Functional-Logic Programming

- Lecture Notes -

Harold Boley
NRC-IIT Fredericton
University of New Brunswick

CS 6715 FLP

11 April 2010

Principles of Functional and Logic Programming

Principles

Principles

About Knowledge Representation (KR), Software Specification, and Programming

$$\mathbf{KR}_{\mathbf{AI}} \quad \approx \quad \mathbf{Specification}_{\mathbf{Software}}$$

When

KRs / Specifications

are executable,

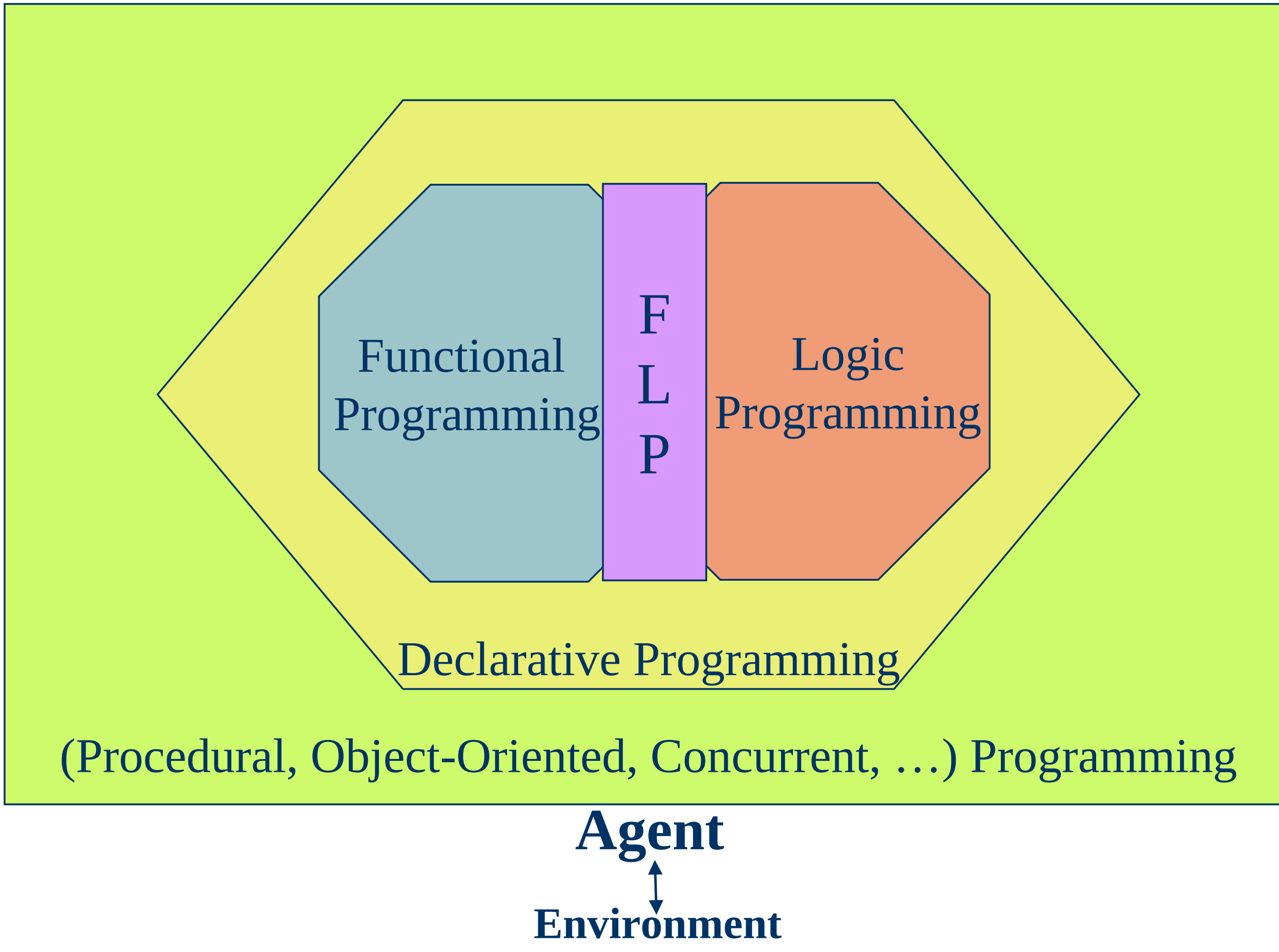
such as those studied here,

they can be considered as

(Declarative) Programs,

and their creation as Programming

Programming: Functional (FP), Logic (LP), and Functional-Logic (FLP) for Agent Core



Top-Level Terminology for Functions (FP), Relations (LP), and Their Combinations (FLP)

- FP: Function
- LP: Relation (or Predicate)
- FLP

Operation

Preview of Foundations of Functional-Logic Programming (FLP)

FLP is founded on Horn logic with **oriented** equations in rule conclusions, defining functions (applied to arguments), thus specializing, e.g., W3C's recent RIF-BLD, founded on Horn logic with **symmetric** equations

$$\mathit{head}\ \ {:}\text{-}\ \ \mathit{body}\ \ \&\ \ \mathit{foot}.$$

is a **specialization** and Prolog-**extending** syntax of

$$\mathit{head} = \mathit{foot} \Leftarrow \mathit{body}$$

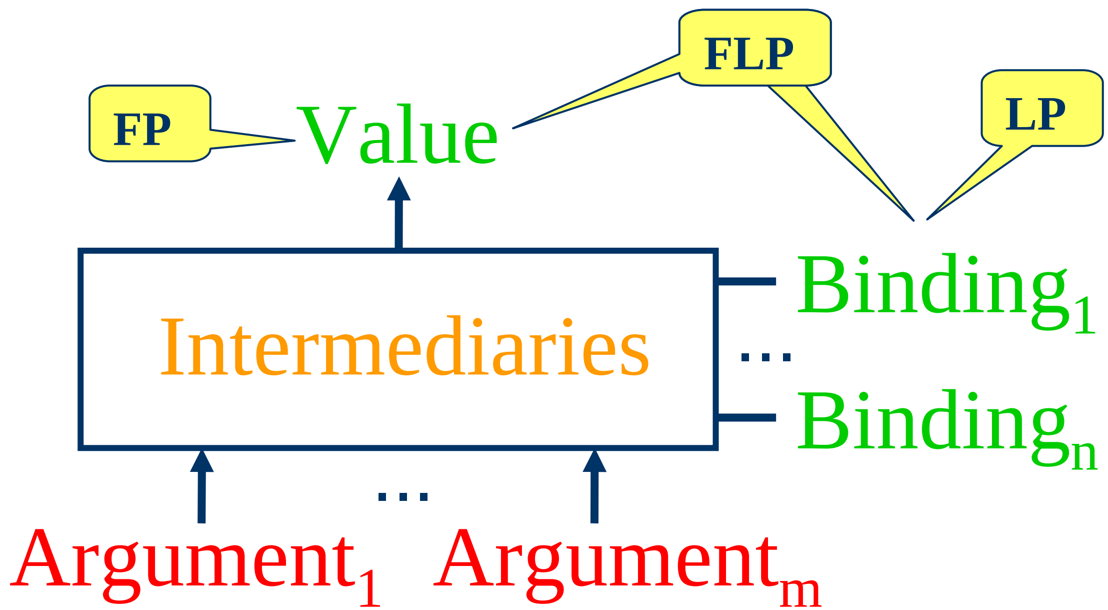
Declarative Programs: Joint Treatment of Functional and Logic Programming

- Declarative programs as executable specifications:
  - Founded on mathematical-logical formalisms
  - Easier analysis, verification, transformation, maintenance
  - Efficiency through compilation and parallel execution
  - Extensible to state-change/systems-level programming
- Reasons for a joint functional and logic treatment:
  - Overlap of / commonality between many FP and LP notions
  - Added value through combined functional-logic programs
  - Shared interfaces to / combination with other (procedural, object-oriented, concurrent, …) programming paradigms
  - Economy in learning/teaching declarative programming: Will be practiced in the following, as implemented in Relfun
- FP+LP ideas in other paradigms such as OOP and Relational DBs (e.g., FP: Generic Java, LP: SQL-99)

Basic Color-Coded Visualization of Operations

Red: Input Arguments
Orange: Thruput Intermediaries
Green: Output (Returned) Value
and (Result) Bindings

Operation

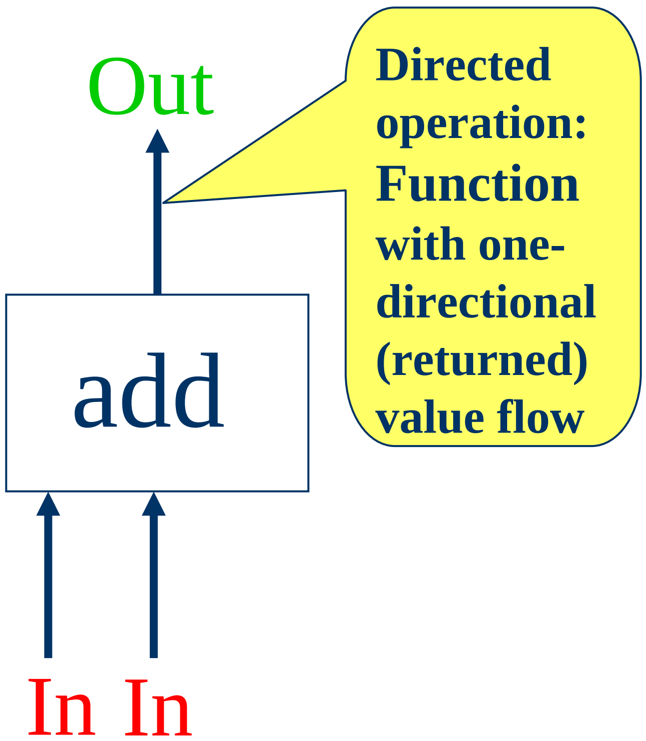


(Multi-)Directionality Principle

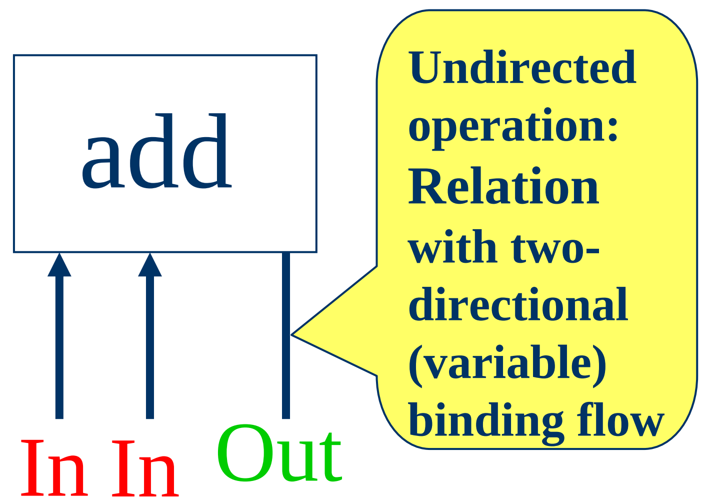
- Pure Functional Programming: **Functions** are operations with **one direction** of computation from ‘input’ arguments to ‘output’ values (definable with oriented equations)
- Pure Logic Programming: **Relations** are operations with **multiple directions** of computation between ‘input’/‘output’ arguments (definable via unification)

Declarative Programs as Data Flow Diagrams: Example – "Addition Agent" (I-O Modes)

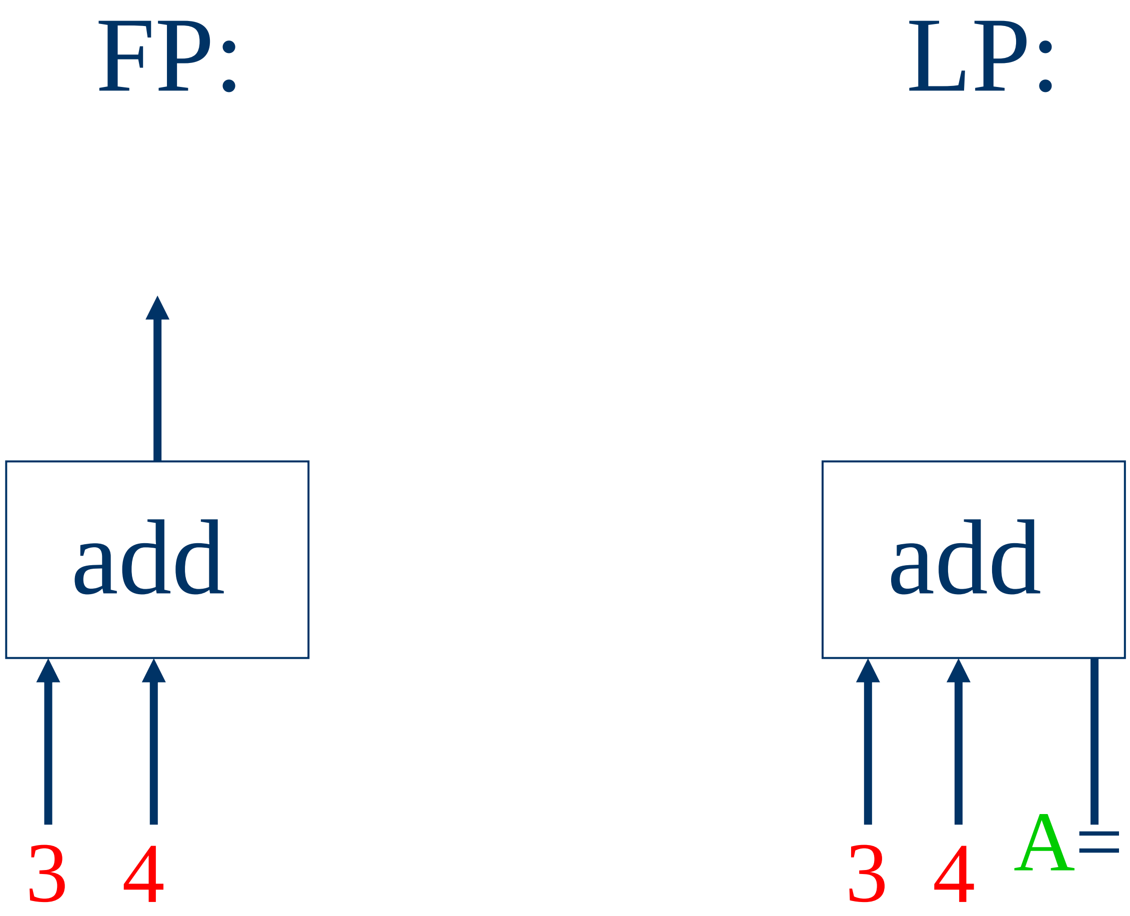
FP:



LP:

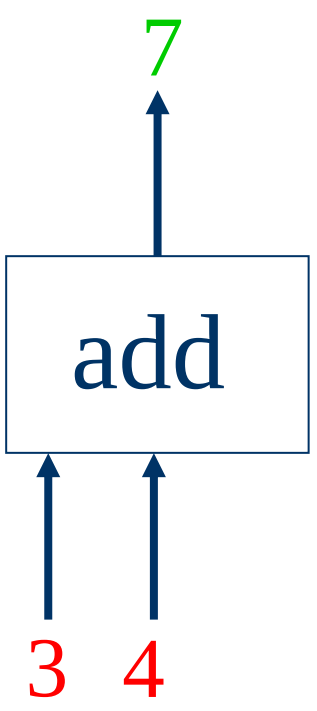


Declarative Programs as Data Flow Diagrams: Example – "Addition Agent" (Input)

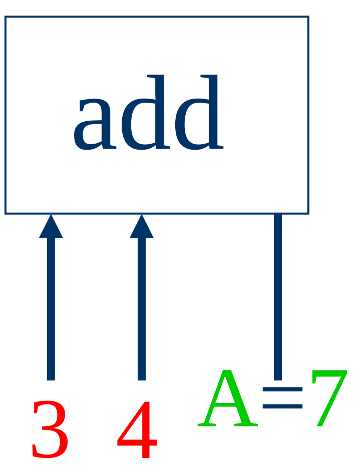


Declarative Programs as Data Flow Diagrams: Example – “Addition Agent” (Output)

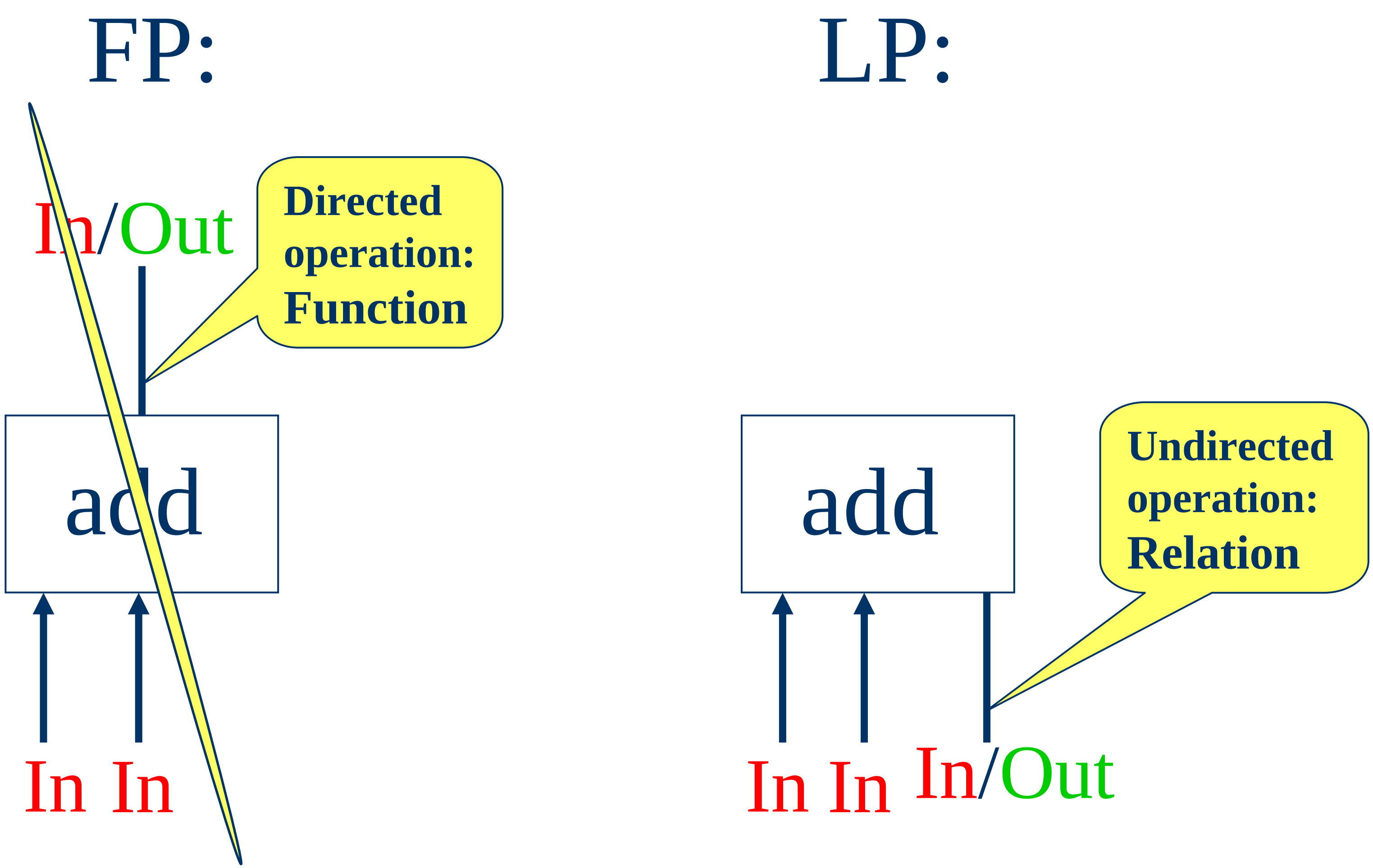
FP:



LP:

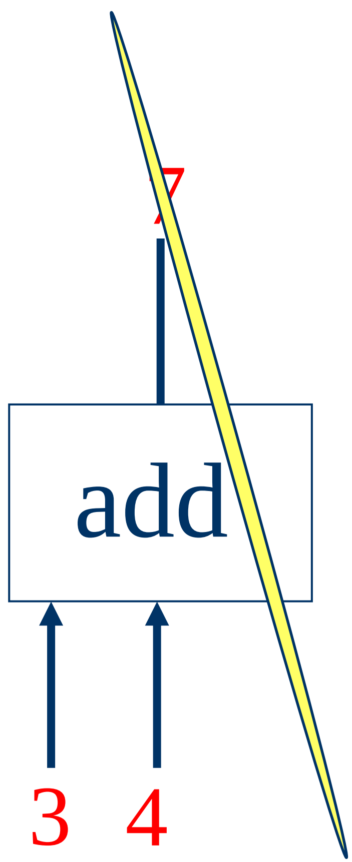


Declarative Programs as Data Flow Diagrams: Example – "Addition Agent" (I-O Modes)

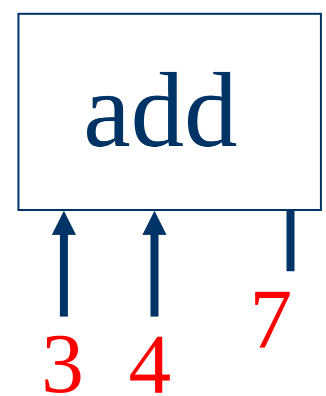


Declarative Programs Used for Testing: Example – “Addition Agent” (Input)

FP:

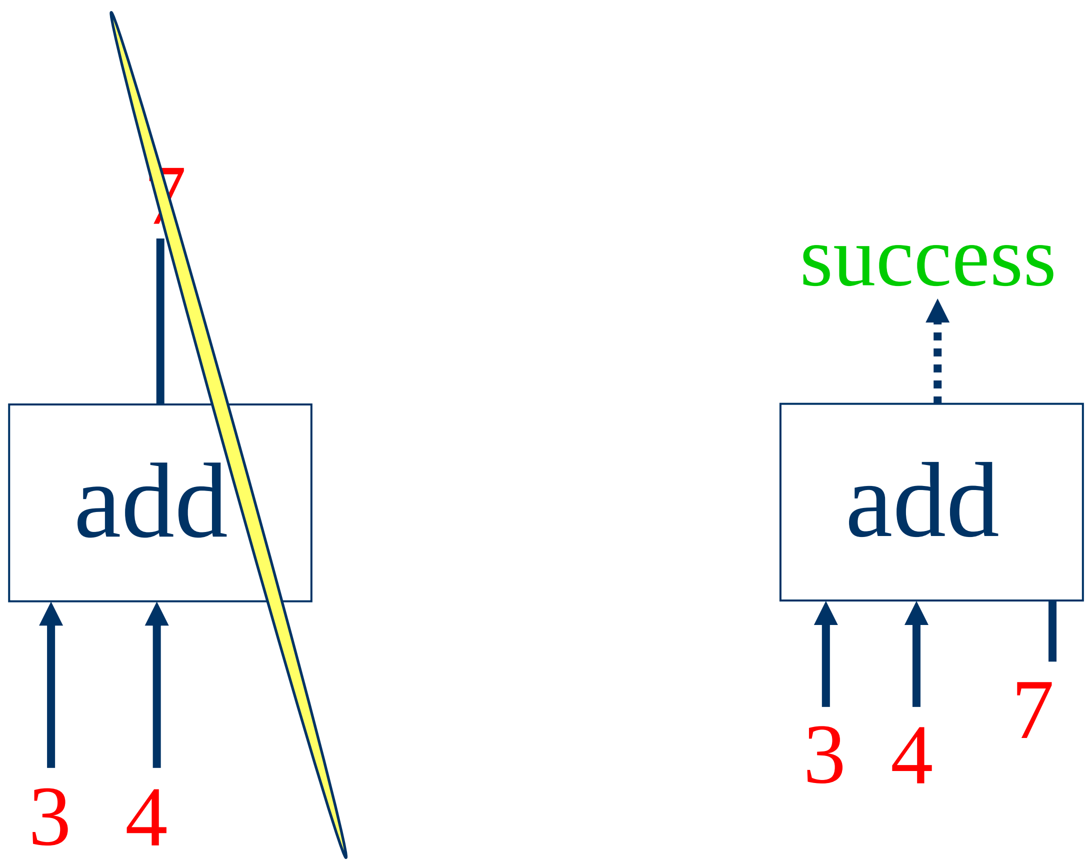


LP:



Declarative Programs Used for Testing: Example – "Addition Agent" (Output)

FP: LP:



Declarative Programs in Symbolic Notation: Example – "Addition Agent"

FP:

I-O Mode:

add: In × In → Out

Input-Output Trace:

add(3, 4)
7

LP:

I-O Modes:

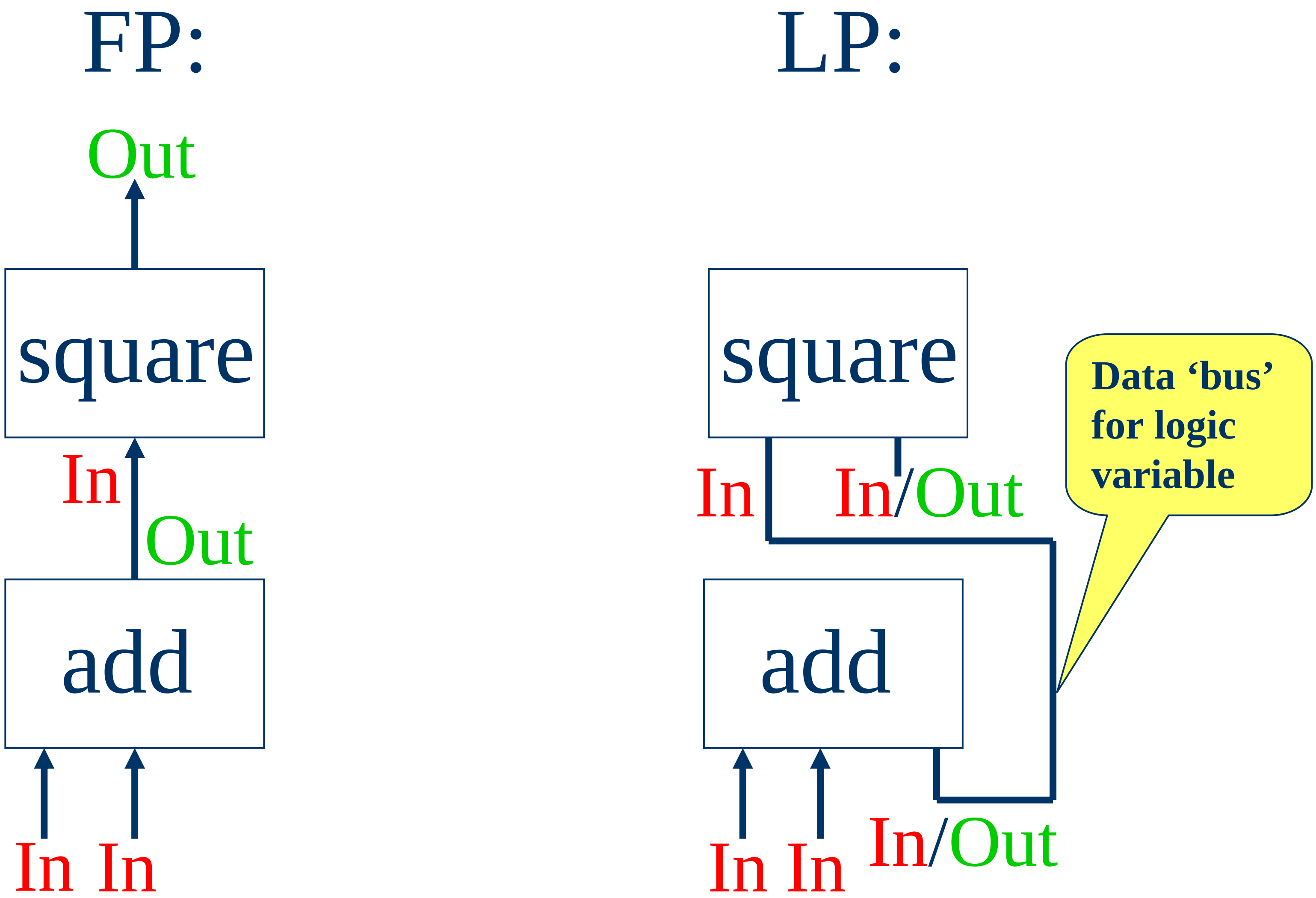
add ⊆ In × In × In/Out

Input-Output Traces:

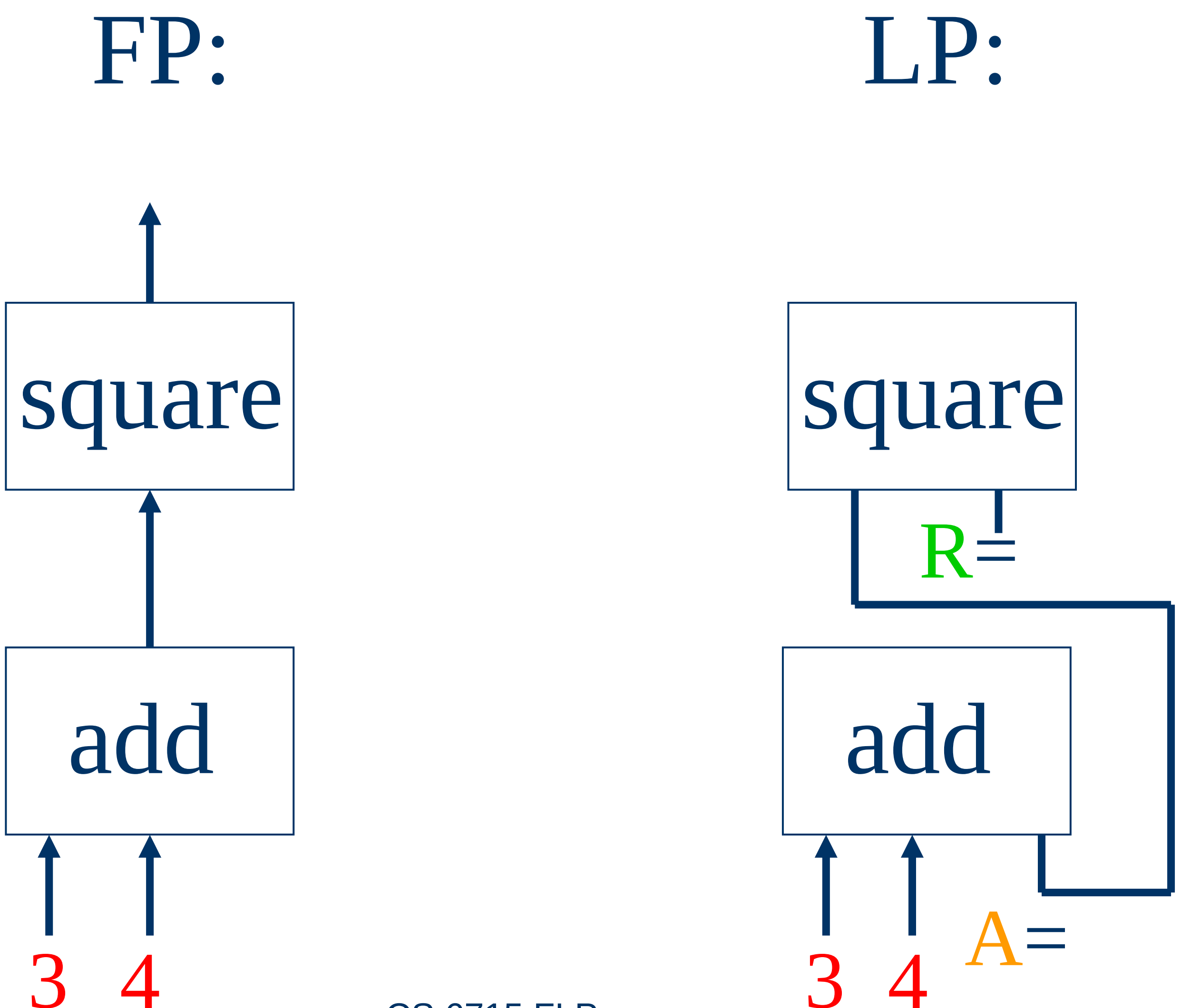
add(3, 4, A)
A=7

add(3, 4, 7)
success

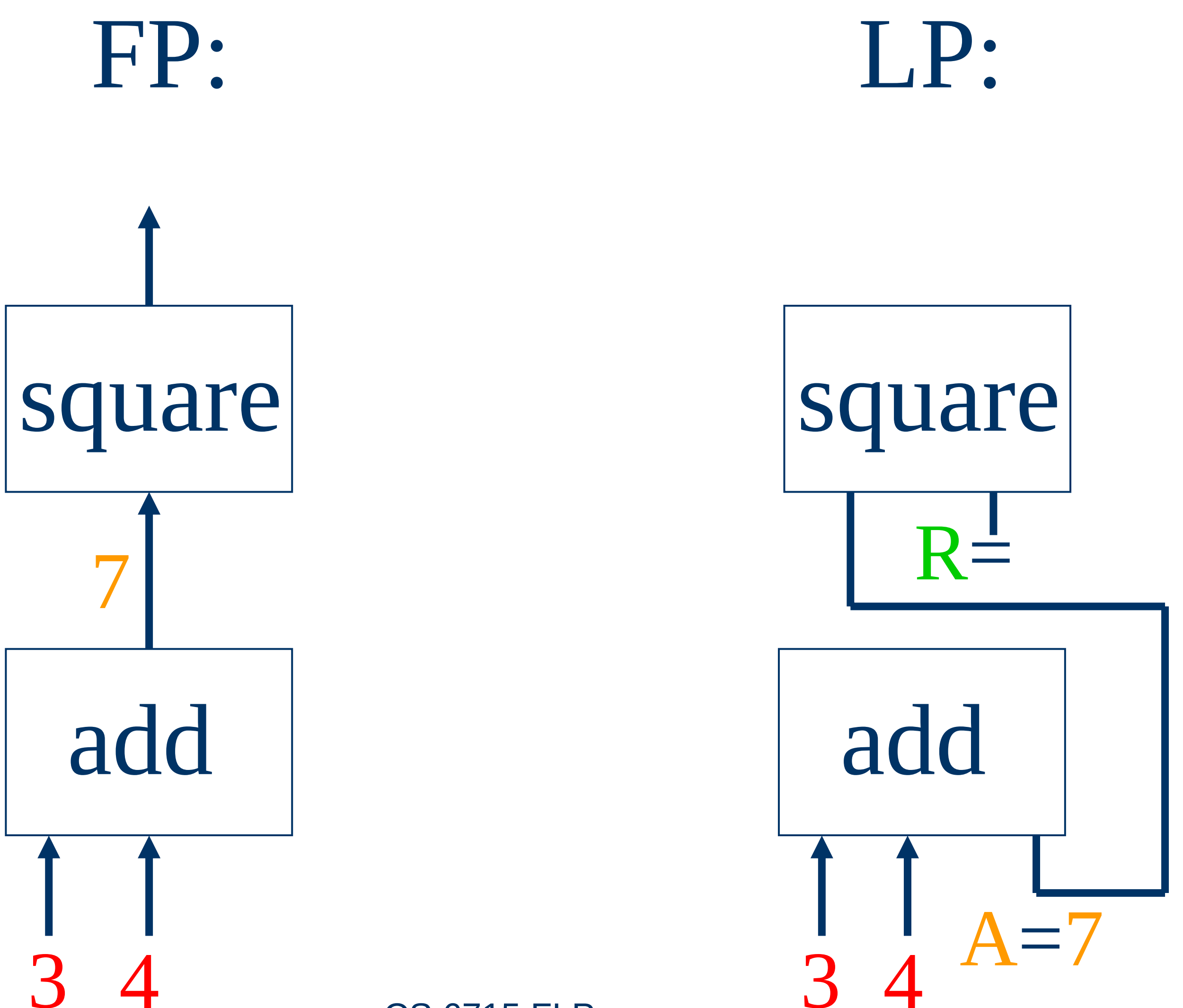
Declarative Programs as Data Flow Diagrams: Example – “Square-of-Add Agent” (Combination)



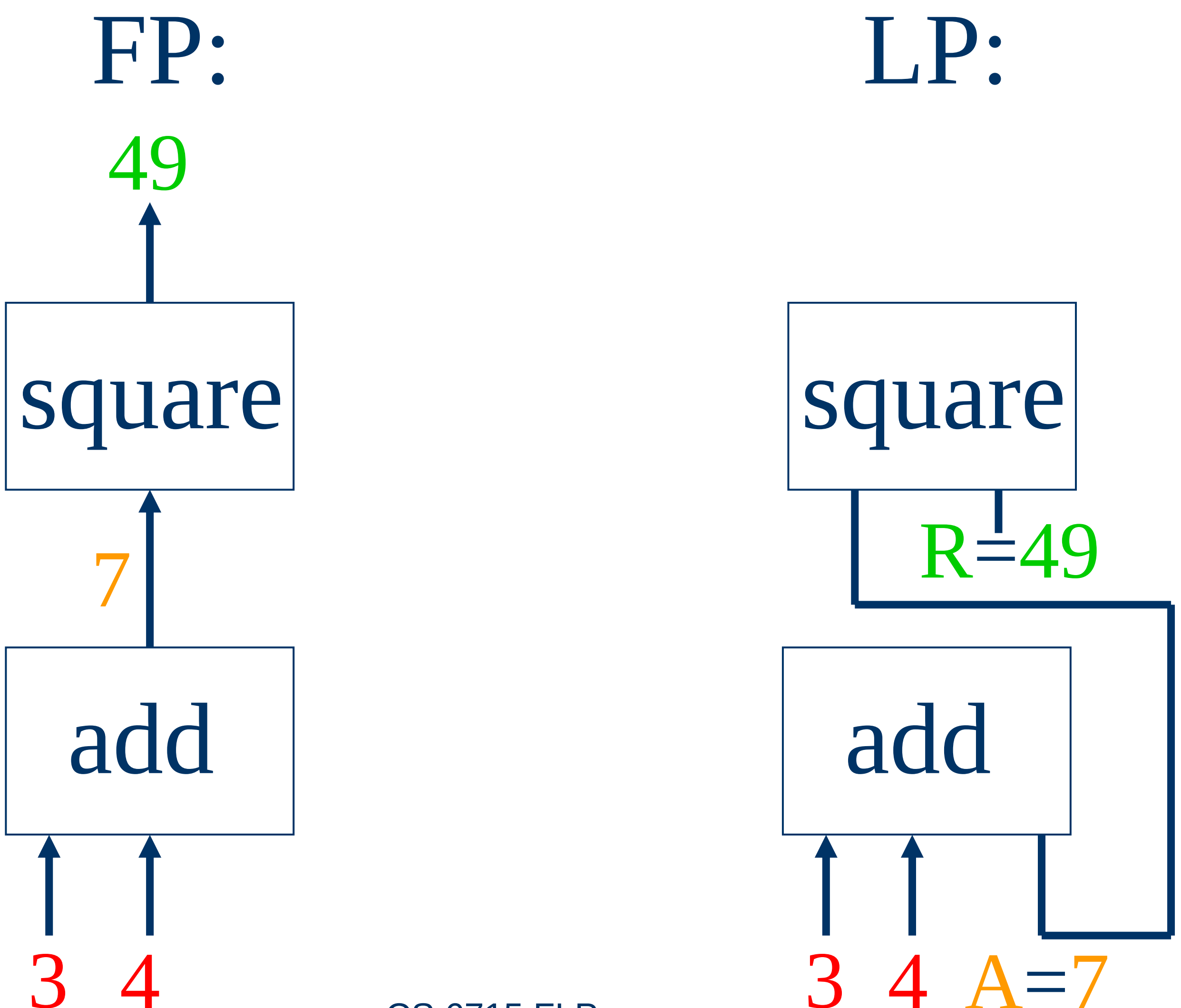
Declarative Programs as Data Flow Diagrams: Example – "Square-of-Add Agent" (Input)



Declarative Programs as Data Flow Diagrams: Example – “Square-of-Add Agent” (Thruput)



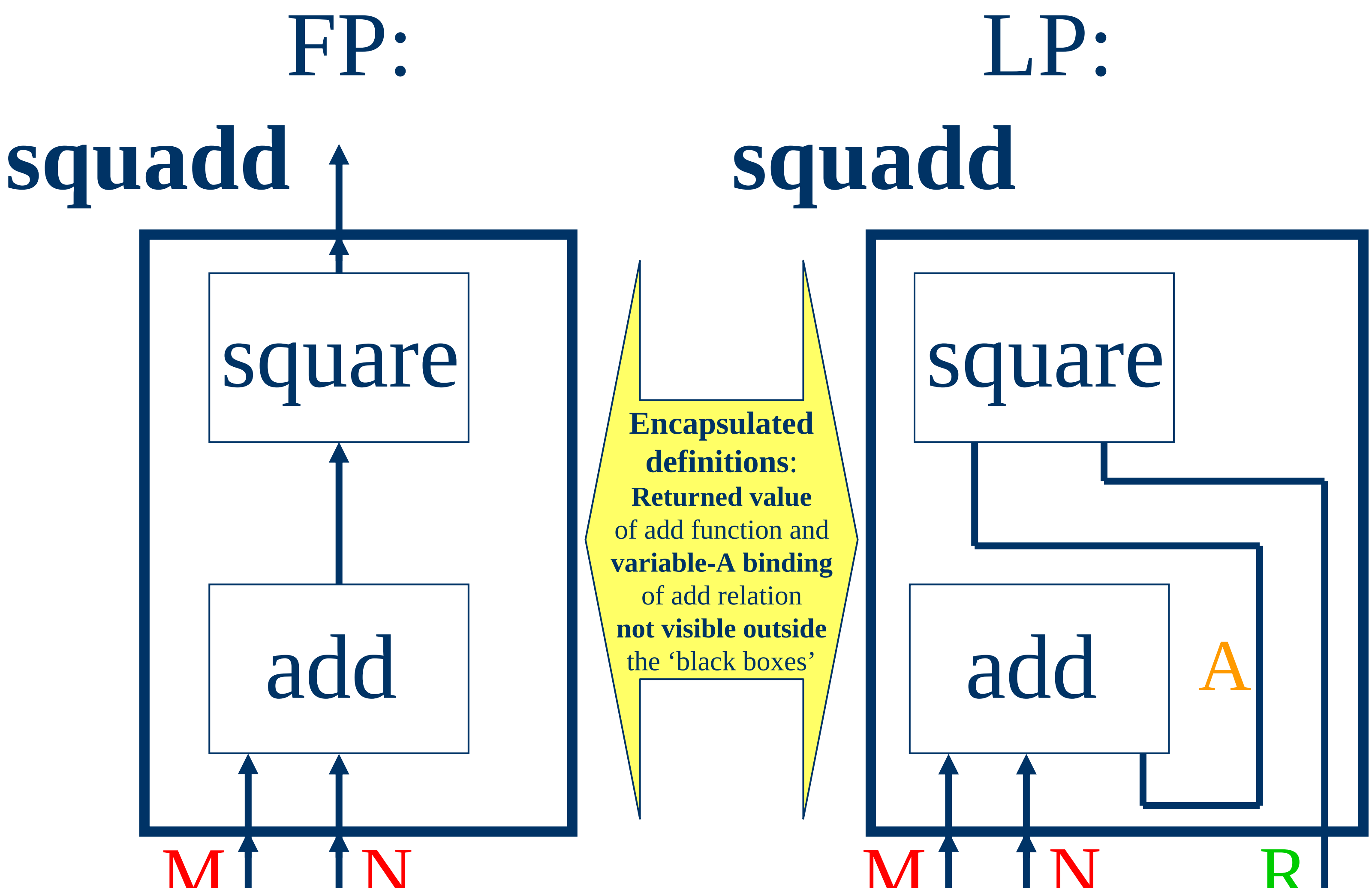
Declarative Programs as Data Flow Diagrams: Example – "Square-of-Add Agent" (Output)



Encapsulation Principle

- Functional-Logic Programming: New **operations** (functions and relations) become (user-)defined by **encapsulating** a combination of existing (built-in and/or user-defined) operations, and specifying the interface of that combination
- Functional-Logic Programs can be tested through queries before plugging them – often abstracted – into a 'body' conjunct (relational queries) or the 'foot' (functional queries) of a rule (a new program), encapsulating variables in the rule scope
- Goal: Referential Transparency → Compositionality (e.g. emphasized in a presentation by Tony Morris)

Declarative Programs as Data Flow Diagrams: Example – “Square-of-Add Agent” (Named)



Encapsulated definitions: **Returned value** of add function and **variable-A binding** of add relation **not visible outside** the ‘black boxes’

Declarative Programs in Symbolic Notation: Example – “Square-of-Add Agent”

FP:	LP:

Rewrite Traces of Unnamed Compound Agent:

FP	LP
square(add(3, 4))	add(3, 4, A), square(A, R)
square(7)	A=7: square(7, R)
49	A=7, R=49

Definitions of Named Compound Agent:

FP	LP
squadd(M, N) :&	squadd(M, N, R) :-
square(add(M, N)).	add(M, N, A), square(A , R).

Rewrite Traces of Named Compound Agent:

FP	LP
squadd(3, 4)	squadd(3, 4, R)
49	R=49

Syntax of Basic Declarative Definitions

FP:

Oriented Equation:

head = *foot*
written here as
head **:&** *foot*.

squadd(M, N) **:&**
  square(
    add(M, N)).

LP:

Implication:

head ⇐ *body*
written as Prolog-like
head **:-** *body*.

squadd(M, N, R) **:-**
  add(M, N, A),
  square(A, R).

FLP:

Conditional Oriented Equation (FP-LP Amalgamation):

head = *foot* ⇐ *body*
written as Prolog-extending
head **:-** *body* **&** *foot*.

squadd(M, N) **:-**
  add(M, N, A) **&**
  square(A).

Semantics of Purely Declarative Definitions

(Pure,1st-order) FP:

Horn logic with equality's **semantic structures including $I_=$ mapping**

(Pure) LP:

Horn logic's **semantic structures**

See RIF-BLD for FLP with **undirected (symmetric) equality**:
http://www.w3.org/2005/rules/wiki/BLD#Semantic_Structures

Can be specialized to **Herbrand semantic structures**

See RIF-FLD:
http://www.w3.org/2005/rules/wiki/FLD#Appendix:_A_Subframework_for_Herbrand_Semantic_Structures

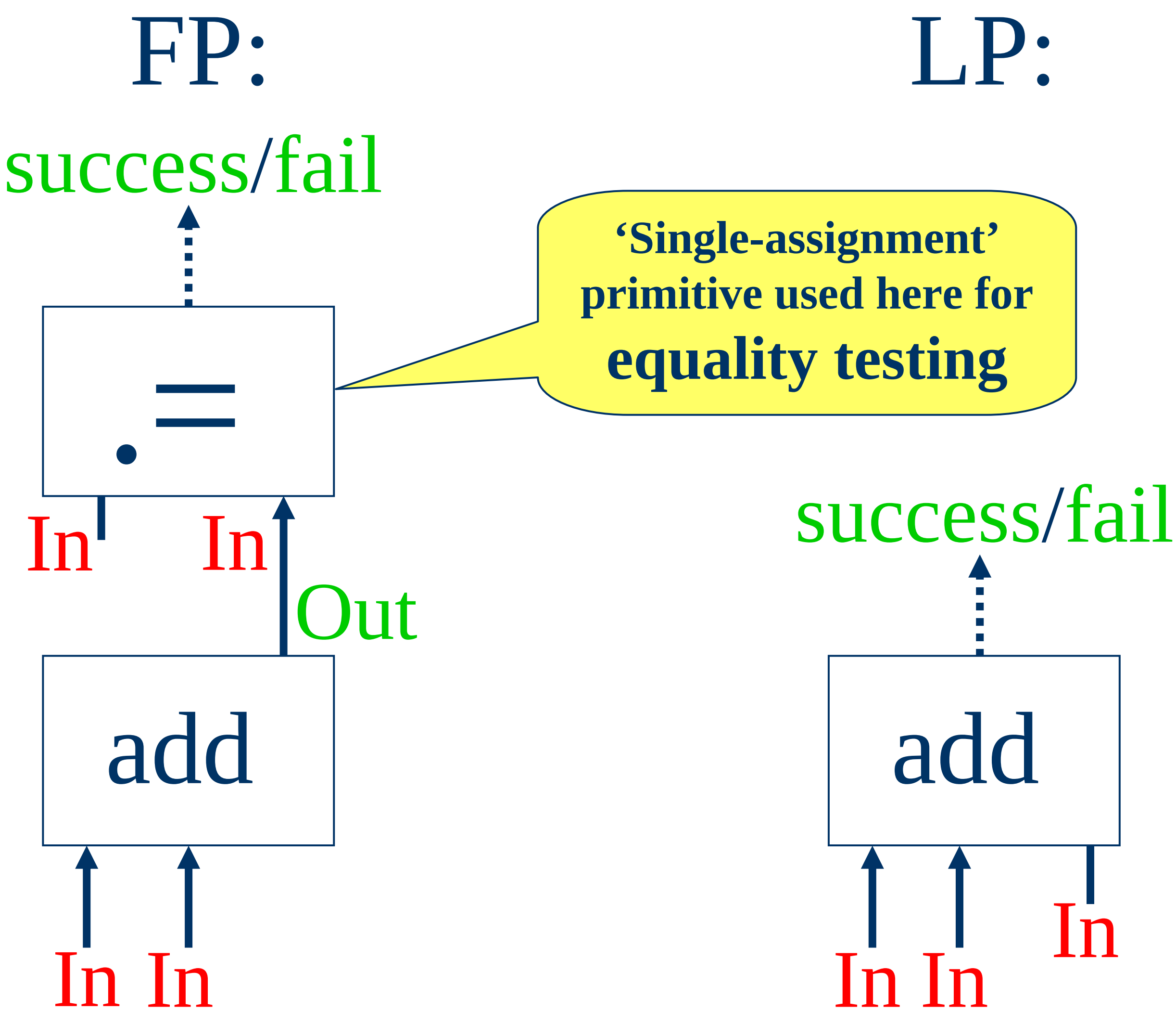
Is further specialized here to **directed (oriented) equality**

See Relfun:
http://www.cs.unb.ca/~boley/papers/semanticsb.pdf

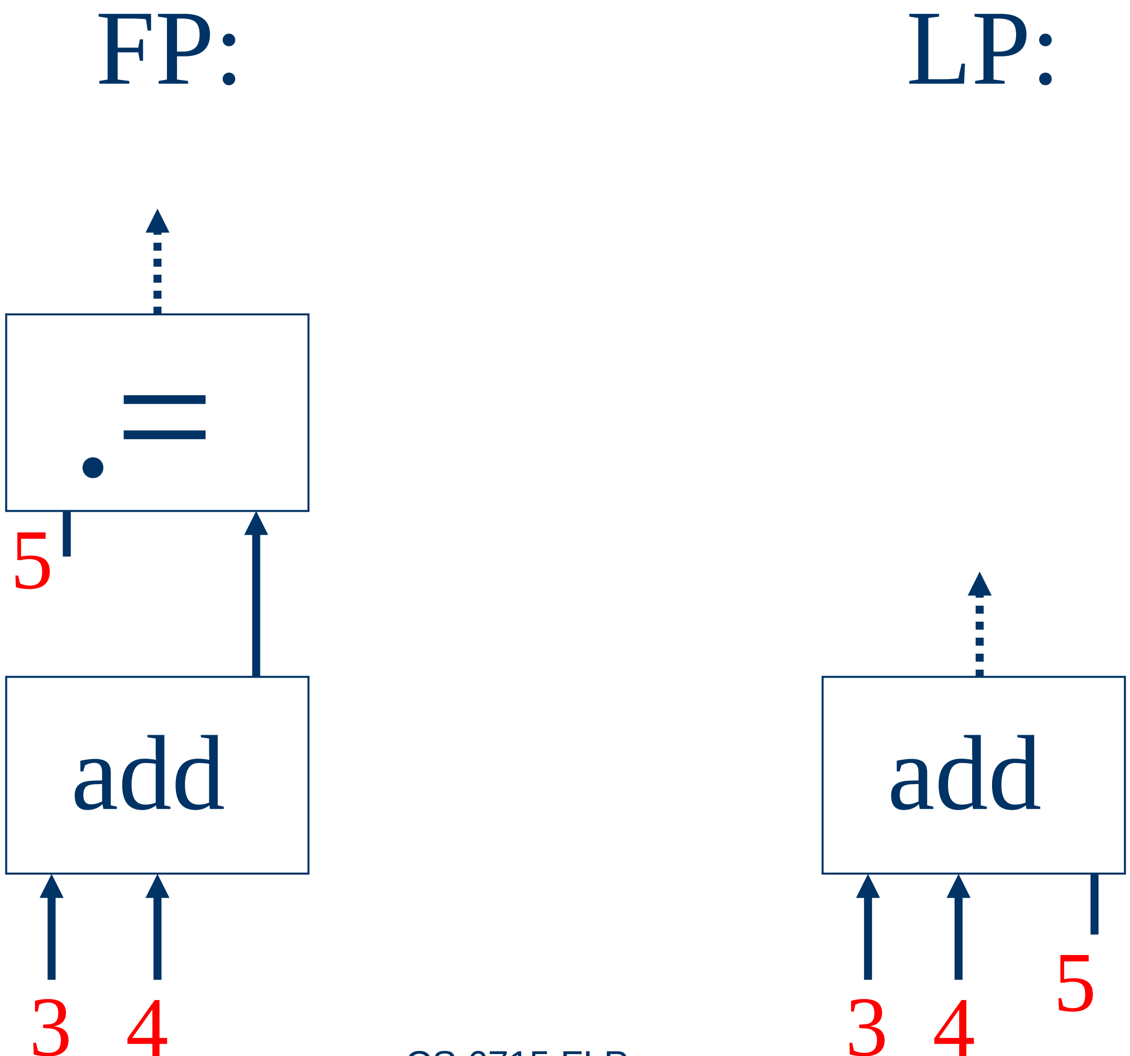
Generate-Test Separation/Integration Principle

- Functional Programming: Functions separate the generation of values from testing their equality
- Logic Programming: Relations integrate the generation and testing of their arguments

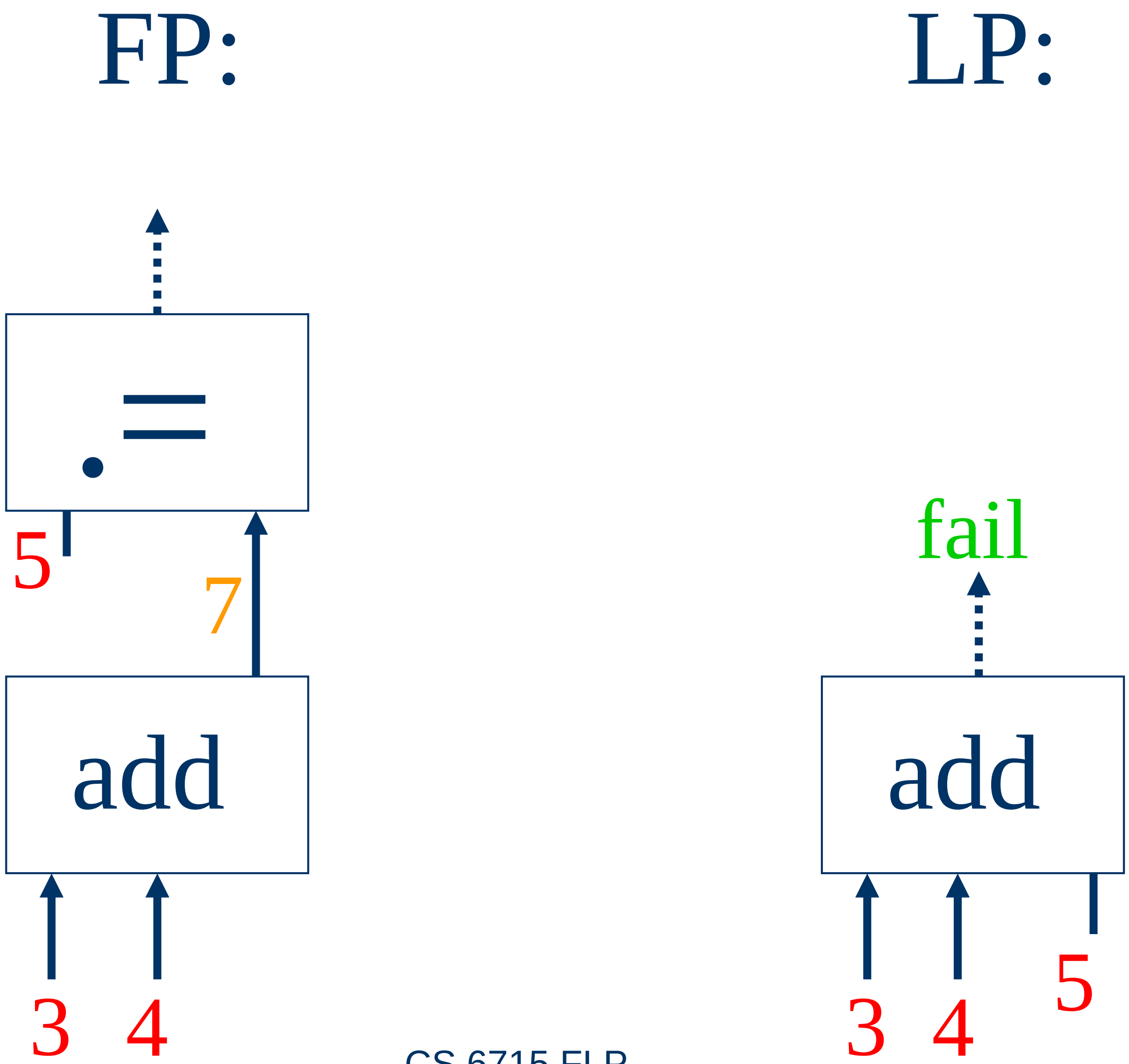
Declarative Programs Used for Testing: Example – “Addition Agent” (I-O Modes)



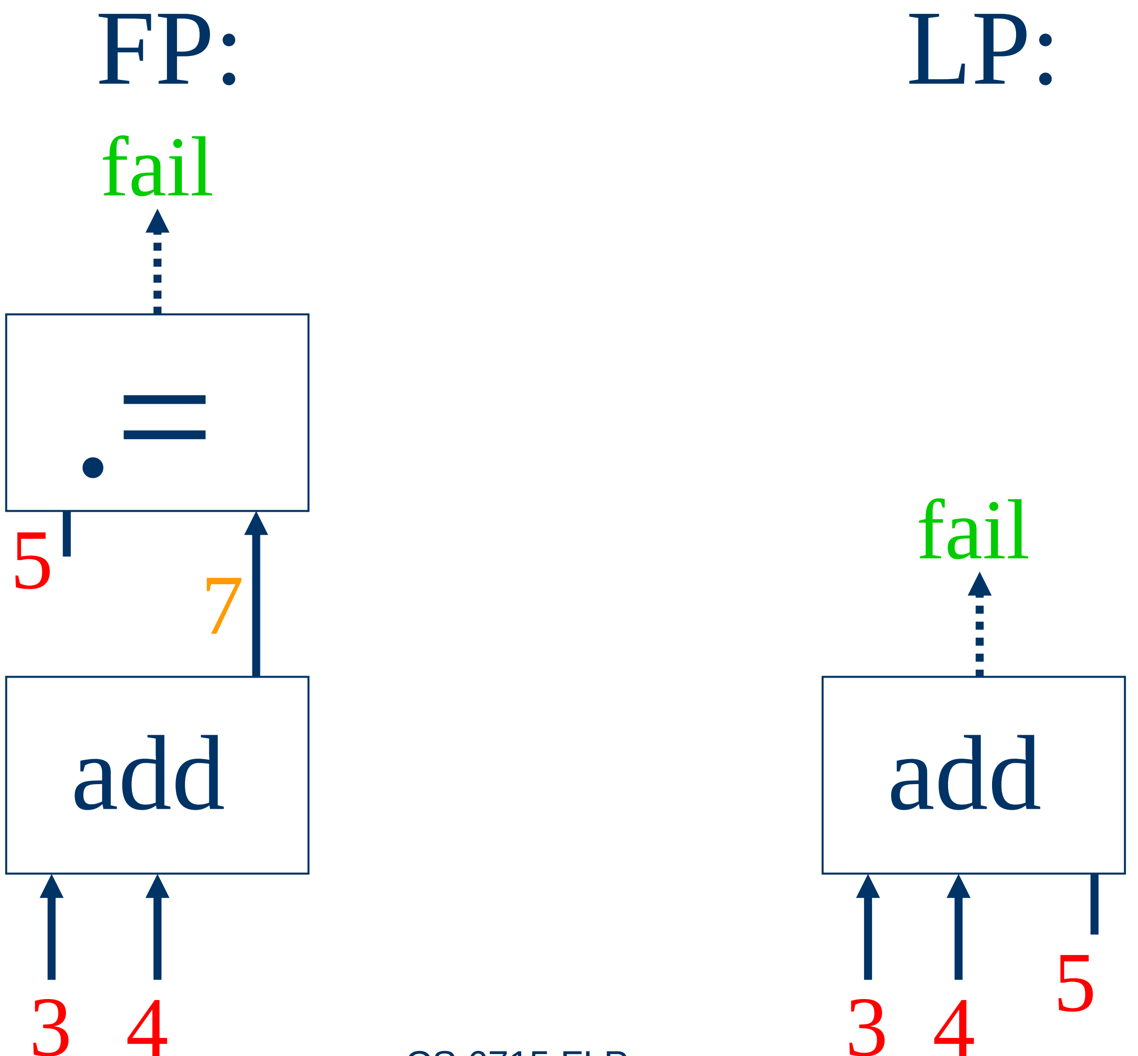
Declarative Programs Used for Testing: Example – “Addition Agent” (Input)



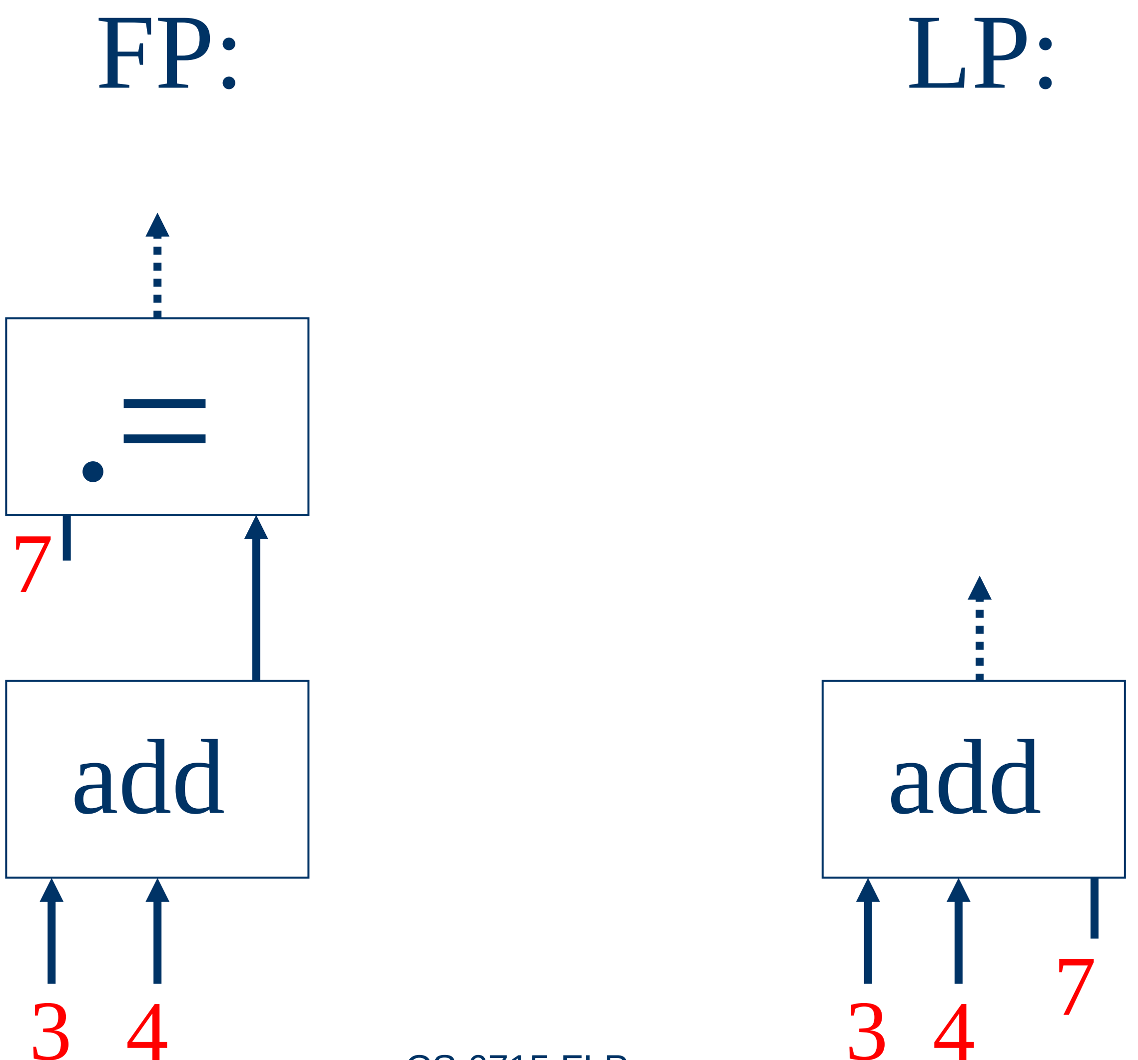
Declarative Programs Used for Testing: Example – “Addition Agent” (Thru/Output)



Declarative Programs Used for Testing: Example – “Addition Agent” (Output)

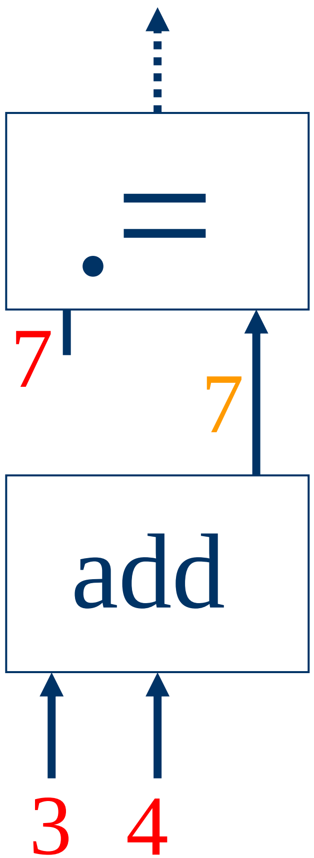


Declarative Programs Used for Testing: Example – “Addition Agent” (Input)

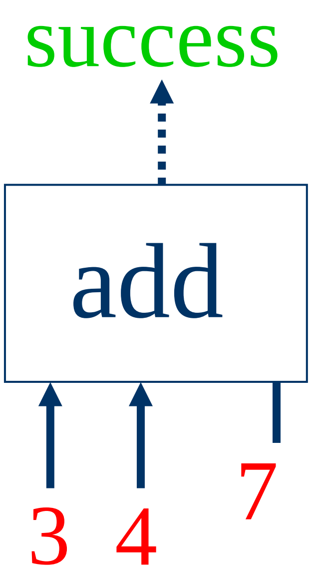


Declarative Programs Used for Testing: Example – “Addition Agent” (Thru/Output)

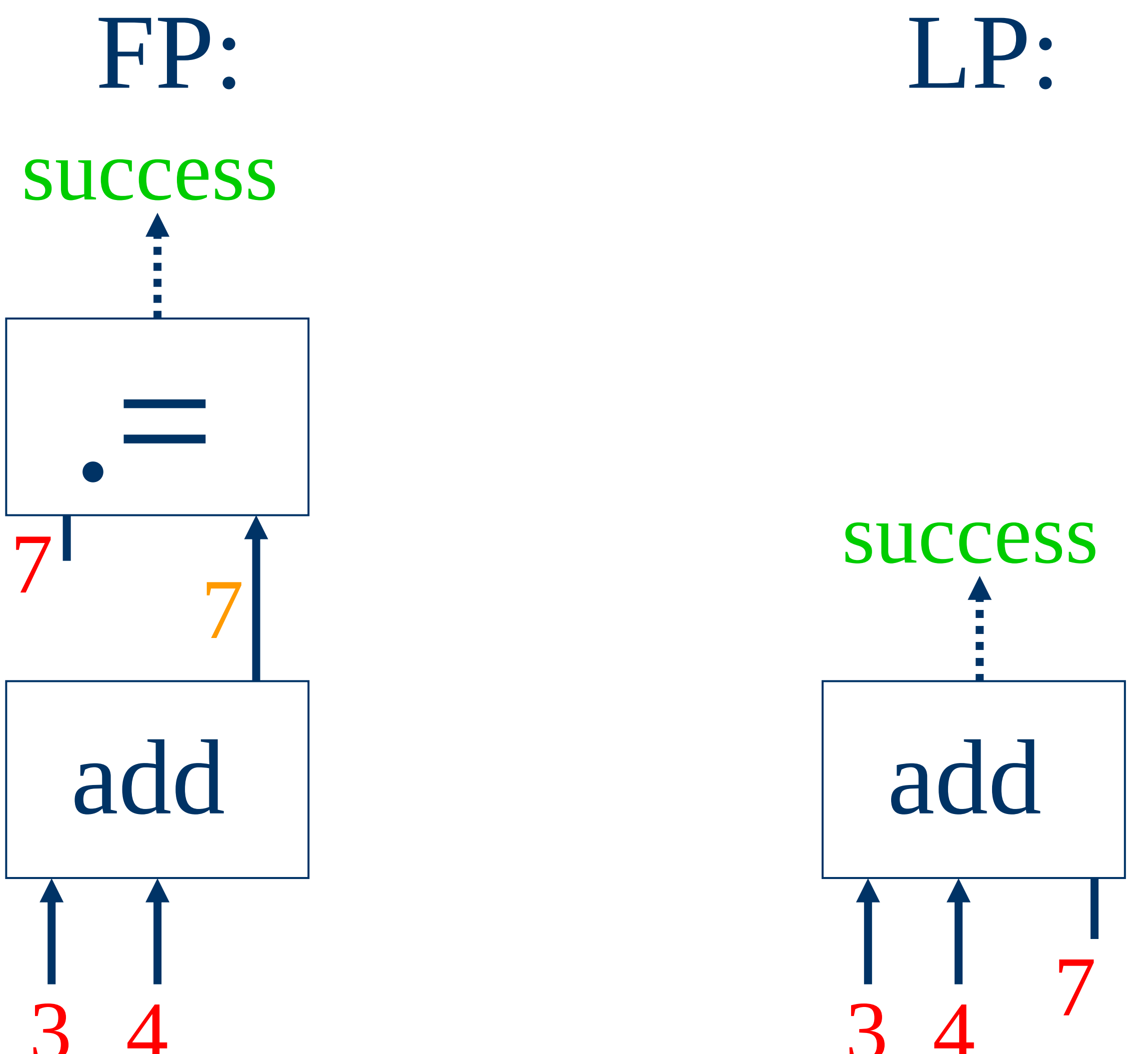
FP:



LP:



Declarative Programs Used for Testing: Example – “Addition Agent” (Output)



Declarative Testing Programs in Symbolic Notation: Example – “Addition Agent”

FP:

LP:

Rewrite Traces:

FP:

5 .= add(3, 4)
5 .= 7
fail

7 .= add(3, 4)
7 .= 7
success

LP:

add(3, 4, 5)
fail

add(3, 4, 7)
success

List-Universality Principle

- Functional-Logic Programming: (Nested) Lists are the universal ‘semi-structured’ **complex datatype** of declarative programming – predating XML trees.
- Functional-Logic Programming: Lists can be reduced to binary structures (see a later chapter)

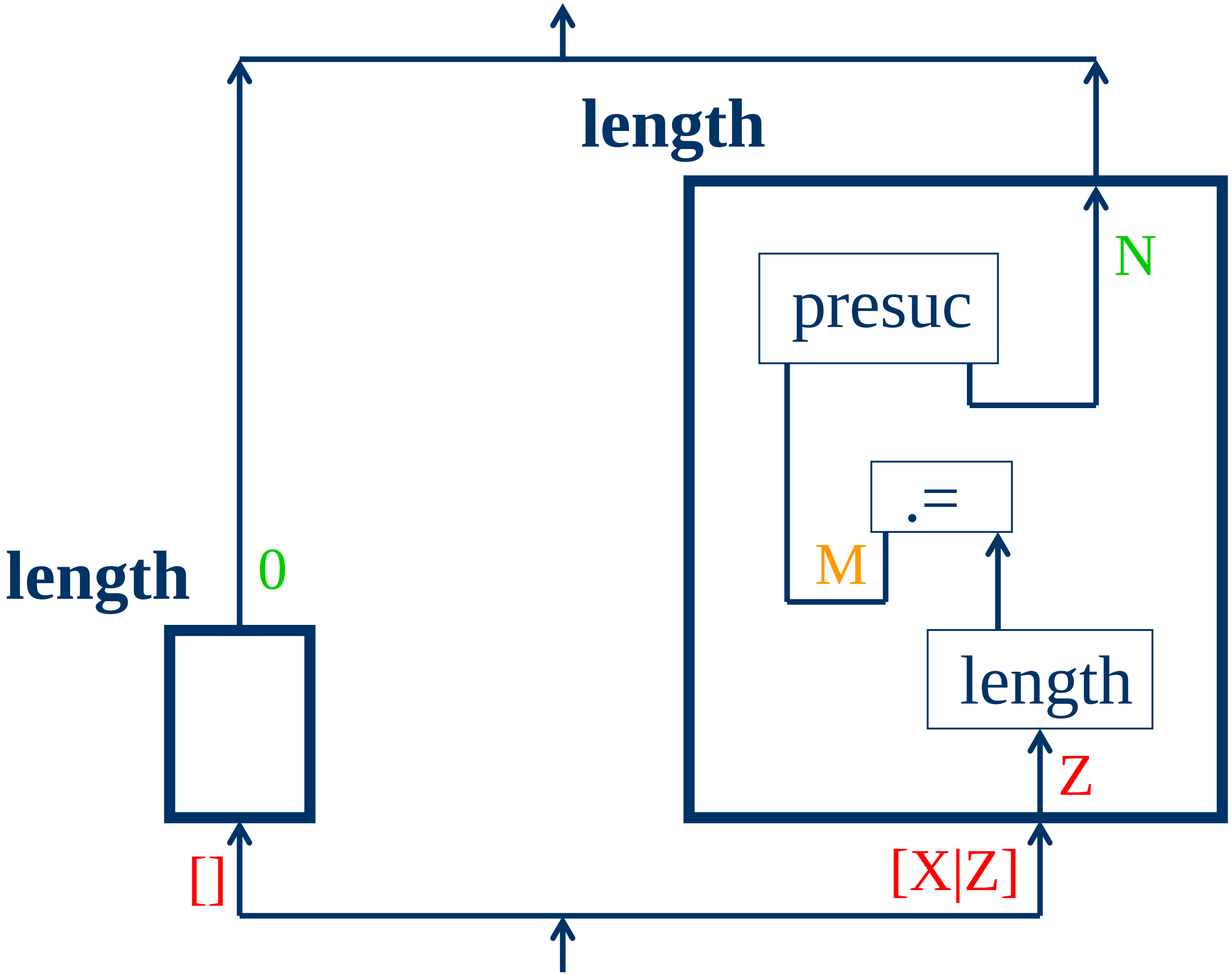
Declarative Programs Operating on Lists: Example “Length-and-Shape Agents”

- A *list* is a – comma-separated – finite sequence $e_1, e_2, \ldots, e_n$ of elements collected into a unit as a new – square-bracketed – element $[e_1, e_2, \ldots, e_n]$
- The *(natural-number) length* of a list $[e_1, e_2, \ldots, e_n]$ is the number n of its elements
- The *(list-pattern) shape* for a natural number n is a list $[x_1, x_2, \ldots, x_n]$ of n unspecified elements
- We now give declarative “Length-Shape Agents” as a functional program length and its (non-ground, here pattern-valued) functional ‘inverse’ shape, and then as a single logic program shalen
- The following chapters study the FP/LP trade-offs

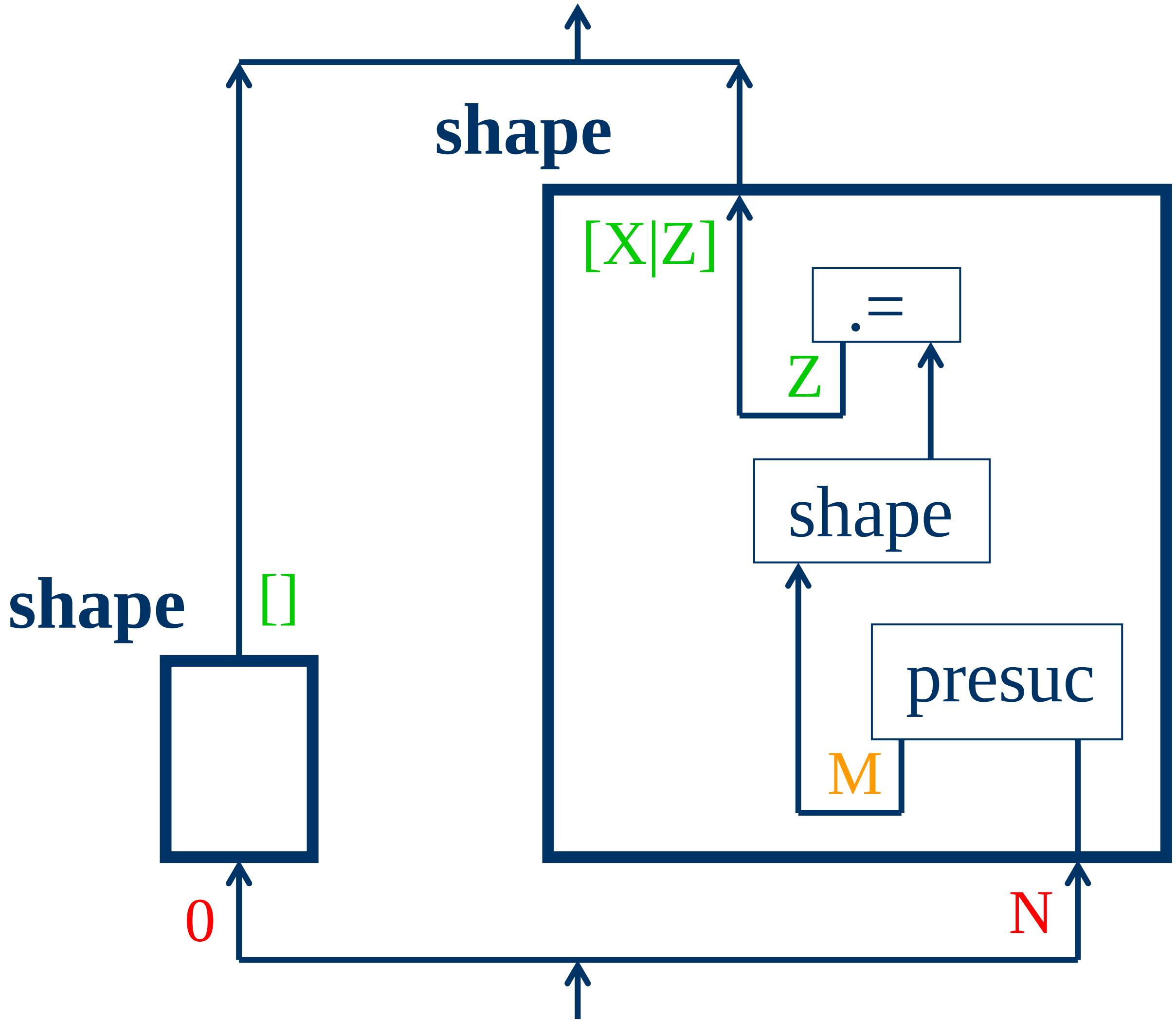
Invertibility Principle

- Functional Programming: A function and its inverses are usually specified via **multiple definitions**
- Pure Logic Programming: A relation and its inverses are usually specified via a **single definition**

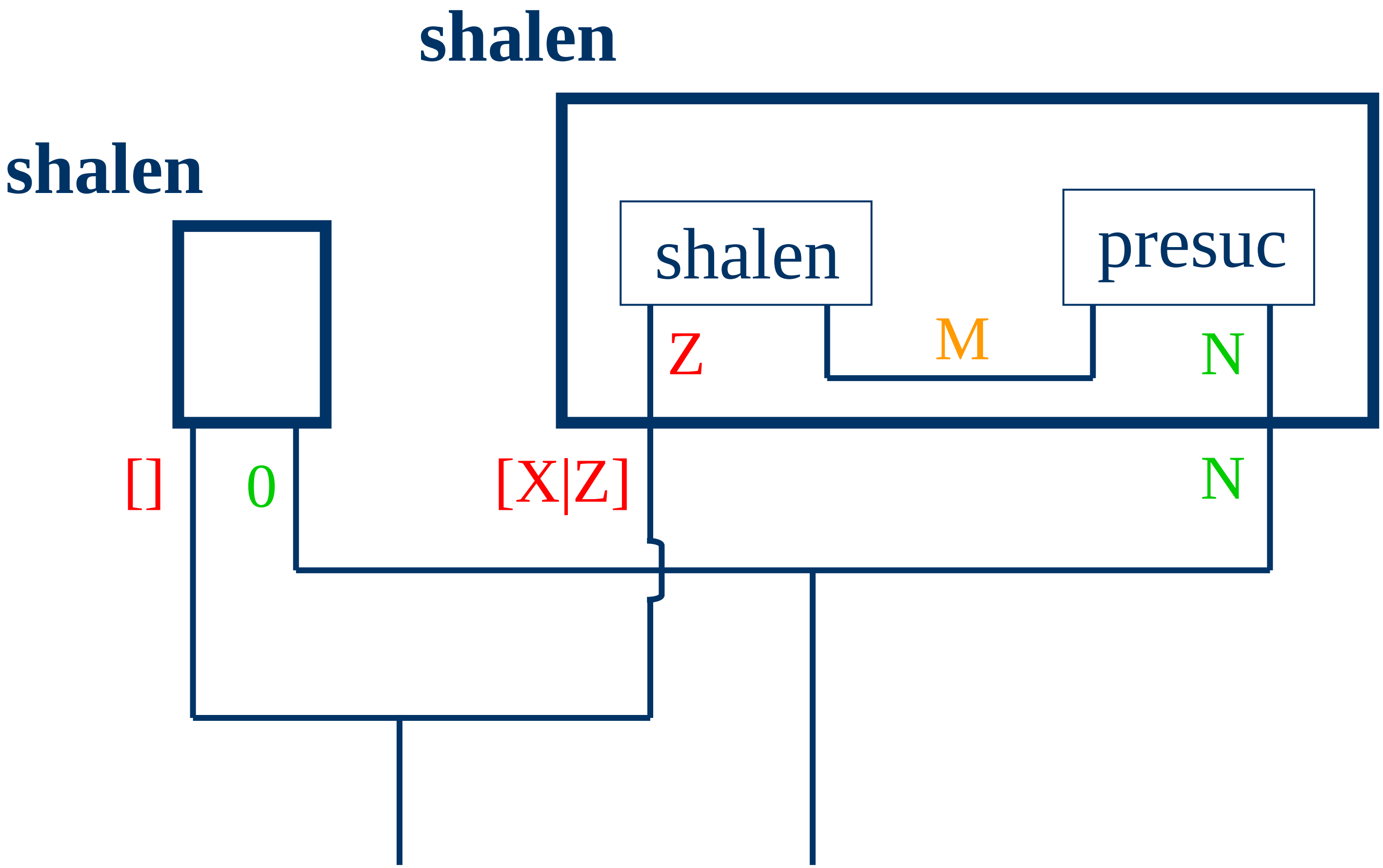
Function length as Data Flow Diagram



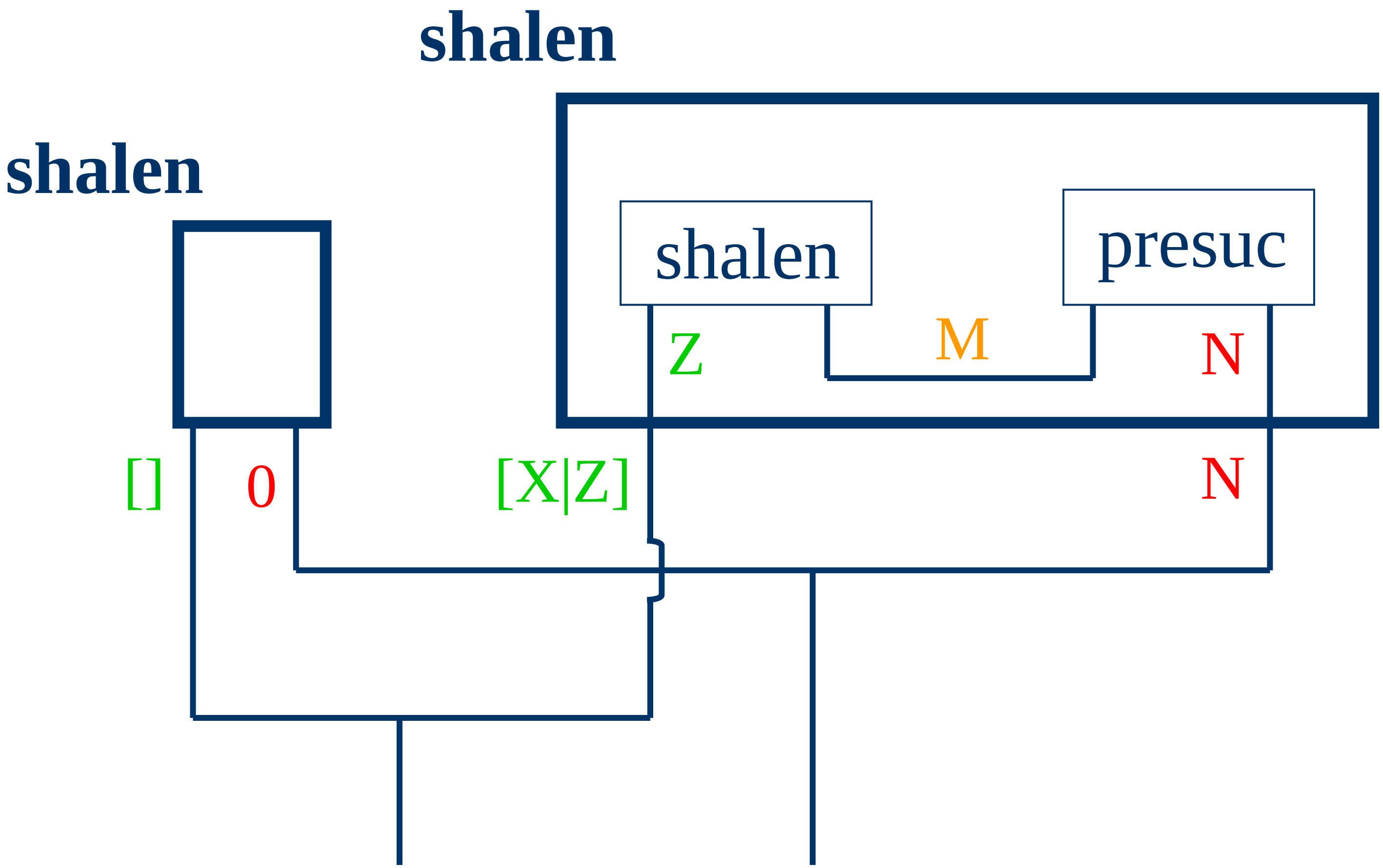
Function shape as Data Flow Diagram



Relation shalen as Data Flow Diagram



Relation shalen as Data Flow Diagram



Functional Programs length and shape Become One Logic Program shalen

Functional program (normal)

```
length([]) :& 0.
length([X|Z]) :- M .= length(Z), presuc(M,N) & N.
```

'First,Second,...|Rest' list pattern

Functional program ('inverse')

```
shape(0) :& [].
shape(N) :- presuc(M,N), Z .= shape(M) & [X|Z].
```

If body conjunction succeeds, return foot

Logic program (both directions)

```
shalen([],0).
shalen([X|Z],N) :- shalen(Z,M), presuc(M,N).
```

Logic program (auxiliary)

```
presuc(0,1).
presuc(1,2).
presuc(2,3).
. . .
```

Alternatively (always giving one solution):

```
presuc(M,N) :- nonvar(M) ! N .= 1+(M).
presuc(M,N) :- nonvar(N) ! M .= 1-(N).
presuc(0,1).
```

Functional Programs length and shape Become One Logic Program shalen

Functional program (normal)

length([]) :& 0.
length([X|Z]) :- M .= length(Z), presuc(M,N) & N.

'First,Second,...|Rest' list pattern

Functional program ('inverse')

shape(0) :& [].
shape(N) :- presuc(M,N), Z .= shape(M) & [X|Z].

If body conjunction succeeds, return foot

Logic program (both directions)

shalen([],0).
shalen([X|Z],N) :- shalen(Z,M), presuc(M,N).

Logic program (auxiliary)

presuc(0,1).
presuc(1,2).
presuc(2,3).
. . .

Alternatively (always giving one solution):
presuc(M,N) :- nonvar(M) ! N .= 1+(M).
presuc(M,N) :- nonvar(N) ! M .= 1-(N).
presuc(0,1).

Computation with Functional Program length as Term Rewriting: Stack Trace

Functional program (normal)

length([]) :& 0.
length([X|Z]) :- M .= length(Z), presuc(M,N) & N.

Functional trace

length([a,b,c]) → 3
length([b,c]) → 2
length([c]) → 1
length([]) → 0

Logic program (auxiliary)

presuc(0,1).
presuc(1,2).
presuc(2,3).
. . .

Alternatively (always giving one solution):
presuc(M,N) :- nonvar(M) ! N .= 1+(M).
presuc(M,N) :- nonvar(N) ! M .= 1-(N).
presuc(0,1).

Computation with Functional Program shape as Term Rewriting: Stack Trace

Functional program ('inverse')

shape(0) :& [].
shape(N) :- presuc(M,N), Z .= shape(M) & [X|Z].

Functional trace

shape(3) → [X',X'',X''']
shape(2) → [X'',X''']
shape(1) → [X''']
shape(0) → []

Logic program (auxiliary)

presuc(0,1).
presuc(1,2).
presuc(2,3).
. . .

Alternatively (always giving one solution):
presuc(M,N) :- nonvar(M) ! N .= 1+(M).
presuc(M,N) :- nonvar(N) ! M .= 1-(N).
presuc(0,1).

Computations with Logic Program shalen as Term Rewriting: Stack Traces

Logic program (both)

shalen([],0).
shalen([X|Z],N) :- shalen(Z,M), presuc(M,N).

Logic traces

	I		L
↓	shalen([a,b,c],3)	↑ ↑	shalen([X',X'',X'''],3)
	shalen([b,c],2)		shalen([X'',X'''],2)
	shalen([c],1)		shalen([X'''],1)
	shalen([],0)		shalen([],0) ↓

Logic program (auxiliary)

presuc(0,1).
presuc(1,2).
presuc(2,3).
. . .

Alternatively (always giving one solution):
presuc(M,N) :- nonvar(M) ! N .= 1+(M).
presuc(M,N) :- nonvar(N) ! M .= 1-(N).
presuc(0,1).

Nesting/Conjunction Principle

- Functional-Logic Programming: Properties of **functional nestings** correspond to properties of **relational conjunctions** (to be exemplified with generalized inverse properties)

Generalized Inverse Property of the Functional Programs length and shape (I)

General – Nestings:

length(shape(*n*)) = *n*

shape(length([e_1 , e_2 , …, e_n])) = $[X' , X'' , \ldots, X^{\overbrace{'\cdots'}^{n}}]$

Most general pattern
for lists of length *n*

Examples – Nestings:

length(shape(3)) = 3

shape(length([a,b,c])) = [X', X'', X''']

Generalized Inverse Property of the Functional Programs length and shape (II)

General – Nestings Flattened to Conjunctions:

L .= shape(*n*) & length(L) = *n*

I .= length([e_1 , e_2 , …, e_n]) & shape(I) = $[X' , X'' , \ldots, X\overbrace{'\cdots'}^{n}]$

Most general pattern for lists of length *n*

Examples – Nestings Flattened to Conjunctions:

L .= shape(3) & length(L) = 3

I .= length([a,b,c]) & shape(I) = [X', X'', X''']

Generalized Self-Inverse Property of the Logic Program shalen

General – Conjunctions:

shalen(L,n), shalen(L,I) binds I = n

shalen([e_1 , e_2 , …, e_n],I), shalen(L,I) binds
L= [X' , X'' , …, $X^{\overbrace{'\cdots'}^{n}}$]

Most general pattern for lists of length n

Examples – Conjunctions:

shalen(L,3), shalen(L,I) binds I = 3

shalen([a,b,c],I), shalen(L,I) binds L = [X', X'', X''']

Unification Principle

- Logic Programming: Uses **unification** to equate, analyze, and refine complex data structures, in particular lists; also – with programs used as data – for invoking operations
- Functional Programming: Can generalize asymmetric pattern-instance **matching** to symmetric pattern-pattern **unification** as in Logic Programming

Duplication of Non-Ground List Values: Generating Matrix Patterns with shalen (I)

(m,n)-Matrices of Equal Rows:

$$\text{shalen}(L,n)\ \&\ [\underbrace{L, \ldots, L}_{m}] \;=\; \left.\begin{array}{l}[[X', X'', \ldots, \overbrace{X'\ldots'}^{n}] \\ \qquad\quad \ldots \\ \;[X', X'', \ldots, \overbrace{X'\ldots'}^{n}]]\end{array}\right\}m$$

(2,3)-Matrices of Equal Rows:

$$\text{shalen}(L,3)\ \&\ [L,L] \;=\; \begin{array}{l}[[X', X'', X'''], \\ \;[X', X'', X''']]\end{array}$$

Refinement of Non-Ground List Values: Generating Matrix Patterns with shalen (II)

(m,n)-Matrices of Equal Rows and $1^{st} = 2^{nd}$ Column:

$$\text{shalen}(L, n),\ [C,C|R] \mathrel{.=} L\ \&\ \underbrace{[L, \ldots, L]}_{m} \;=\; \left.\begin{array}{l} [[X'', X'', \ldots, \overbrace{X'\ldots'}^{n}] \\ \qquad\quad \ldots \\ \ [X'', X'', \ldots, \overbrace{X'\ldots'}^{n}]] \end{array}\right\} m$$

‘Single-assignment’ primitive used for unification

(2,3)-Matrices of Equal Rows and $1^{st} = 2^{nd}$ Column:

$$\text{shalen}(L, 3),\ [C,C|R] \mathrel{.=} L\ \&\ [L, L] \;=\; \begin{array}{l} [[X'', X'', X'''], \\ \ [X'', X'', X''']] \end{array}$$

Refinement of Non-Ground List Values: Generating Matrix Patterns with shalen (III)

(m,n)-Matrices of Equal Rows and $1^{st} = 3^{rd}$ Column:

$$\text{shalen}(L,n),\ [D,A,D|S]\ .\!=\ L\ \&\ [\underbrace{L, \ldots, L}_{m}]\ = \left.\begin{array}{l} [[X''', X'', X''', \ldots, \overbrace{X'\ldots'}^{n}] \\ \qquad \cdot\quad \cdot\quad \cdot \\ \ [X''', X'', X''', \ldots, \overbrace{X'\ldots'}^{n}]] \end{array}\right\} m$$

‘Single-assignment’ primitive used for unification

(2,3)-Matrices of Equal Rows and $1^{st} = 3^{rd}$ Column:

$$\text{shalen}(L,3),\ [D,A,D|S]\ .\!=\ L\ \&\ [L,L]\ = \begin{array}{l} [[X''', X'', X'''], \\ \ [X''', X'', X''']] \end{array}$$

Double Refinement of Non-Ground List Values: Generating Matrix Patterns with shalen (IV)

(m,n)-Matrices of Equal Rows and $1^{st}=2^{nd}=3^{rd}$ Column:

shalen(L,*n*), [C,C|R] .= L,

$$[D,A,D|S] \;.\!= L \;\&\; \underbrace{[L, \ldots, L]}_{m} \;=\; \left.\begin{array}{l} [[\overbrace{X''', X''', X''', \ldots, X'\ldots'}^{n}] \\ \quad\quad . \;\; . \;\; . \\ \;[\overbrace{X''', X''', X''', \ldots, X'\ldots'}^{n}]] \end{array}\right\} m$$

‘Single-assignment’ primitive used for **unification**

(2,3)-Matrices of Equal Rows and $1^{st}=2^{nd}=3^{rd}$ Column:

shalen(L,3), [C,C|R] .= L,

[D,A,D|S] .= L & [L,L] = [[X''', X''', X'''],
[X''', X''', X''']]

Amalgamation/Integration Principle

- Functional-Logic **Amalgamation**: Function and relation calls can be combined in the same definition where appropriate
- Functional-Logic **Integration**: Functions and relations can inherit each others' expressiveness; e.g., in FLP certain functions – even when mapping from ground (variablefree) lists to ground lists – can be more easily defined using intermediate non-ground lists (generally, partial data structures), as pioneered by relation definitions in LP
  - Partial data structures may be dynamically generated with fresh variables that make operation calls succeed (paradigm: zip or pairlists function)

Functional-Relational Call Amalgamation: Quicksort Example

Directed, Conditional Equations:

```
qsort([]) :& [].
qsort([X|Y]) :-
   partition(X,Y,Sm,Gr) &
   cat(qsort(Sm),tup(X|qsort(Gr))).
```

Subrelation call with two output variables

Subfunction call with two embedded calls becomes value of main function call

Rules and Fact:

```
partition(X,[Y|Z],[Y|Sm],Gr) :-
   <(Y,X),  partition(X,Z,Sm,Gr).
partition(X,[Y|Z],Sm,[Y|Gr]) :-
   <(X,Y),  partition(X,Z,Sm,Gr).
partition(X,[X|Z],Sm,Gr) :-
   partition(X,Z,Sm,Gr).
partition(X,[],[],[]).
```

'Duplicates' eliminated

Auxiliary Function (append or catenate):

```
cat([],L) :& L.
cat([H|R],L) :& tup(H|cat(R,L)).
```

Higher-Order Operations Defined: Quicksort Parameterized by Comparison Relation

Functional and relational arguments plus values. User-defined comparison relations Cr. **Restriction** to *named* functions and relations (no λ-expressions), as they are dominant in practice and more easily integrated (avoids λ/logic-variable distinction and higher-order unification): `apply`-*reducible* to 1st order.

```
qsort[Cr]([X|Y]) :-
  partition[Cr](X,Y,Sm,Gr) &
  cat(qsort[Cr](Sm),tup(X|qsort[Cr](Gr))).

partition[Cr](X,[Y|Z],[Y|Sm],Gr) :-
  Cr(Y,X),  partition[Cr](X,Z,Sm,Gr).
. . .

before([X1,Y1],[X2,Y2]) :- string<(X1,X2).
```

Comparison relation handed through here

Comparison relation becomes called there

Higher-Order Operations Called: Quicksort Parameterized by Comparison Relation

Cr bound to `<`:

>>>>>> qsort[<]([3,1,4,2,3])

[1,2,3,4]

Cr bound to `before`:

>>>>>> qsort[before]([[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]])

[[a,Y2],[d,Y1],[l,Y3],[s,Y6]]

Y4=Y3

Y5=Y2

Logic Variables and Non-Ground Terms: pairlists Example

Function calls can – like relation calls – use (free) logic variables as actual **arguments** and, additionally, return them as **values**. Likewise, *non-ground terms*, which contain logic variables, are permitted. Processing is based on unification: Call with R creates inner Y1,Y2, ..., used as 2nd pair elements

```
pairlists([],[]) :& [].
pairlists([X|L],[Y|M]) :&
  tup([X,Y]|pairlists(L,M)).
```

Non-ground pair list term ('partial data structure') containing six logic variables

```
>>>>>> pairlists([d,a,l,l,a,s],R)
[[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]]
R=[Y1,Y2,Y3,Y4,Y5,Y6]
```

Flat list of these logic variables

Function Calls Nested in Operation Calls: numbered Example

Call-by-value nestings allow (built-in and user-defined) functions to be nested into other such functions or relations. Built-in function + nested here into user-defined relation numbered

Instantiate logic variables in 2nd pair elements with successive integers initialized by main call

```
numbered([],N).
numbered([[X,N]|R],N) :- numbered(R,+(N,1)).
```

```
>>>>>> numbered([[a,Y2],[d,Y1],[l,Y3],[s,Y6]],1)
true
Y2=1,   Y1=2,   Y3=3,   Y6=4
```

Integrated Functional-Logic Programming Using Intermediate Non-Ground Terms: `serialise` Example

Task (*D.H.D. Warren, L.M. Pereira, F. Pereira 1977*)*:*
Transform a list of symbols into the list of their lexicographic serial rank numbers
Example: [d,a,l,l,a,s] → [2,1,3,3,1,4]

Specific Solution for Example:

>>>>>> numbered(qsort[before](pairlists([d,a,l,l,a,s],R)),1)
& R
[2,1,3,3,1,4], R=[2,1,3,3,1,4]

General Solution by Abstraction [d,a,l,l,a,s] = L*:*

```
serialise(L) :-
  numbered(qsort[before](pairlists(L,R)),1)
  & R.
```

Derivation of the `serialise` Solution

```
>>>>>> pairlists([d,a,l,l,a,s],R)
[[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]]
R=[Y1,Y2,Y3,Y4,Y5,Y6]
```

Intermediate non-ground pair list (unsorted)

Non-ground result list ‘waiting’ for bindings

```
>>>>>> qsort[before]([[d,Y1],[a,Y2],[l,Y3],[l,Y4],[a,Y5],[s,Y6]])
[[a,Y2],[d,Y1],[l,Y3],[s,Y6]]
Y4=Y3
Y5=Y2
```

Intermediate non-ground pair list (sorted, w/o ‘duplicates’)

```
>>>>>> numbered([[a,Y2],[d,Y1],[l,Y3],[s,Y6]],1)
true
Y2=1,   Y1=2,   Y3=3,   Y6=4
```

Bindings of inner variables produced

```
serialise([d,a,l,l,a,s]) :-
          numbered(qsort[before](pairlists([d,a,l,l,a,s],R)),1)
          & R
[2,1,3,3,1,4]
```

Bindings used for result list instantiation

Online Execution of `serialise` Specification: `serialise([d,a,l,l,a,s,t,e,x,a,s,u,s,a])`

R E L F U N Interface Page

(http://www.dfki.uni-kl.de/~vega/relfun-cgi/cgi-bin/rfi.cgi)

Database: PROLOG Syntax

```
t1() :& serialise([d,a,l,l,a,s]).
t2() :& serialise([d,a,l,l,a,s,t,e,x,a,s,u,s,a]).

serialise(L) :-
  numbered(qsort[before](pairlists(L,R)),1)
  & R.

pairlists([],[]) :& [].
pairlists([X|L],[Y|M]) :&
  tup([X,Y]|pairlists(L,M)).

numbered([],N).
numbered([[X,N]|R],N) :- numbered(R,+(N,1)).

qsort[Cr]([]) :& [].
qsort[Cr]([X|Y]) :-
   partition[Cr](X,Y,Sm,Gr) &
   cat(qsort[Cr](Sm),tup(X|qsort[Cr](Gr))).

partition[Cr](X,[Y|Z],[Y|Sm],Gr) :-
   Cr(Y,X),  partition[Cr](X,Z,Sm,Gr).
partition[Cr](X,[Y|Z],Sm,[Y|Gr]) :-
   Cr(X,Y),  partition[Cr](X,Z,Sm,Gr).
partition[Cr](X,[X|Z],Sm,Gr) :-
            partition[Cr](X,Z,Sm,Gr).
partition[Cr](X,[],[],[]).

before([X1,Y1],[X2,Y2]) :- string<(X1,X2).

cat([],L) :& L.
cat([H|R],L) :& tup(H|cat(R,L)).
```

Copy & paste ready

Query (batch):

t1()

t2()

Result:

relfun
rfi-p> t1()
[2,1,3,3,1,4]
rfi-p>
rfi-p> t2()
[2,1,4,4,1,5,6,3,8,1,5,7,5,1]

Try again with tracer

Query (batch):

trace pairlists numbered qsort[Cr]

Summary

- (Multi-)Directionality of declarative computation
- Encapsulation of declarative operation combinations
- Generate-Test Separation/Integration in FP/LP
- List-Universality as complex declarative datatype
- Invertibility via multiple/single definitions in FP/LP
- Nesting/Conjunction correspondence of properties
- Unification to equate, analyze, refine data in LP (FP)
- Amalgamation and Integration of function & relations,

  e.g. FLP operation: Ground $\xrightarrow{\text{non-ground}}$ Ground

Introduction to Functional and Logic Programming

Intro

Chapter 1

Declarative Programs: Running Example "Bilingual Antonym Agent"

- An *antonym* of a word in some natural language is a word having the opposite meaning (e.g., *hot* – *cold*)
- Suppose we want to program an Antonym Agent for both English and French based on a single catalog of antonyms (for English words) and on translators (between French and English), as found in the Web: As in some Semantic Web approaches, we'll use a single 'canonical' language for internal operations
- The development of this "Bilingual Antonym Agent" will be used as a running example for discussing declarative programs
- It will permit to introduce FP and LP, to show some of their trade-offs, and to motivate FLP

Functional Programs: Basic Notions

- A *function call* applies a function to (actual) arguments and returns a value – no side-effects
  - Each argument may or may not be a *reduced value* (completely evaluated)
  - The application may start before all arguments are reduced (e.g., in *call-by-need / lazy strategy*) or after all arguments are reduced (in *call-by-value / eager strategy*)
  - In 1st*-order (higher-order) functional programming* arguments and returned values cannot (can) be functions
- A *functional clause* associates a function name and (formal) arguments with [a possible conjunction of ground, deterministic relation calls and] a term (e.g., a constant or variable) or a function-call nesting
- A *functional program* is a set of functional clauses

Functional Definition Example: "French Antonym Agent"

- We define a function **fr-antonym**, which – applied to an argument **Mot** (French for 'word') – returns the value of the function nesting
  - **en2fr** applied to **en-antonym** applied to **fr2en**
  - applied to **Mot**
- The functions **fr2en** and **en2fr** perform translations to and from the function **en-antonym**
- This "English Antonym Agent" **en-antonym** acts as a catalog mapping English words to their antonyms (in both directions)
- Variables start with a capital letter; constants and function (and relation) names, with a small letter

Functional Programs: Returned Values from Nested Calls and Pointwise Definitions

Definition by Function Nesting

fr-antonym(Mot) = en2fr(en-antonym(fr2en(Mot)))

en-antonym(black) = white
en-antonym(white) = black
en-antonym(big) = small
en-antonym(small) = big
. . .

Returned Values

fr2en(noir) = black
fr2en(blanc) = white
fr2en(grand) = big
fr2en(petit) = small
. . .

Pointwise Definitions

en2fr(black) = noir
en2fr(white) = blanc
en2fr(big) = grand
en2fr(small) = petit
. . .

Returned Values

Functional Computation Example: “French Antonym Agent”

- The functional agent **fr-antonym** – applied to the argument **noir** – delegates subtasks as follows:
  - **fr-antonym**’s argument **noir** is passed to the agent **fr2en** for French-to-English translation
  - **fr2en**’s returned value **black** is passed to the agent **en-antonym** for English antonym look-up
  - **en-antonym**’s value **white** is passed to the agent **en2fr** for English-to-French translation
- Finally, **en2fr**’s value **blanc** is passed out as the returned value of the agent **fr-antonym**
- In each computation step the function application to be selected next is <u>underlined</u>; results are put in *italics*

Functional Programs: Call-by-Value Computation of Nestings

Call-by-value Computation

fr-antonym(noir)	=	*en2fr(en-antonym(fr2en(noir)))*
	=	en2fr(en-antonym(*black*))
	=	en2fr(*white*)
	=	*blanc*

en-antonym(black)	=	*white*
en-antonym(white)	=	black
en-antonym(big)	=	small
en-antonym(small)	=	big

fr2en(noir)	=	*black*	en2fr(black)	=	noir
fr2en(blanc)	=	white	en2fr(white)	=	*blanc*
fr2en(grand)	=	big	en2fr(big)	=	grand
fr2en(petit)	=	small	en2fr(small)	=	petit

Functional Computation Example: Web Services

- The function composition **en2fr**$\circ$**en-antonym**$\circ$**fr2en** is pre-specified here by the agent **fr-antonym**; a corresponding Web service should find and compose its subfunctions ‘on-the-fly’ in the Web: A library of functions could use UDDI “meta service” (Universal Description, Discovery and Integration)
- The three subfunction calls in a **fr-antonym** Web service could use remote procedure calls of the XML-based SOAP (Simple Object Access Protocol)
- Because of its lack of side-effects, this pure kind of Web-distributed functional programming provides a simplified use case for Web Services

French - English On-line Dictionary Search Results - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Back Search Favorites Media

Address http://dictionaries.travlang.com/FrenchEnglish/dict.cgi?query=pain&max=50

travlang

French-English On-line Dictionary Search Results

This dictionary database is from the freeware multilingual program Ergane.
It contains over 10,000 terms. Also see travlang's English-French Dictionary.

Enter a word or words to search for: pain

Submit Reset Max. Hits: 20 50 100 200

Notes: Searches are case insensitive. You can use a * as a wildcard. Boolean searches are allowed.

Result of search for "pain":

```
douleur
  1. pain

mal
  1. pain
  2. pity
  3. bad, miserable, nasty, poor
  4. badly

pain
  1. bread, loaf [n]

peine
  1. pain
  2. attempt, effort
```

Functional Definition Example: “Bidirectional French-English Translator”

- We define a function **bitranslate**, which – applied to an argument **X** – returns the value of
  - **en2fr** applied to **X** if **X** is an English word
  - **fr2en** applied to **X** if **X** is a French word
- The auxiliary relations **english** and **french** just ‘test-call’ the functions **en2fr** and **fr2en**, respectively
- Since a given argument (such as *pain*) can be both an English and a French word, **bitranslate** will be treated as a *non-deterministic function*, which can enumerate two values (such as *douleur* and *bread*)
- Single-assignments in condition parts here use ‘=’; anonymous variables are written as ‘_’

Functional Programs: Case Analysis (and Pointwise Definitions)

Definition by Case Analysis

$$\text{bitranslate}(X) \;=\; \begin{cases} \text{en2fr}(X) & \textbf{if} \;\; \text{english}(X) \\ \text{fr2en}(X) & \textbf{if} \;\; \text{french}(X) \end{cases}$$

$$\text{english}(X) \quad \textbf{if} \quad \_ = \text{en2fr}(X)$$

$$\text{french}(X) \quad \textbf{if} \quad \_ = \text{fr2en}(X)$$

Auxiliary Definition

fr2en(noir) = black		en2fr(black) = noir	
fr2en(blanc) = white		en2fr(white) = blanc	
fr2en(grand) = big	Pointwise Definitions	en2fr(big) = grand	
fr2en(petit) = small		en2fr(small) = petit	
. . .		. . .	

Functional Definition Example: "Generic Antonym Agent"

- We define a function **antonym**, which – applied to an argument **X** – generically returns the value of the function
  - **en-antonym** applied to **X** if **X** is an English word
  - **fr-antonym** applied to **X** if **X** is a French word
- However, in order to exemplify nested calls within a case analysis, **fr-antonym** will be unfolded into its definition's right-hand side
- Since many words (such as *bread*) do not have an antonym, all **antonym** functions are *partial*, and *fail* for these arguments; for certain words (e.g., *pain*) the internal non-determinism of **antonym** thus disappears before it can spread (e.g., leaving us *joy*)
- An alternative syntax for case analysis introduces a **then** part that returns the function's value

Functional Programs: Case Analysis and Returned Values from Nested Calls

Definitions by Case Analysis

$$\text{antonym}(X) = \begin{cases} \text{en-antonym}(X) \ \ \textbf{if} \ \ \text{english}(X) \\ \text{en2fr}(\text{en-antonym}(\text{fr2en}(X))) \ \ \textbf{if} \ \ \text{french}(X) \end{cases}$$

Function Nesting as Returned Value

Clause Syntax for Case Analysis and Returned Values:

antonym(X) **if** english(X) **then** en-antonym(X)

antonym(X) **if** french(X) **then** en2fr(en-antonym(fr2en(X)))

Function Nesting as Returned Value

Logic Programs: Basic Notions

- A *relation call (‘query’)* applies a relation to (actual) arguments and yields fail or success plus bindings of logic variables – no reassignment side-effects
  - Each argument must from the outset be a *reduced value* (completely evaluated)
  - Roughly speaking, in 1st-*order (higher-order) logic programming* arguments and binding values cannot (can) again be relations; actually, only the Horn-logic subset of 1st-order logic is normally used in LP
- A *relational clause* associates a relation name and (formal) arguments with a [possibly empty] conjunction of (non-)ground, (non-)deterministic relation calls
- A *logic program* is a set of relational clauses

Logic Definition Example: “French Antonym Agent”

- We now define **fr-antonym** as a relation, which is applied to an input argument **Mot** and binds an output argument **Franto** (**Fr**ench **anto**nym) via the following conjunction of relation calls:
  - A relation **fr4en** uses **Mot**, as input, to bind **Word**, as output, to the French-to-English translation result
  - A relation **en-antonym** uses this **Word**, as input, to bind **Enanto**, as output, to the antonym-catalog look-up result
  - The relation **fr4en** now uses **Enanto**, as input, to bind **Franto**, as output (also, of **fr-antonym**), to the English-to -French translation result
- The relational **fr4en** is ‘economically’ accessed in two I/O modes, saving two functions; for the relational **en-antonym**, its symmetry prevents this

Logic Programs: Variable Bindings from Conjunctive Calls (and Base Relations)

Rule:
Definition by
Conjoined Calls

fr-antonym(Mot,Franto) **if** fr4en(Mot,Word) **and**
en-antonym(Word,Enanto) **and**
fr4en(Franto,Enanto)

Variable
Bindings

en-antonym(black,white)
en-antonym(white,black)
en-antonym(big,small)
en-antonym(small,big)
. . .

fr4en(noir,black)
fr4en(blanc,white)
fr4en(grand,big)
fr4en(petit,small)
. . .

Facts: Base Relations

Logic Computation Example: "French Antonym Agent"

- The logic agent **fr-antonym** with input argument **Mot** = **noir** and output argument **Franto** = **Result** (a request variable) delegates subtasks as follows:
  - **fr-antonym**'s binding **Mot** = **noir** is passed to the agent **fr4en** for French-English translation
  - **fr4en**'s binding **Word** = **black** is passed to the agent **en-antonym** for English antonym look-up
  - **en-antonym**'s binding **Enanto** = **white** is passed again to **fr4en** for the inverse task of English-French translation
- Finally, **fr4en**'s binding **Franto** = **Result** = **blanc** is passed out as the result binding of the agent **fr-antonym**
- In each computation step the next relation application(s) is/are <u>underlined</u>; results are *italicized*

Logic Programs: Left-to-Right Computation of Conjunctions

Left-Right Computation

fr-antonym(noir,Result) **if** *fr4en(noir,Word)* ***and*** *en-antonym(Word,Enanto)* ***and*** *fr4en(Result,Enanto)*

if fr4en(noir,*black*) **and** en-antonym(*black*,Enanto) **and** fr4en(Result,Enanto)

if en-antonym(black,*white*) **and** fr4en(Result,*white*)

if fr4en(*blanc*,white) **if** *true*

en-antonym(black,*white*)
en-antonym(white,black)
en-antonym(big,small)
en-antonym(small,big)

fr4en(noir,*black*)
fr4en(*blanc*,white)
fr4en(grand, big)
fr4en(petit,small)

Logic Definition Example: "Bidirectional French-English Translator"

- We now define **bitranslate** as a relation, which is applied to an input argument **X** and binds an output argument **Y** as follows:
  - **fr4en** uses input **X** as 2nd argument and output **Y** as 1st argument if **X** is an English word
  - **fr4en** uses input **X** as 1st argument and output **Y** as 2nd argument if **X** is a French word
- The auxiliary relations **english** and **french** just 'test-call' the relation **fr4en**, in two ways
- Since a given argument (such as *pain*) can be both an English and a French word, **bitranslate** is a *non-deterministic relation*, which enumerates two values (such as *douleur* and *bread*)

Logic Programs: Case Analysis (with Conjunctive Calls and Base Relations)

Rules:
Definition by Case Analysis
with Conjoined Calls

bitranslate(X,Y) **if** english(X) **and** fr4en(Y,X)
bitranslate(X,Y) **if** french(X) **and** fr4en(X,Y)

english(X) **if** fr4en(_,X)
french(X) **if** fr4en(X,_)

Auxiliary
Definition

Facts: Base Relation

fr4en(noir,black)
fr4en(blanc,white)
fr4en(grand,big)
fr4en(petit,small)
. . .

Logic Definition Example: "Generic Antonym Agent"

- We now define **antonym** as a relation, which is applied to an input argument **X** and binds an output argument **Y** generically to the binding of the relation
  - **en-antonym** of input **X**, output **Y** if **X** is an English word
  - **fr-antonym** of input **X**, output **Y** if **X** is a French word
- However, in order to exemplify conjunctive calls within a case analysis, **fr-antonym** will be unfolded into its definition's right-hand side
- Since many words (such as *bread*) do not have an antonym, all **antonym** relations are *partial*, and *fail* for these arguments; for certain words (e.g., *pain*) the internal non-determinism of **antonym** thus disappears before it can spread (e.g., leaving us *joy*)

Logic Programs:
Case Analysis and Conjunctive Calls

Definition by
Case Analysis

antonym(X,Y) **if** english(X) **and** en-antonym(X,Y)

antonym(X,Y) **if** french(X) **and**
fr4en(X,Word) **and**
en-antonym(Word,Enanto) **and**
fr4en(Y,Enanto)

Conjoined Calls

english(X) **if** fr4en(_,X)
french(X) **if** fr4en(X,_)

Auxiliary
Definition

Logic Optimization Example: "Generic Antonym Agent"

- Analyzing this declarative **antonym** program, we can see that the **french** relation call is redundant, since its 'test-call' of **fr4en** is covered by another **fr4en** call:
  - The second **antonym** clause calls **french(X)**, which can be statically unfolded to **fr4en(X,_)**
  - This can be optimized away, since the conjunction already contains the call **fr4en(X,Word)**
- In each optimization step the next abstract relation application(s) is/are <u>underlined</u>; results are *italicized*

Logic Programs:
Static Optimization in Conjunctive Calls

antonym(X,Y) **if** french(X) **and**
fr4en(X,Word) **and**
en-antonym(Word,Enanto) **and**
fr4en(Y,Enanto)

antonym(X,Y) **if** *fr4en(X,_)* **and**
fr4en(X,Word) **and**
en-antonym(Word,Enanto) **and**
fr4en(Y,Enanto)

antonym(X,Y) **if** *fr4en(X,Word)* **and**
en-antonym(Word,Enanto) **and**
fr4en(Y,Enanto)

french(X) **if** *fr4en(X,_)*

Functional-Logic Programs: Elementary Notions

- A *functional-logic program* embodies the following combination of FP and LP:
  1) A relation call can have nested function calls as arguments
  2) The value of a function call can be assigned to a logic variable via single-assignments
  3) A relation definition can use relation calls as in 1) and function calls as in 2)
  4) A function definition can use a conjunction of non-ground, non-deterministic relation calls in its condition (**if**) part and utilize their local bindings in its value-returning (**then**) part (as exemplified below)
- The notions of *function* and *relation* can be further combined for tightly integrated FLP

Functional-Logic Definition Example: “Generic Antonym Agent”

- We again define **antonym** as a *function*, which – applied to an argument **X** – generically returns as its value the (local) output binding **Y** of the *relation*
  - **en-antonym** of input **X**, output **Y** if **X** is an English word
  - **fr-antonym** of input **X**, output **Y** if **X** is a French word
- Again, in order to exemplify nested calls within a case analysis, **fr-antonym** will be unfolded into its definition’s right-hand side
- Advantages of FLP form for the **antonym** operation:
  - From FP: Captures directedness of **antonym** operation: its symmetry prevents two useful I/O modes in LP form
  - From LP: Internally exploits I/O invertibility of **fr4en**: replaces separate functions **fr2en** and **en2fr** of FP form

Functional-Logic Programs: Case Analysis, Conjunctive Calls, and Returned Values

Definition by Case Analysis

antonym(X) **if** english(X) **and** en-antonym(X,Y) **then** Y

Local Variable Bindings

antonym(X) **if** french(X) **and**
fr4en(X,Word) **and**
en-antonym(Word,Enanto) **and**
fr4en(Y,Enanto) **then** Y

Local Variable Bindings

Returned Values

Conjoined Calls

Functional-Logic Programs: Non-Deterministic Operations

- For English and French, or other natural languages with overlapping dictionaries, our earlier function **bitranslate** becomes a *non-deterministic function*, for some arguments *enumerating a set of values*:
  <u>**bitranslate(pain)**</u> **= *{douleur, bread}***
- Such a function – mapping to a power set – could also be regarded as a relation, except that its computation is specified in a directed manner:
  <u>**bitranslate(pain,R)**</u> **= *{R=douleur, R=bread}***
- Hence, non-deterministic functions are often seen as belonging to FLP rather than FP

Functional-Logic Programs: Non-Ground Calls

1) FP uses only variablefree or *ground* function calls:
   fr2en(noir) = *black*, **fr2en(blanc) = *white***, …
2) FLP also permits *non-ground* function calls as in:
   fr2en(A) = *{black/A=noir, white/A=blanc, …}*
3) Moreover, 2) is a *non-deterministic* function call, enumerating returned values and the bindings that the request variable **A** assumes for them
4) LP relation calls equivalent to 1) are *non-ground*:
   fr4en(noir,R) if *R=black*, **fr4en(blanc,R) if *R=white***, …
5) The LP relation call equivalent to 2) again is a *non-ground* and *non-deterministic* call:
   fr4en(A,R)* if *{A=noir/R=black, A=blanc/R=white, …}

Summary

- Notions of Functional and Logic Programming can be treated in a joint manner
- FP's nested calls correspond to LP's conjoint calls; case analysis works similarly in both
- Functional-Logic Programming permits a further integration of both declarative paradigms
- All introduced FP, LP, and FLP constructs run in Relfun (and are marked up in Functional RuleML)
- This introduction has focused 1st-order operations and deliberately used several further restrictions
- The next chapter will overcome the restriction of only using simple data (FP: Datafun; LP: Datalog)

Simple vs. Complex Terms, Ground vs. Non-Ground Terms, and Term Unification

Terms

Chapter 2

Terms

Terms as the Explicit Data Values of FP and LP

- Terms are used as – possibly complex – values passed explicitly as arguments to functions and relations, and returned as values from functions
- Terms can also be stored permanently in relation and function definitions, and temporarily, in (logic) variables, which are renamed on each definition use
- Variables in FP and LP are *single-assignment*, i.e. – once assigned – variables *cannot be re-assigned* (their values can be *refined* via single-assignments to possible other variables within complex values)
- A complex value may have a constructor indicative of its arity and argument types; but FP+LP variables are still often untyped (types can be added: RuleML)

Taxonomy of Terms: Two Trees with Overlapping Distinctions

- Term
  - Simple Term
    - Constant
      - Symbol
      - Number
    - Variable
      - Named
        - -Upper-cased
        - -Underscored
      - Anonymous
  - Complex Term
    - Structure (application of constructor to terms)
    - List (short form for nested binary **cns** structure)

- Term
  - Ground (variablefree)
  - Non-ground (variableful)

•FP permits only ground terms as arguments and returned values
•LP also permits non-ground terms as arguments
•FLP even permits non-ground terms as arguments and as returned values

Simple Terms: Constants and Variables

- An *(individual) constant* is a name for a given entity. It starts with a lower-case letter, a digit, or with "**-**"

Examples:

Symbols:	**u**	**i**	**john**	**mary**	**peter**	**susan**
Numbers:	**9**	**42**	**-1**	**-89**	**-3.14**	**-276.0131**

- A *(logic) variable* is a place-holder for some term, where all occurrences of the same named variable must stand for the **same** term.
  A variable starts with an upper-case letter or with an "_" (a single "_" acts as an *anonymous variable*)

Examples:

Upper-cased:	**X**	**Y**	**Word**	**Mot**	**Anon**
Underscored:	**_9**	**_42**	**_rs2**	**_mot**	**_**

Terms

Complex Terms: Structures

- A *constructor* is a name for a fixed structure former much like an XML start tag (in LP often called a *functor* or – different from FP – a *function symbol*)

Examples: **c** **rs** **duo** **addr**

- A *structure* is a '**[…]**'-application of a constructor to a sequence of zero or more '**,**'-separated argument terms, possibly including other structured terms (then called a *nested structure*; otherwise, a *flat structure*)

Examples:	Flat structures:	Nested structures:
Ground:	**c[] rs[1] duo[u,i]**	**addr[john,loc[ny,ny]]**
Non-ground:	**rs[_] duo[X,Y]**	**addr[john,loc[X,X]]**

Term Unification: Algorithmic Principles

- The *unification algorithm* compares two terms, treated symmetrically, for structural compatibility:
  - If both are ground terms, it *succeeds* if they are equal
  - If at least one is a non-ground term, it *succeeds* if they can be made equal by binding variables consistently across both terms
  - Otherwise it *fails*
- Unification can start in a pre-existing *environment* (or *substitution*) of variable bindings, to which it must be consistent
- Unification, if successful, can create new variable bindings for extending the environment
- Unification creates the *least number of variable bindings* necessary to succeed (the set of these bindings is called the *most general unifier* or *mgu*)

Term Unification: Variable Dereferencing and Case Analysis

- Unification, whenever one of its terms is a variable, first dereferences that variable in the current binding environment by taking its ultimate value at the end of a possibly long chain of variable-variable bindings (the ultimate value can still be a – free – variable)
- Unification then performs a case analysis as shown in the following slides (in Relfun, unification can be explicitly performed via ".=")

Example:

addr[john,loc[ny,ny]]
addr[john,loc[X,X]]

addr[john,loc[ny,ny]] .= addr[john,loc[X,X]]

In an empty environment succeeds, creating the binding **X=ny**
In an environment with **X=ny** succeeds, creating no new binding
In an environment with **X=Y** succeeds, creating binding **Y=ny**
In an environment with **X=Y**, **Y=Z**, and **Z=sf** fails

Term Unification: Two Constants

- If both terms are constants, unification *succeeds* if they are equal; otherwise it *fails*

Examples:

Term 1:	**u**	**i**	**john**	**peter**	**9**	**-276.0131**
Term 2:	**u**	**u**	**mary**	**peter**	**42**	**-276.0131**
Result:	*succ*	*fail*	*fail*	*succ*	*fail*	*succ*

In many systems, constants can also be "…" strings, where, e.g., the terms **"peter miller"** and **"peter miller"** give *succ*, while the terms **"peter miller"** and **"peter meyer"** give *fail* (also, **"u"** and **u** give *fail*; **"X"** and **X** will give *succ* with **X = "X"**)

Terms

Term Unification: Constant and Structure

- If one term is a constant and the other a structure, unification *fails*

Examples:

Term 1:	**u**	**c[]**	**rs[1]**	**duo[X,Y]**	**duo**
Term 2:	**c[]**	**c**	**mary**	**peter**	**duo[X,Y]**
Result:	*fail*	*fail*	*fail*	*fail*	*fail*

Here, even if a constant such as **c** has the same name as the constructor of a nullary (argumentless) structure such as **c[]**, we define unification to *fail* (some systems actually forbid to use the same name for constants and constructors; but others would <u>identify</u> constants with nullary structures and succeed)

Terms

Term Unification: Variable and Constant

Terms

- If one term is a variable and the other a constant, unification *succeeds*, binding the variable to this constant value (except for an anonymous variable)

Examples:

Term 1:	**X**	**i**	**john**	**_**	**_rs2**	**-276.0131**
Term 2:	**u**	**Y**	**_9**	**peter**	**42**	**_**
Result:	*succ*	*succ*	*succ*	*succ*	*succ*	*succ*
Bindings:	**X=u**	**Y=i**	**_9=john**		**_rs2=42**	

Term Unification: Variable and Structure

- If one term is a variable and the other a *structure not containing the variable* (so-called *occurs check*), unification *succeeds*, binding the variable to this structure (except for an anonymous variable)

Examples:

Term 1:	**X**	**c[]**	**rs[1]**	**duo[X,Y]**	**duo[X,Y]**
Term 2:	**c[]**	**_**	**Y**	**_9**	**X**
Result:	*succ*	*succ*	*succ*	*succ*	*fail*
Bindings:	**X=c[]**		**Y=rs[1]**	**_9=duo[X,Y]**	

The occurs check is omitted from many Prolog implementations for efficiency reasons, and is currently also absent from Relfun. It is implemented in the theorem-prover-like LP engine *j*DREW

Term Unification: Variable and Variable

- If the terms are two variables, unification *succeeds*, binding the first variable to the second variable iff these are different variables (a *trivial occurs check*)

Examples:

Term 1:	**X**	**_rs2**	**_9**	**X**	**_**	**X**
Term 2:	**Y**	**_9**	**Mot**	**_**	**_**	**X**
Result:	*succ*	*succ*	*succ*	*succ*	*succ*	*succ*
Bindings:	**X=Y**	**_rs2=_9**	**_9=Mot**	**X=_**		

Leaving a variable unbound after it was unified with itself has been a useful part of defining unification in practice, hence is implemented in Relfun.

Anonymous variables are really treated via name generation, but their rough treatment is indicated by two examples above

Terms

Term Unification: Two Structures (I)

Terms

- If both terms are structures, unification *succeeds* if they have the same constructor, the same number of arguments, and unification is successful for each pair of corresponding arguments, where bindings must be consistent across the entire structures; otherwise it *fails*

Examples: Flat structures:

Term 1:	**c[]**	**rs[1]**	**rs[1]**	**rs[1]**	**trio[1,X,Y]**	**trio[1,X,X]**
Term 2:	**c[]**	**rs[2]**	**jk[1]**	**rs[Z]**	**trio[1,u,i]**	**trio[1,u,i]**
Result:	*succ*	*fail*	*fail*	*succ*	*succ*	*fail*
Bindings:				**Z=1**	**X=u, Y=i**	

Term Unification: Two Structures (II)

Examples:	Nested structures:	
Term 1:	**addr[john,loc[ny,ny]]**	**addr[X,loc[ny,ny]]**
Term 2:	**addr[john,loc[X,X]]**	**addr[john,loc[X,X]]**
Result:	*succ*	*fail*
Bindings:	**X=ny**	

Complex Terms: Lists as **cns** Structures

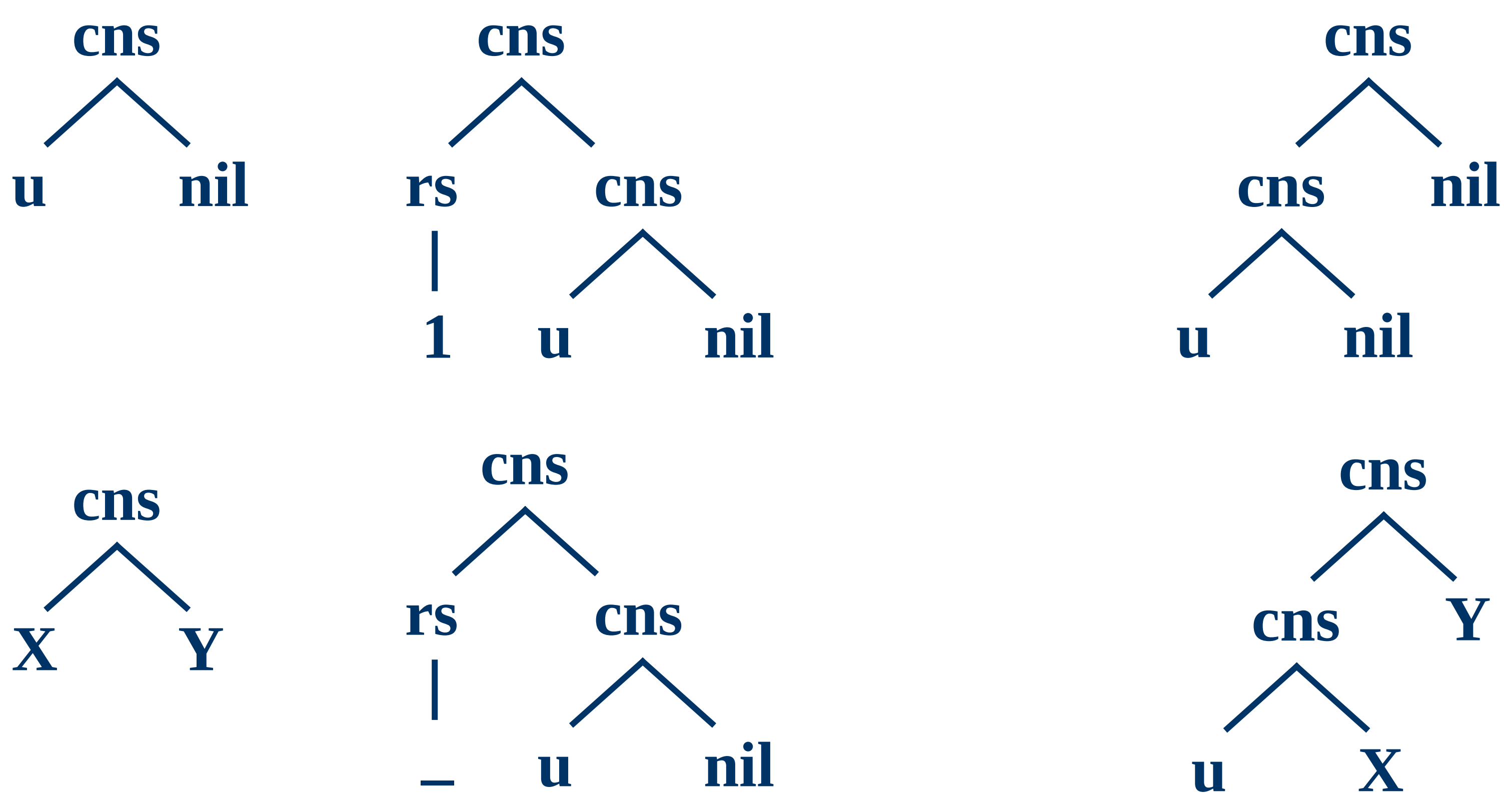
- The *constructor* **cns** forms binary **cns** structures (much like **cons** cells or 'dotted pairs' in Lisp)
- The *constant* **nil** terminates second-argument nestings of **cns** (much like in Lisp)
- A *list* is **nil** (*empty list*) or is a '**[…]**'-application of **cns** to a sequence of two '**,**'-separated element terms (*non-empty list*), the second of which must be a list or a variable while the first one may be any term (if it is a list, the entire list is called a *nested list*)

Examples:	Flat lists (**cns** right-recursive):		Nested lists:
Ground:	**cns[u,nil]**	**cns[rs[1],cns[u,nil]]**	**cns[cns[u,nil],nil]**
Non-ground:	**cns[X,Y]**	**cns[rs[_],cns[u,nil]]**	**cns[cns[u,X],Y]**

Terms

Complex Terms: Lists as cns Trees

Terms



Flat lists (**cns** right-recursive):

cns[u,nil] **cns[rs[1],cns[u,nil]]**

cns[X,Y] **cns[rs[_],cns[u,nil]]**

Nested lists:

cns[cns[u,nil],nil]

cns[cns[u,X],Y]

Complex Terms: N-ary List Notation

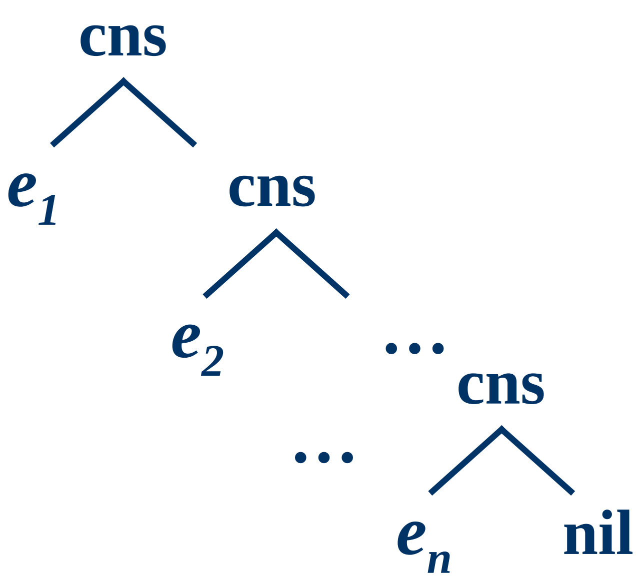
The n-ary short notation of lists, for n≥0, can be obtained from lists as **cns** structures as follows:

- The *empty list* **nil** is rewritten as **[ ]**, for n=0
- A *non-empty list* **cns[e_1 , cns[e_2 , …cns[e_n , t]…]]**, for n≥1, is rewritten as **[e_1' , e_2' , …, e_n']**, if t is **nil**, and is rewritten as **[e_1' , e_2' , …, e_n' | t]**, if t is a variable, where the primes indicate recursive rewritings

Examples:	Flat **cns** (original) lists:		Nested **cns** lists:
Ground:	**cns[u,nil]**	**cns[rs[1],cns[u,nil]]**	**cns[cns[u,nil],nil]**
Non-ground:	**cns[X,Y]**	**cns[rs[_],cns[u,nil]]**	**cns[cns[u,X],Y]**

Examples:	Flat n-ary (rewritten) lists:		Nested n-ary lists:
Ground:	**[u]**	**[rs[1],u]**	**[[u]]**
Non-ground:	**[X|Y]**	**[rs[_],u]**	**[[u|X]|Y]**

Complex Terms: N-ary Tree Notation (I)

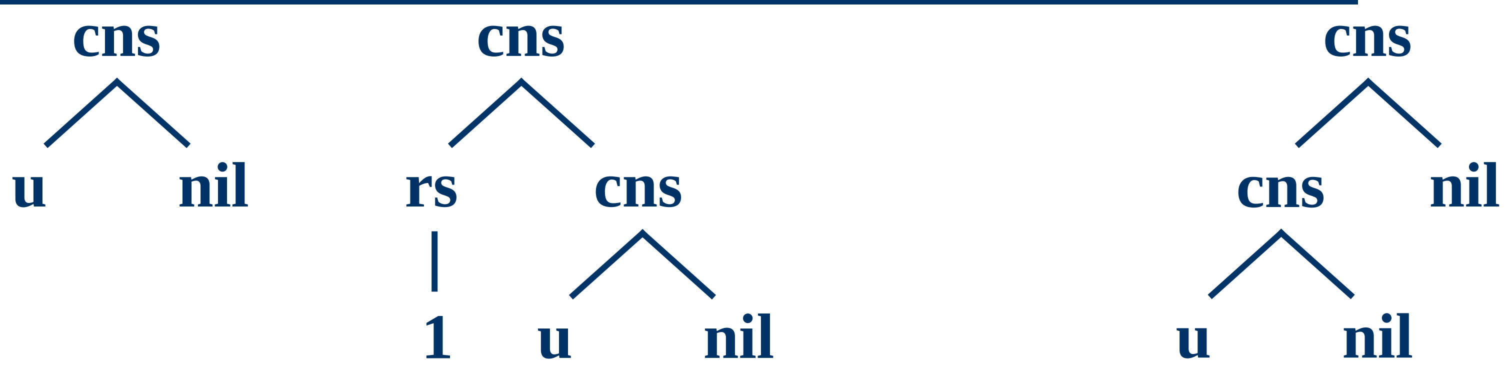


$\text{cns}[e_1, \text{cns}[e_2, \ldots \text{cns}[e_n, \text{nil}]\ldots]]$

$[e_1', e_2', \ldots, e_n']$

Terms

Complex Terms: N-ary Tree Notation (II)



Flat lists (**cns** right-recursive):

cns[u,nil] **cns[rs[1],cns[u,nil]]**

Nested lists:

cns[cns[u,nil],nil]

Flat n-ary (rewritten) lists:

[u] **[rs[1],u]**

Nested n-ary lists:

[[u]]

Terms

Terms

List Unification

Lists as **cns** structures do not change the earlier unification algorithm: The n-ary list notation permits a variable after a "|" to unify with a *rest segment* of another list, but in the **cns** form such a segment is just a **cns** structure nested into the second argument

Examples:	Flat **cns** (original) lists:		Nested **cns** lists:
Term 1:	**cns[u,nil]**	**cns[rs[1],cns[u,nil]]**	**cns[cns[u,nil],nil]**
Term 2:	**cns[X,Y]**	**cns[rs[_],cns[u,nil]]**	**cns[cns[u,Y],Z]**
Examples:	Flat n-ary (rewritten) lists:		Nested n-ary lists:
Term 1:	**[u]**	**[rs[1],u]**	**[[u]]**
Term 2:	**[X|Y]**	**[rs[_],u]**	**[[u|Y]|Z]**
Result:	*succ*	*succ*	*succ*
Bindings:	**X=u**, **Y=nil**		**Y=nil**, **Z=nil**

Implementing Anonymous Variables as Freshly Generated Named Variables

Anonymous variables cannot be just implemented by generating no bindings for their unification partners, but must be treated via name generation. Otherwise the second example below would erroneously succeed with the binding **X=suc[_]**:

Example:	Named variable in structure (in list):		
Term 1:	**[suc[N],**	**suc[0],**	**suc[1]]**
Term 2:	**[X,**	**X,**	**X]**
Example:	Anonymous variable in structure (in list):		
Term 1:	**[suc[_],**	**suc[0],**	**suc[1]]**
Term 2:	**[X,**	**X,**	**X]**
Result:	*fail*		

In Relfun, all occurrences of "_" are thus implemented by generating fresh versions of the variable name "**Anon**"

Summary

- Terms are the explicit data values of FP and LP
- A taxonomy of simple vs. complex terms, and ground vs. non-ground terms, was introduced
- Principles and a full (term-)case analysis of unification were illustrated via examples
- Implemented versions of unification algorithms, e.g. in functional programming itself, are usually quite compact; can also be used for call invocation
- The n-ary list short notation was introduced as a rewriting of lists as **cns** structures
- List unification with one segment variable per (sub)list was discussed as a notational variant

Functional and Logic Definition Clauses

Clauses

Clauses

Chapter 3

Clauses as the Smallest Functional and Logic Definition Units

- An *operation (name*) is a function or relation (name)
- A *clause* associates a *head* of an operation name and argument terms with an optional *body* of a (non-)ground, (non-)deterministic call conjunction and an optional *foot* consisting of a term or a nesting
  - The head's call pattern acts as a first, deterministic filter on operation calls
  - A body conjunction acts as the main, (non-)deterministic condition on operation calls and can accumulate consistent local variable bindings
  - A foot denotes or computes an explicit returned value
- A *program* is a set of clauses; a *procedure* is a subset of clauses with the same operation name

Taxonomy and Syntax of Clauses

- Clause
  - Logic Clause
    - Fact: *head*.
      - Ground Fact
    - Rule: *head* **:-** *body*.
  - Functional Clause
    - Unconditional Equation: *head* **:&** *foot*.
      - Molecule: foot is a term ('solved' equation)
        - Point (Ground Molecule, pointwise definition)
    - Conditional Equation: *head* **:-** *body* **&** *foot*.
  - Functional-Logic Clause

Empty body:

true body

Empty body:

true body

:- . Syntax as in Prolog

& Syntax from Relfun (for value returning)

Resolution: The Computation Method of Functional and Logic Programming

- In any pre-existing variable binding environment, the *resolution* of an operation call, from a body conjunction or a foot, with a candidate clause
  1) uses unification between the **call** and the **head** of the clause in this environment to determine whether, and with which new bindings, the clause can be invoked by the call (unification treats call and head as structures)
  2) on unification success, inserts the possible **body** and/or **foot** of the clause in place of the call and yields the extended binding environment
- This process continues until either
  – Success: the body conjunction is empty (true) and the foot is a reduced value
  – Failure: no (more) clauses can be invoked

Logic Clauses: A Fact in English, Pseudo-Code, and Prolog/Relfun

(Controlled) English Definition of a Logic Business Fact:

“Peter Miller's spending has been min 5000 euro in the previous year”

Pseudo-Code Relation Definition with a Ground Fact:

spending(Peter Miller,min 5000 euro,previous year)

Prolog/Relfun Relation Definition with a Ground Fact:

spending("Peter Miller","min 5000 euro","previous year").

Logic Clauses: A Ground Call Resolved via Unification

Relfun Relation Ground Call:

After finding the above fact, the call (in Prolog ended by a period)
spending("Peter Miller","min 5000 euro","previous year")
returns **true**

Unification Computes Whether (and How) the Call Can Use the Fact:

Form 1: spending("Peter Miller","min 5000 euro","previous year")
Form 2: spending("Peter Miller","min 5000 euro","previous year").

Internally, call and head are treated like structures:

Term 1: spending["Peter Miller","min 5000 euro","previous year"]
Term 2: spending["Peter Miller","min 5000 euro","previous year"]

Result: ***SUCC*** **(Whether: yes)**
Bindings: **(How: Directly equal)**

Logic Clauses: Non-Ground Calls Resolved via Unification (I)

Relfun Relation Non-Ground Call:

After again finding the above fact, the call
spending("Peter Miller",Amount,"previous year")
returns **true** with the binding **Amount="min 5000 euro"**

Unification Computes Whether (and How) the Call Can Use the Fact:

Form 1: spending("Peter Miller",Amount,"previous year")
Form 2: spending("Peter Miller","min 5000 euro","previous year").

Result: ***SUCC*** **(Whether: yes)**
Bindings: Amount="min 5000 euro" **(How: Output Amount)**

Logic Clauses: Non-Ground Calls Resolved via Unification (II)

Relfun Relation Non-Ground Call:

After again finding the above fact, the call
spending("Peter Miller","min 5000 euro",Time)
returns **true** with the binding **Time="previous year"**

Unification Computes Whether (and How) the Call Can Use the Fact:

Form 1: spending("Peter Miller","min 5000 euro",Time)
Form 2: spending("Peter Miller","min 5000 euro","previous year").

Result: *SUCC* **(Whether: yes)**
Bindings: Time="previous year" **(How: Output Time)**

Logic Clauses: Non-Ground Calls Resolved via Unification (III)

Relfun Relation Non-Ground Call:

After again finding the above fact, the call
spending("Peter Miller",Amount,Time)
returns **true** with the bindings **Amount="min 5000 euro",**
Time="previous year"

Unification Computes Whether (and How) the Call Can Use the Fact:

Form 1: spending("Peter Miller",Amount,Time)
Form 2: spending("Peter Miller","min 5000 euro","previous year").

Result: *SUCC* **(Whether: yes)**
Bindings: Amount="min 5000 euro" **(How: Output Amount)**
Time="previous year" **(How: Output Time)**

Logic Clauses: Non-Ground Calls Resolved via Unification (IV)

Relfun Relation Non-Ground Call:

After again finding (only) the above fact, the call
spending("Peter Miller",AT,AT)
yields **unknown** (Prolog's closed-world assumption yields **false**)

Unification Computes Whether (and How) the Call Can Use the Fact:

Form 1: spending("Peter Miller",AT,AT)
Form 2: spending("Peter Miller","min 5000 euro","previous year").

Result: *fail* **(Whether: no)**
Bindings:

Clauses

Logic Clauses: Non-Ground Calls Resolved via Unification (V)

Relfun Relation Non-Ground Call:

After again finding the above fact, the call
spending("Peter Miller",_,_)
returns **true**

Unification Computes Whether (and How) the Call Can Use the Fact:

Form 1: spending("Peter Miller",_,_)
Form 2: spending("Peter Miller","min 5000 euro","previous year").

Result: ***SUCC*** **(Whether: yes)**
Bindings:

Functional Clauses: A Point in English, Pseudo-Code, and Relfun

English Definition of a Functional Business 'Point' (Pointwise Definition):

"Peter Miller's spending in the previous year has been min 5000 euro"

Pseudo-Code Function Definition with an Unconditional Equation:

spending(Peter Miller,previous year) = min 5000 euro

Relfun Function Definition with an Unconditional Equation (left-hand-side *head*: spending("…","…"), **right-hand-side *foot*:** "…"**):**

spending("Peter Miller","previous year") **:&** "min 5000 euro".

Clauses

Functional Clauses: A Ground Call Resolved via Unification

Relfun Function Ground Call –
Corresponds to Relation Non-Ground Call (I):

After finding the above point, the call
spending("Peter Miller","previous year")
returns **"min 5000 euro"** (Amount is returned, rather than bound)

Unification Computes Whether (and How) the Call Can Use the Point:

Form 1: spending("Peter Miller","previous year")
Form 2: spending("Peter Miller","previous year").

Result: *SUCC* **(Whether: yes)**
Bindings: **(How: Directly equal)**

Further Resolution Computes the Returned Value:

Value: "min 5000 euro"
Bindings:

Functional-Logic Clauses: A Non-Ground Call Resolved via Unification

Relfun Function Non-Ground Call –
Corresponds to Relation Non-Ground Call (III):

After again finding the above function point, the FLP call
spending("Peter Miller",Time)
returns **"min 5000 euro"** with the binding **Time="previous year"**

Unification Computes Whether (and How) the Call Can Use the Point:

Form 1: spending("Peter Miller",Time)
Form 2: spending("Peter Miller","previous year").

Result: ***succ*** **(Whether: yes)**
Bindings: Time="previous year" **(How: Output Time)**

Further Resolution Computes the Returned Value:

Value: "min 5000 euro"
Bindings: Time="previous year"

Logic Clauses: 1st Rule in English, Pseudo-Code, and Prolog/Relfun

Clauses

English Definition of a Logic Business Rule:

"A customer is premium if
  their spending has been min 5000 euro in the previous year"

Pseudo-Code Relation Definition with a Single-Condition Datalog Rule:

premium(Customer) **if**
  spending(Customer,min 5000 euro,previous year)

Prolog/Relfun Relation Definition with a Single-Condition Datalog Rule:

```
premium(Customer) :-
    spending(Customer,"min 5000 euro","previous year").
```

Logic Clauses: A Ground Call Resolved via Unification and a Subcall

Relfun Relation Ground Call:

After finding the above rule, the call
premium("Peter Miller")
returns **true**

Unification Computes Whether (and How) the Call Can Use the Rule:

Form 1: premium("Peter Miller")
Form 2: premium(Customer) :-

Result: ***SUCC*** **(Whether: yes)**
Bindings: Customer="Peter Miller" **(How: Input Customer)**

Further Resolution Invokes Another Ground Call:

With the above Customer binding, the subcall
spending("Peter Miller","min 5000 euro","previous year")
returns **true** as shown earlier

Classes

Functional Clauses: Mimic 1st Logic Rule in English, Pseudo-Code, and Relfun

English Definition of a (Characteristic-)Functional Business Rule:

"That a customer is premium,
given min 5000 euro equaled their spending in the previous year,
is true"

Pseudo-Code Function Definition with true-Valued Conditional Equation:

premium(Customer) **if**
min 5000 euro = spending(Customer,previous year)
then true

Relfun Function Definition with a true-Valued Conditional Equation:

premium(Customer) **:-**
"min 5000 euro" .= spending(Customer,"previous year")
& true.

Functional Clauses: A Ground Call Resolved via Unification and a “.=” Subcall

Relfun (Characteristic-)Function Ground Call:

After finding the above rule, the call
premium("Peter Miller")
returns **true**

Unification Computes Whether (and How) the Call Can Use the Rule:

Form 1: premium("Peter Miller")
Form 2: premium(Customer) :-

Result: ***SUCC*** **(Whether: yes)**
Bindings: Customer="Peter Miller" **(How: Input Customer)**

Further Resolution Unifies String with Value of Another Ground Call:

With the above Customer binding, the right-hand-side subcall of
"min 5000 euro" .= spending("Peter Miller","previous year")
returns **"min 5000 euro"** as shown earlier, unifying with the lhs

Functional Clauses: Extend 1st Logic Rule in English, Pseudo-Code, and Relfun

English Definition of a (Constant-)Functional Business Rule:

"When a customer is premium,
given min 5000 euro equaled their spending in the previous year,
they get a bonus"

Pseudo-Code Function Definition with bonus-Valued Conditional Equation:

premium(Customer) **if**
min 5000 euro = spending(Customer,previous year)
then bonus

Relfun Function Definition with a bonus-Valued Conditional Equation:

```
premium(Customer)  :-
    "min 5000 euro" .= spending(Customer,"previous year")
                      & bonus.
```

Logic Clauses: 2nd Rule in English, Pseudo-Code, and Prolog/Relfun

English Definition of a Logic Business Rule:

"The discount for a customer buying a product is 5.0 percent if the customer is premium and the product is regular"

Pseudo-Code Relation Definition with a Two-Condition Datalog Rule:

discount(Customer,Product,5.0 percent) **if**
premium(Customer) **and** regular(Product)

Prolog/Relfun Relation Definition with a Two-Condition Datalog Rule:

discount(Customer,Product,"5.0 percent") **:-**
premium(Customer) **,** regular(Product).

Logic Clauses: A Non-Ground Call Resolved via Unification and Subcalls (I)

Relfun Relation Non-Ground Call:

After finding the above rule, and with another fact, the call
discount("Peter Miller","Honda",Rebate)
returns **true** with the binding **Rebate="5.0 percent"**

Unification Computes Whether (and How) the Call Can Use the Rule:

Form 1: discount("Peter Miller","Honda",Rebate)
Form 2: discount(Customer,Product,"5.0 percent") :-

Result: ***SUCC*** **(Whether: yes)**
Bindings: Customer="Peter Miller" **(How: Input Customer)**
Product ="Honda" **(How: Input Product)**
Rebate="5.0 percent" **(How: Output Rebate)**

Logic Clauses: A Non-Ground Call Resolved via Unification and Subcalls (II)

Further Resolution Invokes a Conjunction of two Ground Calls:

With the above Customer and Product bindings, the subcalls
premium("Peter Miller") , regular("Honda")
both return **true**:

premium("Peter Miller") as shown earlier

regular("Honda") with another fact, **regular("Honda").**

Clauses

Functional Clauses: 2nd Rule in English, Pseudo-Code, and Relfun

English Definition of a Functional Business Rule:

"The discount for a customer buying a product,
the customer being premium and the product being regular,
is 5.0 percent"

Pseudo-Code Function Definition with a Conditional Equation:

discount(Customer,Product) **if**
premium(Customer) **and** regular(Product)
then "5.0 percent"

Relfun Function Definition with a Conditional Equation:

```
discount(Customer,Product)  :-
      premium(Customer) , regular(Product)
                          & "5.0 percent".
```

Functional Clauses: A Ground Call Resolved via Unification and Subcalls (I)

Relfun Function Ground Call:

After finding the above rule, and with another point, the call
discount("Peter Miller","Honda")
returns **"5.0 percent"** (Rebate is returned, rather than bound)

Unification Computes Whether (and How) the Call Can Use the Rule:

Form 1: discount("Peter Miller","Honda")
Form 2: discount(Customer,Product) :-

Result:	***SUCC*** **(Whether: yes)**
Bindings:	Customer="Peter Miller" **(How: Input Customer)**
	Product ="Honda" **(How: Input Product)**

Functional Clauses: A Ground Call Resolved via Unification and Subcalls (II)

Further Resolution Invokes a Conjunction of two Ground Calls:

With the above Customer and Product bindings, the subcalls
premium("Peter Miller") , regular("Honda")
both return **true**:

premium("Peter Miller") as shown earlier

regular("Honda") with another point,
regular("Honda") :& true.

Finally Resolution Computes the Returned Value:

Value: "5.0 percent"
Bindings:

Our Complete discount Program: Logic Prolog/Relfun Version

```
discount(Customer,Product,"5.0 percent")  :-
      premium(Customer) , regular(Product).

premium(Customer)  :-
      spending(Customer,"min 5000 euro","previous year").

spending("Peter Miller","min 5000 euro","previous year").

regular("Honda").
```

Relational **invertibility** also permits Product queries

discount("Peter Miller","Honda",Rebate) returns **true**
with binding **Rebate="5.0 percent"**

Our Complete discount Program: Functional (Equational) Relfun Version

```
discount(Customer,Product)  :-
      premium(Customer) , regular(Product)
                                          & "5.0 percent".

premium(Customer)  :-
      "min 5000 euro" .= spending(Customer,"previous year")
                              & true.

spending("Peter Miller","previous year") :& "min 5000 euro".

regular("Honda") :& true.
```

Functional **directedness** prevents inverse Product queries

discount("Peter Miller","Honda") returns **"5.0 percent"**

Our Complete discount Program: Functional-Logic Relfun Version

```
discount(Customer,Product)  :-
      premium(Customer) , regular(Product)
                                        & "5.0 percent".

premium(Customer)  :-
       "min 5000 euro" .= spending(Customer,"previous year").

spending("Peter Miller","previous year") :& "min 5000 euro".

regular("Honda").
```

FLP combines **directedness** with **invertibility** to also permit Product queries

discount("Peter Miller","Honda") returns **"5.0 percent"**

Summary

- Clauses are the smallest FP and LP definition units. They consist of a head (in FP+LP), an optional body (in FP+LP), and a possible foot (in FP)
- The taxonomy and syntax of logic, functional, and functional-logic clauses was introduced
- Based on unification, resolution of an operation call with a candidate clause was introduced as the main FP and LP computation method
- Versions of the RuleML discount program were developed in different styles, with logic clauses, functional clauses, and functional-logic clauses
- Relfun users choose their individual clause styles
- The next chapter will proceed from the simple Datafun/Datalog clauses here to Horn clauses

Recursion in the Definition of Clauses

Recursion

Chapter 4

FP: A Tail-Recursive Natural-Number Addition Function (I)

For M>0, this is a recursion (here: loop) *invariant* of add:

add(M,N) = add(M-1,N+1)

Notation:

add(M,N) :& add(1-(M),1+(N)).

Recursion

FP: A Tail-Recursive Natural-Number Addition Function (II)

Un/Conditional Equations with Recursive Call as a Foot (Tail-Recursion):

add(0,N) :& N. ← Base Case: Termination

add(M,N) :- >(M,0) & **add(**1-(M), 1+(N)**).** ← General Case: Recursion

Based on Built-ins: > Greater 1- Predecessor 1+ Successor

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

add(3,4)
add(2,5)
add(1,6)
add(0,7)

← General Case: Recursion

7 ← Base Case: Termination

LP: A Tail-Recursive Natural-Number Addition Relation (I)

Recursion

For M>0, this is a recursion (here: loop) *invariant* of add:

M + N = R **if** M-1 + N+1 = R
add(M,N,R) **if** add(M-1,N+1,R)

Notation:

```
add(M,N,R) :-  P .= 1-(M),  S .= 1+(N),
               add(P,S,R).
```

LP: A Tail-Recursive Natural-Number Addition Relation (II)

Datalog Rule with Recursive Call as a Last Premise (Tail-Recursion):

add(0,N,N). Termination

add(M,N,R) :- >(M,0), P .= 1-(M), S .= 1+(N), **add(**P,S,R**)**. Recursion

Based on Built-ins: > Greater 1- Predecessor 1+ Successor

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

add(3,4,A)
add(2,5,R1)
add(1,6,R2)

Recursion

add(0,7,7)
A=R1=R2=7

Termination

Since built-ins must be called with ground arguments (here: fixed M and N), inverse calls like add(3,W,7), add(V,4,7), or add(V,W,7) are not permitted!

FLP: A Tail-Recursive Natural-Number Addition Relation

Datalog-like Rule with Recursive Call as a Last Premise (Tail-Recursion):

add(0,N,N).

Termination: As Before

add(M,N,R) :- >(M,0), **add(**1-(M),1+(N),R**)**.

Recursion: Over Nestings

Based on Built-ins: > Greater 1- Predecessor 1+ Successor

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

add(3,4,A)
add(2,5,R1)
add(1,6,R2)

Recursion

add(0,7,7)
A=R1=R2=7

Termination

Again, this add cannot be inverted for subtraction etc.!

FP: A Tail-Recursive Successor-Arithmetic Addition Function (I)

For M≥0, this is a recursion (here: loop) ***invariant*** **of add:**

add(M+1,N) = add(M,N+1)

Notation:

add(suc[M],N) :& add(M,suc[N]).

FP: A Tail-Recursive Successor-Arithmetic Addition Function (II)

Unconditional Equations with Recursive Call as a Foot (Tail-Recursion):

add(0,N) :& N.

Base Case: Termination

add(suc[M],N) :& **add(**M,suc[N]**)**.

General Case: Recursion

No Built-ins Required; 1+ replaced by suc (successor) structures

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

add(suc[suc[suc[0]]],suc[suc[suc[suc[0]]]])
add(suc[suc[0]],suc[suc[suc[suc[suc[0]]]]])
add(suc[0],suc[suc[suc[suc[suc[suc[0]]]]]])
add(0,suc[suc[suc[suc[suc[suc[suc[0]]]]]]])

General Case: Recursion

suc[suc[suc[suc[suc[suc[suc[0]]]]]]]

Base Case: Termination

LP: A Tail-Recursive Successor-Arithmetic Addition Relation (I)

For M≥0, this is a recursion (here: loop) *invariant* of add:

M+1 + N = R	**if**	M + N+1 = R
add(M+1,N,R)	**if**	add(M,N+1,R)

Notation:

```
add(suc[M],N,R)  :-   add(M,suc[N],R).
```

LP: A Tail-Recursive Successor-Arithmetic Addition Relation (II)

Horn Logic Rule with Recursive Call as a Single Premise (Tail-Recursion):

add(0,N,N).

Base Case: Termination

add(suc[M],N,R) :- **add(**M,suc[N],R**)**.

General Case: Recursion

No Built-ins Required; 1+ replaced by suc (successor) structures

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

add(suc[suc[suc[0]]],suc[suc[suc[suc[0]]]],A)
add(suc[suc[0]],suc[suc[suc[suc[suc[0]]]]],R1)
add(suc[0],suc[suc[suc[suc[suc[suc[0]]]]]],R2)

General: Recursion

add(0,suc[suc[suc[suc[suc[suc[suc[0]]]]]]],
suc[suc[suc[suc[suc[suc[suc[0]]]]]]])

Base: Termination

A=R1=R2=suc[suc[suc[suc[suc[suc[suc[0]]]]]]]

LP: A Tail-Recursive Successor-Arithmetic Addition Relation (III)

Additions like $3 + 4 = A$ can be inverted for subtraction:

$3 + W = 7$ *or* $W = 7 - 3$

add(suc[suc[suc[0]]],W,suc[suc[suc[suc[suc[suc[suc[0]]]]]]])

W=suc[suc[suc[suc[0]]]]

$V + 4 = 7$ *or* $V = 7 - 4$

add(V,suc[suc[suc[suc[0]]]],suc[suc[suc[suc[suc[suc[suc[0]]]]]]])

V=suc[suc[suc[0]]]

LP: A Tail-Recursive Successor-Arithmetic Addition Relation (IV)

Can also be inverted for non-deterministic partitioning:

V + W = 7

add(V,W,suc[suc[suc[suc[suc[suc[suc[0]]]]]]])

V=0, W=suc[suc[suc[suc[suc[suc[suc[0]]]]]]]
V=suc[0], W=suc[suc[suc[suc[suc[suc[0]]]]]]
. . .
V=suc[suc[suc[0]]], W=suc[suc[suc[suc[0]]]]
. . .
V=suc[suc[suc[suc[suc[suc[suc[0]]]]]]], W=0

LP: An Equivalent Successor-Arithmetic Addition Relation (I)

For M≥0, this was the recursion (here: loop) *invariant* of add:

$$\mathrm{M}{+}1 + \mathrm{N} = \mathrm{R} \quad \textbf{if} \quad \mathrm{M} + \mathrm{N}{+}1 = \mathrm{R}$$

$$\mathrm{add}(\mathrm{M}{+}1,\mathrm{N},\mathrm{R}) \quad \textbf{if} \quad \mathrm{add}(\mathrm{M},\mathrm{N}{+}1,\mathrm{R})$$

For M≥0, this is the equivalent (*R*+1 = R) *invariant* of new add:

$$M{+}1 + N = R{+}1 \quad \textbf{if} \quad M + N = R$$

$$\mathrm{add}(M{+}1,N,R{+}1) \quad \textbf{if} \quad \mathrm{add}(M,N,R)$$

Notation:

add(suc[M],N,suc[R]) :- add(M,N,R).

LP: An Equivalent Successor-Arithmetic Addition Relation (II)

Horn Logic Rule with Recursive Call as a Single Premise (Tail-Recursion):

add(0,N,N).

Base Case: Termination

add(suc[M],N,suc[R]) :- **add(**M,N,R**)**.

General Case: Recursion

No Built-ins Required; 1+ replaced by suc (successor) structures

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

add(suc[suc[suc[0]]],suc[suc[suc[suc[0]]]],A)
add(suc[suc[0]],suc[suc[suc[suc[0]]]],R1) bind: A=suc[R1]
add(suc[0],suc[suc[suc[suc[0]]]],R2) bind: R1=suc[R2]
add(0,suc[suc[suc[suc[0]]]],R3) bind: R2=suc[R3]
R3=suc[suc[suc[suc[0]]]]
A=suc[suc[suc[suc[suc[suc[suc[0]]]]]]]

General: Recursion

Base: Termination

Recursion

FP: A Tail-Recursive Float-Number Compound Interest Function

Un/Conditional Equations with Recursive Call as a Foot (Tail-Recursion):

compint(0,I,C) :& C. % T: Time, I: Interest, C: Capital — Termination

compint(T,I,C) :- >(T,0) & **compint(**1-(T),I,+(C,*(C,I))**)**. — Recursion

Built-ins: > Greater 1- Predecessor + (Float) Addition * Multiplication

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

compint(3,0.1,100)
compint(2,0.1,110.0)
compint(1,0.1,121.0)
compint(0,0.1,133.1)

General Case: Recursion

133.1

Base Case: Termination

LP: A Tail-Recursive Float-Number Compound Interest Relation

Datalog Rule with Recursive Call as a Last Premise (Tail-Recursion):

compint(0,I,C,C). % T: Time, I: Interest, C: Capital, R: Result

Termination

compint(T,I,C,R) :- >(T,0), S .= 1-(T), D .= +(C,*(C,I)),
compint(S,I,D,R**)**.

Recursion

Built-ins: > Greater 1- Predecessor + (Float) Addition * Multiplication

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

compint(3,0.1,100,A)
compint(2,0.1,110.0,R1)
compint(1,0.1,121.0,R2)

General Case: Recursion

compint(0,0.1,133.1,133.1)
A=R1=R2=133.1

Base Case: Termination

FLP: A Tail-Recursive Float-Number Compound Interest Relation

Datalog-like Rule with Recursive Call as a Last Premise (Tail-Recursion):

compint(0,I,C,C). % T: Time, I: Interest, C: Capital, R: Result — Termination

compint(T,I,C,R) :- >(T,0),
compint(1-(T),I,+(C,*(C,I)),R**)**. — Recursion: Over Nestings

Built-ins: > Greater 1- Predecessor + (Float) Addition * Multiplication

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

compint(3,0.1,100,A)
compint(2,0.1,110.0,R1)
compint(1,0.1,121.0,R2) — General Case: Recursion

compint(0,0.1,133.1,133.1) — Base Case: Termination
A=R1=R2=133.1

FLP and ‘while’ Program: A Tail-Recursive and an Iterative Interest Relation

Declarative (Tail-Recursive FLP) Version Can Exchange Clause Order:

compint(T,I,C,R) :- >(T,0),
compint(1-(T),I,+(C,*(C,I)),R**)**.

Recursion: Over Nestings

compint(0,I,C,C).   % T: Time, I: Interest, C: Capital, R: Result

Termination

Imperative Version (‘while’ program) Uses Fixed Statement Order:

define compint(T,I,C,R) **as**
begin
  while >(T,0) **do**
    begin T := 1-(T); C := +(C,*(C,I)) **end**;

Iteration: Over Nestings

  if =(T,0) **then** R := C
end

Result Assignment after Termination

Instantiating cns Structures and the N-ary List Notation

Structures with constructor cns were introduced in the 'Terms' chapter:

cns[a,nil]	cns[a,cns[7,nil]]	cns[First,Rest]

They have been shortened via the N-ary list notation:

[a]	[a,7]	[First|Rest]

Variables as elements of (cns) structures are instantiated:

X .= a & cns[X,nil]	Y .= add(3,4) & cns[a,cns[Y,nil]]	First .= 1, Rest .= nil & cns[First,Rest]
cns[a,nil]	cns[a,cns[7,nil]]	cns[1,nil]

Variables as elements of the N-ary list notation are likewise instantiated:

X .= a & [X]	Y .= add(3,4) & [a,Y]	First .= 1, Rest .= nil & [First|Rest]
[a]	[a,7]	[1]

The cns Function for Constructing Lists as Structures or in N-ary List Notation

Function applications are forbidden as elements of structures and lists (variable instantiations as above permit to construct the desired data):

~~cns[a,cns[**add(3,4)**,nil]]~~ ~~[a,**add(3,4)**]~~

"No (active) round parentheses inside [passive] square brackets"

However, besides the *constructor* cns, also a *function* cns can be defined in either of the following ways (acting like Lisp's built-in function cons):

cns(First,Rest) :& cns[First,Rest]. cns(First,Rest) :& [First|Rest].

Actual cns arguments are evaluated to elements of cns structures or lists:

cns(a,cns(**add(3,4)**,nil))

cns[a,cns[*7*,nil]] [a,*7*]

FP: A Recursive List-Concatenation Function (I)

For first argument ≠ nil, this is a recursion *invariant* of cat (‘concatenate’ or just ‘catenate’, often named ‘append’, here alternatively written as a ⊕ infix):

$$[F|R] \oplus L \quad = \quad cns(F, R \oplus L)$$

$$cat([F|R],L) \quad = \quad cns(F,cat(R,L))$$

Notation:

$$cat([F|R],L) \;\; :\& \;\; cns(F,cat(R,L)).$$

FP: A Recursive List-Concatenation Function (II)

Unconditional Equations with Recursive Call inside cns (Full Recursion):

cat([],L) :& L. — Base Case: Termination

cat([F|R],L) :& cns(F,**cat(**R,L**)**). — General Case: Recursion

No Built-ins Required; cns regarded as a user-defined auxiliary

Full-Recursive Computation Grows and Shrinks an Activation Stack:

cat([a,b],[c,d,e])
cns(a, cat([b],[c,d,e]))
cns(a, *cns(b, cat([],[c,d,e]))*)
cns(a, cns(b, *[c,d,e]*))
cns(a, *[b,c,d,e]*)
[a,b,c,d,e]

General Case: Recursion

Base Case: Termination

LP: A Tail-Recursive List-Concatenation Relation (I)

For first argument ≠ nil, this is a recursion *invariant* of cat:

$[F|R] \oplus L = [F|S]$ **if** $R \oplus L = S$

cat([F|R],L,[F|S]) **if** cat(R,L,S)

Note analogy to the previous ‘new add’:

add(1+M,N,1+R) **if** add(M,N,R)

[lists ‘generalize’ natural numbers:
list concatenation ‘generalizes’ addition]

Notation:

cat([F|R],L,[F|S]) :- cat(R,L,S).

LP: A Tail-Recursive List-Concatenation Relation (II)

Horn Logic Rule with Recursive Call as a Single Premise (<u>Tail</u>-Recursion):

cat([],L,L).

Base Case: Termination

cat([F|R],L,[F|S]) :- **cat(**R,L,S**)**.

General Case: Recursion

No Built-ins Required

Tail-Recursive Computation Loops over a Fixed-Size Activation Record:

cat([a,b],[c,d,e],A)

cat([b],[c,d,e],S1) bind: A=[a|S1]

General: Recursion

cat([],[c,d,e],S2) bind: S1 =[b|S2]

A=[a|S1]=[a|[b|S2]]=[a|[b|[c,d,e]]]=[a,b,c,d,e]

Base: Termination

LP: A Tail-Recursive List-Concatenation Relation (III)

Catenations can be inverted for list ‘subtraction’:

[a,b] $\oplus$ *W* = [a,b,c,d,e]

cat([a,b],W,[a,b,c,d,e])

W=[c,d,e]

V $\oplus$ [c,d,e] = [a,b,c,d,e]

cat(V,[c,d,e],[a,b,c,d,e])

V=[a,b]

LP: A Tail-Recursive List-Concatenation Relation (IV)

Can also be inverted for non-deterministic partitioning:

$V \oplus W$ = [a,b,c,d,e]

cat(V,W,[a,b,c,d,e])

V=[], W=[a,b,c,d,e]
V=[a], W=[b,c,d,e]
V=[a,b], W=[c,d,e]
V=[a,b,c], W=[d,e]
V=[a,b,c,d], W=[e]
V=[a,b,c,d,e], W=[]

FP: A Recursive List-Reversal Function (I)

For first argument ≠ nil, this is a recursion *invariant* of rev:

$$\mathrm{rev}([F|R]) \quad = \quad \mathrm{rev}(R) \oplus [F]$$

$$\mathrm{rev}([F|R]) \quad = \quad \mathrm{cat}(\mathrm{rev}(R),[F])$$

Notation:

$$\mathrm{rev}([F|R]) \quad :\& \quad \mathrm{cat}(\mathrm{rev}(R),[F]).$$

FP: A Recursive List-Reversal Function (II)

Unconditional Equations with Recursive Call inside cat (Full Recursion):

rev([]) :& [].

Base Case: Termination

rev([F|R]) :& cat(**rev(**R**)**,[F]).

General Case: Recursion

No Built-ins Required; cat is our user-defined auxiliary

Full-Recursive Computation Grows and Shrinks an Activation Stack:

<u>rev([a,b,c])</u>
cat(<u>rev([b,c])</u>, [a])
cat(*cat(<u>rev([c])</u>, [b])*, [a])
cat(cat(*cat(<u>rev([])</u>, [c])*, [b]), [a])
cat(cat(<u>cat(*[]*, [c])</u>, [b]), [a])
. . .
[c,b,a]

General Case: Recursion

Base Case: Termination

LP: A Recursive List-Reversal Relation (I)

For first argument ≠ nil, this is a recursion *invariant* of rev:

rev([F|R]) = L **if** rev(R,K) **and** K $\oplus$ [F] = L
rev([F|R],L) **if** rev(R,K) **and** cat(K,[F],L)

Notation:

rev([F|R],L) **:-** rev(R,K) **,** cat(K,[F],L).

LP: A Recursive List-Reversal Relation (II)

Horn Logic Rule with Recursive Call as a First Premise (Full Recursion):

rev([],[]).

Base Case: Termination

rev([F|R],L) :- **rev(**R,K**)** , cat(K,[F],L).

General Case: Recursion

No Built-ins Required; cat is our user-defined auxiliary

Full-Recursive Computation Grows and Shrinks an Activation Stack:

<u>rev([a,b,c],A)</u>

General

<u>rev([b,c],K1)</u>, cat(K1,[a],L1) bind: A=L1

<u>rev([c],K2)</u>, cat(K2,[b],L2), cat(K1,[a],L1) bind: K1=L2

<u>rev([],K3)</u>, cat(<u>K3</u>,[c],L3), cat(K2,[b],K1), cat(K1,[a],L1) bind: K2=L3

Base

<u>cat(*[]*,[c],K2)</u>, cat(K2,[b],K1), cat(K1,[a],L1)

. . .

A=L1=*[c,b,a]*

Summary

- Recursion is the basic ‘control structure’ of both FP and LP
- A taxonomy of recursion includes tail recursion (corresponding to iteration) and full recursion
- Recursion invariants were given for all operations before their actual definitions
- Recursive definitions of arithmetic and list operations were compared for FP and LP
- Relations not calling built-ins permit inverted calls
- Certain programs are tail-recursive in LP but fully recursive in FP

Higher-Order Operations (Higher-Order Functions and Relations)

Higher Order

Higher Order

Chapter 5

Higher-Order Operations: Operations as 1st-Class Citizens

In ***higher-order operations***,
operations (functions and relations)
are **1st-class citizens**
in that they can themselves be

- Passed to calls as (actual) parameters/arguments
- **D**elivered from operation calls:
  - Returned as values of function calls
  - Assigned to request variables of relation calls
- **U**sed as elements of structures (and of lists)
- **A**ssigned to local variables (single-assignment)

Taxonomy of 1st-Order and Higher-Order Operations

- Operation
  - Function (FP)
    - 1st-Order
      (no functions as
      arguments or values)
    - Higher-Order
      (functions as
      arguments or values)
  - Function (FLP)
    - 1st-Order
      (no operations as
      arguments, values or bindings;
      no operation variables)
    - Higher-Order
      (operations as arguments, values or bindings;
      operation variables)
  - Relation (LP)
    - 1st-Order
      (no relations as
      arguments or bindings;
      no relation variables)
    - Higher-Order
      (relations as
      arguments or bindings;
      relation variables)
      - 2nd-Order
        (used relations are
        themselves 1st-order)

FP: Function Composition as a Higher-Order Function (I)

- In the introductory chapter, we discussed the *function composition* **en2fr∘en-antonym∘fr2en** constituting the function **fr-antonym**
- The '∘' can be regarded as the infix version of an (associative) binary **compose** *higher-order function*, which – when passed two functional arguments – delivers (returns) their composition as a new function:

  en-antonym∘fr2en becomes **compose(en-antonym,fr2en)**

  en2fr∘en-antonym∘fr2en becomes **compose(en2fr,compose(en-antonym,fr2en))** or **compose(compose(en2fr,en-antonym),fr2en)**

FP: Function Composition as a Higher-Order Function (II)

- However, we want to permit simple definitions of higher-order functions (without so-called λ-*variables* for defining new anonymous functions)
- Hence '∘' is regarded here as the infix version of an (associative) binary *higher-order constructor* **compose** while the entire structure **compose[*f*,*g*]** is regarded as a complex *higher-order function* name:

en-antonym∘fr2en becomes **compose[en-antonym,fr2en]**

en2fr∘en-antonym∘fr2en becomes **compose[en2fr,compose[en-antonym,fr2en]]** or **compose[compose[en2fr,en-antonym],fr2en]**

FP: Application of Compose as a Higher-Order Function

Such a higher-order function structure can be applied to arguments as follows:

en-antonym∘fr2en(noir) becomes
compose[en-antonym,fr2en](noir)

returning **white**

parameters argument

en2fr∘en-antonym∘fr2en(noir) becomes
compose[en2fr,compose[en-antonym,fr2en]](noir) or
compose[compose[en2fr,en-antonym],fr2en](noir)

returning **blanc**

FP: Definition of Compose as a Higher-Order Function

The higher-order operation **compose** can be defined as follows, where **F** and **G** are *function variables* (their values should be function names or terms), while **X** is an *object variable* (its values should be normal terms):

Math: **compose(F,G)(X) = F(G(X))**

Relfun: **compose[F,G](X) :& F(G(X)).**

FP: Computation with Simple Compose as a Higher-Order Function

compose[en-antonym,fr2en](noir)

en-antonym(fr2en(noir))

en-antonym(*black*)

white

FP: Computation with Nested Compose as a Higher-Order Function

compose[en2fr,compose[en-antonym,fr2en]](noir)
en2fr(compose[en-antonym,fr2en](noir))
en2fr(*en-antonym(fr2en(noir))*)
en2fr(en-antonym(*black*))
en2fr(*white*)
blanc

compose[compose[en2fr,en-antonym],fr2en](noir)
compose[en2fr,en-antonym](fr2en (noir))
compose[en2fr,en-antonym](*black*)
en2fr(en-antonym(black))
en2fr(*white*)
blanc

LP: Relational Product as a Higher-Order Relation (I)

- The relation **fr-antonym** of the introductory chapter can be viewed as constituting a *relational product* **fr4en•en-antonym•en4fr**, where **en4fr** inverts **fr4en**:

  en4fr(En,Fr) :- fr4en(Fr,En).

- The '•' can be regarded as the infix version of an (associative) binary **product** *higher-order operation*:

  fr4en•en-antonym becomes
  product(fr4en,en-antonym)

  fr4en•en-antonym•en4fr becomes
  product(fr4en, product(en-antonym,en4fr)) or
  product(product(fr4en,en-antonym),en4fr)

LP: Relational Product as a Higher-Order Relation (II)

- However, we want to use simple definitions of pure higher-order relations (again avoiding λ-variables)
- Hence '•' is regarded here as the infix version of an (associative) binary *higher-order constructor* **product** while the entire structure **product[*r*,*s*]** is regarded as a *higher-order relation*:

fr4en•en-antonym becomes **product[fr4en,en-antonym]**

fr4en•en-antonym•en4fr becomes **product[fr4en,product[en-antonym,en4fr]]** or **product[product[fr4en,en-antonym],en4fr]**

LP: Application of Product as a Higher-Order Relation

Such a higher-order relation structure can be applied to arguments as follows:

fr4en•en-antonym(noir,Res) becomes **product[fr4en,en-antonym](noir,Res)**

binding **Res**=**white**

fr4en•en-antonym•en4fr(noir,Res) becomes **product[fr4en,product[en-antonym,en4fr]](noir,Res)** or **product[product[fr4en,en-antonym],en4fr](noir,Res)**

binding **Res**=**blanc**

LP: Definition of Product as a Higher-Order Relation

The higher-order operation **product** can be defined as follows, where **R** and **S** are *relation variables* (their values should be relation names or terms), while **X**, **Y**, and **Z** are *object variables* (their values should be normal terms):

Math: **product(R,S)(X,Z) if R(X,Y) and S(Y,Z)**

Relfun: **product[R,S](X,Z) :- R(X,Y), S(Y,Z).**

LP: Computation with Simple Product as a Higher-Order Relation

product[fr4en,en-antonym](noir,Res)

*fr4en(noir,Y1), en-antonym(Y1,*Res*)*

en-antonym(*black*,Res)

Res = white

LP: Computation with Nested Product as a Higher-Order Relation

product[fr4en,product[en-antonym,en4fr]](noir,Res)
fr4en(noir,Y1), product[en-antonym,en4fr](Y1,Res)
product[en-antonym,en4fr](*black*,Res)
en-antonym(black,Y2), en4fr(Y2,Res)
en4fr(*white*,Res)
Res = blanc

product[product[fr4en,en-antonym],en4fr](noir,Res)
product[fr4en,en-antonym](noir,Y1), en4fr(Y1,Res)
fr4en(noir,Y2), en-antonym(Y2,Y1), en4fr(Y1,Res)
en-antonym(*black*,Y1), en4fr(Y1,Res)
en4fr(*white*,Res)
Res = blanc

FP: A Function-Mapping Higher-Order Function

1. Consider a higher-order function for mapping a function over – applying it to – all elements of a list; e.g., **a2a[sqrt]([1,4,9])** maps built-in function **sqrt** over the elements **1**, **4**, and **9**, returning **[1,2,3]**
2. Versions of this have been used in many functional languages; in Common Lisp it is a binary function; e.g., **(mapcar #'sqrt '(1 4 9))** returns **(1 2 3)**
3. The unary version 1. will, however, permit nestings: **a2a[a2a[sqrt]]([[1,4,9],[16,25]])** maps **a2a[sqrt]** over **[1,4,9]** and **[16,25]**, returning **[[1,2,3],[4,5]]**
4. Can also be combined with higher-order **compose**: **a2a[compose[sqrt,1+]]([0,3,8])** returns **[1,2,3]**

FP: Definition of, and Computation with, the a2a Higher-Order Function

a2a[F]([]) :& [].
a2a[F]([First|Rest]) :& cns(F(First), a2a[F](Rest)).

a2a[sqrt]([1,4,9])
cns(sqrt(1), a2a[sqrt]([4,9]))
cns(*1* , *cns(sqrt(4), a2a[sqrt]([9]))*)
cns(1 , cns(*2, cns(sqrt(9), a2a[sqrt]([]))*))
cns(1 , cns(2, cns(*3 , []*)))
cns(1 , cns(2, *[3]*))
cns(1 , *[2,3]*)
[1,2,3]

LP: A Relation-Mapping Higher-Order Relation

- Similarly, consider a higher-order relation for mapping a relation over all elements of a list
- Since there are few built-in relations, assume a user-defined relation, e.g. **dup(N,[N,N]).**
- Now, e.g. **a2a[dup]([1,4,9],Res)** maps the relation **dup** over **1**, **4**, and **9**, binding **Res = [[1,1],[4,4],[9,9]]**
- The mapped list may be non-ground, as in **a2a[dup]([1,J,9],Res)**, giving **Res = [[1,1],[J,J],[9,9]]**
- The mapped relation may be non-deterministic, leading to several bindings for the result list
- Versions of such higher-order syntax have been used in many logic languages, e.g. in ISO Prolog

LP: Relation Variables as 2nd-Order Syntactic Sugar (I)

- Consider an RDF-like binary fact base describing individuals or resources in the first argument, e.g.:

  transmission("Honda","Automatic").
  air-conditioning("Honda","Automatic").
  color("Honda","Eternal Blue Pearl").

- 1st-order queries – relation given, object asked:
  transmission("Honda",Kind)
  binds object variable **Kind = "Automatic"**
- 2nd-order queries – objects given, relation asked:
  Feature("Honda","Automatic")
  binds relation variable **Feature = transmission**
  and then binds **Feature = air-conditioning**

LP: Relation Variables as 2nd-Order Syntactic Sugar (II)

- LP 2nd-order queries are useful in practice, but are 'syntactic sugar' that can be eliminated in the semantics and in the implementation – a ternary dummy relation **apply** shifts the original relation into the first argument position, e.g.:

  apply(transmission,"Honda","Automatic").
  apply(air-conditioning,"Honda","Automatic").
  apply(color,"Honda","Eternal Blue Pearl").

- This leaves us with only 1st-order queries:
  apply(Feature,"Honda","Automatic")
  binds object variable **Feature = transmission**
  and then binds **Feature = air-conditioning**

FLP: Function Variables as 2nd-Order Syntactic Sugar (I)

- Similarly, consider the unary point base describing individuals or resources in the single argument, e.g.:

  transmission("Honda") :& "Automatic".
  air-conditioning("Honda") :& "Automatic".
  color("Honda") :& "Eternal Blue Pearl".

- 1st-order queries – function given, object asked:
  transmission("Honda")
  returns object **"Automatic"**
- 2nd-order queries – objects given, function asked:
  "Automatic" .= Feature("Honda")
  binds function variable **Feature = transmission**
  and then binds **Feature = air-conditioning**

FLP: Function Variables as 2nd-Order Syntactic Sugar (II)

- FLP 2nd-order queries are also useful in practice, but again are syntactic sugar that can be eliminated in the semantics and in the implementation – a binary dummy function **apply** shifts the original function into the first argument position, e.g.:

  apply(transmission,"Honda") :& "Automatic".
  apply(air-conditioning,"Honda") :& "Automatic".
  apply(color,"Honda") :& "Eternal Blue Pearl".

- This leaves us with only 1st-order queries:
  "Automatic" .= apply(Feature,"Honda")
  binds object variable **Feature = transmission**
  and then binds **Feature = air-conditioning**

Summary

- In higher-order operations, operations are 1st-class citizens that are allowed at most ‘places’
- A taxonomy of 1st-order and higher-order operations was introduced, the latter permitting operations as arguments, values or bindings, as well as operation variables
- Function composition was discussed as a higher-order operation in FP; relational product as a corresponding higher-order operation in LP
- Higher-order operations that map functions or relations over lists were discussed for FP and LP
- Relation variables were considered as 2nd-order syntactic sugar for LP; function variables, for FLP
- Structure-named operations were used instead of λ-expressions (avoiding higher-order unification)

Non-Deterministic Definitions and Calls

Non-Determinism

Non-Determinism

Chapter 6

What is Non-Determinism?

- We have seen non-deterministic calls in earlier chapters
- Distinguished from indeterminism or random behavior, ***non-determinism*** gives computations limited choice on which control branches to follow
- Two versions of non-determinism have been studied (we will consider here only version 2.):
    1. ***Don't-care*** non-determinism: Once a choice has been made, the other alternatives at this point are discarded
    2. ***Don't-know*** non-determinism: When a choice is made, the other alternatives at this point are stored for later follow-up
- (Don't-know) Non-determinism is here – as in Prolog – realized by *depth-first search* (backtracking), but also *breadth-first search* or versions of *best-first search* have been used

Taxonomy of Deterministic vs. Non-Deterministic Definitions and Calls

- Definition
  - Deterministic (ground calls must generate 0 or 1 results; non-ground calls can generate > 1 result)
  - Non-Deterministic (ground calls can generate > 1 results)
- Call (ground or non-ground)
  - Deterministic (must have 0 or 1 results)
    - LP: ≤ 1 binding set
    - FP: ≤ 1 return value
    - FLP: ≤ 1 binding-return combination
  - Non-Deterministic (can have > 1 results)
    - LP: > 1 binding set
    - FP: > 1 return value
    - FLP: > 1 binding-return combination

LP: Deterministic Product-Offer Definition and its Ground Deterministic Calls

Facts on offered furniture products in available quantities at merchants:

offer(desk,10,furniffice).
offer(desk,15,moebureau).
offer(chair,**19**,furniffice).
offer(chair,20,moebureau).

Deterministic Definition

Deterministic (ground) call:

offer(desk,15,moebureau)

Does moebureau offer 15 desks?

succeeds, returning true

Deterministic (ground) call:

offer(chair,20,moebureau)

Does moebureau offer 20 chairs?

succeeds, returning true

FP: Deterministic Product-Offer Definition and its Ground Deterministic '.=' Calls

Points on offered furniture products in available quantities at merchants:

offer(desk,10) :& furniffice.
offer(desk,15) :& moebureau.
offer(chair,**19**) :& furniffice.
offer(chair,20) :& moebureau.

Deterministic Definition

Deterministic (ground) call:

moebureau .= offer(desk,15)

Does moebureau offer 15 desks?

succeeds, returning   moebureau

Deterministic (ground) call:

moebureau .= offer(chair,20)

Does moebureau offer 20 chairs?

succeeds, returning   moebureau

LP: Deterministic Product-Offer Definition and its Non-Ground Deterministic Calls

Facts on offered furniture products in available quantities at merchants:

offer(desk,10,furniffice).
offer(desk,15,moebureau).
offer(chair,**19**,furniffice).
offer(chair,20,moebureau).

Deterministic Definition

Deterministic (non-ground) call:

offer(desk,15,Merchant)

Which merchants offer 15 desks?

binds Merchant to moebureau

Deterministic (non-ground) call:

offer(chair,20,Merchant)

Which merchants offer 20 chairs?

binds Merchant to moebureau

FP: Deterministic Product-Offer Definition and its Ground Deterministic Calls

Points on offered furniture products in available quantities at merchants:

offer(desk,10) :& furniffice.
offer(desk,15) :& moebureau.
offer(chair,**19**) :& furniffice.
offer(chair,20) :& moebureau.

Deterministic Definition

Deterministic (ground) call:

offer(desk,15)

returns   moebureau

Which merchants offer 15 desks?

Deterministic (ground) call:

offer(chair,20)

returns   moebureau

Which merchants offer 20 chairs?

LP: Deterministic Product-Offer Definition and Deterministic/Non-Deterministic Calls

Facts on offered furniture products in available quantities at merchants:

offer(desk,10,furniffice).
offer(desk,15,moebureau).
offer(chair,**20**,furniffice).
offer(chair,20,moebureau).

Deterministic Definition

Deterministic (non-ground) call:

offer(desk,15,Merchant)

Which merchants offer 15 desks?

binds Merchant to moebureau

Non-deterministic (non-ground) call:

offer(chair,20,Merchant)

Which merchants offer 20 chairs?

binds Merchant to furniffice and (then) to moebureau

FP: Non-Deterministic Product-Offer Definition and its Non-/Deterministic Calls

Points on offered furniture products in available quantities at merchants:

offer(desk,10) :& furniffice.
offer(desk,15) :& moebureau.
offer(chair,**20**) :& furniffice.
offer(chair,20) :& moebureau.

Non-Deterministic Definition

Deterministic (ground) call:

offer(desk,15)

Which merchants offer 15 desks?

returns moebureau

Non-deterministic (ground) call:

offer(chair,20)

Which merchants offer 20 chairs?

returns furniffice and (then) moebureau

LP: Deterministic Product-Offer Definition and its Non-Deterministic Calls

Facts on offered furniture products in available quantities at merchants:

offer(desk,10,furniffice).
offer(desk,15,moebureau).
offer(chair,**21**,furniffice).
offer(chair,20,moebureau).

Deterministic Definition

Non-deterministic (non-ground) call:

offer(desk,Quantity,Merchant)

Which merchants offer how many desks?

binds Quantity=10, Merchant=furniffice
and Quantity=15, Merchant=moebureau

Non-deterministic (non-ground) call:

offer(chair,Quantity,Merchant)

Which merchants offer how many chairs?

binds Quantity=21, Merchant=furniffice
and Quantity=20, Merchant=moebureau

FLP: Deterministic Product-Offer Definition and its Non-Deterministic Calls

Points on offered furniture products in available quantities at merchants:

offer(desk,10) :& furniffice.
offer(desk,15) :& moebureau.
offer(chair,**21**) :& furniffice.
offer(chair,20) :& moebureau.

Deterministic Definition

Non-deterministic (non-ground) call:

offer(desk,Quantity)
returns furniffice, binding Quantity=10
returns moebureau, binding Quantity=15

Which merchants offer how many desks?

Non-deterministic (non-ground) call:

offer(chair,Quantity)
returns furniffice, binding Quantity=21
returns moebureau, binding Quantity=20

Which merchants offer how many chairs?

LP: Deterministic Offer+Contact Definitions for Non-/Deterministic Conjunctions

Facts on offered furniture products and their merchants' contact persons:

offer(desk,10,furniffice).
offer(desk,15,moebureau).
offer(chair,20,furniffice).
offer(chair,20,moebureau).

contact(furniffice,roberts).
contact(furniffice,sniders).
contact(furniffice,tellers).
contact(moebureau,leblanc).

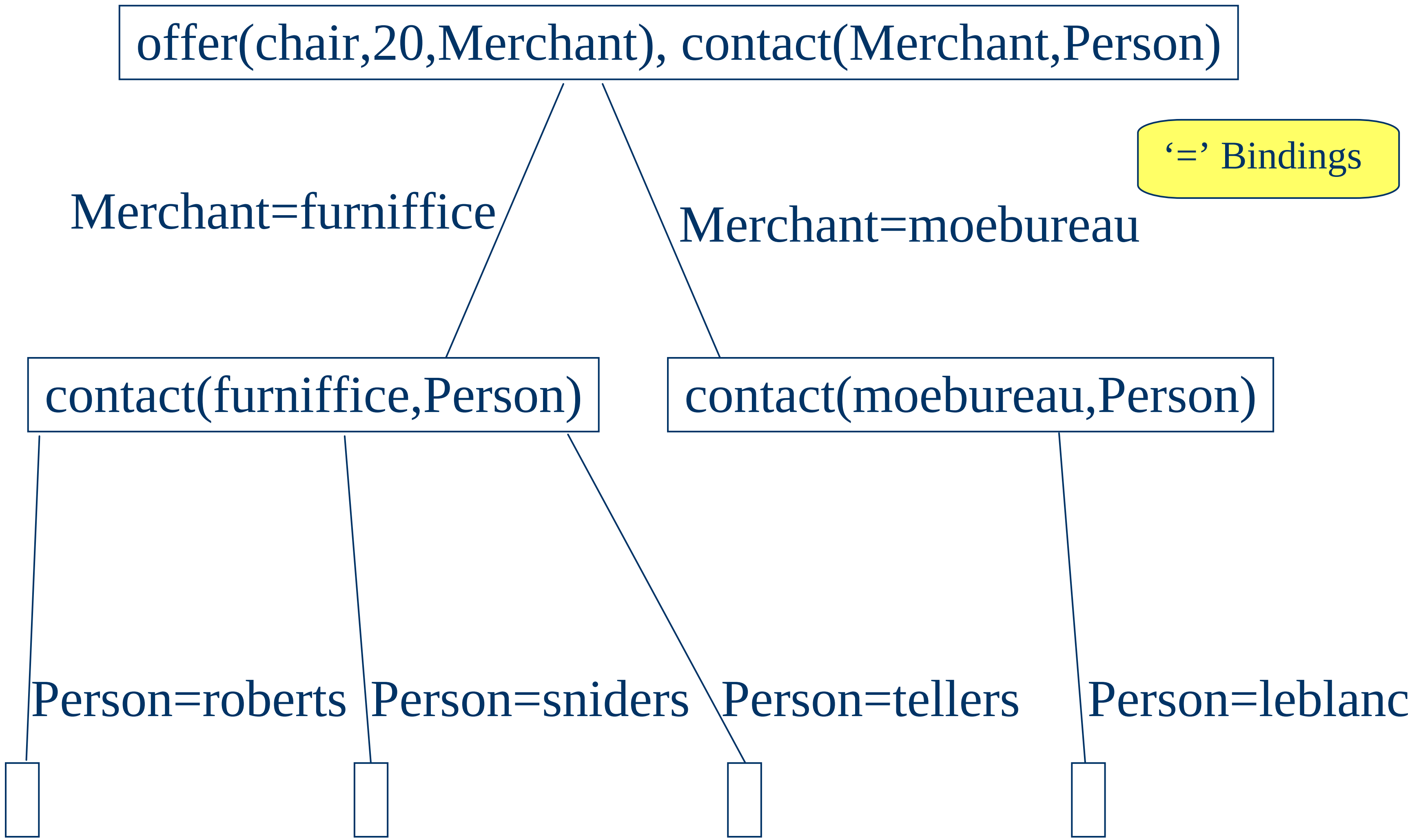
Deterministic (non-ground) call conjunction – relational join:

offer(desk,15,Merchant), contact(Merchant,Person)
binds Merchant to moebureau and Person to leblanc

Non-deterministic (non-ground) call conjunction – relational join:

offer(chair,20,Merchant), contact(Merchant,Person)
binds Merchant to furniffice and Person to roberts, sniders, tellers
and Merchant to moebureau and Person to leblanc

LP: Proof Tree for the Non-Deterministic Call Conjunction



FP: Non-Deterministic Offer+Contact Definitions for Non-/Deterministic Nestings

Points on offered furniture products and their merchants' contact persons:

offer(desk,10) :& furniffice.
offer(desk,15) :& moebureau.
offer(chair,20) :& furniffice.
offer(chair,20) :& moebureau.

contact(furniffice) :& roberts.
contact(furniffice) :& sniders.
contact(furniffice) :& tellers.
contact(moebureau) :& leblanc.

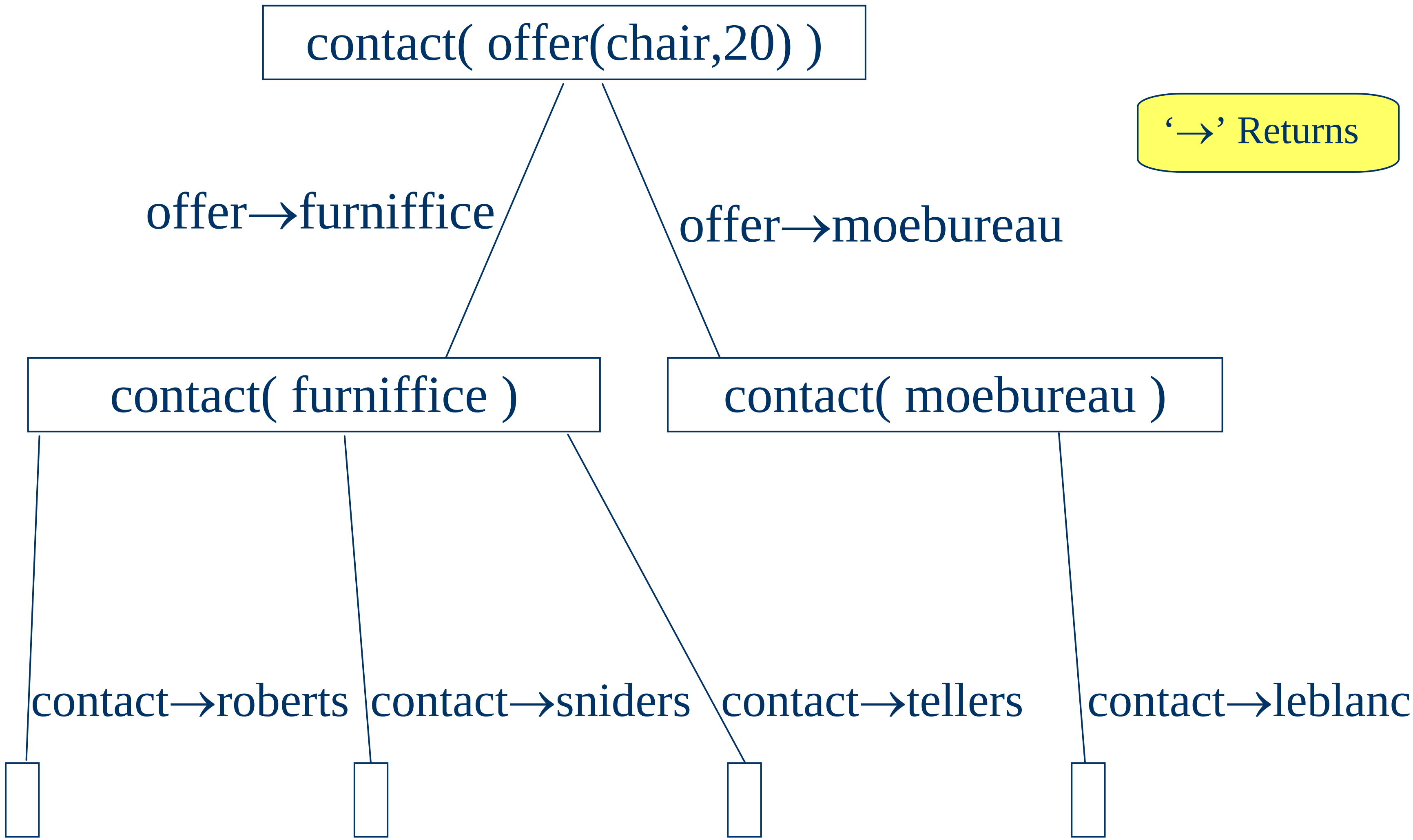
Deterministic (ground) call nesting:

contact(offer(desk,15))
via contact(moebureau) returns leblanc

Non-deterministic (ground) call nesting:

contact(offer(chair,20))
via contact(furniffice) returns roberts, sniders, tellers
via contact(moebureau) returns leblanc

FP: Proof Tree for the Non-Deterministic Call Nesting



LP: Deterministic Offer+Site Definitions for Non-/Deterministic Conjunctions

Facts on offered furniture products and their merchants' sites:

offer(desk,10,furniffice).
offer(desk,15,moebureau).
offer(chair,20,furniffice).
offer(chair,20,moebureau).

site(furniffice,fredericton).
site(furniffice,moncton).
site(moebureau,moncton).

Deterministic (non-ground) call conjunction – relational join:

offer(desk,15,Merchant), site(Merchant,Town)
binds Merchant to moebureau and Town to moncton

Non-deterministic (non-ground) call conjunction – relational join:

offer(chair,20,Merchant), site(Merchant,Town)
binds Merchant to furniffice and Town to fredericton, moncton
and Merchant to moebureau and Town again to moncton

FP: Non-Deterministic Offer+Site Definitions for Non-/Deterministic Nestings

Points on offered furniture products and their merchants' sites:

offer(desk,10) :& furniffice.
offer(desk,15) :& moebureau.
offer(chair,20) :& furniffice.
offer(chair,20) :& moebureau.

site(furniffice) :& fredericton.
site(furniffice) :& moncton.
site(moebureau) :& moncton.

Deterministic (ground) call nesting:

site(offer(desk,15))
via site(moebureau) returns moncton

Non-deterministic (ground) call nesting:

site(offer(chair,20))
via site(furniffice) returns fredericton, moncton
via site(moebureau) again returns moncton

LP: Deterministic Offer+Site Definitions for Deterministic Conjunctions

Facts on offered furniture products and their merchants' sites:

offer(desk,10,furniffice).
offer(desk,15,moebureau).
offer(chair,20,furniffice).
offer(chair,20,moebureau).

site(furniffice,fredericton).
site(furniffice,moncton).
site(moebureau,moncton).

Deterministic conjunction:

offer(desk,15,Merchant), site(Merchant,moncton)
binds Merchant to moebureau

Internally non-deterministic, externally deterministic conjunction:

offer(chair,20,Merchant), site(Merchant,fredericton)
binds Merchant to furniffice
(then, with Merchant=moebureau, site(Merchant,fredericton) fails)

FP: Non-Deterministic Offer+Site Definitions for Deterministic Nestings

Points on offered furniture products and their merchants' sites:

offer(desk,10) :& furniffice.
offer(desk,15) :& moebureau.
offer(chair,20) :& furniffice.
offer(chair,20) :& moebureau.

site(furniffice) :& fredericton.
site(furniffice) :& moncton.
site(moebureau) :& moncton.

Deterministic nesting:

moncton .= site(offer(desk,15))
via moncton .= site(moebureau) returns moncton

Internally non-deterministic, externally deterministic nesting:

fredericton .= site(offer(chair,20))
via fredericton .= site(furniffice) returns fredericton
(then, via fredericton .= site(moebureau) fails)

LP: Deterministic Offer+Site Definitions for Non-/Deterministic Conjunctions

Facts on offered furniture products and their merchants' sites:

offer(desk,10,furniffice).
offer(desk,15,moebureau).
offer(chair,20,furniffice).
offer(chair,20,moebureau).

site(furniffice,fredericton).
site(furniffice,moncton).
site(moebureau,moncton).

Non-deterministic conjunction:

offer(desk,Quantity,Merchant), site(Merchant,moncton)
binds Quantity=10, Merchant=furniffice
binds Quantity=15, Merchant=moebureau

Internally non-deterministic, externally deterministic conjunction:

offer(chair,Quantity,Merchant), site(Merchant,fredericton)
binds Quantity=20, Merchant=furniffice
(then, with Merchant=moebureau, site(Merchant,fredericton) fails)

FLP: Non-Deterministic Offer+Site Definitions for Non-/Deterministic Nestings

Points on offered furniture products and their merchants' sites:

offer(desk,10) :& furniffice.
offer(desk,15) :& moebureau.
offer(chair,20) :& furniffice.
offer(chair,20) :& moebureau.

site(furniffice) :& fredericton.
site(furniffice) :& moncton.
site(moebureau) :& moncton.

Non-deterministic nesting:

moncton .= site(offer(desk,Quantity))
via moncton .= site(furniffice) returns moncton, binds Quantity=10
via moncton .= site(moebureau) returns moncton, with Quantity=15

Internally non-deterministic, externally deterministic nesting:

fredericton .= site(offer(chair,Quantity))
via fredericton .= site(furniffice) gives fredericton, Quantity =20
(then, via fredericton .= site(moebureau) fails)

LP: Deterministic Site Definition for Deterministic Conjunction

Facts on merchants' sites:

site(furniffice,fredericton).
site(furniffice,moncton).
site(moebureau,moncton).

(Externally) Deterministic conjunction:

Which merchants (only different ones, and in alphabetical order) are in the same town?

Based on Built-in:
string< String-Less

site(Merch1,Town), site(Merch2,Town), string<(Merch1,Merch2)

binds Merch1= furniffice, Merch2=moebureau, Town= moncton
(Merch1=Merch2=furniffice etc. are rejected by string<)

FLP: Deterministic Site Definition for Deterministic Conjunction

Points on merchants' sites:

site(furniffice) :& fredericton.
site(furniffice) :& moncton.
site(moebureau) :& moncton.

(Externally) Deterministic conjunction:

Based on Built-in:
string< String-Less

Which merchants (only different ones, and in alphabetical order) are in the same town?

Town .= site(Merch1), Town .= site(Merch2),
string<(Merch1,Merch2)

binds Merch1= furniffice, Merch2=moebureau, Town= moncton
(Merch1=Merch2=furniffice etc. are rejected by string<)

FP: Cartesian Product by a Repeated Non-Deterministic Call

Points on offers and a pair definition:

```
offer(chair,20) :& furniffice.
offer(chair,20) :& moebureau.

pair(First,Second) :& [First,Second].      % similar to active cns
```

Repeated non-deterministic call enumerates entire Cartesian product:

```
pair(offer(chair,20),offer(chair,20))  % {[X,Y] | X,Y ∈ offer(chair,20)}

returns    [furniffice,furniffice]      % {[furniffice,furniffice],
returns    [furniffice,moebureau]       %  [furniffice,moebureau],
returns    [moebureau,furniffice]       %  [moebureau,furniffice],
returns    [moebureau,moebureau]        %  [moebureau,moebureau]}
```

FP: Subset of Cartesian Product by a Named Non-Deterministic Call

Points on offers and a pair definition:

offer(chair,20) :& furniffice.
offer(chair,20) :& moebureau.

pair(First,Second) :& [First,Second].   % similar to active cns

Named non-deterministic call enumerates Cartesian product subset:

Oc .= offer(chair,20) & pair(Oc,Oc) % {[Oc,Oc] | Oc ∈ offer(chair,20)}

returns   [furniffice,furniffice],   binding Oc=furniffice
returns   [moebureau,moebureau], binding Oc=moebureau

FP: Cartesian Product Multiset via a Repeated Non-Deterministic Call

Points on offers, their merchants' contacts + sites, and a pair definition:

offer(chair,20) :& furniffice.
offer(chair,20) :& moebureau.

contact(furniffice) :& tellers.
contact(moebureau) :& leblanc.

site(furniffice) :& fredericton.
site(furniffice) :& moncton.
site(moebureau) :& moncton.

pair(First,Second) :& [First,Second].   % similar to active cns

Repeated non-deterministic call enumerates entire Cartesian product:

pair(site(offer(chair,20)),contact(offer(chair,20)))
returns [fredericton,tellers]
returns [fredericton,leblanc]   while moebureau's site is moncton
returns [moncton,tellers]
returns [moncton,leblanc]
again returns [moncton,tellers]
again returns [moncton,leblanc]

FP: Subset of Cartesian Product Multiset via a Named Non-Deterministic Call

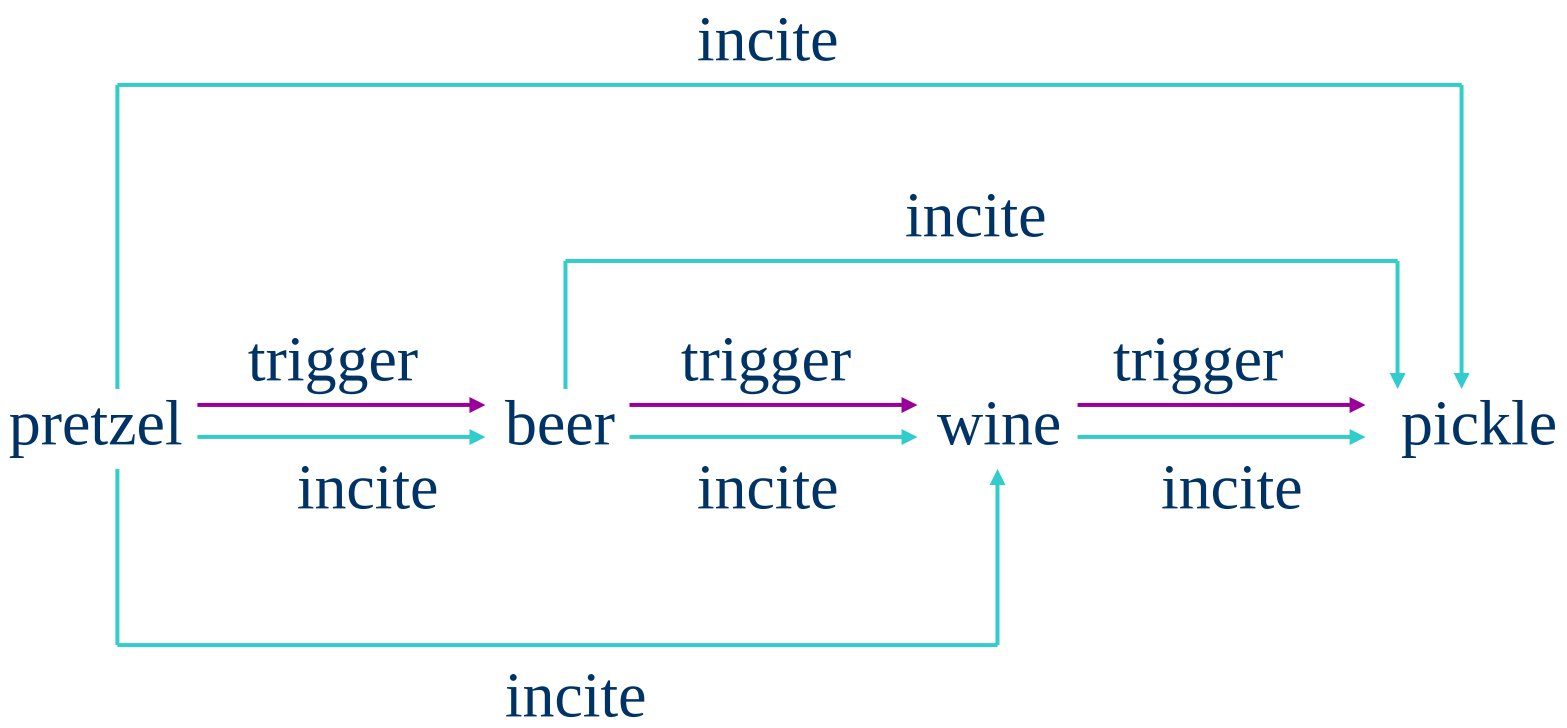
Points on offers, their merchants' contacts + sites, and a pair definition:

```
offer(chair,20) :& furniffice.        contact(furniffice) :& tellers.
offer(chair,20) :& moebureau.         contact(moebureau) :& leblanc.

site(furniffice) :& fredericton.      site(furniffice) :& moncton.
                    site(moebureau) :& moncton.

pair(First,Second) :& [First,Second].      % similar to active cns
```

Named non-deterministic call enumerates Cartesian product subset:

```
Oc .= offer(chair,20)  &  pair(site(Oc),contact(Oc))
returns        [fredericton,tellers],   binding Oc=furniffice
returns        [moncton,tellers],       binding Oc=furniffice
returns        [moncton,leblanc],       binding Oc=moebureau
```

Preview of a Transitive Closure



The base relation or function trigger
has the transitive closure relation or function incite

LP: A Recursive Non-Deterministic Relational Closure Definition

Ground facts on the purchase of certain products triggering further ones:

```
trigger(pretzel,beer).
trigger(beer,wine).
trigger(wine,pickle).
```

Datalog Rules on recursive product incitement based on triggering:

```
incite(ProductA,ProductB)  :-  trigger(ProductA,ProductB).
incite(ProductA,ProductC)  :-  trigger(ProductA,ProductB),
                               incite(ProductB,ProductC).
```

These non-deterministic clauses are used to compute the **transitive closure** relation, incite, over a base relation, trigger

LP: A Recursive Non-Deterministic Relational Closure Computation

In each computation step:

- The call to be selected next is underlined
- Call results are put in *italics*
- A call with non-deterministic alternatives is **bold-faced**

incite(pretzel,Result)
trigger(pretzel,ProductB1)
Result=ProductB1=beer

trigger(pretzel,ProductB1), incite(ProductB1,ProductC1)
trigger(pretzel,beer), **incite(*beer*,ProductC1)**
trigger(beer,ProductC1)
Result=ProductC1=wine

trigger(beer,ProductB2), incite(ProductB2,ProductC2)
trigger(beer,wine), **incite(*wine*,ProductC2)**
trigger(wine,ProductC2)
Result=ProductC2=pickle

FP: A Recursive Non-Deterministic Functional Closure Definition

Ground points on the purchase of certain products triggering further ones:

trigger(pretzel) :& beer.
trigger(beer) :& wine.
trigger(wine) :& pickle.

Datafun Rules on recursive product incitement based on triggering:

incite(Product) :& trigger(Product).
incite(Product) :& incite(trigger(Product)).

These non-deterministic clauses are used to compute the **transitive closure** function, incite, over a base function, trigger

FP: A Recursive Non-Deterministic Functional Closure Computation

incite(pretzel)
trigger(pretzel)
beer

incite(trigger(pretzel))
incite(*beer*)
trigger(beer)
wine

incite(trigger(beer))
incite(*wine*)
trigger(wine)
pickle

In each computation step:

- The call to be selected next is underlined
- Call results are put in *italics*
- A call with non-deterministic alternatives is **bold-faced**

Summary

- (Don't-know) Non-determinism permits choice alternatives to be stored for later follow-up (e.g. via backtracking)
- Deterministic vs. non-deterministic definitions and calls were discussed in a taxonomy for FP and LP
- Non-ground calls can be non-deterministic even for deterministic definitions
- Non-deterministic FLP computations can be regarded as always resulting in a (finite or infinite) **set of value-binding combinations**:
  1. Empty set: Failure
  2. Singleton set: Special case of deterministic result
  3. Set with ≥ two elements: Non-deterministic result that can be 'unioned' with other such sets, incl. 1. and 2.

The Relational-Functional Markup Language (RFML)

RFML

Chapter 7

A 10-Step Strategy to Publish and Reuse Declarative Programs as XML Markups

① **Specify the declarative programming language** through an XML document type definition (DTD)

② **Convert** any to-be-published declarative program **from its source syntax to an XML document** according to the DTD

③ **Upload** such an XML document to a Web server for publication

④ Also offer the declarative programs for **server-side querying (e.g. CGI)** and advertise their XML-document version to search engines etc., ideally using **metadata markup (e.g. RDF/XML)**

⑤ **Distribute** these documents to requesting clients **via standard Web protocols (e.g. HTTP)**

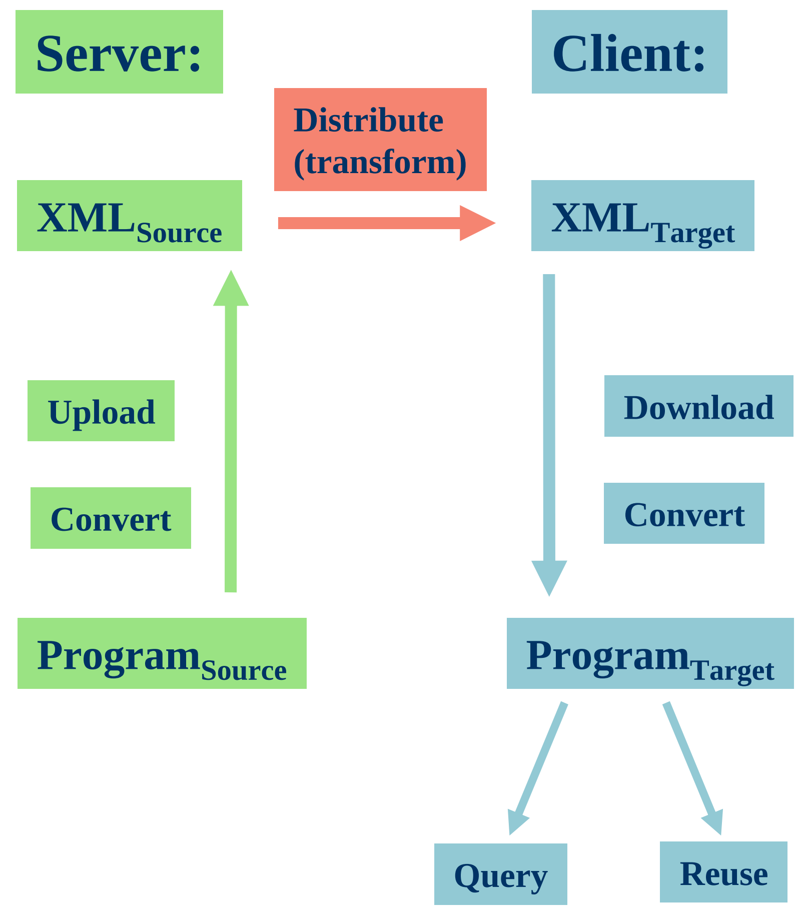
⑥ **If necessary, transform** such an XML document **to** a declarative target language with a **different DTD**, possibly using an (XSLT) stylesheet

⑦ **Download** any requested XML document at the client site

⑧ **Convert** this XML document **to** the client's **target** syntax, possibly using (XSLT + CSS) stylesheets

⑨ **Query** the **target version** via the client's program interpreter and optionally download the server's source-program interpreter (once) for client-side querying, ultimately as a browser plug-in

⑩ **Reuse** the target version, say in existing programs



Note: **Program**$_{\textbf{Source}}$ may be identical to **Program**$_{\textbf{Target}}$

Cross-Fertilizations of XML and Declarative Programming Languages

- Separate vs. joint assertion and query languages:
  - XML: Still separate schema and query of elements
  - DPL: Mostly joint storage and retrieval of clauses
- Generating XML markup from more compact special-purpose notations (and vice versa)
- XML validators and DPL compilers
- XML stylesheets and DPL transformers
- Specification, correctness, and efficiency technology
- Early case study done with the declarative language RFML (Relational-Functional Markup Language)
- Design of Functional RuleML draws on RFML for interchange of declarative programs: http://www.ruleml.org/fun

Basics of the Relational-Functional Markup Language RFML

- Much of Web knowledge constitutes definitions of relations and functions
- Kernel of Relational-Functional language (Relfun) suited for XML knowledge markup:
  - Uniform, rather small language
  - Sufficient expressive power for practical use
- RFML is an XML application for integrated relational-functional information
- Relational (hn) and functional (ft) clauses together define a unified notion of operators
- RFML DTD small and open to various extensions

Relational Facts: From Tables to Prolog

Collect data on consumer behavior in ...

Relational Table:

satisfied(	Customer,	Item,	Price	)
	john	wine	17.95	
	peter	beer	06.40	

Prolog (Ground) Facts:

satisfied(	Customer,	Item,	Price	)
satisfied(	john,	wine,	17.95	).
satisfied(	peter,	beer,	06.40	).

Relational Facts: From Prolog to RFML

Prolog (Ground) Facts:

satisfied(john,wine,17.95).

satisfied(peter,beer,6.40).

RFML (Ground) Markup:

```
<hn>
  <pattop>
    <con>satisfied</con>
    <con>john</con>
    <con>wine</con>
    <con>17.95</con>
  </pattop>
</hn>
```

```
<hn>
  <pattop>
    <con>satisfied</con>
    <con>peter</con>
    <con>beer</con>
    <con>6.40</con>
  </pattop>
</hn>
```

Relational Rules: From Prolog to RFML

Infer data on consumer behavior via ...

Prolog (Non-Ground) Rule:

satisfied(C,I,P) :- buy(week1,C,I,P), buy(week2,C,I,P).

RFML (Non-Ground) Markup:

```
<hn>
 <pattop>
  <con>satisfied</con><var>C</var><var>I</var><var>P</var>
 </pattop>
 <callop>
  <con>buy</con><con>week1</con><var>C</var><var>I</var><var>P</var>
 </callop>
 <callop>
  <con>buy</con><con>week2</con><var>C</var><var>I</var><var>P</var>
 </callop>
</hn>
```

Functional Facts (Definition Points): From Unconditional Equations to RFML

Discriminate on payment method via ...

Unconditional (Ground) Equations:

pay(john,fred,17.95) = cheque

pay(peter,fred,6.40) = cash

RFML (Ground) Markup:

```
<ft>
  <pattop>
    <con>pay</con>
    <con>john</con>
    <con>fred</con>
    <con>17.95</con>
  </pattop>
  <con>cheque</con>
</ft>
```

```
<ft>
  <pattop>
    <con>pay</con>
    <con>peter</con>
    <con>fred</con>
    <con>6.40</con>
  </pattop>
  <con>cash</con>
</ft>
```

RFML

Functional Queries: Joint Assertion and Query Language

The pay function can be queried (non-ground) **directly** via a callop markup:

```
<callop>
   <con>pay</con>
   <con>john</con>
   <var>merchant</var>
   <var>price</var>
</callop>
```

binding the two variables to the corresponding constants in the definition pattern and returning the constant 'cheque'

Same **indirectly** as the right side of a conditional equation ...

Functional Rules: From Conditional Equations to Relfun

Predict consumers' acquisition behavior via ...

Conditional (Non-Ground) Equation:

acquire(Customer,Merchant,Item,Price) =
pay(Customer,Merchant,Price)
if satisfied(Customer,Item,Price)

Relfun (Non-Ground) Footed Rule:

acquire(Customer,Merchant,Item,Price) :-
satisfied(Customer,Item,Price) &
pay(Customer,Merchant,Price).

Functional Rules: From Relfun to RFML

Relfun (Non-Ground) Footed Rule:

acquire(C,M,I,P) :- satisfied(C,I,P) & pay(C,M,P).

RFML (Non-Ground) Markup:

```
<ft>
  <pattop>
    <con>acquire</con><var>c</var><var>m</var><var>i</var><var>p</var>
  </pattop>
  <callop>
    <con>satisfied</con><var>c</var><var>i</var><var>p</var>
  </callop>
  <callop>
    <con>pay</con><var>c</var><var>m</var><var>p</var>
  </callop>
</ft>
```

Relational-Functional Computations: “What Items John Buys, and How”

A query of the acquire function now leads to the following RFML computation (4-step animation):

```
<con item="wine">
  cheque
</con>
```

It binds the variable 'item' to the constant 'wine' (RFML bindings represented as XML attributes) and returns the constant 'cheque'

Relational-Functional Computations: “What Items John Buys, and How”

A query of the acquire function now leads to the following RFML computation (4-step animation):

```
<callop>
  <con>acquire</con>
  <con>john</con>
  <con>fred</con>
  <var>item</var>
  <con>17.95</con>
</callop>
```

It binds the variable 'item' to the constant 'wine' (RFML bindings represented as XML attributes) and returns the constant 'cheque'

Relational-Functional Computations: “What Items John Buys, and How”

A query of the acquire function now leads to the following RFML computation (4-step animation):

```
<callop>
  <con>satisfied</con>
  <con>john</con>
  <var>item</var>
  <con>17.95</con>
</callop>
```

&

```
<callop>
  <con>pay</con>
  <con>john</con>
  <con>fred</con>
  <con>17.95</con>
</callop>
```

It binds the variable 'item' to the constant 'wine' (RFML bindings represented as XML attributes) and returns the constant 'cheque'

Relational-Functional Computations: "What Items John Buys, and How"

A query of the acquire function now leads to the following RFML computation (4-step animation):

<con item="wine">
true
</con>

&

<callop>
<con>pay</con>
<con>john</con>
<con>fred</con>
<con>17.95</con>
</callop>

It binds the variable 'item' to the constant 'wine' (RFML bindings represented as XML attributes) and returns the constant 'cheque'

Relational-Functional Computations: “What Items John Buys, and How”

A query of the acquire function now leads to the following RFML computation (4-step animation):

```
<con item="wine">
   cheque
</con>
```

It binds the variable 'item' to the constant 'wine' (RFML bindings represented as XML attributes) and returns the constant 'cheque'

The RFML DTD (1)

```
<!-- ENTITIES use non-terminals of Relfun grammar (Boley 1999) 'untagged',  -->
<!-- e.g. term ::= con | var | anon | struc | tup, just specifying, say,    -->
<!-- <var> X </var> term instead of nesting <term> <var> X </var> </term>   -->

<!ENTITY % variable      "(var | anon)" >
<!ENTITY % appellative   "(con | %variable; | struc)" >
<!ENTITY % term          "(%appellative; | tup)" >

<!-- ELEMENTS use non-terminals of Relfun grammar 'tagged', so var ::= ...  -->
<!-- itself becomes <var> X </var>                                          -->

<!-- rfml is the document root, the possibly empty knowledge-base top-level -->
<!-- of hn or ft clauses:                                                   -->

<!ELEMENT rfml          (hn | ft)* >

<!-- hn clauses are a pattop before zero (facts) or more terms or callop's; -->
<!-- ft clauses are a pattop before at least one term or callop (the foot): -->

<!ELEMENT hn            (pattop, (%term; | callop)*) >
<!ELEMENT ft            (pattop, (%term; | callop)+) >

<!-- a pattop clause head is an operator appellative and a (rest) pattern:  -->

<!ELEMENT pattop        (%appellative;,
                         (%term;)*,
                         (rest, (%variable; | tup))?) >
```

RFML

The RFML DTD (2)

```
<!-- a callop clause body premise or foot is a (nested) operator call:      -->

<!ELEMENT callop        ((%appellative; | callop),
                         (%term; | callop)*,
                         (rest, (%variable; | tup | callop))?) >

<!-- a struc is a constructor appellative with argument terms (and a rest): -->

<!ELEMENT struc         (%appellative;,
                         (%term;)*,
                         (rest, (%variable; | tup))?) >

<!-- a tup is a list of terms (zero or more), perhaps followed by a rest:   -->

<!ELEMENT tup           ((%term;)*,
                         (rest, (%variable; | tup))?) >

<!-- con and var are just parsed character data (character permutations):   -->

<!ELEMENT con           (#PCDATA)>
<!ELEMENT var           (#PCDATA)>

<!-- anon (Relfun: "_") and rest (Relfun: "|") are always-empty elements:   -->

<!ELEMENT anon          EMPTY >
<!ELEMENT rest          EMPTY >
```

Summary

- RFML combines relational-functional knowledge-representation and declarative-programming languages on the Web
- It has been implemented as a (Web-)output syntax for declarative knowledge bases and computations
- XSLT stylesheets have been developed for
  - rendering RFML in Prolog-like Relfun syntax
  - translating between RFML and RuleML
- Further descriptions, examples, the DTD, and download information are available at http://www.relfun.org/rfml

Source-to-Source (Horizontal) Transformation

Horizon

Chapter 8

What is Source-to-Source (Horizontal) Transformation?

- A Functional-Logic Programming language such as Relfun can be considered to consist of
  - One or two inner kernel(s): Functional or logic kernel
  - Several surrounding shells: List notation, higher-order, …
- The shells can be automatically reduced towards the kernel(s) using techniques of source-to-**source** (horizontal) transformation
- This preprocessing makes the FLP language
  - Easier to understand for various groups of humans
  - Well-prepared for source-to-**instruction** (vertical) compilations into various machine languages
- Some of the key transformation techniques will be introduced here via examples

An Overview of Source-to-Source (Horizontal) Transformation

- We first show how functions can be transformed into a logic kernel language (from FP to LP)
- We then indicate how relations can be transformed into a functional or into a functional-logic language (from LP to FP or to FLP)
- Another kind of transformation (prior to compilation) will replace list notation by cns structures
- These and several further transformations can be executed interactively as commands in Relfun, and most of them are combined by the **horizon** command, also used by the Relfun compiler

Relationalizing Functions: Flattening (Pseudo-Code Syntax)

Functional Program (fully nested):

Definition by Function Nesting

fr-antonym(Mot) = en2fr(en-antonym(fr2en(Mot)))

Functional-Logic Program (partially flattened):

1st Flattening Step: Variable _1

fr-antonym(Mot) **if** _1 = en-antonym(fr2en(Mot)) **then** en2fr(_1)

Functional-Logic Program (fully flattened):

2nd Flattening Step: Variable _2

fr-antonym(Mot) **if** _2 = fr2en(Mot) **and** _1 = en-antonym(_2)
then en2fr(_1)

Relationalizing Functions: Flattening (Relfun Syntax)

Functional Program (fully nested):

Definition by Function Nesting

Command: `flatten`

fr-antonym(Mot) **:&** en2fr(en-antonym(fr2en(Mot))) **.**

Functional-Logic Program (partially flattened):

1st Flattening Step: Variable _1

fr-antonym(Mot) **:-** _1 **.=** en-antonym(fr2en(Mot)) **&** en2fr(_1) **.**

Functional-Logic Program (fully flattened):

2nd Flattening Step: Variable _2

fr-antonym(Mot) **:-** _2 **.=** fr2en(Mot) **,** _1 **.=** en-antonym(_2)
& en2fr(_1) **.**

Relationalizing Functions: Extra-Argument Insertion (Pseudo-Code Syntax)

Functional-Logic Program (results returned):

Flat Definition: Variables _1, _2

fr-antonym(Mot) **if** _2 = fr2en(Mot) **and** _1 = en-antonym(_2)
then en2fr(_1)

Logic Program (results bound):

New 1st Argument: Variable _3

fr-antonym(_3,Mot) **if** fr2en(_2,Mot) **and** en-antonym(_1,_2)
and en2fr(_3,_1)

New 1st Argument:
Variable _2 from '='

...

Call Pattern (query variable):

fr-antonym(Franto,noir)

Relationalizing Functions: Extra-Argument Insertion (Relfun Syntax)

Command: `extrarg`

Functional-Logic Program (results returned):

Flat Definition: Variables _1, _2

fr-antonym(Mot) :- _2 .= fr2en(Mot) , _1 .= en-antonym(_2)
& en2fr(_1) .

Logic Program (results bound):

New 1st Argument: Variable _3

fr-antonym(_3,Mot) :- fr2en(_2,Mot) , en-antonym(_1,_2)
, en2fr(_3,_1) .

New 1st Argument: Variable _2 from '.='

…

Call Pattern (query variable):

fr-antonym(Franto,noir)

Combined Command: `relationalize`

Functionalizing Relations: Footening of Facts (Pseudo-Code Syntax)

Logic Program
(implicit true value):

Fact Definition

spending(Peter Miller,min 5000 euro,previous year)

Functional Program A
(explicit true value):

‘true’-Footening

spending(Peter Miller,min 5000 euro,previous year) = true

Functional Program B
(explicit 1 value):

‘1’-Footening

spending(Peter Miller,min 5000 euro,previous year) = 1

Functionalizing Relations: Footening of Facts (Relfun Syntax)

Logic Program
(implicit true value):

Fact Definition

spending(Peter Miller,min 5000 euro,previous year) **.**

Command: `footer true`

Functional Program A
(explicit true value):

'true'-Footening

spending(Peter Miller,min 5000 euro,previous year) **&** true **.**

Command: `footer 1`

Functional Program B
(explicit 1 value):

'1'-Footening

spending(Peter Miller,min 5000 euro,previous year) **&** 1 **.**

Functionalizing Relations: Footening of Rules (Pseudo-Code Syntax)

Logic Program (implicit true value):

Definition by Single Premise Call

premium(Customer) **if**
spending(Customer,min 5000 euro,previous year)

Functional-Logic Program A (explicit true value):

'true'-Footening

premium(Customer) **if**
spending(Customer,min 5000 euro,previous year) **then** true

Functional-Logic Program B (explicit 1 value):

'1'-Footening

premium(Customer) **if**
spending(Customer,min 5000 euro,previous year) **then** 1

Functionalizing Relations: Footening of Rules (Relfun Syntax)

Logic Program
(implicit true value):

Definition by
Single Premise Call

premium(Customer) **:-**
    spending(Customer,min 5000 euro,previous year) **.**

Functional-Logic Program A
(explicit true value):

Command: `footen true`

'true'-Footening

premium(Customer) **:-**
    spending(Customer,min 5000 euro,previous year) **&** true **.**

Functional-Logic Program B
(explicit 1 value):

Command: `footen 1`

'1'-Footening

premium(Customer) **:-**
    spending(Customer,min 5000 euro,previous year) **&** 1 **.**

Four Variants of Non-Deterministic Even-Number Generation: Definitions

```
% Functional (Numeric):
evenfn() :& 0.
evenfn() :& 1+(1+(evenfn())).

% Relational (Numeric):
evenrn(0).
evenrn(R) :- evenrn(N), R .= 1+(1+(N)).

% Functional (Symbolic):
evenfs() :& 0.
evenfs() :- H .= evenfs() & suc[suc[H]].

% Relational (Symbolic):
evenrs(0).
evenrs(suc[suc[N]]) :- evenrs(N).
```

HORNIFUL

Four Variants of Non-Deterministic Even-Number Generation: Calls

```
rfi-p> evenfn()
0
rfi-p> more
2
rfi-p> more
4
rfi-p> evenrn(Res)
true
Res=0
rfi-p> more
true
Res=2
rfi-p> more
true
Res=4
```

```
rfi-p> evenfs()
0
rfi-p> more
suc[suc[0]]
rfi-p> more
suc[suc[suc[suc[0]]]]
rfi-p> evenrs(Res)
true
Res=0
rfi-p> more
true
Res=suc[suc[0]]
rfi-p> more
true
Res=suc[suc[suc[suc[0]]]]
```

Four Variants of Non-Deterministic Even-Number Generation: Flattened …

% Functional (Numeric):
evenfn() :& 0.
evenfn() :- _2 .= evenfn(), _1 .= 1+(_2) & 1+(_1).

2-Step Flattening

% Relational (Numeric):
evenrn(0).
evenrn(R) :- evenrn(N), _1 .= 1+(N), R .= 1+(_1).

1-Step Flattening

% Functional (Symbolic):
evenfs() :& 0.
evenfs() :- H .= evenfs() & suc[suc[H]].

Unchanged

% Relational (Symbolic):
evenrs(0).
evenrs(suc[suc[N]]) :- evenrs(N).

Unchanged

Four Variants of Non-Deterministic Even-Number Generation: … + Extrarged

(= Relationalized)

```
% Functional (Numeric):
evenfn(0).
evenfn(_3) :- evenfn(_2), _1 .= 1+(_2), _3 .= 1+(_1).
```

Identical (up to variable renaming)

```
% Relational (Numeric):
evenrn(0).
evenrn(R) :- evenrn(N), _1 .= 1+(N), R .= 1+(_1).

% Functional (Symbolic):
evenfs(0).
evenfs(suc[suc[H]]) :- evenfs(H).
```

Identical (up to variable renaming)

```
% Relational (Symbolic):
evenrs(0).
evenrs(suc[suc[N]]) :- evenrs(N).
```

Four Variants of Non-Deterministic Even-Number Generation: Horizoned

% Functional (Numeric):

evenfn() :& 0.

evenfn() :- _2 .= evenfn(), _1 .= 1+(_2) & 1+(_1).

% Relational (Numeric):

'true'-Footening

evenrn(0).

evenrn(R) :- evenrn(N), _1 .= 1+(N), R .= 1+(_1) & true.

% Functional (Symbolic):

evenfs() :& 0.

evenfs() :- H .= evenfs(), _1 .= suc[H] & suc[_1].

% Relational (Symbolic):

Structure Flattening

'true'-Footening

evenrs(0).

evenrs(_1) :- _2 .= suc[N], _1 .= suc[_2], evenrs(N) & true.

Eliminating the N-ary List Notation: Untupping

Examples:	Flat n-ary (external) lists:		Nested n-ary lists:
Ground:	**[u]**	**[rs[1],u]**	**[[u]]**
Non-ground:	**[X|Y]**	**[rs[_],u]**	**[[u|X]|Y]**

Examples:	Flat **cns** (internal) lists:		Nested **cns** lists:
Ground:	**cns[u,nil]**	**cns[rs[1],cns[u,nil]]**	**cns[cns[u,nil],nil]**
Non-ground:	**cns[X,Y]**	**cns[rs[_],cns[u,nil]]**	**cns[cns[u,X],Y]**

ground-test(**[u]**, **[rs[1],u]**, **[[u]]**).

Command: `untup`

ground-test(**cns[u,nil]**, **cns[rs[1],cns[u,nil]]**, **cns[cns[u,nil],nil]**).

non-ground-test(**[X|Y]**,**[rs[_],u]**,**[[u|X]|Y]**).

Command: `untup`

non-ground-test(**cns[X,Y]**,**cns[rs[_],cns[u,nil]]**,**cns[cns[u,X],Y]**).

Deterministic Even-Number Generation: evenfn Source, Untupped, and Horizoned

% Functional (Numeric):

```
evenfn(1) :&  [0].
evenfn(I) :-  >(I,1),
              [H|R] .= evenfn(1-(I)),
              H2 .= 1+(1+(H))  &  [H2,H|R].
```

% Functional (Numeric) – `untup` :

```
evenfn(1) :& cns[0,nil].
evenfn(I) :-  >(I,1),
              cns[H,R] .= evenfn(1-(I)),
              H2 .= 1+(1+(H))  &  cns[H2,cns[H,R]].
```

% Functional (Numeric) – `horizon` (_1=_4=cns[H,R] by normalizer):

```
evenfn(1) :&  cns[0,nil].
evenfn(I) :-  >(I,1),
              _1 .= cns[H,R], _2 .= 1-(I), _1 .= evenfn(_2),
              _3 .= 1+(H), H2 .= 1+(_3), _4 .= cns[H,R] & cns[H2,_4].
```

Summary

- Horizontal transformation techniques were introduced and illustrated via Relfun examples
- Relfun's **horizon** command transforms FP, LP, and FLP source programs into a **flatten**ed (but not **extrarg**ed) form, which also uses **footen true**
- After **untup** for transforming lists to cns structures, **horizon** also flattens all structures much like active nestings, for preparing their efficient indexing
- Other **horizon**tal steps are the replacement of anonymous variables and of active cns calls
- All **horizon**tal results *can still be interpreted*, but subsequent WAM compilation *increases efficiency*