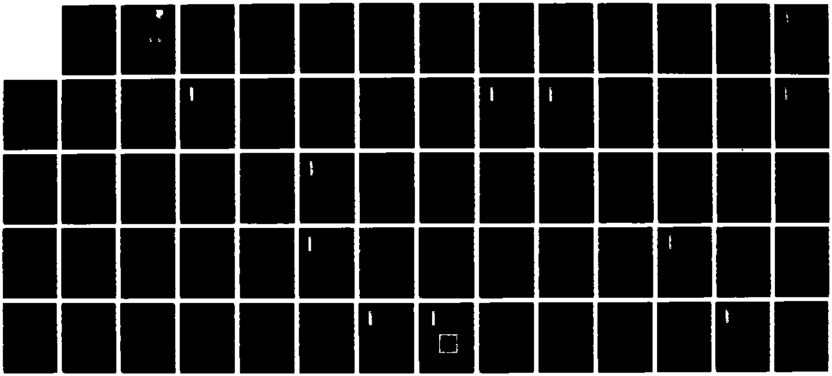
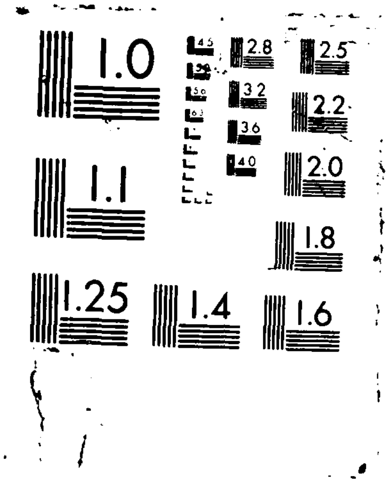
AD-A190 167 NEW GENERATION KNOWLEDGE PROCESSING VOLUME 3(U) 1/1
SYRACUSE UNIV NY J A ROBINSON ET AL. OCT 87
RADC-TR-87-165-VOL-3 F30602-84-K-0001

UNCLASSIFIED F/G 12/5 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

RADC-TR-87-165, Vol III (of three)
Final Technical Report
October 1987

NEW GENERATION KNOWLEDGE PROCESSING

Syracuse University

J. Alan Robinson and Kevin J. Greene

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-87-165, Vol III (of three) has been reviewed and is approved for publication.

APPROVED: Northrup Fowler

NORTHRUP FOWLER III
Project Engineer

APPROVED: Raymond P. Urtz Jr

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: Richard W. Pouliot

RICHARD W. POULIOT
Directorate of Plans & Programs

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COES) Griffiss AFB NY 13441-5700. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

TYPOGRAPHICAL ERRORS

Vol 1:	p1,	line -3	Insert a space between 1 and P
	p9,	line 13	Remove space at end, or start new paragraph with line 14.
	p14,	line -1	Repeat on top of next page. Delete
	p24,	line 13	Insert a space between the + and the - in λ^+ - <u>normal</u>
	p24,	line 3	The wording makes the proper referent for the pronoun "there" hard to find.
	p25,	line -11	Suggest replacement of "evaluate (= reduce)" with "reduce".
	p66,	line -9	Delete the first "the".
	p77,	line -1	Bold face the "S" in "S-redex"
	p77,	line -7	Bold face the "K" in "K-redex"
	p77,	line -11	Bold face the "I" in "I-redex"
	p84,	line 8	Change "Annexe" to "volume"
	p84,	line -2	Change "as" to "at"
	p89,	line -9	Delete "in"

The contents section should reference appropriate page numbers (this applies equally to Vol 3).

PAGINATION

Vol 1,	page 8	Lines 1-7 should be on previous page. Start new page with start of section 1.
	page 15	Lines -1, -2 heading for a table should appear on same page as the table!
	page 33	Bottom line should go to next page.
	page 68	" " " "
	page 78	" " " "
	page 85	" " " "

COPYRIGHT

Review by our legal staff RADC indicates that the following notice must be on each page upon which Syracuse has indicated a copyright:

> "This material may be reproduced by or for
> the U.S. Government pursuant to the copyright
> license under DAR clause 7-104.9(a) (1981 May)."

Volume III, pages 6, 10, 15, 16, 20, 26, 40, 46, 55, 56, and 61 are among the pages requiring this notice.

Northrup Fowler

NORTHRUP FOWLER III
Project Engineer
Knowledge Engineering Branch

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-87-165, Vol III (of three)

6a. NAME OF PERFORMING ORGANIZATION Syracuse University	6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COES)
6c. ADDRESS *(City, State, and ZIP Code)* Syracuse NY 13244		7b. ADDRESS *(City, State, and ZIP Code)* Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Rome Air Development Center	8b. OFFICE SYMBOL *(If applicable)* COES	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-84-K-0001

8c. ADDRESS *(City, State, and ZIP Code)*
Griffiss AFB NY 13441-5700

10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO	WORK UNIT ACCESSION NO
62702F	5581	27	10

11. TITLE *(Include Security Classification)*
NEW GENERATION KNOWLEDGE PROCESSING

12. PERSONAL AUTHOR(S)
J. Alan Robinson, Kevin J. Greene

13a. TYPE OF REPORT Final	13b. TIME COVERED FROM Dec 83 TO Jan 87	14. DATE OF REPORT *(Year, Month, Day)* October 1987	15. PAGE COUNT

16. SUPPLEMENTARY NOTATION
N/A Fr. back

17. COSATI CODES

FIELD	GROUP	SUB-GROUP
12	05	

18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
Artificial Jntelligence, Graph Reduction,
I gic Programming, Combinators,
Functional Programming, Programming Languages

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

The main goal of this project was to design a high-level programming system (which we have named SUPER, an acronym for "Syracuse University Parallel Expression Reducer") with two parts: a language which would combine the functional (as in LISP, SASL or ML) with the relational (as in PROLOG) programming concepts into a single new paradigm and a machine which would execute programs written in the language, using reduction and a multiprocessor architecture.

The SUPER language is an extension of the basic lambda-calculus which we call lambda plus. It is formally a collection of expressions together with some rules and definitions which give them meaning and make it possible to do deductive reasoning and computation with them. The expressions of the SUPER language fall into three main syntactic categories: atoms, abstractions, and combinations.

Volume I describes the SUPER system, and discusses the conceptual background in terms of (over)

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL Northrup Fowler III	22b. TELEPHONE *(Include Area Code)* (315) 330-7794	22c. OFFICE SYMBOL RADC (COES)

Previous editions are obsolete SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED

Block 19. Abstract (Cont'd)

which it can best be understood. In developing these ideas over the period of the project we devised and implemented two related single-processor reduction systems, LNF and LNF-Plus, as experimental tools to help us learn more about SUPER language design issues. These systems have turned out to be of considerable interest and utility in their own right, and they have taken on separate and independent identities.

Volume 2 contains a detailed presentation of the single-processor software programming system LNF which was developed to serve as a test bed and simulation tool for the "classical" part of the SUPER system.

Volume 3 presents the final, enhanced version of LNF, which we call LNF-Plus and which provides the user with as close an approximation as we can achieve on a single processor of the SUPER system. Volume 3 is also designed as a useful guide to someone who wishes to use the system for experimental computations.

Table of Contents

1. Introduction. The LNF-Plus system is an implementation of the LNF-Plus language on a *sequential* machine (a Symbolics Lisp Machine) - i.e. it is a *one reduction at a time* graph reduction system. The SUPER system, comprising both the LNF-Plus language and the abstract SUPER machine (a *many reductions at a time* graph reduction system), is described in [Robinson 1987]. The LNF-Plus language, a combined functional and relational language, is an extension of the purely functional LNF language defined in [Greene 1985]. The LNF-Plus language results from adding *absolute set abstraction* expressions (ASA-expressions), which take the form:

$$\{template \mid \exists \; (variables) \; predication1 \; \& \ldots \& \; predicationP\}$$

to the LNF language. The predicates present in the predications above may be defined either by λ-expressions (abstractions) or Horn clauses, thus allowing both functional and relational styles of programming in the same language.

It is assumed that the reader is somewhat familiar with the λ-calculus, the SKI-calculus, combinator graph reduction, the first-order predicate calculus (and its Horn Clause subset), and the workings (at least the user interface) of a Symbolics Lisp Machine. Descriptions of the λ-calculus, SKI-calculus, and combinator graph reduction may be found in [Greene 1985].

The LNF-Plus graph reduction machine is almost identical to the machine employed by the LNF system. A detailed description of the LNF-machine may be found in [Greene 1985]. The extensions required to transform the LNF machine into the LNF-Plus machine are detailed herein.

The purpose of this document is twofold. Besides providing a technical summary of absolute set abstraction expression (ASA-expression) reduction, instructions are provided which tell users how to initialize and utilize the LNF-Plus system.

The document begins by providing the sequence of operations required to set up the LNF-Plus environment. Following this the user interface is described to a degree which will allow novice users to: ask for simple expressions to be reduced, get online help from the system, define new symbols, monitor reduction sequences, trace calls on user specified functors, record LNF-Plus sessions in files, turn on/off garbage collection, interpret some of the reduction statistics provided, and interact with both the Lisp Machine's text editor (ZMACS) and file system.

The facility for simulating concurrent reduction in the system is then presented. The main use of this facility is for the reduction of ASA-expressions. As ASA-expressions are the only type of expression new to the LNF-Plus language (not already present in the LNF language) their manner of compilation and reduction is detailed. For details on the method of compilation and reduction for LNF-Plus expression types which are also in the LNF language the reader is encouraged to read [Greene 1985].

Each of the functors built into the LNF-Plus system and their associated reduction rules have been placed in Appendix 1. Appendix 2 is a copy of the system's *standard prelude* - a collection of definitions of some of the more commonly used functions. A presentation of a series of example programs and their execution on the system is included in Appendix 3.

2. Getting Started. Before the system can be used it must be created - the next several sections detail how this is to be accomplished, explain the uses of the various panes of the LNF-Plus frame (the system's interface), and explain how to begin programming in LNF-Plus.

2.1 Setting up the LNF-Plus environment. It is assumed that the tape containing the LNF-Plus system has been loaded onto disk and the *sys:site;lnf-plus.translations* file has been edited appropriately. If this has not been done, please follow the instructions given in the hardcopy of the file *-read-me-.text* provided with the tape.

To load the LNF-Plus environment, simply type (at a Lisp Listener):

```
Load System LNF-Plus
```

After the system has been loaded, an LNF-Plus frame (collection of window panes making up the system - similar to the Lisp Machine's Document Examiner or Inspector) can be created by either typing *SELECT* R (use the *SELECT* key) or choosing LNF-Plus from the System menu. It takes about one minute for the frame to be created. After creation, the LNF-Plus environment may be exited and reentered just like the Lisp Machine's other systems (ZMACS, the Inspector, et al.), e.g. to leave LNF-Plus for ZMACS type *SELECT* E and to return type *SELECT* R.

2.2 The LNF-Plus frame. The LNF-Plus frame is initially divided into two panes. The pane on the left is called the *interaction pane* and the pane next to it the *statistics pane*. User input and system output is typed on the interaction pane. The interaction pa. e's prompt (initially) is *LNF of*. During each reduction, statistics are gathered and then displayed on the statistics pane. Statistics on all phases of the computation are recorded. Some of the more important ones (to the user) will be discussed as this introduction proceeds.

In addition to the two panes which are present in the initial configuration of the LNF-Plus frame, two more panes may be created: the monitor pane (for monitoring the reduction sequence at a very fine grain) and the trace pane (for tracing specific functors or user defined functions) and their arguments. Descriptions of these optional panes will be given later.

The mouse line (in reverse video at the bottom of the screen) reminds the user how LNF-Plus' facilities may be invoked by the mouse. The use of the mouse will also be described later.

2.3 The Read-Reduce-Print loop. As has been noted above LNF-Plus is a reduction system. The user types in an expression in the LNF-Plus language and asks the system to reduce it. The system does so and prints the reduced result. If *E* is the input expression and *RE* is the result printed, then *E* and *RE* stand in the following relation. *RE* is a reduction of *E* having the same denotation as *E*.

The reduction from *E* to *RE* is achieved by the following transformation:

$$RE = \text{UNCOMPILE}[\text{REDUCE}[\text{COMPILE}[E]]].$$

Compiling the LNF-Plus expression *E* involves first eliminating all occurrences of bound variables from *E* via an abstraction algorithm (which is a generalized version of D. A. Turner's) yielding a variable free applicative expression *F* and then producing from *F* its graphical representation *G*. This process is detailed in [Greene 1985].

The graph *G* is then reduced to the graph *RG* (as specified by LNF-Plus' set of reduction rules - see Appendix 1) and then uncompiled from a graph to the string *RE* (the result displayed on the screen).

Note the difference here between LNF-Plus (a system with reduction semantics) and Lisp (having denotation semantics). LNF-Plus accepts expressions as input and produces expressions as output claiming that the output has the same denotation as the input. Lisp also accepts expressions as input but instead of producing expressions as output produces instead values (or denotations). If E is the input expression to a Lisp system and V is the output, then Lisp claims that V is the denotation of (value of) E.

In short, LNF-Plus is a *denotation preserving* system, whereas Lisp is a *denotation producing* system.

How much reduction of the input expression is performed by LNF-Plus? The user, to some degree, controls how much work is done by asking either for a completely reduced result (no opportunities for reduction (redexes) left) - such an expression is said to be in *normal form* - or for only the structure (outline or shell) of the result to be determined (where many redexes may still be left but the nature (type) of the result is known) - such an expression is said to be in *lazy-normal form*. For precise definitions of these concepts and many related ones as well please see [Greene 1985].

The user specifies how the result is to be presented (in normal form or lazy-normal form) by changing the prompt in the interaction pane. Initially the prompt is *LNF of ... is* which is short for *the lazy-normal form of ... is*. So, initially the system will be providing only a partially reduced result. The prompt is changed by clicking once on the middle button of the mouse and selecting from the pop up menu the desired form in which future results will be displayed. Choosing a different *printing form* from the menu changes the prompt.

In addition to the prompts *LNF of ... is* and *NF of ... is* there is a third prompt corresponding to a third print form which affects how lists are displayed. Normally, a list containing the two items **A** and **B** is displayed as `[A,B]`. If, however, the *NF of Members of ... is* prompt is selected, then subsequent lists are displayed without the surrounding square brackets and without commas separating the lists' items. Thus, the list `[A,B]` would be printed as **AB**. This form is mainly used for graphical output as pictures are represented by lists of lines and usually the lines are the only part of the output desired. The same amount of reduction is performed when the prompt is *NF of Members of ... is* as when the prompt is *NF of ... is* - i.e. the output is completely reduced.

2.4 Simple expressions. LNF-Plus expressions come in many flavors but the simplest are *atoms*, *combinations*, and *abstractions*. An atom is either a *functor*, a *constructor*, or a *variable*. The set of functors is fixed by the system. They are the atoms which have reduction rules associated with them. Please refer to Appendix 1 for the complete list of functors and their associated reduction rules.

The atoms +, HD, W, APPEND, and IDIV are examples of functors. Variables are denoted by symbols whose first character is a ?. For example, ?1, ?x , and ?v12 are all variables. Constructors are all the rest. The symbol PAIR, all numerals, the truthvalues (TRUE and FALSE) are among the constructors.

Combinations are made by juxtaposing simple expressions. For example,

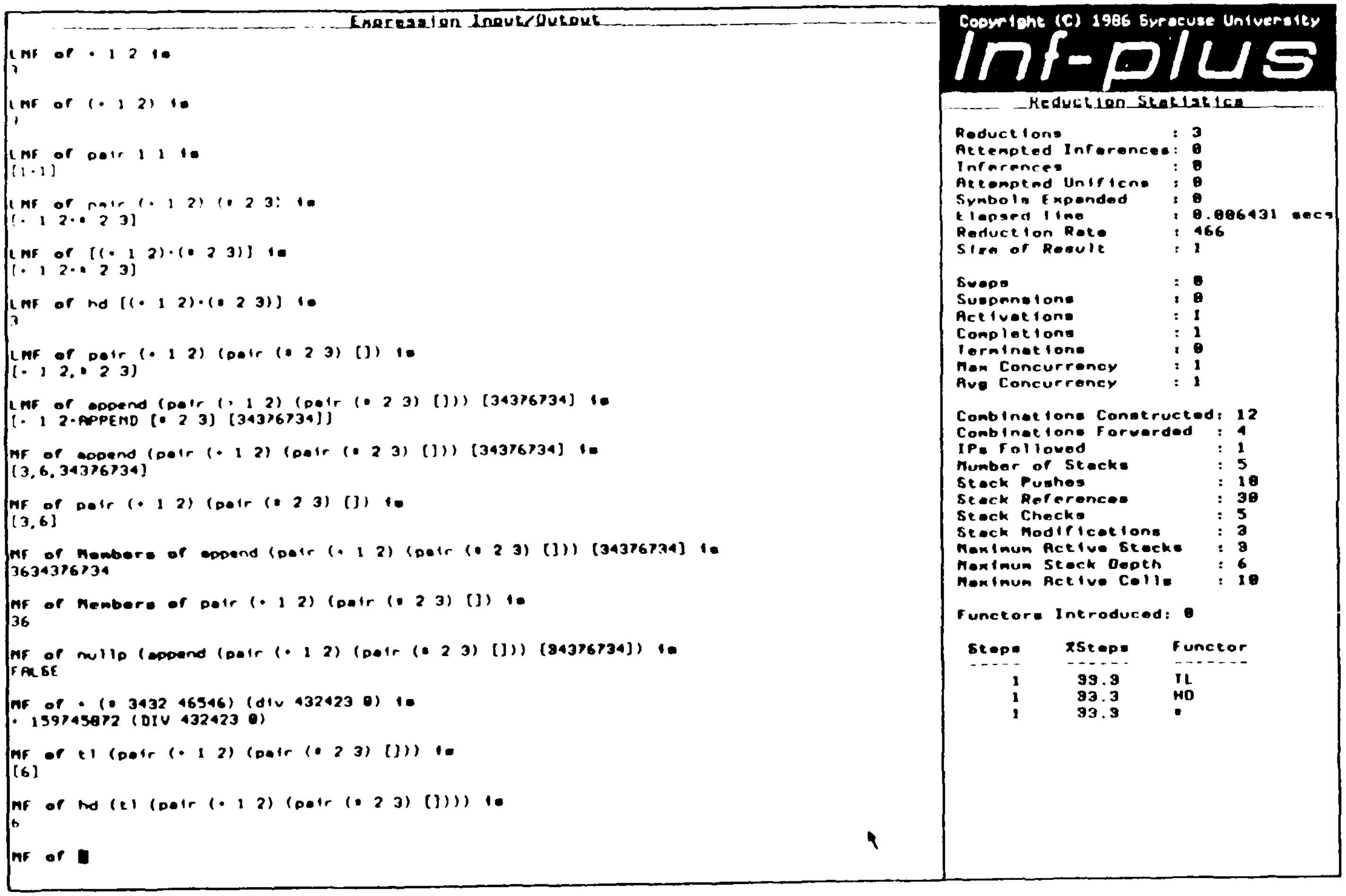
```
+ 3,
W +, and
(HD (PAIR 12 34))
```

are all combinations. The expression on the left of a combination is called the operator and the expression on the right is called the operand. Parentheses are used for grouping. The operation of combination associates to the left, so, for example, the expression + 2 3 is a combination whose operator is + 2 and whose operand is 3. The expression whose operator is + and whose operand is 2 3 would be typed + (2 3).

Abstractions are expressions which denote functions. Their syntax follows: An abstraction has a *binding* and a *body* and is written (λ *binding body*). The abstraction's body may be any LNF-Plus expression. Its binding is either a variable or a parenthesized sequence of *bound expressions*. A bound expression (*be*) is either a constructor, a variable, or a combination whose operator and operand are both be's (with the restriction that the operator may not be a variable). Two abstractions denoting the doubling function are: (λ ?x (+ ?x ?x)) and (λ ?x (* 2 ?x)). The function, applicable only to pairs, which returns the tail of the pair could be written (it need not be as it is a functor and hence a language primitive): (λ ([?x•?xs]) ?xs).

See the next page for some examples of reductions of simple expressions. Note the differences in the displayed result when the prompt is changed. Note also that [A•B] is syntactic sugar for (PAIRA B) and [A,B,C] is sugar for (PAIR A (PAIR B (PAIR C []))).

2.5 Functors and their reduction rules. Reduction takes place when an expression matches the left-hand-side of a *functor's reduction rule*. For example, the expression (+ 2 3) reduces in one step to 5 because there is a reduction rule for the functor + which says that whenever the functor + is the *initial-atom* of an expression and the expression has two *arguments* which are numbers, then the expression may be replaced with the expression representing their sum. The expression (+ (- 3 5) 55) is also reducible as the functor + is *strict* and its first argument is reducible. A functor is strict if it requires its



```
Expression Input/Output

LNF of + 1 2 is
3

LNF of (+ 1 2) is
3

LNF of pair 1 1 is
[1·1]

LNF of pair (+ 1 2) (* 2 3) is
[+ 1 2·* 2 3]

LNF of [(+ 1 2)·(* 2 3)] is
[+ 1 2·* 2 3]

LNF of hd [(+ 1 2)·(* 2 3)] is
3

LNF of pair (+ 1 2) (pair (* 2 3) []) is
[+ 1 2,* 2 3]

LNF of append (pair (+ 1 2) (pair (* 2 3) [])) [34376734] is
[+ 1 2·APPEND [* 2 3] [34376734]]

NF of append (pair (+ 1 2) (pair (* 2 3) [])) [34376734] is
[3,6,34376734]

NF of pair (+ 1 2) (pair (* 2 3) []) is
[3,6]

NF of Members of append (pair (+ 1 2) (pair (* 2 3) [])) [34376734] is
3634376734

NF of Members of pair (+ 1 2) (pair (* 2 3) []) is
36

NF of nullp (append (pair (+ 1 2) (pair (* 2 3) [])) [34376734]) is
FALSE

NF of + (* 3432 46546) (div 432423 0) is
+ 159745872 (DIV 432423 0)

NF of tl (pair (+ 1 2) (pair (* 2 3) [])) is
[6]

NF of hd (tl (pair (+ 1 2) (pair (* 2 3) []))) is
6

NF of █
```

```
Copyright (C) 1986 Syracuse University
Inf-plus
Reduction Statistics

Reductions            : 3
Attempted Inferences: 0
Inferences            : 0
Attempted Unificns    : 0
Symbols Expanded      : 0
Elapsed Time          : 0.006431 secs
Reduction Rate        : 466
Size of Result        : 1

Swaps                 : 0
Suspensions           : 0
Activations           : 1
Completions           : 1
Terminations          : 0
Max Concurrency       : 1
Avg Concurrency       : 1

Combinations Constructed: 12
Combinations Forwarded  : 4
IPs Followed            : 1
Number of Stacks        : 5
Stack Pushes            : 10
Stack References        : 30
Stack Checks            : 5
Stack Modifications     : 3
Maximum Active Stacks   : 3
Maximum Stack Depth     : 6
Maximum Active Cells    : 10

Functors Introduced: 0

 Steps    XSteps    Functor
 -----    ------    -------
   1       33.3     TL
   1       33.3     HD
   1       33.3     *
```

arguments (some functors are only strict in some of their arguments - the functor `IF` for example) to be reduced before it is applied. All of the arithmetic functors are strict. The functor `W`, defined by the single reduction rule: `W f x = f x x` is an example of a nonstrict functor.

As mentioned several times before, the complete list of functors and their associated reduction rules may be found in Appendix 1 of this document. This information is also made available to the user online. To get it, click once on the right mouse button, select the *Document Functor(s)* option from the menu, and then choose the functor or functors on which documentation is desired. The reduction rules for the selected functors will then be displayed in the interaction pane.

3. Defining Symbols. As one types larger and larger expressions into the LNF-Plus system one begins to feel the need for abbreviations. There is an LNF-Plus facility which allows users to name an expression with a symbol and, from that time on, use the symbol in place of the expression. This is accomplished by clicking once on the left mouse button. The prompt changes to *Definition:* and the system waits for a symbol definition to be input.

3.1 Equations. Symbols are defined in the LNF-Plus frame by typing equations at the system's *Definition:* prompt. Equation templates are displayed below which show the form these equations may take. Definitions may also be entered in ZMACS buffers and saved in Lisp Machine files. The form of these definitions differs only slightly from the form of the equations typed directly at the interaction pane. The ZMACS utterance displayed just below each equation template is the ZMACS equivalent of the equation above it.

LNF-Plus FRAME DEFINITION:
x = LNF-Plus-exp
where x is a symbol and LNF-Plus-exp is any LNF-Plus expression.
e.g. the definition `sum = (+ 3 45)` sets up the symbol `sum` to name the expression `(+ 3 45)`.
ZMACS EQUIVALENT:
`(define` x LNF-Plus-exp`)`, e.g. `(define sum (+ 3 45))`.

LNF-Plus FRAME DEFINITION:

f bv1 ... bvN = LNF-Plus-exp,
where f is a symbol, the bvi's are bound expressions
(a variable or a construction whose arguments
are each bound expressions), and LNF-Plus-exp is a LNF-Plus expression.
e.g. the definition

```
 factorial ?x =
   if (zerop ?x) 1 (* ?x (factorial (sub1 ?x)))
```

causes the symbol **factorial** to be associated with
the lambda expression (abstraction):

```
 (λ (?x) (if (zerop ?x) 1 (* ?x (factorial (sub1 ?x)))))
```

ZMACS EQUIVALENT:

```
(define (f bv1 ... bvN) LNF-Plus-exp).
```

This method of defining the function f is equivalent to:

```
(define f (λ (bv1 ... bvN) LNF-Plus-exp)).
```

LNF-Plus FRAME DEFINITION:

```
(f bv11 ... bv1N = LNF-Plus-exp1) &
(f bv21 ... bv2N = LNF-Plus-exp2) &
 ...
(f bvM1 ... bvMN = LNF-Plus-expM),
```

where f is again a symbol, the bvij's are again bound expressions
(with the restriction that (bvi1 ... bviN) and (bvj1 ... bvjN) 1≤i≠j≤M are
not unifiable), and the LNF-Plus-expi's are LNF-Plus expressions.
The &s are optional.

ZMACS EQUIVALENT:

```
(define
   (f bv11 ... bv1N) LNF-Plus-exp1
   (f bv21 ... bv2N) LNF-Plus-exp2
   ...
   (f bvM1 ... bvMN) LNF-Plus-expM)
```

This form of definition is sugar for the following definition:

```
(define f (λ (v1 ... vN)
   (case (OPDS v1 ... vN)
        (OPDS bv11 ... bv1N)  → LNF-Plus-exp1 |
        (OPDS bv21 ... bv2N)  → LNF-Plus-exp2 |
        ...
        (OPDS bvM1 ... bvMN)  → LNF-Plus-expM
   endcase))).
```

3.2 Horn Clauses. In addition to symbol definitions entered via equations, predicate definitions (used to deduce the normal form of some set expressions - see the section on sets below) may be entered via *Horn clauses* having the following syntax.

LNF-Plus FRAME DEFINITION: (defining the predicate p)

```
  ((p t11 ... t1N)  ← B11 & ... & B1K1)  &
  ((p t21 ... t2N)  ← B21 & ... & B1K2)  &
  ...
  ((p tM1 ... tMN)  ← BM1 & ... & BMKM),
 defines the predicate p via the M Horn Clauses.
 The ←s and &s are optional.
```

ZMACS EQUIVALENT:

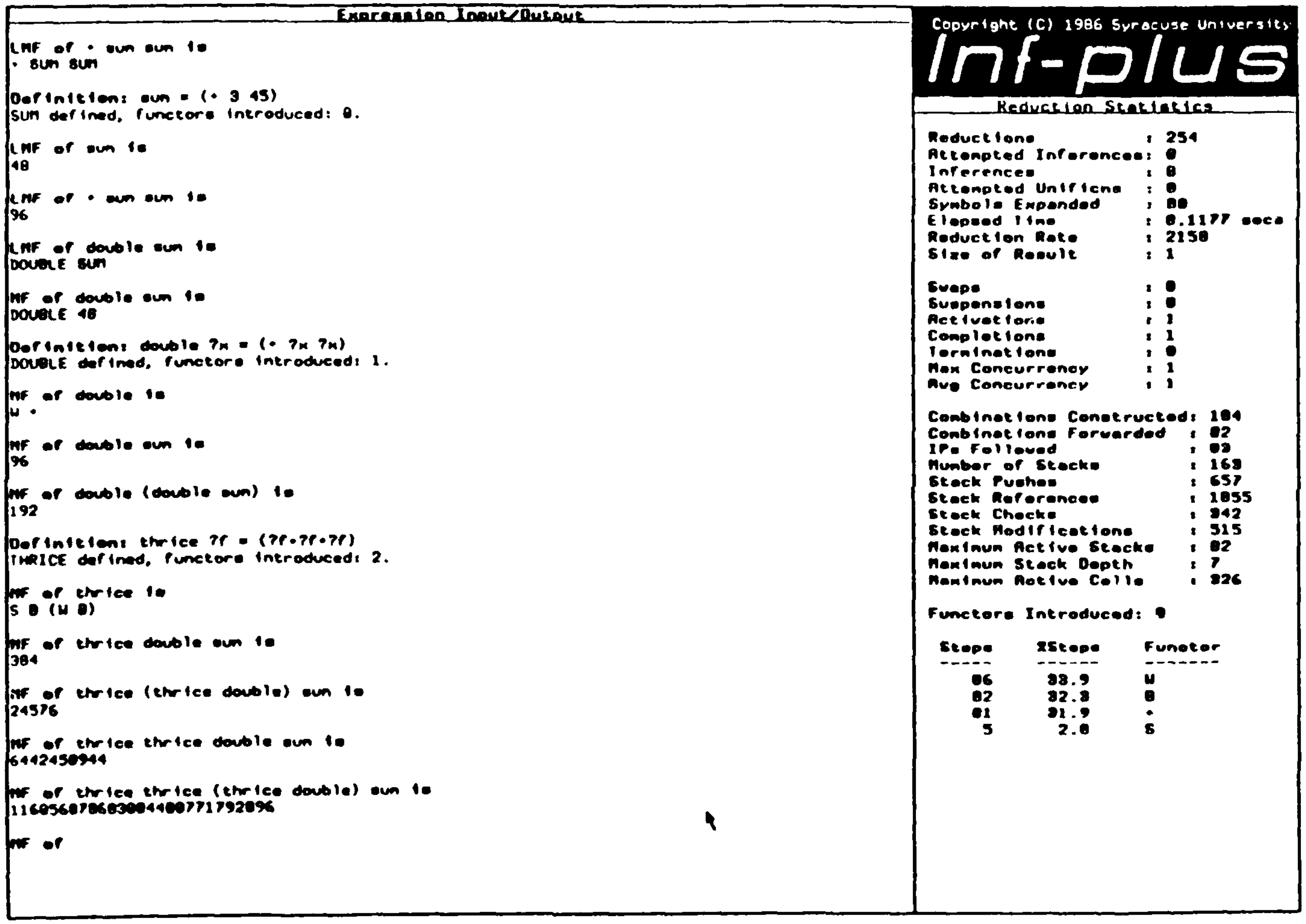
```
  (define
     ((p t11 ... t1N)  ← B11 & ... & B1K1)
     ((p t21 ... t2N)  ← B21 &  ... & B1K2)
     ...
     ((p tM1 ... tMN)  ← BM1 & ... & BMKM))
```

For example, the two relations naive reverse (**nrev**) and append (**app**) could be defined in ZMACS as follows:

```
(define
  ((nrev [] []) ←)
  ((nrev [?x•?xs] ?zs) ← (nrev ?xs ?ys) & (app ?ys [?x] ?zs)))

(define
  ((app [] ?xs ?xs) ←)
  ((app [?x•?xs] ?ys [?x•?zs]) ← (app ?xs ?ys ?zs)))
```

Please refer to the next page (which shows a sample session) for some examples of symbol definition. Note the use of the character "⊕" (achieved by typing **SYMBOL-SH-+**, i.e. the **SYMBOL**, **SHIFT**, and + keys struck simultaneously) as an infix operator in the definition of **thrice**. The expression (**f** ⊕ g) is simply sugar for (B f g) - i.e. ⊕ is the *infix* functional composition operator whereas the B combinator is the *prefix* functional composition operator.

4. LNF-Plus Facilities. The system provides several primitive facilities to the user which enable him/her to easily interact with the Lisp Machine's editor (ZMACS); load, save, and remove symbol definitions; monitor an ongoing reduction, trace the action of specified functors, record a session with the



```
Expression Input/Output

LNF of + sum sum is
+ SUM SUM

Definition: sum = (+ 3 45)
SUM defined, functors introduced: 0.

LNF of sum is
48

LNF of + sum sum is
96

LNF of double sum is
DOUBLE SUM

NF of double sum is
DOUBLE 48

Definition: double ?x = (+ ?x ?x)
DOUBLE defined, functors introduced: 1.

NF of double is
W +

NF of double sum is
96

NF of double (double sum) is
192

Definition: thrice ?f = (?f.?f.?f)
THRICE defined, functors introduced: 2.

NF of thrice is
S B (W B)

NF of thrice double sum is
384

NF of thrice (thrice double) sum is
24576

NF of thrice thrice double sum is
6442450944

NF of thrice thrice (thrice double) sum is
116056878683004400771792896

NF of
```

```
Copyright (C) 1986 Syracuse University
Inf-plus

Reduction Statistics

Reductions                : 254
Attempted Inferences: 0
Inferences                : 0
Attempted Unifications    : 0
Symbols Expanded          ; 80
Elapsed Time              : 0.1177 secs
Reduction Rate            : 2158
Size of Result            : 1

Swaps                     : 0
Suspensions               : 0
Activations               : 1
Completions               : 1
Terminations              : 0
Max Concurrency           : 1
Avg Concurrency           : 1

Combinations Constructed: 184
Combinations Forwarded  : 82
IPs Followed            : 83
Number of Stacks        : 163
Stack Pushes            : 657
Stack References        : 1855
Stack Checks            : 342
Stack Modifications     : 515
Maximum Active Stacks   : 82
Maximum Stack Depth     : 7
Maximum Active Cells    : 326

Functors Introduced: 0

 Steps     %Steps    Functor
 -----     ------    -------
    86      33.9     W
    82      32.3     B
    81      31.9     +
     5       2.0     S
```

system in a file for later perusal, and to control whether or not Lisp is performing garbage collection underneath the system. Each of these facilities is explained briefly below.

4.1 The interaction between LNF-Plus and ZMACS. As mentioned above, the definitions entered in the LNF-Plus frame (shown on the previous page) could have also been typed in a ZMACS buffer. To tell LNF-Plus about a definition typed into a buffer, i.e. install the symbol definition in the LNF-Plus environment, one simply evaluates the definition as one would a Lisp definition typed in a buffer. This is done by typing **C-SH-E** (the **CONTROL**, **SHIFT**, and **E** keys hit simultaneously) - after placing the cursor *inside* the definition. A buffer full of definitions may be installed at once by invoking the extended ZMACS command: **`M-X evaluate buffer`**.

Buffers of LNF-Plus definitions may be saved into (and retrieved from) the file system just like any other file. It is suggested that a `.LISP` file extension be used for LNF-Plus files so as to make use of the automatic parenthesis blinking, automatic indenting, etc. of ZMACS' Lisp Mode. In order to make LNF-Plus definitions readable by the Lisp reader, some characters (which the Lisp system wants to treat specially) must be *slashified* - i.e. prefixed with the slash character (/). These three characters are ",", "|" and ";". For example, the LNF-Plus list `[1,2,3]`, when entered in ZMACS, must be typed `[1/,2/,3]`. The font `super-font` has been created from the Lisp Machine's built-in font `cptfont` which maps the key **SYMBOL-SH-+** to ⊕ (circle, the infix functional composition operator).

The next page is a copy of the file *lnf-plus:exs;digits.lisp* demonstrating the structure of LNF-Plus definitions in a Lisp Machine file. Note that circle prints as circle-plus (a limitation of the laser printer software) - in a ZMACS buffer the circle is displayed properly.

4.2 Loading, saving, and removing definitions. Files containing LNF-Plus definitions may be installed directly from the LNF-Plus frame (without having to go to ZMACS, read the file into a buffer, and then evaluate it). This is done by clicking the left mouse button twice and selecting the *Load Definitions* option from the pop-up menu. The file name is then entered and the box labeled **EXIT** is clicked. Following this last click, the user's file is read and the definitions installed.

Symbol definitions, regardless of how they are installed, may be saved into a Lisp Machine file from the LNF-Plus frame. To do this, perform a double left click with the mouse and select the *Save Definitions* option from the pop-up menu. The symbol definitions are saved into the file whose name you then enter.

```
;;; -*- Mode: LISP; Syntax: Zetalisp; Package: USER; Base: 10; -*-


;;; An example taken from Bird and Wadler's draft of their
;;; "An Introduction to Functional Programming" book -- pg. 132

;;; (TAKE-WHILE p list) is the initial segment of list (init-list)
;;; such that for all el in init-list (p el) is TRUE.
;;; An interesting definition of take-while in terms of PRIM-REC:
(DEFINE (TAKE-WHILE ?P) (PRIM-REC (NOT⊕?P⊕HD) (PAIR⊕HD) TL []))

;;; A concise and efficient definition of list reversal:
(DEFINE REVERSE (LREDUCE (λ (?X ?Y) (PAIR ?Y ?X)) []))

;;; Two auxiliary functions.
;;; Division by 10 and remainder after division by 10:
(DEFINE REM-BY-10 (λ (?X) (REM ?X 10)))
(DEFINE DIV-BY-10 (λ (?X) (IDIV ?X 10)))

;;; The function DIGITS, which takes as input an integer (I)
;;; and returns as result the list of I's digits.  Note that
;;; the definition makes use of lazy-evaluation, infinite
;;; lists, and higher order functions:
(DEFINE DIGITS
  (REVERSE⊕
  (MAP REM-BY-10)⊕
  (TAKE-WHILE (NOT⊕ZEROP))⊕
  (ITERATE DIV-BY-10)))

;;; An example: (to be typed at LNF-plus' interaction pane)
;;; (DIGITS 3981)
;;; (REVERSE⊕(MAP REM-BY-10)⊕(TAKE-WHILE (NOT⊕ZEROP)))
;;;   [3981,398,39,3,0,0,0,...]
;;; (REVERSE⊕(MAP REM-BY-10)) [3981,398,39,3]
;;; REVERSE [1,8,9,3]
;;; [3,9,8,1]
```

After having defined a symbol, one may want to *undefine* it, i.e. remove its definition from the system. This is accomplished by performing a double left click on the mouse button, selecting the *Remove Definitions* option, and then clicking on the symbol or symbols whose definition(s) is no longer desired.

4.3 Reduction monitoring. Reductions may be *monitored* - this means that not only is the final result displayed but as many of the intermediate forms of the result as desired by the user are also displayed. Monitoring is enabled, meaning that subsequent reductions will be monitored until monitoring is disabled, by clicking twice on the middle mouse button. The interaction pane splits into two panes one sitting on top of the other. The top pane is the interaction pane (now reduced in size) and the bottom pane is the newly created *monitor pane*. Following this restructuring of the LNF-Plus frame, the user is asked to supply the *monitor's period*. The monitor's period is the number of reductions after which an intermediate result (along with its accompanying statistics) is displayed. Any positive integer may be supplied as the monitor's period (the default being 1, requiring *each* intermediate form of the result to be displayed).

A monitored reduction (with period N) of the expression E proceeds as follows. N reductions are performed on E, the result, say En, is displayed in the monitor pane and the reduction pauses. Just above the display of En is a line of statistics of the form:

```
Step: n  APs: m  Fnctr: f  Fwded: k  IPs Flwd: l  RPQ: o
expression
```

where *n* is the number of reduction steps taken so far, *m* is the number of application (combination) vertices created to this point, *f* is the functor whose rule was just applied, *k* is the number of forwarding pointers created, *l* is the the number of forwarding pointers followed, and *o* is the current length of the process queue.

The first three items in the line are self explanatory, the others require some discussion. Graph reduction is performed by *overwriting* unreduced graphs with equivalent reduced graphs. Sometimes this requires that a vertex (at the root of a redex) be forwarded to another vertex. The effect of forwarding a vertex v to a vertex u is to have all references (pointers) to v in the graph forwarded to u, i.e. v is no longer *looked at* but *looked through* to see u. The statistics *k* and *l* in the monitor line should now be clear. It remains to explain the statistic *o*. The process queue is discussed below in connection with ASA-expression reduction. The relevance of the statistic *o*, displaying the length of this queue, will be appreciated only after that discussion.

See the following page for a simple example showing the monitoring of the reduction of the expression `thrice add1 1`.

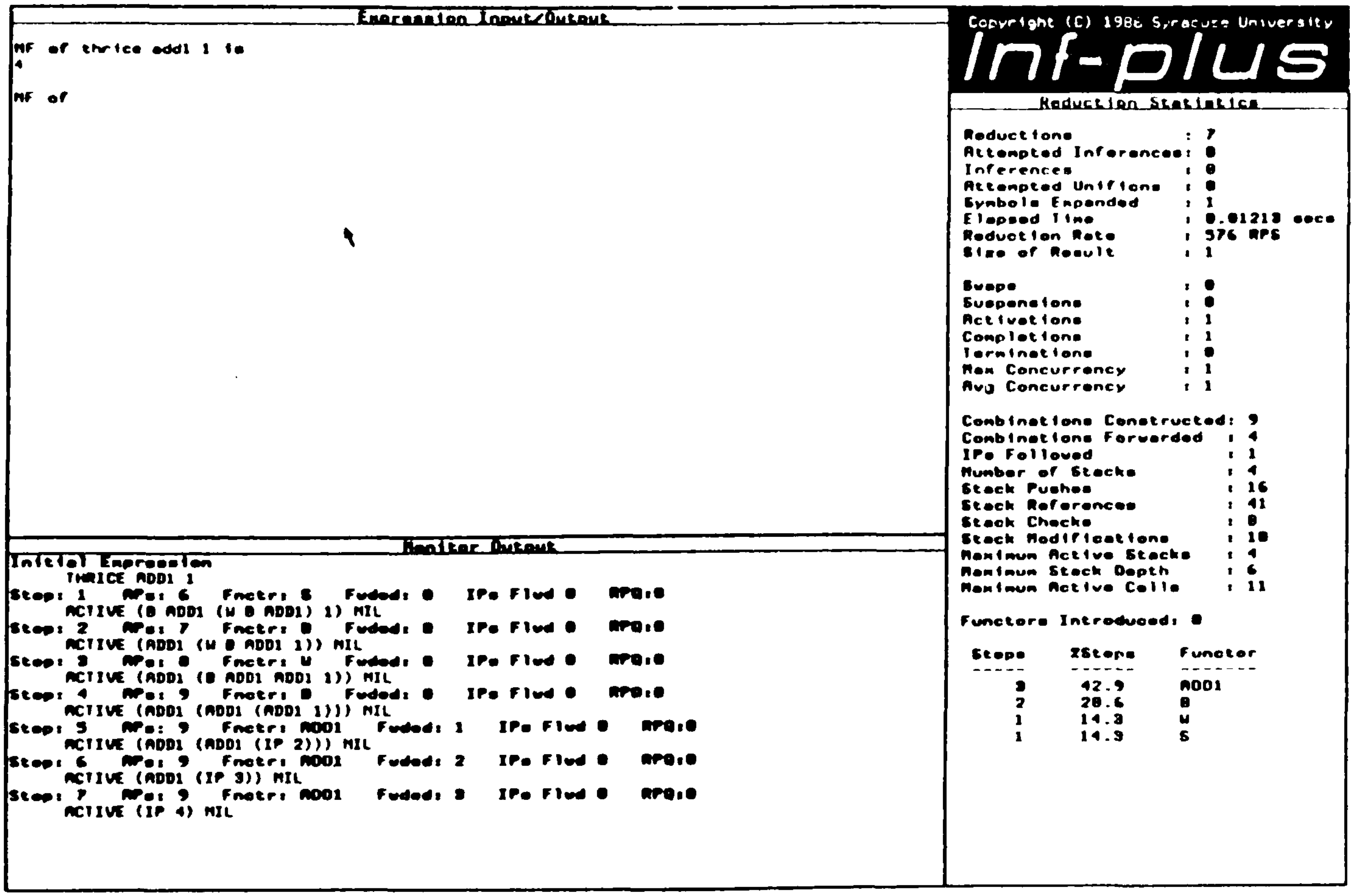
4.4 Functor tracing. Instead of monitoring a reduction, the user may only wish to see the action taken by specific functors. The trace facility is provided for this purpose. Tracing is enabled by clicking once on the right mouse button, selecting the *Start Tracing / Add Some Symbols to Trace List* option, and then choosing which functor(s) is(are) to be traced. The set of functors being traced may be changed from problem to problem. Functor tracing is disabled by again clicking once on the right mouse button and then selecting the *End Tracing / Remove Some Symbols from Trace List* option.

The page after next shows the result of tracing the functor **add1** during the reduction of the expression `thrice add1 1`. Additionally, this snapshot of the LNF-Plus frame shows the menu which pops up when the user performs a single right mouse click.

4.5 Reduction statistics. Many detailed statistics are recorded and displayed for each reduction. Most of them were recorded to facilitate the design of the system but are not of general interest. Several of them, however, might be of interest to users. These, more generally useful statistics, are given brief explanations in the table below

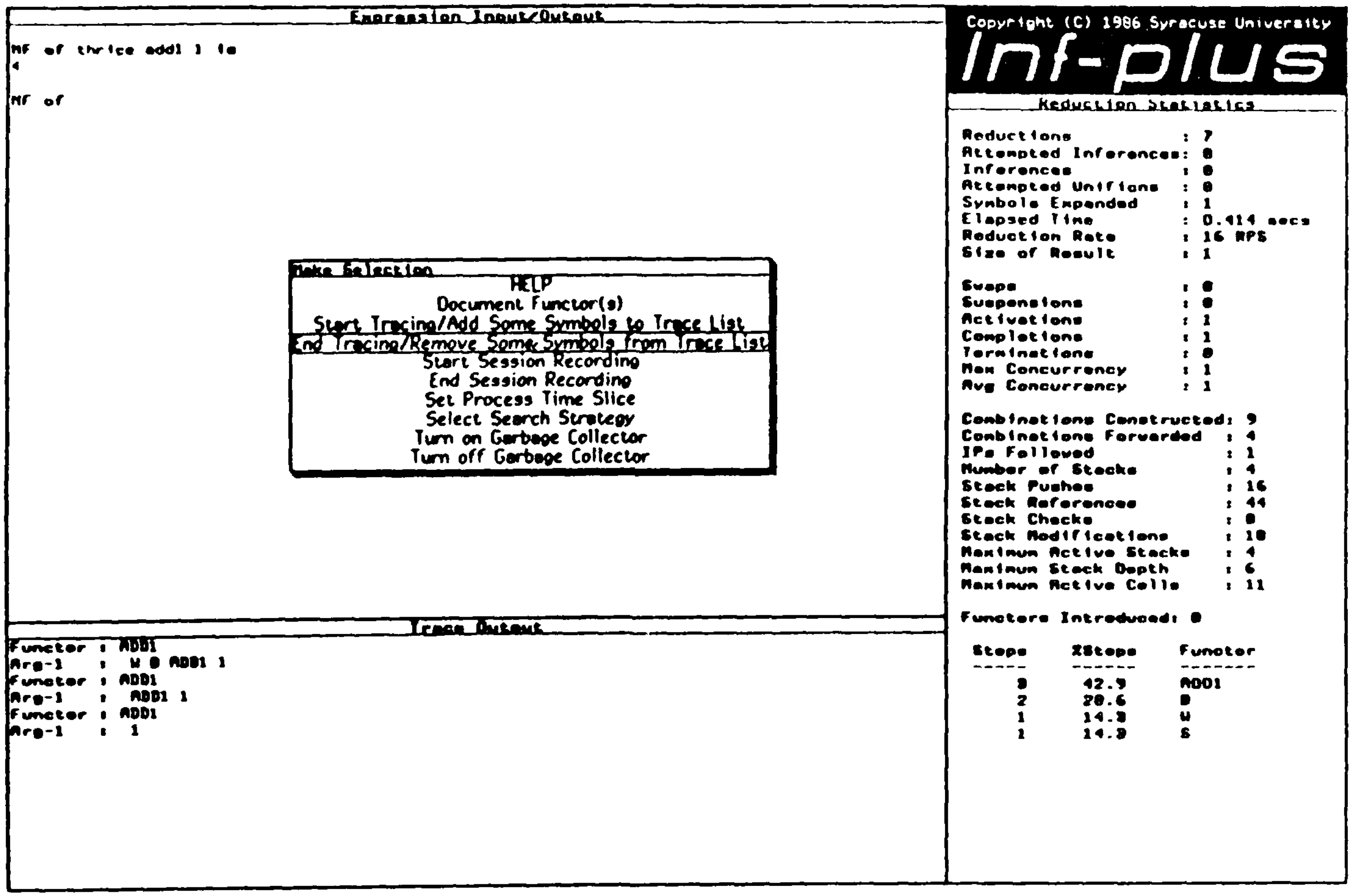
Statistic	*Meaning*
Reductions	number of reductions performed
Elapsed Time	time to perform reduction (compile time not included)
Reduction Rate	number of reductions performed per second
Size of Result	number of combinations + number of atoms
Max Concurrency	maximum length of process queue
Avg Concurrency	average length of process queue
Combinations Const.	number of combination cells created during the reduction

4.6 Recording sessions in files. Sessions with the LNF-Plus system may be recorded in files for later inspection. To begin recording, click once on the right mouse button, select the *Start Session Recording* option from the pop-up menu, and enter the name of the file into which the record of the session is to be placed. To stop recording, simply click once on the right mouse button and select the *End Session Recording* option.



```
Expression Input/Output

NF of thrice add1 1 is
4

NF of

Monitor Output
Initial Expression
     THRICE ADD1 1
Step: 1    RPs: 6    Fnctr: S    Fuded: 0    IPs Flwd 0    RPQ:0
     ACTIVE (B ADD1 (W B ADD1) 1) NIL
Step: 2    RPs: 7    Fnctr: B    Fuded: 0    IPs Flwd 0    RPQ:0
     ACTIVE (ADD1 (W B ADD1 1)) NIL
Step: 3    RPs: 8    Fnctr: W    Fuded: 0    IPs Flwd 0    RPQ:0
     ACTIVE (ADD1 (B ADD1 ADD1 1)) NIL
Step: 4    RPs: 9    Fnctr: B    Fuded: 0    IPs Flwd 0    RPQ:0
     ACTIVE (ADD1 (ADD1 (ADD1 1))) NIL
Step: 5    RPs: 9    Fnctr: ADD1    Fuded: 1    IPs Flwd 0    RPQ:0
     ACTIVE (ADD1 (ADD1 (IP 2))) NIL
Step: 6    RPs: 9    Fnctr: ADD1    Fuded: 2    IPs Flwd 0    RPQ:0
     ACTIVE (ADD1 (IP 3)) NIL
Step: 7    RPs: 9    Fnctr: ADD1    Fuded: 3    IPs Flwd 0    RPQ:0
     ACTIVE (IP 4) NIL
```

```
Copyright (C) 1986 Syracuse University
Inf-plus
Reduction Statistics

Reductions            : 7
Attempted Inferences: 0
Inferences            : 0
Attempted Unifions    : 0
Symbols Expanded      : 1
Elapsed Time          : 0.01213 secs
Reduction Rate        : 576 RPS
Size of Result        : 1

Swaps                 : 0
Suspensions           : 0
Activations           : 1
Completions           : 1
Terminations          : 0
Max Concurrency       : 1
Avg Concurrency       : 1

Combinations Constructed: 9
Combinations Forwarded  : 4
IPs Followed            : 1
Number of Stacks        : 4
Stack Pushes            : 16
Stack References        : 41
Stack Checks            : 0
Stack Modifications     : 10
Maximum Active Stacks   : 4
Maximum Stack Depth     : 6
Maximum Active Cells    : 11

Functors Introduced: 0

 Steps    %Steps    Functor
 -----    ------    -------
     3     42.9     ADD1
     2     28.6     B
     1     14.3     W
     1     14.3     S
```



```
NF of thrice add1 1 is
4

NF of
```

Make Selection
HELP
Document Functor(s)
Start Tracing/Add Some Symbols to Trace List
End Tracing/Remove Some Symbols from Trace List
Start Session Recording
End Session Recording
Set Process Time Slice
Select Search Strategy
Turn on Garbage Collector
Turn off Garbage Collector

```
Functor : ADD1
Arg-1   :  W B ADD1 1
Functor : ADD1
Arg-1   :  ADD1 1
Functor : ADD1
Arg-1   :  1
```

Copyright (C) 1986 Syracuse University

```
Reductions              : 7
Attempted Inferences: 0
Inferences              : 0
Attempted Unifions      : 0
Symbols Expanded        : 1
Elapsed Time            : 0.414 secs
Reduction Rate          : 16 RPS
Size of Result          : 1

Swaps                   : 0
Suspensions             : 0
Activations             : 1
Completions             : 1
Terminations            : 0
Max Concurrency         : 1
Avg Concurrency         : 1

Combinations Constructed: 9
Combinations Forwarded  : 4
IPs Followed            : 1
Number of Stacks        : 4
Stack Pushes            : 16
Stack References        : 44
Stack Checks            : 0
Stack Modifications     : 10
Maximum Active Stacks   : 4
Maximum Stack Depth     : 6
Maximum Active Cells    : 11

Functors Introduced: 0

 Steps     %Steps     Functor
 -----     ------     -------
     3      42.9      ADD1
     2      28.6      B
     1      14.3      W
     1      14.3      S
```

4.7 Controlling garbage collection. Garbage collection is performed by the Lisp Machine, not by the implementation of the LNF-Plus system. The garbage collector (the ephemeral garbage collector - an on the fly collector) is initially enabled. It may be disabled (or enabled) by clicking once on the right mouse button and then selecting the option *Turn on Garbage Collector* (*Turn off Garbage Collector*).

5. Reduction Processes and Simulated Concurrency. The LNF-Plus system implements a pseudo-parallel graph reduction system. That is to say it can simulate the behavior of a collection of graph reducers working concurrently. Most reductions only activate one reduction process. Many processes may be activated, however, during the reduction of ASA-expressions - it is for this purpose that the PAR annotation (placed by the system *not* the user) described below has been added. The implementation ideas can be found in [Hughes 1986b]. In this paper Hughes adds the annotation PAR (with the following semantics) to a sequential reducer of a purely functional language yielding an implementation in which (simulated) parallel reductions may take place.

The application `(PAR f x)` reduces to the same expression as does the expression `(f x)` but instead of reducing `(f x)` sequentially, the expression `(f x)` and its subexpression x are reduced concurrently. Of course, since there is only a single processor, this must be simulated. Informally, a process queue is maintained which contains the list of processes (represented by roots of the graph - remember expressions are programs in this system!) which would be active (on a multiprocessor system) but (in this single processor system simulating a multiprocessor environment) must wait for the (single) Lisp machine processor. Each process in the process queue runs (is reduced), in turn, for some time. A running process either finishes (gets reduced to lazy normal form) or runs out of its alloted fuel and is swapped out to the end of the queue. Each process is not actually given a *time slice* per se but, instead, is given an allotment of reduction steps that it may perform. Each process is given the same number of reduction steps - a number determined by the user. This number may be changed from problem to problem. Changing the reduction allocation is accomplished by clicking once on the right mouse button and selecting the *Select Process Time Slice* option and then providing a positive integer as the amount of *reduction fuel* each process will receive for subsequent reductions.

6. Sets. Set expressions are one of several very-high level constructs made available to programmers by the LNF-Plus language. Sets in the LNF-Plus implementation are, in reality, LNF-Plus lists with duplicates removed. For example, if the user asks for the normal form of the expression: {1,2,3,2,3,4,5} then LNF-Plus will respond with: [1,2,3,4,5].

A user may think of the LNF-Plus set: {...} as syntactic sugar for the application: (**mkset** [...]), where **mkset** is a functor which, when given a list *L* as input, returns as result a list *M*, where *M* is *L* with duplicates removed - the ordering of elements in *L* is preserved in *M*.

Since LNF-Plus sets are lists in disguise, all of the functions which accept lists as arguments (e.g. **hd**, **tl**, **nullp**, **append**, ...) also accept sets. For example, if the expression: (tl {2,2,3,3}) is reduced to normal form, the result would be the expression: [3] since the set {2,2,3,3} represents the list [2,3] and the tail of [2,3] is [3].

The LNF-Plus language provides a set union operator **union** which expects to be given two sets (lists without duplicates) and produces their union (again represented as a list). For example, the following expression: (union {1,2,3,4,5} {3,4,5,6}) has the expected normal form: [1,2,3,4,5,6].

In addition to *explicit set representations*, e.g. {e1,...,eN} where the elements of the set are spelled out, the language supports two flavors of *implicit set representations*: relative (Zermelo-Frankel) set abstraction expressions and absolute (Godel) set abstraction expressions.

6.1 Relative set abstraction. The usefulness of Zermelo-Frankel set expressions (ZF-expressions) in functional languages has been ably demonstrated by David Turner in several papers. ZF-expressions, as a construct in a functional language, first appeared in Turner's Kent Recursive Calculator (*KRC*) language. Turner has also made them a part of his latest functional language *Miranda*. The syntax of the ZF-expressions in LNF-Plus (which differs only slightly from Turner's) was inherited from Greene's LNF language. As an example, given the existence of a list of people (**people**) and predicates **male** and **smokes**, the set of all non-smoking males could be represented by the expression:

```
{m | m ε people; (male m); (not (smokes m))}.
```

The first occurrence of m in the above expression is called the *template*, the phrase m ε people is called a *generator* and the phrases (male m) and (not (smokes m)) are called *guards*. In general, a ZF-expression, takes the form:

{*template* | *generator*; *0-or-more-intermixed-guards-and-generators*}.

The only difference between the syntax for ZF-expressions in *KRC* (or *Miranda*) and LNF-Plus is the epsilon (ε) -- *KRC* uses an arrow (<-) . ZF-expressions are reduced to explicit sets by, essentially, a generate-and-test scheme. This scheme is implemented by translating the ZF-expression into an equivalent (variable free) expression involving the functors: **map**, **filter**, **flatmap**, and **enumerate**.

For example, the sample ZF-expression above is compiled into the following expression:

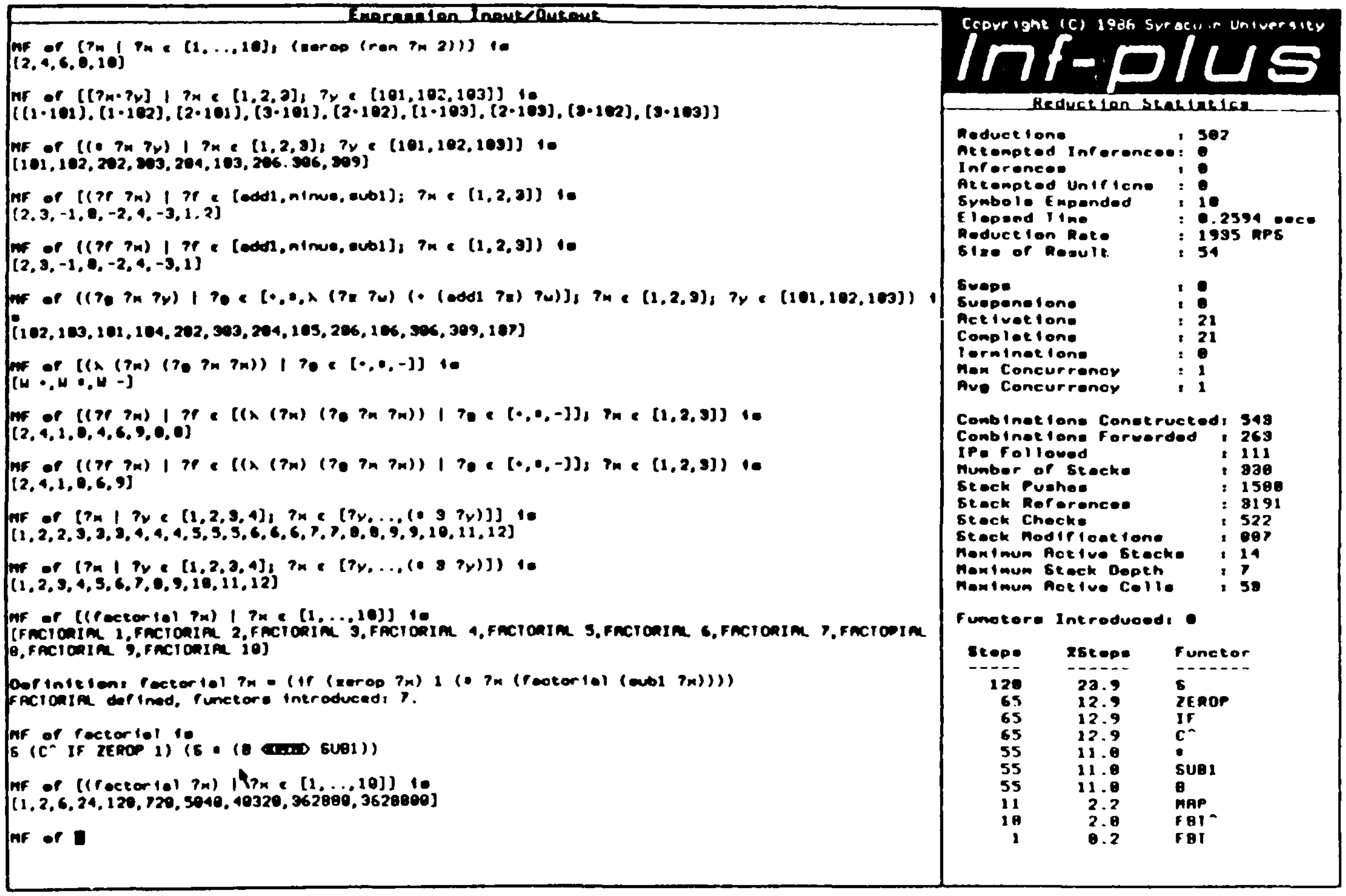
```
(MKSET (MAP I (FILTER (S^ AND MALE (B NOT SMOKES)) PEOPLE)))
```

before it is reduced to normal form. Note that all occurrences of the bound variable m have been removed. This is equivalent to the following expression in the λ-calculus:

```
(MKSET
 (MAP
  (λ m m)
  (FILTER
   (λ (m) (AND (MALE m) (NOT (SMOKES m))))
   PEOPLE))).
```

The *next page* is *another* LNF-Plus *session snapshot*; this time showing some examples of the use to which ZF-expressions may be put.

6.2 Absolute set abstraction. *Sets described by absolute set abstraction* (called ASA-expressions) are reduced by a wholly different mechanism. The ASA-expression is the vehicle by which programmers invoke the Horn Clause resolution mechanism available in the language.



```
NF of [?x | ?x ε [1,...,10]; (zerop (rem ?x 2))] is
[2,4,6,8,10]

NF of [[?x·?y] | ?x ε [1,2,3]; ?y ε [101,102,103]] is
[[1·101],[1·102],[2·101],[3·101],[2·102],[1·103],[2·103],[3·102],[3·103]]

NF of [(* ?x ?y) | ?x ε [1,2,3]; ?y ε [101,102,103]] is
[101,102,202,303,204,103,206.306,309]

NF of [(?f ?x) | ?f ε [add1,minus,sub1]; ?x ε [1,2,3]] is
[2,3,-1,0,-2,4,-3,1,2]

NF of {(?f ?x) | ?f ε [add1,minus,sub1]; ?x ε [1,2,3]} is
[2,3,-1,0,-2,4,-3,1]

NF of {(?g ?x ?y) | ?g ε [+,*,λ (?z ?w) (+ (add1 ?z) ?w)]; ?x ε [1,2,3]; ?y ε [101,102,103]} i
s
[102,103,101,104,202,303,204,105,206,106,306,309,107]

NF of [(λ (?x) (?g ?x ?x)) | ?g ε [+,*,-]] is
[W +,W *,W -]

NF of [(?f ?x) | ?f ε [(λ (?x) (?g ?x ?x)) | ?g ε [+,*,-]]; ?x ε [1,2,3]] is
[2,4,1,0,4,6,9,0,0]

NF of {(?f ?x) | ?f ε [(λ (?x) (?g ?x ?x)) | ?g ε [+,*,-]]; ?x ε [1,2,3]} is
[2,4,1,0,6,9]

NF of [?x | ?y ε [1,2,3,4]; ?x ε [?y,...,(* 3 ?y)]] is
[1,2,2,3,3,3,4,4,4,5,5,5,6,6,6,7,7,8,8,9,9,10,11,12]

NF of {?x | ?y ε [1,2,3,4]; ?x ε [?y,...,(* 3 ?y)]} is
{1,2,3,4,5,6,7,8,9,10,11,12}

NF of [(factorial ?x) | ?x ε [1,...,10]] is
[FACTORIAL 1,FACTORIAL 2,FACTORIAL 3,FACTORIAL 4,FACTORIAL 5,FACTORIAL 6,FACTORIAL 7,FACTOPIAL
8,FACTORIAL 9,FACTORIAL 10]

Definition: factorial ?x = (if (zerop ?x) 1 (* ?x (factorial (subl ?x))))
FACTORIAL defined, functors introduced: 7.

NF of factorial is
S (C^ IF ZEROP 1) (S * (B [illegible] SUB1))

NF of [(factorial ?x) | ?x ε [1,...,10]] is
[1,2,6,24,120,720,5040,40320,362880,3628800]

NF of █
```

```
Reductions                : 502
Attempted Inferences: 0
Inferences                : 0
Attempted Unificns   : 0
Symbols Expanded     : 10
Elapsed Time          : 0.2594 secs
Reduction Rate        : 1935 RPS
Size of Result         : 54

Swaps                  : 0
Suspensions            : 0
Activations            : 21
Completions            : 21
Terminations          : 0
Max Concurrency       : 1
Avg Concurrency       : 1

Combinations Constructed: 548
Combinations Forwarded  : 263
IPs Followed                 : 111
Number of Stacks          : 838
Stack Pushes                 : 1508
Stack References          : 8191
Stack Checks                : 522
Stack Modifications       : 887
Maximum Active Stacks   : 14
Maximum Stack Depth     : 7
Maximum Active Cells     : 58

Functors Introduced: 0

 Steps     %Steps    Functor
 -----     ------    -------
   120       23.9    S
    65       12.9    ZEROP
    65       12.9    IF
    65       12.9    C^
    55       11.0    *
    55       11.0    SUB1
    55       11.0    B
    11        2.2    MAP
    10        2.0    FBI^
     1        0.2    FBI
```

In general, an ASA-expression, takes the form:

{*template* | ∃ (*variables*) *predication1* & ... & *predicationP*}

where

- *template* is any LNF-Plus expression, its free variables are considered binding instances whose scope is the set's conjunction part,
- *variables* are auxiliary variables with the same scope (as the template variables), and
- each *predication* is a LNF-Plus expression.

Instead of giving a blow-by-blow account of the low level steps involved in the reduction of ASA-expressions, an imprecise but informative high level description of their reduction will be given. Let the following expression be our prototypical ASA-expression:

X: {*template* | ∃ (*variables*) *pred* & *restps*}.

The meta-variables *pred* and *restps* stand for the first and remaining predications of the conjunction respectively. X is reduced by first reducing *pred* to lazy normal form. If this happens to turn out to be the atom **true**, then X reduces to:

{*template* | ∃ (*variables*) *restps*}.

If *pred* reduces to the atom **false**, then X reduces to {} (represented by []). If, however, *pred* reduces to an expression of the form **(p a1 ... ak)** (call it *predr*) and p has been defined by a collection of Horn clauses (call it *assertions* - whose first clause is *head* ← *body*), then X reduces to the following union of two subsets:

```
(union
  {template | ∃ (variables1) body & restps}
  (constrained-set template variables predr restps remaining-assertions))
```

if *predr* unifies with *head*, and to:

```
  (constrained-set template variables predr restps remaining-assertions))
```

if *predr* and *head* fail to unify.

A *constrained set* expression differs from an ASA-expression (or unconstrained set expression) by limiting the ways a single predication may be proved. This distinction is illustrated by the following simple example. Given the three Horn clauses defining the unary predicate **person**:

```
((person Kevin) ←)
((person Alan) ←)
((person Tracy) ←)
```

the (unconstrained) set expression **{x | (person x)}** reduces to the three-list **[Kevin,Alan,Tracy]** but the constrained set expression

```
(constrained-set
  x                    ; template
  []                   ; variables
  (person x)           ; selected goal
  []                   ; remaining goals
  [((person Alan) ←),((person Tracy) ←)]   ; remaining assertions
  )
```

reduces to only the two-list **[Alan,Tracy]**.

The set is constrained by being limited to using only the *remaining available assertions* when attempting to prove *the selected goal.* Other occurrences of the goal (either in *restps* or predications generated later) are not thus constrained.

The functor **union** is realized by creating a reduction process for each of the subsets and letting them reach lazy normal form concurrently. Using this technique, the *depth first runaway* problem of Prolog imlementations is avoided as solutions found in the reduction of the subsets are added to the union *as they are found* ! Since the set is constructed lazily (elements of the set are computed only as needed), sets with infinitely many members may be specified by ASA-expressions and used in larger computations which only require a finite number of their members.

6.3 Unification. The unification algorithm employed by the functors which perform resolution inferences is a rather sophisticated one. Its job is to determine the truth of statements of the form: ∃ VS A=B where VS are the variables existentially quantified and A and B are expressions in lazy normal form. The algorithm may answer either NO, YES, or YES under the condition C is true. It answers NO in the case that there are no *expressions* which can substituted for the *variables* in VS in the expressions A and B which make A and B identical, it answers YES if it can find such expressions, and it answers conditionally YES when it can find such expressions but only if some other equations can be solved. Some examples of inputs and outputs might clarify the actions of the algorithm.

Inputs			Output
VS: ()	A: 3	B: 3	YES
VS: ()	A: 2	B: 3	NO
VS: (?x)	A: 3	B: ?x	YES
VS: ()	A: 3	B: ?x	YES, if ?x=3
VS: ()	A: [1•(add1 3)]	B:[(sub1 2)•4]	YES
VS: ()	A: [1•(add1 3)]	B:[(sub1 3)•4]	NO
VS: (?x)	A: [?x•(add1 3)]	B:[(sub1 2)•4]	YES
VS: ()	A: [?x•(add1 3)]	B:[(sub1 2)•4]	YES, if ?x=1
VS: ()	A: [?x•(add1 3)]	B:[(sub1 2)•5]	NO
VS: (?x)	A: [1•(add1 ?x)]	B:[(sub1 2)•?y]	YES, if (add1 ?x)=?y

The conditional YES answers are given when not enough information (about one of the equatees) has been given to determine *unconditionally* if the equation can be solved. If some of the equation's free variables (those variables in A or B but not in VS) become (at some later time in the computation) instantiated, then these conditional answers may become definite. These equations (the conjunction of which make up the condition C mentioned above) returned by the unfication algorithm are added to the end of the current goal list. It is hoped that by the time these equations become selected that enough of their free variables will have become instantiated so as to be able to determine whether or not they can be solved.

The Lisp realization of the system's unification algorithm (copied in from the system's source code file *lnf-plus:sys;sets.lisp*) is displayed below:

```
(defun unify (x y bindable-vars &optional eqns)
 "returns NIL if unable to unify x and y in the scope of
bindable-variables, or makes x and y unifiable (via graph
modification) and returns possibly reduced bindable-vars
structure and equation list"
  (cond ((same-atoms x y)
      ;; equation of the form: a=a (a, an atom) so
      ;; return an unconditional YES
      (values bindable-vars eqns))
      ((and (variable-p x) (bindable x bindable-vars))
      ;; equation of the form: v=E (v, a variable in VS) so
      ;; bind v to E, return an unconditional YES
      (values (bind x y bindable-vars) eqns))
      ((and (variable-p y) (bindable y bindable-vars))
      ;; equation of the form: E=v (v, a variable in VS) so
      ;; bind v to E, return an unconditional YES
      (values (bind y x bindable-vars) eqns))
      ((or (unknownp x) (unknownp y))
      ;; equation of the form: unk=E or E=unk
      ;; (i.e. not enough info to make a decision), so
      ;; return a conditional YES
      (values bindable-vars (cons (make-equation x y) eqns)))
      ((combinationp x)
      ;; equation of the form (oprx opdx)=Y, so check on Y's form
      (cond ((combinationp y)
            ;; Y also a combination, say (opry opdy) so split
            ;; the job into two parts: oprx=opry and
            ;; (LNF-OF opdx)=(LNF-OF opdy)
            (multiple-value-bind (new-bvs extended-eqns)
               (unify (operator x) (operator y) bindable-vars eqns)
              (if new-bvs
                ;; oprx=opry equation can be solved, so try to solve
                ;; (LNF-OF opdx)=(LNF-OF opdy)
                (unify (lnf-of-subexp (operand x))
                     (lnf-of-subexp (operand y))
                     new-bvs
                     extended-eqns))))))
       ;; otherwise, fail
       ))
```

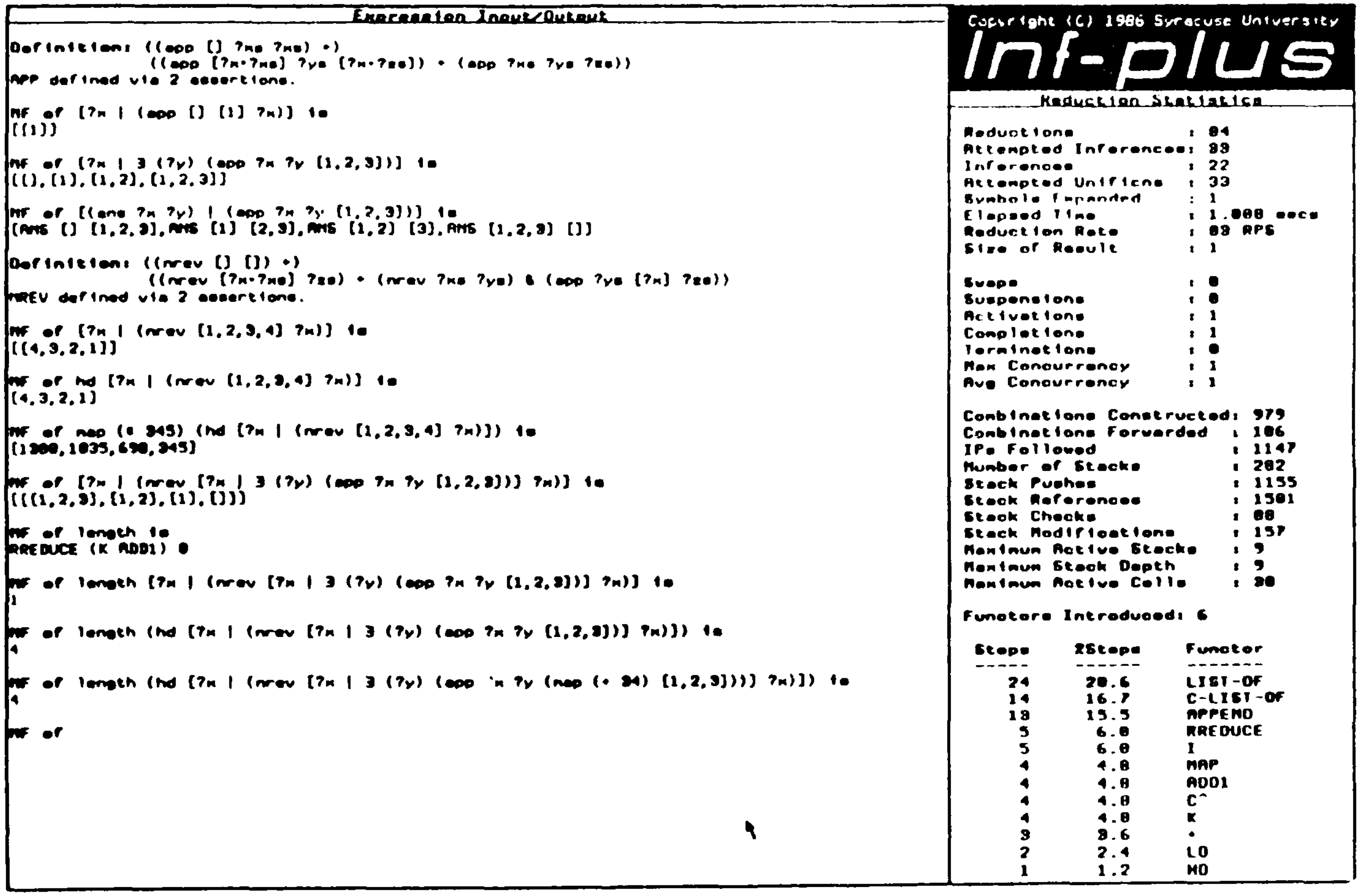
Goals of the form (= A B) in the goal list of an ASA-expression are solved via the unification algorithm above. For example, given the ASA-expression:

```
{?x | ∃ (?y) (= (add1 ?x) (sub1 ?y)) &
         (= (factorial 4) ?x) &
         (= ?y 26)}
```

as input, the following reduction sequence would take place:

```
{?x | ∃ (?y) (= (factorial 4) ?x) &
         (= ?y 26) &
         (= (add1 ?x) (sub1 ?y))}

{24 | ∃ (?y) (= ?y 26) &
         (= 25 (sub1 ?y))}

{24 | ∃ () (= 25 25)}

{24}.
```

The next page demonstrates how some simple predicates can be defined and used in the specification and reduction of ASA-expressions.



```
                              Expression Input/Output

Definition: ((app [] ?xs ?xs) ←)
            ((app [?x·?xs] ?ys [?x·?zs]) ← (app ?xs ?ys ?zs))
APP defined via 2 assertions.

NF of [?x | (app [] [1] ?x)] is
[[1]]

NF of [?x | ∃ (?y) (app ?x ?y [1,2,3])] is
[[],[1],[1,2],[1,2,3]]

NF of [(ans ?x ?y) | (app ?x ?y [1,2,3])] is
[ANS [] [1,2,3],ANS [1] [2,3],ANS [1,2] [3],ANS [1,2,3] []]

Definition: ((nrev [] []) ←)
            ((nrev [?x·?xs] ?zs) ← (nrev ?xs ?ys) & (app ?ys [?x] ?zs))
NREV defined via 2 assertions.

NF of [?x | (nrev [1,2,3,4] ?x)] is
[[4,3,2,1]]

NF of hd [?x | (nrev [1,2,3,4] ?x)] is
[4,3,2,1]

NF of map (* 345) (hd [?x | (nrev [1,2,3,4] ?x)]) is
[1380,1035,690,345]

NF of [?x | (nrev [?x | ∃ (?y) (app ?x ?y [1,2,3])] ?x)] is
[[[1,2,3],[1,2],[1],[]]]

NF of length is
RREDUCE (K ADD1) 0

NF of length [?x | (nrev [?x | ∃ (?y) (app ?x ?y [1,2,3])] ?x)] is
1

NF of length (hd [?x | (nrev [?x | ∃ (?y) (app ?x ?y [1,2,3])] ?x)]) is
4

NF of length (hd [?x | (nrev [?x | ∃ (?y) (app `x ?y (map (+ 34) [1,2,3]))] ?x)]) is
4

NF of
```

```
Copyright (C) 1986 Syracuse University
Inf-plus
         Reduction Statistics

Reductions                : 84
Attempted Inferences: 33
Inferences                : 22
Attempted Unificns        : 33
Symbols Expanded          : 1
Elapsed Time              : 1.008 secs
Reduction Rate            : 83 RPS
Size of Result            : 1

Swaps                     : 0
Suspensions               : 0
Activations               : 1
Completions               : 1
Terminations              : 0
Max Concurrency           : 1
Avg Concurrency           : 1

Combinations Constructed: 979
Combinations Forwarded    : 106
IPs Followed              : 1147
Number of Stacks          : 282
Stack Pushes              : 1155
Stack References          : 1501
Stack Checks              : 88
Stack Modifications       : 157
Maximum Active Stacks     : 9
Maximum Stack Depth       : 9
Maximum Active Cells      : 30

Functors Introduced: 6

 Steps     %Steps     Functor
 -----     ------     -------
    24       28.6     LIST-OF
    14       16.7     C-LIST-OF
    13       15.5     APPEND
     5        6.0     RREDUCE
     5        6.0     I
     4        4.8     MAP
     4        4.8     ADD1
     4        4.8     C^
     4        4.8     K
     3        3.6     *
     2        2.4     LO
     1        1.2     HO
```

References

[Bird 1986]

Bird R, Wadler P, DRAFT COPY OF *An Introduction to Functional Programming*, July 1986, Programming Research Group, Oxford University.

[Greene 1985]

Greene KJ, *A Fully Lazy Higher Order Purely Functional Programming Language with Reduction Semantics*, August 1985, CASE Center Technical Report No. 8503, Syracuse University.

[Henderson 1982]

Henderson P, *Functional Geometry*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming.

[Hughes 1986a]

Hughes RJM, *Why Functional Programming Matters*, January 1986, Programming Methodology Group Memo PMG-40, Chalmers Tekniska Hogskola, Goteborg, Sweden.

[Hughes 1986b]

Hughes RJM, *A Simple Implementation of Concurrent Graph Reduction*, September 1986, Proceedings of the Workshop on Graph Reduction (to be published), Santa Fe, New Mexico.

[Robinson 1984]

Robinson JA & Sibert EE, *The LogLisp Programming System*, March 1984, Technical Report, Logic Programming Research Center, Syracuse University.

[Robinson 1987]

Robinson JA & Greene KJ, *New Generation Knowledge Processing - Volume 1*, January 1987, Syracuse University.

[Sterling 1986]

Sterling L & Shapiro E, *The Art of Prolog*, The MIT Press.

Appendix 1 - LNF-Plus Reduction Rules

Combinators

```
S f g x → f x (g x)
K x y → x
I x → x
B f g x → f (g x)
C f g x → f x g
W f x → f x x
Y f → f (f (f ...))
R x y → y x
S^ k a g x → k (a x) (g x)
B^ k a g x → k a (g x)
NB^ k a g x → k (a (g x))
C^ k a g x → k (a x) g
```

Arithmetic Functors

```
NUMBERP n → TRUE
NUMBERP cfn → FALSE, if cfn not a number
+ n m → n+m
- n m → n-m
ADD1 n → n+1
SUB1 n → n-1
MINUS n → -n
* n m → n*m
EXP i j → the integer 'i to the j', if j≥0
EXP i j → the float 'i to the j', if j<0
EXP s i → the float 's to the i'
EXP n s → the float 'n to the s'
DIV n m → n/m, if m≠0
IDIV n m → integral quotient after n/m, if m≠0
REM n m → remainder after n/m, if m≠0
< n m → n<m
> n m → n>m
ZEROP n → n=0
```

Boolean Functors

```
BOOLEANP b → TRUE
BOOLEANP cfn → FALSE, if cfn not a boolean
OR TRUE y → TRUE
OR FALSE b → b
AND FALSE y → FALSE
AND TRUE b → b
NOT TRUE → FALSE
NOT FALSE → TRUE
IF TRUE a b → a
IF FALSE a b → b
```

List Oriented Functors

```
HD [x•y] → x
TL [x•y] → y
APPEND [] list → list
APPEND [x•xs] list → [x•(APPEND xs list)]
NTH 1 [x•xs] → x
NTH n [x•xs] → NTH (n-1) xs, if n>1
MAP f [x•xs] → [(f x)•(MAP f xs)]
MAP f [] → []
FILTER p [] → []
FILTER p [x•xs] → IF (p x) [x•(FILTER p xs)] (FILTER p xs)
MEMBER x [] → FALSE
MEMBER x [z•zs] → OR x=z (MEMBER x zs)
REDUCE f [x] → x
REDUCE f [x,y•r] → (f x (REDUCE f [y•r]))
RREDUCE f nv [] → nv
RREDUCE f nv [x•r] → (f x (RREDUCE f nv r))
LREDUCE f acc [] → acc
LREDUCE f acc [x•r] → (LREDUCE f (f acc x) r)
ACCUMULATE f acc [] → [acc]
ACCUMULATE f acc [x•r] → [acc•(ACCUMULATE f (f x acc) r)]
ITERATE f x → [x•(iterate f (f x))]
MKSET list → removes duplicate elements from list
INTERLEAVE [x•r] list → [x•INTERLEAVE list r]
INTERLEAVE [] list → list
FLATMAP f [x•xs] → INTERLEAVE (f x) (FLATMAP f xs)
FLATMAP f [] → []
```

```
UNSAFE-MERGE [a•rest] y → [a•(PAR (UNSAFE-MERGE y) rest)]
UNSAFE-MERGE [] y       → y
UNSAFE-MERGE y [a•rest] → [a•(PAR (UNSAFE-MERGE y) rest)]
UNSAFE-MERGE y []       → y
 otherwise swap this process out
ND-MERGE x y → PAR (PAR UNSAFE-MERGE x) y
```

Resolution Oriented Functors

```
LO n m t-fn p-list-fn →
 LIST-OF <tv> (t-fn vars-n) nm-ht (p-list-fn vars-n-m)
LIST-OF var template vars goals →
  [intantiated-template•LIST-OF var' template' vars' goals'] or []
C-LIST-OF var template vars goals assertions →
  [intantiated-template•rest] or []
```

Other Functors

```
= cf1 cf2 → cf1=cf2
= cfn1 cfn2 →
 AND (= (OPERATOR cfn1) (OPERATOR cfn2))
      (= (OPERAND cfn1) (OPERAND cfn2))
e< cn1 cn2 → cn1 'less than (in the lexicographic ordering)' cn2
NULLP cfn → if (= [] cfn) TRUE FALSE
ATOMP cfn → num-args[cfn]=0
PAIRP cfn → if cfn of the form [x•y] then TRUE else FALSE
FB n 0 → [n,n,..]
FB n m → [n•(FB^ (+ n m) m)], if m≠0
FBT n 0 lim → [n,n,..]
FBT n m lim →
  if (≤ n lim) then [n•(FBT^ (+ n m) m lim)] else [], m>0
FBT n m lim →
  if (≥ n lim) then [n•(FBT^ (+ n m) m lim)] else [], m<0
FB^ n m → [n•(FB^ (+ n m) m)]
COMBINATIONP cn → if cn a combination then TRUE else FALSE
A-S c n f (c A1  .. An) → f A1 ... An
A-S^ c n f (c A1  .. An) → f A1 ... An
A-S-E c n then-exp else-exp test-exp →
 IF (AND (= c (initial-atom test-exp))
   (= n (number-of-args test-exp)))
   then-exp
   else-exp
```

```
A-S-E c n then-exp else-exp test-exp →
 IF (AND (= c (initial-atom test-exp))
         (= n (number-of-args test-exp)))
   then-exp
   else-exp
ARG n (c-or-f e1 e2 ... eM) → en
CONSTRUCTOR (c e1 e2 ... eN) → c
CONSTRUCTIONP (c e1 e2 ... eN) → (constructor-p c)
FUNCTIONP (c e1 e2 ... eN) → (functor-p c)
UNKNOWNP exp → (not (or (functionp exp) (constructionp exp)))
ARITY (a e1 e2 ... eN) → (MAX 0 (- (arity a) n))
NUM-ARGS (c-or-f e1 e2 ... eN) → N
APP-TO-ARGS n f exp → f (ARG 1 exp) ... (ARG n exp)
UNION set1 set2 → union of the two lists representing sets
PAR f x → f p:(ACTIVE x plist) with p put on end of *RPQ*
PRIM-REC donep op next base ds →
 (IF (donep ds)
   base
   (op ds (PRIM-REC donep op next base (next ds))))
```

Appendix 2 - The Standard Prelude

The standard prelude is a file (*lnf-plus:exs;standard-prelude.lisp*) which contains definitions of many of the most commonly used functions. This is a file which may be loaded by the user by clicking twice on the left mouse button and then selecting the *Load Standard Prelude* option from the pop-up menu.

In order two make LNF-Plus definitions readable by the Lisp reader, some characters must be *slashified*. These three characters are: ",", "|" and ";". Another note: the infix functional composition operator prints as ⊕ - sorry.

The contents of the standard prelude follows:

```
;; THREE FUNCTIONS FOR LISP HACKERS

;; Lisp's cons
(define (cons ?l ?r) [?l•?r])

;; Lisp's car
(define car hd)

;; Lisp's cdr
(define cdr tl)

;; first n elements of a list
(define (first ?n ?list)
  (if (or (< ?n 1) (nullp ?list))
     []
     [(hd ?list)•(first (sub1 ?n) (tl ?list))]))

;; last element of a non-empty list
(define (last [?x•?r])
      (if (nullp ?r) ?x (last ?r)))

;; determines the length of a list
(define length
  (lreduce (λ (?x ?y) (add1 ?x)) 0))

;; prefix of list L, each element of which satisfies P,
;; but the next element of L does not.
(define
  (take-while ?p)
  (rreduce (λ (?x) (if (?p ?x) (pair ?x) (k []))) []))
```

```
;; Prefix of list L, each element of which fails to satisfy P, except
;; for the last, which does.
;; P.S. If no element satisfies P, then output list will be all of L
(define
 (until ?p) (rreduce (λ (?x) (if (?p ?x) (k [?x]) (pair ?x))) []))

;; list suffix of L, all elements following first to satisfy P
;; P.S. We assume that at least one does.
(define (after ?p [?x•?r])
 (if (?p ?x) ?r (after ?p ?r)))

;; prefix consed to suffix
(define (until-and-after [?x•?r] ?p)
 (if (?p ?x)
     [[?x]•?r]
   (add-to-prefix ?x (until-and-after ?r ?p))))

(define (add-to-prefix ?x [?list•?r])
 [[?x•?list]•?r])

;; zip two lists into one
(define
 (zip [?x•?XS] [?Y•?YS]) [[?x•?Y]•(zip ?XS ?YS)]
 (zip ? ?) [])

(define (cartesian-product ?list1 ?list2)
 (for-each ?x ε ?list1 and ?y ε ?list2 instantiate [?x•?y]))

;; lookup in an association list
(define
 (assq ?item)
 (rreduce (λ (?hd) (if (= ?item (hd ?hd)) ?hd)) []))

;; predicate which returns true iff (pred el) true
;; for each element el in list
(define (true-for-all ?pred)
 (rreduce (λ (?x ?y) (and (?pred ?x) ?y)) true))

;; true iff length of list > 1
(define (more-than-one-in ?list)
 (and (pairp ?list) (pairp (tl ?list))))
```

```
;; deletes the nth element from a non empty list
(define (delete-nth ?n [?x•?r])
 (if (= 1 ?n) ?r [?x • delete-nth (sub1 ?n) ?r]))

;; sums a list of numbers
(define sum (lreduce + 0))

;; multiplies a list of numbers
;; NB: if 0 a member of the list,
;; then the function immediately returns 0.
(define product
 (rreduce (λ (?x) (if (zerop ?x) (k 0) (* ?x))) 1))

;; appends a list of lists
(define append-list (rreduce append []))

;; like flatmap, but appends instead of interleaves
(define (fmap ?f) (rreduce (append⊕?f) []))
;; alternate definition of FLATMAP, works almost as fast!
(define (flatmap2 ?f) (rreduce (interleave⊕?f) []))

;; and's a list
(define alltrue (rreduce and true))

;; or's a list
(define anytrue (rreduce or false))

;; sorts a list, using the quicksort algorithm
(define (quicksort ?list)
  (if (nullp ?list)
     []
    (append (quicksort (filter (> ?head) ?tail))
          [?head • (quicksort (filter (not⊕(> ?head)) ?tail))]
        where [?head•?tail] = ?list)))

;; list difference
(define (ldiff ?l1 ?l2)
  (if (nullp ?l2)
   ?l1
   (if (member ?l1 (hd ?l2))
    (remove (hd ?l2) (ldiff ?l1 (tl ?l2)))
    (ldiff ?l1 (?tl ?l2)))))
```

```
;;removes element ?x from list ?r
(define (remove ?item)
 (prim-rec
  nullp
  (λ (?list) (if (= ?item (hd ?list)) (tl ?list) (pair (hd ?list))))
  tl
  []))

;; all permutations of a list
(define (perms ?list)
 (if (nullp ?list)
    [[]]
   [[?a•?perm] /| ?a ε ?list /; ?perm ε perms (remove ?a ?list)]))

;; reverses (naively) a list
(define (slow-reverse ?list)
 (if (nullp ?list)
    []
   (append (slow-reverse (tl ?list)) [(hd ?list)])))

;; reverses a list
(define fast-reverse (lreduce (λ (?x ?y) [?y•?x]) []))

;; lisp's CONDitional (almost)
(define cond (rreduce (λ (?x ?y) (if (hd ?x) (tl ?x) ?y)) undef))

;; SOME FUNCTIONS DEALING WITH NUMBERS

;; even predicate
(define (even ?n)
 (zerop (rem ?n 2)))

;; odd predicate
(define (odd ?n) (not (even ?n)))

;; maximum of two numbers
(define (max ?m ?n)
 (if (> ?m ?n) ?m ?n))

;; maximal element of a list
(define maximum (reduce max))
```

```
;; minimum of two numbers
(define (min ?m ?n)
 (if (< ?m ?n) ?m ?n))

;, minimal element of a list
(define minimum (reduce min))

;; integer square root function (not too smart)
(define (integer-square-root ?n)
 ;; exp n 1\2 rarely is an integer
 ((if (nullp ?root-list)
     doesnt-exist
     (hd ?root-list))
  where ?root-list =
    [?x/|?xε[1/,../,(add1 (exp ?n 1\2))] /; (= ?n (* ?x ?x))]))

;; absolute value function
(define (abs ?x)
 (if (< 0 ?x) ?x (minus ?x)))

;; nth power of function f
(define (nth-power ?f ?n) (λ (?x) (nth (add1 ?n) (iterate ?f ?x))))

;; a higher order combining form:
;; Similar to RREDUCE on lists.
(define (num-red ?f) (prim-rec zerop ?f sub1))

;; some defns of factorial
(define factorial  (num-red * 1))
(define (factorial2 ?n) (lreduce * 1 [1/,../,?n]))
(define (factorial3 ?n)
     (if (zerop ?n) 1 (* ?n (factorial3 (sub1 ?n)))))

(define (nth-power2 ?f) (num-red (λ (?x) (b ?f)) i))
```

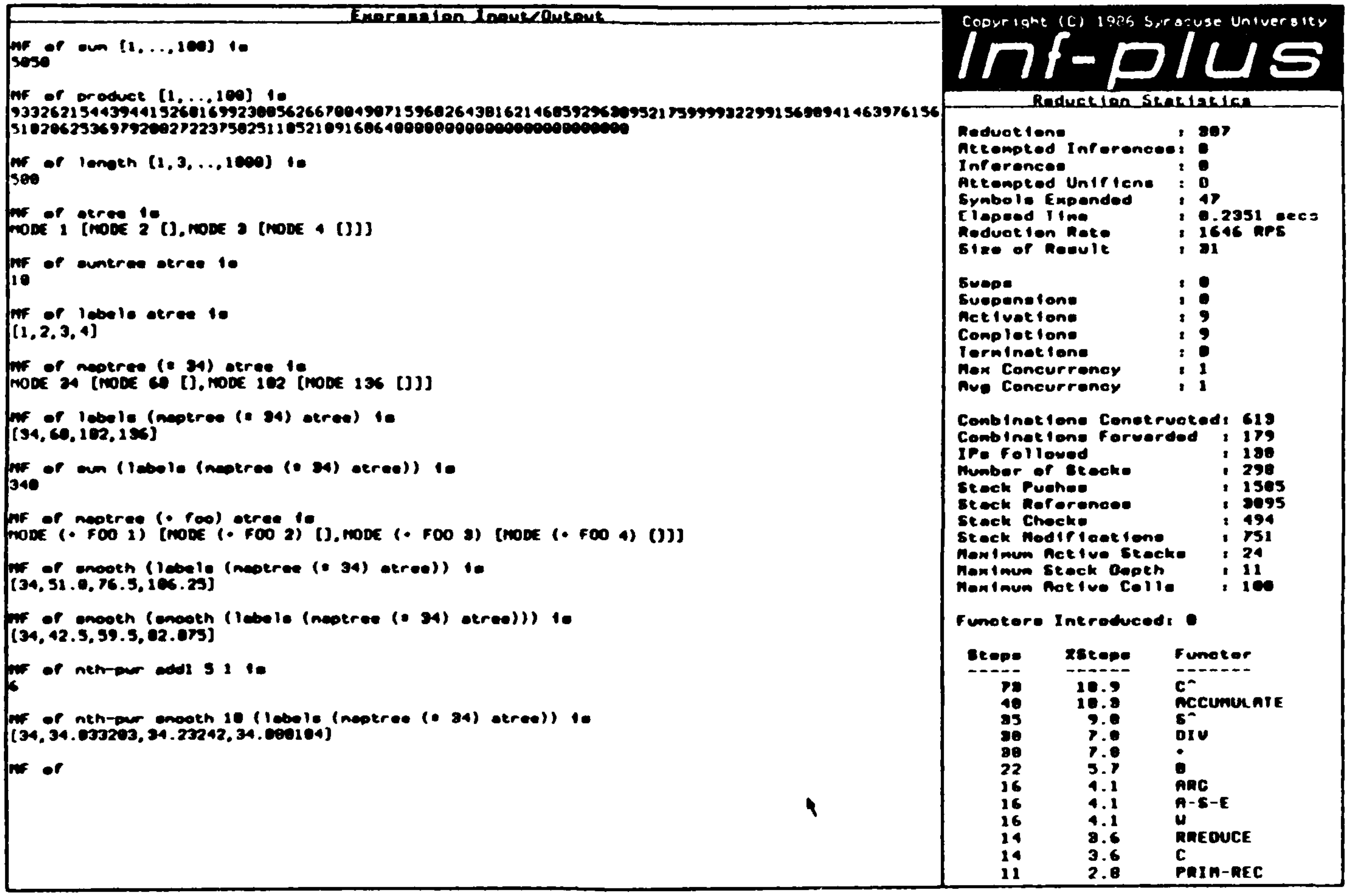
Appendix 3 - Examples.

The following pages are alternately copies of Lisp Machine files containing symbol definitions and snapshots of LNF-Plus sessions making use of them.

The examples are in order:

- utility of higher order functions in declarative programming
- Root Finding [Hughes 1986a] (one snapshot for four examples)
- Matrices (matrices as lists of lists)
- Polya's Sigma Function (demonstrates the importance of memoizing)
- Graph Traversal (using both FP and LP)
- Peter Henderson's Functional Geometry
- Lee routing (adapted from LP implementaion [Sterling 1986])
  our implementation is purely functional

```
;;; These examples illustrate the usefulness of higher order functions
;;; Some of the examples below taken from RJM Hughes' paper:
;;;   "Why Functional Programming Matters"
;;;   Program Methodology Group Memo PGM-40

;;; The definitions of four useful higher-order functions:
;;; (built-in functors of the LNF-Plus system)
;;; REDUCE f [x] = x
;;; REDUCE f [x1,x2•xs] = (f x1 (REDUCE f [x2•xs]))
;;; RREDUCE f a [] = a
;;; RREDUCE f a [x•xs] = (f x (RREDUCE f a xs))
;;; LREDUCE f a [] = a
;;; LREDUCE f a [x•xs] = LREDUCE f (f a x) xs
;;; ACCUMULATE f a [] = [a]
;;; ACCUMULATE f a [x•xs] = [a•(ACCUMULATE f (f a x) xs)]
;;; ITERATE f a = [a•(ITERATE f (f a))]

;;; SUM sums a list of numbers:
(DEFINE SUM (LREDUCE + 0))

;;; PRODUCT multiplies a list of numbers together:
(DEFINE PRODUCT (LREDUCE * 1))

;;; The infamous FACTORIAL function, defined using PRODUCT:
(DEFINE (FACTORIAL ?N) (PRODUCT [1/,../,?N]))

;;; ANYTRUE returns TRUE iff at least one
;;; element of the input list is TRUE:
(DEFINE ANYTRUE (RREDUCE OR FALSE))

;;; ALLTRUE returns TRUE iff all elements of the
;;; input list are TRUE:
(DEFINE ALLTRUE (RREDUCE AND TRUE))

;;; an alternate definition of APPEND
;;; (it's a built-in in LNF-Plus):
(DEFINE (APPEND-VIA-RREDUCE ?L1 ?L2) (RREDUCE PAIR ?L2 ?L1))
;;; NB(1): APPEND-VIA-RREDUCE compiles to the
;;;   very compact code: (C (RREDUCE PAIR))!
;;; NB(2): APPEND-VIA-RREDUCE executes AS FAST AS
;;;   the built-in APPEND!
;;; MAP is another built-in that could
;;; have been defined with RREDUCE:
(DEFINE (MAP-VIA-RREDUCE ?F) (RREDUCE (PAIR⊕?F) []))
;;; NB: This definition is not quite as efficient
;;;   as the built-in (about 15-20% worse)

;;; LENGTH returns the length of the input list:
(DEFINE LENGTH (RREDUCE (K ADD1) 0))

;;; REV reverses a list!
(DEFINE REV (LREDUCE (C PAIR) []))

;;; if TREES are represented as (NODE x (LISTOF trees))
;;; where x is the label on the root of the tree
;;; and trees are the immediate offspring of x,
;;; then the following higher-order function is to
;;; trees as RREDUCE is to lists
(DEFINE
  (REDTREE ?F ?G ?A (NODE ?LABEL ?SUBTREES))
    (?F ?LABEL (REDTREE ?F ?G ?A ?SUBTREES))
  (REDTREE ?F ?G ?A (PAIR ?SUBTREE ?RESTTREES))
    (?G (REDTREE ?F ?G ?A ?SUBTREE)
        (REDTREE ?F ?G ?A ?RESTTREES))
  (REDTREE ?F ?G ?A []) ?A)

;;; Another (more compact) way of defining the same function:
(DEFINE
  (REDTREE ?F ?G ?A (NODE ?LABEL ?SUBTREES))
    (?F ?LABEL (REDTREE ?F ?G ?A ?SUBTREES))
  (REDTREE ?F ?G ?A ?L)
    (RREDUCE (?G⊕(REDTREE ?F ?G ?A)) ?A ?L))

;;; SUMTREE sums all node values of the tree:
(DEFINE SUMTREE (REDTREE + + 0))

;;; an average tree:
(DEFINE ATREE (NODE 1 [(NODE 2 [])/,(NODE 3 [(NODE 4 [])])]))

;;; returns all labels of the tree in a list
;;; (INORDER traversal):
(DEFINE LABELS (REDTREE PAIR APPEND []))

;;; applies the function f to each label in the tree:
(DEFINE (MAPTREE ?F) (REDTREE (NODE⊕?F) PAIR []))
```



```
                    Expression Input/Output
NF of sum [1,...,100] is
5050

NF of product [1,...,100] is
93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156
518286253697920827223758251185210916864000000000000000000000000

NF of length [1,3,...,1000] is
500

NF of atree is
NODE 1 [NODE 2 [],NODE 3 [NODE 4 []]]

NF of sumtree atree is
10

NF of labels atree is
[1,2,3,4]

NF of maptree (* 34) atree is
NODE 34 [NODE 68 [],NODE 102 [NODE 136 []]]

NF of labels (maptree (* 34) atree) is
[34,68,102,136]

NF of sum (labels (maptree (* 34) atree)) is
340

NF of maptree (+ foo) atree is
NODE (+ FOO 1) [NODE (+ FOO 2) [],NODE (+ FOO 3) [NODE (+ FOO 4) []]]

NF of smooth (labels (maptree (* 34) atree)) is
[34,51.0,76.5,106.25]

NF of smooth (smooth (labels (maptree (* 34) atree))) is
[34,42.5,59.5,82.875]

NF of nth-pwr add1 5 1 is
6

NF of nth-pwr smooth 10 (labels (maptree (* 34) atree)) is
[34,34.033203,34.23242,34.800104]

NF of
```

```
Copyright (C) 1986 Syracuse University
Inf-plus
        Reduction Statistics

Reductions              : 387
Attempted Inferences: 0
Inferences              : 0
Attempted Unifications  : 0
Symbols Expanded        : 47
Elapsed Time            : 0.2351 secs
Reduction Rate          : 1646 RPS
Size of Result          : 31

Swaps                   : 0
Suspensions             : 0
Activations             : 9
Completions             : 9
Terminations            : 0
Max Concurrency         : 1
Avg Concurrency         : 1

Combinations Constructed: 613
Combinations Forwarded  : 179
IPs Followed            : 130
Number of Stacks        : 298
Stack Pushes            : 1505
Stack References        : 3095
Stack Checks            : 494
Stack Modifications     : 751
Maximum Active Stacks   : 24
Maximum Stack Depth     : 11
Maximum Active Cells    : 100

Functors Introduced: 0

 Steps    %Steps    Functor
 -----    ------    -------
    73     18.9     C^
    40     10.3     ACCUMULATE
    35      9.0     S^
    30      7.8     DIV
    30      7.8     +
    22      5.7     B
    16      4.1     ARC
    16      4.1     A-S-E
    16      4.1     W
    14      3.6     RREDUCE
    14      3.6     C
    11      2.8     PRIM-REC
```

This material may be reproduced by or for
the U.S. Government pursuant to the copyright
license under DAR clause 7-104.9(a) (1981 May).

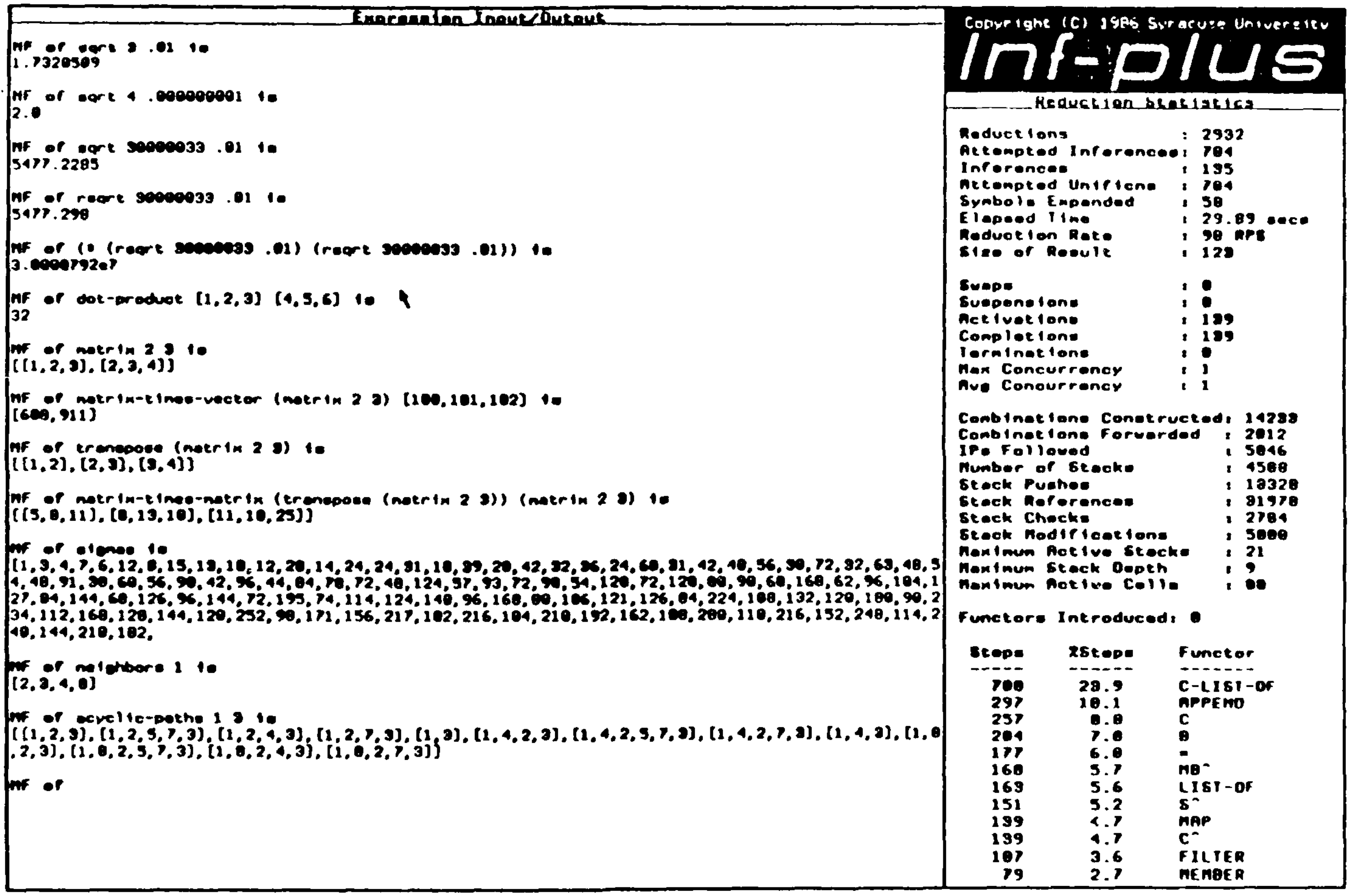
```
;;; Newton-Raphson square root finding via
;;; infinite lists of approximations.

;;; given a number N, which we are trying to find the
;;; square root of, and an an approximation X, the function
;;; NEXT, produces the next approximation.
(DEFINE (NEXT ?N ?X) (DIV (+ ?X (DIV ?N ?X)) 2))

;;; So, (ITERATE (NEXT num) guess) produces the infinite list
;;; of approximations of the square root of num.

;;; We will terminate this process when two successive
;;; approximations are within epsilon of each other.
(DEFINE (ABS ?X) (IF (< ?X 0) (MINUS ?X) ?X))
(DEFINE
 (WITHIN ?EPS ?AS)
 ((IF (< (ABS (- ?A ?B)) ?EPS)
     ?B
     (WITHIN ?EPS ?TLAS))
  WHERE*
   ?TLAS = (TL ?AS) /;
   ?A = (HD ?AS) /;
   ?B = (HD ?TLAS)))

(DEFINE (SQRT ?N ?EPS)
      (WITHIN ?EPS (ITERATE (NEXT ?N) (DIV ?N 2))))

(DEFINE
 (RELATIVE ?EPS ?AS)
 ((IF (< (ABS (SUB1 (DIV ?A ?B))) ?EPS)
     ?B
     (RELATIVE ?EPS ?TLAS))
  WHERE*
   ?TLAS = (TL ?AS) /;
   ?A = (HD ?AS) /;
   ?B = (HD ?TLAS)))

(DEFINE (RSQRT ?N ?EPS)
      (RELATIVE ?EPS (ITERATE (NEXT ?N) (DIV ?N 2))))
```

```
;;; If a matrix is represented as a list of its rows
;;; (which are lists of its elements), e.g.:
;;;
;;;    |  1  2  3 |
;;;    |  4  5  6 |
;;;    |  7  8  9 |
;;;    | 10 11 12 |
;;;
;;; represented as:
(DEFINE MAT [[1/,2/,3]/,[4/,5/,6]/,[7/,8/,9]/,[10/,11/,12]])

;;; or an N by M matrix by:
(DEFINE (MATRIX ?N ?M)
      [[?I/,../,(+ ?I (SUB1 ?M))] /| ?I ε [1/,../,?N]])

;;; then summing all elements of a matrix is accomplished
;;; by the function SUM-MAT:
(DEFINE SUM-MAT (SUM⊕(MAP SUM)))
;;; NB: Constructing and summing a 25 by 25 matrix is accomplished
;;;     in a bit over a second.

;;; RREDUCE-N f init [[a1,a1,a3],[b1,b2,b3]] ->
;;; [(RREDUCE f init [a1,b1]),
;;;  (RREDUCE f init [a2,b2])
;;;  (RREDUCE f init [a3,b3])]
;;; defined with PRIM-REC:
(DEFINE
  (RREDUCE-N ?F ?INIT)
  (PRIM-REC (NULLP⊕HD)
       (λ (?LISTS ?RES)
         [(RREDUCE ?F ?INIT (MAP HD ?LISTS))•?RES])
       (MAP TL)
       []))

(DEFINE (DOT-PRODUCT ?V ?W) (SUM (RREDUCE-N * 1 [?V/,?W])))

(DEFINE (MATRIX-TIMES-VECTOR ?M ?V) (MAP (DOT-PRODUCT ?V) ?M))

(DEFINE TRANSPOSE (RREDUCE-N PAIR []))

(DEFINE (MATRIX-TIMES-MATRIX ?M ?N)
    (MAP (MATRIX-TIMES-VECTOR (TRANSPOSE ?N)) ?M))
```

```
;;; Polya's Sigma Function:

;;; sigma n = sum of prime factors of n
;;; e.g. sigma 4 = 1 + 2 + 4 = 7
;;;      sigma 5 = 1 + 5 = 6 (sigma p = for all primes p (1 + p))
;;;
;;; sigma n = (sigma (n - 1)) + (sigma (n - 2)) -
;;;           (sigma (n - 5)) - (sigma (n - 7))
;;;          +               +
;;;          -               -         ...
;;; sigma (neg) does not contribute
;;; sigma (0) = n

;;; [1,2,3,4,...]
;;; [3,5,7,9,...]
;;; merged is [1  3  2  5  3  7  4  9  ...]
;;;         [1  2  5  7  12 15 22
;;;          diff-list 1 [n1•restns]
;;;          dl n [m•r] = [n+m•(dl n+m r)]

;;; sigma 6 = 1 + 2 + 3 + 6 = 12
;;; sigma 6 = sigma 5 + sigma 4 - sigma 1 = 6 + 7 - 1 = 12

;;; First n elements of a list
(DEFINE (FIRST ?N [?X•?R])
  (IF (ZEROP ?N) [] [?X•(FIRST (SUB1 ?N) ?R)]))

;;; TakeWh p list = longest initial segment of list s.t.
;;; (p el) = true for each element el in the segment
(DEFINE (TAKEWH ?P)
  (RREDUCE (λ (?X) (IF (?P ?X) (PAIR ?X) (K []))) []))

;;; Positive integers
(DEFINE POS-INTS [1/,2/,..])

;;; Positive odd integers starting at 3
(DEFINE ODDS [3/,5/,..])

(DEFINE (ACC ?N [?M•?R])
 ([?S•(ACC ?S ?R)] WHERE ?S = (+ ?N ?M)))
```

```
;;; The list of coefficients for the infinite sum:
;;; [1,2,5,7,12,15,22,...]
(DEFINE FUNNY-NUMS [1•(ACC 1 (INTERLEAVE POS-INTS ODDS))])

;;; Sigma WITHOUT memoizing
(DEFINE (SIGMA ?N)
    (PPMM
      (MAP (λ (?U) (IF (ZEROP ?U) ?N (SIGMA ?U)))
          (TAKEWH (λ (?Z) (NOT (> 0 ?Z)))
              (MAP (λ (?X) (- ?N ?X))
                FUNNY-NUMS)))))

;;; Sigma with memoizing
(DEFINE (MEMO-SIGMA ?N)
    (PPMM
      (MAP (λ (?U) (IF (ZEROP ?U) ?N (NTH ?U SIGMAS)))
          (TAKEWH (λ (?Z) (NOT (> 0 ?Z)))
              (MAP (λ (?X) (- ?N ?X))
                FUNNY-NUMS)))))

;;; [(sigma 1),(sigma 2),(sigma 3),...]
(DEFINE SIGMAS (MAP MEMO-SIGMA [1/,..]))

(DEFINE
  (PMMP []) 0
  (PMMP [?X•?REST]) (+ (MMPP ?REST) ?X))
(DEFINE
  (MMPP []) 0
  (MMPP [?X•?REST]) (- (MPPM ?REST) ?X))
(DEFINE
  (MPPM []) 0
  (MPPM [?X•?REST]) (- (PPMM ?REST) ?X))
(DEFINE
  (PPMM []) 0
  (PPMM [?X•?REST]) (+ (PMMP ?REST) ?X))

(DEFINE (DIVISORS ?X)
  [1•[?D /| ?D ε [2/,../,?X] /; (ZEROP (REM ?X ?D))]])
(DEFINE (ALTERNATE-SIGMA ?X) (LREDUCE + 0 (DIVISORS ?X)))
(DEFINE ALTERNATE-SIGMAS (MAP ALTERNATE-SIGMA [1/,..]))
(DEFINE AS-FOR-SHOW (MAP AS ALTERNATE-SIGMAS))
```

```
;;; Assuming there is a logic program defining an undirected
;;; graph via a collection of clauses of the form (arc ?x ?y)
;;; stating that there is an arc between nodes ?x and ?y, then
;;; following program, when provided with starting and ending
;;; nodes (S and E), produces a list of all acyclic paths from
;;; S to E in the graph. A path from S to E, having
;;; intermediate nodes N1,...,Nk will be represented by the list
;;; [S,N1,...,Nk,E].

(DEFINE (ACYCLIC-PATHS ?S ?E) (PATHS-EXCLUDING ?S ?E []))

(DEFINE
  (PATHS-EXCLUDING ?S ?E ?NS)
  (IF (= ?S ?E)
     [[?E]]
     (MAP (PAIR ?S)
       (FLATTEN
        [(PATHS-EXCLUDING ?N ?E [?S•?NS]) /|
         ?N ε (NEIGHBORS ?S) /;
         (NOT (MEMBER ?N ?NS))]))))

(DEFINE FLATTEN (RREDUCE APPEND []))

(DEFINE (NEIGHBORS ?S) [?N /| (ARC ?S ?N)])

(DEFINE
  ((ARC ?X ?Y) ← (DARC ?X ?Y))
  ((ARC ?X ?Y) ← (DARC ?Y ?X)))

(DEFINE
  ((DARC 1 2))
  ((DARC 1 3))
  ((DARC 1 4))
  ((DARC 2 3))
  ((DARC 2 5))
  ((DARC 3 4))
  ((DARC 4 2))
  ((DARC 5 6))
  ((DARC 5 7))
  ((DARC 7 2))
  ((DARC 7 3))
  ((DARC 1 8))
  ((DARC 8 2)))
```



```
Expression Input/Output

NF of sqrt 3 .01 is
1.7320509

NF of sqrt 4 .000000001 is
2.0

NF of sqrt 30000033 .01 is
5477.2285

NF of rsqrt 30000033 .01 is
5477.298

NF of (* (rsqrt 30000033 .01) (rsqrt 30000033 .01)) is
3.0000792e7

NF of dot-product [1,2,3] [4,5,6] is
32

NF of matrix 2 3 is
[[1,2,3],[2,3,4]]

NF of matrix-times-vector (matrix 2 3) [100,101,102] is
[608,911]

NF of transpose (matrix 2 3) is
[[1,2],[2,3],[3,4]]

NF of matrix-times-matrix (transpose (matrix 2 3)) (matrix 2 3) is
[[5,8,11],[8,13,18],[11,18,25]]

NF of sigmas is
[1,3,4,7,6,12,8,15,13,18,12,28,14,24,24,31,18,39,20,42,32,36,24,60,31,42,40,56,30,72,32,63,48,5
4,48,91,38,60,56,90,42,96,44,84,78,72,48,124,57,93,72,98,54,120,72,120,80,90,60,168,62,96,104,1
27,84,144,68,126,96,144,72,195,74,114,124,140,96,168,80,186,121,126,84,224,108,132,120,180,90,2
34,112,168,128,144,120,252,98,171,156,217,102,216,104,210,192,162,108,280,110,216,152,248,114,2
40,144,210,182,

NF of neighbors 1 is
[2,3,4,8]

NF of acyclic-paths 1 3 is
[[1,2,3],[1,2,5,7,3],[1,2,4,3],[1,2,7,3],[1,3],[1,4,2,3],[1,4,2,5,7,3],[1,4,2,7,3],[1,4,3],[1,8
,2,3],[1,8,2,5,7,3],[1,8,2,4,3],[1,8,2,7,3]]

NF of
```

```
Copyright (C) 1986 Syracuse University
Inf-plus
Reduction Statistics

Reductions              : 2932
Attempted Inferences: 784
Inferences              : 135
Attempted Unifions      : 784
Symbols Expanded        : 58
Elapsed Time            : 29.89 secs
Reduction Rate          : 98 RPS
Size of Result          : 129

Swaps                   : 0
Suspensions             : 0
Activations             : 139
Completions             : 139
Terminations            : 0
Max Concurrency         : 1
Avg Concurrency         : 1

Combinations Constructed: 14233
Combinations Forwarded  : 2812
IPs Followed            : 5846
Number of Stacks        : 4588
Stack Pushes            : 18328
Stack References        : 81978
Stack Checks            : 2784
Stack Modifications     : 5000
Maximum Active Stacks   : 21
Maximum Stack Depth     : 9
Maximum Active Cells    : 88

Functors Introduced: 0

 Steps     %Steps     Functor
 -----     ------     -------
   700      23.9      C-LIST-OF
   297      10.1      APPEND
   257       8.8      C
   204       7.0      B
   177       6.0      =
   168       5.7      MB^
   163       5.6      LIST-OF
   151       5.2      S^
   139       4.7      MAP
   139       4.7      C^
   107       3.6      FILTER
    79       2.7      MEMBER
```

```
;;; An implementation of Peter Henderson's "Functional Geometry"
;;; in LNF. PH's paper was presented at the 1982 Lisp Symposium.
;;; This implementation makes use of the fact that all
;;; LNF functions are "Schonfinkeled".

;; A PLOTTABLE-PICTURE is simply a list of PLOTTABLE-LINEs
;; where a PLOTTABLE-LINE is a construction of the form
;;         LINE (VEC x0 y0) (VEC x1 y1).
;; LINE and VEC are constructors.

;; A PICTURE is a function, which when applied to
;; three arguments (each a vector of the form (VEC x y)), is
;; a PLOTTABLE-PICTURE.

;; TWO HELPER FUNCTIONS:

;; vector-vector addition
(define (vec+vec (vec ?x0 ?y0) (vec ?x1 ?y1))
  (vec (+ ?x0 ?x1) (+ ?y0 ?y1)))

;; scalar-vector multiplication
(define (scalar*vec ?n (vec ?x ?y))
  (vec (* ?n ?x) (* ?n ?y)))

;; THE BASIC FUNCTIONS:

;; Implements PH's nil (the empty picture), i.e. a function
;; of arity 3 which, when applied, ignores its arguments and
;; returns the empty list.
(define (empty-pic ? ? ?) [])

;; Implements PH's: plot(grid(m,n,s),a-vec,b-vec,c-vec)
;; (grid m n segs) -> picture
;; (grid m n segs avec bvec cvec) -> plottable-picture
;; NOTE: plot is unnecessary in this implementation.
(define (grid ?m ?n ?segments ?a-vec ?b-vec ?c-vec)
  (for-each (segment ?x0 ?y0 ?x1 ?y1) in ?segments
   instantiate
   (line (vec+vec ?a-vec
           (vec+vec (scalar*vec (div ?x0 ?m) ?b-vec)
                (scalar*vec (div ?y0 ?n) ?c-vec)))
      (vec+vec ?a-vec
           (vec+vec (scalar*vec (div ?x1 ?m) ?b-vec)
                (scalar*vec (div ?y1 ?n) ?c-vec))))))
```

```
;; FRACTALS

(define (fractalize ?n ?fractal-fn ?pic ?a-vec ?b-vec ?c-vec)
  ((if (zerop ?n)
      ?plottable-picture
      (fractalize1 (sub1 ?n)
                ?fractal-fn
                (flatmap ?fractal-fn ?plottable-picture)))
  where ?plottable-picture = (?pic ?a-vec ?b-vec ?c-vec)))

(define (fractalize1 ?n ?fractal-fn ?plottable-pic)
  (if (zerop ?n)
     ?plottable-pic
     (fractalize1 (sub1 ?n)
               ?fractal-fn
               (flatmap ?fractal-fn ?plottable-pic))))

(define (make-lines [?v1/,?v2•?vecs])
  [(line ?v1 ?v2)•
   (if (nullp ?vecs)
      []
      (make-lines [?v2•?vecs]))])

;; a not so terrible fractal function
(define (fractal-fn-1 (line (vec ?x0 ?y0) (vec ?x1 ?y1)))
  ((make-lines
    [(vec ?x0 ?y0)/,
     (vec (+ ?x0 (* 1\3 ?sum))
         (- (- ?y1 (* 1\3 ?length)) (* 2\3 ?height)))/,
     (vec (+ (+ ?x0 (* 1\3 ?height))
         (* 2\3 ?length)) (- ?y1 (* 1\3 ?sum)))/,
     (vec ?x1 ?y1)])
  where* ?length = (- ?x1 ?x0) /;
        ?height = (- ?y1 ?y0) /;
       ?sum = (+ ?length ?height)))

(define man-and-wife
  (beside 1 1 man (fractalize 3 fractal-fn-1 man)
        (vec 100 100) (vec 500 0) (vec 0 500)))
```

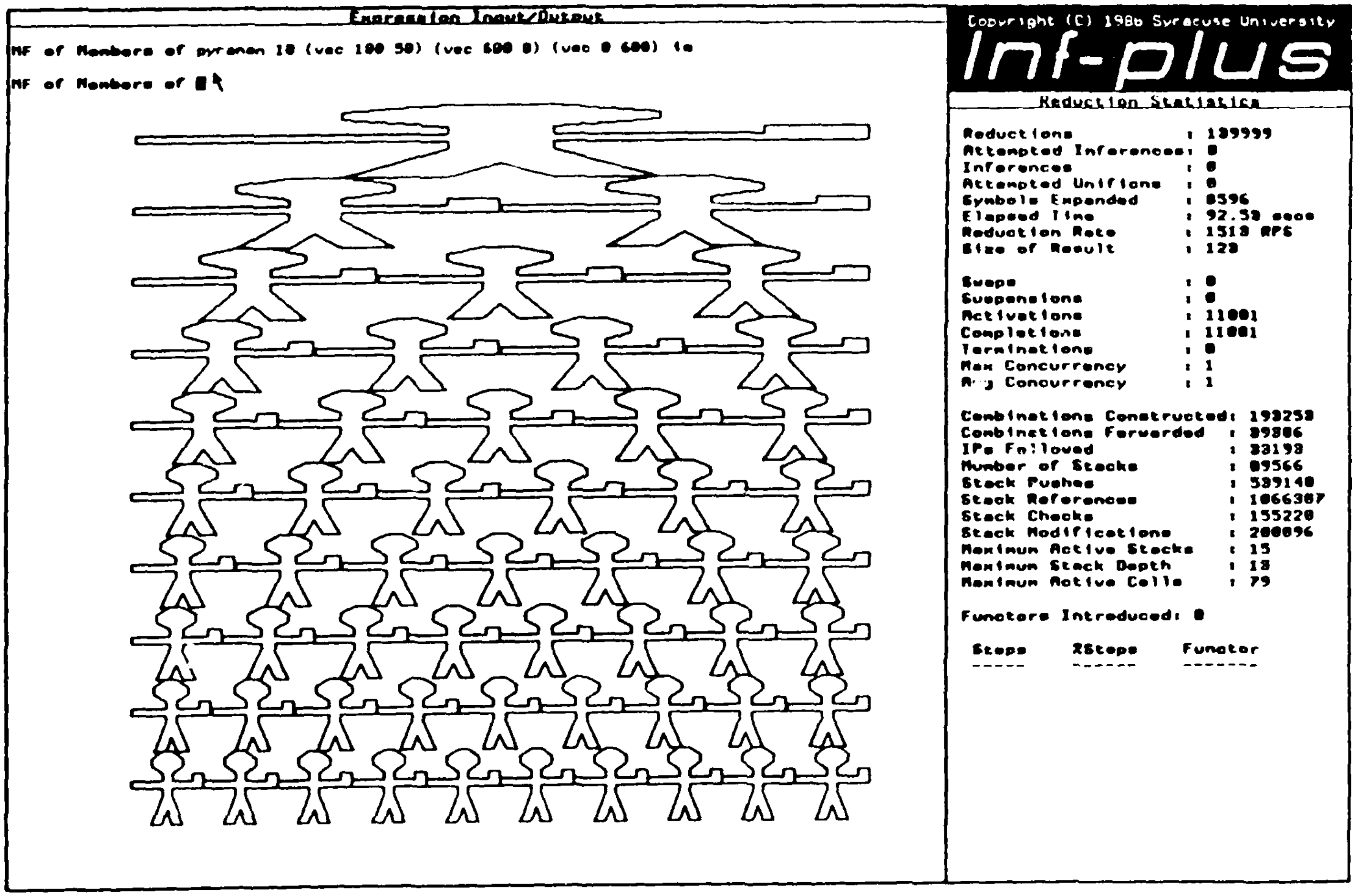
```
(define (pyraman ?n)
    (if (= 1 ?n)
      man
      (above (sub1 ?n) 1 (pyraman (sub1 ?n)) (men-in-a-row ?n))))

(define (men-in-a-row ?n)
    (if (= 1 ?n)
      man
      (beside 1 (sub1 ?n) man (men-in-a-row (sub1 ?n)))))

(define (men-on-men ?n) (men1 ?n (add1 ?n)))

(define (men1 ?n ?m)
    (if (zerop ?n)
      empty-pic
      (beside 1
          ?m
          (above (- ?m ?n) ?n man man)
          (men1 (sub1 ?n) ?m))))

(define poodle
  (fractalize 4 fractal-fn-1 triangle
          (vec 100 100) (vec 500 0) (vec 0 500)))

(define replicated-pod
  (fractalize 4 fractal-fn-1 (square 100 99)
          (vec 100 100) (vec 500 0) (vec 0 500)))

(define four-men (fractal-quartet man fractal-fn-1))

(define (fractal-quartet ?pic ?fn)
  ((quartet ?pic ?f-1 ?f-2 (fractalize 1 ?fn ?f-2))
  where* ?f-1 = (fractalize 1 ?fn ?pic) /;
       ?f-2 = (fractalize 1 ?fn ?f-1)))
```

```
;; Some examples from PH's paper:
;; PH's man
(define man
    (grid 14 20
    [segment 6 10 0.05 10/,
     segment 0.05 10 0.05 12/,
     segment 0.05 12 6 12/,
     segment 6 12 6 14/,
     segment 6 14 4 16/,
     segment 4 16 4 18/,
     segment 4 18 6 19.95/,
     segment 6 19.95 8 19.95/,
     segment 8 19.95 10 18/,
     segment 10 18 10 16/,
     segment 10 16 8 14/,
     segment 8 14 8 12/,
     segment 8 12 12 12/,
     segment 12 12 12 14/,
     segment 12 14 13.95 14/,
     segment 13.95 14 13.95 10/,
     segment 13.95 10 8 10/,
     segment 8 10 8 8/,
     segment 8 8 10 0.05/,
     segment 10 0.05 8 0.05/,
     segment 8 0.05 7 4/,
     segment 7 4 6 0.05/,
     segment 6 0.05 4 0.05/,
     segment 4 0.05 6 8/,
     segment 6 8 6 10])))

(define fatboy (above 1 1 empty-pic man))

(define boy (beside 1 1 fatboy empty-pic))

(define (rectangle ?grid-size ?x ?y )
  (grid ?grid-size ?grid-size
    [segment 0 0 0 ?y/,
     segment 0 ?y ?x ?y/,
     segment ?x ?y ?x 0/,
     segment ?x 0 0 0])))

(define (square ?grid-size ?x)
  (rectangle ?grid-size ?x ?x))
```

```
;; Escher drawing components and functions:

;; PH's p, figure 18
(define mce-p
 (grid 36 36
   [;; left eye
   segment 0 7 6 9/,    segment 6 9 0 18/,   segment 0 18 0 7/,
   ;; line between eyes
   segment 13 0 9 9/,
   ;; right eye
   segment 9 12 9 23/,  segment 9 23 16 14/, segment 16 14 9 12/,
   ;; side of head
   segment 24 0 22 9/,  segment 22 9 18 18/,
    segment 18 18 9 30/, segment 9 30 0 36/,
   ;; top of tail
   segment 0 36 13 34/, segment 13 34 18 36/,
    segment 18 36 26 27/,segment 26 27 36 27/,
   ;; line in tail
   segment 18 27 36 23/,
   ;; bottom of tail
   segment 18 18 27 21/,segment 27 21 36 18/,
   ;; tiny line in upper right
   segment 32 36 36 34/,
   ;; next one down
   segment 27 36 29 34/,segment 29 34 36 32/,
   ;; and the next
   segment 22 36 26 32/,segment 26 32 36 29/,
   ;; first line below tail
   segment 20 14 27 16/,segment 27 16 36 14/,
   ;; the next
   segment 22 9 29 11/, segment 29 11 36 9/,
   ;; and, finally, the last
   segment 24 0 31 5/,  segment 31 5 36 5]))

;; PH's q, figure 19
(define mce-q
 (grid 36 36
   [;; left side of fish
   segment 0 27 7 29/, segment 7 29 11 31/,
    segment 11 31 16 34/, segment 16 34 18 36/,
   ;; line in middle of fish
   segment 0 23 16 25/,
    ;; left edge
    segment 0 27 0 36/, segment 0 0 0 18/,
    ;; right side of fish
    segment 0 18 9 16/, segment 9 16 13 16/,
    segment 13 16 27 22/, segment 27 22 36 36/,
    ;; leftmost line above fish
    segment 4 36 7 29/,
    ;; next one
    segment 9 36 11 31/,
    ;; rightmost line above fish
    segment 14 36 16 34/,
    ;; left eye
    segment 18 34 25 34/,segment 25 34 20 30/, segment 20 30 18 34/,
    ;; right eye
    segment 20 27 27 27/, segment 27 27 22 23/, segment 22 23 20 27/,
    ;; right side of tail
    segment 36 36 34 22/, segment 34 22 36 18/,
    segment 36 18 29 9/, segment 29 9 27 0/,
    ;; three lines to the right of the tail
    segment 29 0 36 14/, segment 32 0 36 9/, segment 34 0 36 4/,
    ;; line in tail
    segment 32 25 23 0/,
    ;; four lines left of tail (left to right)
    segment 5 0 9 11/, segment 9 11 9 16/,
    segment 9 0 13 11/, segment 13 11 13 16/,
    segment 14 0 18 13/, segment 18 13 18 18/,
    segment 18 0 22 14/, segment 22 14 22 20]))

;; PH's quartet
;; (quartet picture picture picture picture) -> picture
(define (quartet ?p1 ?p2 ?p3 ?p4)
  (above 1 1 (beside 1 1 ?p1 ?p2) (beside 1 1 ?p3 ?p4)))

;; PH's cycle
;; (cycle picture) -> picture
(define (cycle ?pic)
  ((quartet ?pic
       (rot ?rot-rot-pic)
       ?rot-pic
       ?rot-rot-pic)
   where* ?rot-pic = (rot ?pic) /;
        ?rot-rot-pic = (rot ?rot-pic)))
```

```
;; PH's r, figure 20
(define mce-r
  (grid 36 36
    [;; top of fish
     segment 24 36 27 28/, segment 27 28 36 18/,
     ;; bottom of fish
     segment 0 36 4 27/, segment 4 27 10 22/, segment 10 22 17 18/,
     segment 17 18 31 14/, segment 31 14 36 9/,
     ;; line thru fish
     segment 13 36 25 23/, segment 25 23 36 14/,
     ;; lines above fish
     segment 27 28 36 36/, segment 29 30 36 23/,
      segment 31 32 36 28/,segment 33 34 36 32/,
     ;; bottom semi-horizontal lines
     segment 2 2 8 0/, segment 4 4 18 0/, segment 7 7 18 4/,
     segment 18 4 27 0/, segment 10 11 27 7/, segment 27 7 36 0/,
     ;; lower diagonal lines
     segment 0 0 17 18/, segment 0 8 10 22/,
      segment 0 18 4 27/, segment 0 27 2 32]))

;; PH's t, figure 22
(define mce-t
  (quartet mce-p mce-q mce-r mce-s))

;; PH's u, figure 23
(define mce-u
  (cycle (rot mce-q)))

;; PH's s, figure 21
(define mce-s
  (grid 36 36
    [;; left fish
     segment 18 36 16 30/, segment 16 30 16 23/, segment 16 23 16 18/,
     segment 16 18 18 14/, segment 18 14 23 9/, segment 23 9 36 0/,
     ;; line in fish
     segment 23 36 25 23/,
     ;; right fish
     segment 27 36 30 30/, segment 30 30 32 25/,
     segment 32 25 34 21/, segment 34 21 36 18/,
     ;; right eye
     segment 29 16 34 18/, segment 34 18 34 11/, segment 34 11 29 16/,
     ;; left eye
```

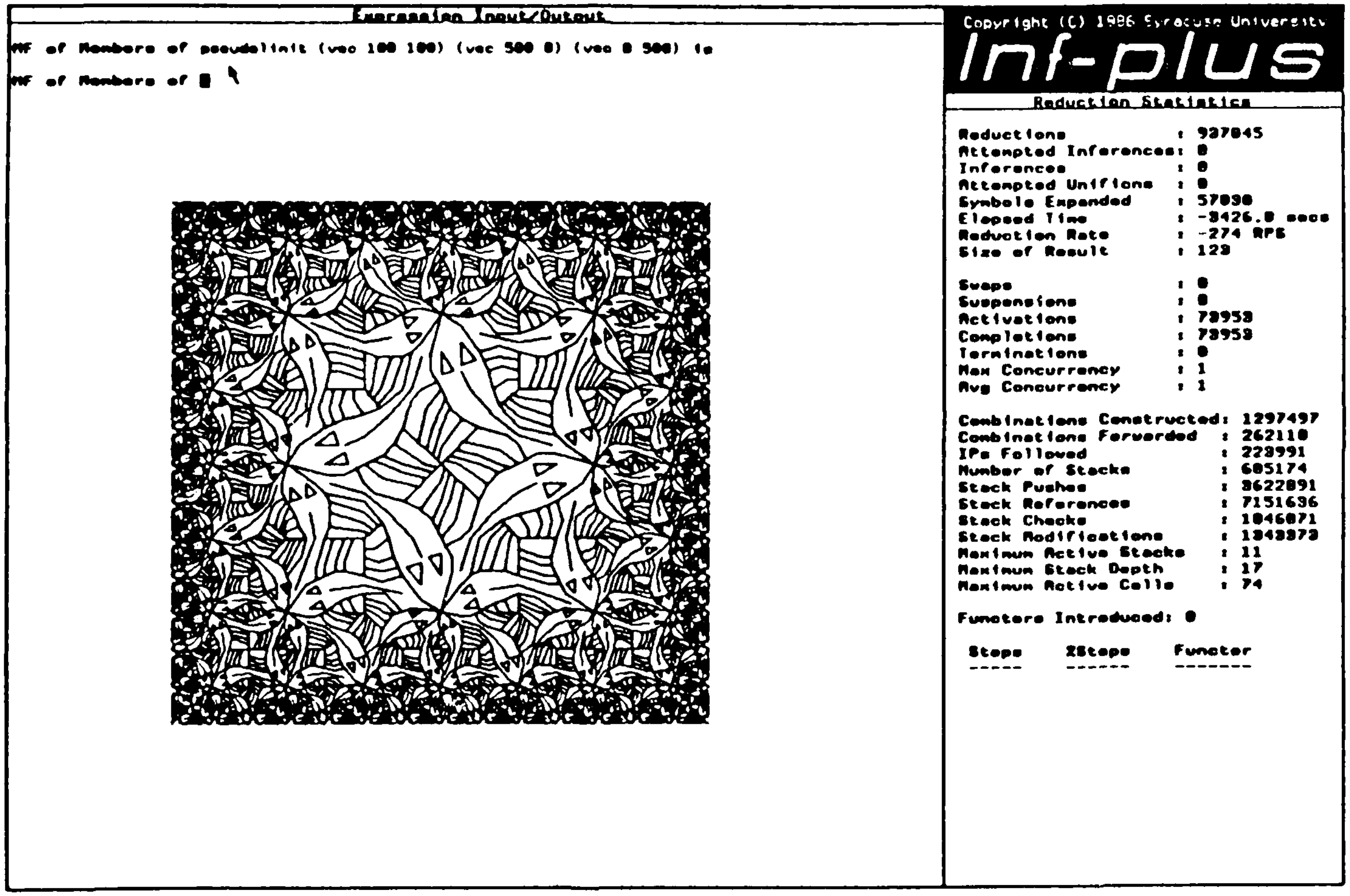
```
      segment 22 14 27 16/, segment 27 16 27 9/, segment 27 9 22 14/,
      ;; lines right of fish
      segment 30 30 36 32/, segment 32 25 36 27/, segment 34 21 36 22/,
      ;; bottom hump
      segment 0 0 9 5/, segment 9 5 17 5/, segment 17 5 36 0/,
      ;; next up
      segment 0 9 4 2/, segment 0 14 16 9/,
       segment 0 18 18 14/, segment 0 23 16 18/,
      segment 0 28 16 23/, segment 0 32 16 30/,
      ;; top border lines
      segment 0 36 18 36/, segment 27 36 36 36]))

;; AND THE REST:
(define side1 (quartet empty-pic empty-pic (rot mce-t) mce-t))

(define side2 (quartet side1 side1 (rot mce-t) mce-t))

(define corner1 (quartet empty-pic empty-pic empty-pic mce-u))

(define corner2 (quartet corner1 side1 (rot side1) mce-u))

(define pseudocorner
  (quartet corner2 side2 (rot side2) (rot mce-t)))

(define pseudolimit (cycle pseudocorner))

(define (nonet ?p1 ?p2 ?p3 ?p4 ?p5 ?p6 ?p7 ?p8 ?p9)
  (above 1 2
     (beside 1 2 ?p1 (beside 1 1 ?p2 ?p3))
     (above 1 1
        (beside 1 2 ?p4 (beside 1 1 ?p5 ?p6))
        (beside 1 2 ?p7 (beside 1 1 ?p8 ?p9)))))

(define corner
  ((nonet
    corner2     side2      side2
    ?rot-side2  mce-u      ?rot-mce-t
    ?rot-side2  ?rot-mce-t (rot mce-q))
  where ?rot-side2 = (rot side2) /&
       ?rot-mce-t = (rot mce-t)))

(define squarelimit (cycle corner))
```



```
Expression Input/Output

NF of Members of pyramen 10 (vec 100 50) (vec 600 0) (vec 0 600) is

NF of Members of █
```

```
Reduction Statistics

Reductions              : 139999
Attempted Inferences: 0
Inferences              : 0
Attempted Unifions      : 0
Symbols Expanded        : 8596
Elapsed Time            : 92.38 secs
Reduction Rate          : 1518 RPS
Size of Result          : 128

Swaps                   : 0
Suspensions             : 0
Activations             : 11001
Completions             : 11001
Terminations            : 0
Max Concurrency         : 1
Avg Concurrency         : 1

Combinations Constructed: 193258
Combinations Forwarded  : 39806
IPs Followed            : 33198
Number of Stacks        : 89566
Stack Pushes            : 539140
Stack References        : 1066387
Stack Checks            : 155220
Stack Modifications     : 200896
Maximum Active Stacks   : 15
Maximum Stack Depth     : 18
Maximum Active Cells    : 79

Functors Introduced: 0

 Steps    %Steps    Functor
 -----    ------    -------
```

This material may be reproduced by or for
the U.S. Government pursuant to the copyright
license under DAR clause 7-104.9(a) (1981 May).



Expression Input/Output

```
WF of Members of pseudolimit (vec 100 100) (vec 500 0) (vec 0 500) is
WF of Members of █
```

Copyright (C) 1986 Syracuse University

Inf-plus

Reduction Statistics

```
Reductions              : 937045
Attempted Inferences: 0
Inferences              : 0
Attempted Unifions      : 0
Symbols Expanded        : 57030
Elapsed Time            : -3426.0 secs
Reduction Rate          : -274 RPS
Size of Result          : 129

Swaps                   : 0
Suspensions             : 0
Activations             : 73953
Completions             : 73953
Terminations            : 0
Max Concurrency         : 1
Avg Concurrency         : 1

Combinations Constructed: 1297497
Combinations Forwarded  : 262110
IPs Followed            : 228991
Number of Stacks        : 605174
Stack Pushes            : 3622891
Stack References        : 7151636
Stack Checks            : 1046071
Stack Modifications     : 1343373
Maximum Active Stacks   : 11
Maximum Stack Depth     : 17
Maximum Active Cells    : 74

Functors Introduced: 0

 Steps    %Steps    Functor
 -----    ------    -------
```

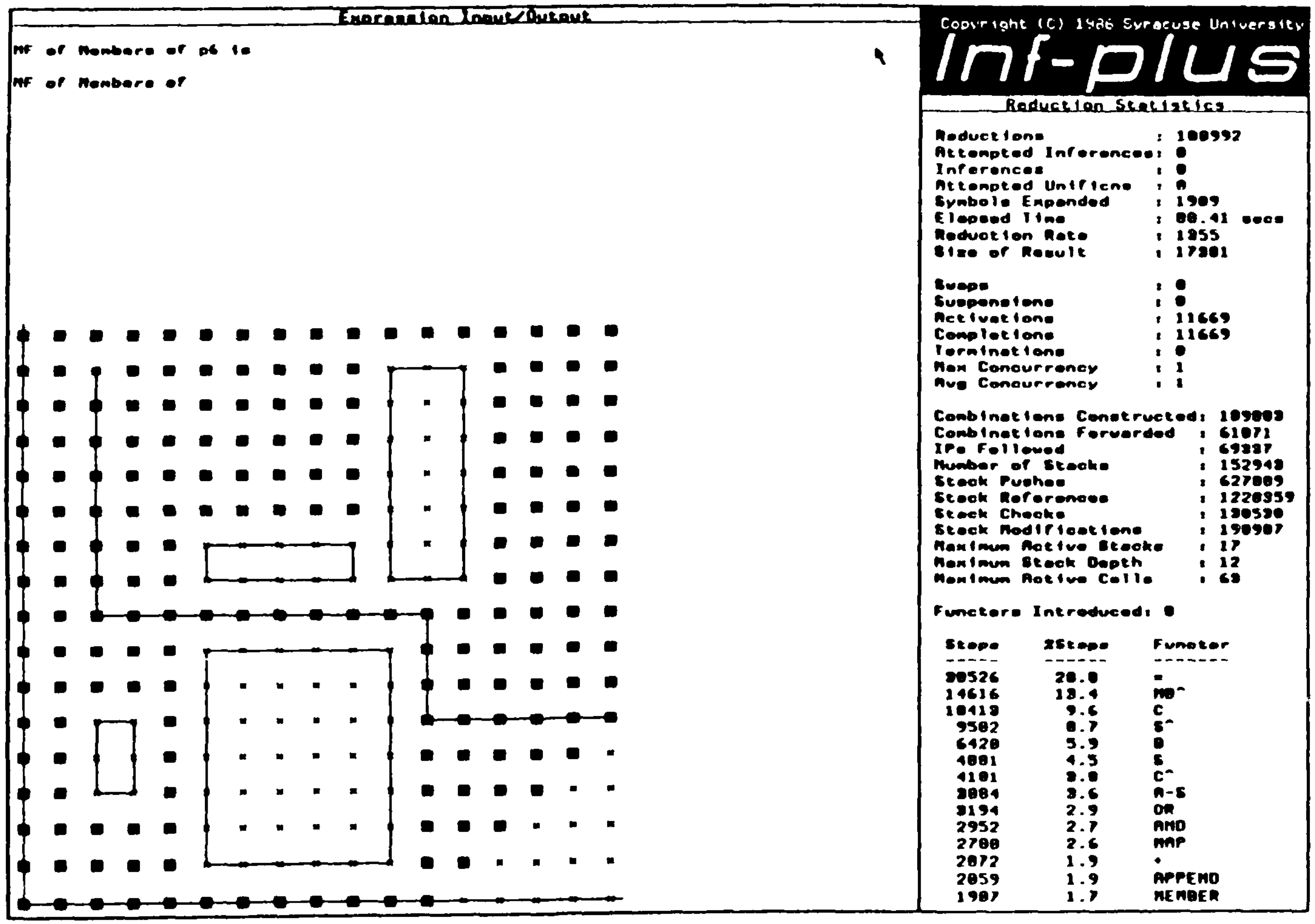
This material may be reproduced by or for
the U.S. Government pursuant to the copyright
license under DAR clause 7-104.9(a) (1981 May).

```
;;; Lee routing:

(DEFINE (LEE-PIC ?S ?D ?OBS)
    (APPEND LEE-GRID
     (APPEND (SQ (ADJ-VEC ?S) 6)
       (APPEND (SQ (ADJ-VEC ?D) 6)
         (APPEND (OBSTACLES (ADJ-OBS ?OBS))
           (LEE-MAP (LEE-WAVES-AND-ROUTE ?S ?D ?O3S)))))))

(DEFINE (ADJ-VEC (VEC ?X ?Y))
    (VEC (+ 10 (* 30 ?X)) (+ 10 (* 30 ?Y))))
(DEFINE ADJ-OBS (MAP ADJ-OB))
(DEFINE (ADJ-OB (OB ?V1 ?V2)) (OB (ADJ-VEC ?V1) (ADJ-VEC ?V2)))

;;; Some examples (P6 pictured after code)
(DEFINE P5
    (LEE-PIC (VEC 4 11) (VEC 16 5)
         [(OB (VEC 6 8) (VEC 9 10))/,
          (OB (VEC 2 3) (VEC 3 5))/,
          (OB (VEC 5 1) (VEC 10 7))/,
          (OB (VEC 10 9) (VEC 12 15))]))

(DEFINE P6
    (LEE-PIC (VEC 2 15) (VEC 16 5)
         [(OB (VEC 5 9) (VEC 9 10))/,
          (OB (VEC 2 3) (VEC 3 5))/,
          (OB (VEC 5 1) (VEC 10 7))/,
          (OB (VEC 10 9) (VEC 12 15))]))

(DEFINE P7
    (LEE-PIC (VEC 2 15) (VEC 3 15)
         [(OB (VEC 5 9) (VEC 9 10))/,
          (OB (VEC 2 3) (VEC 3 5))/,
          (OB (VEC 5 1) (VEC 1C 7))/,
          (OB (VEC 10 9) (VEC 12 15))]))

(DEFINE MAX 500)

(DEFINE LEE-GRID
    [(LINE (VEC 10 10) (VEC MAX 10))/,
     (LINE (VEC 10 10) (VEC 10 MAX))•
     (LEE-POINTS 10)])
```

```
(DEFINE (LEE-POINTS ?X)
    (IF (> ?X MAX)
       []
       (APPEND (LEE-LINE-OF-POINTS ?X)
           (LEE-POINTS (+ 30 ?X)))))

(DEFINE (LEE-LINE-OF-POINTS ?X)
    (LLPTS 10 ?X))

(DEFINE (LLPTS ?Y ?X)
    (IF (> ?Y MAX)
       []
       (APPEND (CROSS (VEC ?X ?Y) 4)
           (LLPTS (+ 30 ?Y) ?X))))

(DEFINE (SQ (VEC ?X ?Y) ?L)
    ([(LINE (VEC (- ?X ?L2) (- ?Y ?L2)) (VEC (- ?X ?L2) (+ ?Y ?L2)))/,
     (LINE (VEC (- ?X ?L2) (+ ?Y ?L2)) (VEC (+ ?X ?L2) (+ ?Y ?L2)))/,
     (LINE (VEC (+ ?X ?L2) (+ ?Y ?L2)) (VEC (+ ?X ?L2) (- ?Y ?L2)))/,
     (LINE (VEC (+ ?X ?L2) (- ?Y ?L2)) (VEC (- ?X ?L2) (- ?Y ?L2)))]
    WHERE ?L2 = IDIV ?L 2))

(DEFINE (CROSS (VEC ?X ?Y) ?L)
    ([(LINE (VEC (- ?X ?L2) (- ?Y ?L2)) (VEC (+ ?X ?L2) (+ ?Y ?L2)))/,
     (LINE (VEC (- ?X ?L2) (+ ?Y ?L2)) (VEC (+ ?X ?L2) (- ?Y ?L2)))]
    WHERE ?L2 = IDIV ?L 2))

(DEFINE OBSTACLES (RREDUCE (B APPEND RECTANGLE) []))

(DEFINE (RECTANGLE (OB (VEC ?XLL ?YLL) (VEC ?XUR ?YUR)))
    [(LINE (VEC ?XLL ?YLL) (VEC ?XUR ?YLL))/,
     (LINE (VEC ?XUR ?YLL) (VEC ?XUR ?YUR))/,
     (LINE (VEC ?XUR ?YUR) (VEC ?XLL ?YUR))/,
     (LINE (VEC ?XLL ?YUR) (VEC ?XLL ?YLL))])

(DEFINE (LEE-MAP [?WAVES•?ROUTE])
    (APPEND (MAP LEE-WAVE ?WAVES)
        (LEE-PATH ?ROUTE)))
```

```
(DEFINE (LEE-PATH ?R)
    (IF (NULLP (TL ?R))
       []
       [(LEE-SEG (HD ?R) (HD (TL ?R)))•
        (LEE-PATH (TL ?R))]))

(DEFINE (LEE-SEG (VEC ?X ?Y) (VEC ?Z ?W))
    (LINE (ADJ-VEC (VEC ?X ?Y)) (ADJ-VEC (VEC ?Z ?W))))

(DEFINE LEE-WAVE (MAP (λ (?V) (SQ (ADJ-VEC ?V) 8))))

(DEFINE REV (LREDUCE (C PAIR) []))

(DEFINE (LEE-WAVES-AND-ROUTE ?S ?D ?OBS)
    ([?WS•?P] WHERE*
     ?WS = (WAVES-LEADING-TO ?D [[?S]/,[]] ?OBS) /;
     ?P = (PATH-LEADING-TO ?D (TL (APPEND (REV ?WS) [[?S]/,[]])))))

(DEFINE (WAVES-LEADING-TO ?D ?WS ?OBS)
    (IF (MEMBER ?D (HD ?WS))
       []
       ([?NW•(WAVES-LEADING-TO ?D [?NW•?WS] ?OBS)]
       WHERE ?NW = (NEXT-WAVE ?WS ?OBS))))

(DEFINE (NEXT-WAVE [?W1/,?W2•?] ?OBS)
    [?N /| ?N ε (MKSET (FLATMAP NEIGHBORS ?W1)) /;
         (AND (NOT (MEMBER ?N ?W1))
          (AND (NOT (MEMBER ?N ?W2))
           (NOT (OBSTRUCTED-BY-ANY ?N ?OBS))))])

(DEFINE (OBSTRUCTED-BY-ANY ?N ?OBS)
  (MEMBER TRUE (MAP (OBSTRUCTED ?N) ?OBS)))

(DEFINE (≤ ?X ?Y ?Z)
  (AND (OR (< ?X ?Y) (= ?X ?Y)) (OR (< ?Y ?Z) (= ?Y ?Z))))

(DEFINE (OBSTRUCTED (VEC ?X ?Y) (OB (VEC ?XL ?YL) (VEC ?XU ?YU)))
    (OR (AND (OR (= ?X ?XL)
            (= ?X ?XU))
         (≤ ?YL ?Y ?YU))
       (AND (OR (= ?Y ?YL)
            (= ?Y ?YU))
         (≤ ?XL ?X ?XU))))
```

```
(DEFINE (PATH-LEADING-TO ?D ?WS)
    [?D•(IF (NULLP (TL ?WS))
        []
        (PATH-LEADING-TO
         (HD [?NBR /| ?NBR ε NEIGHBORS ?D /;
                  (MEMBER ?NBR (HD ?WS))])
                 (TL ?WS)))])

(DEFINE (NEIGHBORS (VEC ?X ?Y))
    (APPEND [(VEC ?N ?Y) /| ?N ε (NEXT-TO ?X)]
        [(VEC ?X ?N) /| ?N ε (NEXT-TO ?Y)]))

(DEFINE (NEXT-TO ?N)
    (APPEND (IF (< ?N 16) [(ADD1 ?N)] [])
        (IF (< 0 ?N) [(SUB1 ?N)] [])))

(DEFINE SAMPLE-WAVES
    (MAP (λ (?N) [(VEC ?N 15)/,(VEC ?N 14)/,(VEC ?N 13)])
        [1/,../,10]))

(DEFINE SAMPLE-ROUTE [(VEC 15 15)/,(VEC 15 14)/,(VEC 15 13)])
```



```
NF of Members of p6 is

NF of Members of
```

Copyright (C) 1986 Syracuse University

Inf-plus

Reduction Statistics

```
Reductions              : 108992
Attempted Inferences: 0
Inferences              : 0
Attempted Unificns      : 0
Symbols Expanded        : 1909
Elapsed Time            : 80.41 secs
Reduction Rate          : 1355
Size of Result          : 17381

Swaps                   : 0
Suspensions             : 0
Activations             : 11669
Completions             : 11669
Terminations            : 0
Max Concurrency         : 1
Avg Concurrency         : 1

Combinations Constructed: 109003
Combinations Forwarded    : 61871
IPs Followed              : 69337
Number of Stacks          : 152948
Stack Pushes              : 627809
Stack References          : 1228359
Stack Checks              : 130530
Stack Modifications       : 190907
Maximum Active Stacks     : 17
Maximum Stack Depth       : 12
Maximum Active Cells      : 63

Functors Introduced: 0

 Steps      %Steps     Functor
 -----      ------     -------
 30526       28.0      =
 14616       13.4      MB^
 10418        9.6      C
  9502        8.7      S^
  6420        5.9      B
  4881        4.5      S
  4101        3.8      C^
  3884        3.6      A-S
  3194        2.9      OR
  2952        2.7      AND
  2788        2.6      MAP
  2072        1.9      +
  2059        1.9      APPEND
  1907        1.7      MEMBER
```

END
DATE
FILMED

4-88
DTIC