Principles of Functional and Logic Programming
About Knowledge Representation (KR),
Software Specification, and Programming

\[ \text{KR}_{\text{AI}} \approx \text{Specification}_{\text{Software}} \]

When

KR\text{s} / 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.

\[
\text{head} \;::=\; \text{body} \;\&\;\text{foot}.
\]

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

\[
\text{head} = \text{foot} \iff \text{body}
\]
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

Intermediaries

FP Value FLP

... Binding₁ Bindingₙ

Argument₁ Argumentₙ
(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:

Directed operation: Function with one-directional (returned) value flow

Undirected operation: Relation with two-directional (variable) binding flow
Declarative Programs as Data Flow Diagrams: Example – "Addition Agent" (Input)
Declarative Programs as Data Flow Diagrams: Example – “Addition Agent” (Output)

FP:

LP:

\[ \text{add} \]

\[ 3 \quad 4 \]

\[ 7 \]

\[ \text{add} \]

\[ 3 \quad 4 \quad A=7 \]
Declarative Programs as Data Flow Diagrams: Example – “Addition Agent” (I-O Modes)

FP:

LP:

Directed operation: Function

Undirected operation: Relation
Declarative Programs Used for Testing: Example – “Addition Agent” (Input)
Declarative Programs Used for Testing: Example – “Addition Agent” (Output)

FP:

```
3 4
add
```

LP:

```
3 4
success
```

```
7
```
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)

FP:

LP:

```
add
square

FP:

LP:

3 4

R =

A =

3 4
```
Declarative Programs as Data Flow Diagrams: Example – “Square-of-Add Agent” (Thruput)

FP:

square

add

3 4

7

LP:

square

add

R=

A=7

3 4
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:
Rewrite Traces of Unnamed Compound Agent:
square(add(3, 4))
square(7)
49

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

Definitions of Named Compound Agent:
squadd(M, N) &: square(add(M, N)).
squadd(M, N, R) :-
add(M, N, A), square(A, R).

Rewrite Traces of Named Compound Agent:
squadd(3, 4)
49

R=49
Syntax of Basic Declarative Definitions

FP:
Oriented Equation:
\[ \text{head} = \text{foot} \]
written here as
\[ \text{head} :\& \text{foot}. \]
squadd(\(M, N\)) :&
square(\(\text{add}(M, N)\)).

LP:
Implication:
\[ \text{head} \iff \text{body} \]
written as Prolog-like
\[ \text{head} :- \text{body}. \]
squadd(\(M, N, R\)) :-
\(\text{add}(M, N, A)\),
square(\(A, R\)).

FLP:
Conditional Oriented Equation (FP-LP Amalgamation):
\[ \text{head} = \text{foot} \iff \text{body} \]
written as Prolog-extending
\[ \text{head} :- \text{body} \& \text{foot}. \]
squadd(\(M, N\)) :-
\(\text{add}(M, N, A)\) \&
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:

CS 6715 FLP
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)

FP:

success/fail

\[ = \]

In In

Out

add

In In

LP:

success/fail

‘Single-assignment’ primitive used here for equality testing

add

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

FP:

```
add
3 4
5
```

LP:

```
add
3 4 5
```
Declarative Programs Used for Testing: Example – “Addition Agent” (Thru/Output)

FP:

LP:

```
add
```

```
add
```

5 7

3 4 5

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

FP:
\[ \text{fail} \]

LP:
\[ \text{fail} \]

\[ \begin{align*}
\text{FP:} & \quad \text{LP:} \\
\text{fail} & \quad \text{fail} \\
\text{add} & \quad \text{add} \\
3 & \quad 3 \quad\quad 4 & \quad 4 \quad\quad 5 & \quad 5
\end{align*} \]
Declarative Programs Used for Testing: Example – “Addition Agent” (Input)

FP: `add` (3, 4) → 7

LP: `add` (3, 4, 7)
Declarative Programs Used for Testing: Example – “Addition Agent” (Thru/Output)

FP:  

LP:  

Example – “Addition Agent” (Thru/Output)

add

success

3 4

3 4 7
Declarative Programs Used for Testing: Example – “Addition Agent” (Output)

FP:

success

LP:

success

The diagram shows two programs, FP and LP, both labeled with the success status. Each program involves the `add` function with inputs 3, 4, and 7.
Declarative Testing Programs in Symbolic Notation: Example – “Addition Agent”

FP:  

LP:  

Rewrite Traces:

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

7 .:= add(3, 4)  
7 .:= 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 \texttt{length} and its (non-ground, here pattern-valued) functional ‘inverse’ \texttt{shape}, and then as a single logic program \texttt{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

```
length

presuc

length

N
M

[X|Z]

Z

length

0

[]
```

CS 6715 FLP
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.`

**Functional program (‘inverse’)**

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

**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.

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

Functional program (‘inverse’)

Logical program (both directions)

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

Logical 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

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]`.  

- **Shape Definitions**
  - `shape(3) -> [X',X'',X''']`  
  - `shape(2) -> [X'',X''']`  
  - `shape(1) -> [X''']`  
  - `shape(0) -> []`

- **Presuc Definitions**
  - `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).
```

---

**Logic program (auxiliary)**

```
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

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

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:

\[
\text{length(shape}(n)) = n
\]

\[
\text{shape(length}([e_1, e_2, \ldots, e_n])) = [X', X'', \ldots, X'^{n-1}]
\]

Examples – Nestings:

\[
\text{length(shape}(3)) = 3
\]

\[
\text{shape(length}([a,b,c])) = [X', X'', X''']
\]
Generalized Inverse Property of the Functional Programs \text{length} and \text{shape} (II)

General – Nestings Flattened to Conjunctions:

L. = \text{shape}(n) \land \text{length}(L) = n

I. = \text{length}([e_1, e_2, \ldots, e_n]) \land \text{shape}(I) = [X', X'', \ldots, X'^n]

Examples – Nestings Flattened to Conjunctions:

L. = \text{shape}(3) \land \text{length}(L) = 3

I. = \text{length}([a,b,c]) \land \text{shape}(I) = [X', X'', X''']

Most general pattern for lists of length $n$
Generalized Self-Inverse Property of the Logic Program \texttt{shalen}

**General – Conjunctions:**

\[ \text{shalen}(L, n), \text{shalen}(L, I) \quad \text{binds} \quad I = n \]

\[ \text{shalen}([e_1, e_2, \ldots, e_n], I), \text{shalen}(L, I) \quad \text{binds} \]

\[ L = [X', X'', \ldots, X'''] \]

**Examples – Conjunctions:**

\[ \text{shalen}(L, 3), \text{shalen}(L, I) \quad \text{binds} \quad I = 3 \]

\[ \text{shalen}([a, b, c], I), \text{shalen}(L, I) \quad \text{binds} \quad L = [X', X'', X'''] \]
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 (l)

(m,n)-Matrices of Equal Rows:

\[ \text{shalen}(L, n) \& [L, \ldots, L] = \left[ \begin{array}{c} [X', X'', \ldots, X'\ldots'] \\ \vdots \\ [X', X'', \ldots, X'\ldots'] \end{array} \right] \]

(2,3)-Matrices of Equal Rows:

\[ \text{shalen}(L, 3) \& [L, L] = \left[ \begin{array}{c} [X', X'', X'''] \\ [X', X'', X'''] \end{array} \right] \]
Refinement of Non-Ground List Values: Generating Matrix Patterns with shalen (II)

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

$$\text{shalen}(L, n), [C, C|R] = L & [L, ..., L] = \left\{ \left[ X'', X'', ..., X'' \right] \right\}_{m}$$

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

$$\text{shalen}(L, 3), [C, C|R] = L & [L, L] = \left\{ \left[ X'', X'', X''' \right], \left[ X'', X'', X''' \right] \right\}_{m,n}$$
Refinement of Non-Ground List Values: Generating Matrix Patterns with \textit{shalen} (III)

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

\[
\text{shalen}(L,n), [D,A,D|S] = L \land [L, \ldots, L] = \begin{bmatrix} X'', X'', X'', \ldots, X' \ldots' \end{bmatrix}
\]

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

\[
\text{shalen}(L,3), [D,A,D|S] = L \land [L, L] = \begin{bmatrix} X'', X'', X'' \end{bmatrix}, [X'', X'', X''']
\]

\textbf{‘Single-assignment’ primitive used for unification}
Double Refinement of Non-Ground List Values: Generating Matrix Patterns with \texttt{shalen} (IV)

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

\texttt{shalen}(L,n), \[C, C| R \] := L,
\[D, A, D| S \] := L & \{L, ..., L\} = \{[X'', X'', X'', ..., X'...']\}

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

\texttt{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 (variable-free) 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:
\[
\begin{align*}
qsort([ ]) : & \ [].
qsort([X|Y]) : & \ -
\quad \text{partition}(X,Y,Sm,Gr) & \ & \&
\quad \text{cat}(qsort(Sm),\text{tup}(X|qsort(Gr))).
\end{align*}
\]

Rules and Fact:
\[
\begin{align*}
\text{partition}(X,[Y|Z],[Y|Sm],Gr) : & \ -
\quad <(Y,X), \ \text{partition}(X,Z,Sm,Gr).
\text{partition}(X,[Y|Z],Sm,[Y|Gr]) : & \ -
\quad <(X,Y), \ \text{partition}(X,Z,Sm,Gr).
\text{partition}(X,[X|Z],Sm,Gr) : & \ -
\quad \text{partition}(X,Z,Sm,Gr).
\text{partition}(X,[][],[]) : & \ .
\end{align*}
\]

Auxiliary Function (append or catenate):
\[
\begin{align*}
cat([],L) : & \ & \& L.
cat([H|R],L) : & \ & \& \text{tup}(H|\text{cat}(R,L)).
\end{align*}
\]
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 \(\lambda\)-expressions), as they are dominant in practice and more easily integrated (avoids \(\lambda/\)logic-variable distinction and higher-order unification): apply-reducible to 1\(^{st}\) order.

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

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

\[
\text{before}(\{X1,Y1\},\{X2,Y2\}) :- \text{string}<(X1,X2).
\]
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)).

>>>>>> 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]
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

Transform a list of symbols into the list of their lexicographic serial rank numbers
Example: \([d,a,l,l,a,s] \rightarrow [2,1,3,3,1,4]\)

Specific Solution for Example:

\[\text{numbered}(\text{qsort}[\text{before}] (\text{pairlists}([d,a,l,l,a,s],R)), 1)\] & 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) :-

\[\text{numbered}(\text{qsort}[\text{before}] (\text{pairlists}(L,R)), 1)\] & 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]

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

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

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

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

**Database:** PROLOG Syntax

```prolog
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]([],[]) & [].
qusort[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],[Y|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)).
```

**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]

**Query (batch):**

trace pairlists numbered qsort[Cr]

Copy & paste ready

Try again with tracer
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: \text{Ground} \rightsquigarrow \text{non-ground} \Rightarrow \text{Ground}
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 1*st-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

\[ \text{fr-antonym}(\text{Mot}) = \text{en2fr}(\text{en-antonym}(\text{fr2en}(\text{Mot}))) \]

\[ \begin{align*}
\text{en-antonym}(\text{black}) &= \text{white} \\
\text{en-antonym}(\text{white}) &= \text{black} \\
\text{en-antonym}(\text{big}) &= \text{small} \\
\text{en-antonym}(\text{small}) &= \text{big}
\end{align*} \]

\[ \begin{align*}
\text{fr2en}(\text{noir}) &= \text{black} \\
\text{fr2en}(\text{blanc}) &= \text{white} \\
\text{fr2en}(\text{grand}) &= \text{big} \\
\text{fr2en}(\text{petit}) &= \text{small}
\end{align*} \]

\[ \begin{align*}
\text{en2fr}(\text{black}) &= \text{noir} \\
\text{en2fr}(\text{white}) &= \text{blanc} \\
\text{en2fr}(\text{big}) &= \text{grand} \\
\text{en2fr}(\text{small}) &= \text{petit}
\end{align*} \]
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 underlined; results are put in *italics*
Functional Programs: Call-by-Value Computation of Nestings

\[
\text{fr-antonym(noir)} = \text{en2fr(\text{en-antonym}(\text{fr2en(noir)}))} \\
= \text{en2fr( \text{en-antonym}(black) )} \\
= \text{en2fr( white) } \\
= \text{blanc}
\]

\[
\begin{align*}
\text{en-antonym(black)} & = \text{white} \\
\text{en-antonym(white)} & = \text{black} \\
\text{en-antonym(big)} & = \text{small} \\
\text{en-antonym(small)} & = \text{big}
\end{align*}
\]

\[
\begin{align*}
\text{fr2en(noir)} & = \text{black} \\
\text{fr2en(blanc)} & = \text{white} \\
\text{fr2en(grand)} & = \text{big} \\
\text{fr2en(petit)} & = \text{small}
\end{align*}
\]

\[
\begin{align*}
\text{en2fr(black)} & = \text{noir} \\
\text{en2fr(white)} & = \text{blanc} \\
\text{en2fr(big)} & = \text{grand} \\
\text{en2fr(small)} & = \text{petit}
\end{align*}
\]
Functional Computation Example: Web Services

- The function composition $en2fr \circ en\text{-antonym} \circ fr2en$ is pre-specified here by the agent $fr\text{-antonym}$; 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 $fr\text{-antonym}$ 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

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

**Definition by Case Analysis**

bitranslate(X) =

\[
\begin{align*}
\text{en2fr}(X) & \quad \text{if} \quad \text{english}(X) \\
\text{fr2en}(X) & \quad \text{if} \quad \text{french}(X)
\end{align*}
\]

**Pointwise Definitions**

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

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

define fr2en(noir) = black

define fr2en(blanc) = white

define fr2en(grand) = big

define fr2en(petit) = small

define en2fr(black) = noir

define en2fr(white) = blanc

define en2fr(big) = grand

define 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

\[
antonym(X) = \begin{cases} 
  \text{en-antonym}(X) & \text{if } \text{english}(X) \\
  \text{en2fr}(\text{en-antonym}(\text{fr2en}(X))) & \text{if } \text{french}(X) 
\end{cases}
\]

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)))
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 \textit{fr-antonym} as a relation, which is applied to an input argument \texttt{Mot} and binds an output argument \texttt{Franto (French antonym)} via the following conjunction of relation calls:
  - A relation \texttt{fr4en} uses \texttt{Mot}, as input, to bind \texttt{Word}, as output, to the French-to-English translation result
  - A relation \texttt{en-antonym} uses this \texttt{Word}, as input, to bind \texttt{Enanto}, as output, to the antonym-catalog look-up result
  - The relation \texttt{fr4en} now uses \texttt{Enanto}, as input, to bind \texttt{Franto}, as output (also, of \textit{fr-antonym}), to the English-to-French translation result

- The relational \texttt{fr4en} is ‘economically’ accessed in two I/O modes, saving two functions; for the relational \texttt{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) \(\text{if} \) fr4en(Mot,Word) \text{and} en-antonym(Word,Enanto) \text{and} fr4en(Franto,Enanto)

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

\[\ldots\]

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

\[\ldots\]

Variable Bindings

Facts: Base Relations

CS 6715 FLP
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 **underlined**; results are **italicized**
Logic Programs: Left-to-Right Computation of Conjunctions

fr-antonym(noir, Result)

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)

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
Logic Definition Example: “Bidirectional French-English Translator”

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

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

Facts: Base Relation
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 underlined; 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,_)
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

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
- CS 6715 FLP
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*: `bitranslate(pain) = {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: `bitranslate(pain,R) = {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:
   \[ \text{fr2en(noir)} = \text{black}, \text{fr2en(blanc)} = \text{white}, \ldots \]

2) FLP also permits *non-ground* function calls as in:
   \[ \text{fr2en(A)} = \{ \text{black}/A=\text{noir}, \text{white}/A=\text{blanc}, \ldots \} \]

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*:
   \[ \text{fr4en(noir,R)} \text{ if } R=\text{black}, \text{fr4en(blanc,R)} \text{ if } R=\text{white}, \ldots \]

5) The LP relation call equivalent to 2) again is a *non-ground* and *non-deterministic* call:
   \[ \text{fr4en(A,R)} \text{ if } \{ A=\text{noir}/R=\text{black}, A=\text{blanc}/R=\text{white}, \ldots \} \]
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

Chapter 2
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 cons structure)
  - Term
    - Ground (variable free)
    - 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 _
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: \[ c \quad rs \quad duo \quad addr \]

- 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:**
  - Ground: \[ c[1] \quad rs[1] \quad duo[u,i] \]
  - Non-ground: \[ rs[_] \quad duo[X,Y] \]

- **Nested structures:**
  - \[ addr[john,loc[ny,ny]] \]
  - \[ addr[john,loc[X,X]] \]
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 Refun, unification can be explicitly performed via “.=”):

Example:

\[ \text{addr[john,loc[ny,ny]]} \]
\[ \text{addr[john,loc[X,X]]} \]
\[ \text{addr[john,loc[ny,ny]]} .\text{=} \text{addr[john,loc[X,X]]} \]

In an empty environment succeeds, creating the binding \( X=\text{ny} \)  
In an environment with \( X=\text{ny} \) succeeds, creating no new binding  
In an environment with \( X=Y \) succeeds, creating binding \( Y=\text{ny} \)  
In an environment with \( X=Y, Y=Z, \text{and} Z=\text{sf} \) 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 \( X = "X" \))
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:  \( fail \ fail \ fail \ fail \ fail \ fail \ fail \)

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 identify constants with nullary structures and succeed).
Term Unification: Variable and Constant

- 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: \( \text{succ succ succ succ succ succ succ} \)
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: \[X \quad c[\] \quad rs[1] \quad duo[X,Y] \quad duo[X,Y]\]
Term 2: \[c[\] \quad _ \quad Y \quad _9 \quad X\]

Result: \[\text{succ} \quad \text{succ} \quad \text{succ} \quad \text{succ} \quad \text{fail}\]

Bindings: \[X=c[\] \quad Y=rs[1] \quad _9=\text{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 *jDREW*.
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: \( Y \_9 \text{Mot} \_X \)

Result: \( \text{succ succ succ succ succ succ succ succ} \)

Bindings: \( X=Y, \_rs2=\_9, \_9=\text{Mot}, X=\_ \)

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.
Term Unification: Two Structures (I)

- 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: \( c[] \ \text{rs}[1] \ \text{rs}[1] \ \text{rs}[1] \ \text{trio}[1,X,Y] \ \text{trio}[1,X,X] \)

Term 2: \( c[] \ \text{rs}[2] \ \text{jk}[1] \ \text{rs}[Z] \ \text{trio}[1,u,i] \ \text{trio}[1,u,i] \)

Result: \( \text{succ} \ \text{fail} \ \text{fail} \ \text{succ} \ \text{succ} \ \text{fail} \)

Bindings: \( Z=1 \ \ X=u, \ Y=i \)
Term Unification: Two Structures (II)

Examples: Nested structures:

Term 1: \( \text{addr}[\text{john}, \text{loc}[\text{ny}, \text{ny}]] \quad \text{addr}[X, \text{loc}[\text{ny}, \text{ny}]] \)
Term 2: \( \text{addr}[\text{john}, \text{loc}[X, X]] \quad \text{addr}[\text{john}, \text{loc}[X, X]] \)

Result: \( \text{succ} \quad \text{fail} \)

Bindings: \( X=\text{ny} \)
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):
  - `cns[u,nil]`
  - `cns[rs[1],cns[u,nil]]`
  - `cns[cns[u,nil],nil]`

- **Nested lists**:
  - `cns[X,Y]`
  - `cns[rs[_],cns[u,nil]]`
  - `cns[cns[u,X],Y]`
Complex Terms: Lists as cns Trees

Flat lists (cns right-recursive):
- cns[u,nil]
- cns[X,Y]

Nested lists:
- cns[cns[u,nil],nil]
- cns[cns[u,X],Y]

CS 6715 FLP
Complex Terms: N-ary List Notation

The n-ary short notation of lists, for \( n \geq 0 \), can be obtained from lists as \( \text{cns} \) structures as follows:

- The empty list \( \text{nil} \) is rewritten as \([\ ]\), for \( n=0 \)
- A non-empty list \( \text{cns}[e_1, \text{cns}[e_2, \ldots \text{cns}[e_n, t] \ldots]] \), for \( n \geq 1 \), is rewritten as \([e_1', e_2', \ldots, e_n']\), if \( t \) is \( \text{nil} \),
  and is rewritten as \([e_1', e_2', \ldots, e_n' | t]\), if \( t \) is a variable,
  where the primes indicate recursive rewritings.

Examples:

- Flat \( \text{cns} \) (original) lists:
  - Ground: \( \text{cns}[u, \text{nil}] \), \( \text{cns}[\text{rs}[1], \text{cns}[u, \text{nil}]] \), \( \text{cns}[\text{cns}[u, \text{nil}], \text{nil}] \)
  - Non-ground: \( \text{cns}[X, Y] \), \( \text{cns}[\text{rs}[\_], \text{cns}[u, \text{nil}]] \), \( \text{cns}[\text{cns}[u, X], Y] \)

- Nested \( \text{cns} \) lists:
  - Flat \( \text{n-ary} \) (rewritten) lists:
    - Ground: \([u]\), \([\text{rs}[1], u]\)
    - Non-ground: \([X|Y]\), \([\text{rs}[\_], u]\)
  - Nested \( \text{n-ary} \) lists:
    - \([[u]]\)
    - \([[[u]|X]|Y]\)
Complex Terms: N-ary Tree Notation (I)

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

\[
\left[ e_1', e_2', \ldots, e_n' \right]
\]
Complex Terms: N-ary Tree Notation (II)

Flat lists (\texttt{cns} right-recursive):
- \texttt{cns[u,nil]}
- \texttt{cns[rs[1],cns[u,nil]]}

Nested lists:
- \texttt{cns[cns[u,nil],nil]}

Flat n-ary (rewritten) lists:
- \texttt{[u]}
- \texttt{[rs[1],u]}

Nested n-ary lists:
- \texttt{[[u]]}
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:

Term 1:
- Flat cns (original) lists:
  - cns[u,nil]  cns[rs[1],cns[u,nil]]  cns[cns[u,nil],nil]
- Nested cns lists:
  - cns[X,Y]  cns[rs[_,cns[u,nil]]]  cns[cns[u,Y],Z]

Term 2:
- Flat n-ary (rewritten) lists:
  - [u]  [rs[1],u]
- Nested n-ary lists:
  - [[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 $X=\text{suc}[_]$:  

Example:  
Named variable in structure (in list):  
Term 1: $\left[ \text{suc}[N], \, \_ \_ \_ \text{suc}[0], \, \_ \_ \_ \text{suc}[1] \right]$  
Term 2: $\left[ X, \rightarrow X, \rightarrow X \right]$  

Example:  
Anonymous variable in structure (in list):  
Term 1: $\left[ \text{suc}[\_], \, \_ \_ \_ \text{suc}[0], \, \_ \_ \_ \text{suc}[1] \right]$  
Term 2: $\left[ X, \rightarrow X, \rightarrow X \right]$  

Result: $\text{fail}$

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 cons structures.
- List unification with one segment variable per (sub)list was discussed as a notational variant.
Chapter 3
Clauses as the Smallest Functional and Logic Definition Units

- An operation \((\text{name})\) is a function or relation \((\text{name})\).
- A clause associates a \textit{head} of an operation name and argument terms with an optional \textit{body} of a (non-)ground, (non-)deterministic call conjunction and an optional \textit{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 \textit{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

:-. 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: SUCG (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: `SUCCE (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: SUC (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:
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".
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: SUC (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

**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:**

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

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

```prolog
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: SUC (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
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:

\[
\text{premium(Customer) if} \\
\text{min 5000 euro} = \text{spending(Customer,previous year)} \\
\text{then true}
\]

Relfun Function Definition with a true-Valued Conditional Equation:

\[
\text{premium(Customer) :-} \\
"\text{min 5000 euro}" .\text{= spending(Customer,"previous year")} \\
\text{& true.}
\]
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:

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

Relfun Function Definition with a bonus-Valued Conditional Equation:

```plaintext
premium(Customer) :-
  "min 5000 euro" .=" spending(Customer,"previous year")
& bonus.
```
Logic Clauses: 2\textsuperscript{nd} 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:

\begin{verbatim}
discount(Customer,Product,5.0 percent)  if
   premium(Customer)  and  regular(Product)
\end{verbatim}

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

\begin{verbatim}
discount(Customer,Product,"5.0 percent") :-
   premium(Customer) , regular(Product).
\end{verbatim}
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: **SUC** (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")`. 
Functional Clauses: 2\textsuperscript{nd} 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:

\[
\text{discount(Customer,Product) if } \text{premium(Customer) and regular(Product) then "5.0 percent"}
\]

Relfun Function Definition with a Conditional Equation:

\[
\text{discount(Customer,Product) :- premium(Customer), regular(Product) & "5.0 percent".}
\]
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: \textit{Succ} (Whether: yes)

Bindings: Customer="Peter Miller" (How: Input Customer)
Product ="Honda" (How: Input Product)
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").

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.

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").

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

FLP combines directedness with invertibility to also permit Product queries
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

Chapter 4
FP: A Tail-Recursive Natural-Number Addition Function (I)

For $M>0$, this is a recursion (here: loop) invariant of $\text{add}$:

$$\text{add}(M,N) = \text{add}(M-1,N+1)$$

Notation:

$$\text{add}(M,N) :& \text{add}(1-(M),1+(N)).$$
Un/Conditional Equations with Recursive Call as a Foot (Tail-Recursion):

\[
\text{add}(0,N) : \& N.
\]

\[
\text{add}(M,N) :- \text{>(M,0)} \& \text{add}(1-(M),1+(N)).
\]

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

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

- \( \text{add}(3,4) \)
- \( \text{add}(2,5) \)
- \( \text{add}(1,6) \)
- \( \text{add}(0,7) \)

\[ 7 \]
LP: A Tail-Recursive Natural-Number Addition Relation (I)

For \( M > 0 \), this is a recursion (here: loop) invariant of \( \text{add} \):

\[
M + N = R \quad \text{if} \quad M - 1 + N + 1 = R
\]
\[
\text{add}(M, N, R) \quad \text{if} \quad \text{add}(M - 1, N + 1, R)
\]

Notation:

\[
\text{add}(M, N, R) \quad :- \quad P \, = \, 1 - (M), \quad S \, = \, 1 + (N), \quad \text{add}(P, S, R).
\]
Datalog Rule with Recursive Call as a Last Premise (Tail-Recursion):

\[
\text{add}(0,N,N).
\]

\[
\text{add}(M,N,R) : - \text{>(}M,0\text{), } P = 1-(M), \ S = 1+(N), \ \text{add}(P,S,R).
\]

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

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

- \(\text{add}(3,4,A)\)
- \(\text{add}(2,5,R1)\)
- \(\text{add}(1,6,R2)\)
- \(\text{add}(0,7,7)\)

\(A=R1=R2=7\)

Since built-ins must be called with ground arguments (here: fixed \(M\) and \(N\)), inverse calls like \(\text{add}(3,W,7)\), \(\text{add}(V,4,7)\), or \(\text{add}(V,W,7)\) are not permitted!
FLP: A Tail-Recursive Natural-Number Addition Relation

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

\[
\text{add}(0,N,N).
\]

\[
\text{add}(M,N,R) :- >\text{(M,0)}, \text{add}(1-(M),1+(N),R).
\]

Based on Built-ins: \(>\) Greater, \(1-\) Predecessor, \(1+\) Successor

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

\[
\text{add}(3,4,A)
\]
\[
\text{add}(2,5,R1)
\]
\[
\text{add}(1,6,R2)
\]
\[
\text{add}(0,7,7)
\]
\[
A=R1=R2=7
\]

Termination: As Before

Recursion: Over Nestings

Again, this add cannot be inverted for subtraction etc.!
FP: A Tail-Recursive Successor-Arithmetic Addition Function (I)

For \( M \geq 0 \), this is a recursion (here: loop) invariant of \( \text{add} \):

\[
\text{add}(M+1,N) = \text{add}(M,N+1)
\]

Notation:

\[
\text{add}(\text{suc}[M],N) :\& \text{ add}(M,\text{suc}[N]).
\]
FP: A Tail-Recursive Successor-Arithmetic Addition Function (II)

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

\[
\text{add}(0, N) : \& N.
\]

\[
\text{add}(\text{suc}[M], N) : \& \text{add}(M, \text{suc}[N]).
\]

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

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

\[
\text{add}(\text{suc}[\text{suc}[\text{suc}[0]]], \text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]])
\]
\[
\text{add}(\text{suc}[\text{suc}[0]], \text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]])
\]
\[
\text{add}(\text{suc}[0], \text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]]])
\]
\[
\text{add}(0, \text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]]])
\]
\[
\text{\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]]]]}
\]

Base Case: Termination

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

For $M \geq 0$, this is a recursion (here: loop) invariant of $\text{add}$:

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

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

Notation:

$$\text{add}(\text{suc}[M],N,R) \quad :) \quad \text{add}(M,\text{suc}[N],R).$$
LP: A Tail-Recursive Successor-Arithmetic Addition Relation (II)

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

\[
\text{add}(0,N,N).
\]

\[
\text{add}(\text{suc}[M],N,R) :\text{ add}(M,\text{suc}[N],R).
\]

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

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

\[
\begin{align*}
\text{add}(\text{suc}[\text{suc}[\text{suc}[0]]],\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]],A) \\
\text{add}(\text{suc}[\text{suc}[0]],\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]],R1) \\
\text{add}(\text{suc}[0],\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]],R2) \\
\text{add}(0,\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]]]], \\
\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]]]]) \\
A=R1=R2=\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]]]])
\end{align*}
\]
Additions like $3 + 4 = A$ can be inverted for subtraction:

$3 + W = 7 \text{ or } W = 7 - 3$

\[ \text{add(suc[suc[suc[0]]],W,suc[suc[suc[suc[suc[suc[suc[0]]]]]]]])} \]

\[ W = \text{suc[suc[suc[0]]]} \]

$V + 4 = 7 \text{ or } V = 7 - 4$

\[ \text{add(V,suc[suc[suc[0]]],suc[suc[suc[suc[suc[suc[suc[0]]]]]]]])} \]

\[ V = \text{suc[suc[suc[0]]]} \]
Can also be inverted for non-deterministic partitioning:

\[ V + W = 7 \]

\[ \text{add}(V, W, \text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]]])) \]

\[ V=0, \ W=\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]]]) \]
\[ V=\text{suc}[0], \ W=\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]]]) \]
\[ \ldots \]
\[ V=\text{suc}[\text{suc}[\text{suc}[0]]], \ W=\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]) \]
\[ \ldots \]
\[ V=\text{suc}[\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]]], \ W=0 \]
LP: An Equivalent Successor-Arithmetic Addition Relation (I)

For $M \geq 0$, this was the recursion (here: loop) invariant of \textit{add}:

\[
M + 1 + N = R \quad \text{if} \quad M + N + 1 = R \\
\text{add}(M+1,N,R) \quad \text{if} \quad \text{add}(M,N+1,R)
\]

For $M \geq 0$, this is the equivalent ($R+1 = R$) invariant of new \textit{add}:

\[
M + 1 + N = R + 1 \quad \text{if} \quad M + N = R \\
\text{add}(M+1,N,R+1) \quad \text{if} \quad \text{add}(M,N,R)
\]

Notation:

\[
\text{add}(\text{suc}[M],N,\text{suc}[R]) :- \text{add}(M,N,R)
\]
LP: An Equivalent Successor-Arithmetic Addition Relation (II)

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

\[
\text{add}(0,N,N).
\]

\[
\text{add}(\text{suc}[M],N,\text{suc}[R]) :- \text{add}(M,N,R).
\]

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

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

\[
\text{add}(\text{suc}[\text{suc}[\text{suc}[0]]],\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]],A) \\
\text{add}(\text{suc}[\text{suc}[0]],\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]],R1) \text{ bind: } A=\text{suc}[R1] \\
\text{add}(0,\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]],R2) \text{ bind: } R1=\text{suc}[R2] \\
\text{add}(0,\text{suc}[\text{suc}[\text{suc}[\text{suc}[0]]]],R3) \text{ bind: } R2=\text{suc}[R3] \\
A=\text{suc}[ \text{suc}[ \text{suc}[ \text{suc}[\text{suc}[0]]]] ] ]
\]
FP: A Tail-Recursive Float-Number Compound Interest Function

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

\[
\text{compint}(0, I, C) : \& \ C. \ % \ T: \text{Time, } I: \text{Interest, } C: \text{Capital}
\]

\[
\text{compint}(T, I, C) :- (T, 0) \& \text{compint}(1-(T), I, +(C, *(C, I))).
\]

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

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

\[
\text{compint}(3, 0.1, 100)
\]
\[
\text{compint}(2, 0.1, 110.0)
\]
\[
\text{compint}(1, 0.1, 121.0)
\]
\[
\text{compint}(0, 0.1, 133.1)
\]

133.1
**LP: A Tail-Recursive Float-Number Compound Interest Relation**

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

```prolog
compint(0,I,C,C). % T: Time, I: Interest, C: Capital, R: Result
compint(T,I,C,R) :- >(T,0), S .= 1-(T), D .= +(C,*(C,I)),
                     compint(S,I,D,R).
```

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

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

```prolog
compint(3,0.1,100,A)
compint(2,0.1,110.0,R1)
compint(1,0.1,121.0,R2)
compint(0,0.1,133.1,133.1)
A=R1=R2=133.1
```

Termination

Recursion

General Case: Recursion

Base Case: Termination
FLP: A Tail-Recursive Float-Number Compound Interest Relation

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

\[
\text{compint}(0,I,C,C). \quad \% \text{T: Time, I: Interest, C: Capital, R: Result}
\]
\[
\text{compint}(T,I,C,R) :- \quad >(T,0),
\]
\[
\quad \text{compint}(1-(T),I,+(C,\times(C,I)),R).
\]

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

\[
\text{compint}(3,0.1,100,A)
\]
\[
\text{compint}(2,0.1,110.0,R1)
\]
\[
\text{compint}(1,0.1,121.0,R2)
\]
\[
\text{compint}(0,0.1,133.1,133.1)
\]
\[
A=R1=R2=133.1
\]

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

Recursion: Over Nestings

Termination

General Case: Recursion

Base Case: Termination

CS 6715 FLP
FLP and ‘while’ Program: A Tail-Recursive and an Iterative Interest Relation

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

\[
\text{compint}(T,I,C,R) :- \ > (T,0), \\
\text{compint}(1-(T),I,+(C,*(C,I)),R).
\]

\[
\text{compint}(0,I,C,C). \quad \% \ T: \text{Time}, \ I: \text{Interest}, \ C: \text{Capital}, \ R: \text{Result}
\]

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

\[
\text{define} \ \text{compint}(T,I,C,R) \ \text{as} \\
\begin{align*}
\text{begin} & \\
& \text{while} \ >(T,0) \ \text{do} \\
& \quad \text{begin} \ T := 1-(T); \ C := +(C,*(C,I)) \ \text{end}; \\
& \quad \text{if} \ =(T,0) \ \text{then} \ R := C \\
\text{end}
\end{align*}
\]

Recursion: Over Nestings
Iteration: Over Nestings
Result Assignment after Termination
Instantiating \texttt{cns} Structures and the N-ary List Notation

Structures with constructor \texttt{cns} were introduced in the ‘Terms’ chapter:
\[
\text{cns}[a,\text{nil}] \quad \text{cns}[a,\text{cns}[7,\text{nil}]] \quad \text{cns}[\text{First},\text{Rest}]
\]

They have been shortened via the N-ary list notation:
\[
[a] \quad [a,7] \quad [\text{First}|\text{Rest}]
\]

Variables as elements of (\texttt{cns}) structures are instantiated:
\[
X \; .\; = \; a \quad Y \; .\; = \; \text{add}(3,4) \quad \text{First} \; .\; = \; 1, \; \text{Rest} \; .\; = \; \text{nil} \\
& \text{cns}[X,\text{nil}] \quad & \text{cns}[a,\text{cns}[Y,\text{nil}]] \quad & \text{cns}[\text{First},\text{Rest}]
\]

\[
\text{cns}[a,\text{nil}] \quad \text{cns}[a,\text{cns}[7,\text{nil}]] \quad \text{cns}[1,\text{nil}]
\]

Variables as elements of the N-ary list notation are likewise instantiated:
\[
X \; .\; = \; a \quad Y \; .\; = \; \text{add}(3,4) \quad \text{First} \; .\; = \; 1, \; \text{Rest} \; .\; = \; \text{nil} \\
& \text{[X]} \quad & \text{[a,Y]} \quad & \text{[First|Rest]}
\]

\[
[a] \quad [a,7] \quad [1]
\]
The \textit{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):

\begin{align*}
\text{cns}[a,\text{cns}[\text{add}(3,4),\text{nil}]] & \quad [a,\text{add}(3,4)] \\
\end{align*}

\textit{“No (active) round parentheses inside [passive] square brackets”}

However, besides the \textit{constructor} \texttt{cns}, also a \textit{function} \texttt{cns} can be defined in either of the following ways (acting like Lisp’s built-in function \texttt{cons}):

\begin{align*}
\text{cns}(\text{First},\text{Rest}) &: \& \text{cns}[\text{First},\text{Rest}]. \quad \text{cns}(\text{First},\text{Rest}) &: \& [\text{First}|\text{Rest}]. \\
\end{align*}

Actual \texttt{cns} arguments are evaluated to elements of \texttt{cns} structures or lists:

\begin{align*}
\text{cns}(a,\text{cns}(\text{\underline{add}(3,4)},\text{nil})) & \quad \text{cns}[a,\text{cns}[7,\text{nil}]] \quad [a,7] \\
\end{align*}
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):

\[
[F|R] \oplus L = \text{cns}(F, R \oplus L) \\
\text{cat}([F|R],L) = \text{cns}(F,\text{cat}(R,L))
\]

Notation:

\[
\text{cat}([F|R],L) \& \text{cns}(F,\text{cat}(R,L)).
\]
FP: A Recursive List-Concatenation Function (II)

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

\[ \text{cat}([], L) : \& L. \]
\[ \text{cat}([F|R], L) : \& \text{cns}(F, \text{cat}(R, L)). \]

Base Case: Termination
General Case: Recursion

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

Full-Recursive Computation Grows and Shrinks an Activation Stack:

\[ \text{cat}([a,b], [c,d,e]) \]
\[ \text{cns}(a, \text{cat}([b], [c,d,e])) \]
\[ \text{cns}(a, \text{cns}(b, \text{cat}([], [c,d,e]))) \]
\[ \text{cns}(a, \text{cns}(b, [c,d,e])) \]
\[ \text{cns}(a, [b,c,d,e]) \]
\[ [a,b,c,d,e] \]

Base Case: Termination
General Case: Recursion
LP: A Tail-Recursive List-Concatenation Relation (I)

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

\[
[F|R] \oplus L = [F|S] \quad \text{if} \quad R \oplus L = S
\]
\[
\text{cat}([F|R], L, [F|S]) \quad \text{if} \quad \text{cat}(R, L, S)
\]

Note analogy to the previous ‘new add’:
\[
\text{add}(1+M, N, 1+R) \quad \text{if} \quad \text{add}(M, N, R)
\]

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

Notation:
\[
\text{cat}([F|R], L, [F|S]) \quad :- \quad \text{cat}(R, L, S).
\]
LP: A Tail-Recursive List-Concatenation Relation (II)

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

\[
\text{cat}([], L, L).
\]

\[
\text{cat}([F|R], L, [F|S]) \quad :\quad \text{cat}(R, L, S).
\]

No Built-ins Required

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

\[
\text{cat}([a,b], [c,d,e], A)
\]

\[
\text{cat}([b], [c,d,e], S1) \quad \text{bind: } A = [a|S1]
\]

\[
\text{cat}([], [c,d,e], S2) \quad \text{bind: } S1 = [b|S2]
\]

\[
A = [a|S1] = [a|[b|S2]] = [a|[b|[c,d,e]]] = [a,b,c,d,e]
\]
Catenations can be inverted for list ‘subtraction’:

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

\[\text{cat}([a,b], W, [a,b,c,d,e])\]

\[W = [c,d,e]\]

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

\[\text{cat}(V, [c,d,e], [a,b,c,d,e])\]

\[V = [a,b]\]
LP: A Tail-Recursive List-Concatenation Relation (IV)

Can also be inverted for non-deterministic partitioning:

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

\[ \text{cat}(V,W,[a,b,c,d,e]) \]

\[
\begin{align*}
V &= [], \quad W = [a,b,c,d,e] \\
V &= [a], \quad W = [b,c,d,e] \\
V &= [a,b], \quad W = [c,d,e] \\
V &= [a,b,c], \quad W = [d,e] \\
V &= [a,b,c,d], \quad W = [e] \\
V &= [a,b,c,d,e], \quad W = []
\end{align*}
\]
FP: A Recursive List-Reversal Function (I)

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

\[
\begin{align*}
\text{rev}([F|R]) & = \text{rev}(R) \oplus [F] \\
\text{rev}([F|R]) & = \text{cat}(\text{rev}(R),[F])
\end{align*}
\]

Notation:

\[
\text{rev}([F|R]) :\& \text{cat}(\text{rev}(R),[F]).
\]
FP: A Recursive List-Reversal Function (II)

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

\[ \text{rev}([]) :& [] \]  

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

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

Full-Recursive Computation Grows and Shrinks an Activation Stack:

\[ \text{rev}([a,b,c]) \]
\[ \text{cat}( \text{rev}([b,c]), [a]) \]
\[ \text{cat}( \text{cat}( \text{rev}([c]), [b]), [a]) \]
\[ \text{cat}( \text{cat}( \text{cat}( [], [c]), [b]), [a]) \]
\[ \ldots \]
\[ [c,b,a] \]
For first argument ≠ nil, this is a recursion invariant of rev:

\[
\begin{align*}
\text{rev}([F|R]) &= L \quad \text{if} \quad \text{rev}(R,K) \text{ and } K \oplus [F] = L \\
\text{rev}([F|R],L) &= \text{rev}(R,K) \text{ and } \text{cat}(K,[F],L)
\end{align*}
\]

Notation:

\[
\text{rev}([F|R],L) \quad :\!-\!\quad \text{rev}(R,K) \text{, cat}(K,[F],L).
\]
LP: A Recursive List-Reversal Relation (II)

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

\[ \text{rev([],[]).} \]
\[ \text{rev([F|R],L) : - rev(R,K) , cat(K,[F],L).} \]

No Built-ins Required; \text{cat} is our user-defined auxiliary

Full-Recursive Computation Grows and Shrinks an Activation Stack:

\[ \text{rev([a,b,c],A)} \]
\[ \text{rev([b,c],K1), cat(K1,[a],L1) bind: A=L1} \]
\[ \text{rev([c],K2), cat(K2,[b],L2), cat(K1,[a],L1) bind: K1=L2} \]
\[ \text{rev([],K3), cat(K3,[c],L3), cat(K2,[b],K1), cat(K1,[a],L1) bind: K2=L3} \]
\[ \text{cat([],K2), cat(K2,[b],K1), cat(K1,[a],L1)} \]
\[ \ldots \]
\[ \text{A=L1}=[[\text{c,b,a}]] \]
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)

Chapter 5
Higher-Order Operations: Operations as 1\textsuperscript{st}-Class Citizens

In \textit{higher-order operations}, operations (functions and relations) are \textbf{1\textsuperscript{st}-class citizens} in that they \textbf{can} themselves be

- Passed to calls as (actual) parameters/arguments
- \textbf{Delivered} from operation calls:
  - Returned as values of function calls
  - Assigned to request variables of relation calls
- Used as elements of structures (and of lists)
- \textbf{Assigned} to local variables (single-assignment)
Taxonomy of 1\textsuperscript{st}-Order and Higher-Order Operations

- **Operation**
  - Function (FP)
    - 1\textsuperscript{st}-Order
      (no functions as arguments or values)
    - Higher-Order
      (functions as arguments or values)
  - Function (FLP)
    - 1\textsuperscript{st}-Order
      (no operations as arguments, values or bindings; no operation variables)
    - Higher-Order
      (operations as arguments, values or bindings; operation variables)
- **Relation (LP)**
  - 1\textsuperscript{st}-Order
    (no relations as arguments or bindings; no relation variables)
  - Higher-Order
    (relations as arguments or bindings; relation variables)
  - 2\textsuperscript{nd}-Order
    (used relations are themselves 1\textsuperscript{st}-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 $\lambda$-variables for defining new anonymous functions).
- Hence ‘$\circ$’ 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:

  - $\text{en-antonym} \circ \text{fr2en}$ becomes compose[en-antonym,fr2en]
  - $\text{en2fr} \circ \text{en-antonym} \circ \text{fr2en}$ becomes compose[en2fr,compose[en-antonym,fr2en]] or compose[compose[en2fr,compose[en-antonym,fr2en]],fr2en]
Such a higher-order function structure can be applied to arguments as follows:

\[ \text{en-antonym} \circ \text{fr2en}(\text{noir}) \] becomes
\[ \text{compose}[\text{en-antonym}, \text{fr2en}](\text{noir}) \]
returning **white**

\[ \text{en2fr} \circ \text{en-antonym} \circ \text{fr2en}(\text{noir}) \] becomes
\[ \text{compose}[\text{en2fr}, \text{compose}[\text{en-antonym}, \text{fr2en}]](\text{noir}) \] or
\[ \text{compose}[\text{compose}[\text{en2fr}, \text{en-antonym}], \text{fr2en}](\text{noir}) \]
returning **blanc**
The higher-order operation \texttt{compose} can be defined as follows, where \texttt{F} and \texttt{G} are \textit{function variables} (their values should be function names or terms), while \texttt{X} is an \textit{object variable} (its values should be normal terms):

\textbf{Math:} \quad \texttt{compose}(F,G)(X) = F(G(X))

\textbf{Relfun:} \quad \texttt{compose}[F,G](X) :\& F(G(X)).
FP: Computation with Simple Compose as a Higher-Order Function

\[
\text{compose[en-antonym,fr2en]}( \text{noir}) \\
\text{en-antonym( fr2en( noir) )} \\
\text{en-antonym( black) } \\
\text{white}
\]
FP: Computation with Nested Compose as a Higher-Order Function

\[
\text{compose[en2fr, compose[en-antonym, fr2en]]( noir )}
\]
\[
\text{en2fr( compose[en-antonym, fr2en]( noir ) )}
\]
\[
\text{en2fr( en-antonym( fr2en( noir ) ) )}
\]
\[
\text{en2fr( en-antonym( black ) )}
\]
\[
\text{en2fr( white )}
\]
\[
\text{blanc}
\]

\[
\text{compose.compose[en2fr, en-antonym, fr2en] ( noir )}
\]
\[
\text{compose[en2fr, en-antonym]( fr2en( noir ) )}
\]
\[
\text{compose[en2fr, en-antonym]( black )}
\]
\[
\text{en2fr( en-antonym( black ) )}
\]
\[
\text{en2fr( white )}
\]
\[
\text{blanc}
\]
The relation \texttt{fr-antonym} of the introductory chapter can be viewed as constituting a relational product \texttt{fr4en \textbullet en-antonym \textbullet en4fr}, where \texttt{en4fr} inverts \texttt{fr4en}: 
\[
\text{en4fr}(\text{En,Fr}) ::= \text{fr4en}(\text{Fr,En}).
\]

The ‘\textbullet’ can be regarded as the infix version of an (associative) binary \texttt{product} higher-order operation:

\[
\text{fr4en \textbullet en-antonym} \quad \text{becomes} \quad \text{product(fr4en, en-antonym)}
\]

\[
\text{fr4en \textbullet en-antonym \textbullet en4fr} \quad \text{becomes} \quad \text{product(fr4en, product(en-antonym, en4fr))} \quad \text{or} \quad \text{product(product(fr4en, en-antonym), en4fr)}
\]
However, we want to use simple definitions of pure higher-order relations (again avoiding $\lambda$-variables)

Hence ‘$\cdot$’ 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:

\[ \text{fr4en}\cdot \text{en-antonym}(\text{noir}, \text{Res}) \] becomes
\[ \text{product}[\text{fr4en}, \text{en-antonym}](\text{noir}, \text{Res}) \]

binding \( \text{Res} = \text{white} \)

\[ \text{fr4en}\cdot \text{en-antonym}\cdot \text{en4fr}(\text{noir}, \text{Res}) \] becomes
\[ \text{product}[\text{fr4en}, \text{product}[\text{en-antonym}, \text{en4fr}]](\text{noir}, \text{Res}) \text{ or } \]
\[ \text{product}[\text{product}[\text{fr4en}, \text{en-antonym}], \text{en4fr}](\text{noir}, \text{Res}) \]

binding \( \text{Res} = \text{blanc} \)
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:** \[ \text{product}(R,S)(X,Z) \text{ if } R(X,Y) \text{ and } S(Y,Z) \]

**Relfun:** \[ \text{product}[R,S](X,Z) \text{ :- } R(X,Y), S(Y,Z). \]
LP: Computation with Simple Product as a Higher-Order Relation

\[
\text{product[fr4en, en-antonym]}(\text{noir, Res}) \\
\text{fr4en(noir, Y1), en-antonym(Y1, Res)} \\
\text{en-antonym(black, Res)} \\
\text{Res = white}
\]
LP: Computation with Nested Product as a Higher-Order Relation

\[
\]

\[
\text{product[product[fr4en, en-antonym], en4fr](noir, Res) product[fr4en, en-antonym](noir, Y1), en4fr(Y1, Res) fr4en(noir, Y2), en-antonym(Y2, Y1), en4fr(Y1, Res) en-antonym(black, Y1), en4fr(Y1, Res) en4fr(white, Res) Res = blanc}
\]
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., \((\text{mapcar} #\text{'sqrt} '(1 4 9))\) 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 \( \text{compose} \): \( a2a[\text{compose}[sqrt,1+]]([0,3,8]) \) 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. \( \text{dup}(N,[N,N]) \).
- Now, e.g. \( a2a[\text{dup}](\{1,4,9\},\text{Res}) \) maps the relation \( \text{dup} \) over 1, 4, and 9, binding \( \text{Res} = \{[1,1],[4,4],[9,9]\} \).
- The mapped list may be non-ground, as in \( a2a[\text{dup}](\{1,J,9\},\text{Res}) \), giving \( \text{Res} = \{[1,1],[J,J],[9,9]\} \).
- 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.
Consider an RDF-like binary fact base describing individuals or resources in the first argument, e.g.:

\[
\text{transmission("Honda","Automatic").}
\]
\[
\text{air-conditioning("Honda","Automatic").}
\]
\[
\text{color("Honda","Eternal Blue Pearl").}
\]

1\textsuperscript{st}-order queries – relation given, object asked:

\[
\text{transmission("Honda",Kind)}
\]

binds object variable \( \text{Kind = "Automatic"} \)

2\textsuperscript{nd}-order queries – objects given, relation asked:

\[
\text{Feature("Honda","Automatic")}
\]

binds relation variable \( \text{Feature = transmission} \)

and then binds \( \text{Feature = air-conditioning} \)
LP 2\textsuperscript{nd}-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 \texttt{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 1\textsuperscript{st}-order queries:
apply(Feature,"Honda","Automatic")
binds object variable Feature = transmission
and then binds Feature = air-conditioning
FLP: Function Variables as 2\textsuperscript{nd}-Order Syntactic Sugar (I)

- Similarly, consider the unary point base describing individuals or resources in the single argument, e.g.:
  
  \begin{verbatim}
  transmission("Honda") :& "Automatic".
  air-conditioning("Honda") :& "Automatic".
  color("Honda") :& "Eternal Blue Pearl".
  \end{verbatim}

- 1\textsuperscript{st}-order queries – function given, object asked:
  
  \begin{verbatim}
  transmission("Honda")
  returns object "Automatic"
  \end{verbatim}

- 2\textsuperscript{nd}-order queries – objects given, function asked:
  
  \begin{verbatim}
  "Automatic" .= Feature("Honda")
  binds function variable Feature = transmission
  and then binds Feature = air-conditioning
  \end{verbatim}
FLP 2\textsuperscript{nd}-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 \texttt{apply} shifts the original function into the first argument position, e.g.:

\begin{verbatim}
apply(transmission,"Honda") :& "Automatic".
apply(air-conditioning,"Honda") :& "Automatic".
apply(color,"Honda") :& "Eternal Blue Pearl".
\end{verbatim}

This leaves us with only 1\textsuperscript{st}-order queries:

\begin{verbatim}
"Automatic" = apply(Feature,"Honda")
\end{verbatim}
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

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 (ground) call:
offer(desk,15,moebureau) succeeds, returning true

Deterministic (ground) call:
offer(chair,20,moebureau) 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 (ground) call:
moebureau . = offer(desk,15)
succeeds, returning moebureau

Deterministic (ground) call:
moebureau . = offer(chair,20)
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 (non-ground) call:
offer(desk,15,Merchant) binds Merchant to moebureau

Deterministic (non-ground) call:
offer(chair,20,Merchant) binds Merchant to moebureau

Deterministic Definition

Which merchants offer 15 desks?
Which merchants offer 20 chairs?
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 (ground) call:
offer(desk,15)
returns moebureau

Deterministic (ground) call:
offer(chair,20)
returns moebureau

Which merchants offer 15 desks?
Which merchants offer 20 chairs?
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 (non-ground) call:**
offer(desk,15,Merchant) binds Merchant to moebureau

**Non-deterministic (non-ground) call:**
offer(chair,20,Merchant) binds Merchant to furniffice and (then) to moebureau

**Which merchants offer 15 desks?**

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

Points on offered furniture products in available quantities at merchants:

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

Non-Deterministic (ground) call:
offer(desk,15) returns moebureau

Which merchants offer 15 desks?

Which merchants offer 20 chairs?
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).

Non-deterministic (non-ground) call:

- offer(desk,Quantity,Merchant) binds Quantity=10, Merchant=furniffice
  and Quantity=15, Merchant=moebureau

Non-deterministic (non-ground) call:

- offer(chair,Quantity,Merchant) binds Quantity=21, Merchant=furniffice
  and Quantity=20, Merchant=moebureau

Deterministic Definition

Which merchants offer how many desks?

Which merchants offer how many chairs?
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.

Non-deterministic (non-ground) call:
offer(desk,Quantity) returns furniffice, binding Quantity=10
returns moebureau, binding Quantity=15

Non-deterministic (non-ground) call:
offer(chair,Quantity) returns furniffice, binding Quantity=21
returns moebureau, binding Quantity=20
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).

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

\[ \text{offer(} \text{chair,20,Merchant)} \text{, contact(} \text{Merchant,Person)} \]

- Merchant = furniffice
  - contact(merchant,Person)
    - Person = roberts
    - Person = sniders
- Merchant = moebureau
  - contact(merchant,Person)
    - Person = tellers
    - Person = leblanc
FP: Non-Deterministic Offer+Contact
Definitions for Non-/Deterministic Nestings

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

offer(desk,10) :& furniffice.  contact(furniffice) :& roberts.
offer(desk,15) :& moebureau. contact(furniffice) :& sniders.
offer(chair,20) :& furniffice. contact(furniffice) :& tellers.
offer(chair,20) :& moebureau. 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

\[
\text{contact( offer(chair,20) )}
\]

offer → furniffice

\[
\text{contact( furniffice )}
\]

contact → roberts

contact → sniders

contact → tellers

contact → leblanc

offer → moebureau

\[
\text{contact( moebureau )}
\]

‘→’ Returns
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).

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.

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).    site(furniffice,fredericton).
offer(desk,15,moebureau).    site(furniffice,moncton).
offer(chair,20,furniffice).   site(moebureau,moncton).
offer(chair,20,moebureau).

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.  
  site(furniffice) :& fredericton.
- offer(desk,15) :& moebureau.  
  site(furniffice) :& moncton.
- offer(chair,20) :& furniffice.  
  site(moebureau) :& moncton.
- offer(chair,20) :& moebureau.

**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).

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.  site(furniffice) &: fredericton.
offer(desk,15) &: moebureau.  site(furniffice) &: moncton.
offer(chair,20) &: furniffice.  site(moebureau) &: moncton.
offer(chair,20) &: moebureau.

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

(Externally) Deterministic conjunction:

Based on Built-in:
string< String-Less

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

binds Merch1 = furnifrice, Merch2 = moebureau, Town = moncton
(Merch1 = Merch2 = furnifrice 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

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

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

Which merchants (only different ones, and in alphabetical order) are in the same town?
FP: Cartesian Product by a Repeated Non-Deterministic Call

Points on offers and a pair definition:

\[
\text{offer(chair,20) :& furniffice.} \\
\text{offer(chair,20) :& moebureau.}
\]

\[
\text{pair(First,Second) :& [First,Second].} \quad \% \text{similar to active cns}
\]

Repeated non-deterministic call enumerates entire Cartesian product:

\[
\text{pair(offer(chair,20),offer(chair,20)) % \{[X,Y] \mid X,Y \in offer(chair,20)\}}
\]

\[
\text{returns [furniffice,furniffice] % \{[furniffice,furniffice],}
\]
\[
\text{returns [furniffice,moebureau] % [furniffice,moebureau],}
\]
\[
\text{returns [moebureau,furniffice] % [moebureau,furniffice],}
\]
\[
\text{returns [moebureau,moebureau] % [moebureau,moebureau]}
\]
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 \in offer(chair,20)\}

returns \[ [\text{furniffice, furniffice}], \text{binding Oc=furniffice} \]
returns \[ [\text{moebureau, moebureau}], \text{binding Oc=moebureau} \]
Points on offers, their merchants’ contacts + sites, and a pair definition:

offer(chair, 20) :& furnace.  contact(furnissue) :& tellers.
offer(chair, 20) :& moebureau.  contact(moebureau) :& leblanc.
site(furnissue) :& fredericton.  site(furnissue) :& 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:

\[
\begin{align*}
\text{offer(chair,20) :& furniffice.} & \quad \text{contact(furniffice) :& tellers.} \\
\text{offer(chair,20) :& moebureau.} & \quad \text{contact(moebureau) :& leblanc.} \\
\text{site(furniffice) :& fredericton.} & \quad \text{site(furniffice) :& moncton.} \\
\text{site(moebureau) :& moncton.} & \\
\text{pair(First,Second) :& [First,Second].} & \quad \% \text{similar to active cns}
\end{align*}
\]

Named non-deterministic call enumerates Cartesian product subset:

\[
\begin{align*}
\text{Oc} &= \text{offer(chair,20) \ & \ pair(site(Oc),contact(Oc))} \\
\text{returns} & \quad [\text{fredericton,tellers}], \quad \text{binding Oc=furniffice} \\
\text{returns} & \quad [\text{moncton,tellers}], \quad \text{binding Oc=furniffice} \\
\text{returns} & \quad [\text{moncton,leblanc}], \quad \text{binding Oc=moebureau}
\end{align*}
\]
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, 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

\text{incite}(\text{pretzel}, \text{Result})

\text{trigger}(\text{pretzel}, \text{ProductB1})

\text{Result} = \text{ProductB1} = \text{beer}

\text{trigger}(\text{pretzel}, \text{ProductB1}), \text{ incite}(\text{ProductB1}, \text{ProductC1})

\text{trigger}(\text{pretzel}, \text{beer}), \text{ incite}(\text{beer}, \text{ProductC1})

\text{trigger}(\text{beer}, \text{ProductC1})

\text{Result} = \text{ProductC1} = \text{wine}

\text{trigger}(\text{beer}, \text{ProductB2}), \text{ incite}(\text{ProductB2}, \text{ProductC2})

\text{trigger}(\text{beer}, \text{wine}), \text{ incite}(\text{wine}, \text{ProductC2})

\text{trigger}(\text{wine}, \text{ProductC2})

\text{Result} = \text{ProductC2} = \text{pickle}

In each computation step:

- The call to be selected next is \textit{underlined}
- Call results are put in \textit{italics}
- A call with non-deterministic alternatives is \textbf{bold-faced}
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.
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\textsubscript{Source} may be identical to Program\textsubscript{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:

<table>
<thead>
<tr>
<th>satisfied( Customer, Item, Price )</th>
</tr>
</thead>
<tbody>
<tr>
<td>john wine 17.95</td>
</tr>
<tr>
<td>peter beer 06.40</td>
</tr>
</tbody>
</table>

#### Prolog (Ground) Facts:

<table>
<thead>
<tr>
<th>satisfied( Customer, Item, Price )</th>
</tr>
</thead>
<tbody>
<tr>
<td>satisfied( john, wine, 17.95 )</td>
</tr>
<tr>
<td>satisfied( peter, beer, 06.40 )</td>
</tr>
</tbody>
</table>
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:

\[
\text{pay}(\text{john}, \text{fred}, 17.95) = \text{cheque} \quad \text{pay}(\text{peter}, \text{fred}, 6.40) = \text{cash}
\]

RFML (Ground) Markup:

\[
<\text{ft}>
\begin{align*}
&<\text{pattop}>
&<\text{con}>\text{pay}</con>
&<\text{con}>\text{john}</con>
&<\text{con}>\text{fred}</con>
&<\text{con}>17.95</con>
&<\text{/pattop}>
&<\text{con}>\text{cheque}</con>
\end{align*}
\begin{align*}
&<\text{pattop}>
&<\text{con}>\text{pay}</con>
&<\text{con}>\text{peter}</con>
&<\text{con}>\text{fred}</con>
&<\text{con}>6.40</con>
&<\text{/pattop}>
&<\text{con}>\text{cash}</con>
\end{align*}
<\text{/ft}>
\]
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:

\[
\text{acquire}(\text{Customer}, \text{Merchant}, \text{Item}, \text{Price}) = \text{pay}(\text{Customer}, \text{Merchant}, \text{Price}) \text{ if } \text{satisfied}(\text{Customer}, \text{Item}, \text{Price})
\]

Relfun (Non-Ground) Footed Rule:

\[
\text{acquire}(\text{Customer}, \text{Merchant}, \text{Item}, \text{Price}) :- \text{satisfied}(\text{Customer}, \text{Item}, \text{Price}) \& \text{pay}(\text{Customer}, \text{Merchant}, \text{Price}).
\]
Functional Rules: From Relfun to RFML

Relfun (Non-Ground) Footed Rule:

\[ \text{acquire}(C,M,I,P) :- \text{satisfied}(C,I,P) \land \text{pay}(C,M,P). \]

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):

```
<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'

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

```xml
<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):

```xml
<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):

```
<callop>
  <con item="wine">true</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))? ) >
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
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 \texttt{cns} structures
- These and several further transformations can be executed interactively as commands in Relfun, and most of them are combined by the \texttt{horizon} command, also used by the Relfun compiler
Relationalizing Functions: Flattening (Pseudo-Code Syntax)

Functional Program (fully nested):

\[
\text{fr-antonym}(\text{Mot}) = \text{en2fr}(\text{en-antonym}(\text{fr2en}(\text{Mot})))
\]

Functional-Logic Program (fully flattened):

\[
\text{fr-antonym}(\text{Mot}) \quad \text{if} \quad _1 = \text{en-antonym}(\text{fr2en}(\text{Mot})) \quad \text{then} \quad \text{en2fr}(\_1)
\]

Functional-Logic Program (partially flattened):

\[
\text{fr-antonym}(\text{Mot}) \quad \text{if} \quad _2 = \text{fr2en}(\text{Mot}) \quad \text{and} \quad _1 = \text{en-antonym}(\_2) \quad \text{then} \quad \text{en2fr}(\_1)
\]
Relationalizing Functions: Flattening (Relfun Syntax)

**Functional Program (fully nested):**

\[ \text{fr-antonym}(\text{Mot}) \ & \text{en2fr}(\text{en-antonym}(\text{fr2en}(\text{Mot}))) . \]

**Definition by Function Nesting**

**Functional-Logic Program (partially flattened):**

1st Flattening Step: Variable \(_1\)

\[ \text{fr-antonym}(\text{Mot}) :\_1 .\ = \text{en-antonym}(\text{fr2en}(\text{Mot})) \ & \text{en2fr}(\_1) . \]

**Command: flatten**

**Functional-Logic Program (fully flattened):**

2nd Flattening Step: Variable \(_2\)

\[ \text{fr-antonym}(\text{Mot}) :\_2 .\ = \text{fr2en}(\text{Mot}) \ & , \_1 .\ = \text{en-antonym}(\_2) \ & \text{en2fr}(\_1) . \]
Relationalizing Functions: Extra-Argument Insertion (Pseudo-Code Syntax)

Functional-Logic Program (results returned):

\[ \text{fr-antonym}(\text{Mot}) \text{ if } _2 = \text{fr2en}(\text{Mot}) \text{ and } _1 = \text{en-antonym}(_2) \text{ then } \text{en2fr}(_1) \]

Logic Program (results bound):

\[ \text{fr-antonym}(_3,\text{Mot}) \text{ if } \text{fr2en}(_2,\text{Mot}) \text{ and } \text{en-antonym}(_1,_2) \text{ and } \text{en2fr}(_3,_1) \]

Call Pattern (query variable):

\[ \text{fr-antonym}(\text{Franto},\text{noir}) \]
Relationalizing Functions: Extra-Argument Insertion (Relfun Syntax)

Functional-Logic Program (results returned):

\[
\text{fr-antonym}(\text{Mot}) : - \_2 .\ = \ \text{fr2en}(\text{Mot}) \ , \ \_1 .\ = \ \text{en-antonym}(\_2) \\
& \ \& \ \text{en2fr}(\_1) \ .
\]

Flat Definition: Variables \_1, \_2

Logic Program (results bound):

\[
\text{fr-antonym}(\_3,\text{Mot}) : - \ \text{fr2en}(\_2,\text{Mot}) \ , \ \text{en-antonym}(\_1,\_2) \\
& \ \& \ \text{en2fr}(\_3,\_1) \ .
\]

New 1st Argument: Variable \_3

Call Pattern (query variable):

\[
\text{fr-antonym}(\text{Franto},\text{noir})
\]

Combined Command: relationalize
Functionalizing Relations: Footening of Facts (Pseudo-Code Syntax)

**Logic Program** (implicit true value):

\[ \text{spending(Peter Miller, min 5000 euro, previous year)} = \text{true} \]

**Functional Program A** (explicit true value):

\[ \text{spending(Peter Miller, min 5000 euro, previous year)} = \text{true} \]

**Functional Program B** (explicit 1 value):

\[ \text{spending(Peter Miller, min 5000 euro, previous year)} = 1 \]
Functionalizing Relations: Footening of Facts (Relfun Syntax)

Logic Program (implicit true value):

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

Fact Definition

Functional Program A (explicit true value):

```
spending(Peter Miller, min 5000 euro, previous year) & true.
```

Command: `footer true`

‘true’-Footening

Functional Program B (explicit 1 value):

```
spending(Peter Miller, min 5000 euro, previous year) & 1.
```

Command: `footer 1`

‘1’-Footening
Functionalizing Relations: Footening of Rules (Pseudo-Code Syntax)

Logic Program (implicit true value):

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

Definition by Single Premise Call

Functional-Logic Program A (explicit true value):

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

‘true’-Footening

Functional-Logic Program B (explicit 1 value):

```
premium(Customer) if spending(Customer,min 5000 euro,previous year) then 1
```

‘1’-Footening
Functionalizing Relations: Footening of Rules (Relfun Syntax)

Logic Program (implicit true value):

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

Functional-Logic Program A (explicit true value):

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

Functional-Logic Program B (explicit 1 value):

premium(Customer) :-
    spending(Customer,min 5000 euro,previous year) & 1.

Command: footen true

Command: footen 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).
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).

% Relational (Numeric):
evenrn(0).
evenrn(R) :- evenrn(N), _1 .= 1+(N), R .= 1+(_1).

% Functional (Symbolic):
evenfs() :& 0.
evenfs() :- H .= evenfs() & suc[suc[H]].

% Relational (Symbolic):
evenrs(0).
evenrs(suc[suc[N]]) :- evenrs(N).
Four Variants of Non-Deterministic Even-Number Generation: ... + Extraged

% Functional (Numeric):
evenfn(0).
evenfn(_3) :- evenfn(_2), _1 .= 1+(_2), _3 .= 1+(_1).

% Relational (Numeric):
evenrn(0).
evenrn(R) :- evenrn(N), _1 .= 1+(N), R .= 1+(_1).

% Functional (Symbolic):
evenfs(0).
evenfs(suc[suc[H]]) :- evenfs(H).

% Relational (Symbolic):
evenrs(0).
evenrs(suc[suc[N]]) :- evenrs(N).

(= Relationalized)

Identical (up to variable renaming)
Four Variants of Non-Deterministic Even-Number Generation: Horizoned

% Functional (Numeric):

\begin{verbatim}
evenfn() & 0.
evenfn() :- _2 = evenfn(), _1 = 1+(_2) & 1+(_1).
\end{verbatim}

% Relational (Numeric):

\begin{verbatim}
evenrn(0).
evenrn(R) :- evenrn(N), _1 = 1+(N), R = 1+(_1) & true.
\end{verbatim}

% Functional (Symbolic):

\begin{verbatim}
evenfs() & 0.
evenfs() :- H = evenfs(), _1 = suc[H] & suc[_1].
\end{verbatim}

% Relational (Symbolic):

\begin{verbatim}
evenrs(0).
evenrs(_1) :- _2 = suc[N], _1 = suc[_2], evenrs(N) & true.
\end{verbatim}
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]]).
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]).
non-ground-test(cns[X,Y],cns[rs[_],cns[u,nil]],cns[cns[u,X],Y]).

Command: untup
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:

\[
evenfn(1) : \& \text{cns}[0,\text{nil}].
\]
\[
evenfn(I) :- >(I,1),
   \text{cns}[H,R] .= evenfn(1-(I)),
   H2 .= 1+(1+(H)) \& \text{cns}[H2,\text{cns}[H,R]].
\]

% Functional (Numeric) – horizon (_1=_4=cns[H,R] by normalizer):

\[
evenfn(1) : \& \text{cns}[0,\text{nil}].
\]
\[
evenfn(I) :- >(I,1),
   _1 .= \text{cns}[H,R], _2 .= 1-(I), _1 .= evenfn(_2),
   _3 .= 1+(H), H2 .= 1+(_3), _4 .= \text{cns}[H,R] \& \text{cns}[H2,_4].
\]
Summary

- Horizontal transformation techniques were introduced and illustrated via Relfun examples.
- Relfun’s `horizon` command transforms FP, LP, and FLP source programs into a flattened (but not extrarged) 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 `horizontal` steps are the replacement of anonymous variables and of active `cns` calls.
- All `horizontal` results can still be interpreted, but subsequent `WAM compilation` increases efficiency.