Rule Based Language in Practice

Pierre-Etienne Moreau
Mines Nancy - Université de Lorraine
August, 25th 2014, Valparaíso

Term Rewriting is wonderful

- Expressive
- Formal
- Executable

You can

- study properties of rewriting
- use rewriting to describe algorithms, type systems, provers, compilers, etc.
- but also use rewriting to implement such systems

Can we use term rewriting for real programming ?

Many tools are based on rewriting

- CiME
- daTac
- Larch Prover
- Otter
- ReDuX
- Reve
- RRL
- Spike
- Caml
- Clean
- SML
- …

Some programming environments

main paradigm: term + rule + strategies

1975 Equational Interpreter (Chicago)

1977 OBJ (Menlo Park)

(Menlo Park) Maude

(Ishikawa) CafeOBJ

1985 ELAN (Nancy)

1980 ASF+SDF (Amsterdam)

1999 Stratego (Utrecht)

2001 Tom (Nancy)

ASF+SDF

Developed at CWI (Amsterdam)

Team headed by Paul Klint

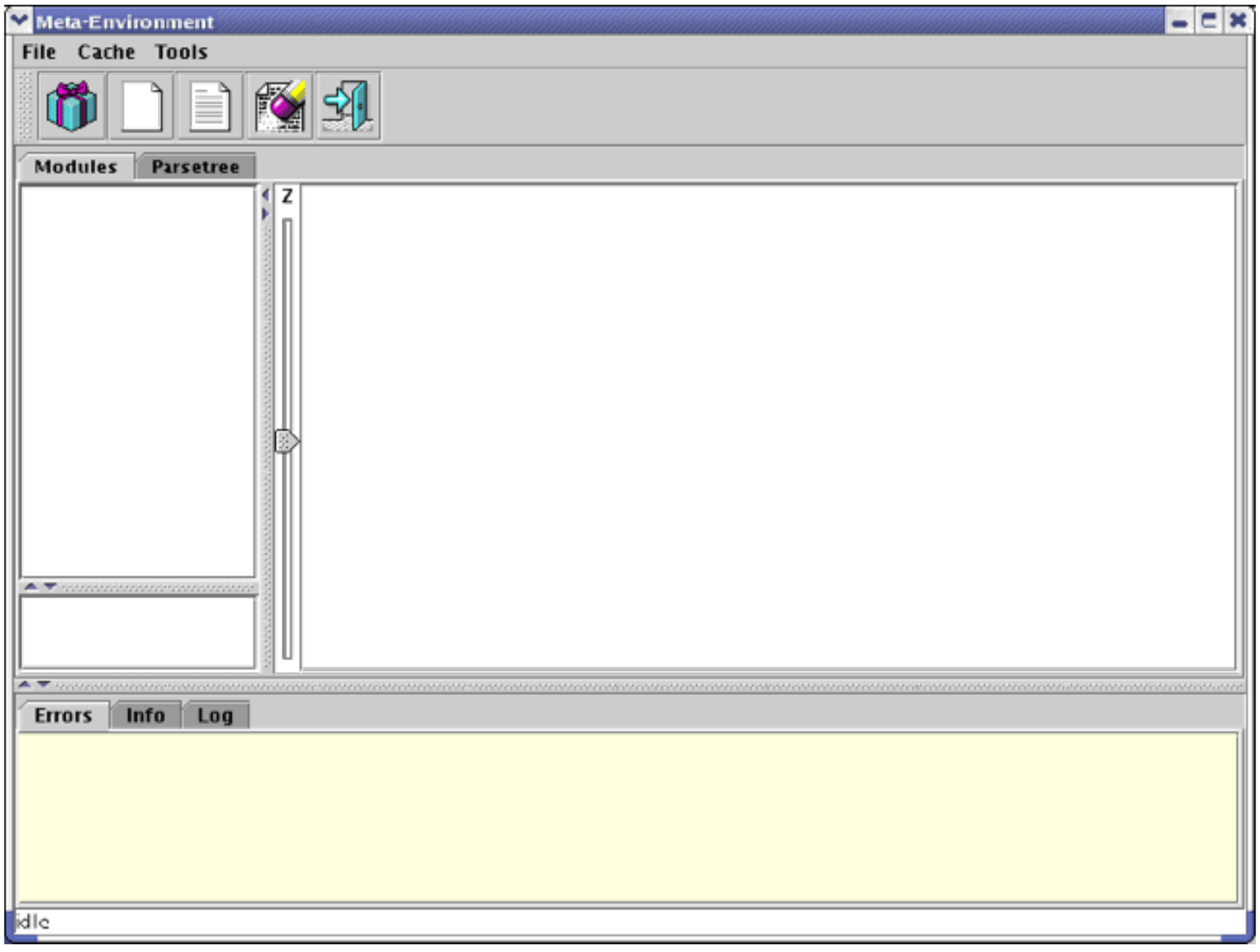
Goal:

- Develop Domain Specific Languages
- Analyse existing code (Cobol)
- Applications in banking domain

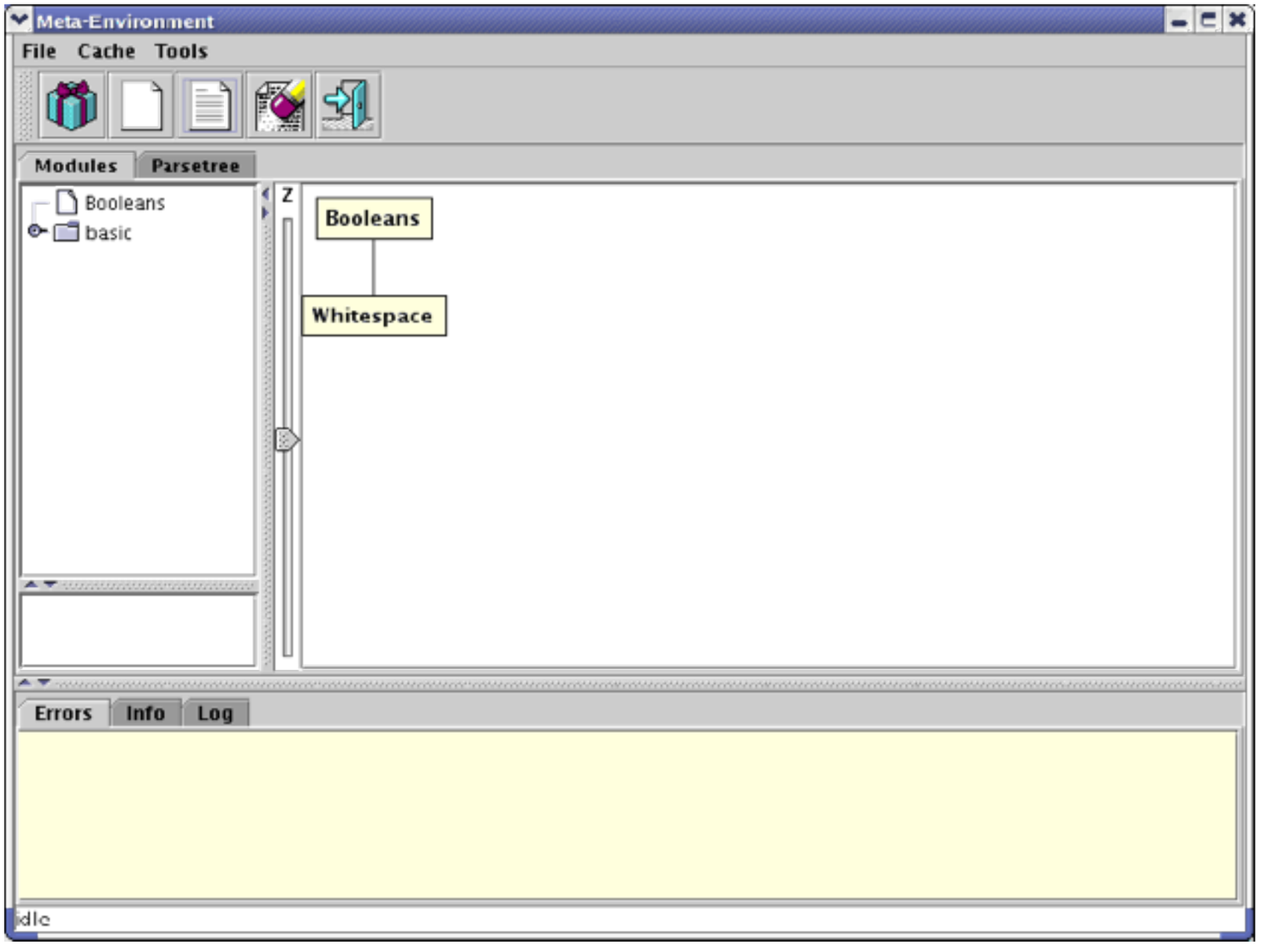
Consequences :

- graphical environment
- powerful syntax

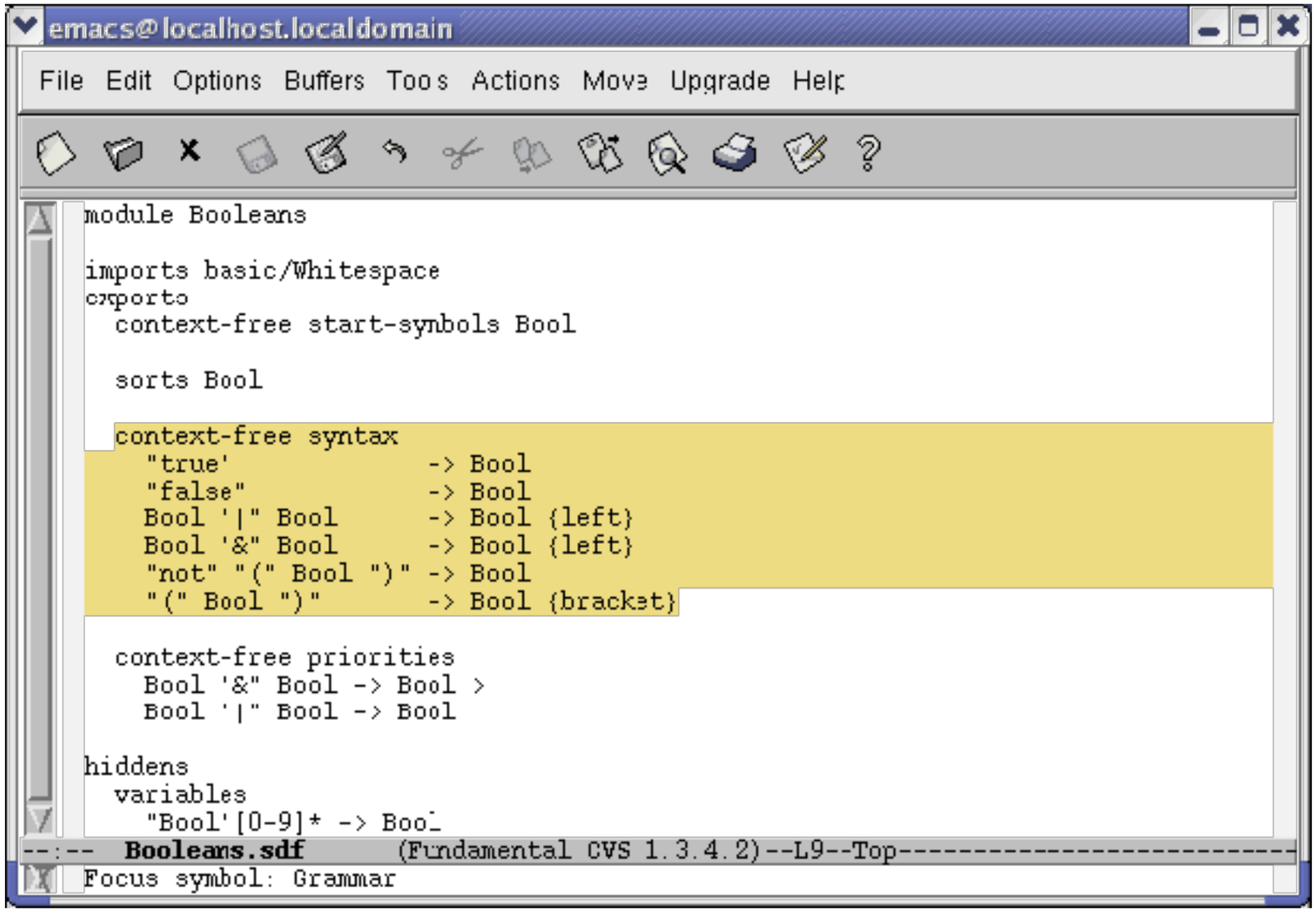
Running the environment



Defining a module



Editing the syntax



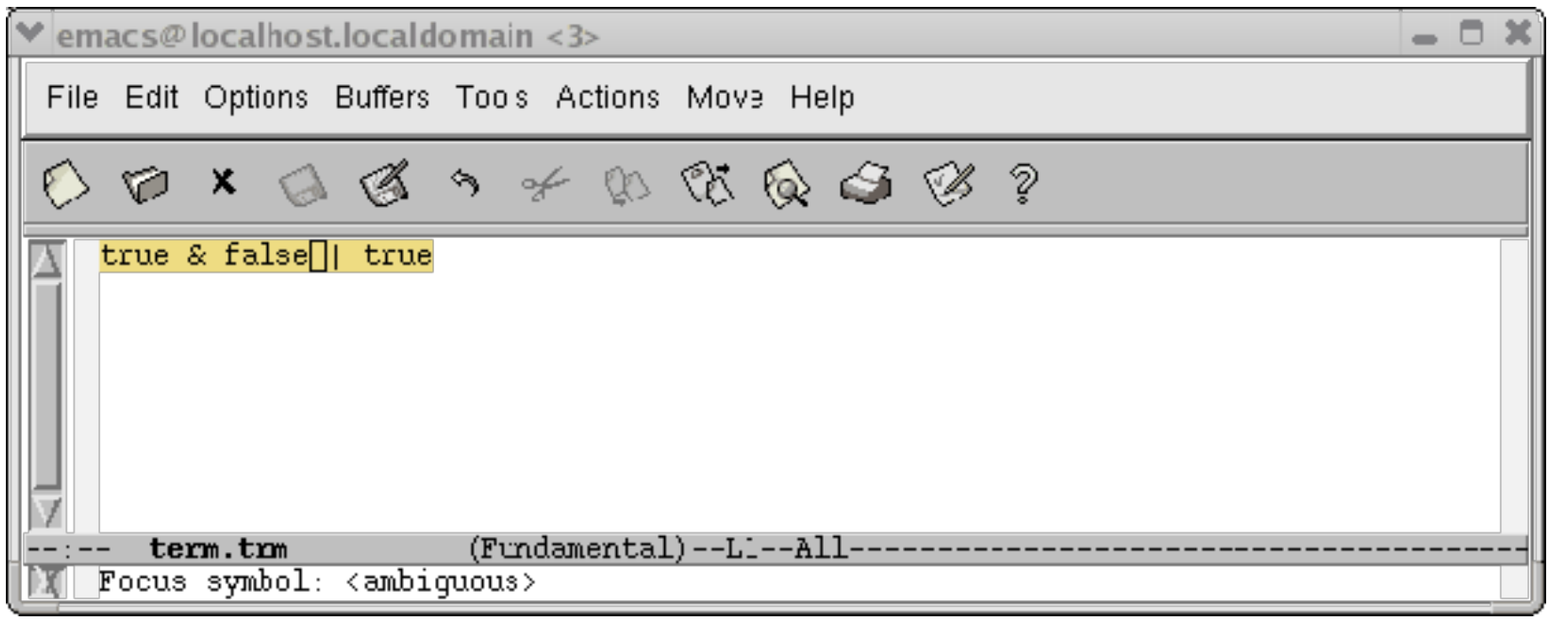
SDF:
Syntax Definition Formalism

```
module Booleans
imports basic/Whitespace
exports sorts Bool
context-free syntax
  "true"                -> Bool
  "false"               -> Bool
   Bool "|" Bool         -> Bool {left}
   Bool "&" Bool         -> Bool {left}
   "not" "(" Bool ")"    -> Bool
   "(" Bool ")"          -> Bool {bracket}
```

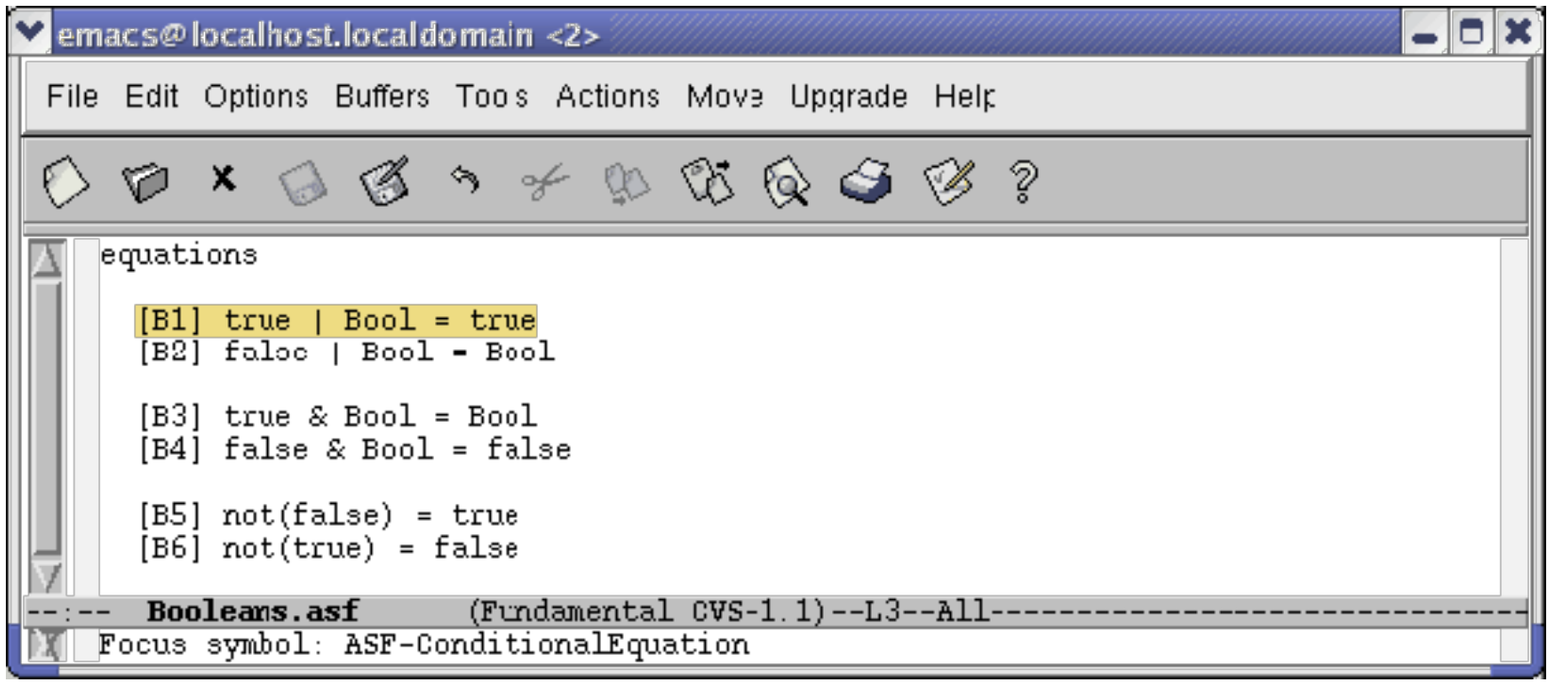
Solving ambiguities

```
context-free priorities

  Bool "&" Bool -> Bool >

  Bool "|" Bool -> Bool

Hiddens variables

  "Bool"[0-9]* -> Bool
```

Editing a term



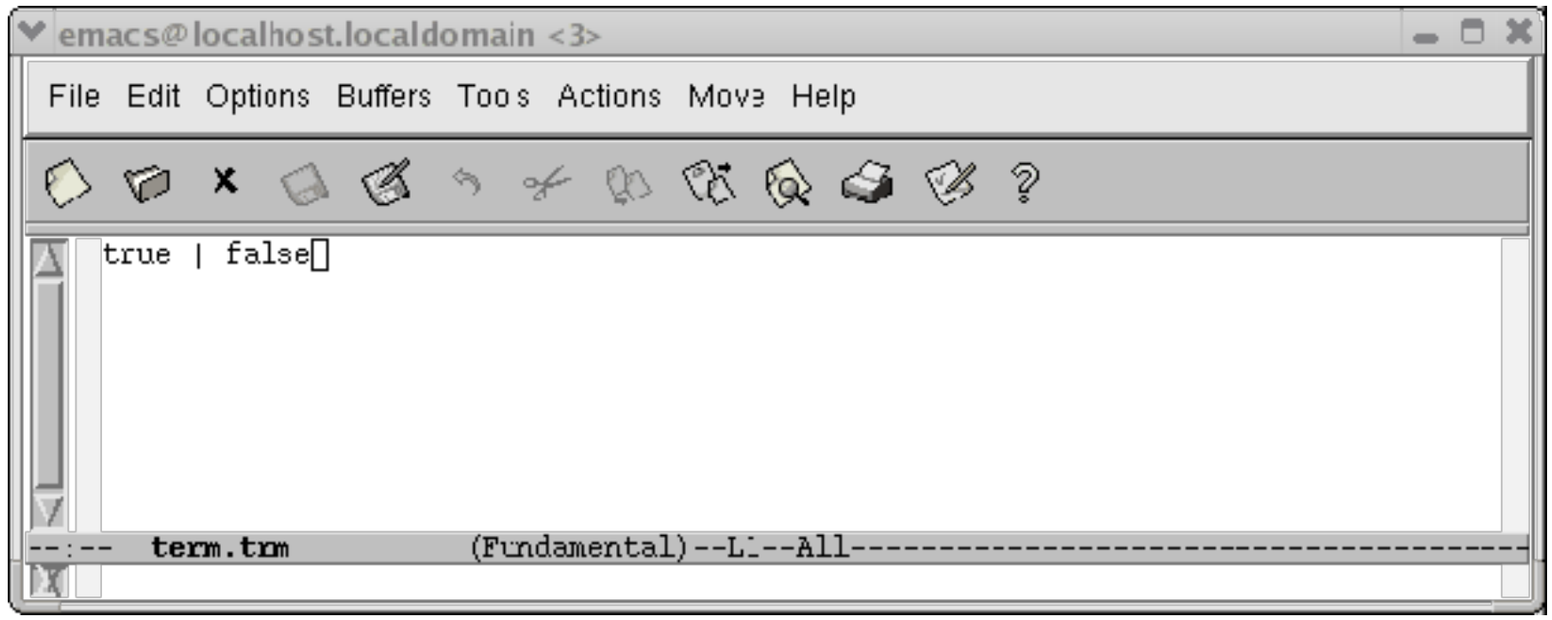
Editing rewriting rules



Equations

```
equations
[B1] true | Bool = true
[B2] false | Bool = Bool
[B3] true & Bool = Bool
[B4] false & Bool = false
[B5] not(false) = true
[B6] not(true) = false
```

Computing a normal form



Review

- Pro:
  - user friendly graphical environment
  - very powerful Syntax Definition Formalism
- Cons:
  - lack of powerful strategies
  - difficult to maintain stable

OBJ, CafeOBJ, ELAN, Maude

- Developed at **SRI, JAIST, Inria**
- Teams headed by J. Goguen, K. Futatsugi, C. Kirchner, J. Meseguer
    - make rewriting based languages usable in practice
    - to develop automatic provers
    - use the provers to prove properties about program written in the languages
- Consequences:
    - Equational theories (associativity, commutativity, …)
    - Non terminating rewrite systems
    - Strategies for controlling/exploring

Maude signature

```
fmod PEANO-NAT-EXTRA is
    sort Nat .

    op 0 :                  -> Nat [ctor] .
    op s :      Nat         -> Nat [ctor iter] .
    op _+_ :    Nat Nat     -> Nat .

    vars M N : Nat .

    eq 0 + N = N .
    eq s(M) + N = s(M + N) .
endfm
```

Environment

```
Maude> reduce s(0) + s(s(0))

result Nat: s(s(s(0)))
```

Associative theory

```
fmod LIST is
  sorts Elt List .
  subsort Elt < List .
  op nil : -> List [ctor] .
  op _ _ : List List -> List [ctor assoc id: nil] .

endfm
```

Meta-level

```
op op_:_->_[_] :  Qid QidList Qid AttrSet -> OpDecl [ctor] .
```

meta-meta-encoding: of “op t : -> Truth”

```
'__['op_:_->_`[_`].[ {''t}'Qid, {'nil}'QidList, {''Truth}'Qid, {'none}'AttrSet]

op fmod_is_sorts_.____endfm :
                Qid ImportList SortSet SubsortDeclSet OpDeclSet
                MembAxSet EquationSet -> FModule [ctor] .
```

Programming the meta-level

```
fmod 'PEANO-NAT-EXTRA is
    nil sorts 'Nat . none
    op '0 : nil -> 'Nat [ctor] .
    op 's : nil -> 'Nat [ctor] .
    op '_+_ : 'Nat 'Nat -> 'Nat [none] .
    None
    eq '_+_['N:Nat, '0.Nat] =    'N:Nat [none] .
    eq '_+_['s['M:Nat], 'N:Nat] =  's['_+_['M:Nat, 'N:Nat]]    [none] .
endfm
```

```
Maude> red metaReduce(
    ['PEANO-NAT-EXTRA],  '_+_['s['s['0.Nat]], '0.Nat]
) .
result ResultPair: {'s['s['0.Nat]], 'Nat}
```

Review

- Pro:
  - very powerful equational matching
  - very powerful meta-level
- Cons:
  - meta-level difficult to use
  - no compiler (interpreter only)

ELAN signature

```
module fib

    import global int;

end

operators global

    fib(@) : (int) int;

end
```

ELAN rules

```
rules for int

    n : int ;

global

    [] fib(0) => 1 end

    [] fib(1) => 1 end

    [] fib(n) => fib(n - 1) + fib(n - 2) if n > 1 end

end
```

Two problems

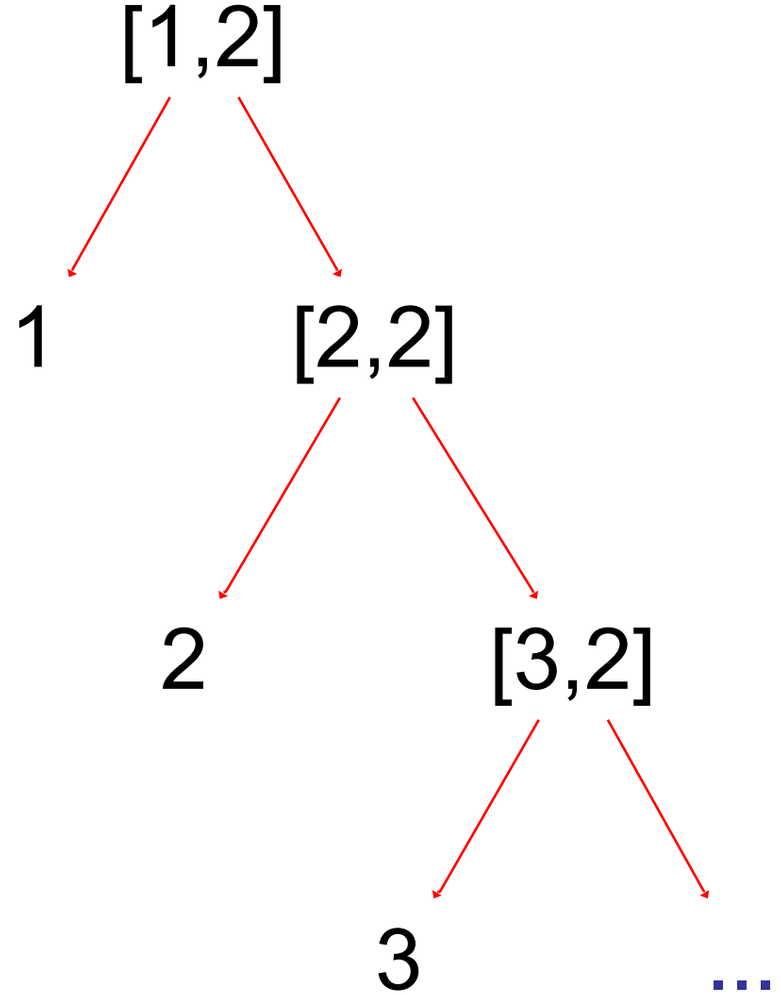
2 rules

[] [x,y] → x

[] [x,y] → [x+1,y]

Non-confluent

Non-terminating



Controlling a non confluent system

Named rules:

[R1] [x,y] → x

[R2] [x,y] → [x+1,y]

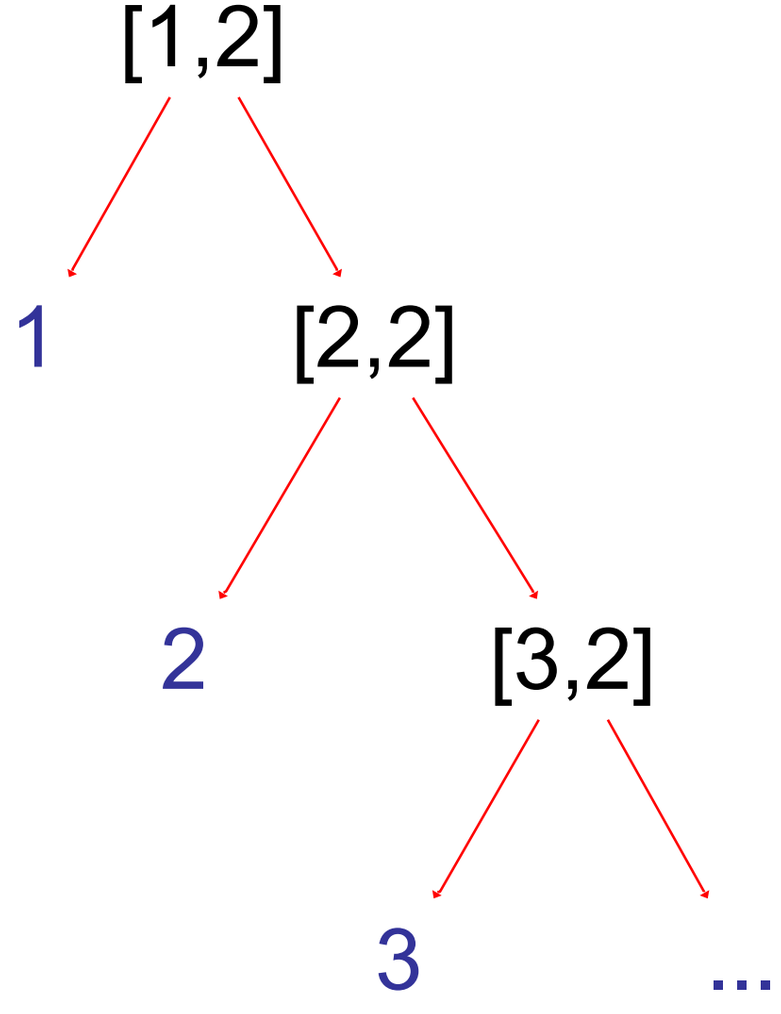
Strategies:

repeat(first one(R1,R2))

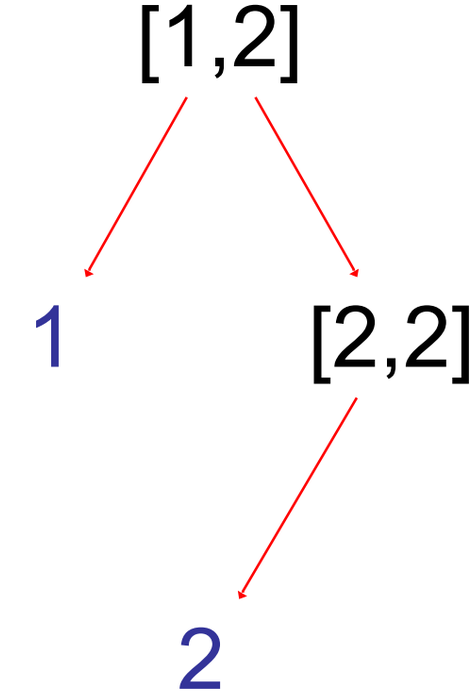
1

repeat(dk(R1,R2))

{ 1, 2, 3, … }



Controlling a non terminating system



Adding a condition :

[R1] [x,y] → x

[R2] [x,y] → [x+1,y] if x < y

Adding a rule :

[R1] [x,y] → x

[R2] [x,y] → [x+1,y]

[check] [x,y] → [x,y] if x < y

Strategy

repeat(dk(R1, check ; R2))

{ 1, 2 }

Review

- Pro:
  - powerful equational matching
  - powerful strategy language
  - non-deterministic search
- Cons:
  - I/O can be improved
  - difficult to integrate in existing environments

To make a rule based language usable in practice

- expressive (equational matching and strategies)
- efficient
- good debugging and I/O facilities
- run in any environment (Windows, Unix)
- simple and “removable”

Tom

- Terms
- Matching
- Strategies

in general purpose languages

Used in academic and industrial context

- express transformations
- research and prototyping
- compilers
- automatic provers
- query transformation

tom.loria.fr

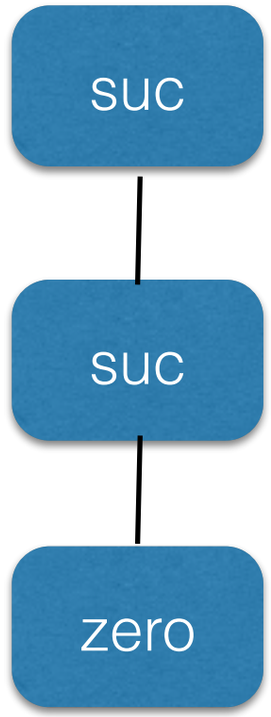
Main constructs

- Signature

```
%gom {
  module Peano
  abstract syntax
  Nat     = zero()
          | suc(n:Nat)
}
```

Basic mechanism: the ` operator

- used to build terms
- Example: `suc(suc(zero()))



The term should be well-formed and well-typed

Simple TOM program

```
public class Peano{
 %gom{
      module peano
      abstract syntax
      Nat =   zero()
          |   suc(n:Nat)
 }
 public static void main(String[] args){
    System.out.println("One="+`suc(zero()));
 }
}
```

The program prints:

suc(zero())

Don’t forget the `

```
System.out.println("One="+ suc(zero()));
```

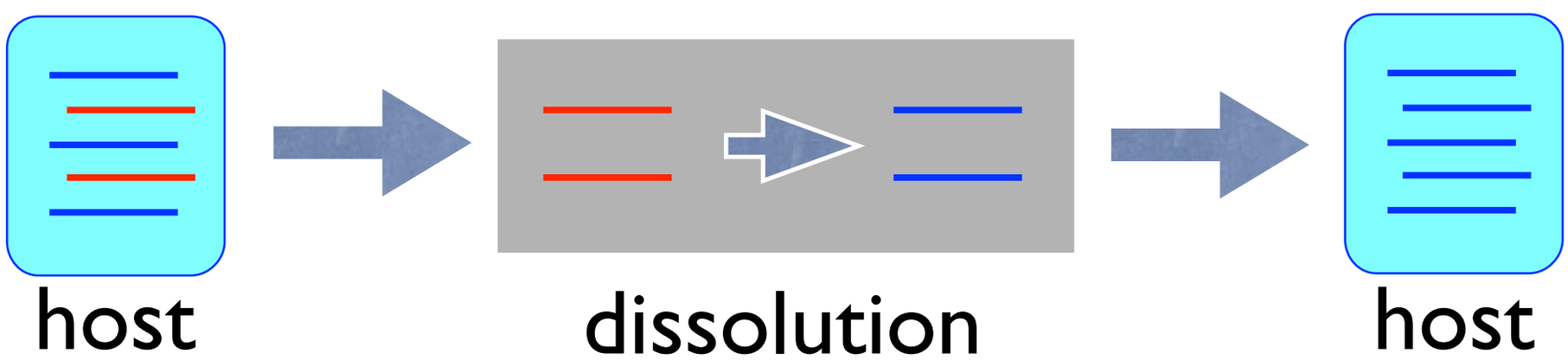
generates an error since suc is not a Java method

Implement addition with rewrite rules

```
module Peano:rules() {
    one() -> suc(zero())
    two() -> suc(one())
    plus(x,zero()) -> x
    plus(x,suc(y)) -> suc(plus(x,y))
}
```

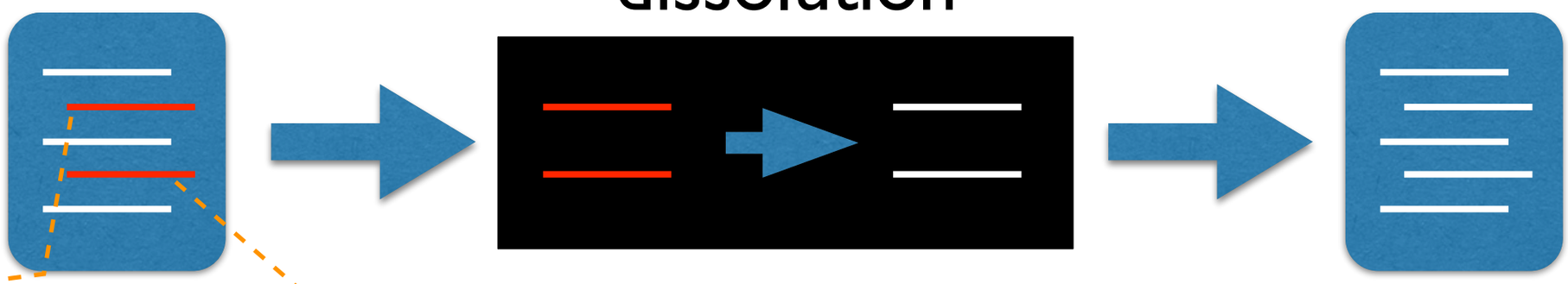
- System.out.println(`plus(one(),two()))
- Note that variable types are inferred

A piggyback ride



TOM compilation

dissolution



- smooth integration of rewrite rules into Java, C, etc.

```
Nat  = zero()
     | suc(Nat)
     | plus(Nat,Nat)
```

ex

```
plus(x,zero()) -> x
plus(x,suc(y)) -> suc(plus(x,y))
```

Demo

Sumup

- %gom can be used to define signatures
- rules() can be used to define a rewrite system containing:
  - constructors and variables below the top symbol of a lhs
  - constructors, variables and defined symbols in the rhs
- `` ` `` is used to build terms out of constructors, defined symbols and Java variables and functions

Exercise (homework)

- define a signature to handle sets
- define the operations
  - occurs : E x S -> Bool
  - union : S x S -> S
  - subset, eq : S x S -> Bool
  - diff : S x S -> S

Exercise

- Define a function

  plusInt(Nat, Nat) -> int

  which returns the addition of two Nat terms as an integer

- The instruction

  System.out.println(« res = » + `plusInt(one,two));

  should print

  res = 3

Solution
(which doesn’t work)

```
%gom {

    …

    imports int

    …

    int = plusInt(Nat, Nat)

    …

}

module ...:rules() { ... }
```

Another solution: combine matching and Java

```
public Nat plus(Nat n1, Nat n2) {

  %match(Nat n1, Nat n2) {

      x, zero() -> { return `x; }

      x, suc(y) -> { return `suc(plus(x,y)); }

  }

}
```

- plus is no longer a defined symbol but a Java function
- x et y are variables instantiated by the matching
- they become local variables for each of the rules

Tom, the following…

Exercise

- Write the program that prints the result as a builtin integer

- System.out.println(« res = » + `integer(suc(plus(one,two))));

  > res = 4

Solution

```
public int integer(Nat n) {

    %match(n) {

        zero()    -> { return 0; }

        suc(x)    -> { return 1 + integer(`x); }

    }

}
```

Right-hand side can mix Tom and Java

Semantics of %match

- Select the first pattern that matches
- Evaluate the associated action
- Execution continues
  - after the action (i.e. with the next matchable pattern) if there is no break or return
  - according to the control flow

Example

```
public int toInteger(Nat n) {

        %match(n) {

                zero() -> { return 0; }

                suc(zero()) -> { System.out.println("A zero!"); }

                suc(suc(zero())) -> { System.out.println("A one!"); }

                suc(y) -> { return toInteger(`y) + 1; }

        }

}
```

Right-hand side can mix Tom and Java

Questions

- Can we encode a conditional rewrite system with %match?

- Can we encode non-linear patterns?

Conditional rules

- Yes, the corresponding condition should be put in the action-part of the rule

```
public int toInteger(Nat n) {
    %match(n) {
        zero() -> { return 0; }
        suc(x) -> {
            if(`x != `zero()) { return 1 + `toInteger(x); }
        }
        suc(zero())     -> { return 1; }
    }
}
```

Part 2

List handling

List representation

Usually we consider an empty list nil and a concatenation operator cons

$$nil : \rightarrow L$$

$$cons : E \times L \rightarrow L$$

The list [a, b, c] is represented by

cons(a,cons(b,cons(c,nil)))

Question

- How to retrieve a given element?

in(cons(a,cons(b,cons(c,nil))),a) → true

in(cons(a,cons(b,cons(c,nil))),b) → true

in(cons(a,cons(b,cons(c,nil))),d) → false

Possible solution

in(cons(x,l),x) → true

in(cons(y,l),x) → in(l,x)

in(nil,x) → false

Associative symbols

- a symbol f is associative if

  f(x,f(y,z)) = f(f(x,y),z)

- which one of these statements is true?
  - f(a,f(b,f(c,d))) = f(f(a,b),f(c,d)) ?
  - σ such that σf(x,f(c,y)) = f(f(a,b),f(c,d)) ?
  - f(x,f(c,y)) << f(f(a,b),f(c,d)) ?
  - f(x,f(c,y)) << f(a,f(b,f(c,d))) ?
  - f(x,f(d,y)) << f(a,f(b,f(c,d))) ?

Associativity with neutral element

- a symbol f is associative with neutral element e if

$$f(x,f(y,z)) = f(f(x,y),z)$$

$$f(x,e) = f(e,x) = x$$

- In this case do we have
  - f(x,f(c,y)) << f(a,f(b,f(c,d))) ?
  - f(x,f(d,y)) << f(a,f(b,f(c,d))) ?

Associative symbols in Tom

- A variadic symbol f with profile:

  f : E x … x E -> L

- is denoted

  f(E*) -> L

- and we can write

  f(a(), b(), c(), d())

  f(x, a(), y, b(), z)

Associative matching

- We consider the flat forms
  - f(a,b,c,d)
  - f(x,c,y)
- and we can write
  - f(x,c,y) << f(a,b,c,d)
  - f(x,d,y) << f(a,b,c,d)

Associative symbols in Tom

```
%gom {
      module ListSet
      imports int
      abstract syntax
       Set = conc(int*)
}

public static void main(String[] args){
     Set s = `conc(1,2,3,4);
     System.out.println("s = "+s);
}
```

Question

Check if the second element of a list of four elements is a “2”

```
%match(s) {
   conc(x,2,y,z) -> { System.out.println("2 found"); }
}
```

Questions

- What are the types of x, y and z in conc(x,2,y,z)?
  - answer : int
- Fill in the "..." in conc(...,2,...) to obtain a pattern that matches against conc(1,2,3,4) and conc(1,2,3)?
  - we need variables of type Set
  - which are denoted with a * like in conc(x*,2,y*)

Question

- Check if a list contains a “2”

```
%match(s) {
   conc(X*,2,Y*) -> { System.out.println("2 found"); }
}
```

- Check if the second element of a list is a “2”

```
%match(s) {
   conc(x,2,Y*) -> { System.out.println("2 found"); }
}
```

Multisets to Sets

- How to get rid of doubles?
  - conc(X*,y,y,Z*) -> conc(X*,y,Z*)
    - eliminates consecutive doubles
- How to eliminate distant doubles?
  - conc(X*,y,T*,y,Z*) -> conc(X*,y,T*,Z*)

Demo

Pattern matching
(see CK's slide 71)

Decompose: $(f\ P1 \ldots Pn) << (f\ A1 \ldots An) \rightarrow \bigwedge_{i=1\ldots n} Pi << Ai$

SymbolClash: $(f\ P1 \ldots Pn) << (g\ A1 \ldots An) \rightarrow$ False

Delete: $P << P \rightarrow$ True

PropagageClash: $S \wedge$ False $\rightarrow$ False

PropagateSuccess: $S \wedge$ True $\rightarrow S$

Exercise: implement and run these examples

f(x,g(y)) << f(a,g(b))

→ x << a ∧ g(y) << g(b)

→ x << a ∧ y << b

f(b,g(y)) << f(a,g(b))

→ b << a ∧ g(y) << g(b)

→ False ∧ y << b

→ False

Exercise

- Implement an addressbook with the names and birthdates of a list of persons
- give the pattern that looks for the persons with the same name
- give the pattern that finds the persons with a given birthday

Persons

```
%gom{
    module Persons
    imports int String
    abstract syntax

    Date = date(year:int, month:int, day:int)
    Person = person(fn:String,ln:String,bdate:Date)

    PersonList = concPerson( Person* )
}
```

Happy birthday

```
void happyBirthday(PersonList book, Date date) {
  %match(book) {
    concPerson(_*, person(fn, _, date), _*) -> {
      System.out.println("Happy birthday " + `fn);
    }
  }
}
```

Same name

```
%match(book) {
   concPerson(_*, p1@person(fn1,name,_), _*,
                  p2@person(fn2,name,_), _*) -> {
      System.out.println(`p1+" and " +`p2 +
                  "have the same family name");
   }
}
```

Exercise

- Given an order on the elements of the list
- Define a sorting algorithm for the list
- Implement it in Tom

```
public static PersonList sort(PersonList book) {
%match(book) {
  concPerson(X*, p1@person(prenom,name1,_), Y*,
                 p2@person(firstname,name2,_), Z*)
   -> {
    if(`name1.compareTo(`name2) > 0) {
        return sort(`concPerson(X*, p2, Y*, p1, Z*));
    }
  }
}
return book;
}
```

Exercise

- Find the members of the same family that have a different firstname

Different firstname

```
%match(book) {
concPerson(_*, p1@person(fn1,name,_), _*,
                  p2@person(fn2,name,_), _*)  -> {
    if(`fn1 != `fn2)
      System.out.println(`p1+" and " +`p2 +
          "have different firstnames");
}
}
```

Different firstname

```
%match(book) {
concPerson(_*, p1@person(fn,name,_), _*,
                    p2@person(!fn,name,_), _*)  -> {
        System.out.println(`p1+" and " +`p2 +
            "have different firstnames");
}
}
```

Anti-patterns

- complements used in patterns
- specify what you don't want to match
- use the '!' symbol
- !a => everything that is not a
  - g(!a) << g(b)
  - f(x,!x) << f(a,b)

Tomorrow

- How to use strategies
- How to implement term rewriting
  - pattern matching
  - replacement
  - leftmost-innermost normalization
- How to implement user defined strategies