Programming with

L ogic
I nheritance
F unctions
E quations

Hassan Aït-Kaci

ILOG, Inc.

Outline

- Generalities
- LIFE's basic data structure: the ψ-term
- Predicates
- Functions
- Sorts
- Programming examples
- Conclusion

Generalities

- Idea:

  To mix programming with:

  - logical relations (defined as Horn clauses),
  - functional expressions (including higher-order),
  - object approximations (using inheritance).

- Key:

  Using a universal and flexible data structure called ψ-term.

Syntax

LIFE is a generalization of Prolog:

most Prolog programs run under LIFE.

Same syntactic conventions:

- variables are capitalized (or start with `_`)
- other identifiers start with a lower-case letter
- the unification predicate is `=`
- defining Horn clauses uses `:-`
- the cut control operator is `!`
- *etc.*

Syntax

Syntactic conventions differing from Prolog's:

- queries are terminated with a `?`
- assertions are terminated with a `.`

Interactive querying is incremental:

- levels are marked by `--`$\cdots n$`>`
- backtracking brings to previous level.

Ψ-Terms

- `42`
- `int`
- `-5.66`
- `real`
- `"a piece of rope"`
- `string`
- `foo_bar`
- `date(friday,13)`
- `date(1 => friday, 2 => 13)`
- `freddy(nails => long,face => ugly)`
- `[this,is,a,list]`
- `cons(this,cons(too,[]))`

Sorts

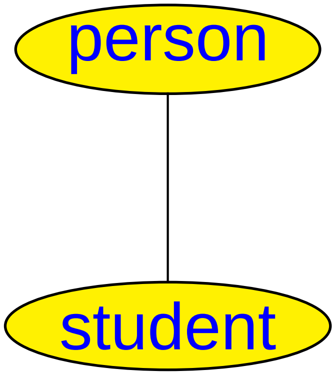
Sorts are the data constructors of LIFE.

Sorts are partially ordered by `<|` in a sort hierarchy.

For example, declaring:

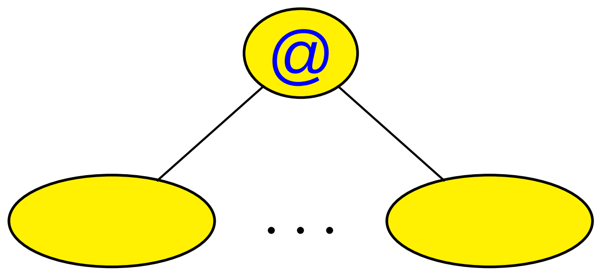
```
student <| person.
```

augments the hierarchy with:

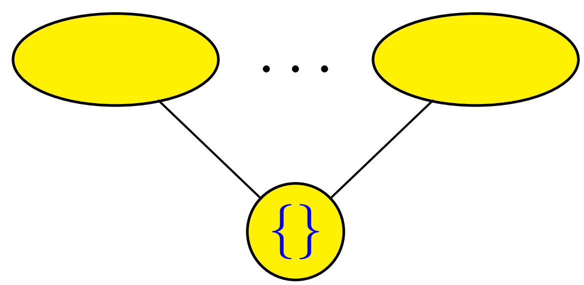


Sorts

@ is the most general sort ($\top$):

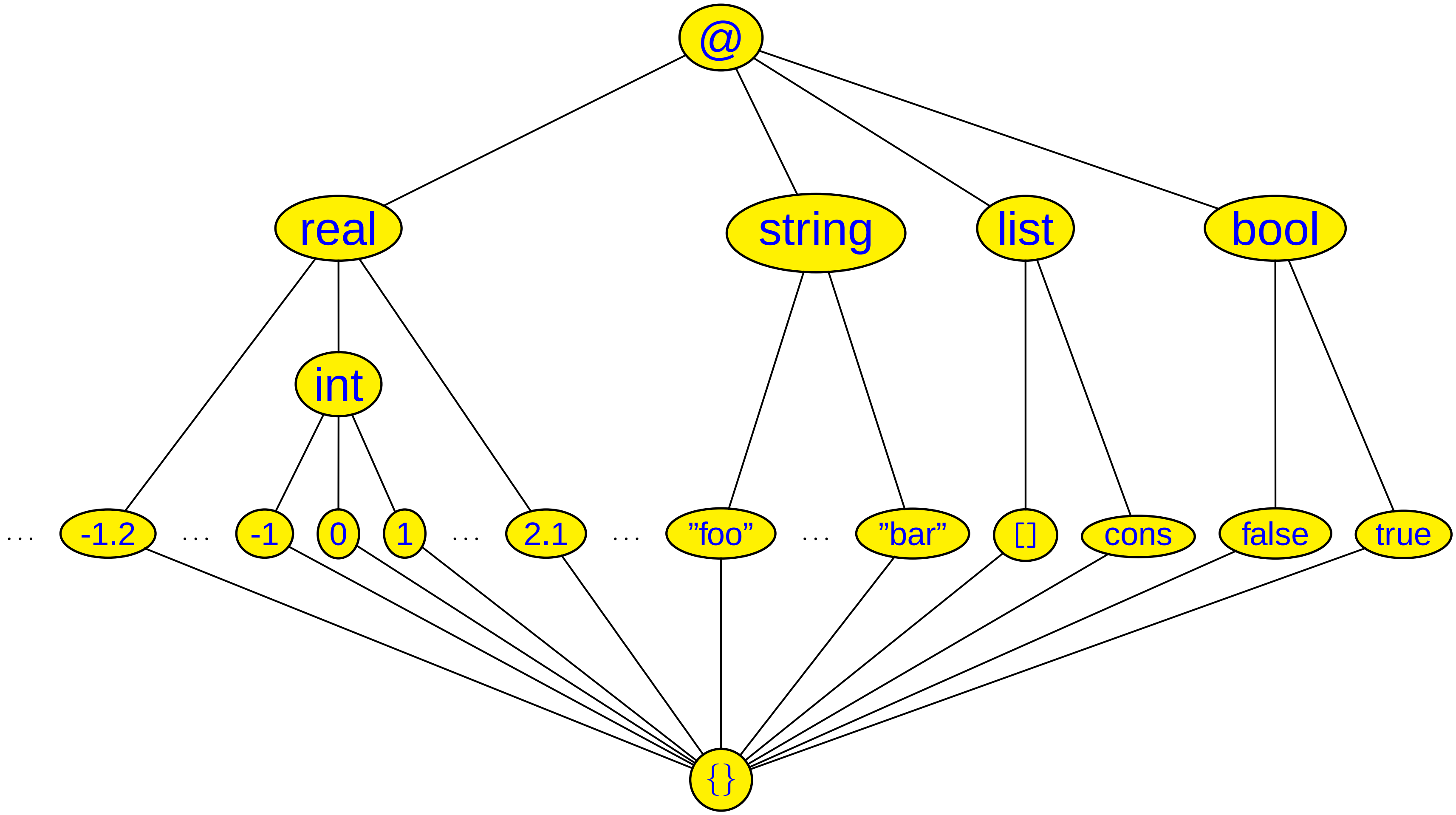


{} is the least sort ($\perp$):



Values are sorts like all others.

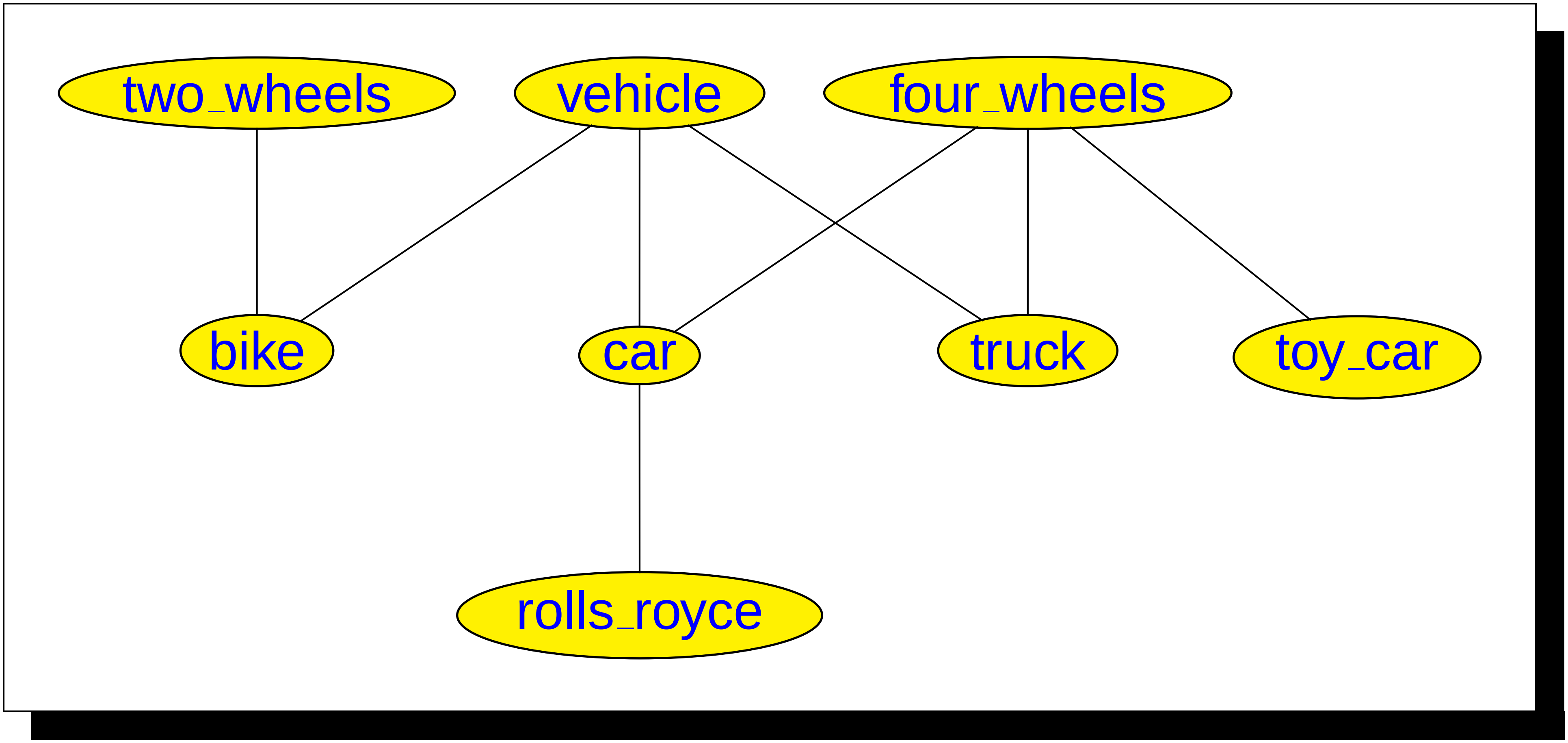
LIFE's built-in sorts



Sort intersection

```
bike        <| two_wheels.
bike        <| vehicle.
truck       <| four_wheels.
truck       <| vehicle.
car         <| four_wheels.
car         <| vehicle.
toy_car     <| four_wheels.
rolls_royce <| car.
```

Sort intersection



Sort intersection

- $\mathtt{two\_wheels} \wedge \mathtt{vehicle} = \mathtt{bike}$
- $\mathtt{four\_wheels} \wedge \mathtt{vehicle} = \{\mathtt{car}; \mathtt{truck}\}$
- $\mathtt{two\_wheels} \wedge \mathtt{four\_wheels} = \perp$
- $\mathtt{rolls\_royce} \wedge \mathtt{car} = \mathtt{rolls\_royce}$
- $\mathtt{truck} \wedge \mathtt{@} = \mathtt{truck}$

Variables as Tags

- Like Prolog's, LIFE's variables start with _ or an upper case letter.
- Unlike Prolog's, LIFE's variables can occur anywhere within terms.
- They are used as reference tags into a ψ-term's structure.
- References may be cyclic: a tag can occur in a ψ-term tagged by it.
- `X:t` denotes a ψ-term `t` tagged by a variable `X`.
- `X` occurring alone is the same as `X:@`.
- `X:t1&t2` is the same as `X=t1, X=t2`.

Disjunctive terms

A disjunctive term is an expression of the form:

$$\{\mathsf{t}_1; \cdots; \mathsf{t}_n\}$$

where $n \geq 0$ and each t_i is either a ψ-term or a disjunctive term.

Disjunctive terms are enumerated by left-right depth-first backtracking, exactly as Prolog's (and LIFE's) predicate level resolution.

Disjunctive terms

- `A={1;2;3}?` behaves like `A=1;A=2;A=3?`

  where `;` means "or" in Edinburgh Prolog syntax.

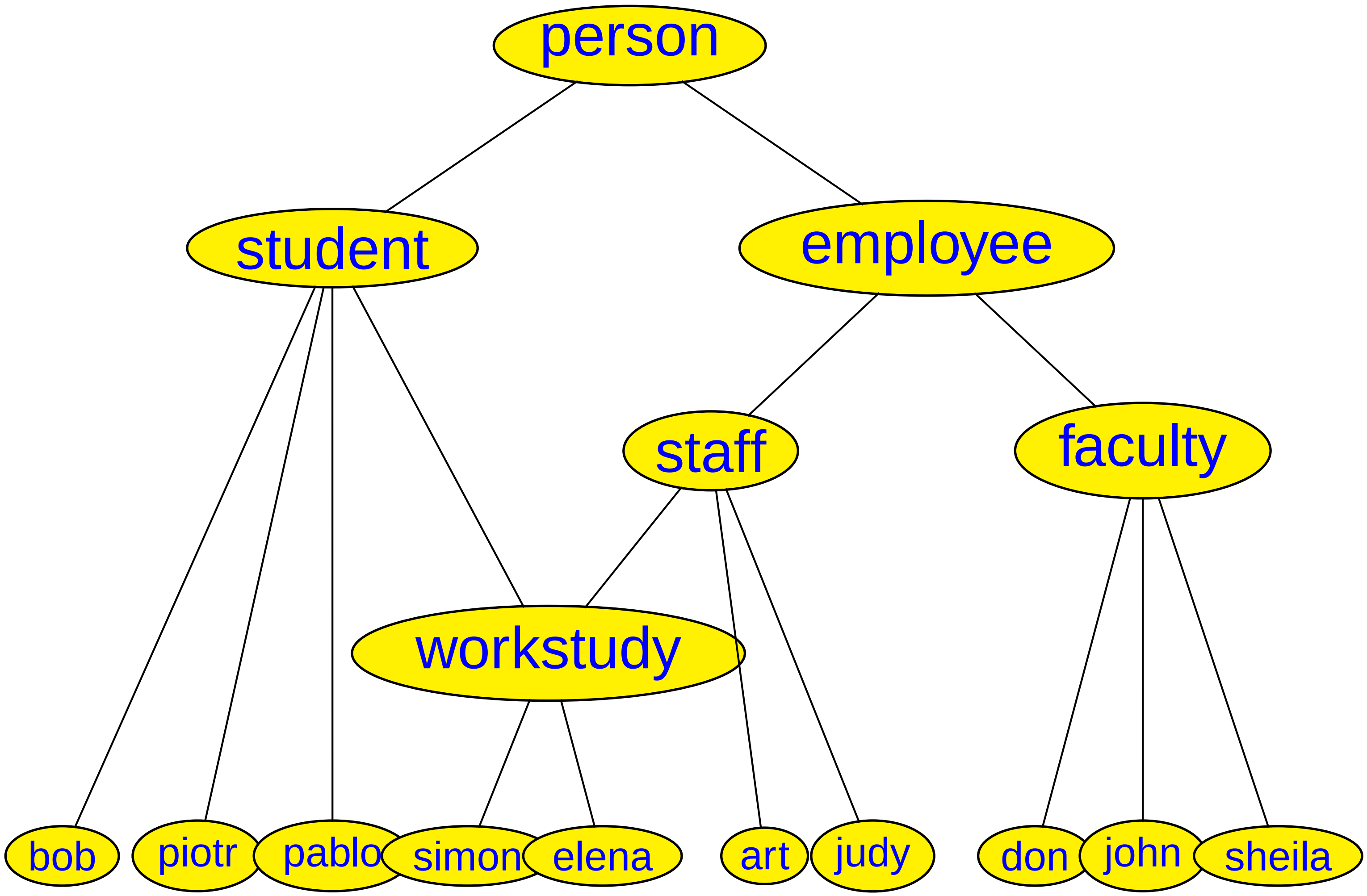
- `p({a;b}).`

  is like asserting `p(a). p(b).`

- `write(vehicle&four_wheels)?`

  prints `car`, then on backtracking will print `truck`.

Ψ-Term Unification



Ψ-Term Unification

```
X = student
     (roommate => person(rep => E:employee),
      advisor  => don(secretary => E)),
```

```
Y = employee
     (advisor  => don(assistant => A),
      roommate => S:student(rep => S),
      helper   => simon(spouse => A)),
```

```
X= Y?
```

Ψ-Term Unification

```
X = workstudy
       (advisor => don(assistant => _A,
                       secretary => _B),
        helper => simon(spouse => _A),
        roommate => _B:workstudy(rep => _B))
Y = X.
```

Predicates

LIFE's predicates are defined as Prolog's, with ψ-terms replacing terms.

Predicates are executed using ψ-term unification.

With the "vehicle" hierarchy, consider the definitions:

```
useful(vehicle).

mobile(four_wheels).

fun(X) :- mobile(X:@(color=>green)),useful(X).
```

Predicates

```
> fun(X)?

*** Yes
X = car(color => green).
--1> ;

*** Yes
X = truck(color => green).
--1> ;

*** No
```

LIFE vs. Prolog

A difference with Prolog is that LIFE terms have no fixed arity.

```
pred(A,B,C) :- write(A,B,C).
```

In (SICStus) Prolog:

```
?- pred(1,2,3).
123
?- pred(A,B,C).
_26_60_94
?- pred(A,B,C,D).
WARNING: predicate 'pred/4' undefined.
?- pred(A,B).
WARNING: predicate 'pred/2' undefined.
```

LIFE vs. Prolog

```
> pred(1,2,3)?
123
*** Yes
> pred(A,B,C)?
@@@
*** Yes
A = @, B = @, C = @.
> pred(A,B,C,D)?
@@@
*** Yes
A = @, B = @, C = @, D = @.
> pred?
@@@
*** Yes
```

User interaction

Interaction with user is more flexible than Prolog's: Once a query is answered, a user can extend it in the current context by entering:

$\langle CR \rangle$	to quit this query and go back to the previous level
;	to force backtracking and look for another answer
a goal	followed by **?** to extend this query
.	to pop to top-level from any depth

User interaction

Example:

```
father(john,harry).
father(john,mike).
father(harry,michael).

grandfather(X,Y) :- father(X,Z),
                    father(Z,Y).
```

User interaction

```
> grandfather(A,B)?
*** Yes
A = john, B = michael.
--1> father(A,C)?
*** Yes
A = john, B = michael, C = harry.
----2> ;
*** Yes
A = john, B = michael, C = mike.
----2> ;
*** No
A = john, B = michael.
```

User interaction

```
--1> father(C,B)?
*** Yes
A = john, B = michael, C = harry.
----2> father(A,C)?
*** Yes
A = john, B = michael, C = harry.
------3>
*** No
A = john, B = michael, C = harry.
----2> .
>
```

Functions

Functions are rewrite rules transforming ψ-terms into ψ-terms.

Function calls use ψ-term matching, NOT unification.

A functional expression may occur anywhere a ψ-term is expected.

```
fact(0) -> 1.
fact(N:int) -> N*fact(N-1).
```

```
> write(fact(5))?
120
*** Yes
```

Residuation

```
> A=fact(B)?
*** Yes
A = @, B = @~.
--1> B=real?
*** Yes
A = @, B = real~.
----2> B=5?
*** Yes
A = 120, B = 5.
```

Residuation

```
------3>
*** No
A = @, B = real~.
----2> A=123?
*** Yes
A = 123, B = real~.
------3> B=6?
*** No
A = 123, B = real~.
------3>
```

Functions

Functions are deterministic—they require no value guessing and no backtracking.

NB: If `foo` and `bar` are non-unifiable, calling:

```
f(foo,bar)
```

will skips a definition such as:

```
f(X,X) -> ...
```

otherwise, it residuates. It will use it only if, and when, the two args are unified by the context.

Functions

Some built-in functions are inverted: *e.g.*, `0=B-C` causes `B` and `C` to be unified.

```
> A = F(B), F = /(2=>A), A = 5?

*** Yes
A = 5, B = 25, F = /(2 => A).
```

Note that here `/` (division) is curryed before being inverted.

Currying

Currying is not the same as residuation, because the result of currying is a function, not $\top$.

In curryed form, `f(a => X,b => Y)` is:

```
f(a => X) & @(b => Y)
```

but also:

```
f(b => Y) & @(a => X)
```

Currying

Arguments may be passed out of order:

```
> f(X,Y,Z) -> [X,Y,Z].
*** Yes
> A=f(a,3 => c)?
*** Yes
A = f(a,3 => c).
--1> A=f(2 => b)?
*** Yes
A = [a,b,c].
```

Functional variables

Functional variables are allowed.

That is, a functional expression may have a variable where a root symbol is expected.

Example:

```
map(F,[]) -> [].
map(F,[H|T]) -> [F(H)|map(F,T)].
```

Functional variables

```
> L=M(F,[1,2,3,4])?
*** Yes
F = @, L = @, M = @~.
--1> M=map?
*** Yes
F = @~~~~, L = [@,@,@,@], M = map.
----2> F= +(2=>1)?
*** Yes
F = +(2 => 1), L = [2,3,4,5], M = map.
------3>
```

Functions

Residuation, currying, and functional variables give functions extreme flexibility:

```
quadruple -> *(2=>4).
pick_arg({5;3;7}).
pick_func({quadruple;fact}).

test :- R=F(A),
        pick_arg(A), pick_func(F),
        write("function ",F," of ",A," is ",R),
        nl, fail.
```

Functions

```
> test?
function *(2 => 4) of 5 is 20
function fact of 5 is 120
function *(2 => 4) of 3 is 12
function fact of 3 is 6
function *(2 => 4) of 7 is 28
function fact of 7 is 5040
*** No
```

Quote and eval

LIFE's functions use eager evaluation. This can be prevented using a quoting operator `.

```
> X =1+2?

*** Yes
X = 3.
--1> Y=`(1+2)?

*** Yes
X = 3, Y = 1 + 2
```

Quote and eval

Dually, a function called `eval` may be used to compute the result of a quoted form.

```
----2> Z=eval(Y)?

*** Yes
X = 3, Y = 1 + 2, Z = 3.
```

Note that `eval` does not modify the quoted form.

Another function called `evalin` works like `eval` but evaluates the expression side-effecting it “in-place.”

Arbitr-Arity (varargs)

In LIFE everything is a ψ-term!

This can be exploited to great benefit to express that some predicates or functions take an unspecified number of arguments.

```
S:sum -> add(features(S),S).

add([H|T],V) -> V.H+add(T,V).
add([],V) -> 0.
```

Arbitr-Arity (varargs)

```
> X = sum(1,2,3,4)?

*** Yes
X = 10.
--1> Y=sum(1,2,3,4,5)?

*** Yes
X = 10, Y = 15.
----2>
```

Constrained sorts

Properties can be attached to sorts: attributes or arbitrary relational or functional dependency constraints. These properties are inherited by subsorts and verified at execution.

```
> :: person(age => int).
*** Yes
> man <| person.
*** Yes
> A=man?
*** Yes
A = man(age => int).
```

Constrained sorts

```
:: vehicle(make => string,
           number_of_wheels => int).

:: car(number_of_wheels => 4).

car <| vehicle.


> X=car?
*** Yes
X = car(make => string,
        number_of_wheels => 4).
--1>
```

Sort definitions

```
man := person(gender => male).
```

is sugaring for:

```
man <| person.
:: man(gender => male).
```

Sort definitions

```
tree := { leaf ; node(left => tree,
                      right => tree) }.
```

is sugaring for:

```
leaf <| tree.
node <| tree.
:: node(left => tree, right => tree).
```

Constrained sorts

```
:: rectangle(long_side => L:real,
             short_side => S:real,
             area => L*S).

square := rectangle(side => S,
                    long_side => S,
                    short_side => S).
```

Constrained sorts

```
> R=rectangle(area => 16, short_side => 4)?
*** Yes
R = rectangle(area => 16,
              long_side => 4,
              short_side => 4).
--1> R=square?
*** Yes
R = square(area => 16,
           long_side => _A: 4,
           short_side => _A,
           side => _A).
```

Constrained sorts

```
:: devout(faith => F, pray_to => X)
 | holy_figure(F,X).

holy_figure(muslim,allah).
holy_figure(jewish,yahveh).
holy_figure(christian,jesus_christ).
```

```
> X=devout?
*** Yes
X = devout(faith => muslim,
           pray_to => allah).
--1> ;
*** Yes
X = devout(faith => jewish,
           pray_to => yahveh).
--1> ;
*** Yes
X = devout(faith => christian,
           pray_to => jesus_christ).
--1> ;
*** No
```

Sorts constraints as impromptu demons

```
> :: I:int | write(I," ").
*** Yes
> A=5*7?
5 7 35
*** Yes
A = 35.
--1> B=fact(5)?
5 1 4 1 3 1 2 1 1 1 0 1 1 2 6 24 120
*** Yes
A = 35, B = 120.
----2>
```

Sorts constraints as impromptu demons

```
> :: C:cons | write(C.1), nl.
*** Yes
> A=[a,b,c,d] ?
d
c
b
a
*** Yes
A = [a,b,c,d].
```

Recursive sorts

Recursive sorts can also be defined. For example, the (built-in) list sort is defined as:

```
list := {[] ; [@|list]}.
```

But there is a safe form of recursion and an unsafe one:

- safe recursion: the recursive occurrence of the sort is in a strictly more specific sort.
- unsafe recursion: the recursive occurrence of the sort is in an equal or more general sort.

Recursive sorts

Example of unsafe recursion:

```
:: person(best_friend => person).
```

This loops for ever...

Need to declare:

```
> delay_check(person)?
```

That will prevent checking the definition of `person` if it has no attributes.

Constrained sorts

```
:: P:person(best_friend => Q:person)
 | get_along(P,Q).
*** Yes
> delay_check(person)?
*** Yes
> cleopatra := person(nose => pretty,
                      occupation => queen).
*** Yes
> julius := person(last_name => caesar).
*** Yes
```

Constrained sorts

```
> get_along(cleopatra,julius).
*** Yes
> A=person?
*** Yes
A = person.
--1> A=@(nose => pretty)?
*** Yes
A = cleopatra(best_friend => julius,
              nose => pretty,
              occupation => queen).
```

Classes and Instances

It is important to relate LIFE's concepts to concepts that are empirically known in O-O programming, like that of class and instance.

Classes are declared by sort definitions:

```
:: class(field1=>value1,
         field2=>value2,
         ...).
```

Like a `struct`, this adds fields to a class definition.

To say that `class1` inherits all properties of `class2`:

```
class1 <| class2.
```

Instances are created by mentioning the class name in the program. For example, executing:

```
> X=foo?
```

creates an instance of the class `foo`. Each mention of `foo` creates a fresh instance. Thus,

```
> X=42, Y=42?
```

creates two different instances of the class `42` in `X` and `Y`. We can do:

```
> X=42, Y=42, X=@(foo => bar), Y=@(foo => buz)?
```

This would not be possible if `X` and `Y` were the same instance.

Classes and Instances

Wild LIFE assumes that mentioning a class name in the program always creates a fresh instance that is different from all other instances of the class.

For example:

```
> X=23, Y=23?
```

creates two different instances of the class 23.

If we have the function defined as:

```
f(A,A) -> hello.
```

then the call `f(X,Y)` will not fire, since `X` and `Y` are different instances.

Classes and Instances

To make `f(X,Y)` fire, `X` and `Y` must be the same instance.

In Wild LIFE, the only way to do this is to unify them explicitly:

```
> X=23, Y=23, X=Y, write(f(X,Y))?
```

will write `hello` (*i.e.*, the function `f` will fire).

Examples of LIFE Programs

Dictionary

```
delay_check(tree)?

:: tree(name => string,
        def => string,
        left => tree,
        right => tree).

contains(tree(name => N,def => D),N,D).
contains(T:tree(name => N),Name,Def)
    :- cond(N $> Name,
            contains(T.left,  Name, Def),
            contains(T.right, Name, Def)).
```

Dictionary

```
test_dictionary :-
        CN = "cat", CD = "furry feline",
        DN = "dog", DD = "furry canine",
        contains(T,CN,CD),  % Insert cat definition
        contains(T,DN,DD),  % Insert dog definition
        contains(T,CN,Def), % Look up cat definition
        nl,write("A ",CN," is a ",Def),nl,!.
```

```
> test_dictionary?
A cat is a furry feline
*** Yes
```

Hamming numbers

```
mult_list(F,N,[H|T]) ->
        cond(R:(F*H) =< N,
             [R|mult_list(F,N,T)],
             []).
merge(L,[]) -> L.
merge([],L) -> L.
merge(L1:[H1|T1],L2:[H2|T2]) ->
        cond(H1 =:= H2,
             [H1|merge(T1,T2)],
             cond(H1 > H2,
                  [H2|merge(L1,T2)],
                  [H1|merge(T1,L2)])).
```

Hamming numbers

```
hamming(N) ->
        S:[1|merge(mult_list(2,N,S),
                   merge(mult_list(3,N,S),
                         mult_list(5,N,S)))].
```

```
> H=hamming(26)?
H = [1,2,3,4,5,6,8,9,10,12,15,16,18,20,24,25]
*** Yes
>
```

Quick Sort

```
q_sort(L,order => O)
   -> undlist(dqsort(L,order => O)).

undlist(X\Y) -> X | Y=[].

dqsort([]) -> L\L.
dqsort([H|T],order => O)
   -> (L1\L2)
    | (Less,More) = split(H,T,([],[]),order => O),
      (L1\[H|L3]) = dqsort(Less,order => O),
      (L3\L2)     = dqsort(More,order => O).
```

```
split(@,[],P) -> P.
split(X,[H|T],(Less,More),order => O)
   -> cond(O(H,X),
           split(X,T,([H|Less],More),order => O),
           split(X,T,(Less,[H|More]),order => O)).
```

```
> L = q_sort([2,1,3],order => <)?
*** Yes
L = [1, 2, 3]
> L = q_sort([2,1,3],order => >)?
*** Yes
L = [3, 2, 1]
```

SEND+MORE=MONEY

```
smm :- % M=0 is uninteresting:
       M=1,
       % Arithmetic constraints:
         C3 + S + M = O + 10*M,
         C2 + E + O = N + 10*C3,
         C1 + N + R = E + 10*C2,
              D + E = Y + 10*C1,
       % Disequality constraints:
         diff_list([S,E,N,D,M,O,R,Y]),
```

SEND+MORE=MONEY

```
% Generate binary digits:
  C1=carry,
  C2=carry,
  C3=carry,
% Generate decimal digits:
  S=decimal, E=decimal,
  N=decimal, D=decimal,
  O=decimal, R=decimal,
  Y=decimal,
```

SEND+MORE=MONEY

```
        % Print the result:
          nl, write(" SEND  ",S,E,N,D), nl,
              write("+MORE +",M,O,R,E), nl,
              write("----- -----"),nl,
              write("MONEY ",M,O,N,E,Y), nl,
        % Fail to iterate:
          fail.

decimal -> {0;1;2;3;4;5;6;7;8;9}.
carry   -> {0;1}.
```

SEND+MORE=MONEY

```
diff_list([]).
diff_list([H|T]) :- generate_diffs(H,T),
                    diff_list(T),
                    H=<9, H>=0.

generate_diffs(H,[]).
generate_diffs(H,[A|T]) :- generate_diffs(H,T),
                           A =\= H.
```

Primes

```
prime :=  P:int | factors(P) = one.

factors(N) -> cond(N < 2, {}, factors_from(N,2)).

factors_from(N:int,P:int) ->
        cond(P*P > N,
             one,
             cond(R:(N/P) =:= floor(R),
                  many,
                  factors_from(N,P + 1))).
```

Primes

```
primes_to(N:int) :-
        write(int_to(N) & prime),
        nl, fail.

int_to(N:int) ->
        cond(N < 1,
             {},
             {1;1 + int_to(N-1)}).
```

Primes

```
> primes_to(20)?
2: prime
3: prime
5: prime
7: prime
11: prime
13: prime
17: prime
19: prime

*** No
>
```

Backtrackable Tag Assignment

The statement `X<-Y` overwrites `X` with `Y`. Backtracking past this statement will restore the original value of `X`.

```
> X=1,write(X),nl, (X <- 2,write(X),nl,fail ; true) ?
1
2
*** Yes
X = 1
```

This is very useful for building “black boxes” that have clean logical behavior when viewed from the outside but that need destructive assignment to be implemented efficiently.

PERT Scheduling

Define the class of task objects:

```
:: A:task (duration      => D:real,
           earlyStart    => early(R),
           lateStart     => {infinity;real},
           prerequisites => R:{[];list} )
 | !, late(A,R).

infinity -> 1e500.
```

This waits until the value is an integer before assigning it:

```
assign(A,B:int) -> succeed | A<-B.
```

PERT Scheduling

Pass 1: Calculate the earliest time when A can start.

```
early([]) -> 0.
early([B|Tasks]) ->
        max(B.earlyStart+B.duration,
            early(Tasks)).
```

PERT Scheduling

Pass 2: Calculate the latest time when A's prerequisites can start and still finish before A starts.

```
late(A,[]) -> succeed.
late(A,[B:task|Tasks])
    -> late(A,Tasks)
     | assign(LSB:(B.lateStart),
              min(LSB, A.earlyStart-B.duration)).
```

PERT Scheduling

A sample input for the PERT scheduler: any permutation of the specified order of tasks would work, illustrating that calculations in LIFE do not depend on order of execution.

```
schedule :-
A1=task(duration=>10),
A2=task(duration=>20),
A3=task(duration=>30),
A4=task(duration=>18,prerequisites=>[A1,A2]),
A5=task(duration=>8 ,prerequisites=>[A2,A3]),
A6=task(duration=>3 ,prerequisites=>[A1,A4]),
A7=task(duration=>4 ,prerequisites=>[A5,A6]),
display_tasks([A1,A2,A3,A4,A5,A6,A7]).
```

```
> schedule?
Task 1: **********
                  ----------
Task 2: ********************
        --------------------
Task 3: ******************************
        ------------------------------
Task 4:                     ******************
                            ------------------
Task 5:                               ********
                                         --------
Task 6:                                       ***
                                              ---
Task 7:                                          ****
                                                 ----
```

Encapsulated programming

Create a routine that behaves like a process with encapsulated data. The caller cannot access the routine's local data except through the access functions ("methods") provided by the routine.

Initialization:

```
new_counter(C) :- counter(C,0).
```

Access predicate:

```
send(X,C) :- C=[X|C2], C<-C2.
```

Encapsulated programming

```
counter([inc|S],V)    -> counter(S,V+1).
counter([set(X)|S],V) -> counter(S,X).
counter([see(X)|S],V) -> counter(S,V) | X=V.
counter([stop|S],V)   -> true
                           | write("Counter stopped.").
counter([],V)         -> true
                           | write("End of counter.").
counter([_|S],V)      -> counter(S,V)
                           | write("Bad message.\n").
```

The internal state of the process is the value of the counter, which is held in the second argument.

Create a new counter object (with initial value 0), increment it twice, and access its value:

```
> new_counter(C)?
*** Yes
C = @~.
--1> send(inc,C)?
*** Yes
C = @~.
----2> send(inc,C)?
*** Yes
C = @~.
------3> send(see(X),C)?
*** Yes
C = @~, X = 2.
--------4>
```

Tiny linguistics

A simple term expansion facility:

```
op(1200,xfx, --> )?
(A --> B) :-
Rule = (gram(A&@(L:[]),In,Out):-expand(B,In,Out,L)),
assert(Rule).

expand((A,B),In,Out,History)
   -> gram(A,In,Out2), expand(B,Out2,Out,H2)
    | History <- [A|H2].

expand(A,In,Out,H) -> gram(A,In,Out) | H <- [A].
```

Tiny linguistics

The main call is:

```
gram(Analysis,Instream,Leftover)
```

```
dynamic(gram)?

gram(A:@(X),[X|T],T) :- X :=< A.

analyse(P) :-
        gram(A,P,[]),
        pretty_write(A),
        nl.
```

Tiny linguistics

A tiny French grammar:

```
phrase --> sujet, verbe_intransitif?
phrase --> sujet, verbe_transitif,
           complement_d_objet ?
phrase --> sujet, pronom, verbe_transitif?
phrase --> sujet, verbe_transitif_indirect,
           complement_d_objet_indirect ?
phrase --> sujet, verbe_etre, adjectif?
```

Tiny linguistics

```
complement_d_objet --> groupe_nominal ?
complement_d_objet_indirect
      --> conjonction, groupe_nominal ?
sujet --> groupe_nominal ?
groupe_nominal --> article, nom_commun?
groupe_nominal --> article, nom_commun,
                   adjectif_postfixe?
groupe_nominal --> article, adjectif_prefixe,
                   nom_commun?
groupe_nominal --> nom_propre?
```

Tiny linguistics

Higher classes of words:

```
adjectif_postfixe <| adjectif.
adjectif_prefixe <| adjectif.
article_indefini <| article.
nom_propre <| etre_anime.
verbe_etre <| verbe_transitif.
```

Tiny linguistics

A lexicon of word sorts:

```
a <| conjonction.
a <| verbe_transitif.
anglais <| adjectif_postfixe.
anglais <| nom_commun.
animal <| etre_anime.
apres <| conjonction.
article <| nom_commun.
belle <| adjectif_prefixe.
belle <| nom_commun.
```

Tiny linguistics

```
blanc <| adjectif_postfixe.
blanche <| adjectif_postfixe.
blanche <| femme. % Special!
...
femme <| personne.
fille <| personne.
francais <| adjectif_postfixe.
francais <| nom_commun.
garcon <| personne_.
```

Tiny linguistics

```
la <| article.
la <| pronom.
le <| article.
le <| pronom.
les <| pronom.
noir <| adjectif_postfixe.
noir <| homme. % Special!
noire <| adjectif_postfixe.
porte <| nom_commun.
porte <| verbe_transitif.
voile <| nom_commun.
voile <| verbe_transitif.
```

Tiny linguistics

```
> analyse([la,femme,blanche,porte,le,voile])?

phrase([sujet([groupe_nominal
                  ([article(la),
                    nom_commun(femme),
                    adjectif_postfixe(blanche)])]),
        verbe_transitif(porte),
        complement_d_objet
            ([groupe_nominal
                  ([article(le),
                    nom_commun(voile)])])])
```

Tiny linguistics

```
> analyse([ted,est,un,noir,blanc])?

phrase([sujet([groupe_nominal([nom_propre(ted)])]),
        verbe_transitif(est),
        complement_d_objet
            ([groupe_nominal
                  ([article(un),
                    nom_commun(noir),
                    adjectif_postfixe(blanc)])])])
```

Tiny linguistics

```
> analyse([ted,est,noir])?

phrase([sujet([groupe_nominal([nom_propre(ted)])]),
        verbe_etre(est),
        adjectif(noir)])
```

Conclusion

LIFE offers conveniences meant to reconcile different programming styles.

It is particularly suited for:

- structured objects
- computational linguistics
- constrained graphics
- expert systems
- ...

More features can be added to complement it with like:

- other CLP constraint solving:
  - arithmetic
  - boolean
  - finite domains
  - intervals
  - . . .
- better language features:
  - extensional sorts
  - partial features
  - lexical scoping
  - method encapsulation
  - compositional inheritance