

NPS52-86-007

NAVAL POSTGRADUATE SCHOOL

Monterey, California



EXPERIENCE WITH *Omega*.

IMPLEMENTATION OF A
PROTOTYPE PROGRAMMING ENVIRONMENT,
PART IV.

Bruce J. MacLennan

January 1986

Approved for public release; distribution unlimited

Prepared for:

Chief of Naval Research
Arlington, VA 22217

20091105008

NAVAL POSTGRADUATE SCHOOL
Monterey, California

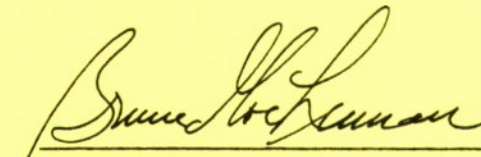
Rear Admiral R. H. Shumaker
Superintendent

D. A. Schradly
Provost

The work reported herein was supported by Contract from the
Office of Naval Research.

Reproduction of all or part of this report is authorized.

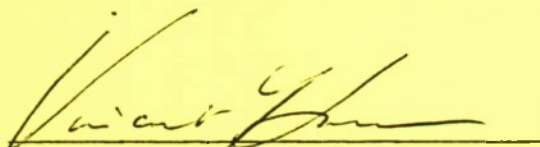
This report was prepared by:



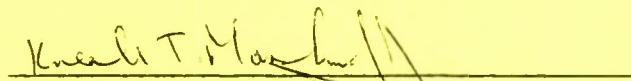
Bruce J. MacLennan
Associate Professor
Computer Science

Reviewed by:

Released by:



VINCENT Y. LUM
Chairman
Department of Computer Science



KNEALE T. MARSHALL
Dean of Information and
Policy Science

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER NPS52-86-007	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) EXPERIENCE WITH Ω IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT PART IV		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Bruce J. MacLennan		8. CONTRACT OR GRANT NUMBER(s) N00014-86-WR-24092
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93943-5100		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Chief of Naval Research Arlington, VA 22217		12. REPORT DATE January 1986
		13. NUMBER OF PAGES 52
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report)
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number)		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This is the fourth report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report extends the interpreter, unparser, syntax directed editor, command interpreter and debugger to accommodate recursive function definition and invocation, and completes the extension of the language into an applicative programming system supporting higher-order functions. An implementation of		

DD FORM 1473
1 JAN 73

EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102- LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

these ideas is listed in the appendices.

S/N 0102-LF-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

EXPERIENCE WITH Ω
IMPLEMENTATION OF A
PROTOTYPE PROGRAMMING ENVIRONMENT
PART IV

Bruce J. MacLennan
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract:

This is the fourth report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report extends the interpreter, unparser, syntax directed editor, command interpreter and debugger to accommodate recursive function definition and invocation, and completes the extension of the language into an applicative programming system supporting higher-order functions. An implementation of these ideas is listed in the appendices.

1. Introduction

Our goal in this series of reports* [MacLennan85b, MacLennan85c, MacLennan86] is to explore in the context of a very simple language the use of the Ω programming notation [MacLennan83, MacLennan85a] to implement some of the tools that constitute a programming environment.

The structure of this report is as follows: First we outline the requirements for the function definition facility. Next we define the abstract structure of function definitions and invocations. We proceed to the dynamic structures required to support recursive, statically scoped procedures. This leads naturally to the topic of evaluation. We finish by discussing possible debugger support for the new facilities. As in previous reports, a running system demonstrating these ideas is listed in the appendices.

* Support for this research was provided by the Office of Naval Research under contract N00014-86-WR-24092.

2. Goal

We want to permit the definition and invocation of statically scoped recursive functions. For example, the following program defines factorial recursively and invokes the resulting definition with argument $K = 4$:

```
[ func fac n =  
  (if (n= 0)  
    then 1  
    else (n×fac (n-1)) )  
  
[ let K = 4  
  fac K ] ]
```

It's easy to see that the general form of a function definition is:

```
[ func F N = B  
  X ]
```

For simplicity we restrict our attention to monadic functions.

3. Abstract Structure

3.1 Function Definition

The abstract structure of a function definition block is represented in a straight-forward way as a node with four descendents, corresponding to the function name, formal parameter, function body and block body. These are defined by the following declarations:

- FunDef (E)

E is a function definition

Degree (FunDef, 1).

- FunName (F, E)

F is the function name of E

Function (FunName, FunDef, string).

- FunFormal (N, E)

N is the formal of E

Function (FunFormal, FunDef, string).

- FunBody (B, E)

B is the body of E

Function (FunBody, FunDef, expr).

- FunScope (X, E)

X is the scope of E

Function (FunScope, FunDef, expr).

Note that, for convenience (and consistency with let blocks) the FunName and FunFormal attributes are strings, rather than variable nodes. This complicates editing and is probably, in the long run, a bad decision. The problem is solved in Part VI, where table-driven syntax-directed editing is discussed.

3.2 Function Invocation

The abstract syntax of function invocations is straight-forward. Note that the function is allowed to be an arbitrary expression, which (as we'll see later) goes through the usual evaluation process. This, in conjunction with the representation of closures, permits general functional programming. The abstract structure is represented by the relations:

- Call (E)

E is a call

Degree (Call, 1).

- Rator (F, E)

F is the operator of E

Function (Rator, Call, Var).

- Rand (X, E)

X is the operand of E

Function (Rand, Call, expr).

4. Dynamic Structures

4.1 Closures

Recall that in statically scoped languages a function executes in its *environment of definition* rather than its *environment of call*. Thus, when a function binding is made, it is necessary to record the function's environment of definition. This is done by binding the function's name to a *closure* object.

A closure has three parts:

1. EP: environment part (environment of definition)
2. IP: instruction part (body of function)
3. FP: formal parameter

The abstract structure of closures is represented by the following relations:

- Closure (K)

K is a closure

Degree (Closure, 1).

- EP (C, K)

C is the environment part of K

Function (EP, Closure, Context).

- IP (B, K)

B is instruction part of K

Function (IP, Closure, expr).

- FP (N, K)

N is formal parameter of K

Function (FP, Closure, string).

4.2 Dynamic Link

In addition to the closure, which determines the environment in which a function executes, it is also necessary to determine the *caller*, within whom's execution the execution of the *callee* is dynamically

nested. This is called the *dynamic link* of the current context, and is represented by the relation:

- Caller (E, C, B, A)
- E in C is caller of B in A
- Function (Caller, $\text{expr} \times \text{Context}$, $\text{expr} \times \text{Context}$).

Thus, the Caller relation refers back from the callee's expression/context (IP/EP) pair to the caller's expression/context pair.

Why do we not simply make the Caller relation a link from the callee's body to the caller node: Caller (E, B)? In the presence of recursive function invocations it's possible for function bodies to be *multiply active*, that is, there may be several evaluations of a function body in progress at the same time. These different evaluations are distinguished only by the fact that they occur in different contexts (which is guaranteed by our creating new context objects on block and function entry). Thus an expression/context pair is necessary to uniquely identify a particular evaluation process. This will become more apparent when we discuss the return process below, for it's necessary for a particular function activation to return to the proper caller activation.

5. Evaluation

5.1 Invocation and Return

Evaluation of a function invocation begins with evaluation of the Rator and Rand components of the Call node. Notice that by running the Rator through the usual evaluation process we permit it to be any expression, including another function call. This permits functional programming, that is, the use of higher-order functions. The analysis rule for Calls is:

$$* \text{Eval} (E, C), \text{Call} (E), \text{Rator} (F, E), \text{Rand} (X, E)$$

$$\Rightarrow \text{Eval} (F, C), \text{Eval} (X, C).$$

The synthesis rule expects a closure to be returned as the result of evaluating the Rator. The closure in turn provides access to the body (IP), formal parameter (FP) and environment of definition (EP) of the callee. Evaluation of the function's body B is initiated in the appropriate environment (A), which

results from binding the formal N to the value V of the actual, and linking the resulting context A to the environment of definition D . It's also necessary to construct a dynamic link reflecting that E in context C is the caller of B in context A . The required rule is:

$$\begin{aligned} & \text{Call } (E), \text{ Rator } (F, E), \text{ Rand } (X, E), *Value (K, F, C), *Value (V, X, C), \\ & \text{Closure } (K), \text{ EP } (D, K), \text{ IP } (B, K), \text{ FP } (N, K), *Avail (A) \\ \Rightarrow & \text{Context } (A), \text{ Nonlocals } (D, A), \text{ Binds } (A, N, V), \text{ Caller } (E, C, B, A), \text{ Eval } (B, A). \end{aligned}$$

Eventually evaluation of the functions body completes. Then the dynamic link is used to transfer the returned value from the function's body to the Call node, thus triggering resumption of evaluation in the caller. The rule is:

$$\begin{aligned} & *Caller (E, C, B, A), *Value (V, B, A) \\ \Rightarrow & \text{Value } (V, E, C). \end{aligned}$$

Notice that if the Caller relation did not include the contexts C and B it would be possible for a value to become attached to a function's body, and be returned to the wrong one of several waiting callers.

5.2 Function Definition

For recursion to work correctly, the environment of definition of a function must include the binding of the function name itself. Thus, the context referred to by the EP of the Closure is that same Context that results from binding the function name to that Closure. We will have to ensure that the Context constructed by a function definition node (FunDef) has this reflexive property.

Evaluation of a function definition block is similar to that of a let block, except that the bound value (function body) is not evaluated at this time. Instead, a closure for the function is constructed, and the function's name is bound to this closure. This binding forms the context for the evaluation of the block's body. The analysis rule initiates evaluation of the block's body in this context:

$$\begin{aligned} & *Eval (E, C), \text{ FunDef } (E), \text{ FunName } (F, E), \text{ FunFormal } (N, E), \\ & \text{FunBody } (B, E), \text{ FunScope } (X, E), *Avail (D, K) \\ \Rightarrow & \text{Context } (D), \text{ Nonlocals } (C, D), \text{ Binds } (D, F, K), \end{aligned}$$

$\text{Closure } (K), \text{EP } (D, K), \text{IP } (B, K), \text{FP } (N, K), \text{Eval } (X, D).$

A synthesis rule waits for a value to arrive at the block's body, and attaches the value to the function block itself (i.e., the value of the function definition block is the value of the block's body):

$\text{*FunDef } (E), \text{FunScope } (X, E), \text{*Value } (V, X, D), \text{Nonlocals } (C, D)$

$\Rightarrow \text{Value } (V, E, C).$

An script demonstrating these rules is listed in Appendix B.

6. Debugging

Suppose we have the following program:

show

```
let K = 4
func fac n =
  (if (n=0)
    then {error} (1/0)
    else (n×fac(n-1)) )
fac K
```

When evaluation reaches the bottom of the recursion the zero division suspends execution. We would like to be able to explore the context of the error as indicated in the following example:

evaluate

division be zero

context

fac (n = 0)

caller

fac (n = 1)

callee

fac (n = 0)

callee

fac (n = 1)

callee

fac (n = 2)

out_context

fac = ... function ...

out_context

K = 4

Notice that the **callee** command is not single-valued, since there may be several calls being evaluated at one time. For example, in the program

```
[func f x = ....  
  (f 1 + f 2) ]
```

the two invocations of 'f' could be evaluated in parallel. Thus there would be dynamic links from both of these activations to the block body, and the **callee** command would not know which of these to pick. The reader should consider possible solutions to this problem.

First we consider the evaluator modifications necessary to support these debugging facilities. To accomplish our goal we need to record the name of a function along with its context. This is analogous to storing the function's name in its activation record. Hence, we modify the Enter Body Rule to record the function's name in the Name relation, which is defined:

- Name (M, C)
- M is the name of C
- Function (Name, Context, string).

The new Enter Body Rule is straight-forward:

```
Call (E), Rator (F, E), Rand (X, E), Var (F), Ident (M, F), *Value (K, F, C), *Value (V, X, C),  
Closure (K), EP (D, K), IP (B, K), FP (N, K), *Avail (A)  
  
⇒ Context (A), Nonlocal (D, A), Binds (A, N, V), Name (M, A), Caller (E, C, B, A),
```

Eval (B , A).

We alter the **context** command rule to notice when a variable binding is a result of function invocation, so that we can show the name of the function:

*Command (**context**), CurrentContext (C), Binds (C , N , V), Name (M , C)

\Rightarrow Display (M ^" (" ^ N ^" = " ^string-int [V] ^")).

For function bindings, rather than trying to interpret the closure, we simply note the fact that the name is bound to a function.

*Command (**context**), CurrentContext (C), Binds (C , N , K), Closure (K)

\Rightarrow Display (N ^" = ... function ...").

The reader can take it as an exercise to write the rule to unparse the function's body, should that be desired.

Implementation of the **caller** command is simply a matter of following the dynamic link:

*Command (**caller**), CurrentContext (A), Caller (E , C , B , A)

\Rightarrow CurrentContext (C), Command (**context**).

The rule for 'callee' is analogous.

What other debugging commands would be useful? It would be useful to exit from a function to its caller by supplying a return value. Exercise for the reader: Define the 'exit v ' command with this meaning.

7. References

- [MacLennan83] MacLennan, B. J., A View of Object-Oriented Programming, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-001, February 1983.
- [MacLennan84] MacLennan, B. J., The Four Forms of Ω : Alternate Syntactic Forms for an Object-Oriented Language, Naval Postgraduate School Computer Science Department Technical Report NPS52-84-026, December 1984.
- [MacLennan85a] MacLennan, B. J., A Simple Software Environment Based on Objects and Relations, *Proc. of ACM SIGPLAN 85 Conf. on Language Issues in Prog. Environments*, June 25-28, 1985, and Naval Postgraduate School Computer Science Department Technical Report NPS52-85-005, April 1985.
- [MacLennan85b] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part I, Naval Postgraduate School Computer Science Department Technical Report NPS52-85-006, May 1985.
- [MacLennan85c] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part II, Naval Postgraduate School Computer Science Department Technical Report NPS52-85-015, December 1985.
- [MacLennan86] MacLennan, B. J., Experience with Ω : Implementation of a Prototype Programming Environment Part III, Naval Postgraduate School Computer Science Department Technical Report NPS52-86-004, January 1986.
- [McArthur84] McArthur, Heinz M., *Design and Implementation of an Object-Oriented, Production-Rule Interpreter*, MS Thesis, Naval Postgraduate School Computer Science Department, December 1984.
- [Ufford85] Ufford, Robert P., *The Design and Analysis of a Stylized Natural Grammar for an Object Oriented Language (Omega)*, MS Thesis, Naval Postgraduate School Computer Science Department, June 1985.

APPENDIX A: Prototype Programming Environment

The following is a loadable input file for the prototype programming environment described in this report. It is accepted by the McArthur interpreter [McArthur84], which differs in a few details from the Ω notation used in this report (see [MacLennan84]). A transcript of a test execution of this environment is shown in Appendix B.

```
! _____
!
!           PI-4
!
!
!  A simple programming environment for an arithmetic
!  expression language, including interpreter, unparser,
!  syntax directed editor and debugger.
!
!  Features included in the language:
!  - Constants
!  - Arithmetic Operations
!  - Statically Nested Declarations
!  - Comments
!  - Conditional Expressions
!  - Recursive Function Definition and Invocation
!
! _____
!
!  PERVASIVE RELATIONS
!
!  Evaluation
!
newrelation {"Eval"};
newrelation {"Check"};
newrelation {"Value"};
newrelation {"Meaning"};
```

```
newrelation {"Explanation"};
```

```
! Contexts and Bindings
```

```
newrelation {"Context"};
```

```
newrelation {"Binds"};
```

```
newrelation {"Nonlocal"};
```

```
newrelation {"Looking"}.
```

```
! Unparsing
```

```
newrelation {"Unparse"};
```

```
newrelation {"Image"};
```

```
newrelation {"Template"};
```

```
! Comments
```

```
newrelation {"Comment"}.
```

```
! Format Control Constants
```

```
define {root, "NL", "  
"};
```

```
define {root, "TabIn", ""};
```

```
define {root, "TabOut", ""};
```

```
! Logical Constants
```

```
define {root, "true", 1};
```

```
define {root, "false", 0}.
```

! COMMAND INTERPRETER

! Command Interpreter Relations

newrelation {"Command"};

newrelation {"Argument"};

newrelation {"Root"};

newrelation {"Undef"};

newrelation {"CurrentNode"};

newrelation {"CurrentContext"};

newrelation {"SuspendedEval"};

newrelation {"Break"};

newrelation {"EvalPending"};

newrelation {"ShowPending"};

newrelation {"CommandPending"};

newrelation {"CreateRoot"};

newrelation {"CreateContext"}.

define {root, "ComIntRules", < <

! evaluate Command

if *Command ("evaluate"), CurrentNode (E), CurrentContext (C)

-> Eval (E, C), EvalPending (E), CommandPending (E);

if *Value (V, E, C), *EvalPending (E), *CommandPending (-)

-> displayn {V};

! Error Handler

if *Break (M, E, C), *CommandPending (-), *EvalPending (R), *SuspendedEval (-)

```

-> displayn {M}, SuspendedEval (R), CurrentNode (E), CurrentContext (C);

! resume Command

if *Command ("resume"), SuspendedEval (Nil)

-> displayn {"no evaluation in progress"}

else if *Command ("resume"), CurrentNode (E), CurrentContext (C), *SuspendedEval (R)

-> Eval (E, C), EvalPending (R), SuspendedEval (Nil);

! return Command

if *Command ("val"), *Argument (V), CurrentNode (E)

-> Value (V, E, C);

! show Command

if *Command ("show"), CurrentNode (E)

-> Unparse (E), ShowPending (E), CommandPending (E);

if *Image (S, E), *ShowPending (E), *CommandPending (-)

-> displayn {S};

! abort Command

if Command ("abort"), *Eval (E, C) -> ;

if Command ("abort"), *Value (V, E, C) -> ;

if Command ("abort"), *Check (V, E, C) -> ;

if Command ("abort"), *Nonlocal (C, D) -> ;

if Command ("abort"), *Binds (D, N, V) -> ;

if *Command ("abort"), ~Eval (E, C), ~Value (V, E, C), ~Nonlocal (C, D), ~Binds (D, N, V),

```

```
*SuspendedEval ( - ), *CurrentContext ( - )  
-> CurrentContext (Nil), SuspendedEval (Nil), displayn {"aborted"};  
  
! done Command  
  
if *Command ("done") -> displayn {"PI system stopped"};
```

! Syntax Directed Editing

```
if *Command ("delete"), CurrentNode (E), Undef (E)
```

```
-> displayn ("already deleted");
```

! begin Command

```
if *Command ("begin"), *CurrentNode (- )
```

```
-> CreateRoot (newobj {}), CommandPending (Nil);
```

```
if *CreateRoot (E), *CommandPending (- )
```

```
-> Root (E), Undef (E), CurrentNode (E);
```

! root Command

```
if *Command ("root"), *CurrentNode (- ), Root (E)
```

```
-> CurrentNode (E), Command ("show");
```

! Debugging Commands

! out_context Command

```
if *Command ("out_context"), *CurrentContext (D), Nonlocal (C, D)
```

```
-> CurrentContext (C), Command ("context")
```

```
else if *Command ("out_context")
```

```
-> displayn ("at outermost level");
```

! in_context Command

```
if *Command ("in_context"), *CurrentContext (C), Nonlocal (C, D)
```

```
-> CurrentContext (D), Command ("context")
```

```
else if *Command ("in_context")
```


-> displayn ("at innermost level");

! alter Command

if *Command ("alter"), *Argument (U), CurrentContext (C), *Binds (C, N, V)

-> Binds (C, N, U), Command ("context")

else if *Command ("alter"), *Argument (-)

-> displayn ("no binding");

> > }.

act {ComIntRules}.

! COMMENTS

define {root, "RemRules", < <

! rem Command

if *Command ("rem"), *Argument (S), CurrentNode (E), ~Comment (- , E)

-> Comment (S, E);

if *Command ("rem"), *Argument (-), CurrentNode (E), Comment (- , E)

-> displayn ("node already commented");

! delete_rem Command

if *Command ("delete_rem"), CurrentNode (E), *Comment (- , E)

-> displayn ("done");

if *Command ("delete_rem"), CurrentNode (E), ~Comment (- , E)

-> displayn ("no comment");

> > }.

act {RemRules}.

! INCOMPLETE PROGRAM

! Tables

Explanation ("incomplete program", ["error", 0]).

define {root, "IncomProgRules", < <

! Evaluation

if *Eval (E, C), Undef (E), *CurrentNode (-)

-> Break ("Incomplete", E, C);

! Unparsing

if *Unparse (E), Undef (E)

-> Image ("< expr> ", E);

> > }.

act {IncomProgRules}.

! CONSTANT NODES

! Relations

newrelation {"Con"};

newrelation {"Litval"}.

! Functions

fn Id [x]: x.

! Tables

Meaning (Id, "lit").

Template (int_str, "lit").

define {root, "ConRules", < <

! Evaluation

if *Eval (e, c), Con (e), Litval (v, e), Meaning (f, "lit")

-> Value (f [v], e, c);

! Unparsing

if *Unparse (e), Con (e), Litval (v, e), Template (f, "lit"), Comment (s, e)

-> Image (f [v] + " {" + s + "}", e)

else if *Unparse (e), Con (e), Litval (v, e), Template (f, "lit")

-> Image (f [v], e);

! # Command

if *Command ("#"), *Argument (V), IsInt [V], CurrentNode (E), *Undef (E)

-> Con (E), Litval (V, E);

if *Command ("#"), *Argument (V), CurrentNode (E), ~Undef (E)

-> displayn ("defined node");

! delete Command

if *Command ("delete"), CurrentNode (E), *Con (E), *Litval (V, E)

-> Undef (E), Command ("show");

> > }.

act {ConRules}.

! VARIABLE NODES

! Relations

newrelation {"Var"};

newrelation {"Ident"}.

define {root, "VarRules", < <

! Evaluation

if *Eval (E, C), Var (E), Ident (N, E)

-> Looking (N, C, E, C);

if *Looking (N, C, E, D), Binds (C, N, V)

-> Value (V, E, D)

else if *Looking (N, C, E, D), Nonlocal (Cprime, C)

-> Looking (N, Cprime, E, D)

else if *Looking (N, C, E, D), *CurrentNode (-), *CurrentContext (-)

-> Break ("Unbound: " + N, E, D);

! Unparsing

if *Unparse (E), Var (E), Ident (N, E), Comment (S, E)

-> Image (N + " {" + S + "}", E)

else if *Unparse (E), Var (E), Ident (N, E)

-> Image (N, E);

! var Command

if *Command ("var"), *Argument (N), CurrentNode (E), *Undef (E)

-> Var (E), Ident (N, E);

! delete Command

if *Command ("delete"), CurrentNode (E), *Var (E), *Ident (N, E)

-> Undef (E), Command ("show");

> > }.

act {VarRules}.

! APPLICATION NODES

! Relations

```
newrelation {"Appl"};
```

```
newrelation {"Op"};
```

```
newrelation {"Left"};
```

```
newrelation {"Right"};
```

```
newrelation {"CreateAppl"}.
```

! Evaluation Functions

```
fn Sum [x, y]: x + y;
```

```
fn Dif [x, y]: x - y;
```

```
fn Product [x, y]: x * y;
```

```
fn Quotient [x, y]:
```

```
  if y = 0 -> ["error", 1]
```

```
  else x / y;
```

```
fn Equal [x, y]: if x = y -> true else false;
```

```
fn IsErrorcode [w]:
```

```
  if ~IsList [w] | w = Nil -> Nil
```

```
  else first [w] = "error";
```

! Unparsing Functions

```
fn upSum [x, y]: "(" + x + " + " + y + ")";
```

```
fn upDif [x, y]: "(" + x + " - " + y + ")";
```

```
fn upProd [x, y]: "(" + x + " x " + y + ")";
```

```
fn upQuot [x, y]: "(" + x + " / " + y + ")";
```

```
fn upEqua [x, y]: "(" + x + " = " + y + ")";
```

! Evaluation Tables

Meaning (Sum, "+ ");

Meaning (Dif, "-");

Meaning (Product, "x");

Meaning (Quotient, "/");

Meaning (Equal, "=").

! Unparsing Tables

Template (upSum, "+ ");

Template (upDif, "-");

Template (upProd, "x");

Template (upQuot, "/");

Template (upEqua, "=").

! Other Tables

Explanation ("division by zero", ["error", 1]).

define {root, "ApplRules", < <

! Evaluation

if *Eval (e, c), Appl (e), Left (x, e), Right (y, e)

-> Eval (x, c). Eval (y, c);

if *Value (u, x, c), *Value (v, y, c), Appl (e), Op (n, e), Left (x, e), Right (y, e), Meaning (f, n)

-> Check (f [u, v], e, c);

if *Check (w, e, c), ~IsErrorcode [w]

-> Value (w, e, c);

if *Check (w, e, c), IsErrorcode [w], Explanation (s, w), *CurrentNode (q)

-> Break (s, e, c);

! Unparsing

if *Unparse (e), Appl (e), Left (x, e), Right (y, e)

-> Unparse (x), Unparse (y);

! Unparsing Comments on Applications

if Appl (E), Op (N, E), Left (X, E), Right (Y, E), *Image (U, X), *Image (V, Y), Comment (S, E)

-> Image ("{" + S + "}" (" + U + N + V + ") ", E)

else if *Image (u, x), *Image (v, y), Appl (e), Op (n, e), Left (x, e), Right (y, e), Template (f, n)

-> Image (f [u, v], e);

! +, -, x, /, = Commands

if *Command (op), member [op, "+", "-", "x", "/", "= "], *CurrentNode (E), *Undef (E)

-> CommandPending (E), CreateAppl (op, E, newobj {}, newobj {});

if *CreateAppl (op, E, X, Y), *CommandPending (E)

-> {Appl (E), Op (op, E), Left (X, E), Right (Y, E), Undef (X), Undef (Y), CurrentNode (X);

Command ('show')};

! delete Command

if *Command ("delete"), CurrentNode (E), *Appl (E), *Op (N, E), *Left (X, E), Right (Y, E)

-> Undef (E), Command ("show");

! in Command

if *Command ("in"), *CurrentNode (E), Left (X, E)

-> CurrentNode (X), Command ("show");

! out Command

if *Command ("out"), *CurrentNode (X), Left (X, E)

-> CurrentNode (E), Command ("show");

if *Command ("out"), *CurrentNode (Y), Right (Y, E)

-> CurrentNode (E), Command ("show");

! next Command

if *Command ("next"), *CurrentNode (X), Left (X, E), Right (Y, E)

-> CurrentNode (Y), Command ("show");

! prev Command

if *Command ("prev"), *CurrentNode (Y), Right (Y, E), Left (X, E)

-> CurrentNode (X), Command ("show");

> > }.

act {ApplRules}.

! BLOCK

! Relations

newrelation {"Block"};

newrelation {"BndVar"};

newrelation {"BndVal"};

newrelation {"Body"};

newrelation {"CreateLet"}.

define {root. "BlockRules". < <

! Evaluation

if *Eval (E, C), Block (E), BndVal (X, E)

-> Eval (X, C);

if Block (E), BndVar (N, E), BndVal (X, E), Body (B, E), *Value (V, X, C), Comment (S, E)

-> CreateContext (newobj {}, N, V, C, B, S)

else if Block (E), BndVar (N, E), BndVal (X, E), Body (B, E), *Value (V, X, C)

-> CreateContext (newobj {}, N, V, C, B);

if *CreateContext (D, N, V, C, B, S)

-> CreateContext (D, N, V, C, B), Comment (S, D);

if *CreateContext (D, N, V, C, B)

-> Context (D), Binds (D, N, V), Nonlocal (C, D), Eval (B, D);

if Block (E), Body (B, E), *Value (V, B, D), *Nonlocal (C, D), *Binds (D, N, W), *Context (D)

-> Value (V, E, C);

! Unparsing


```
if *Unparse (E), Block (E), BndVal (X, E), Body (B, E)
```

```
-> Unparse (X), Unparse (B);
```

```
! Unparsing comments on blocks
```

```
if Block (E), BndVar (N, E), BndVal (X, E), Body (B, E), *Image (U, X), *Image (V, B), Comment (S, E)
```

```
-> Image (
```

```
  TabIn + NL + "[let {" + S + "}"
```

```
  + TabIn + NL + N + "= " + U
```

```
  + NL + V + "]"
```

```
  - TabOut + TabOut, E)
```

```
else if Block (E), BndVar (N, E), BndVal (X, E), Body (B, E), *Image (U, X), *Image (V, B)
```

```
-> Image ( TabIn + NL
```

```
  + "[let " + N + "= " + U
```

```
  + TabIn + NL + V + "]"
```

```
  + TabOut + TabOut,
```

```
E);
```

```
! let Command
```

```
if *Command ("let"), *Argument (N), *CurrentNode (E), *Undef (E)
```

```
-> CommandPending (E), CreateLet (N, E, newobj {}, newobj {});
```

```
if *CreateLet (N, E, X, B), *CommandPending (E)
```

```
-> {Block (E), BndVar (N, E), BndVal (X, E), Body (B, E),
```

```
  Undef (X), Undef (B), CurrentNode (X);
```

```
  Command ("show");
```

```
! in Command
```

```
if *Command ("in"), *CurrentNode (E), BndVal (X, E)
```

```

-> CurrentNode (X), Command ("show");

! out Command

if *Command ("out"), *CurrentNode (X), BndVal (X, E)
-> CurrentNode (E), Command ("show");

if *Command ("out"), *CurrentNode (B), Body (B, E)
-> CurrentNode (E), Command ("show");

! next Command

if *Command ("next"), *CurrentNode (X), BndVal (X, E), Body (B, E)
-> CurrentNode (B), Command ("show");

! prev Command

if *Command ("prev"), *CurrentNode (B), Body (B, E), BndVal (X, E)
-> CurrentNode (X), Command ("show");

> > }.

act {BlockRules}.

```

! CONDITIONAL EXPRESSION NODES

! Relations

newrelation {"ConEx"};

newrelation {"Cond"};

newrelation {"Conseq"};

newrelation {"Alt"};

newrelation {"CreateConEx"}.

define {root. "ConExRules". < <

! Evaluation

if *Eval (E, C), ConEx (E), Cond (B, E)

-> Eval (B, C);

if ConEx (E), Cond (B, E), Conseq (T, E), *Value (true, B, C)

-> Eval (T, C);

if ConEx (E), Cond (B, E), Alt (F, E), *Value (false, B, C)

-> Eval (F, C);

if ConEx (E), Conseq (T, E), *Value (V, T, C)

-> Value (V, E, C);

if ConEx (E), Alt (F, E), *Value (V, F, C)

-> Value (V, E, C);

! Unparsing

if *Unparse (E), ConEx (E), Cond (B, E), Conseq (T, E), Alt (F, E)

-> Unparse (B), Unparse (T), Unparse (F);

```

if ConEx (E), Cond (B, E), Conseq (T, E), Alt (F, E), *Image (U, B), *Image (V, T), *Image (W, F)
-> Image ( TabIn + NL +
    "if " + U + NL +
    " then " + V + NL +
    " else " + W + " )" +
    TabOut + NL, E);

```

! Editing

! if Command

```

if *Command ("if"), *CurrentNode (E), *Undef (E)
-> CommandPending (E), CreateConEx (E, newobj {}, newobj {}, newobj {});

if *CreateConEx (E, B, T, F), *CommandPending (E)
-> {ConEx (E), Cond (B, E), Conseq (T, E), Alt (F, E),
    Undef (B), Undef (T), Undef (F), CurrentNode (B);
    Command ("show")};

```

! in Command

```

if *Command ("in"), *CurrentNode (E), ConEx (E), Cond (B, E)
-> CurrentNode (B), Command ("show");

```

! out Command

```

if *Command ("out"), *CurrentNode (B), Cond (B, E), ConEx (E)
-> CurrentNode (E), Command ("show");

```

```

if *Command ("out"), *CurrentNode (T), Conseq (T, E), ConEx (E)
-> CurrentNode (E), Command ("show");

```

```

if *Command ("out"), *CurrentNode (F), Alt (F, E), ConEx (E)

```

```

-> CurrentNode (E), Command ('show');

! next Command

if *Command ('next'), *CurrentNode (B), Cond (B, E), Conseq (T, E)
-> CurrentNode (T), Command ('show');

if *Command ('next'), *CurrentNode (T), Conseq (T, E), Alt (F, E)
-> CurrentNode (F), Command ('show');

! prev Command

if *Command ('prev'), *CurrentNode (F), Alt (F, E), Conseq (T, E)
-> CurrentNode (T), Command ('show');

if *Command ('prev'), *CurrentNode (T), Conseq (T, E), Cond (B, E)
-> CurrentNode (B), Command ('show');

> > }.

act {ConExRules}.

```

! FUNCTION DEFINITION AND INVOCATION

! Definition Abstract Structure

```
newrelation {"FunDef"};
newrelation {"FunName"};
newrelation {"FunFormal"};
newrelation {"FunBody"};
newrelation {"FunScope"};
```

! Invocation Abstract Structure

```
newrelation {"Call"};
newrelation {"Rator"};
newrelation {"Rand"};
```

! Runtime Relations

```
newrelation {"Closure"};
newrelation {"EP"};
newrelation {"IP"};
newrelation {"FP"};
newrelation {"Caller"};
newrelation {"Name"};
newrelation {"Argument2"};

newrelation {"CreateCall"};
newrelation {"CreateFunDef"};
newrelation {"CreateActRecord"};
newrelation {"CreateFunContext"}.
```

```
define {root, "FunRules", < <
```

! FUNCTION INVOCATION

! Editing

! call Command

if *Command ("call"), *CurrentNode (E), *Undef (E)

-> CommandPending (E), CreateCall (newobj {}, newobj {}, E);

if *CreateCall (F, X, E), *CommandPending (E)

-> Call (E), Rator (F, E), Rand (X, E), Undef (F), Undef (X), CurrentNode (F);

! next Command

if *Command ("next"), *CurrentNode (F), Rator (F, E), Call (E), Rand (X, E)

-> CurrentNode (X), Command ("show");

! Unparsing

if *Unparse (E), Call (E), Rator (F, E), Rand (X, E)

-> Unparse (F), Unparse (X);

if Call (E), Rator (F, E), Rand (X, E), *Image (U, F), *Image (V, X)

-> Image (U + " " + V, E);

! Evaluation

! Evaluate Rator and Rand

if *Eval (E, C), Call (E), Rator (F, E), Rand (X, E)

-> Eval (F, C), Eval (X, C);

! Evaluate Body

if Call (E), Rator (F, E), Rand (X, E), Var (F), Ident (M, F),

```

*Value (K, F, C), *Value (V, X, C),
Closure (K), EP (D, K), IP (B, K), FP (N, K)
-> CreateActRecord (newobj {}, D, N, V, M, E, C, B);

if *CreateActRecord (A, D, N, V, M, E, C, B)
-> Context (A), Nonlocal (D, A), Binds (A, N, V),
Name (M, A), Caller (E, C, B, A), Eval (B, A);

! Return Value

if *Caller (E, C, B, A), *Value (V, B, A)
-> Value (V, E, C);

```


! FUNCTION DEFINITION

! Editing

! func Command

if *Command ("func"), *Argument (F), *Argument2 (N), *CurrentNode (E), *Undef (E)

-> CommandPending (E), CreateFunDef (newobj {}, newobj {}, F, N, E);

if *CreateFunDef (B, X, F, N, E), *CommandPending (E)

-> FunDef (E), FunName (F, E), FunFormal (N, E), FunBody (B, E), FunScope (X, E),

Undef (B), Undef (X), CurrentNode (B);

! next Command

if *Command ("next"), *CurrentNode (B), FunBody (B, E), FunScope (X, E)

-> CurrentNode (X), Command ("show");

! in Command

if *Command ("in"), *CurrentNode (E), FunDef(E), FunBody (B, E)

-> CurrentNode (B), Command ("show");

! out Command

if *Command ("out"), *CurrentNode (B), FunBody (B, E)

-> CurrentNode (E), Command ("show");

if *Command ("out"), *CurrentNode (X), FunScope (X, E)

-> CurrentNode (E), Command ("show");

! Unparsing

if *Unparse (E), FunDef (E), FunBody (B, E), FunScope (X, E)

-> Unparse (B), Unparse (X);

```

if FunDef (E), FunName (F, E), FunFormal (N, E), FunBody (B, E), FunScope (X, E),
  *Image (U, B), *Image (V, X)
-> Image (TabIn + NL
  + "|func " + F + " " + N + " = "
  + TabIn + U +
  + NL + V + "|"
  + TabOut + TabOut,
E);

! Evaluation

! Analysis

if *Eval (E, C), FunDef (E), FunName (F, E), FunFormal (N, E), FunBody (B, E), FunScope (X, E)
-> CreateFunContext (newobj {}, newobj {}, C, F, B, N, X);

if *CreateFunContext (D, K, C, F, B, N, X)
-> Context (D), Nonlocal (C, D), Binds (D, F, K),
  Closure (K), EP (D, K), IP (B, K), FP (N, K),
  Eval (X, D);

! Synthesis

if FunDef (E), FunScope (X, E), *Value (V, X, D), Nonlocal (C, D)
-> Value (V, E, C);

! Debugging

! context Command

if *Command ("context"), CurrentContext (C), Binds (C, N, K), Closure (K)
-> displayn {N + " = ... function ..."}

```

```

else if *Command ("context"), CurrentContext (C), Binds (C, N, V), Name (M, C)
-> displayn {M + " (" + N + " = " + int_str [V] + ")"}

else if *Command ("context"), CurrentContext (C), Binds (C, N, V), Comment (S, C)
-> displayn ( N + " = " + int_str [V] + " {" + S + "}")

else if *Command ("context"), CurrentContext (C), Binds (C, N, V)
-> displayn ( N + " = " + int_str [V] )

else if *Command ("context")
-> displayn ("no bindings");

! caller Command

if *Command ("caller"), CurrentContext (A), Caller (E, C, B, A)
-> CurrentContext (C), Command ("context");

! callee Command

-> > }.

act {FunRules}.

```

! TEST DRIVER

! Relations

newrelation {"Script"};

newrelation {"Test"}.

! Monadic Command List

define {root, "MonadicCommands",

["#", "val", "let", "var", "alter", "rem"]}

define {root, "TestRules", < <

! Script Sequencer

if *Script (A, Nil), ~Command (-), ~CommandPending (-)

-> A ("Script completed")

else if *Script (A, L), ~Command (-), ~CommandPending (-), first [L] = "func"

-> { displayn {" ... " + first [rest [L]]

+ " " + first [rest [rest [L]]] + " " + first [L]};

Command (first [L]), Argument (first [rest [L]]), Argument2 (first [rest [rest [L]]]);

Script (A, rest [rest [rest [L]]]) }

else if *Script (A, L), ~Command (-), ~CommandPending (-), member [first [L], MonadicCommands]

-> { display {" ... "};

display {first [rest [L]]};

displayn {" " + first [L]};

Command (first [L]), Argument (first [rest [L]]);

Script (A, rest [rest [L]]) }

else if *Script (A, L), ~Command (-), ~CommandPending (-)

```
-> { displayn {"... " + first [L]};
```

```
    Command (first [L]);
```

```
    Script (A, rest [L]) };
```

! Test Scripts

```
if *Test (A, 1) -> { Script {[
```

```
    "begin", "let", "K", "#", 4, "next", "func", "fac", "n",
```

```
    "if", "=", "var", "n", "next", "#", 0, "out", "next", "#", 1, "next",
```

```
    "x", "var", "n", "next", "call", "var", "fac", "next",
```

```
    ".", "var", "n", "next", "#", 1, "root", "in", "next", "in", "next",
```

```
    "call", "var", "fac", "next", "var", "K", "root", "evaluate"
```

```
    ]};
```

```
A ("Test done");
```

```
};
```

```
if *Test (A, 2) -> { Script {[
```

```
    "in", "next", "in", "in", "next", "delete",
```

```
    "rem", "error", "/", "#", 1, "next", "#", 0, "root", "evaluate",
```

```
    "context", "caller", "callee", "caller", "caller",
```

```
    "out_context", "out_context", "done"
```

```
    ]};
```

```
A ("Test done");
```

```
};
```

```
>> }.
```

```
act {TestRules}.
```

! Initialize Data Structures

```
CurrentNode (Nil).
```

CurrentContext (Nil).

SuspendedEval (Nil).

displayn {"PI-4 System loaded"}.

APPENDIX B: Transcript of Ω Session

The following is a transcript of an Ω session illustrating the operation of the prototype programming environment shown in Appendix A. The assertion 'Script {testscript}' causes the commands in testscript to be executed in order. The n th testscript is executed by 'Test{n}'. Each command is printed on a separate line, followed by whatever output is generated by the programming environment. This transcript was produced by the McArthur interpreter [McArthur84].

```
% omega
```

```
OMEGA-1 11/30/84
```

```
Use Cntl-D or exit{} to quit.
```

```
For help, enter help{"?"}.
```

```
To report a bug, enter Bugs{}.
```

```
newrelation rule activated.
```

```
> do{"PI4.rul"}.
```

```
PI-4 System loaded
```

```
OK
```

```
> {Test{1}; Test{2}}.
```

```
... begin
```

```
... K let
```

```
< expr>
```

```
... 4 #
```

```
... next
```

```
< expr>
```

```
... fac n func
```

```
... if
```

```
< expr>
```

```
... =
```

< expr>

... n var

... next

< expr>

... 0 #

... out

(n = 0)

... next

< expr>

... 1 #

... next

< expr>

... x

< expr>

... n var

... next

< expr>

... call

... fac var

... next

< expr>

... -

< expr>

... n var

... next

< expr>

... 1 #

... root


```
[let K = 4
  func fac n =
    (if (n = 0)
      then 1
      else (n x fac (n - 1)) )
```

```
< expr> ] ]
```

```
... in
```

```
4
```

```
... next
```

```
[func fac n =
  (if (n = 0)
    then 1
    else (n x fac (n - 1)) )
```

```
< expr> ]
```

```
... in
```

```
(if (n = 0)
  then 1
  else (n x fac (n - 1)) )
```

```
... next
```

```
< expr>
```

```
... call
```

```
... fac var
```

```
... next
```

```
< expr>
```

```
... K var
```

```
... root
```

```
[let K = 4
```

```
  [func fac n =
```

```
    (if (n = 0)
```

```
      then 1
```

```
      else (n x fac (n - 1)) )
```

```
    fac K ] ]
```

```
... evaluate
```

```
24
```

```
... in
```

```
4
```

```
... next
```

```
[func fac n =
```

```
  (if (n = 0)
```

```
    then 1
```

```
    else (n x fac (n - 1)) )
```

```
  fac K ]
```

```
... in
```

```
(if (n = 0)
```

```
  then 1
```

```
  else (n x fac (n - 1)) )
```

```
... in
```

```
(n = 0)
```

```
... next
```

```
1
```

```
... delete
```

< expr>

... error rem

... /

< expr>

... 1 #

... next

< expr>

... 0 #

... root

|let K = 4

|func fac n =

 (if (n = 0)

 then {error} (1/0)

 else (n x fac (n - 1)))

 fac K |]

... evaluate

division by zero

... context

fac (n = 0)

... caller

fac (n = 1)

... callee

fac (n = 0)

... caller

fac (n = 1)

... caller

fac (n = 2)

... out_context
fac = ... function ...

... out_context

K = 4

... done

PI system stopped

Test done

> exit{}

Goodbye.

%

INITIAL DISTRIBUTION LIST

Defense Technical Information Center
Cameron Station
Alexandria, VA 22314

2

Dudley Knox Library
Code 0142
Naval Postgraduate School
Monterey, CA 93943

2

Office of Research Administration
Code 012
Naval Postgraduate School
Monterey, CA 93943

1

Chairman, Code 52
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

40

Associate Professor Bruce J. MacLennan
Code 52ML
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

12

Dr. Robert Grafton
Code 433
Office of Naval Research
800 N. Quincy
Arlington, VA 22217-5000

1

Dr. David Mizell
Office of Naval Research
1030 East Green Street
Pasadena, CA 91106

1

Dr. Stephen Squires
DARPA
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

1

Professor Jack M. Wozencraft, 62Wz
Department of Electrical and Comp. Engr.
Naval Postgraduate School
Monterey, CA 93943

1

Professor Rudolf Bayer
Institut für Informatik
Technische Universität
Postfach 202420
D-8000 München 2
West Germany

1

Dr. Robert M. Balzer
USC Information Sciences Inst.
4676 Admiralty Way
Suite 10001
Marina del Rey, CA 90291

1

Mr. Ronald E. Joy
Honeywell, Inc.
Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, MI 55402

1

Mr. Ron Laborde
INMOS
Whitefriars
Lewins Mead
Bristol
Great Britain

1

Mr. Lynwood Sutton
Code 424, Building 600
Naval Ocean Systems Center
San Diego, CA 92152

1

Mr. Jeffrey Dean
Advanced Information and Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040

1

Mr. Jack Fried
Mail Station D01/31T
Grumman Aerospace Corporation
Bethpage, NY 11714

1

Mr. Dennis Hall
New York Videotext
104 Fifth Avenue, Second Floor
New York, NY 10011

1

Professor S. Ceri
Laboratorio di Calcolatori
Dipartimento di Elettronica
Politecnico di Milano
20133 - Milano
Italy

1

Mr. A. Dain Samples
Computer Science Division - EECS
University of California at Berkeley
Berkeley, CA 94720

1

Antonio Corradi
Dipartimento di Elettronica
Informatica e Sistemistica
Universita Degli Studi di Bologna
Viale Risorgimento, 2

Bologna Italy	1
Dr. Peter J. Welcher Mathematics Dept., Stop 9E U.S. Naval Academy Annapolis, MD 21402	1
Dr. John Goodenough Wang Institute Tyng Road Tyngsboro, MA 01879	1
Professor Richard N. Taylor Computer Science Department University of California at Irvine Irvine, CA 92717	1
Dr. Mayer Schwartz Computer Research Laboratory MS 50-662 Tektronix, Inc. Post Office Box 500 Beaverton, OR 97077	1
Professor Lori A. Clarke Computer and Information Sciences Department LGRES ROOM A305 University of Massachusetts Amherst, MA 01003	1
Professor Peter Henderson Department of Computer Science SUNY at Stony Brook Stony Brook, NY 11794	1
Dr. Mark Moriconi SRI International 333 Ravenswood Avenue Menlo Park, CA 95025	1
Professor William Waite Department of Electrical and Computer Engineering The University of Colorado Campus Box 425 Boulder, CO 80309-0425	1
Professor Mary Shaw Software Engineering Institute Carnegie-Mellon University Pittsburgh, PA 15213	1
Dr. Warren Teitelman Engineering/Software Sun Microsystems Federal, Inc. 2550 Garcia Avenue	

Mountain View, CA 94031

1

Prof. Raghu Ramakrishnan
Univ. of Texas at Austin
Dept. of Computer Science
Austin, TX 79712

1