Declarative Programming

General idea:

- no coding of algorithms
- description of logical relationships
- powerful abstractions
  - ➜ domain specific languages
- higher programming level
- reliable and maintainable programs
  - ➜ pointer structures ⇒ algebraic data types
  - ➜ complex procedures ⇒ comprehensible parts (pattern matching, local definitions)

Declarative Programming: Paradigms

Functional programming:

- ➜ functions, λ-calculus
- ➜ equations
- ➜ (lazy) deterministic reduction

Logic programming:

- ➜ predicates, predicate logic
- ➜ logical formulas, Horn clauses
- ➜ constraint solving (unification)
- ➜ non-deterministic search for solutions

Functional Logic Languages

- efficient execution principles of functional languages
- flexibility of logic languages
- avoid non-declarative features of Prolog (arithmetic, I/O, cut)
- combine best of both worlds in a single model
  - ➜ higher-order functions $\leadsto$ design patterns
  - ➜ declarative I/O
  - ➜ concurrent constraints

Imperative vs. Declarative Programming

Readability, safety:

```
function fac(n: nat): nat =
begin
   z := 1;  p := 1;
   while z<n+1 do
   begin  p := p*z;  z := z+1  end;
   return(n)
end
```

```
fac 0     = 1
fac (n+1) = (n+1) * (fac n)
```

Quicksort: Classical imperative version:

```
procedure qsort(l,r: index);
var i,j: index; x,w: item
begin
   i := l;  j := r;
   x := a[(l+r) div 2];
   repeat
      while a[i] < x do i := i+1;
      while x < a[j] do j := j-1;
      if i <= j then
      begin w := a[i]; a[i] := a[j]; a[j] := w;
         i := i+1; j := j-1
      end
   until i > j;
   if l < j then qsort(l,j);
   if i < r then qsort(i,r);
end
```


Quicksort: Classical imperative version:

```
procedure qsort(l,r: index);
var i,j: index; x,w: item
begin
   i := l;  j := r;
   x := a[(l+r) div 2];
   repeat
      while a[i] < x do i := i+1;
      while x < a[j] do j := j-1;
      if i <= j then
      begin w := a[i]; a[i] := a[j]; a[j] := w;
         i := i+1; j := j-1
      end
   until i > j;
   if l < j then qsort(l,j);
   if i < r then qsort(i,r);
end
```

Declarative version:

```
qsort [] = []
qsort (x:l) =
    qsort (filter (<x) l)
    ++ [x]
    ++ qsort (filter (>=x) l)
```


IMPERATIVE VS. DECLARATIVE PROGRAMMING

Program development and maintenance:

```
function f(n: nat): nat =
begin
   write('Hallo');
   return(n*n)
end

... z:=f(3)*f(3) ...
```

Optimization: `... x:=f(3); z:=x*x ...` (?)

$\leadsto$ side effects complicate program optimization and transformation

CURRY

As a language for concrete examples, we use **Curry**:
[Dagstuhl'96, POPL'97]

- multi-paradigm language
- extension of Haskell (non-strict functional language)
- developed by an international initiative
- provide a standard for functional logic languages (research, teaching, application)
- several implementations available

Basis of Declarative Programming: Algebraic Data Types

Values in imperative languages: basic types + pointer structures

Declarative languages: **algebraic data types** (Haskell-like syntax)

```
data Bool   = True   | False
data Nat    = Z      | S Nat
data List a = []     | a : List a      -- [a]
data Tree a = Leaf a | Node [Tree a]

data Int = 0 | 1 | -1 | 2 | -2 | ...
```

Value $\approx$ **data term**, **constructor term**:
well-formed expression containing variables and data type constructors

`(S Z)`   `1:(2:[])`   `[1,2]`   `Node [Leaf 3, Node [Leaf 4, Leaf 5]]`

Functional Programs

Functions: operations on values defined by equations (or rules)

$$f\ t_1\ \ldots\ t_n \mid c\ =\ r$$

defined operation — data terms — condition (optional) — expression

```
   Z + y = y                   Z ≤ y       = True
(S x) + y = S(x+y)          (S x) ≤ Z      = False
                            (S x) ≤ (S y)  = x≤y

    [] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

depth (Leaf _)      = 1
depth (Node [])     = 1
depth (Node (t:ts)) = max (1+depth t) (depth (Node ts))
```


Evaluation: Computing Values

Reduce expressions to their values

Replace equals by equals

Apply reduction step to a subterm (redex, *red*ucible *ex*pression):

variables in rule's left-hand side are universally quantified
$\leadsto$ match lhs against subterm (instantiate these variables)

```
   Z + y = y                   Z ≤ y       = True
(S x) + y = S(x+y)          (S x) ≤ Z      = False
                            (S x) ≤ (S y)  = x≤y
```

`(S Z)+(S Z)` $\rightarrow$ `S (Z+(S Z))` $\rightarrow$ `S (S Z)`

Evaluation Strategies

Expressions with several redexes: which evaluate first?

Strict evaluation: select an innermost redex ($\approx$ call-by-value)

Lazy evaluation: select an outermost redex

```
   Z + y = y                   Z ≤ y       = True
(S x) + y = S(x+y)          (S x) ≤ Z      = False
                            (S x) ≤ (S y)  = x≤y
```

Strict evaluation:
`Z ≤ (S Z)+(S Z)` $\rightarrow$ `Z ≤ (S (Z+(S Z))` $\rightarrow$ `Z ≤ (S (S Z))` $\rightarrow$ `True`

Lazy evaluation:
`Z ≤ (S Z)+(S Z)` $\rightarrow$ `True`

Strict evaluation might need more steps, but it can be even worse...

```
      Z + y = y                   Z ≤ y     = True
(S x) + y = S(x+y)            (S x) ≤ Z     = False
                              (S x) ≤ (S y) = x≤y

        f = f
```

Lazy evaluation:

Z+Z ≤ f → Z ≤ f → True

Strict evaluation:

Z+Z ≤f → Z+Z ≤ f → Z+Z ≤ f → ⋯

Ideal strategy: evaluate only needed redexes
(i.e., redexes necessary to compute a value)

Determine needed redexes with definitional trees

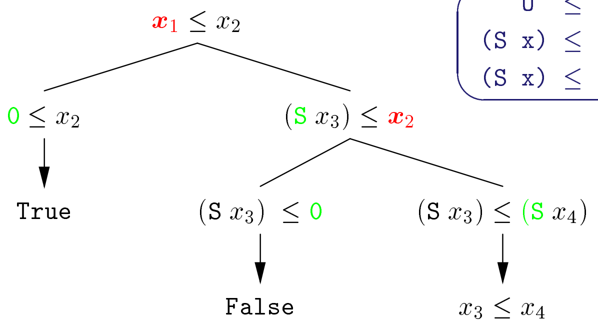
DEFINITIONAL TREES [ANTOY 92]

➜ data structure to organize the rules of an operation

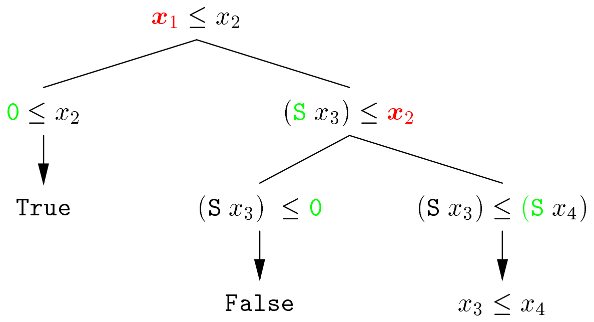
➜ each node has a distinct *pattern*

➜ *branch* nodes (case distinction), *rule* nodes

```
    0   ≤ y     = True
(S x) ≤ 0     = False
(S x) ≤ (S y) = x≤y
```



EVALUATION WITH DEFINITIONAL TREES



Evaluating function call $t_1 \leq t_2$:

① Reduce t_1 to head normal form (constructor-rooted expression)

② If $t_1 = 0$: apply rule

③ If $t_1 = (\mathrm{S} \ldots)$: reduce t_2 to head normal form

PROPERTIES OF REDUCTION WITH DEFINITIONAL TREES

- **Normalizing strategy**
  i.e., always computes value if it exists ≈ sound and complete
- Independent on the order of rules
- Definitional trees can be automatically generated
  → pattern matching compiler
- Identical to lazy functional languages (e.g, Miranda, Haskell) for the subclass of uniform programs
  (i.e., programs with strong left-to-right pattern matching)
- **Optimal strategy:** each reduction step is needed
- Easily extensible to more general classes

Higher-Order Functions

Functions are first class citizens

- ➜ passing functions as parameters and results
- ➜ combinator-oriented programming
- ➜ expressing design patterns
- ➜ code reuse

```
map :: (a -> b) -> [a] -> [b]

map f []     = []
map f (x:xs) = (f x) : map f xs
```

`map (1 +) [2,3,4]` $\leadsto$ `[3,4,5]`

Partial application: `(1 +)` is a function of type `Int->Int`

λ-abstraction: `\x->1+x` (anonymous function)

Higher-order Functions: Examples

Accumulate list elements with a binary operator:

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

Multiply all list elements: `foldr (*) 1 xs`

Concatenate a list of lists: `concat xs = foldr (++) [] xs`

Tree example: computing list of all leaves in a tree:

```
frontier :: Tree a -> [a]

frontier (Leaf v) = [v]
frontier (Node ns) = concat (map frontier ns)
```


Filter all elements in a list satisfying a given predicate:

```
filter            :: (a -> Bool) -> [a] -> [a]
filter p []       = []
filter p (x:xs)   = if p x then x : filter p xs
                              else filter p xs
```

Now the code for quicksort becomes straightforward:

```
qsort [] = []
qsort (x:l) = qsort (filter (<x) l)
              ++ [x] ++ qsort (filter (>=x) l)
```


Application: HTML Programming

Data type for representing HTML expressions:

```
data HtmlExp = HText String
             | HStruct String [(String,String)] [HtmlExp]
```

`HStruct "A" [("HREF","http://...")] [HText "click here"]`

Get all hypertext links in an HTML document:

```
hrefs [] = []
hrefs (HText _ : hs) = hrefs hs
hrefs (HStruct tag attrs shs : hs) =
(if tag=="A" then map snd (filter (\(t,_)->t=="HREF") attrs)
            else [] ) ++ hrefs shs ++ hrefs hs
```

Non-deterministic Evaluation

Previous functions: inductively defined on data structures

Sometimes overlapping rules more natural:

```
True  ∨ x     = True
    x ∨ True  = True
False ∨ False = False
```

First two rules overlap on `True ∨ True`

$\leadsto$ Problem: no needed argument: $\boxed{e_1 \vee e_2}$ evaluate e_1 or e_2?

Functional languages: backtracking: Evaluate e_1, if not successful: e_2

Disadvantage: not normalizing (e_1 may not terminate)

Non-deterministic Evaluation

```
True  ∨ x     = True
    x ∨ True  = True
False ∨ False = False
```

Evaluation of $\boxed{e_1 \vee e_2}$?

1. Parallel reduction of e_1 and e_2 [Sekar/Ramakrishnan 93]
2. **Non-deterministic reduction:** try (*don't know*) e_1 or e_2

Extension to definitional trees / pattern matching:
Introduce or-nodes to describe non-deterministic selection of redexes

$\leadsto$ non-deterministic evaluation: $e \quad \to \quad \underbrace{e_1 \mid \cdots \mid e_n}_{\text{disjunctive expression}}$

$\leadsto$ non-deterministic functions

Non-deterministic Functions

Functions can have more than one result value:

```
choose x y  =  x
choose x y  =  y
```

`choose 1 2` $\to$ `1 | 2`

Non-deterministic list insertion and permutations:

```
insert x []      =  [x]
insert x (y:ys)  =  choose (x:y:ys) (y:insert x ys)

permute []       =  []
permute (x:xs)   =  insert x (permute xs)
```

`permute [1,2,3]` $\to$
`[1,2,3] | [2,1,3] | [2,3,1] | [1,3,2] | [3,1,2] | [3,2,1]`

Logic Programming

Distinguished features:

- ➜ compute with partial information (constraints)
- ➜ deal with free variables in expressions
- ➜ compute solutions to free variables
- ➜ built-in search
- ➜ non-deterministic evaluation

Functional programming: values, no free variables

Logic programming: computed answers for free variables

Operational extension: instantiate free variables, if necessary

From Functional Programming to Logic Programming

```
f 0 = 2
f 1 = 3
```

Evaluate `(f x)`: – bind `x` to `0` and reduce `(f 0)` to `2`, or:

– bind `x` to `1` and reduce `(f 1)` to `3`

Computation step: $\underbrace{\textbf{bind}}_{\textit{logic}}$ and $\underbrace{\textbf{reduce}}_{\textit{functional}}$: $e \leadsto \underbrace{\{\sigma_1\}\ e_1 \mid \cdots \mid \{\sigma_n\}\ e_n}_{\text{disjunctive expression}}$

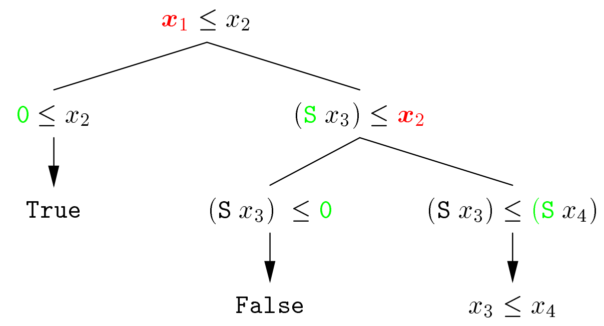
Reduce: `(f 0)` $\leadsto$ `2`

Bind and reduce: `(f x)` $\leadsto$ `{x=0} 2 | {x=1} 3`

Compute necessary bindings with needed strategy

$\leadsto$ **needed narrowing** [Antoy/Echahed/Hanus POPL'94/JACM'00]

Evaluation with Definitional Trees



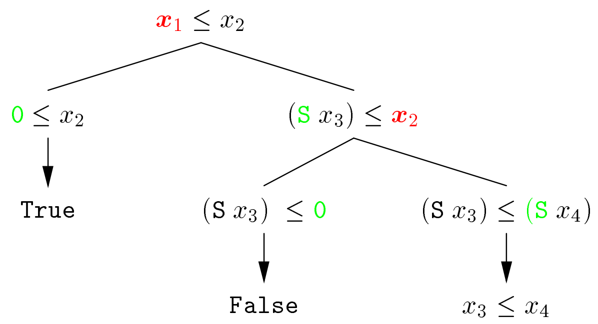
Evaluating function call $t_1 \le t_2$:

① Reduce t_1 to head normal form

② If $t_1 = 0$: apply rule

③ If $t_1 = (S \ldots)$: reduce t_2 to head normal form

Needed Narrowing



Evaluating function call $t_1 \le t_2$:

① Reduce t_1 to head normal form

② If $t_1 = 0$: apply rule

③ If $t_1 = (S \ldots)$: reduce t_2 to head normal form

④ If t_1 variable: bind t_1 to `0` or `(S x)`

Properties of Needed Narrowing

Sound and **complete** (w.r.t. strict equality, no termination requirement)

Optimality:

① **No unnecessary steps:**
Each narrowing step is needed, i.e., it cannot be avoided if a solution should be computed.

② **Shortest derivations:**
If common subterms are shared, needed narrowing derivations have minimal length.

③ **Minimal set of computed solutions:**
Two solutions σ and σ' computed by two distinct derivations are independent.

Properties of Needed Narrowing

Determinism:
No non-deterministic step during the evaluation of ground expressions
(≈ functional programming)

Restriction: inductively sequential rules
(i.e., no overlapping left-hand sides)

Extensible to

- ➜ conditional rules [Hanus ICLP'95]
- ➜ overlapping left-hand sides [Antoy/Echahed/Hanus ICLP'97]
- ➜ multiple right-hand sides [Antoy ALP'97]
- ➜ concurrent evaluation [Hanus POPL'97]

Strict Equality

Problems with equality in the presence of non-terminating rules:

1. Equality on infinite objects undecidable:

```
f = 0:f          g = 0:g
```

Is `f = g` valid?

2. Semantics of non-terminating functions:

```
f x = f (x+1)          g x = g (x+1)
```

Is `f 0 = g 0` valid?

Avoided by **strict equality**: identity on *finite* objects
(both sides reducible to same ground data term)

Equational Constraints

Logic programming: solve goals, compute solutions

Functional logic programming: solve equations

Strict equality: only reasonable notion of equality in the presence of non-terminating functions

Equational constraint $\boxed{e_1 =:= e_2}$

satisfied if both sides evaluable to unifiable data terms

⇒ $e_1 =:= e_2$ does not hold if e_1 or e_2 undefined or infinite

⇒ $e_1 =:= e_2$ and e_1, e_2 data terms ≈ unification in logic programming

Functional Logic Programming: Examples

List concatenation:

```
append :: [a] -> [a] -> [a]

append []     ys  =  ys
append (x:xs) ys  =  x : append xs ys
```

Functional programming:

`append [1,2] [3,4]` ⇝ `[1,2,3,4]`

Logic programming:

`append x y =:= [1,2]` ⇝

`{x=[],y=[1,2]} | {x=[1],y=[2]} | {x=[1,2],y=[]}`

Last list element: `last xs | append ys [x] =:= xs  = x`

FUNCTIONAL LOGIC PROGRAMMING: EXAMPLES

Infinite list of natural numbers:

```
              from x  =  x : from (S x)

   first Z     ys     =  []
   first (S x) (y:ys) =  y : first x ys
```

Lazy functional programming:

`first (S(S Z)) (from Z)` $\rightsquigarrow$ `[Z,(S Z)]`

Lazy functional logic programming:

`first x (from y) =:= [Z]` $\rightsquigarrow$ `{x=(S Z),y=Z}`

PROGRAMMING DEMAND-DRIVEN SEARCH

Non-deterministic functions for generating permutations:

```
insert x []       =  [x]
insert x (y:ys)   =  choose (x:y:ys) (y:insert x ys)

permute []        =  []
permute (x:xs)    =  insert x (permute xs)
```

Sorting lists with test-of-generate principle:

```
sorted []   = []
sorted [x]  = [x]
sorted (x:y:ys) | x<=y  = x : sorted (y:ys)

psort xs = sorted (permute xs)
```


Advantages of non-deterministic functions as generators:

- ➜ demand-driven generation of solutions (due to laziness)
- ➜ modular program structure

`psort [5,4,3,2,1]` $\rightsquigarrow$ `sorted (permute [5,4,3,2,1])`

$\rightsquigarrow^*$ $\underbrace{\texttt{sorted (5 : 4 : permute [3,2,1])}}_{\text{undefined: discard this alternative}}$ $\mid \cdots$

Effect: Permutations of `[3,2,1]` are not enumerated!

Permutation sort for $[n, n-1, \ldots, 2, 1]$: #or-branches/disjunctions

Length of the list:	4	5	6	8	10
generate-and-test	24	120	720	40320	3628800
test-of-generate	19	59	180	1637	14758

SEARCH STRATEGIES AND ENCAPSULATED SEARCH

How to deal with non-deterministic computation steps?

- ➜ explore alternatives in parallel $\rightsquigarrow$ parallel architectures
- ➜ explore alternatives by backtracking $\rightsquigarrow$ Prolog
- ➜ support flexible search strategies $\rightsquigarrow$ encapsulate search

Disadvantages of fixed search (like backtracking):

- ➜ no application-dependent strategy or efficiency control
- ➜ global search: local search has global effects
- ➜ I/O operations not backtrackable
- ➜ problems with concurrency and backtracking

Solution: provide primitives for user-definable search strategies (Oz [Schulte/Smolka 94], Curry [Hanus/Steiner 98])

Encapsulated Search

Idea:
Compute until a non-deterministic step occurs, then give programmer control over this situation

Search:

- ➜ solve constraint
- ➜ evaluate until failure, success, or non-determinism
- ➜ return result in a list

First approach to primitive **search operator:**

```
try :: Constraint -> [Constraint]
```


Search Operator: First Approach

```
try :: Constraint -> [Constraint]
```

```
f 0 = 2
f 1 = 3
```

`try (1=:=2)` $\rightsquigarrow$ `[]`		failure
`try ([x]=:=[0])` $\rightsquigarrow$ `[x=:=0]`		success
`try (f x =:= 3)` $\rightsquigarrow$ `[x=:=0 & f 0 =:= 3,`		
`x=:=1 & f 1 =:= 3]`		disjunction

Problem: incompatible bindings for `x` in disjunctions!

Solution: abstract search variable in constraints: `\x->`c

Search Operator: Final Approach

Search goal: constraint with abstracted search variable

Search operator `try`: maps search goal into list of search goals

```
try :: (a->Constraint) -> [a->Constraint]
```

```
f 0 = 2
f 1 = 3
```

`try \x->1=:=2` $\rightsquigarrow$ `[]`		failure
`try \x->[x]=:=[0]` $\rightsquigarrow$ `[\x->x=:=0]`		success
`try \x->f x =:= 3` $\rightsquigarrow$ `[\x->x=:=0 & f 0 =:= 3,`		
`\x->x=:=1 & f 1 =:= 3]`		disjunction

Encapsulated Search: Search Strategies

`try \`x`->`c: evaluate c, stop after non-deterministic step

Depth-first search: collect all solutions in a list

```
all :: (a->Constraint) -> [a->Constraint]

all g = collect (try g)

 where collect []          = []
       collect [g]         = [g]
       collect (g1:g2:gs)  = concat (map all (g1:g2:gs))
```

`all (\xs -> append xs [1] =:= [0,1])` $\rightsquigarrow$ `[\xs -> xs =:= [0]]`

Encapsulated Search: Further Search Strategies

- compute only the first solution:

```
once g = head (all g)        where head (x:xs) = x
```

Note: lazy evaluation is important here!

(strict languages, like Oz, must define new search operator)

⇝ lazy evaluation supports better reuse

- `findall`, best solution search, parallel search, ...
- negation as failure:

```
naf c = (all \_->c) =:= []
```

⇝ control failures

Handling Solutions

Extract value of the search variable by application of search goal:

```
(\x->x=:=1) freevar  ⇝  freevar=:=1
                     ⇝  {freevar=1} success
```

Prolog's findall:

```
unpack :: (a->Constraint) -> a
unpack g | g x  = x     where x free

findall g = map unpack (all g)
```

Compute all splittings of a list:

```
findall (\(x,y) -> append x y =:= [1,2])
*⇒ [([],[1,2]), ([1],[2]), ([1,2],[])]
```


Exploiting Laziness

Show a list of search goals, as requested by the user:

```
printloop []     = putStr "no\n"
printloop (a:as) = browse a >> putStr "? " >>
                   getChar >>= evalAnswer as

evalAnswer as ';'  = newline >> printloop as
evalAnswer as '\n' = newline >> putStr "yes\n"
```

Prolog's top-level: `prolog g = printloop (all g)`

```
prolog \(x,y) -> append x y =:= [1,2]
*⇒  ([],[1,2]) ?  ;
    ([1],[2]) ?  <-
    yes
prolog \x -> 1=:=2   *⇒   no
```


Laziness easily supports demand-driven encapsulated search

⇒ **Separation of Logic and Control**

⇒ **Modularity:**

- Prolog's top-level with breadth-first search:
  `prolog_bfs g = printloop (bfs g)`
- Prolog's top-level with depth-bounded search:
  `prolog_bound g bd = printloop (bound g bd)`

MONADIC INPUT/OUTPUT

Problem: Handling input/output in a declarative manner?

Solution: Consider the external world as a parameter to all I/O operations (Haskell, Mercury)

I/O actions: transformations on the external world

Interactive program: sequence(!) of actions applied to the external world

Type of I/O actions: `IO a` ≈ `World -> (a,World)`

But: the "world" is implicit parameter, not explicitly accessible!

Some primitive I/O actions:.

```
getChar :: IO Char            -- read character from stdin
putChar :: Char -> IO ()      -- write argument to stdout
return  :: a -> IO a          -- do nothing and return argument
```

`getChar` applied to a world $\leadsto$ character + new (transformed) world

Compose actions: `(>>=) :: IO a -> (a -> IO b) -> IO b`

`getChar >>= putChar`: copy character from input to output

Specialized composition: ignore result of first action:

```
  (>>)   :: IO a -> IO b -> IO b
 x >> y  =  x >>= \_->y
```


Example: output action for strings (`String` ≈ `[Char]`)

```
putStr :: String -> IO ()
putStr []     = return ()
putStr (c:cs) = putChar c >> putStr cs
```

Example: read a line

```
getLine :: IO String
getLine = getChar >>= \c ->
          if c=='\n' then return []
                     else getLine >>= \cs -> return (c:cs)
```


Monadic composition not well readable

$\leadsto$ syntactic sugar: Haskell's `do` notation

$$\begin{array}{ll} \texttt{do}\ p\ \texttt{<-}\ a_1 & \approx \quad a_1\ \texttt{>>=}\ \backslash p\ \texttt{->}\ a_2 \\ \quad a_2 & \end{array}$$

Example: read a line (with `do` notation)

```
getLine = do c <- getChar
             if c=='\n' then return []
                        else do cs <- getLine
                                return (c:cs)
```

Note: no I/O in disjunctions ("cannot copy the world")

$\leadsto$ encapsulate search between I/O actions

Constraint Programming

Logic Programming:

- ➜ compute with partial information (constraints)
- ➜ data structures (constraint domain): constructor terms
- ➜ basic constraint: (strict) equality
- ➜ constraint solver: unification

Constraint Programming: generalizes logic programming by

- ➜ new specific constraint domains (e.g., reals, finite sets)
- ➜ new basic constraints over these domains
- ➜ sophisticated constraint solvers for these constraints

Constraint Programming over Reals

Constraint domain: real numbers

Basic constraints: equations / inequations over real arithmetic expressions

Constraint solvers: Gaussian elimination, simplex method

Examples:

```
5.1 =:= x + 3.5          ⇝  {x=1.6}
x<=1.5 & x+1.3>=2.8      ⇝  {x=1.5}
```


Example: Circuit Analysis

Define relation `cvi` between electrical circuit, voltage, and current

Circuits are defined by the data type

```
data Circuit = Resistor Float
             | Series   Circuit Circuit
             | Parallel Circuit Circuit
             ⋮
```

Rules for relation `cvi`:

```
cvi (Resistor r) v i =  v =:= i * r        -- Ohm's law

cvi (Series   c1 c2) v i =                 -- Kirchhoff's law
    v=:=v1+v2  &  cvi c1 v1 i  &  cvi c2 v2 i

cvi (Parallel c1 c2) v i =                 -- Kirchhoff's law
    i=:=i1+i2  &  cvi c1 v i1  &  cvi c2 v i2
```


Querying the circuit specification:

Current in a sequence of resistors:

```
cvi (Series (Resistor 180.0) (Resistor 470.0)) 5.0 i
⇝ {i = 0.007692307692307693}
```

Relation between resistance and voltage in a circuit:

```
cvi (Series (Series (Resistor r) (Resistor r)) (Resistor r)) v 5.0
⇝ {v=15.0*r}
```

Also synthesis of circuits possible

Constraint Programming with Finite Domains

Constraint domain: finite set of values

Basic constraints: equality / disequality / membership / ...

Constraint solvers: OR methods (e.g., arc consistency)

Application areas: combinatorial problems
(job scheduling, timetabling, routing,...)

General method:

① define the domain of the variables (possible values)
② define the constraints between all variables
③ "labeling", i.e., non-deterministic instantiation of the variables

constraint solver reduces the domain of the variables by sophisticated pruning techniques using the given constraints

Usually: finite domain ≈ finite subset of integers

Example: A Crypto-Arithmetic Puzzle

Assign a different digit to each different letter such that the following calculation is valid:

```
    s e n d
  + m o r e
  ---------
  m o n e y
```

```
puzzle s e n d m o r y =
  domain [s,e,n,d,m,o,r,y] 0 9  &        -- define domain
  s > 0  &  m > 0  &                     -- define constraints
  all_different [s,e,n,d,m,o,r,y]  &
                1000 * s + 100 * e + 10 * n + d
   +            1000 * m + 100 * o + 10 * r + e
   = 10000 * m + 1000 * o + 100 * n + 10 * e + y  &
  labeling [s,e,n,d,m,o,r,y]            -- instantiate variables
```

```
puzzle s e n d m o r y  ⇝  {s=9,e=5,n=6,d=7,m=1,o=0,r=8,y=2}
```


From Functional Logic to Concurrent Programming

Disadvantage of narrowing:

- ➜ functions on recursive data structures ⇝ narrowing may not terminate
- ➜ all rules must be explicitly known ⇝ combination with external functions?

Solution: Delay function calls if a needed argument is free

⇝ **residuation principle** [Aït-Kaci et al. 87]
(used in Escher, Le Fun, Life, NUE-Prolog, Oz,...)

Distinguish: **rigid** (consumer) and **flexible** (generator) functions

Necessary: Concurrent conjunction of constraints: c_1 & c_2

Meaning: evaluate c_1 and c_2 concurrently, if possible

Flexible vs. Rigid Functions

```
f 0 = 2
f 1 = 3
```

rigid/flexible status not relevant for ground calls:

```
f 1  ⇝  3
```

`f` flexible:

```
f x =:= y  ⇝  {x=0,y=2} | {x=1,y=3}
```

`f` rigid:

```
f x =:= y  ⇝  suspend
```

```
f x =:= y & x =:= 1  ⇝  {x=1} f 1 =:= y    (suspend f x)
                     ⇝  {x=1} 3 =:= y      (evaluate f 1)
                     ⇝  {x=1,y=3}
```

Default in Curry: constraints are flexible, all others are rigid

Parallel Functional Programming

Parallel evaluation of arguments:

```
f t1 t2 = letpar  x = g t1
                  y = h t2  in  k x y
```

with concurrent conjunction of equations:

```
f t1 t2 | x =:= g t1 & y = h t2  = k x y
        where x,y free
```

Skeleton-based parallel programming:

`farm`: parallel version of `map`

```
farm f []      = []
farm f (x:xs) | r =:= f x & rs =:= farm f xs
               = r : rs       where r,rs free
```


External Functions

External functions: implemented in another language (e.g., C, Java,...)

Conceptually definable by an infinite set of equations, e.g.,

```
0+0 = 0     1+0 = 1     2+0 = 2     ...
0+1 = 1     1+1 = 2     ...
0+2 = 2     ...
...
```

Definition not accessible, infinite disjunctions

- ➜ suspend external function calls until arguments are fully known, i.e., ground [Bonnier/Maluszynski 88, Boye 91]
- ➜ no extension to presented computation model (external functions are rigid), but *not possible in narrowing-based languages!*
- ➜ reuse of existing libraries

Standard Arithmetic

Implementation of standard arithmetic (+, -, ∗,...) as external functions:

$0, 1, 2, \ldots$: constructors

+, -, ∗,...: external functions

`x =:= 2+3*4` $\rightsquigarrow$ `{x=14}`

`x =:= 2*3+y` $\rightsquigarrow$ `{} x =:= 6+y` (suspend)

`x+x =:= y & x =:= 2`
$\rightsquigarrow$ `{x=2} 2+2 =:= y` (suspend x+x)
$\rightsquigarrow$ `{x=2} 4 =:= y` (evaluate 2+2)
$\rightsquigarrow$ `{x=2, y=4}`

⇒ Rigid functions as passive constraints (Life)

External functions as passive constraints:

```
digit 0 = success
...
digit 9 = success
```

The constraint `digit` acts as a generator:

`x+x =:= y & x*x =:= y & digit x`

$\rightsquigarrow$ `{x=0, y=0} | {x=2, y=4}`

HIGHER-ORDER FUNCTIONAL LOGIC PROGRAMMING

```
map :: (a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = (f x) : map f xs
```

Functional programming: `map (1 +) [2,3,4]` $\leadsto$ `[3,4,5]`

Logic programming: `map f [2,3,4] =:= [3,4,5]` $\leadsto$???

- ➜ consider application function `f $ x = (f x)` as external
- ➜ consider partial applications as data terms
- ➜ first-order definition of application function `($)` (as in [Warren 82]):

```
(+)   $ x = (+ x)    -- right-hand side is data term
(+ x) $ y = x+y      -- evaluate right-hand side
```


Reasonable: application function `($)` is rigid

$\leadsto$ delay applications of unknown functions

$\leadsto$ `map f [2,3,4]` suspends

Other solutions possible but more expensive:

- ➜ `($)` is flexible $\leadsto$ guess unknown functions
- ➜ solver for higher-order equations
  (higher-order unification, higher-order needed narrowing)

UNIFICATION OF DECLARATIVE COMPUTATION MODELS

Computation model	Restrictions on programs
Needed narrowing	inductively sequential rules; optimal strategy
Weakly needed narrowing (∼Babel)	only flexible functions
Resolution (∼Prolog)	only (flexible) predicates (∼ constraints)
Lazy functional languages (∼Haskell)	no free variables in expressions
Parallel functional langs. (∼Goffin, Eden)	only rigid functions, concurrent conjunction
Residuation (∼Life, Oz)	constraints are flexible; all others are rigid

CONCURRENT OBJECTS WITH STATE

Modeling objects with state as a (rigid!) constraint function:

- ➜ first parameter: current state
- ➜ second parameter: message stream (rigid ≈ wait for input)

Example: **Counter object**

```
data CounterMessage = Set Int | Inc | Get Int

counter :: Int -> [CounterMessage] -> Constraint
counter eval rigid  -- declare as rigid
counter _ (Set v : ms)  = counter v ms
counter n (Inc   : ms)  = counter (n+1) ms
counter n (Get v : ms)  = v=:=n & counter n ms
counter _ []            = success
```

Concurrent Objects with State: A Counter

```
counter _ (Set v : ms) = counter v ms
counter n (Inc   : ms) = counter (n+1) ms
counter n (Get v : ms) = v=:=n & counter n ms
counter _ []           = success
```

```
counter 0 s  &  -- create counter object
                s =:= [Set 41, Inc, Get x]
```

$\leadsto$ {x=42, s=...}

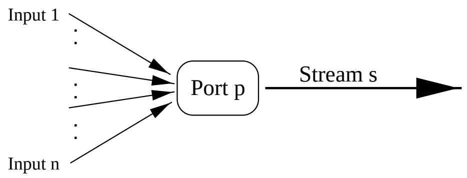
Also: incremental instantiation of s (message sending)

Several sending processes $\leadsto$ merge message streams

Ports for Distributed Systems

Distributed systems: $n \to 1$-communication with dynamic connections

Port [Janson et al. 93, AKL]: constraint between multiset p and stream s satisfied if elements in p and s are identical



Two constraints on ports:

`openPort p s`	open port p with stream s
`send m p`	constrain p to hold message m

Previous counter with two clients:

```
openPort p s &> counter 0 s & client1 p & client2 p
```


Ports for Distributed Systems

- communication based on logic (constraint solving)
- simple extension of base semantics
- `send` instantiates end of stream s (in constant time)

  `s_tail =:= (m:new_s_tail)`

  $\leadsto$ strict communication
- provides efficient implementation
  (senders have no access to old messages)
- free variables in messages $\approx$ reply channels
- dynamic extension of senders (pass port variable)

External Ports

I/O actions for external communication
(between different programs running on different machines):

```
openNamedPort :: String -> IO [a]

connectPort   :: String -> IO (Port a)
```

`openNamedPort pn`: open new external port with global name pn and return stream of incoming messages

`connectPort pn`: return port with global name pn

(similar concepts: external objects in Oz, registered processes in Erlang)

A simple example: a global counter server

The server side: (started on `medoc.cs.uni-kiel.de`)

```
main = openNamedPort "counter" >>= c_server

c_server s | counter 0 s  = done
```

The client side:

```
client pn m = connectPort pn >>= sendPort m

sendPort msg p | send msg p = done
```

Increment the global counter:

```
client "counter@medoc.cs.uni-kiel.de" Inc
```

Ask the counters current value:

```
client "counter@medoc.cs.uni-kiel.de" (Get v)  ⤳  {v=...}
```


A NAME SERVER

Messages: "`PutName` n i" (assign i to name n)   "`GetName` n i"

```
nameserver = openNamedPort "nameserver" >>= serverloop \_->0

serverloop n2i (GetName n i : ms) | i=:=(n2i n)
                                  = serverloop n2i ms

serverloop n2i (PutName n i : ms) = serverloop new_n2i ms
  where new_n2i m = if m==n then i else n2i m
```

The client side:

```
client "nameserver@..." (PutName "talk" 42)
client "nameserver@..." (GetName "talk" x)  ⤳  {x=42}
```


A HIERARCHICAL NAME SERVER

Internet domain name server: ask master server if name locally unknown

Implementation by slight modification of previous name server:

```
serverloop n2i (GetName n i : ms)
   | if (n2i n)==0  then send (GetName n i) master
                    else i=:=(n2i n)
                                  = serverloop n2i ms

serverloop n2i (PutName n i : ms) = serverloop new_n2i ms
  where new_n2i m = if m==n then i else n2i m
```


A COMPUTATION SERVER

Strict communication, no RPCs ⤳ no direct way to distribute work

Computation server: accepts messages (f, x, y)

```
start_cserver = openNamedPort "compserver" >>= compserver

compserver ((f,x,y) : ms) | y=:=(f x) = compserver ms
```

Client side: `client "compserver@cs" (prime,1000,p)  ⤳  {p=7919}`

- ➜ consider partially applied function calls as data terms
- ➜ asynchronous RPCs
  (free result variable ≈ "promise" [Liskov/Shrira 88])
- ➜ concurrent server:

```
compserver eval rigid
compserver ((f,x,y) : ms) = y=:=(f x) & compserver ms
```

A Model for Multi-Paradigm Programming

Integration of different programming paradigms is possible

Functional programming is a good starting point:

- ➜ lazy evaluation ⇝ modularity, optimal evaluation
- ➜ higher-order functions ⇝ code reuse, design patterns
- ➜ polymorphism ⇝ type safety, static checking

Stepwise extensible in a conservative manner to cover

- ➜ logic programming: non-determinism, free variables
- ➜ constraint programming: specific constraint structures
- ➜ concurrent programming: suspending function calls, synchronization on logical variables
- ➜ object-oriented programming: constraint functions, ports
- ➜ imperative programming: monadic I/O, sequential composition
- ➜ distributed programming: external ports

Why Integration of Declarative Paradigms?

- more expressive than pure functional languages (compute with partial information/constraints)
- more structural information than in pure logic programs (functional dependencies)
- more efficient than logic programs (determinism, laziness)
- functions: declarative notion to improve control in logic programming
- avoid impure features of Prolog (arithmetic, I/O)
- combine research efforts in FP and LP
- do not teach two paradigms, but one: **declarative programming** [Hanus PLILP'97]
- choose the most appropriate features for application programming

Application of Multi-Paradigm Languages

So far: high-level approach to

- ➜ search problems
- ➜ constraint solving
- ➜ distributed systems

In the following: appropriate to develop domain-specific languages for

- ➜ graphical user interfaces
- ➜ parsing
- ➜ HTML/CGI programming

Functional Logic GUI Programming

[Hanus PADL'00]

Graphical User Interfaces (GUIs) have a

- ➜ layout structure ⇝ hierarchical structure, algebraic data type
- ➜ logical structure ⇝ dependencies in the layout structure

Tcl/Tk: assign strings to layout elements ⇝ run-time errors

Here: use logical variables as references ⇝ compiler errors

A simple "Hello world" GUI:


```
runWidget "Hello"
  (TkCol [TkLabel [TkText "Hello world!"],
          TkButton tkExit [TkText "Stop"]])
```

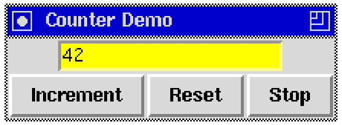
Layout Structure of GUIs

Specify hiearchical GUI layout as a "TkWidget" term:

```
data TkWidget a =
     TkButton (GUIRef -> a) [TkConfItem a]
   | TkCheckButton    [TkConfItem a]
   | TkEntry          [TkConfItem a]
   | TkLabel          [TkConfItem a]
   | TkScale Int Int  [TkConfItem a]
   | TkTextEdit       [TkConfItem a]
   ⋮
   | TkRow [TkWidget a]
   | TkCol [TkWidget a]
```


Example: A Counter GUI

A specification of a counter GUI:




```
TkCol
  [TkEntry [TkRef val, TkText "0"],
   TkRow [TkButton (tkUpdate incr  val) [TkText "Increment"],
          TkButton (tkSetValue val "0") [TkText "Reset"],
          TkButton tkExit               [TkText "Stop"]]]
where val free
```

- ➜ the free variable `val` is a reference to the entry widget
- ➜ `val` is used in the event handlers of other widgets
- ➜ `val` is part of the logical structure of the GUI

Logical Structure of GUIs

Configuration options for GUIs:

```
data TkConfItem a =
   TkText String            -- initial text
 | TkBackground String      -- background color
 | TkRef TkRefType          -- widget reference
 | TkCmd (GUIRef -> a)      -- event handler
 ⋮
```

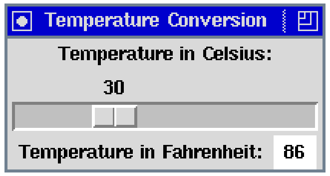
`TkRef`: reference to a widget, used in event handlers
(`TkRefType` is abstract ⇝ argument is a logical variable)

```
tkExit     :: GUIRef -> IO ()
tkGetValue :: TkRefType -> GUIRef -> IO String
tkSetValue :: TkRefType -> String -> GUIRef -> IO ()
tkUpdate   :: (String->String) -> TkRefType -> GUIRef -> IO ()
```

Remark: event handlers also available as constraints

Example: Temperature Converter

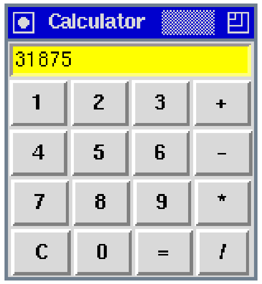
Convert a temperature from Celsius into Fahrenheit:




```
TkCol [TkLabel [TkText "Temperature in Celsius:"],
       TkScale 0 100 [TkRef cels, TkCmd convert],
       TkRow [TkLabel [TkText "Temperature in Fahrenheit: "],
              TkMessage [TkRef fahr, TkBackground "white"]]]
where cels,fahr free

 convert gr =
    tkGetValue cels gr >>= \cs ->
    tkSetValue fahr (show ((parseInt cs)*9 `div` 5+32)) gr
```

GUIs with State: A Desk Calculator



Implementation consists of two parts:

1. Object for storing the state
   state: (operand, accumulator function)
   messages: `Display s`, `Button b`
2. GUI for showing the state

Object for storing the state:

Message `Display s`: instantiate `s` with current display

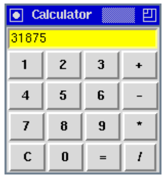
```
calcMgr (d,f) (Display s : ms) = s =:= (show d) &>
                                 calcMgr (d,f) ms
```

Message `Button b`: the user has pressed button `b`

```
calcMgr (d,f) (Button b : ms)
 | isDigit b = calcMgr (10*d + ord b - ord '0', f) ms
 | b=='+'    = calcMgr (0, ((f d) +)) ms
 | b=='-'    = calcMgr (0, ((f d) -)) ms
 | b=='*'    = calcMgr (0, ((f d) *)) ms
 | b=='/'    = calcMgr (0, ((f d) `div`)) ms
 | b=='='    = calcMgr (f d, id) ms
 | b=='C'    = calcMgr (0, id) ms
```


GUI for showing the state with a reference `cm` to calculator object:

```
calc_GUI cm = TkCol [TkEntry [TkRef display, TkText "0"],
                     TkRow (map cbutton ['1','2','3','+']),
                     TkRow (map cbutton ['4','5','6','-']),
                     TkRow (map cbutton ['7','8','9','*']),
                     TkRow (map cbutton ['C','0','=','/'])]
```




```
 where display free

   cbutton b = TkButton (click b) [TkText [b]]

   click b gr = let d free in send (Button  b) cm &>
                              send (Display d) cm &>
                              tkCSetValue display d gr
```

➜ model-view-controller paradigm à la Smalltalk-80

➜ different (distributed) views on one application

Functional Logic GUI Programming: Summary

Functional features useful for

➜ layout specification

➜ event handlers (data structures with functional components)

➜ application-oriented extensions

Logic programming features useful for

➜ dealing with dependencies inside a structure (free variables)

➜ handling state (concurrent objects)

Distributed features $\leadsto$ GUIs for distributed applications

Specification (rather than imperative programming) of GUIs

Domain-specific language for GUIs, but:

no extension to base language necessary

Functional Logic Programming of Parsers

[Caballero/Lopez-Fraguas FLOPS'99]

Logic programming of parsers:

- ➜ nonterminals consume corresponding tokens (difference lists)
- ➜ definite clause grammars for nice notation
- ➜ non-deterministic grammars/parsing
- ➜ resulting representations as arguments

Functional programming of parsers:

- ➜ parsers consume corresponding tokens
- ➜ powerful parser combinators
- ➜ more complex handling of alternatives and representations

Functional logic programming of parsers:

simpler handling of representations and alternatives due to

- ➜ non-deterministic functions
- ➜ free variables as arguments

Parser ≈ function of type `[token] -> [token]`
Argument: list of tokens to be parsed
Result: list of remaining unparsed tokens

A parser recognizing token `'a'`:

```
parse_a ('a':ts) = ts
```

A parser recognizing a given token:

```
terminal sym (t:ts) | sym=:=t = ts
```

Parser recognizing the empty word:

```
empty sentence = sentence
```


Parser Combinators

Parser combinators: higher-order functions to combine parsers

Alternative of two parsers `p` and `q`: combinator `p <|> q`

```
(p <|> q) sentence = p sentence
(p <|> q) sentence = q sentence
```

Sequence of two parsers `p` and `q`: combinator `p <*> q`

```
(p1 <*> p2) s0 | p1 s0 =:= s1  = p2 s1   where s1 free
```

Repetition of a parser: (zero or more times)

```
star p = (p <*> star p) <|> empty
```

Parser for $a(a|b)^*$:

```
terminal 'a' <*> star (terminal 'a' <|> terminal 'b')
```


Example: Parsing Palindromes

A parser for palindromes over the alphabet $\{a, b\}$

```
pali = empty <|> a <|> b <|> a<*>pali<*>a <|> b<*>pali<*>b

a = terminal 'a'
b = terminal 'b'
```

Checking a sentence for a palindrome:

```
pali "abaaba" =:= []
```

Using logic programming features, we can also generate palindromes:

```
pali [x,y,z] =:= []
⇝  {x='a',y='a',z='a'} | {x='a',y='b',z='a'}
  | {x='b',y='a',z='b'} | {x='b',y='b',z='b'}
```

Parsers with Representations

Parsers should not only check a list of tokens but also return a representation (e.g., abstract syntax tree)

- Functional programming: parsers have result $(rep, tokens)$
- Logic programming: parsers have rep argument $\leadsto$ simpler definitions

Parser with representation $\approx$ `rep -> [token] -> [token]`

Representation argument:

- usually free variable
- will be instantiated during parsing

Parser Combinators with Representations

Alternative of two parsers `p` and `q`: combinator `p <||> q`

```
(p <||> q) rep = p rep <|> q rep
```

(reuse combinator for parsers without representation)

Attach representation `exp` to a parser `p`: combinator `p >>> exp`

```
(p >>> exp) rep s_in | p s_in =:= s_out & exp =:= rep   = s_out
                                                          where s_out free
```

Repetition of a parser with representation: (representation is list)

```
star p =    p r <*> (star p) rs   >>> (r:rs)
      <||> empty                  >>> []      where r,rs free
```

At least one repetition of a parser:

```
some p  =  p r <*> star p rs >>> (r:rs)    where r,rs free
```


Example: Parser for Arithmetic Expressions

```
expr   =  term t <*> plus_minus op <*> expr e      >>> (op t e)
     <||> term

term   =  factor f <*> prod_div op <*> term t      >>> (op f t)
     <||> factor

factor =  terminal '(' <*> expr e <*> terminal ')' >>> e
     <||> num

plus_minus  =  terminal '+'  >>> (+)
          <||> terminal '-'  >>> (-)

prod_div    =  terminal '*'  >>> (*)
          <||> terminal '/'  >>> div

num = some digit l >>> numeric_value l
```

Example: `expr val "(10+5*2)/4" =:= []` $\leadsto$ `{val=5}`

Functional Logic Parsing: Summary

Higher-order features useful for

- combining parsers (parsers are functions)
- computing representations

Logic programming features useful for

- dealing with alternatives (non-deterministic functions)
- managing representations (free variables in arguments)
- parsing with constraints

Domain-specific language for parsing, but:

no extension to base language necessary

Application: HTML/CGI Programming

Early days of the World Wide Web: web pages with static contents

Common Gateway Interface (CGI): web pages with dynamic contents

Retrieval of a dynamic page:

- ➜ server executes a program
- ➜ program computes an HTML string, writes it to stdout
- ➜ server sends result back to client

HTML with input elements (forms):

- ➜ client fills out input elements
- ➜ input values are sent to server
- ➜ server program decodes input values for computing its answer

Traditional CGI Programming

CGI programs on the server can be written in any programming language

- ➜ access to environment variables (for input values)
- ➜ writes a string to stdout

Scripting languages: (Perl, Tk,. . .)

- ➜ simple programming of single pages
- ➜ error-prone: correctness of HTML result not ensured
- ➜ difficult programming of interaction sequences

Specialized languages: (MAWL, DynDoc,. . .)

- ➜ HTML support (structure checking)
- ➜ interaction support (partially)
- ➜ restricted or connection to existing languages

CGI Programming in a Multi-Paradigm Language

Library in multi-paradigm language

Exploit functional and logic features for

- ➜ HTML support (data type for HTML structures)
- ➜ simple access to input values (free variables and environments)
- ➜ simple programming of interactions (event handlers)
- ➜ wrapper for hiding details

Exploit imperative features for

- ➜ environment access (files, data bases,. . .)

Domain-specific language for HTML/CGI programming

Modeling HTML

Data type for representing HTML expressions:

```
data HtmlExp = HText String
             | HStruct String [(String,String)] [HtmlExp]
```

Some useful abbreviations:

```
htxt   s     = HText (htmlQuote s)       -- plain string
bold   hexps = HStruct "B"  [] hexps     -- bold font
italic hexps = HStruct "I"  [] hexps     -- italic font
h1     hexps = HStruct "H1" [] hexps     -- main header
...
```

Example: `[h1 [htxt "1. Hello World"],`
`italic [htxt "Hello"], bold [htxt "world!"]]`

⇝ **1. Hello World**

Hello **world!**

Advantages:

- ➜ static checking of HTML structure (well-balanced parentheses)
- ➜ flexible dynamic documents
- ➜ functions for computing HTML documents

Converting tree structure (leaves contain strings) into nested HTML lists:

```
data Tree a = Leaf a | Node [Tree a]

htmlTree :: Tree String -> [HtmlExp]
htmlTree (Leaf s)     = [htxt s]
htmlTree (Node trees) = [ulist (map htmlTree trees)]

ulist       :: [[HtmlExp]] -> HtmlExp
ulist items = HStruct "UL" [] (map litem items)

litem hexps = HStruct "LI" [] hexps
```


HTML INPUT FORMS

Specific HTML elements for dealing with user input

```
<INPUT TYPE="TEXT" NAME="INPTEXT" VALUE="fill out!">
```

Form is submitted $\leadsto$
clients sends the current value of this field (identified by `"INPTEXT"`)

Expressible as HTML term:

```
HStruct "INPUT" [("TYPE","TEXT"),("NAME","INPTEXT"),
                 ("VALUE","fill out!")] []
```

Problems:

- ➜ server program must decode input values
- ➜ server program must know right names of field identifiers (`"INPTEXT"`)
- ➜ error-prone

ABSTRACT INPUT FORMS

Solution:

- ➜ use free variables as references to input fields (CGI references)
- ➜ collect input values in CGI environments:
  mapping from CGI references to strings
- ➜ associate event handlers to submit buttons
- ➜ event handlers take a CGI environment and produces an HTML form

Implementation:

straightforward in a functional logic language!

ABSTRACT INPUT FORMS: IMPLEMENTATION

CGI references:

```
data CgiRef = CgiRef String   -- data constructor not exported
```

- ➜ no construction of wrong references
- ➜ only free variables of type `CgiRef`
- ➜ global wrapper function instantiates with the right strings

HTML elements with CGI references:

```
data HtmlExp = ... | HtmlCRef HtmlExp CgiRef
```

Example: Text fields with a CGI reference and initial contents

```
textfield :: CgiRef -> String -> HtmlExp
textfield (CgiRef ref) contents =
 HtmlCRef (HStruct "INPUT" [("TYPE","TEXT"),
                            ("NAME",ref),("VALUE",contents)])
          (CgiRef ref)
```

HTML form: title + list of HTML expressions

```
data HtmlForm = Form String [HtmlExp]
```

Example: simple form with a single input element (a text field)

```
Form "Form" [h1 [htxt "A Simple Form"],
             htxt "Enter a string:", textfield sref ""]
```

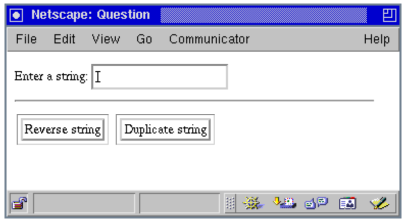
CGI environments: map CGI references to strings

```
type CgiEnv = CgiRef -> String
```

Event handlers have type `CgiEnv -> IO Form`

Event handlers are associated to submit buttons:
user presses a submit button
⇝ execute associated event handler with current environment

EXAMPLE: FORM TO REVERSE/DUPLICATE A STRING




```
Form "Question" [htxt "Enter a string: ", textfield tref "", hr,
                 button "Reverse string"   revhandler,
                 button "Duplicate string" duphandler]
where tref free

 revhandler env = return $ Form "Answer"
        [h1 [htxt ("Reversed input: " ++ rev (env tref))]]

 duphandler env = return $ Form "Answer"
        [h1 [htxt ("Duplicated input: " ++ env tref ++ env tref)]]
```


ACCESSING THE WEB SERVER ENVIRONMENT

Form to show the contents of an arbitrary file stored at the server:

```
Form "Get File" [htxt "Enter local file name:",
                 textfield fileref "",
                 button "Get file!" handler]

where fileref free

 handler env =
   do contents <- readFile (env fileref)
      return $ Form "Answer"
                 [h1 [htxt ("Contents of file " ++ env fileref)],
                  verbatim contents]
```


HTML/CGI PROGRAMMING

The main form is executed by a wrapper function

```
runcgi :: String -> IO HtmlForm -> IO ()
```

➜ takes a title string and a form and transforms it into HTML text
➜ replaces all CGI references by unique strings
➜ decodes input values and invoke associated event handler

Event handlers return forms rather than HTML expressions

➜ sequences of interactions
➜ use control abstractions (branching, recursion) of underlying language
➜ state between interactions handled by CGI environments

Note: no language extension necessary (CGI library)
multi-paradigm languages as scripting languages

A Few Further Multi-Paradigm Languages

Erlang (Ericsson)

- ➜ developed by Ericsson for telecommunication applications
- ➜ concurrent functional language with features to support the development of robust distributed systems
- ➜ reduced development time and maintainance

Escher (University of Bristol)

- ➜ extension of Haskell by features for logic programming
- ➜ functions are evaluated by residuation
- ➜ explicit disjunctions for logic programming
- ➜ simplification rules for logic formulas

Mercury (University of Melbourne)

- ➜ logic/functional language with highly optimized execution algorithm
- ➜ origin: logic programming (syntax) with type/mode/determinism annotations
- ➜ adapted concepts from functional programming, strict semantics

Oz (DFKI Saarbrücken)

- ➜ concurrent constraint language with features for higher-order functional, object-oriented, and distributed programming
- ➜ operational behavior: residuation
- ➜ search via explicit disjunctions and search operators

Toy (Univ. Complutense de Madrid)

- ➜ prototype for a functional logic language
- ➜ based on lazy narrowing, supports non-deterministic functions
- ➜ contraints, in particular, disequality constraints

. . . and, of course, there are many, many more. . .

Conclusions

Appropriate abstractions are important for software development and maintainance

Multi-paradigm languages have the potential to express these abstractions

High-level languages support domain-specific languages

Multi-paradigm programming

- ➜ possible and advantageous
- ➜ constraint functional logic programming: many improvements in recent years
- ➜ imperative/concurrent/distributed + declarative programming: possible but many different approaches

More infos on Curry:

`http://www-i2.informatik.rwth-aachen.de/~hanus/curry`