (60 / 10) / 2.
X = 3.0 ?
yes
| ?- X is 60 / (10 / 2).
X = 12.0 ?
yes
| ?- X is 60 / 10 / 2.
X = 3.0 ?
yes
```





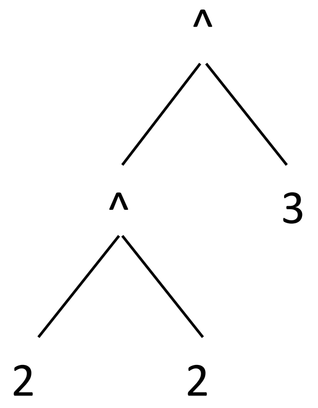
Division is left-associative

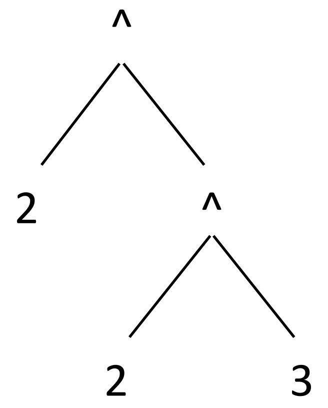
Operators

- Associativity determines how to associate operations

```
X is 2 ^ 2 ^ 3.
```

```
| ?- X is (2^2)^3.
X = 64 ?
yes
| ?- X is 2^(2^3).
X = 256 ?
yes
| ?- X is 2^2^3.
X = 256 ?
yes
```





  - The ^ operator is right-associative

Operators

- The *op/3* predicate can be used to specify new operators

```
op(+Precedence, +Type, +Name).
```

  - Precedence is a number between 1 and 1200
  - Type defines the type and associativity of the operator
    - Prefix – fx or fy
    - Postfix – xf or yf
    - Infix – xfx, xfy or yfx
    - f defines the position of the operator
    - x and y represent the operands
    - x means non-associative
    - y means side-associative

Operators

Built-in operators

```
:- op( 1200, xfx, [ :-, --> ]).
:- op( 1200,  fx, [ :-, ?- ]).
:- op( 1150,  fx, [ mode, public, dynamic, volatile, discontiguous,
                    multifile, block, meta_predicate,
                    initialization ]).
:- op( 1100, xfy, [ ;, do ]).
:- op( 1050, xfy, [ -> ]).
:- op( 1000, xfy, [ ',' ]).
:- op(  900,  fy, [ \+, spy, nospy ]).
:- op(  700, xfx, [ =, \=, is, =.., ==, \==, @<, @>, @=<, @>=,
                            =:=, =\=, <, >, =<, >= ]).
:- op(  550, xfy, [ : ]).
:- op(  500, yfx, [ +, -, \, /\, \/ ]).
:- op(  400, yfx, [ *, /, //, div, mod, rem, <<, >> ]).
:- op(  200, xfx, [ ** ]).
:- op(  200, xfy, [ ^ ]).
:- op(  200,  fy, [ +, -, \ ]).
```

Operators

Defining operators allows for a new syntax

```
:-op(380, xfy, and).
:-op(400, xfx, likes).
:-op(400, xfx, practices).

tom likes wine and cheese.
richard likes cheese.
harry practices tennis and golf.
```

```
| ?- harry practices X and Y.
X = tennis,
Y = golf ?
yes
| ?- richard likes X.
X = cheese ?
yes
| ?- tom likes X.
X = wine and cheese ?
yes
| ?- X likes wine and cheese.
X = tom ?
yes
| ?- X likes Y and Z.
X = tom,
Y = wine,
Z = cheese ?
yes
```

```
likes(tom, and(wine, cheese)).
likes(richard, cheese).
practices(harry, and(tennis, golf)).
```

```
likes(tom, wine).
likes(tom, cheese).
likes(richard, cheese).
practices(harry, tennis).
practices(harry, golf).
and(X, Y) :- X, Y.
```

Operators

To effectively use the new operators, we also need to assign semantic meaning to them, i.e., use them in predicates

```
:-op(400, xfx, parented).
:-op(380, xfy, and).

X and Y parented Z:-
      bagof(S,
            ( parent(X, S),
              parent(Y, S), X@<Y), L),
      as_list(L, Z).

as_list([A, B|T], A and R):- !,
      as_list([B|T], R).
as_list([A], A).
```

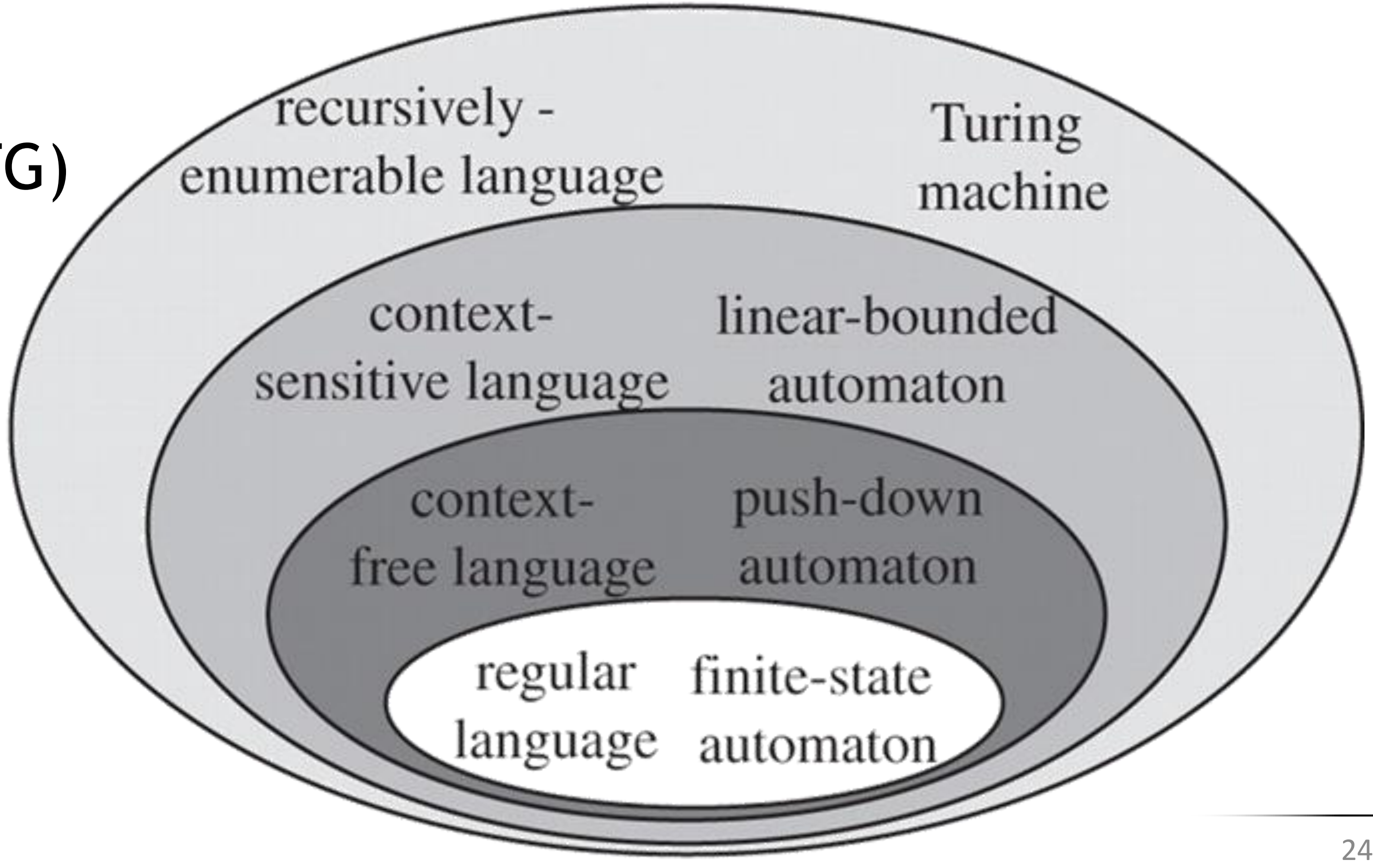
```
| ?- X and Y parented Z.
X = dede,
Y = jay,
Z = claire and mitchell ? ;
X = gloria,
Y = jay,
Z = joe ? ;
X = homer,
Y = marge,
Z = bart and lisa and maggie ? ;
no
```

Agenda

- Meta-Programming
- Operators
- Computational Models
- Incomplete Data Structures
    - Difference Lists
- Statistics
- SICStus Libraries

Computational Models

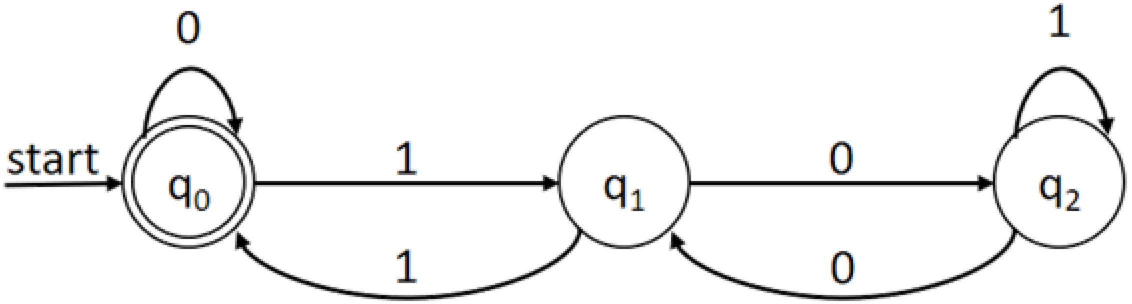
- Remembering Theory of Computation...
  - Finite Automata (DFA / NFA)
  - Pushdown Automata (PDA)
  - Context-Free Grammars (CFG)
  - Turing Machines (TM)



Computational Models

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

DFA = ⟨ Q, ∑, δ, I, F ⟩



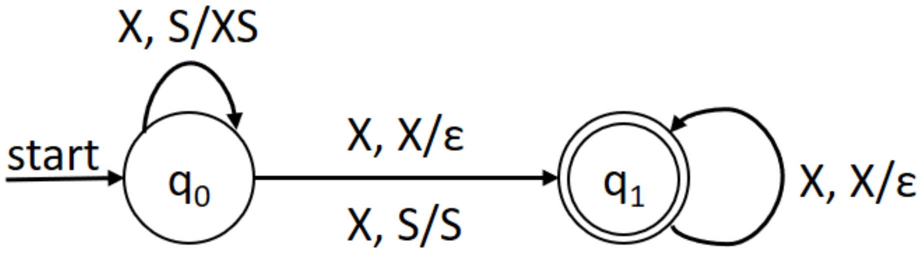

```
accept(Str):-
      initial(State),
      accept(Str, State).

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



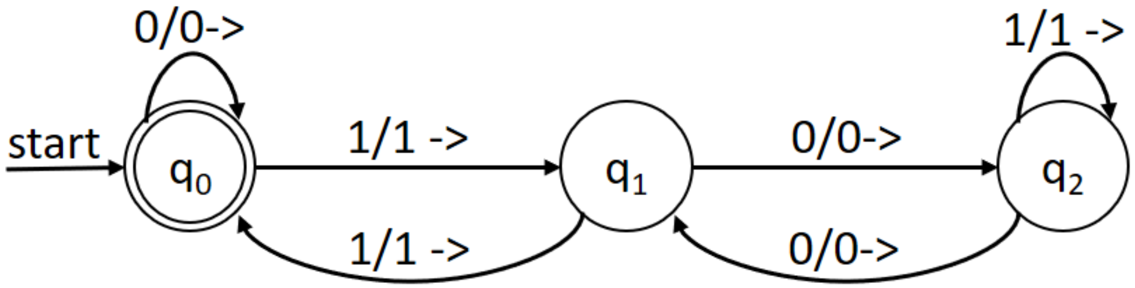

```
initial(q0).
final(q1).
delta(q0, X, Stack, q0, [X|Stack]).
delta(q0, X, Stack, q1, Stack).
delta(q0, X, [X|Stack], q1, Stack).
delta(q1, X, [X|Stack], q1, Stack).
```

```
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 = $\langle$ Q, Σ, Γ, δ, q_0, B, F $\rangle$




```
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 ⟩

$S \rightarrow \varepsilon$
$S \rightarrow X$
$S \rightarrow XSX$

```
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].
```

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

Computational Models

- Verifications can be made as extensions to the grammar rules

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

```
| ?- phrase(palb, "abba").
no
| ?- phrase(palb, "01x10").
no
| ?- phrase(palb, "01010").
true ?
yes
| ?- phrase(palb, "00").
true ?
yes
```

Computational Models

- More complex rules can be used

```
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(X) --> term(X).
term(Z) --> num(X), "*", term(Y), {Z is X * Y}.
term(Z) --> num(Z).
num(X) --> [D], num(R), {"0"=<D, D=<"9", X is (D-"0")*10 + R}.
num(X) --> [D], {"0"=<D, D=<"9", X is D-"0"}.
```

```
| ?- phrase(expr(X), "2+4").
X = 6 ?
yes
| ?- phrase(expr(X), "6+4*3").
X = 18 ?
yes
| ?- phrase(expr(X), "12*4+16").
X = 64 ?
yes
```

Agenda

- Meta-Programming
- Operators
- Computational Models
- Incomplete Data Structures
  - Difference Lists
- Statistics
- SICStus Libraries

Incomplete Data Structures

- Incomplete data structures increase efficiency by allowing ‘partial’ or ‘incomplete’ structures to be specified and incrementally constructed during runtime

  - This is achieved by maintaining a free variable as the final element of the structure, as opposed to a constant (such as [] for lists or null)

  - Changes to the incomplete structure can be made by [partially] instantiating the ending variable, thus not requiring the use of an extra output argument

Incomplete Data Structures

- Implementation of a dictionary using incomplete lists

```
lookup(Key, [ Key-Value | Dic ], Value).
lookup(Key, [ K-V | Dic ], Value):-
        Key \= K,
        lookup(Key, Dic, Value).
```

  - When *Key* is present, *Value* is verified/returned
  - When *Key* is not present, the new *Key-Value* pair is added to the dictionary

```
| ?- Dic = [x-1, y-2, z-3 | _R], lookup(y, Dic, V).
Dic = [x-1,y-2,z-3|_R],
V = 2 ?
yes
| ?- Dic = [x-1, y-2, z-3 | _R], lookup(w, Dic, 4).
Dic = [x-1,y-2,z-3,w-4|_A] ?
yes
```

Incomplete Data Structures

- Dictionary implemented with incomplete binary search tree

```
lookup(Key, dtnode(Key-Value, _L, _R), Value).
lookup(Key, dtnode(K-_V, L, _R), Value):-
      Key < K, lookup(Key, L, Value).
lookup(Key, dtnode(K-_V, _L, R), Value):-
      Key > K, lookup(Key, R, Value).
```

```
| ?- dtree(DTree).
DTree = dtnode(3-b,dtnode(1-(a),_A,_B),dtnode(7-d,dtnode(5-c,_C,_D),dtnode(9-(e),_E,_F))) ?
yes
| ?- dtree(DTree), lookup(5, DTree, V).
DTree = dtnode(3-b,dtnode(1-(a),_A,_B),dtnode(7-d,dtnode(5-c,_C,_D),dtnode(9-(e),_E,_F))),
V = c ?
yes
| ?- dtree(DTree), lookup(4, DTree, g).
DTree = dtnode(3-b,dtnode(1-(a),_A,_B),dtnode(7-d,dtnode(5-c,dtnode(4-g,_C,_D),_E),dtnode(9-(
e),_F,_G))) ?
yes
```

Difference Lists

- While lists are widely used, some common operations may not be very efficient, as is the case of appending two lists
  - Linear on the size of the first list
- Idea: increase efficiency by 'also keeping a pointer to the end of the list'
  - This is accomplished by using difference lists
    - We can use any symbol to separate the two parts of the difference list
    - With this representation, we can have an incomplete list (when the second list is not instantiated)

```
X = [1, 2, 3]

X = [1, 2, 3, 4, 5, 6]\[4, 5, 6]
X = [1, 2, 3, a, b, c]\[a, b, c]
X = [1, 2, 3]\[]
X = [1, 2, 3 | T]\T
```

Difference Lists

- We can now append two (difference) lists in constant time
  - To append X\Y with Z\W, simply unify Y with Z

```
append_dl(X\Y, Y\W, X\W).
```

  - Note that the two lists must be compatible – the tail of the first list must either be uninstantiated or be equal to the second list

```
| ?- append_dl( [a, b, c | Y ]\Y, [d, e, f | W]\W, A).
Y=[d,e,f|W]
A=[a,b,c,d,e,f|W]\W
```

Agenda

- Meta-Programming
- Operators
- Computational Models
- Incomplete Data Structures
    - Difference Lists
- Statistics
- SICStus Libraries

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
```

Agenda

- Meta-Programming
- Operators
- Computational Models
- Incomplete Data Structures
    - Difference Lists
- Statistics
- SICStus Libraries

SICStus Libraries

- SICStus has several (~55) 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 Object-Oriented abstraction
  - …

Aggregate Library

- The *aggregate* library provides operators for SQL-like queries
  - Results can be aggregated using sum, count, min, max, ...

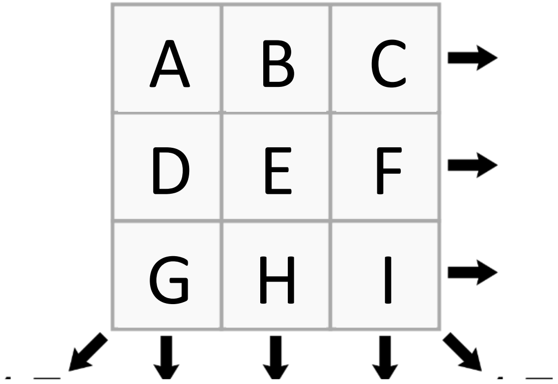
```
| ?- aggregate(count, Child^parent(Person, Child), NChildren), NChildren >1.
Person = cameron,
NChildren = 2 ? ;
Person = claire,
NChildren = 3 ?
yes
```

```
| ?- 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
```

CLPFD Library

- The *clpfd* library provides one of the best constraint programming solvers and library for integers
  - Very good for puzzles, and combinatorial optimization problems
  - Example: solve the 3x3 magic square

```
| ?- L = [A,B,C,D,E,F,G,H,I], domain(L, 1, 9), all_distinct(L),
     A+B+C#=Sum, D+E+F#=Sum, G+H+I#=Sum,
     A+D+G#=Sum, B+E+H#=Sum, C+F+I#=Sum,
     A+E+I#=Sum, C+E+G#=Sum, labeling([], L).
L = [2,7,6,9,5,1,4,3,8],
A = 2,
...
I = 8,
Sum = 15 ?
yes
```



CLPFD Library

- Another 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

Maximum instantaneous energy consumption: 13

CLPFD Library

```
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

Q & A