Generation of Interactive Programming Environments

Arie van Deursen
CWI, Amsterdam

Note: Supported by E.C., project ESPRIT 2177

Presentation of work of the GIPE group, CWI, Amsterdam

GIPE: Generation of Interactive Programming Environments

Started with two people in 1984; by now grown to at about ten people.

Two more years to go; After that continuation in new project

Presentation of some of the most interesting results of this work.

Programming environments with knowledge of programming language will help the programmer.

Environments for different languages will have many aspects in common.

Given a *description* of the programming language *generate* a programming environment.

Advantages: Reduced environment construction time; Intra- and Inter-environmental uniformity.

Investigation of aspects of environment generation; construction of prototype generator.

Generation of programming environments:

- Give an algebraic specification of all relevant aspects of language (syntax, static semantics, dynamic semantics, transformations, etc.)

- Generate tools (parser, type checker, interpreter, optimizer, etc.) from each specification.

- Integrate generated tools into environment by a description of layout and behavior of user interface

- Give support to specification phase by *meta-environment*.

Contents:

Algebraic Specification of the Booleans:

```
sorts
  BOOL
functions
  true:                          -> BOOL
  false:                         -> BOOL
  and:     BOOL # BOOL           -> BOOL
  or:      BOOL # BOOL           -> BOOL
variables
  p, q:                          -> BOOL
equations
  [1]      and(true, p) = p
  [2]      and(false,p) = false
  [3]      or(true,  p) = true
  [4]      or(false, p) = p
```

ASF: Algebraic Specification Formalism

SDF: Syntax Definition Formalism

Combined, ASF+SDF: specifications with free syntax.

```
imports Layout

exports
  sorts BOOL
  context-free syntax
    true                      -> BOOL
    false                     -> BOOL
    BOOL and BOOL             -> BOOL {left}
    BOOL or  BOOL             -> BOOL {left}
  variables
    p[']*                     -> BOOL
priorities
    and   >  or
equations
    [1]   true  and p = p
    [2]   false and p = false
    [3]   true  or  p = true
    [4]   false or  p = p
```

```
module TC

imports Booleans Types

exports
  sorts PROGRAM DECLS STAT EXP ...
  context-free syntax
    ....
    program DECLS begin STAT+ end          -> PROGRAM
    if EXP then STAT fi                    -> STAT

    ....
    tc "(" STAT "," DECLS ")"              -> BOOL
    tc "(" EXP  "," DECLS ")"              -> TYPE
    ....
  variables
    Exp   -> EXP
    Stat  -> STAT
    Decls -> DECLS
    .....

equations

  .....

  [7]              tc( Exp, Decls ) = bool-type
       ================================================
       tc( if Exp then Stat fi, Decls ) = tc(Stat,Decls)

  .....
```

Derive from each module:

- A *parser* for the signature part producing abstract syntax trees.

  From a rule `BOOL "and" BOOL -> BOOL` derive:

  - BNF rule
    $\langle BOOL \rangle ::= \langle BOOL \rangle$ "**and**" $\langle BOOL \rangle$

  - Abstract function
    $and : BOOL \times BOOL \rightarrow BOOL$

- A *Term Rewriting System* able to perform reductions by interpreting each equation $t_1 = t_2$ as a rewrite rule $t_1 \rightarrow t_2$.

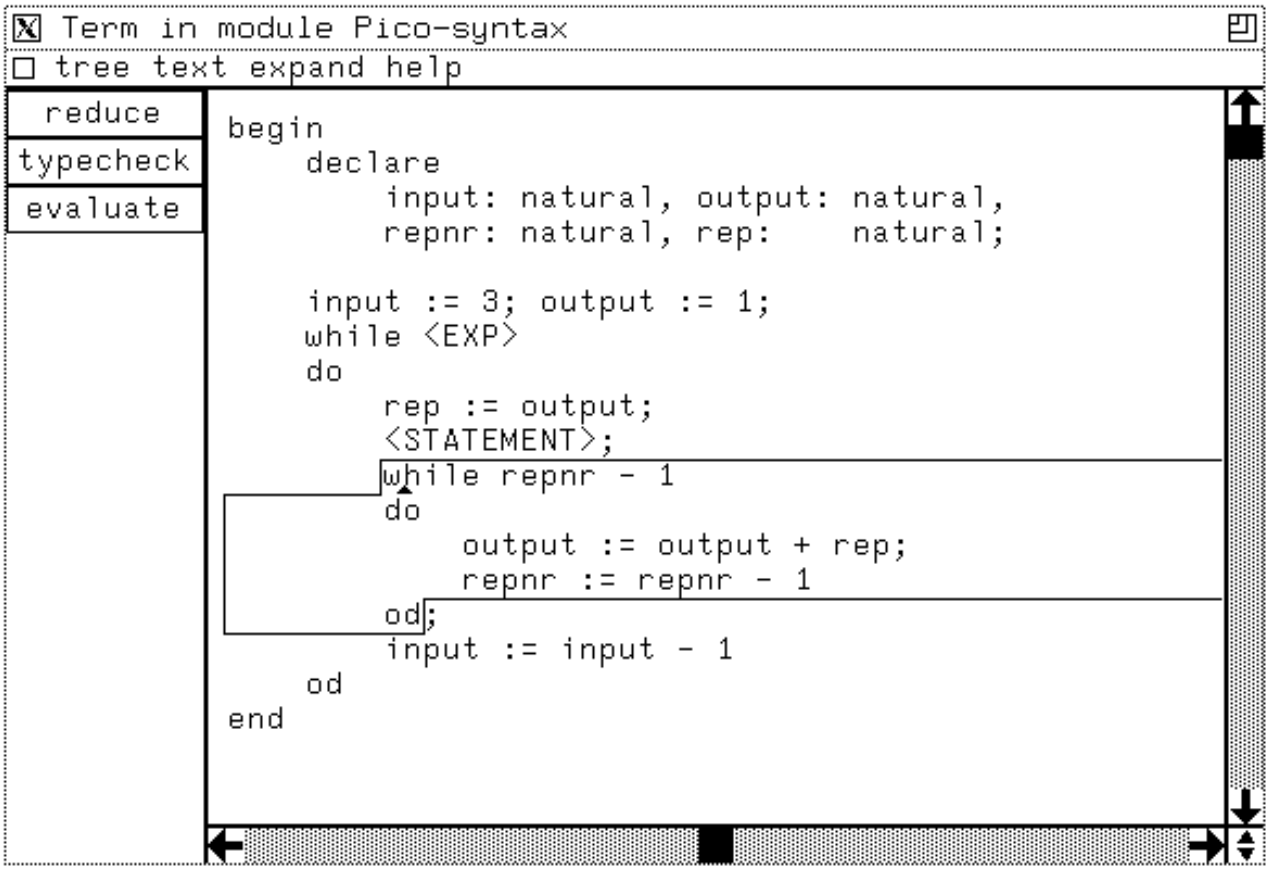
Use parser for syntax-directed editing, and TRS for execution of specified functions.

Given a grammar, a *syntax-directed editor* can be derived.

- Plain *text*-editing within a *focus*.

  A focus designates a subtree in the tree obtained by parsing the complete text in the editor

- *Structural*-editing allows to move the focus around, and to *expand* nonterminal symbols.


Term in module Pico-syntax
tree text expand help
reduce
typecheck
evaluate
begin
declare
input: natural, output: natural,
repnr: natural, rep: natural;
input := 3; output := 1;
while <EXP>
do
rep := output;
<STATEMENT>;
while repnr - 1
do
output := output + rep;
repnr := repnr - 1
od;
input := input - 1
od
end

Algebraically specified functions can be linked to editors by *buttons*.

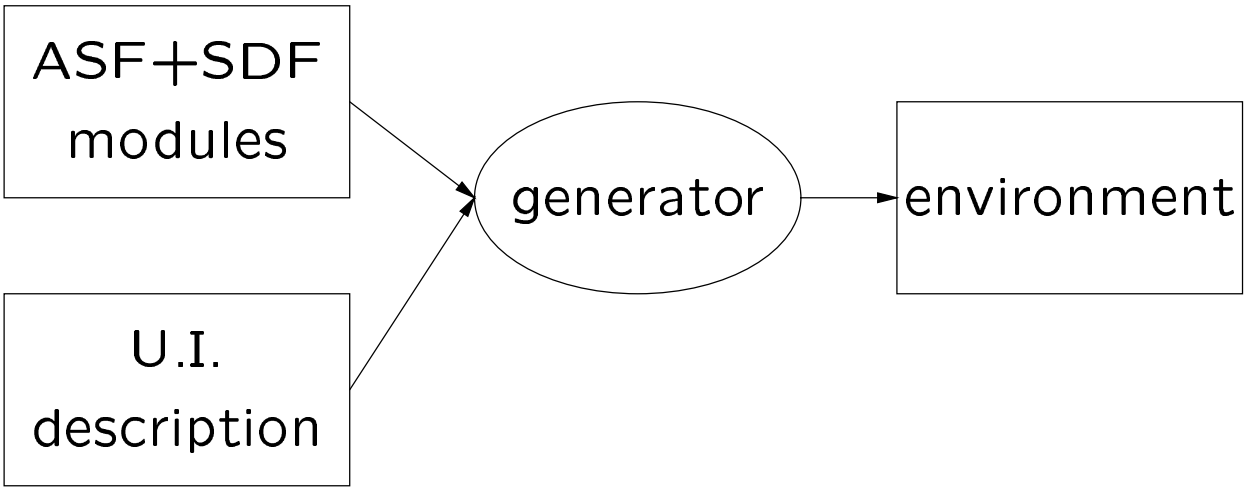
The action for a button typically involves rewriting a specified function with focusses of one (or more) editor(s) as arguments.

The action may replace the current focus, or may retrieve arguments from several editors.

A user interface description language defines precisely which buttons and actions are attached to each editor.

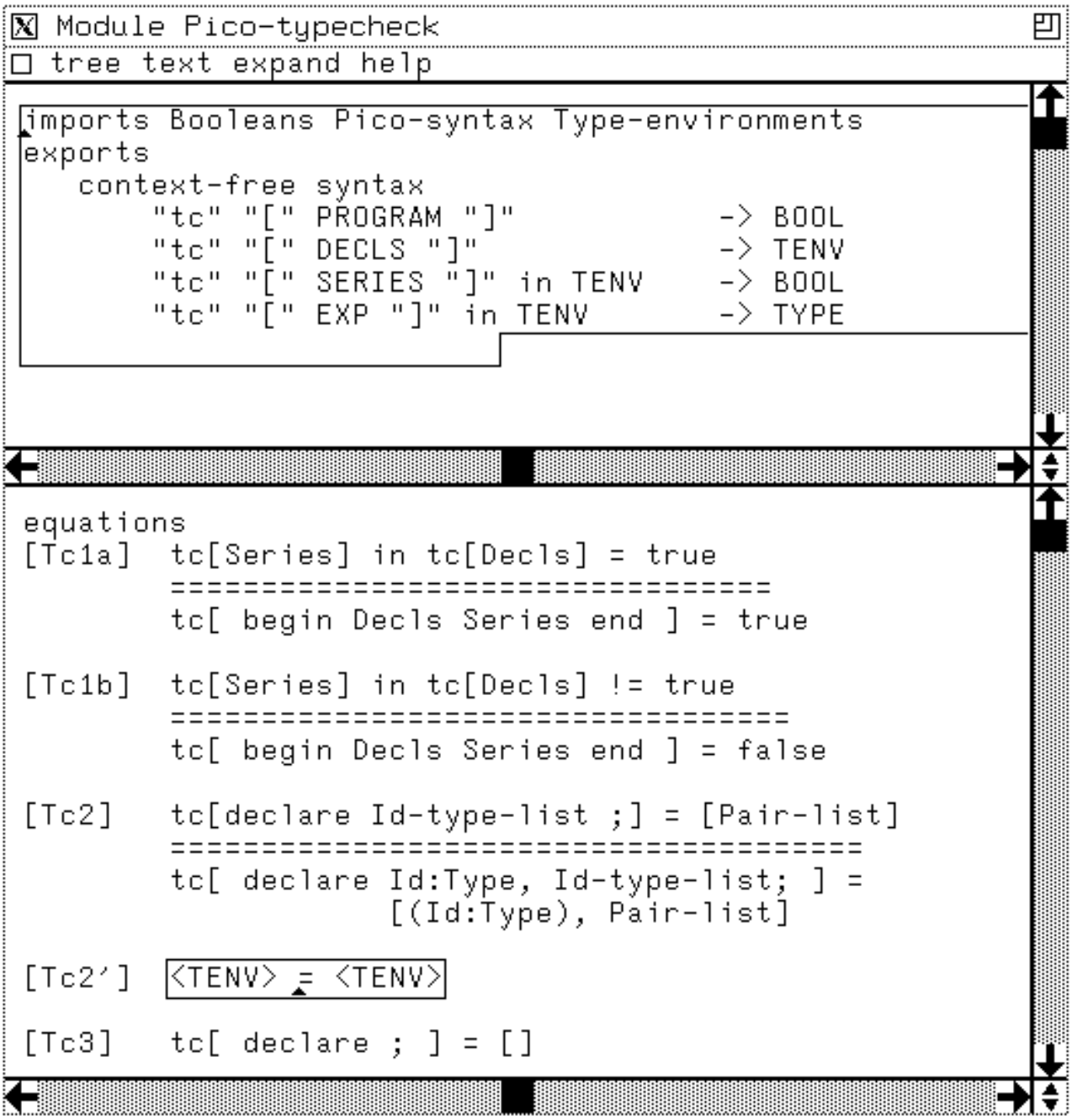
Several editors with all kinds of buttons attached to them constitute a generated environment.

A generated environment consists of editors parameterized with a generated parser, with buttons attached to them performing rewrites in the generated term rewriting system.



The *meta*-environment:

- Syntax-directed editing of specifications.
- Immediate static analysis
- Incremental updates of generated parsers, scanners, and term rewriting systems.
- Generated environment can be tested simultaneously.


Module Pico-typecheck
tree text expand help
imports Booleans Pico-syntax Type-environments
exports
context-free syntax
"tc" "[" PROGRAM "]" -> BOOL
"tc" "[" DECLS "]" -> TENV
"tc" "[" SERIES "]" in TENV -> BOOL
"tc" "[" EXP "]" in TENV -> TYPE
equations
[Tc1a] tc[Series] in tc[Decls] = true
tc[begin Decls Series end] = true
[Tc1b] tc[Series] in tc[Decls] != true
tc[begin Decls Series end] = false
[Tc2] tc[declare Id-type-list ;] = [Pair-list]
tc[declare Id:Type, Id-type-list;] =
[(Id:Type), Pair-list]
[Tc2'] <TENV> = <TENV>
[Tc3] tc[declare ;] = []

The GIPE project thus far:

- Results: Algebraic specification formalism with free syntax, incremental generation of parsers, scanners and term rewriting systems, incremental term rewriting, generic syntax-directed editors

- Testing specifications (and environments) for several toy languages, the λ-calculus, parts of the system, SQL, static semantics of Pascal and subset of ML, LOTOS

- Future work: Automatic error handling, generic debuggers, origin tracking, C-code generation, dynamic syntax, fine-grain incremental rewriting, bootstrapping, user interface description

An example: specification and generation of a λ-calculus environment.

λ-calculus; important for functional programming, denotational semantics, etc.

λ-expressions, α, β, η conversion, left-most reductions, λ-definitions like **zero** $\equiv \lambda fx.x$, or **succ** $\equiv \lambda nfx.nf(fx)$

Environment should be suited to play around with new λ-definitions

```
module Lambda-syntax

imports Identifiers renamed by sorts ID => VAR

exports
  sorts L-EXP
    context-free syntax
      VAR                          -> L-EXP
      "lambda" VAR+ "." L-EXP      -> L-EXP
      L-EXP L-EXP                  -> L-EXP  {left}
      "(" L-EXP ")"                -> L-EXP  {bracket}
    variables
      E[0-9']*                     -> L-EXP
      V[0-9']*                     -> VAR
      V[0-9']*"+"                  -> VAR+

priorities
  { "lambda" VAR+ "." L-EXP  -> L-EXP }  <
    { L-EXP L-EXP -> L-EXP }

equations
  [1]  lambda V+ V . E = lambda V+ . lambda V . E
```

```
module Substitute

imports Booleans  Lambda-syntax
  Sets  parameter P bound by sorts ELM => VAR to Lambda-syntax
        renamed by sorts SET => VAR-SET

exports
  context-free syntax
    L-EXP "[" L-EXP "/" VAR "]"                  -> L-EXP
    free-vars "(" L-EXP ")"                      -> VAR-SET
    get-fresh "(" VAR "," VAR-SET ")"            -> VAR

equations
  [s1]  V  [E/V] = E

  [s2]  V' [E/V] = V' when V != V'

  [s3]  (E1 E2) [E/V] =  (E1[E/V]) (E2[E/V])

  [s4]  (lambda V . E1)  [E/V] = lambda V . E1

  [s5]  (lambda V' . E1) [E/V] = lambda V' . (E1[E/V])
           when V != V', member-of?(V', free-vars(E)) = false

  [s6]  (lambda V' . E1) [E/V] = lambda V'' . ( E1[V''/V'][E/V] )
           when V != V',  member-of?(V', free-vars(E)) = true,
                get-fresh(V', (E E1)) = V''


  [f1]  free-vars(V) = [V]
  [f2]  free-vars(E1 E2) = free-vars(E1) U free-vars(E2)
  [f3]  free-vars(lambda V . E) = free-vars(E) - V


  [g1]  get-fresh(V, E) = get-fresh(prime(V), E)
           when  member-of?(V, free-vars(E)) = true
  [g2]  otherwise: get-fresh(V, E) = V
```

```
module Convert

imports Substitute

exports
  context-free syntax
    alpha "(" L-EXP ")"                  -> L-EXP
    beta "(" L-EXP ")"                   -> L-EXP
    eta  "(" L-EXP ")"                   -> L-EXP

equations
  [b1]  beta( (lambda V . E1) E2 ) = E1 [ E2/V ]

  [a1]  alpha( lambda V1 . E, V2 ) = lambda V2 . (E[V2/V1])
          when  member-of?(V2, free-vars(E)) = false

  [e1]  eta( lambda V . E V ) = E
           when member-of?(V, free-vars(E)) = false
```

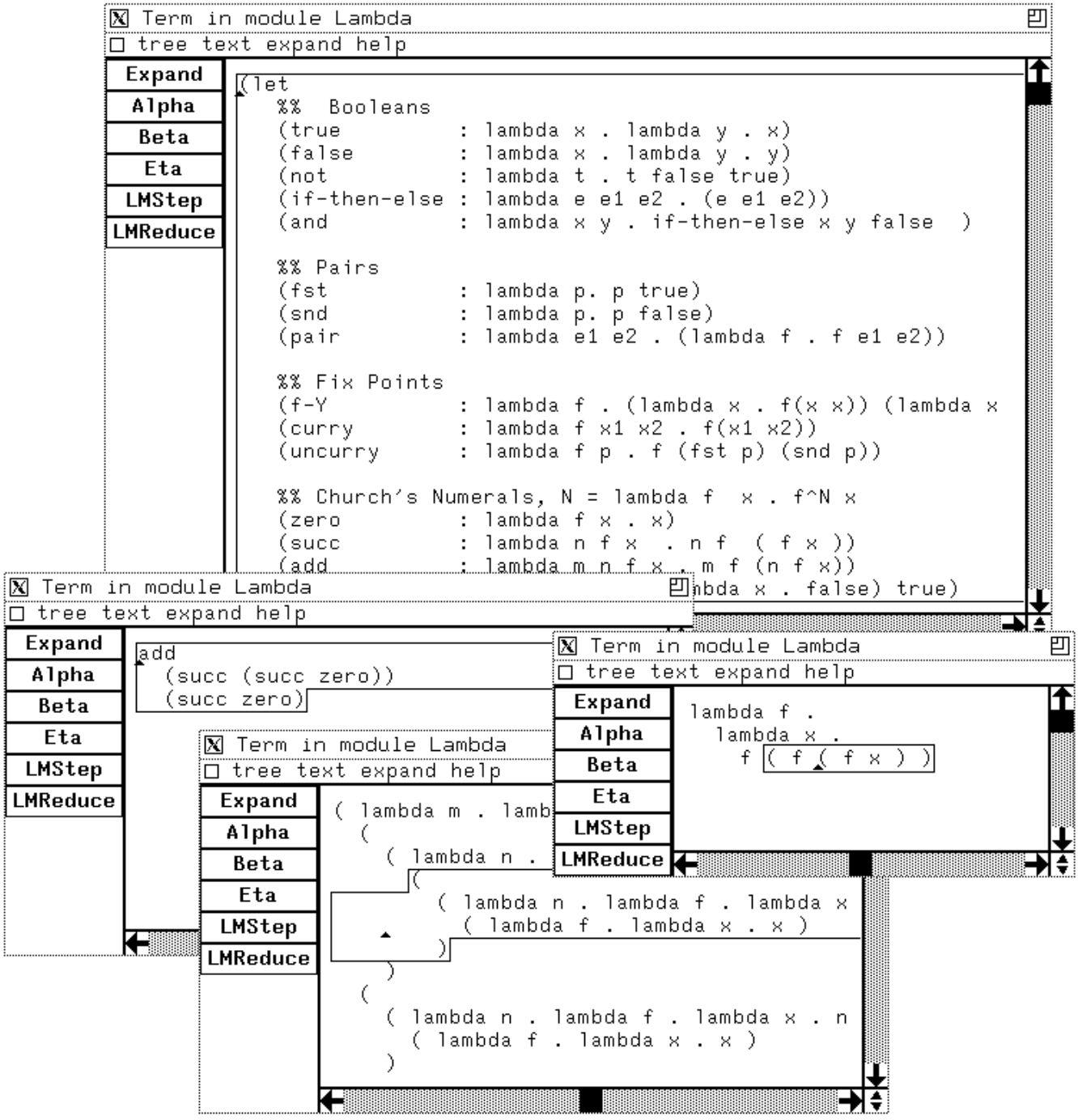
```
module Let

imports Lambda-syntax Substitute

exports
  sorts DEF LET
  context-free syntax
    "expand" "(" L-EXP "," LET ")"      -> L-EXP
    "(" VAR ":" L-EXP ")"               -> DEF
    "(" "let" DEF+ ")"                  -> LET
                                        -> LET

  variables
     D[0-9']*"+"          -> DEF+
     D[0-9']*             -> DEF

equations
  [e0]  expand(E, ) = E
  [e1]  expand(E, (let (V:E'))) = E[E'/V]
  [e2]  expand(E, (let D+ D)) =
          expand(expand(E, (let D)), (let D+))
```


Term in module Lambda
tree text expand help
Expand
Alpha
Beta
Eta
LMStep
LMReduce
(let
%% Booleans
(true : lambda x . lambda y . x)
(false : lambda x . lambda y . y)
(not : lambda t . t false true)
(if-then-else : lambda e e1 e2 . (e e1 e2))
(and : lambda x y . if-then-else x y false)
%% Pairs
(fst : lambda p. p true)
(snd : lambda p. p false)
(pair : lambda e1 e2 . (lambda f . f e1 e2))
%% Fix Points
(f-Y : lambda f . (lambda x . f(x x)) (lambda x
(curry : lambda f x1 x2 . f(x1 x2))
(uncurry : lambda f p . f (fst p) (snd p))
%% Church's Numerals, N = lambda f x . f^N x
(zero : lambda f x . x)
(succ : lambda n f x . n f (f x))
(add : lambda m n f x . m f (n f x))
nbda x . false) true)
add
(succ (succ zero))
(succ zero)
lambda f .
lambda x .
f (f (f x))
(lambda m . lamb
(
(lambda n .
(lambda n . lambda f . lambda x
(lambda f . lambda x . x)
)
(
(lambda n . lambda f . lambda x . n
(lambda f . lambda x . x)
)

Button	Functionality
Alpha	`alpha(` *Current-focus*)
Beta	`beta(` *Current-focus*)
Eta	`eta(` *Current-focus*)
LMStep	`lm-step(` *Current-focus*)
LMReduce	`lm-red(` *Current-focus*)
Expand	`expand(` *Current-focus* , *Defs-focus*)

Concluding Remarks

- Formalism for algebraic specification with user-defined syntax
- Environment Generator
- Meta-environment helping when developing specifications
- Source of inspiration for new ideas