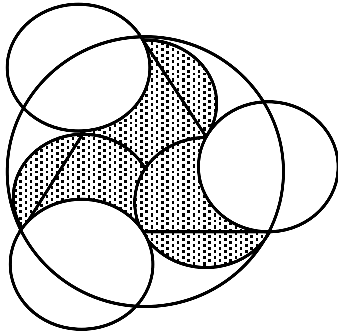


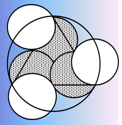
# CLAIRE: un pseudo-code exécutable et compilable pour l'aide à la décision

**18 Février**

EPITA

*Yves Caseau*

*Académie des Technologies*



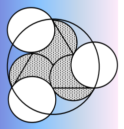
# Plan

- **1ère Partie : Motivations**
  *Genèse d'un pseudo-code exécutable*

- **2ème Partie: Exemples**
  « *The Art of Elegant Programming* » ☺

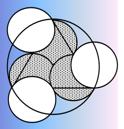
- **3ème Partie: CLAIRE en un coup d'œil**
  *Quelques fonctionnalités qui restent originales et utiles*

- **4ème Partie: 20 ans après, *so what ?***
  *Qu'est-ce qui est réutilisable dans ce projet open-source ?*



# CLAIRE: Spécifications (telles que définie en 94)

- Simple et Lisible
  - pseudo-code exécutable
  - concepts simples et peu nombreux
- Multi-paradigme
  - Objets, Fonctions
  - Règles
  - Versions (exploration d'arbres de recherche)
- compatibilité C++, Open-source
  - génère des objets C++
  - efficacité similaire à C++
  - sources et codes disponibles





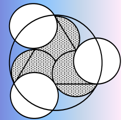
# CLAIRE de 1994 à 2014

## De 1994 à 2004 : 10 ans de RO

- E-Lab (Bouygues), XL Solutions, THALES
- Applications d'optimisation :
  - Ordonnancement, Emplois du temps, Routage
  - Yield Management à TF1 Pub
- Exemple des challenges ROADEF
  - 3 fois finalistes 2001, 2003, 2005
- Utilisé en support de cours à Jussieu/ENS
  - « Contraintes et Algorithmes »
- Introduction de Java comme cible de compilation en 99-2000

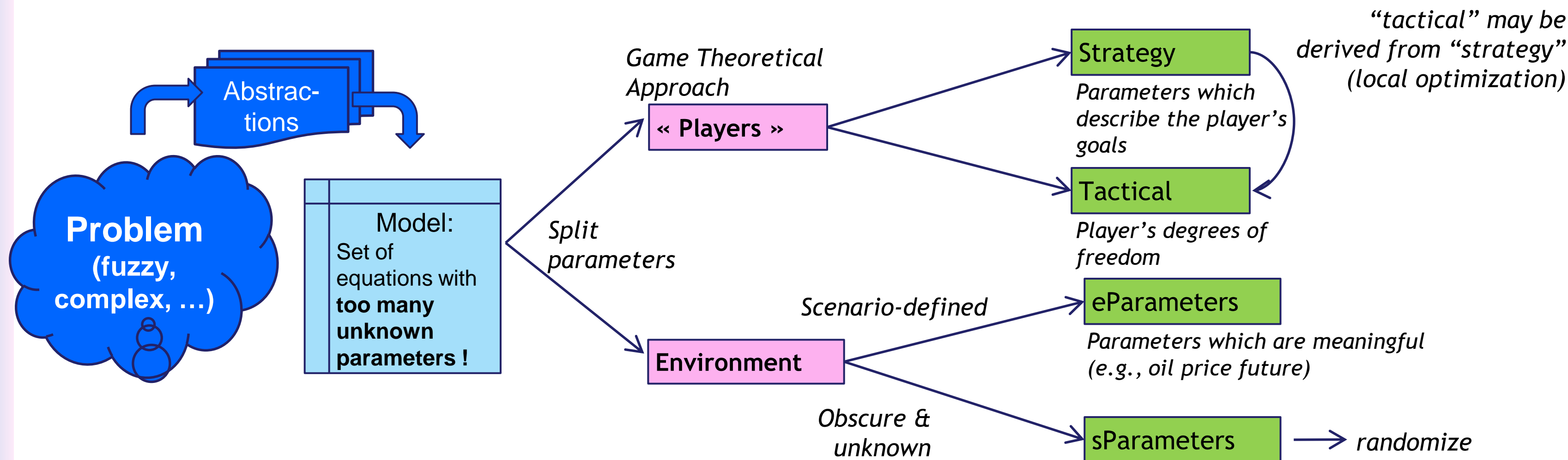
## De 2004 à 2014 : GTES

- Simulation par Jeux et Apprentissage
- Optimisation locale, Théorie des Jeux et Monte-Carlo
- Plateforme pour « *serious games* »

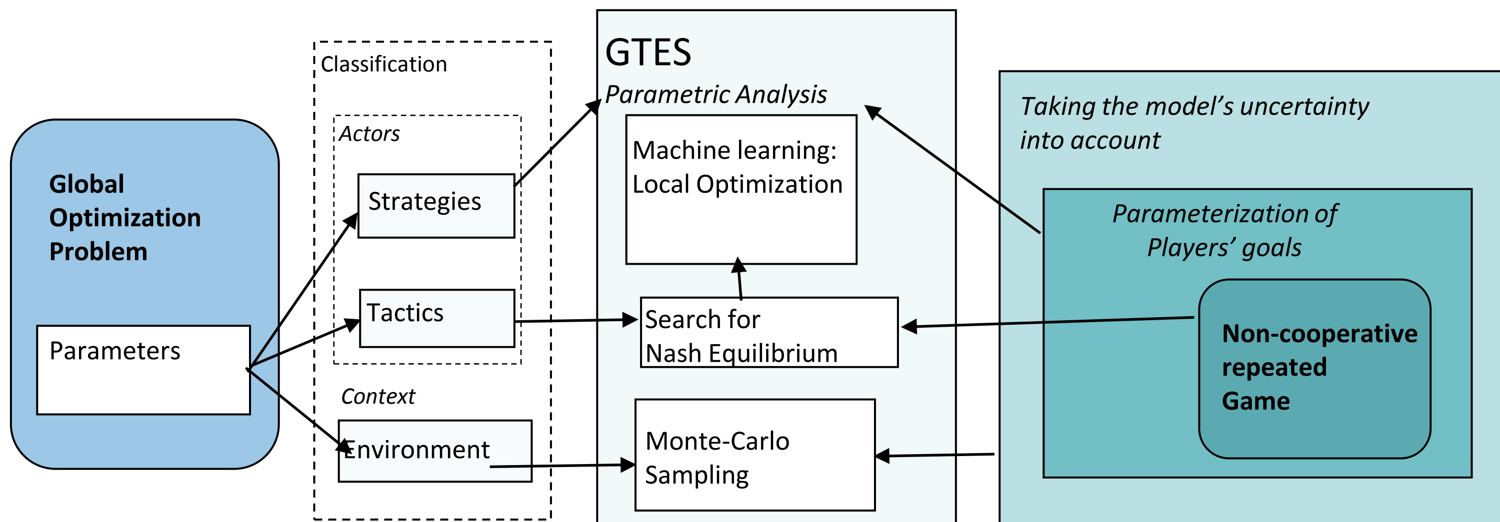


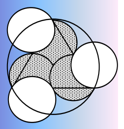
# Game Theoretical Evolutionary Simulation (GTES)

## GTES is a tool for looking at a complex model with too many unknowns




## GTES looks for possible equilibriums in uncertain settings

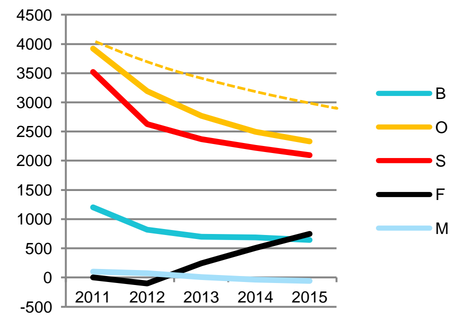






# Applications récentes de CLAIRE

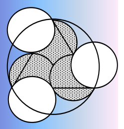
- **Réseaux commerciaux de distribution**
  *Equilibrer commissionnement et fidélisation sur l'ensemble des canaux*




- **CGS: Cellular Game Simulation**
  Simulation de l'arrivée de Free dans un marché à 3 acteurs ☺ - cf. CSDM 2012

- **S3G: Systemic Simulation of SmartGrids**
  *Comprendre les avantages/inconvénients d'une approche distribuée vs. centralisée*

- **Licences LTE**
  *choisir la bonne stratégie de portefeuille pour les enchères à un tour des fréquences LTE fin 2012*





# CLAIRE en tant que système

*Domaines*

Stochastic

RO

GTES

Algorithmes

Langage Elégant (CLAIRE)

Interprète/Debug / Trace / etc. Reader

Méta/Réflexif CLAIRE en CLAIRE Core / Language

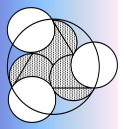
C++ Kernel (LISP en C++) Kernel

Alloc / GC / list / sets / funcall / ...

Optimiseur de code Optimize

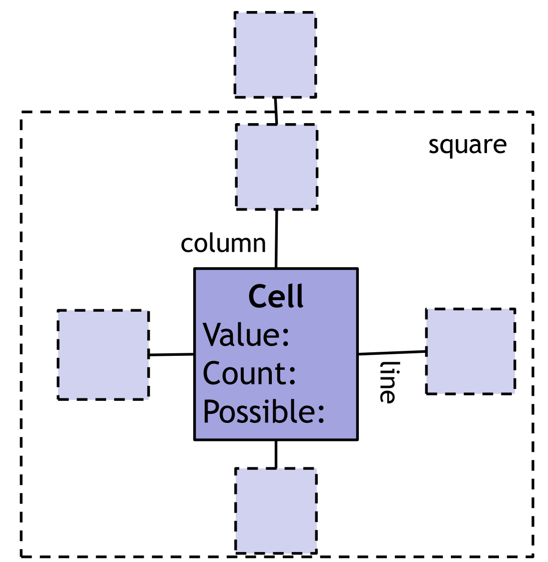
Générateur de code Generate

*Exécutables*

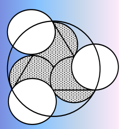


2ème Partie : Exemples

# Sudoku (I)

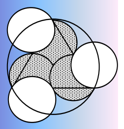



```
// finds a cell with a min count (naive heuristic)
findPivot(g:Grid) : any
  -> let minv := 10, cmin := unknown in
       (for c in g.cells
          (if (c.value = 0 & c.count < minv)
              (minv := c.count, cmin := c)),
        cmin)

// solve a sudoku : branch on possible values using a recursive function
// branch(...) does all the work :)
solve(g:Grid) : boolean
  -> when c := findPivot(g) in
       exists(v in (1 .. 9) |
              (if c.possible[v] branch((c.value := v, solve(g)))
               else false))
     else true
```



# Sudoku (II)

```
// first propagation rule
r1() :: rule(
    c.value := v => (store(c.line.counts,v,0),        // disable counts[v] since v was found !
                     store(c.column.counts,v,0),
                     store(c.square.counts,v,0),
                     for v2 in (1 .. 9)                      // for all values v2 that were still OK
                          (if (v != v2 & c.possible[v2]) noLonger(c,v2),
                     for c2 in (c.line.cells but c) forbid(c2,v),       // v is used for c.line
                     for c2 in (c.column.cells but c) forbid(c2,v),   // … and c.column
                     for c2 in (c.square.cells but c) forbid(c2,v))))  // … and c.square

// second rule : if c.count = 1, the only possible value is certain
r2() :: rule(
    c.count := y & y = 1 =>  c.value := some(y in (1 .. 9) | c.possible[y]))

// third rule (uses the CellSetSupport event) :
// if a value v is possible only in one cell, it is certain
r3() :: rule(
    updateCount(cs,v) & cs.counts[v] <= 1
    => when c := some(c in cs.cells | c.value = 0 & c.possible[v]) in c.value := v
        else contradiction!())
```

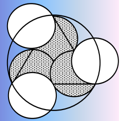


2ème Partie : Exemples

# Configuration : Exemple à base de règle

- compatibility1() :: rule(
  st.speaker :add sp & not(sp.ohms % st.amplifier.ohms) )
  => technical_problem(s = "conflict speakers-amp"))
- compatibility2() :: rule(
  st.sources :add x & size(st.sources) > st.amp.inputs
  => technical_problem(s = "too many sources"))
- compatibility3() :: rule(
  (st.out :add x) & x.maxpower < st.amp.power
  => technical_problem(s = "amp too strong for the speakers"))

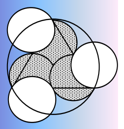
- my_system :: stereo(amp = amp1)
  (exists(sp in speaker |
  (try (my_system.out :add sp, true)
  catch technical_problem
  (//[0] rejects ~S because ~A // sp, exception!().s,
  my_system.out :delete sp,
  false))),



2ème Partie : Exemples

# Exemple: Algorithme Hongrois (I)

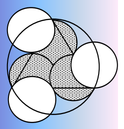
```
// builds a maximum weight complete matching
match()
  -> (...,      // initialization
      while (HN != N)   (if not(grow())  dual_change()))

// a step repeats adding nodes until the forest is hungarian (return value is false)
// or the matching is improved (return value is true)
// explore is the stack of even nodes that have net been explored yet
grow() : boolean
  -> let i := pop(explore) in
      (  exists( j in {j in GpiSet(i,LastExplored[i] + 1,LastValid[i]) | not(odd?[j])} |
                 (if  (sol-[j] != 0)
                      (//[SPEAK] grow: add (~S,~S) to forest// i,j,
                       odd?[j] := true,
                       pushEven+(sol-[j]),
                       tree[j] := i,
                       false)
                  else (augment(i,j), true)))  |
        (if (explore[0] != 0) grow())    )
```





# Exemple: Algorithme Hongrois (II)

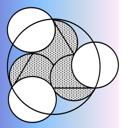
```
// change the dual feasible solution, throw a contradiction if there are no perfect matching
dual_change() : integer
 -> let e := min( list{vclose[i] | i in {j in Dom | not(odd?[j])}}) in
      (//[SPEAK] DUAL CHANGE: we pick epsilon = ~S // e,
       if (e = NMAX) contradiction!(),
       for k in stack(even)     (pi+[k] :+ e, LastExplored[k] := LastValid[k]),
       for j in {j in Dom | odd?[j]}   (pi-[j] :- e, vclose[j] := NMAX)),
       clear(explore),
       for i in stack(even)
          let l := Gpi[i], k := size(l), toExplore := false in
            (while (LastValid[i] < k) (k, toExplore) := reduceStep(i,j,l,k,toexplore),
             if toExplore push(explore,i)))

// look at edges outside the valid set one at a time
reduceStep(i:Dom,j:Dom,l:list,k:integer,toExplore:boolean) : tuple(integer,boolean)
 -> let j := l[k], c := Cpi(i,j) in
           (if (c = 0)  (//[SPEAK] dual_change: Add edge ~S,~S // i,j,
                          Gpiadd(l,i,j,k), toexplore := true)
            else (vclose[j] :min c, k :- 1)),
            list(k, toexplore))
```



# CLAIRE d'un coup d'oeil

3ème Partie : Exemples

- Un langage objet fonctionnel polymorphe
  - compilé et interprété, simple et efficace
- Un langage de modélisation (ensembles, relations)
  - haut niveau d'abstraction, confort d'utilisation
- Un langage de « patterns » (issus de la modélisation)
  - En particulier, facilite la programmation « par ensembles »
- Outils pour la recherche arborescente
  - points de choix et retours arrière



# Objets

- Une hiérarchie de classes avec héritage simple

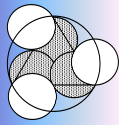
```
point <: object(x:integer, y:integer)
```

- Classes paramétrées

```
stack[of] <: thing(of:type, contents:list)
```

- Une sur-hiérarchie de types complexes avec héritage multiple

```
(1 .. 10) U (20 .. 30) ≤ (1 .. 30) ≤ integer
stack[of = integer] ≤ stack[of:subtype[integer]]
                    ≤ stack
tuple(integer,integer)       ≤ list[integer]
                             ≤ list[integer U float]
```



# Polymorphisme

- Surcharge complètement libre

```
f(x:{o},y:(1 .. 12)) -> 1
f(x:(0 .. 10), y:integer) -> (x + y)
f(x:integer,y:({1,2} U (7 .. 10)) -> (x - y)
```

- Typage et liaisons statiques et dynamiques

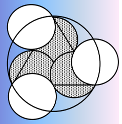
- génération de code

```
sum(s:set[integer]) : integer
   => let d := 0 in (for x in s d :+ x, d)
```

`sum(1 .. 10)` devient

```
let d := 0, m := 10, x := 1 in
    (while (x <= m) (d :+ x, x :+ 1), d)
```

- Polymorphisme de composition
  - Exploite les règles du type f(g(x)) = h(x)
  - exemple : det(A * B) = det(A) * det(B)



# Relations

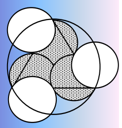
- Relations binaires … étendues

```
comment[c:class] : string := " "
dist[x:(0 .. 100),y:(0 .. 100)] : integer := 0
```

- Modélisation

```
meet[s:set[person]] : date := unknown
course[tuple(person,set[person])] : room := unknown
```

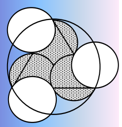
- Gestion des relations inverses
- Gestion de `unknown` (absence de valeur)
- Règles de propagation sur événement
  - Nouvelle valeur ou ajout d’une valeur (relation multi-valuée)
  - `R1() :: rule(x.salary := (y -> z) => x.tax :+ (z-y) * 0.1)`



# Raisonnement Hypothétique

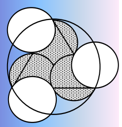
- Mondes
  - `choice()` crée un point de choix
  - `backtrack()` crée un retour arrière
  - `commit()` entérine les *updates* depuis le dernier point de choix
- Recherche arborescente
  - Fonctionnement en pile, optimisé

```
solve() : boolean ->
 when q := pick() in
   exists( c in possible(q) |
           branch(  (column[q] := c, solve())))
 else true

branch(e) :: (choice(),
               ( (try e catch contradiction false) |
                 (backtrack(), false) ))
```



# Ensembles Concrets et Abstraits

- Ensembles en extension
  - classes `for x in person print(x),  size(person)`
  - sets, lists `set(1,2,3,4),    list(Peter, Paul, Mary)`
  - bibliothèque de structures de données (Bitvectors, Hset, ...)
- Types de données
  - Intervalles `1 .. n,`
  - Union, Intersection `array U property, list ^ subytpe[char]`
  - Types paramétrés `for x in person[age:(15 .. 18)] print(x)`
- Expressions ensemblistes
  - image
    `{age(x) | x in person}, list{i + 1 | i in (1 .. 10)}`
  - sélection
    `{x in person | x.age > 0}, list{i in (1 .. n) | f(i) > 0}`



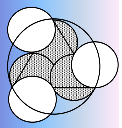
# Extensibilité

- Classes
  - L'ensemble des structures de données représentant un ensemble est extensible

```
Hset[of] <: set_class(of:type,content:list,index:integer)

add(s:Hset[X], y:X) : void
   -> let i := hash(s.content,y) in ....
set!(s:Hset) -> {x in s.content | known?(x)}
```

- Patterns
  - un pattern est un appel fonctionnel qui peut être traité de façon paresseuse

```
but(x:abstract_set,y:any) : set -> {z in x | z != y}

%(x:any, y:but[tuple(abstract_set,any)])
    => (x % y.args[1] & x != y.args(2))
```



# Itération Explicite

- Itération paresseuse en fonction du type d'ensemble

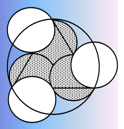
```
for x in person print(x)
for y in (1 .. n) f(y)
```

- Génération de code source

```
for c in person.descendent
    for x in c.instances print(x)
let y := 1 in
   (while (y <= n) (f(y), y :+ 1))
```

- Itération extensible

```
iterate(x:Hset,v:Variable,e:any)
  => for v in x.content (if known?(v) e)
iterate(x:but[tuple(abstract_set,any)],v:Variable,e:any)
  => for v in x.args[1] (if (v != x.args[2]) e)
```



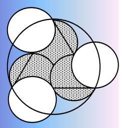
# Itération Implicite

- Chaque expression implique une itération

```
{x in (1 .. 10) | f(x) > 0}
{length(c.subclass) | c in (class but class)}

let s := {}, x := 1 in
   (while (x <= 10) (if (f(x) > 0) s :add x, x :+ 1),
    s)
```

- L'itération d'une expression est aussi paresseuse

```
for x in {x in (1 .. 10) | f(x) > 0} print(x)

for y in {c.slots | c in (class \ relation.ancestors)}
  print(y)

for c in class.instances
   if not(c % relation.ancestors) print(c.slots)
```

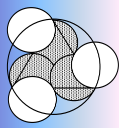


# Itérateurs de Structures

- La combinaison d'itérateurs et de patterns permet d'itérer des structures plus complexes

```
Tree <: object(value:any,right:Tree,left:Tree)
TreeIterator <: object(tosee:list, status:boolean)
iterate(x:by[tuple(Tree,TreeIterator)], v:Variable, e:any)
  =>  let v := start(x.args[2], x.args[1]) in
           while (v != unknown)
                (e, v := next(x.args[2], x.args[1])

TreeIteratorDFS <: TreeIterator()
start(x:TreeIteratorDFS, y:Tree) -> ...
next(x:TreeIteratorDFS, y:Tree) -> ...
DFS :: TreeIteratorDFS()

TreeIteratorBFS <: TreeIterator() ...
```

```
for x in (myTree by DFS) print(x)
{y.weight | y in (myTree by BFS)}
```



# Application: Listes Chaînées

- La représentation la plus efficace utilise des slots

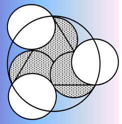
```
Task <: object( ....
               next:Task, prev:Task, ....)
```

- L'utilisation de patterns permet de créer des listes virtuelles

```
chain(x:Task) : list[Task]
    -> let l := list(x), in
         (while known?(next,x) (x := x.next, l :add x),
          l)
insert(x:Task,y:list[Task]) => ...
iterate(x:chain[tuple(Task)],v:Variable,e:any) => ...
```

- Ces listes peuvent être utilisées comme des listes normales

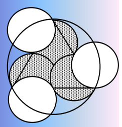
```
count(chain(t1)),
sum({x.weight | x in chain(t0)}),
 ...
```



# Paramétrisation

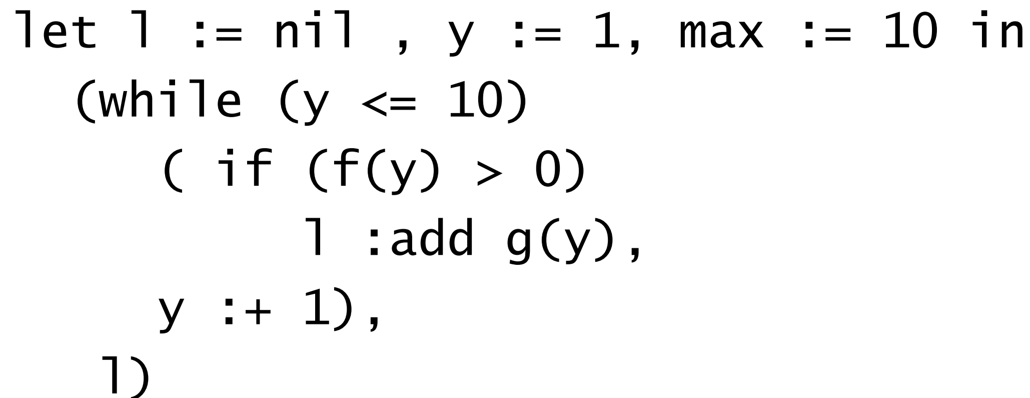
```
[iterate(s:Interval, v:Variable, e:any)
  => for v in users(use(s))  (if SET(v)[index(v)] e)   ]

[<atleast(x:Task, y:Task) => atleast(x) <= atleast(y) ]

[min(s:any, f:property, default:any) : any
   => let x := default in
         (for y in s (if f(x,y) x := y),  x) ]
```

```
min(i but t, <atleast, TEnd)

let x := TEnd in
    (for y in users(use(i))
       (if SET(i)[index(y)]
          (if (y != t)
              (if (atleast(y) <= atleast(x)) x := y))),
     x)
```

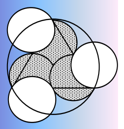


# Génération de Code

- CLAIRE génère un code C++ lisible
- La compilation du noyau objet est directe, sans surcoûts
- La compilation des expressions ensemblistes est optimisée



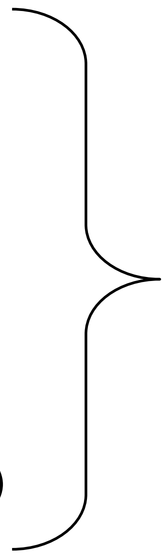
```
list{g(x) | x in {i in (1 .. 10) | f(i) > 0}}

let l := nil , y := 1, max := 10 in
  (while (y <= 10)
     ( if (f(y) > 0)
          l :add g(y),
     y :+ 1),
   l)
```





# CLAIRE 20 ans après : que reste-t-il ?

1. **Un pseudo-code exécutable**
   **… too late ☺**
2. **Noyau « Lisp en C++ »**
   **… bof ☹**
3. **Interprète réflexif**
   **… réutilisable**
4. **Compilateur extensible**
   **… CLAIRE to X ?**
5. **Pas mal de code …**
   **(algorithmes optimisation)**

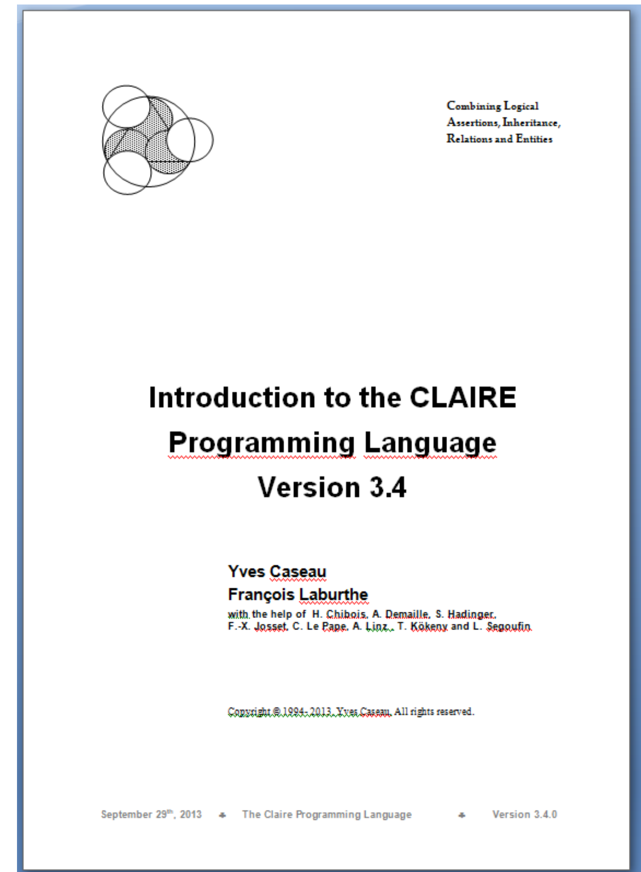


Mais un contexte « favorable » :

1. Retour en force de l'Intelligence Artificielle à toutes les échelles (du *cloud* au Javascript)
2. « Comportement intelligent » donc « *problem solving* » & « optimisation combinatoire »
3. Montée des « jeux » comme paradigme d'interaction

# Comment jouer avec CLAIRE ?

- Site : http://claire3.free.fr/
- Documentation (95 pages)
- GitHub: http://github.com/ycaseau/CLAIRE3.4
- Top level:
  *CLAIRE est en premier lieu un langage interprété, héritier de LISP ☺*
- C++ compiler
  - Version Windows maintenue / à jour
  - Version G++ un peu poussiéreuse …
- Page Facebook CLAIRE ☺

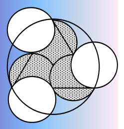

Combining Logical Assertions, Inheritance, Relations and Entities

Introduction to the CLAIRE Programming Language Version 3.4

Yves Caseau
François Laburthe
with the help of H. Chibois, A. Demaille, S. Hadinger, F.-X. Josset, C. Le Pape, A. Linz, T. Kökeny and L. Segoufin

Copyright © 1994- 2013, Yves Caseau. All rights reserved.

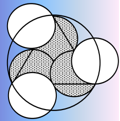
September 29th, 2013 ♣ The Claire Programming Language ♣ Version 3.4.0





# CLAIRE comme plateforme d'expérimentation

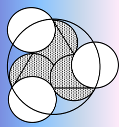
- CLAIRE est entièrement réflexif
  - Tout est objet (à la SMALLTALK)
  - Toute expression du langage est représentée par un objet (à la LISP)
  - L'environnement :
    - toplevel = eval(read()), debug, trace, profile, inspect, …
- Première utilisation : plate-forme de prototypage pour un nouveau langage
  - Extensibilité & capacité à tout redéfinir
  - Par exemple, ajouter du parallélisme ….
- Deuxième utilisation : massacre à la tronçonneuse ☺
  - Interprète langage fonctionnel
  - Système de types (treillis complet avec union/intersection)
  - Mécanismes d'exploration arborescente (choice/backtrack)



# CLAIRE comme générateur de DSL

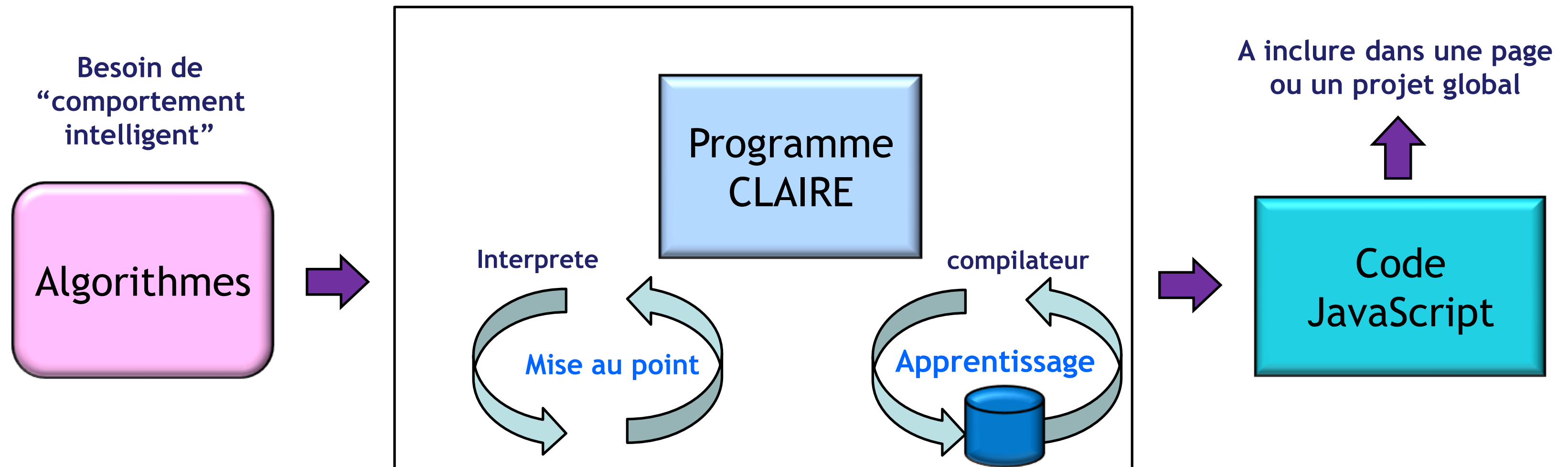


- La syntaxe de CLAIRE est facilement extensible
  - Comme en LISP, le « *reader* » est paramétrable
  - Entièrement en CLAIRE (il a existé une version lex/yacc)
- Beaucoup de savoir-faire dans l'optimiseur de code
  - Inférence de type par interprétation abstraite
  - Optimisation de nombreuses propriétés (allocation mémoire)
  - Gestion de l'abstraction et du polymorphisme (cf. 3e partie)
  - Optimisation des « patterns »
- Générateur de code source en deux étapes
  - Un étage générique, indépendant du code source
  - Un « *producer* » séparé pour C++ ou Java
- Générateur de Java disponible sur demande
  - Prochaine étape : « *producer* » pour JavaScript



# Conclusion

- CLAIRE comme plateforme de mise au point de “modules intelligents” pour les UI de demain :




- Mon principal objectif pour les 5 ans à venir
  - Mettre GTES à disposition en tant que “*framework for serious games*”
  - A venir : S3G, CGS, RTMS, GWDG, SIFOA
  - Cf. mes deux blogs: BDIS & Architecture Organisationnelle
  - Livre en préparation pour 2017 ☺