

# Program Understanding: a teaching experience

Harald Wertz  
Département Informatique  
Université Paris 8

[hw@ai.univ-paris8.fr](mailto:hw@ai.univ-paris8.fr)

# Goals:

- explore basic technics of program understanding
- learn technics for program analysis, observation, manipulation and representation
- introduce some tools for program understanding, analysis, manipulation and verification
- in general: learn to consider programs as physical objects of study (like molecules and galaxies)
- Develop a « physical » theory of programs

# Environment

- *Level*: Licence – Maîtrise (all with previous programming knowledge: Lisp, C, Smalltalk, Prolog)
- *number of students*: 15
- *Total time*: 37:30h (15\* 2:30h)
- paper copies of programs handled in class, availability of programs and documentations on the net, availability of standard tools on department computers

# Domains Covered

## Theoretical Preliminaries:

- Induction techniques (simple, structural, fixpoint)
- Program transformation techniques (pattern rewriting, Burstall/Darlington transformations)
- Program verification techniques (inductive, axiomatic, path-expressions, Boyer-Moore)
- Symbolic execution/evaluation
- Clichés, Plans and Patterns
- Control- and Data-Flow analysis/construction
- Generalities about use/produce, dependencies, etc

# Programs studied

Mainly 4 small programs:

1. 1 page Lisp program **foo** completing arithmetic series
2. Construction of a small Lisp program
3. 4 page Lisp program **VCG**
4. 4 page Smalltalk program (**MemoFib**)

# Program 1: **foo**

- *What:* My adaptation of an adaptation by Jean-François Perrot of an adaptation by Olivier Danvy of an adaptation by Patrick Greussay of a pedagogical adaptation of CMU of an AI-program developed by C. Hedrick at SRI in 1972 completing arithmetic series. Example: (1 1 2 3 3 5 4 7 x x x x x x x x x)  $\Rightarrow$  (1 1 2 3 3 5 4 7 5 9 6 11 7 13 8 15 9)
- *Size and form:* One page of LISP code where most identifiers are intentionally rendered meaningless.
- *Structure:* Composed of 8 Lisp functions, conceptually grouped in 3+3+2 functions, a group of 3 functions for initialization and global control, another of 3 for analysis and a group of 2 functions for calculating the missing members
- *Approach:* handout of listing asking « what does this program do? »

# program **foo** (in CommonLisp)

```
(defun foo (s)
  (let ((max (nc s)))
    (foo2 1))
  'ok)
```

```
(defun foo2 (pas)
  (unless (> pas max)
    (bar pas ())
    (serie2 (1+ pas))))
```

```
(defun bar (x sol)
  (if (= x 0)
      (im s sol pas)
      (bar (1- x)
           (cons (kamm(nthcdr (1- x) s) pas) sol))))
```

```
(defun kamm (s pas)
  (when (numberp (nth pas s))
    (kamm2 (- (nth pas s)(car s))(nthcdr pas s))))
```

```

(defun kamm2 (inc s)(cond
  ((null (and s (numberp (nth pas s)))) inc)
  ((= inc (- (nth pas s)(car s)))
   (kamm2 inc (nthcdr pas s))))))

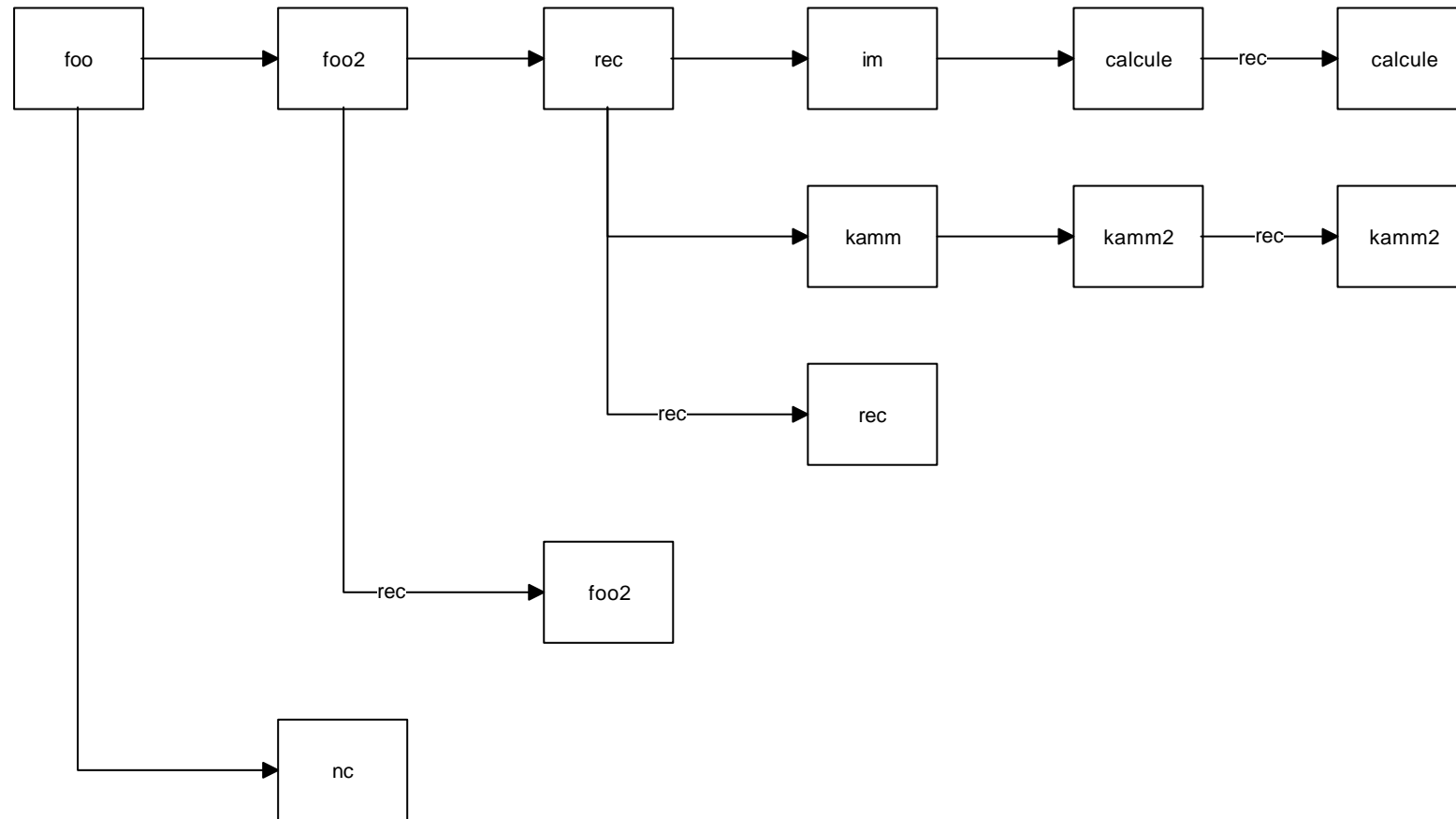
(defun im (l sol pas)
  (unless (member nil sol)
    (let ((lsol (length sol)) (ll (length l)))
      (rplacd (last sol) sol)
      (format T "~%Succes ~A" (cal 0 sol)))))

(defun cal (n sol)
  (unless (>= n ll)
    (cons (+ (nth (\\ n lsol) l)
             (* (truncate(/ n pas))(car sol)))
          (cal (1+ n) (cdr sol)))))

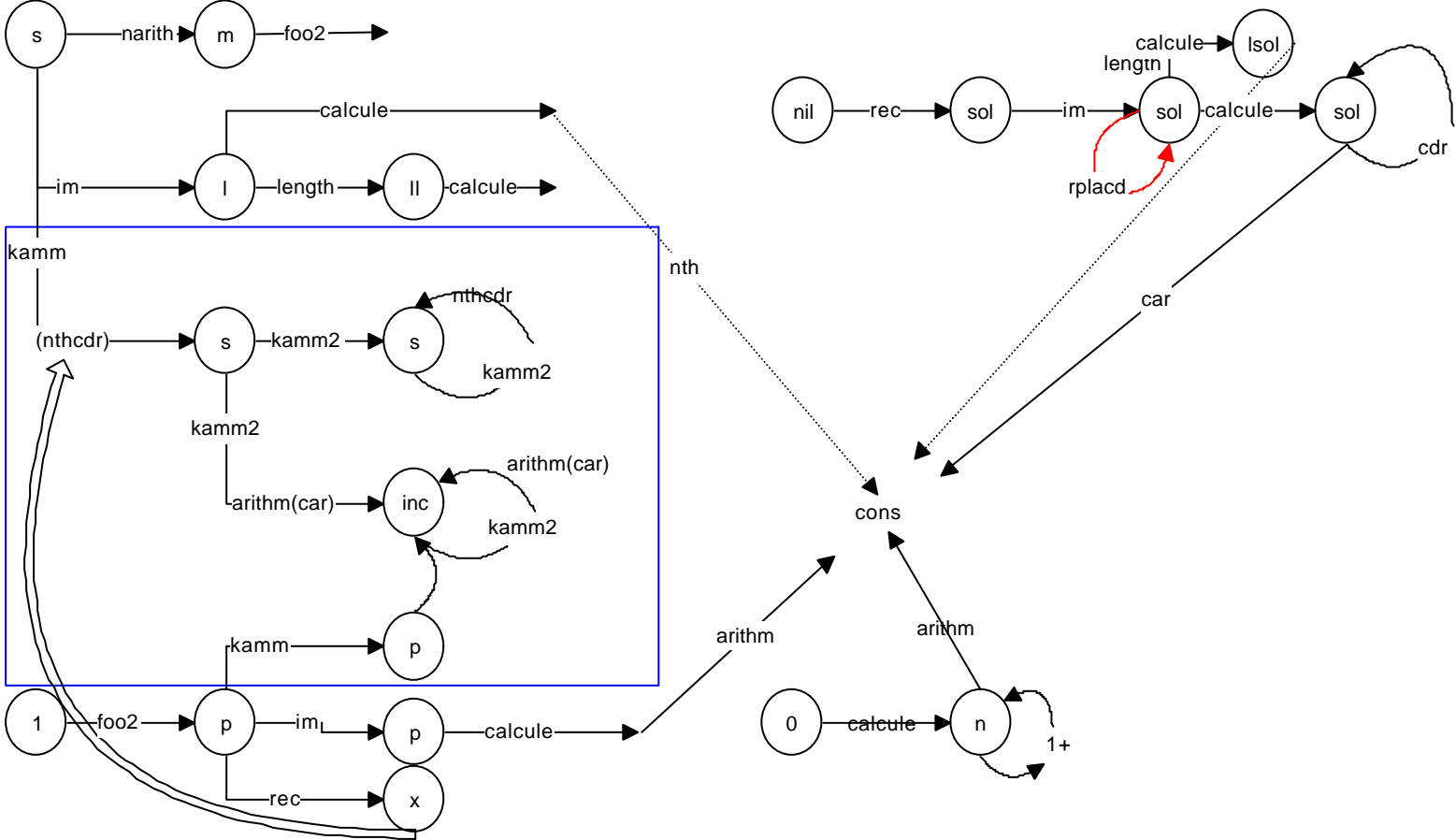
(defun nc (l)
  (if (and l (numberp (car l)))
      (1+ (nc (cdr l))) 0))

```

# Callgraph of `foo`



# A view of the Dataflow of foo



## Some example invocations

solving (1 1 1 X X X)

Succes (1 1 1 1 1 1) ; 1 step

solving (1 2 1 2 X X X) ; 2 steps

Succes (1 2 1 2 1 2 1) pas 2

solving (1 3 2 4 3 5 X X X X X X X) ; alternance

Succes (1 3 2 4 3 5 4 6 5 7 6 8 7) ; 2 steps : une première solution espérée

Succes (1 3 2 4 3 5 7 3 8 10 3 11 13) ; 3 steps : une autre inattendue

solving (1 1 2 3 5 X X X) ; failure

solving (1 1 2 3 3 5 4 7 X X X X X X X X X)

Succes (1 1 2 3 3 5 4 7 5 9 6 11 7 13 8 15 9) ; 2 steps

Succes (1 1 2 3 3 5 4 7 5 9 6 11 7 13 8 15 9) ; 4 steps

solving (1 2 3 2 4 6 3 6 9 X X X X X X) ; a difficult one

Succes (1 2 3 2 4 6 3 6 9 4 8 12 5 10 15) ; 3 steps

solving (12 8 4 9 6 3 6 X X X X X X X X)

Succes (12 8 4 9 6 3 6 4 2 3 2 1 0 0 0) ; 3 steps

## 2) Construction of a small Lisp program

Justification:

- Show some of the reasons why program understanding is difficult

How:

- Begin with a simple concept, ex: «generate all permutations of a set»
- Translate it into a « conceptual » algorithm
- Translate this into a program
- See how constraints of the programming language change your initial « conceptual » algorithm
- Optimise in changing the data representation
- Optimise the algorithm
- Look at what you have obtained and how to reconstruct the «conceptual » algorithm

# Program 3: VCG

**VCG**: my adaptation of Daniel Goossens adaptation of David Luckams *Verification Condition Generator* (based on C.A.R. Hoare's axiomatic schemas)

*Approach:*

First:

- Introduction to Hoare's axiomatic programming logic
- Derivation of a subset of rewrite rules based on these axioms
- Application to some small Pascal-like programs

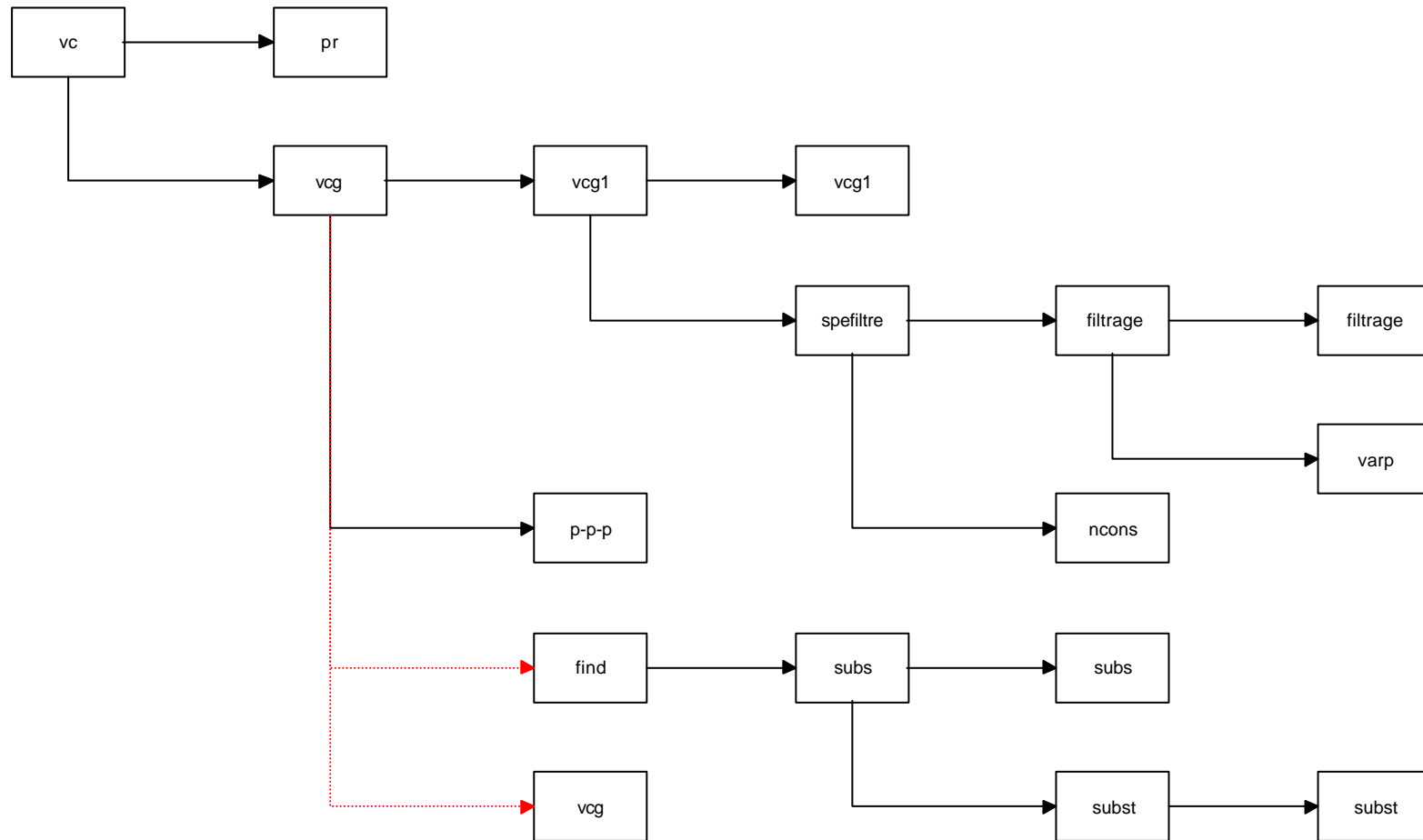
Second:

- Project: think about how to model this in a program

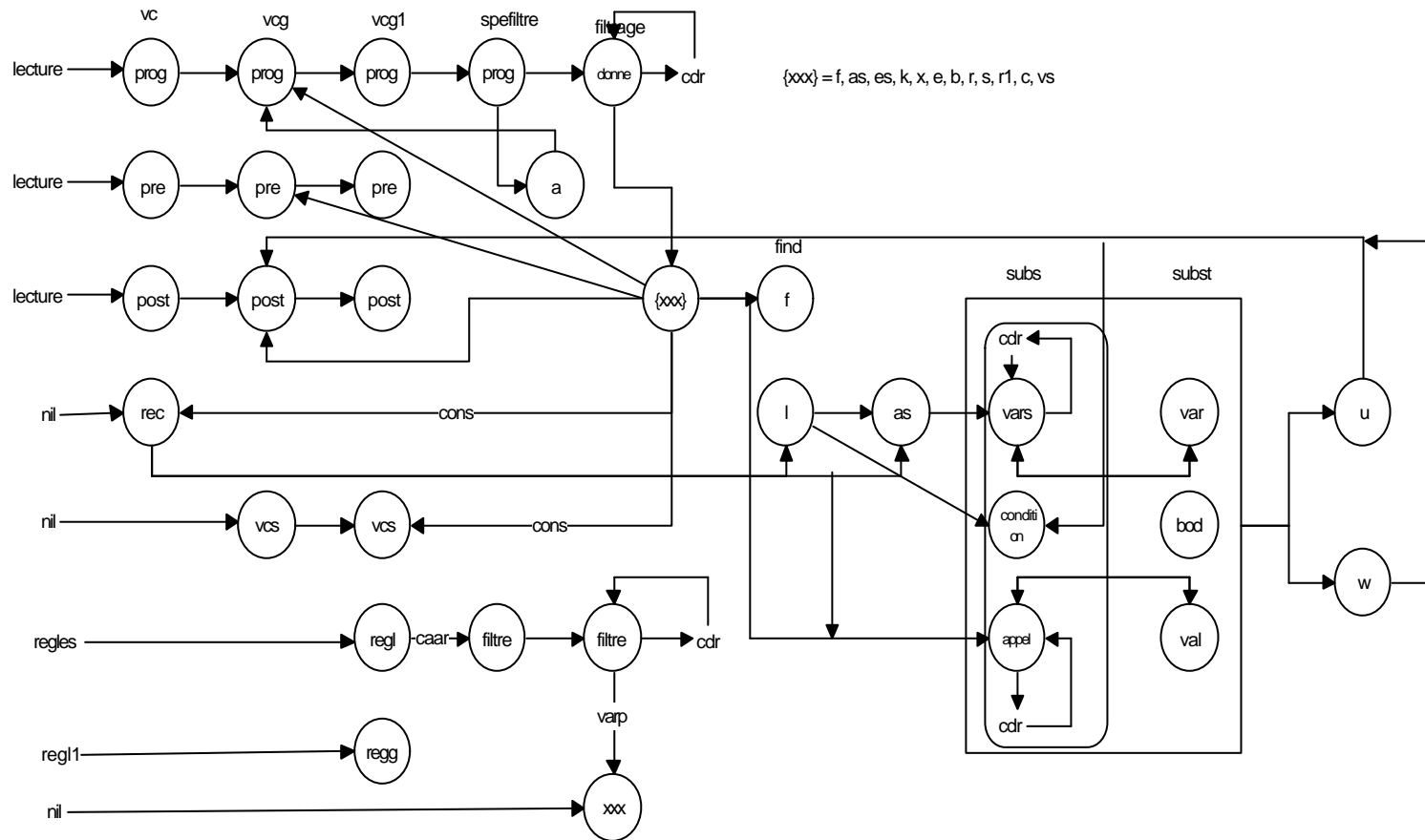
Third:

- Presentation of VCG as one possible implementation
- Collective analysis of it

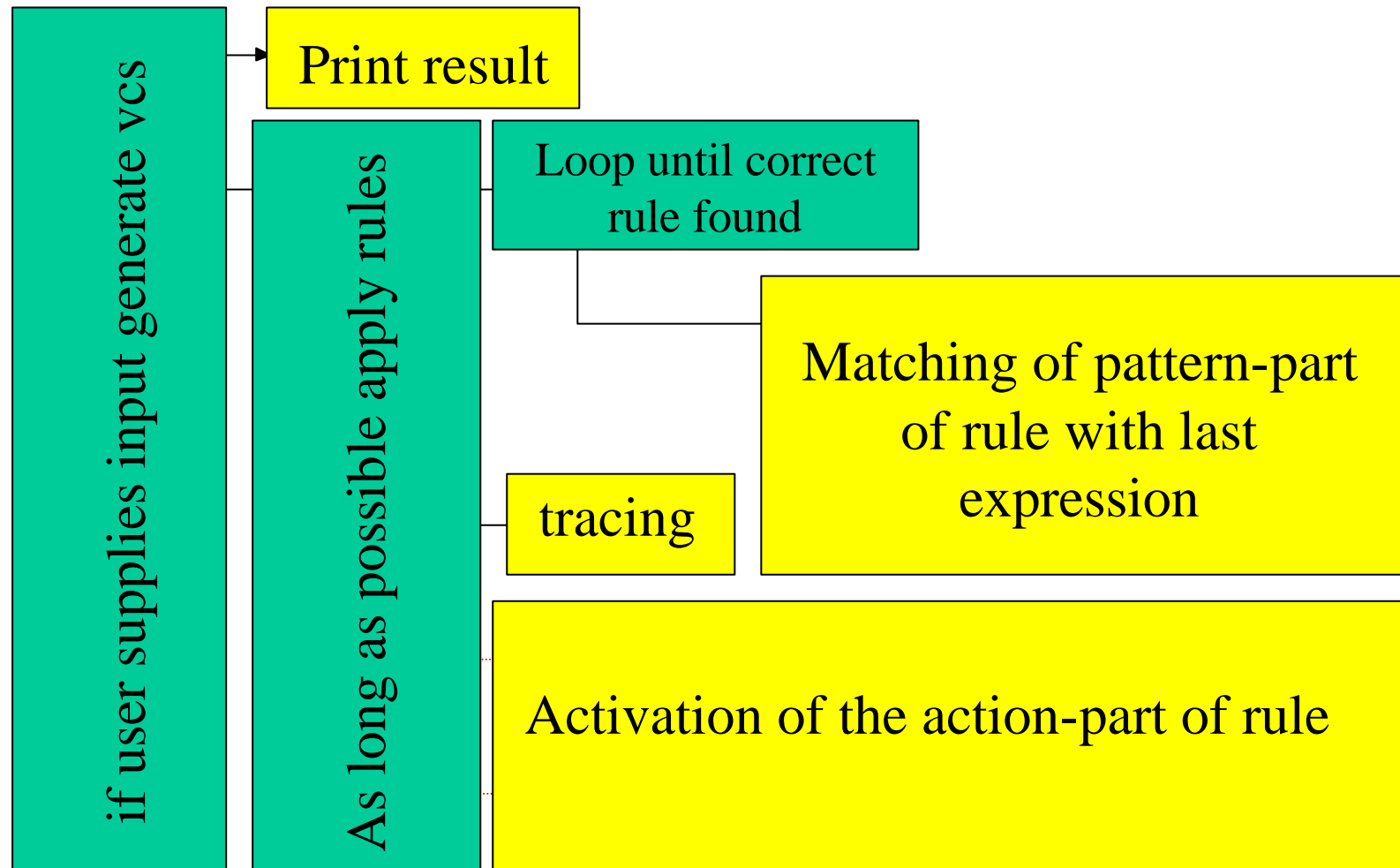
# View of controlflow of VCG



# View of dataflow of VCG



Both together help to get an understanding of VCG



# Problems encountered through VCG

- Rule based system
- Specialized pattern matcher changing values of global variables
- Extensive use of global variables to communicate between program and rewrite rules
- Pattern-directed function invocation
- Kind of « hidden » construction of a global database
- Problems to derive the structure of the input

## Program 4: MemoFib in Smalltalk

*What:* small Smalltalk program implementing and displaying a generator for Fibonacci numbers memorizing numbers already calculated in a doubly linked structure.

*Size:* 13 pages

*Structure:* 6 classes, 67 methods: 15+9+19+4+16+4 (only 3 classes are necessary for the required understanding task)

*Approach:* recall of oo-programming, recall of idea of memo-functions, handout of program listing, collective analysis

# Problems encountered through MemoFib

- Students seemed unfamiliar with concepts of object oriented programming
- Students seemed uneasy with private access-methods
- Algorithm to construct dynamically the memorization seemed too difficult (*it is difficult*)
- Q1: was the program too restricted? So problem space too restricted.
- Q2: is reading oo-programs fundamentally different?

# Back to program **foo**

A more detailed analysis of foo showing increasing abstractions: [scenario3.pdf](#)

# Lessons learned and open questions (1)

- Insufficient programming knowledge
  - ⇒ Teach less constructing new programs, teach more studying existing programs.
- Global variables are often *very* confusing
  - (Except if only one producer)
- Fuzziness of program understanding: different goals seem to need different understanding approaches
  - How to classify? specify?
- Very hard to go from behaviour to purpose
  - Following evolution of data often needs induction. Then, how to match obtained structure with meaning? How to generate meaning?

## Lessons learned and open questions (2)

- Fuzziness of representations or views
  - What are *good, readable, useful, etc* representations or views?
- Need for more (better?) tools to explore, analyze, transform programs
- Need for more goal oriented understanding: modification, extension, change of data-structure(s), etc
- Understanding depends of architecture *and* algorithms
- ✘ One *never* should give a class on Program Understanding without direct availability of computers and program observation/analysis/manipulation/etc tools.