Programming with

L ogic
I nheritance
F unctions
E quations

(An Introduction)

Hassan Aït-Kaci

Simon Fraser University
Intelligent Sofware Group

Outline

- History
- Generalities
- LIFE's Basic Data Structure: ψ-Terms
- Predicates
- Functions
- Sorts
- Toy Programming Examples
- Conclusion

History

LIFE was originally conceived *ca.* 1986 by Hassan Aït-Kaci and colleagues at MCC, in Austin, Texas.

- Idea:
  Reconcile programming with predicate logic, functions, and structured object inheritance.
- Key:
  Use a universal, simple, but powerful, formal data structure called ψ-term.
- How?
  By solving equational and entailment constraints over order-sorted feature graphs.

History

Still LIFE was the first prototype of LIFE, done at MCC in 1987-88 by David Plummer in Quintus Prolog.

- Idea:
  To experiment with the prototype to have a feel of worth.
- Plus:
  Fun to see it work, surprising convenience, fixed the syntax.
- Minus:
  Incomplete, slow, and MCC proprietary.

History

Wild LIFE is the successor of Still LIFE, done by Richard Meyer at Digital's Paris Research Laboratory in C.

- Idea:
  Full independent reimplementation in C.
- Plus:
  Much more complete, reasonably fast, good user conveniences, convincingly solid to support serious applications.
- Minus:
  Interpreted, big, still incomplete (not by much).

A compiler is currently in the works...

History

Wild LIFE Contributors:

- Richard Meyer
  90% of Wild LIFE, and the compiler
- Peter Van Roy
  the missing 10%, and the compiler
- Bruno Dumant
  graphics toolkit, term expansion, static analyzer
- Jean-Claude Hervé
  basic X interface
- Kathleen Milsted
  LIFE shell
- Andreas Podelski
  theorems, theorems, theorems...
- Hassan Aït-Kaci
  watching the rest work and taking all the credit!

Generalities

LIFE is a generalization of Prolog: most Prolog program 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.*

Except for these differences:

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

Ψ-Terms

- 42
- int
- -5.66
- real
- "a piece of rope"
- string
- foo_bar
- '%* PsyCH(a)oTic**SyMboL!'
- 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.

`@` is the most general sort $(\top)$.

$\{\}$ is the least sort $(\bot)$.

values are sorts like all others.

Variables and Tags

Like Prolog, LIFE's variables start with _ or an upper case letter,

Unlike Prolog, LIFE's variables are not restricted to appear only as leaves of terms.

Thus, variables can be used as (reference) tags within a ψ-term's structure.

They are used as explicit handles for referencing the part of ψ-term they tag.

These references may be cyclic; that is, a variable may occur within a ψ-term tagged by it.

Variables and Tags

Tagging of a ψ-term **t** by a variable **X** is of the form **X:t**.

If a variable occurs not as a ψ-term's tag but as a simple isolated variable, it is implicitly be tagging $\top$, exactly as if it had been written **X:@**.

If the same variable needs to be constrained to be the conjunction of two terms, it is written using the **&** connective, as in **X:t1&t2**. This is equivalent to writing **X=t1, X=t2**.

Disjunctive terms

A disjunctive term is an expression of the form:

```
{t1; ... ;tn}
```

$n \geq$ *0*, where each `ti` is either a ψ-term or a disjunctive term.

In Wild LIFE, disjunctive terms are enumerated using a left-right depth-first backtracking strategy, exactly as Prolog's (and LIFE's!) predicate level resolution.

- **A={1;2;3}?** is like **A=1;A=2;A=3?** where **;** signifies "or" in Edinburgh Prolog syntax.
- **p({a;b}).** is like asserting **p(a). p(b).**
- **write(vehicle&four_wheels)?** first prints **car** then on backtracking will print **truck**.

Backtrackable Tag Assignment

The statement `X<-Y` overwrites `X` with `Y`.

The tags `X` and `Y` reference standard (backtrackable) ψ-terms.

Backtracking past this statement will restore the original value of `X`.

For example:

```
> X=5,(X <- 6;X <- 7),write(X),nl,fail?
6
7
*** No
>
```

This predicate 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.

Sort intersection

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

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

Ψ-Term Unification

```
                        person
                          |
                   ------------------
                  /                  \
           student                   employee
          /       \                  /        \
    -------        \           staff          faculty
   /   |  \         \         /  |  \        /   |   \
bob  piotr pablo     workstudy  art judy   don  john  sheila
                    /         \
                  simon      elena
```

```
student  <| person.
employee <| person.

staff    <| employee.   workstudy <| student.
faculty  <| employee.   workstudy <| staff.

bob    <| student.      don    <| faculty.
piotr  <| student.      john   <| faculty.
pablo  <| student.      sheila <| faculty.

elena  <| workstudy.    art    <| staff.
simon  <| workstudy.    judy   <| staff.
```

Ψ-Term Unification

```
X = student
     (roommate => employee(rep => S),
      advisor  => don(secretary => S)),

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

X= Y?
```

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

Predicates

LIFE's predicates are defined exactly as Prolog's, except that terms are replaced by ψ-terms.

They are executed using ψ-term unification.

```
> truck <| vehicle.

*** Yes
> mobile(vehicle).

*** Yes
> useful(truck).

*** Yes
> mobile(X),useful(X)?

*** Yes
X = truck.
```

Compatibility with 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.
```

Compatibility with Prolog

In Wild LIFE:

```
> pred(1,2,3)?
123
*** Yes
> pred(A,B,C)?
@@@
*** Yes
A = @, B = @, C = @.
--1>
*** No
> pred(A,B,C,D)?
@@@
*** Yes
A = @, B = @, C = @, D = @.
--1>
*** No
> 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 abandon 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.

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.
--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

LIFE's function are defined exactly as rewrite rules transforming ψ-terms into ψ-terms.

They are executed using ψ-term matching, NOT unification.

```
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.
------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---no value guessing nor backtracking.

Calling `f(foo,bar)` skips definition `f(X,X) -> ...` if `foo` and `bar` are non-unifiable; otherwise, it residuates. It will use it only if, and when, the two args are unified by the context.

Arithmetic functions are inverted---*e.g.,* the goal `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)
```

Argument order is irrelevant!

```
> 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.

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

```
> 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,
              " applied to ",A,
              " is ",R),
        nl,
        fail.
```

Functions

```
> test?
function *(2 => 4) applied to 5 is 20
function fact applied to 5 is 120
function *(2 => 4) applied to 3 is 12
function fact applied to 3 is 6
function *(2 => 4) applied to 7 is 28
function fact applied to 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
```

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

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.
```

```
> 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

One can attach properties to sorts: attributes or arbitrary relational or functional dependency constraints.

These properties will be verified during execution, and also inherited by subsorts.

```
> :: person(age => int).
*** Yes
> man <| person.
*** Yes
> A=man?
*** Yes
A = man(age => int).
--1>
```

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>
```

Constrained sorts

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

is sugaring for:

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

```
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).
```

```
> 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).
----2>
```

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
```

Constrained sorts

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>
```

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

Constrained 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.

Constrained sorts

Example of unsafe recursion:

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

This loops for ever...

Temporary workaround (hmm... hack!) is to specify:

```
> 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
> 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.
```

Classes and Instances

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

```
> X=int?
```

creates an instance of the class **int**. Each mention of **int** creates a fresh instance. Therefore, executing:

```
> X=int, Y=int?
```

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

```
> X=int, Y=int, X=56, Y=23?
```

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) -> 1.
```

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 **1** (*i.e.*, the function **f** will fire).

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(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)).
```

SEND+MORE=MONEY

```
solve :-
      % 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]),
      % 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}.

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.
```

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",
% Insert cat definition
        contains(T,CN,CD),
% Insert dog definition
        contains(T,DN,DD),
% Look up cat definition
        contains(T,CN,Def),
        nl,write("A ",CN," is a ",Def),nl,!.
```

```
> test_dictionary?

A cat is a furry feline

*** Yes
```

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_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(30)?
2: prime
3: prime
5: prime
7: prime
11: prime
13: prime
17: prime
19: prime
23: prime
29: prime

*** No
>
```

PERT Scheduling

Define the class of activity objects:

```
:: A:activity( duration => D:real,
               earlyStart => earlyCalc(R),
               lateStart  => {1e500;real},
               prerequisites => R:{[];list} )
 | !, lateCalc(A,R).
```

Wait until the value is an integer before assigning it:

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

PERT Scheduling

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

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

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

```
lateCalc(A,[]) -> succeed.
lateCalc(A,[B:activity|ListOfActs])
    -> lateCalc(A,ListOfActs)
     | 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 activities would work, illustrating that calculations in LIFE do not depend on order of execution.

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

PERT Scheduling

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

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

The counter:

```
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("Counter end-of-stream.").
counter([_|S],V)
        -> counter(S,V)
         | write("Unknown message."), nl.
```

Encapsulated programming

Access to the process is by a logical variable. The internal state of the process is the value of the counter, which is held in the second argument.

```
> 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>
```

This creates a new counter object (with initial value 0) which is accessed through **C**. The counter is incremented twice and then its value is accessed.

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,nl,
        fail.
```

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_.
blanc <| adjectif_postfixe_.
blanche <| adjectif_postfixe_.
blanche <| femme. % Special!
...
femme <| personne.
fille <| personne.
francais <| adjectif_postfixe_.
francais <| nom_commun_.
garcon <| personne_.
```

Tiny linguistics

A lexicon of word sorts:

```
...
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

```
demo :- analyse([la,femme,blanche,porte,le,voile]).
demo :- analyse([richard,est,un,noir,blanc]).
demo :- analyse([richard,est,noir]).
```

```
> demo?
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)])])])

phrase([sujet([groupe_nominal([nom_propre_(richard)])]),
        verbe_transitif_(est),
        complement_d_objet
             ([groupe_nominal
                  ([article_(un),
                    nom_commun_(noir),
                    adjectif_postfixe_(blanc)])])])

phrase([sujet([groupe_nominal([nom_propre_(richard)])]),
        verbe_etre_(est),
        adjectif_(noir)])
```

Conclusion

LIFE is still an experimental language. Nevertheless, it offers conveniences meant to reconcile different programming styles.

It is particularly suited for:

- natural linguistics
- constrained graphics
- expert systems

There are other feature to complement it with like:

- other CLP constraint domains (arithmetic, boolean, finite domains, intervals)
- better language features (extensional sorts, partial features, lexical scoping, method encapsulation, etc...)

This is just a beginning...