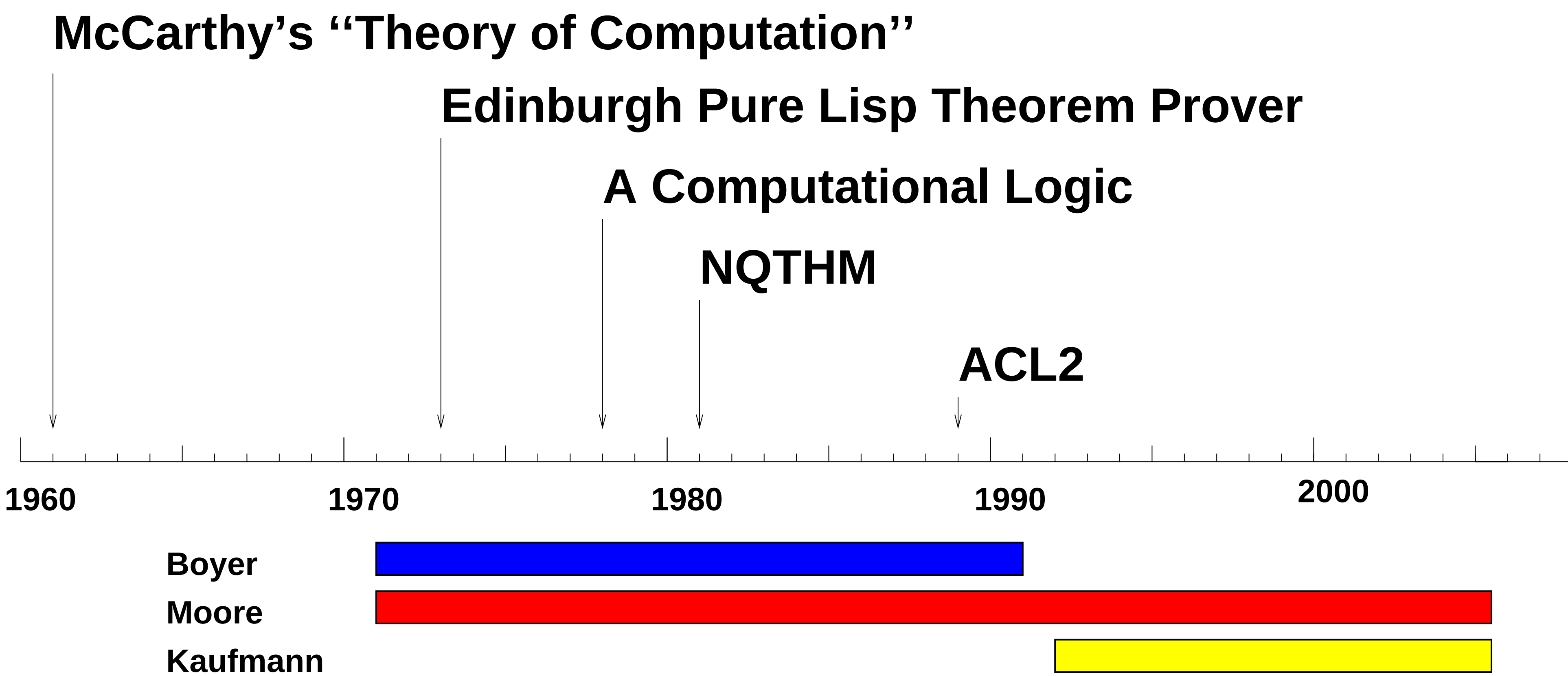
An ACL2 Tutorial

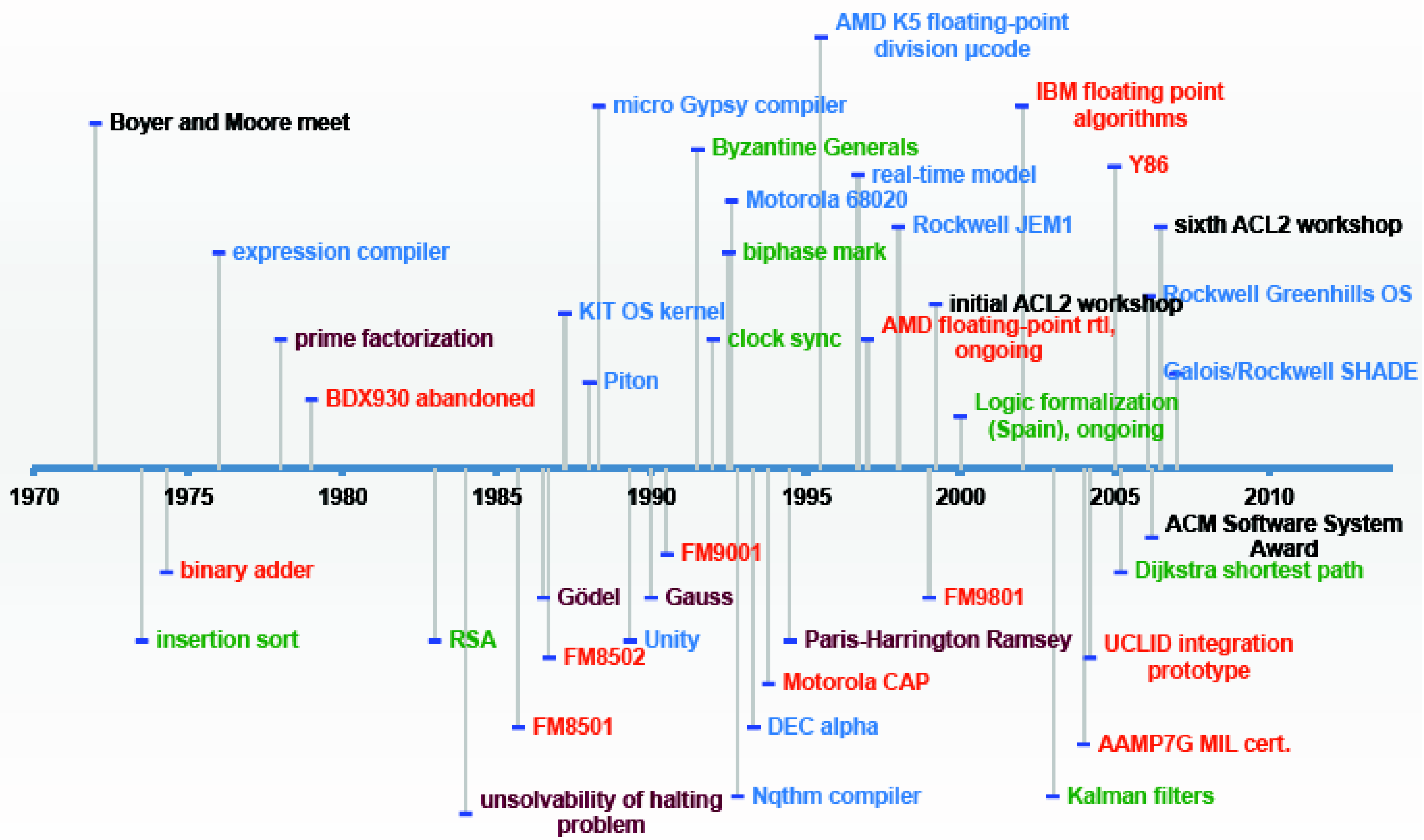
J Strother Moore
Department of Computer Sciences
University of Texas at Austin

Outline

- ACL2 Background
- Elementary Examples
- A Closer Look at a Big JVM Model
- Two Styles of Code Proofs

Boyer-Moore Project




AMD K5 floating-point division µcode
micro Gypsy compiler
IBM floating point algorithms
Boyer and Moore meet
Byzantine Generals
real-time model
Y86
Motorola 68020
Rockwell JEM1
sixth ACL2 workshop
expression compiler
biphase mark
Rockwell Greenhills OS
initial ACL2 workshop
KIT OS kernel
AMD floating-point rtl, ongoing
prime factorization
clock sync
Galois/Rockwell SHADE
Piton
BDX930 abandoned
Logic formalization (Spain), ongoing
1970
1975
1980
1985
1990
1995
2000
2005
2010
ACM Software System Award
FM9001
binary adder
Dijkstra shortest path
Gödel
Gauss
FM9801
insertion sort
RSA
Unity
Paris-Harrington Ramsey
UCLID integration prototype
FM8502
Motorola CAP
FM8501
DEC alpha
AAMP7G MIL cert.
unsolvability of halting problem
Nqthm compiler
Kalman filters

IEEE 754 Floating Point Standard

Elementary operations are to be performed as though the infinitely precise (standard mathematical) operation were performed and then the result rounded to the indicated precision.

AMD K5 Algorithm FDIV($p, d, mode$)

1.	sd_0	$= \mathrm{lookup}(d)$	`[exact  17  8]`
2.	d_r	$= d$	`[away   17 32]`
3.	sdd_0	$= sd_0 \times d_r$	`[away   17 32]`
4.	sd_1	$= sd_0 \times \mathrm{comp}(sdd_0, 32)$	`[trunc  17 32]`
5.	sdd_1	$= sd_1 \times d_r$	`[away   17 32]`
6.	sd_2	$= sd_1 \times \mathrm{comp}(sdd_1, 32)$	`[trunc  17 32]`
...	...	$=$ ...	...
29.	q_3	$= sd_2 \times ph_3$	`[trunc  17 24]`
30.	qq_2	$= q_2 + q_3$	`[sticky 17 64]`
31.	qq_1	$= qq_2 + q_1$	`[sticky 17 64]`
32.	$fdiv$	$= qq_1 + q_0$	$mode$

Using the Reciprocal

$$
\begin{array}{rr|l}
 & & 36. \\
 & + & -.17 \\
 & + & .0034 \\
 & + & -.000066 \\
\cline{3-3}
 & & 35.833334 \\
\cline{3-3}
12 & & 430.000000 \\
 & & 432. \\
\cline{3-3}
 & & -2. \\
 & & -2.04 \\
\cline{3-3}
 & & .04 \\
 & & .0408 \\
\cline{3-3}
 & & -.0008 \\
 & & -.000792 \\
\cline{3-3}
 & & -.000008
\end{array}
$$

Reciprocal Calculation:

$1/12 \;=\; 0.083\overline{3} \approx 0.083 = sd_2$

Quotient Digit Calculation:

$0.083 \times 430.0000 = 35.6900000 \approx 36.000000 = q_0$

$0.083 \times -2.0000 = -.1660000 \approx -.170000 = q_1$

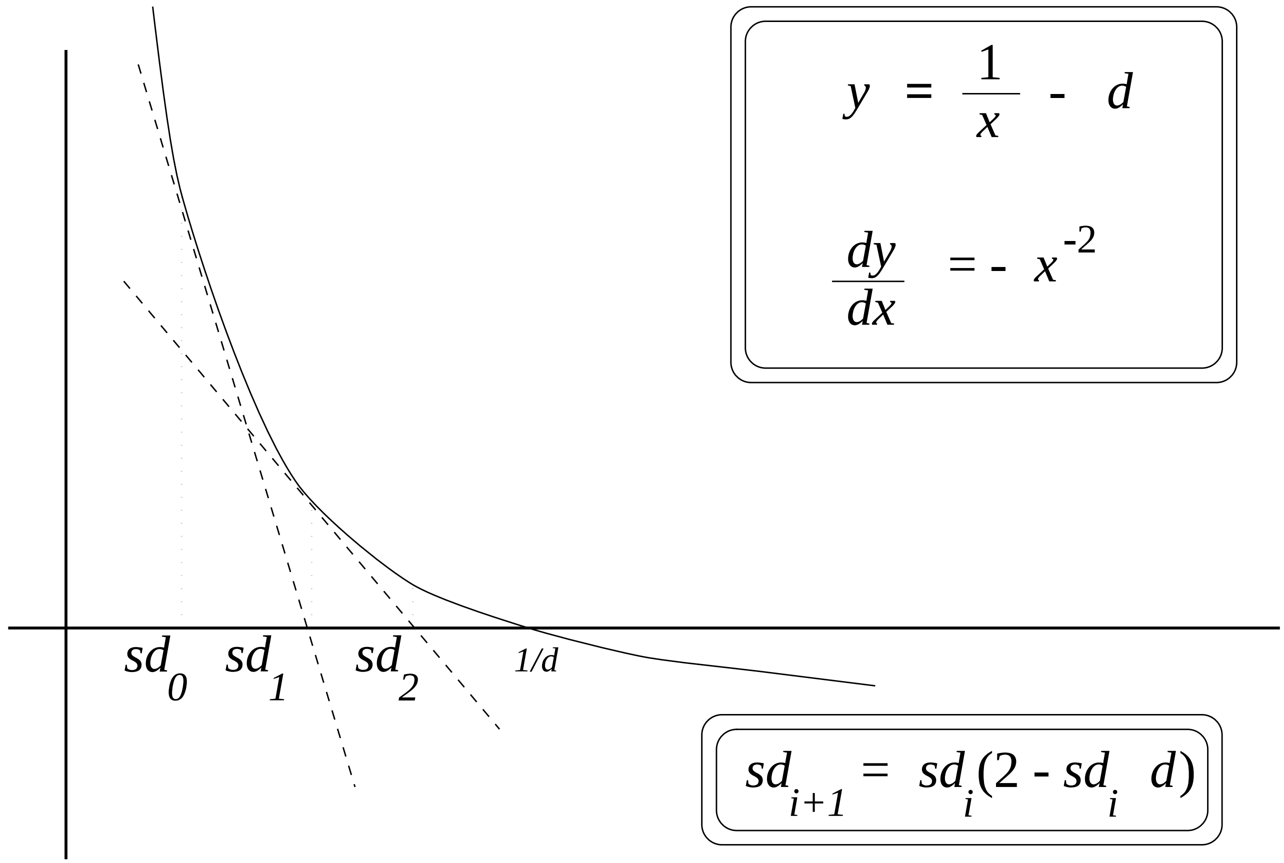
$0.083 \times .0400 = .0033200 \approx .003400 = q_2$

$0.083 \times -.0008 = -.0000664 \approx -.000067 = q_3$

Summation of Quotient Digits:

$q_0 + q_1 + q_2 + q_3 = 35.833333$

Computing the Reciprocal



$$y = \frac{1}{x} - d$$

$$\frac{dy}{dx} = -x^{-2}$$

$$sd_{i+1} = sd_i(2 - sd_i\, d)$$

top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse	top 8 bits of d	approx inverse
1.0000000_2	0.11111111_2	1.0100000_2	0.11001100_2	1.1000000_2	0.10101010_2	1.1100000_2	0.10010010_2
1.0000001_2	0.11111101_2	1.0100001_2	0.11001011_2	1.1000001_2	0.10101001_2	1.1100001_2	0.10010001_2
1.0000010_2	0.11111011_2	1.0100010_2	0.11001010_2	1.1000010_2	0.10101000_2	1.1100010_2	0.10010001_2
1.0000011_2	0.11111001_2	1.0100011_2	0.11001000_2	1.1000011_2	0.10101000_2	1.1100011_2	0.10010000_2
1.0000100_2	0.11110111_2	1.0100100_2	0.11000111_2	1.1000100_2	0.10100111_2	1.1100100_2	0.10001111_2
1.0000101_2	0.11110101_2	1.0100101_2	0.11000110_2	1.1000101_2	0.10100110_2	1.1100101_2	0.10001111_2
1.0000110_2	0.11110100_2	1.0100110_2	0.11000101_2	1.1000110_2	0.10100101_2	1.1100110_2	0.10001110_2
1.0000111_2	0.11110010_2	1.0100111_2	0.11000100_2	1.1000111_2	0.10100100_2	1.1100111_2	0.10001110_2
1.0001000_2	0.11110000_2	1.0101000_2	0.11000010_2	1.1001000_2	0.10100011_2	1.1101000_2	0.10001101_2
1.0001001_2	0.11101110_2	1.0101001_2	0.11000001_2	1.1001001_2	0.10100011_2	1.1101001_2	0.10001100_2
1.0001010_2	0.11101101_2	1.0101010_2	0.11000000_2	1.1001010_2	0.10100010_2	1.1101010_2	0.10001100_2
...	...	...	...	...	...	...	...
1.0010110_2	0.11011010_2	1.0110110_2	0.10110100_2	1.1010110_2	0.10011001_2	1.1110110_2	0.10000101_2
1.0010111_2	0.11011000_2	1.0110111_2	0.10110011_2	1.1010111_2	0.10011000_2	1.1110111_2	0.10000100_2
1.0011000_2	0.11010111_2	1.0111000_2	0.10110010_2	1.1011000_2	0.10010111_2	1.1111000_2	0.10000100_2
1.0011001_2	0.11010101_2	1.0111001_2	0.10110001_2	1.1011001_2	0.10010111_2	1.1111001_2	0.10000011_2
1.0011010_2	0.11010100_2	1.0111010_2	0.10110000_2	1.1011010_2	0.10010110_2	1.1111010_2	0.10000011_2
1.0011011_2	0.11010011_2	1.0111011_2	0.10101111_2	1.1011011_2	0.10010101_2	1.1111011_2	0.10000010_2
1.0011100_2	0.11010001_2	1.0111100_2	0.10101110_2	1.1011100_2	0.10010101_2	1.1111100_2	0.10000010_2
1.0011101_2	0.11010000_2	1.0111101_2	0.10101101_2	1.1011101_2	0.10010100_2	1.1111101_2	0.10000001_2
1.0011110_2	0.11001111_2	1.0111110_2	0.10101100_2	1.1011110_2	0.10010011_2	1.1111110_2	0.10000001_2
1.0011111_2	0.11001101_2	1.0111111_2	0.10101011_2	1.1011111_2	0.10010011_2	1.1111111_2	0.10000000_2

The Formal Model of the Code

```
(defun FDIV (p d mode)
 (let*
   ((sd0  (eround (lookup d)               '(exact  17  8)))
    (dr   (eround d                        '(away   17 32)))
    (sdd0 (eround (* sd0 dr)               '(away   17 32)))
    (sd1  (eround (* sd0 (comp sdd0 32)) '(trunc  17 32)))
    (sdd1 (eround (* sd1 dr)               '(away   17 32)))
    (sd2  (eround (* sd1 (comp sdd1 32)) '(trunc  17 32)))
    ...
    (qq2  (eround (+ q2 q3)                '(sticky 17 64)))
    (qq1  (eround (+ qq2 q1)               '(sticky 17 64)))
    (fdiv (round   (+ qq1 q0)              mode)))
   (or (first-error sd0 dr sdd0 sd1 sdd1 ... fdiv)
       fdiv)))
```

The K5 FDIV Theorem (1200 lemmas)

```
(defthm FDIV-divides
  (implies (and (floating-point-numberp p 15 64)
                (floating-point-numberp d 15 64)
                (not (equal d 0))
                (rounding-modep mode))
           (equal (FDIV p d mode)
                  (round  (/ p d) mode))))
```

(by Moore, Lynch and Kaufmann, in 1995, *before the K5 was fabricated*)

ACL2 =

A **C**omputational **L**ogic

for

Applicative **C**ommon **L**isp

“ACL2” is the name of

- a functional programming language,
- a mathematical logic, and
- an automatic interactive theorem prover.

Demo 0

ACL2 is *untyped*.

ACL2 is *strict* (not lazy).

ACL2 is *first order* (no functional args).

ACL2 is *applicative* (functional).

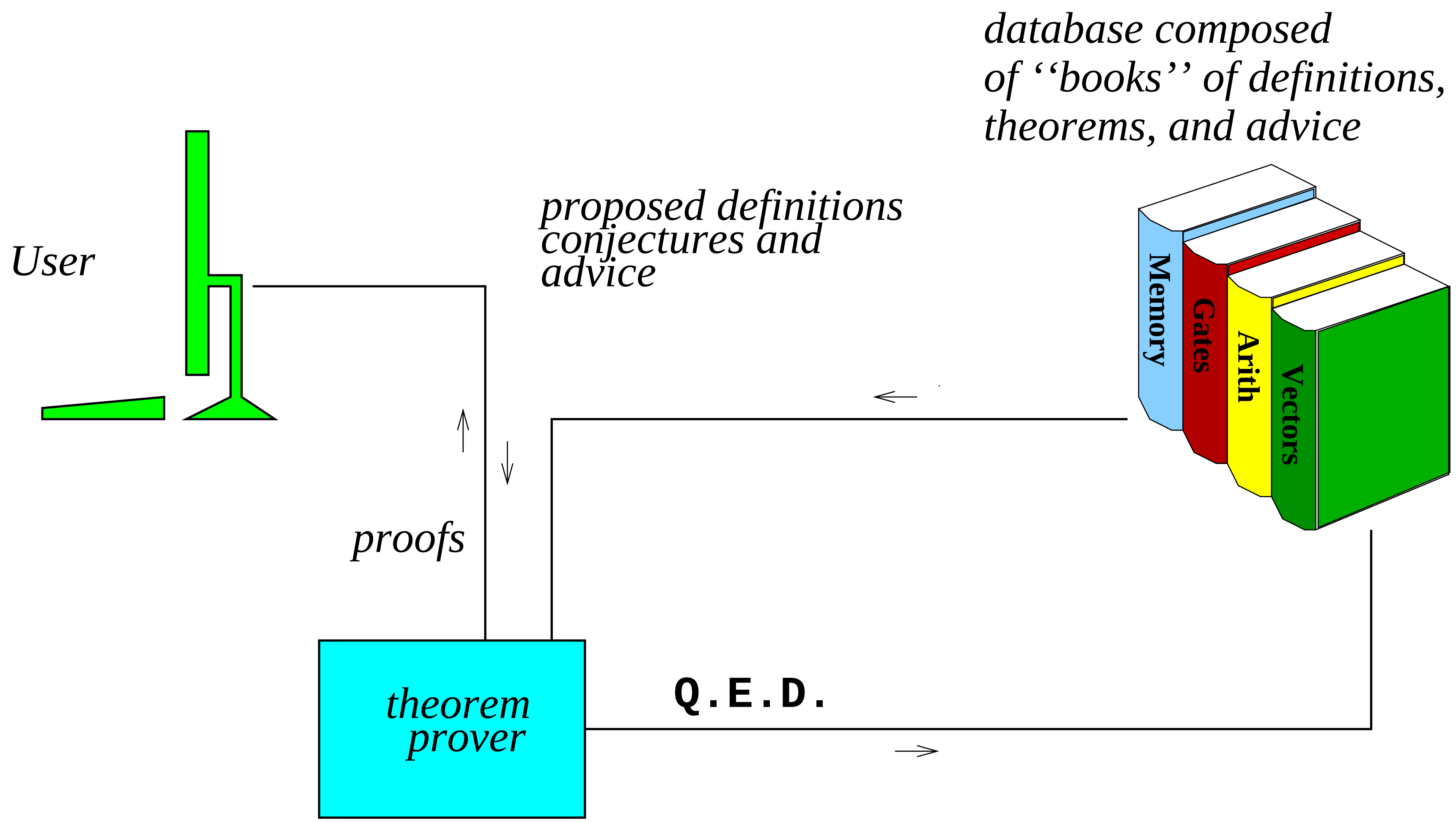
All ACL2 functions are *total* (always terminate on all arguments).

ACL2 is *executable* – almost all functions applied to constants can be reduced to equivalent constants.

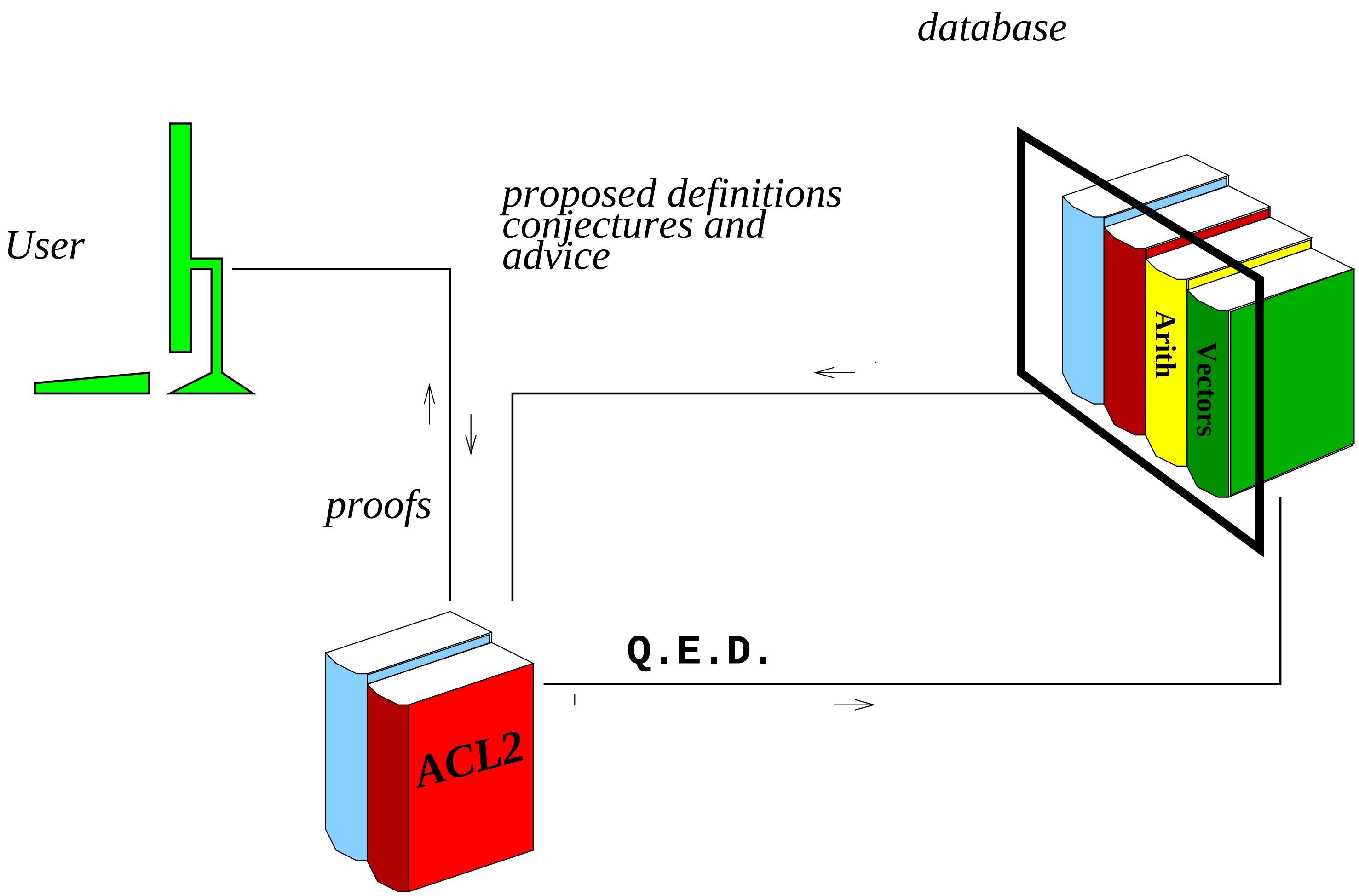
ACL2 is *quantifier-free* – but has the expressive power of full first-order logic thanks to Skolemization.

ACL2 is *automatic* – once the theorem prover starts, the user cannot guide it.

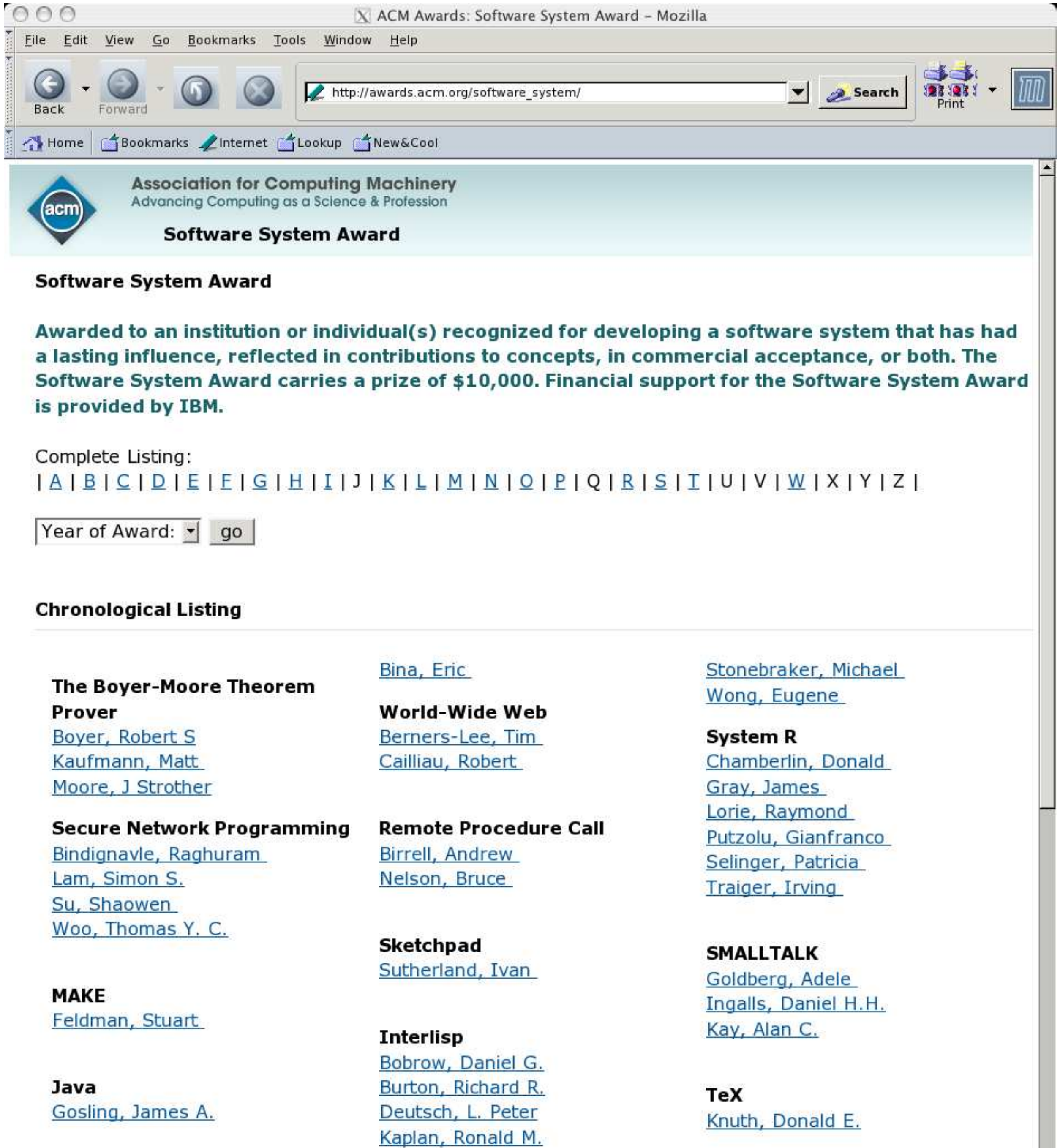
ACL2 is *interactive* – the theorem prover's behavior is influenced by the data base of previously proved lemmas and user-provided advice.


database composed
of ‘‘books’’ of definitions,
theorems, and advice
proposed definitions
conjectures and
advice
User
Memory
Gates
Arith
Vectors
proofs
theorem
prover
Q.E.D.

ACL2 is coded in ACL2.


database
User
proposed definitions
conjectures and
advice
Arith
Vectors
proofs
Q.E.D.
ACL2

ACL2 is the first theorem prover to win the ACM Software System Award.



ACM Awards: Software System Award - Mozilla

File Edit View Go Bookmarks Tools Window Help

Back Forward http://awards.acm.org/software_system/ Search Print

Home Bookmarks Internet Lookup New&Cool

Association for Computing Machinery
Advancing Computing as a Science & Profession

Software System Award

Software System Award

Awarded to an institution or individual(s) recognized for developing a software system that has had a lasting influence, reflected in contributions to concepts, in commercial acceptance, or both. The Software System Award carries a prize of $10,000. Financial support for the Software System Award is provided by IBM.

Complete Listing:
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z |

Year of Award: go

Chronological Listing

The Boyer-Moore Theorem Prover
Boyer, Robert S
Kaufmann, Matt
Moore, J Strother

Secure Network Programming
Bindignavle, Raghuram
Lam, Simon S.
Su, Shaowen
Woo, Thomas Y. C.

MAKE
Feldman, Stuart

Java
Gosling, James A.

Bina, Eric

World-Wide Web
Berners-Lee, Tim
Cailliau, Robert

Remote Procedure Call
Birrell, Andrew
Nelson, Bruce

Sketchpad
Sutherland, Ivan

Interlisp
Bobrow, Daniel G.
Burton, Richard R.
Deutsch, L. Peter
Kaplan, Ronald M.

Stonebraker, Michael
Wong, Eugene

System R
Chamberlin, Donald
Gray, James
Lorie, Raymond
Putzolu, Gianfranco
Selinger, Patricia
Traiger, Irving

SMALLTALK
Goldberg, Adele
Ingalls, Daniel H.H.
Kay, Alan C.

TeX
Knuth, Donald E.

ACL2 is (probably) the first winner that is written in a functional programming language.

ACM Awards: Software System Award - Mozilla

File Edit View Go Bookmarks Tools Window Help

Back Forward http://awards.acm.org/software_system/ Search Print

Home Bookmarks Internet Lookup New&Cool

The Boyer-Moore Theorem Prover
Boyer, Robert S
Kaufmann, Matt
Moore, J Strother

Secure Network Programming
Bindignavle, Raghuram
Lam, Simon S.
Su, Shaowen
Woo, Thomas Y. C.

MAKE
Feldman, Stuart

Java
Gosling, James A.

SPIN
Holzmann, Gerard

The Apache Group
Behlendorf, Brian
Fielding, Roy T.
Hartill, Rob
Robinson, David
Skolnick, Cliff
Terbush, Randy
Thau, Robert S.
Wilson, Andrew

The S System
Chambers, John M.

Tcl/Tk

Bina, Eric

World-Wide Web
Berners-Lee, Tim
Cailliau, Robert

Remote Procedure Call
Birrell, Andrew
Nelson, Bruce

Sketchpad
Sutherland, Ivan

Interlisp
Bobrow, Daniel G.
Burton, Richard R.
Deutsch, L. Peter
Kaplan, Ronald M.
Masinter, Larry
Teitelman, Warren

TCP/IP
Cerf, Vinton G.
Kahn, Robert E.

NLS
Engelbart, Douglas C.
English, William K.
Rulifson, Jeff

PostScript
Brotz, Douglas K.
Geschke, Charles M.
Paxton, William H.
Taft, Edward A.
Warnock, John E.

Stonebraker, Michael
Wong, Eugene

System R
Chamberlin, Donald
Gray, James
Lorie, Raymond
Putzolu, Gianfranco
Selinger, Patricia
Traiger, Irving

SMALLTALK
Goldberg, Adele
Ingalls, Daniel H.H.
Kay, Alan C.

TeX
Knuth, Donald E.

VisiCalc
Bricklin, Daniel
Frankston, Robert

Xerox Alto Systems
Lampson, Butler W.
Taylor, Robert W.
Thacker, Charles P.

UNIX
Ritchie, Dennis M.
Thompson, Ken

Lisp Syntax

$$
\begin{aligned}
< term > :=\ & < var > | \\
& \text{'}< const >)| \\
& (< fn > \quad < term >_1 \\
& \qquad\qquad\qquad \ldots \\
& \qquad\qquad\qquad < term >_n)
\end{aligned}
$$

$$
\begin{aligned}
< const > :=\ & < number > | < char > | \\
& < string > | < symbol > | \\
& < pair >
\end{aligned}
$$

Example Constants

`123`, `22/7`

`\#Newline`, `#\A`, `#\a`

`"Hello world!"`

`x`, `world`, `pt`, `PT`, `Pt`

`((Mon . 1) (Tue . 2) (Wed . 3))`

Example Terms

```
(cons (car x) rest))
```

e.g., $\mathrm{cons}(\mathrm{car}(x), rest)$

```
(if (zp n) 1 (* n (fact (- n 1))))
```

e.g., **if** $n = 0$ **then** 1 **else** $n * \mathrm{fact}(n-1)$ **fi**

About T and NIL

`T` and `NIL` are *symbols*.

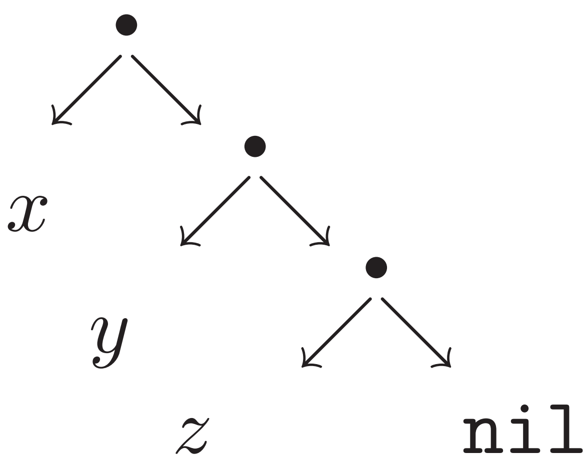
`T` and `t` are the same, as are `NIL` and `nil`.

`T` and `NIL` are used as the “truth values” true and false.

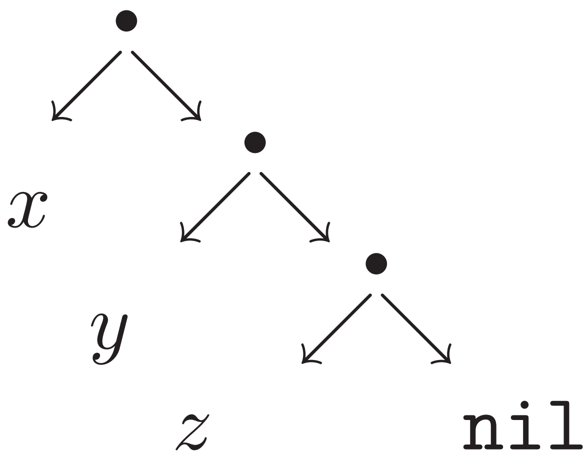
`NIL` is also used as the “terminal marker” on nested pairs representing lists.

About Pairs

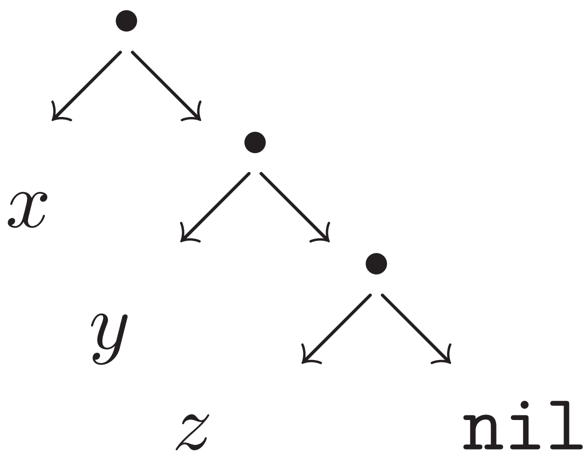
$< x,\ < y,\ < z, \texttt{nil}>>>$



$(x\ \texttt{.}\ (y\ \texttt{.}\ (z\ \texttt{.}\ \texttt{nil})))$

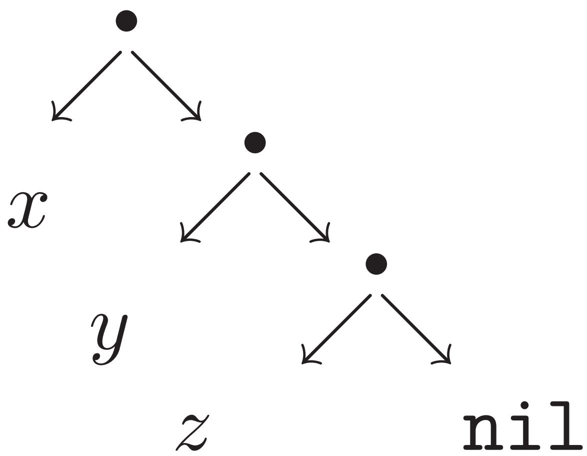


$$(x \;.\; (y \;.\; (z \;.\; \texttt{nil})))$$

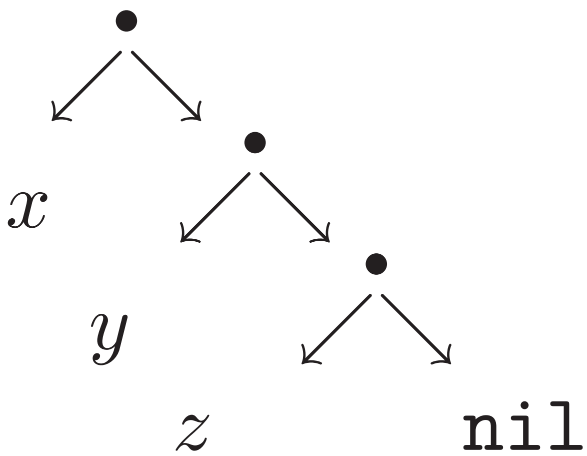


(x . (y . (z . nil)))

(x . (y . (z))) ; *may erase* ‘‘. nil’’




```
(x . (y . (z . nil)))
(x . (y . (z      )))   ; may erase ‘‘. nil’’
(x . (y    z       ))   ; may erase ‘‘. (...)’’
```




```
(x . (y . (z . nil)))
(x . (y . (z      )))   ; may erase ‘‘. nil’’
(x . (y   z        ))   ; may erase ‘‘. (...)’’
(x    y   z         )   ; may erase ‘‘. (...)’’
```

Is it strange that Lisp provides so many ways to write $(x\ \ y\ \ z)$?

$$(x\ \ .\ \ (y\ \ .\ \ (z\ \ .\ \ \texttt{nil})))$$

$$(x\ \ .\ \ (y\ \ .\ \ (z\ \ )))$$

$$(x\ \ .\ \ (y\ \ \ z\ \ ))$$

$$(x\ \ \ y\ \ \ z\ \ )$$

Is it strange that you know so many ways to write 123?

`123`

`0123`

`+123`

$\texttt{01111011}_2$

`0x7B`

Data Types

ACL2 supports five disjoint data types:

- numbers (integers, non-integer rationals, complex rationals)
- characters
- strings
- symbols
- pairs

There are primitive functions for

- creating each type of object from its constituents, e.g., `cons` creates pairs;
- accessing the constituents, e.g., `car` and `cdr`, aka `head` and `tail`;
- recognizing instances of each type, e.g., `consp`;

- other expected operations (e.g., addition of numbers).

Semantics

```
(cons 1 (cons 2 (cons 3 nil)))
```

$\Rightarrow$; *"evaluates to"*

```
(1 2 3)
```

```
(cons 1 '(2 3))
```

$\Rightarrow$

```
(1 2 3)
```

'(1 2 3) $\Rightarrow$ (1 2 3)

(car '(1 2 3)) $\Rightarrow$ 1

(cdr '(1 2 3)) $\Rightarrow$ (2 3)

`(consp '(1 2 3))` $\Rightarrow$ `t`

`(consp 1)` $\Rightarrow$ `nil`

`(consp nil)` $\Rightarrow$ `nil`

A Few Axioms

$\texttt{t} \neq \texttt{nil}$

$\texttt{x} = \texttt{nil} \longrightarrow \texttt{(if x y z)} = \texttt{z}$

$\texttt{x} \neq \texttt{nil} \longrightarrow \texttt{(if x y z)} = \texttt{y}$

$\texttt{(car (cons x y))} = \texttt{x}$

$\texttt{(cdr (cons x y))} = \texttt{y}$

```
(consp (cons x y)) = t

(consp nil) = nil

(endp x) = (not (consp x))
```

Definitions

```
(defun not (x) (if x nil t))
```

is a way to add a

New Axiom

(not x) = (if x nil t)

Propositional Calculus

```
(defun not (x) (if x nil t))

(defun and (x y) (if x y nil))

(defun or  (x y) (if x x y))

(defun implies (x y)
  (if x (if y t nil) t))
```

Inconsistent "Definition"

```
(defun f (x) (not (f x)))
```

Theorem: `t` = `nil`.

Proof.

```
(f x) = (not (f x))
      = (if (f x) nil t).
```

So `(f x)` is either `nil` or `t`.

Case 1: $\texttt{nil} = \texttt{(f x)}$

$= \texttt{(not (f x))} = \texttt{(not nil)} = \texttt{t}.$

Case 2: $\texttt{t} = \texttt{(f x)}$

$= \texttt{(not (f x))} = \texttt{(not t)} = \texttt{nil}.$

Q.E.D.

The Definitional Principle

(`defun` f $(x_1 \ldots x_n)$ $body$)

is *admissible* if and only if:

- f is not already axiomatized;
- the x_i are distinct;
- the only variables in $body$ are the x_i;

- there is a measure of the x_i and a *well-founded* ordering such that for every recursive call of f in $body$ it can be proved that the measure decreases according to the ordering.

The last condition means that ACL2 can admit only provably *terminating* recursive definitions.

Recursive Definition

```
(defun app (x y)
  (if (endp x)
      y
      (cons (car x)
            (app (cdr x) y))))
```

```
(append '(1 2 3) (append '(4 5 6) '(7 8 9)))
```
= '(1 2 3 4 5 6 7 8 9)

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Base Case:  (endp a).
(equal (append (append a b) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: (endp a).

```
(equal (append b c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))

Proof: by induction on a.

Base Case:  (endp a).
(equal (append b c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Base Case: `(endp a)`.

```
(equal (append b c)
       (append b c))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Base Case:  (endp a).
(equal (append b c)
       (append b c))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Base Case:  (endp a).
T
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Induction Step: (not (endp a)).
(equal (append (append a b) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Induction Step: (not (endp a)).
(equal (append (cons (car a)
                     (append (cdr a) b)) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))

Proof: by induction on a.

Induction Step: (not (endp a)).
(equal (append (cons (car a)
                     (append (cdr a) b)) c)
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))

Proof: by induction on a.

Induction Step: (not (endp a)).
(equal (cons (car a)
         (append (append (cdr a) b) c))
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Induction Step: (not (endp a)).
(equal (cons (car a)
             (append (append (cdr a) b) c))
       (append a (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))

Proof: by induction on a.

Induction Step: (not (endp a)).
(equal (cons (car a)
         (append (append (cdr a) b) c))
       (cons (car a)
         (append (cdr a) (append b c))))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Induction Step: (not (endp a)).
(equal (cons (car a)
         (append (append (cdr a) b) c))
       (cons (car a)
         (append (cdr a) (append b c))))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

```
Proof: by induction on a.
```

```
Induction Step: (not (endp a)).
(equal
         (append (append (cdr a) b) c)

         (append (cdr a) (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (append (append (cdr a) b) c)
       (append (cdr a) (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

```
(equal (append (append (cdr a) b) c)
       (append (cdr a) (append b c)))
```

```
(equal (append (append a b) c)
       (append a (append b c)))
```

Proof: by induction on a.

Induction Step: (not (endp a)).

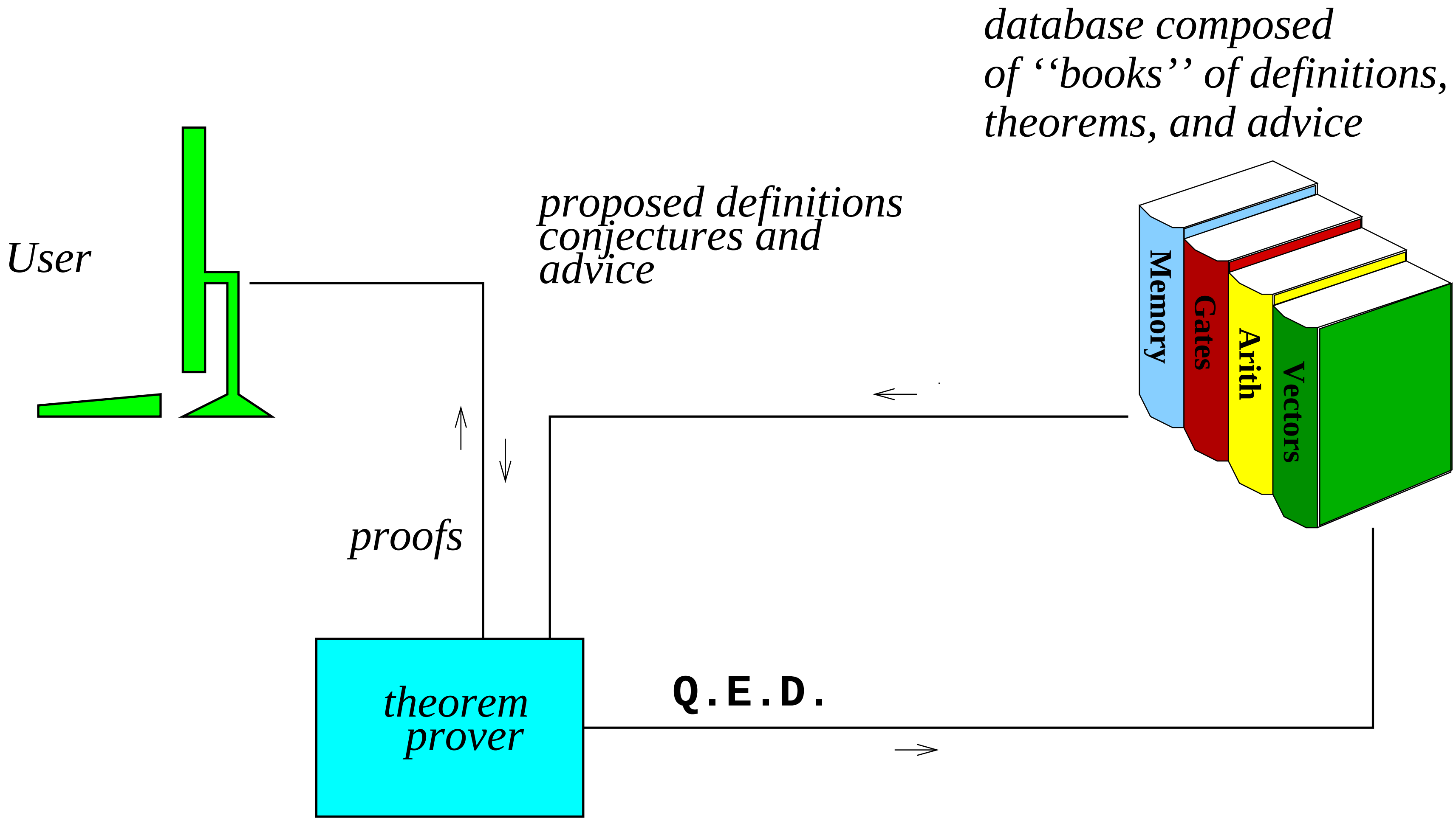
T

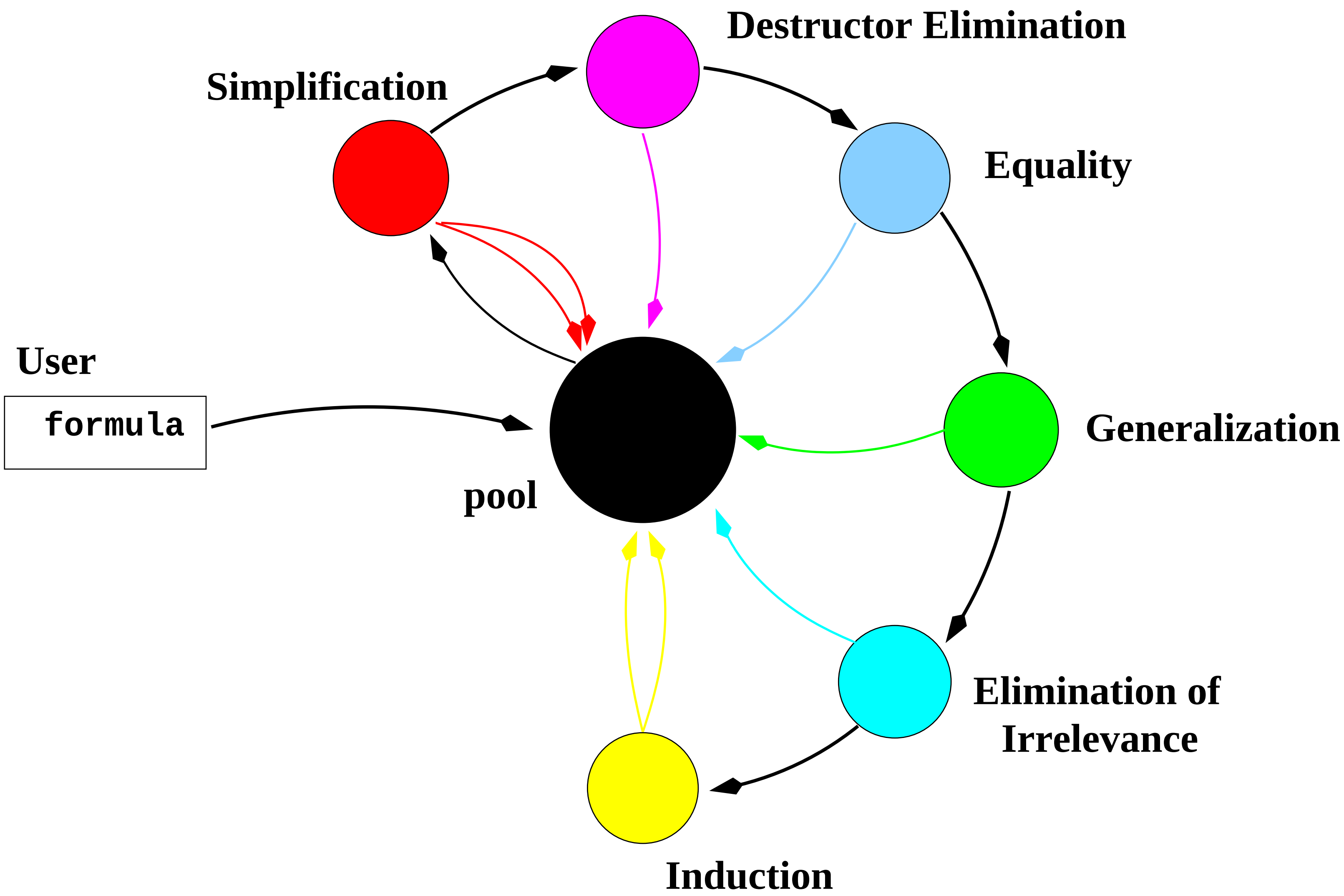
```
(equal (append (append a b) c)
       (append a (append b c)))
```

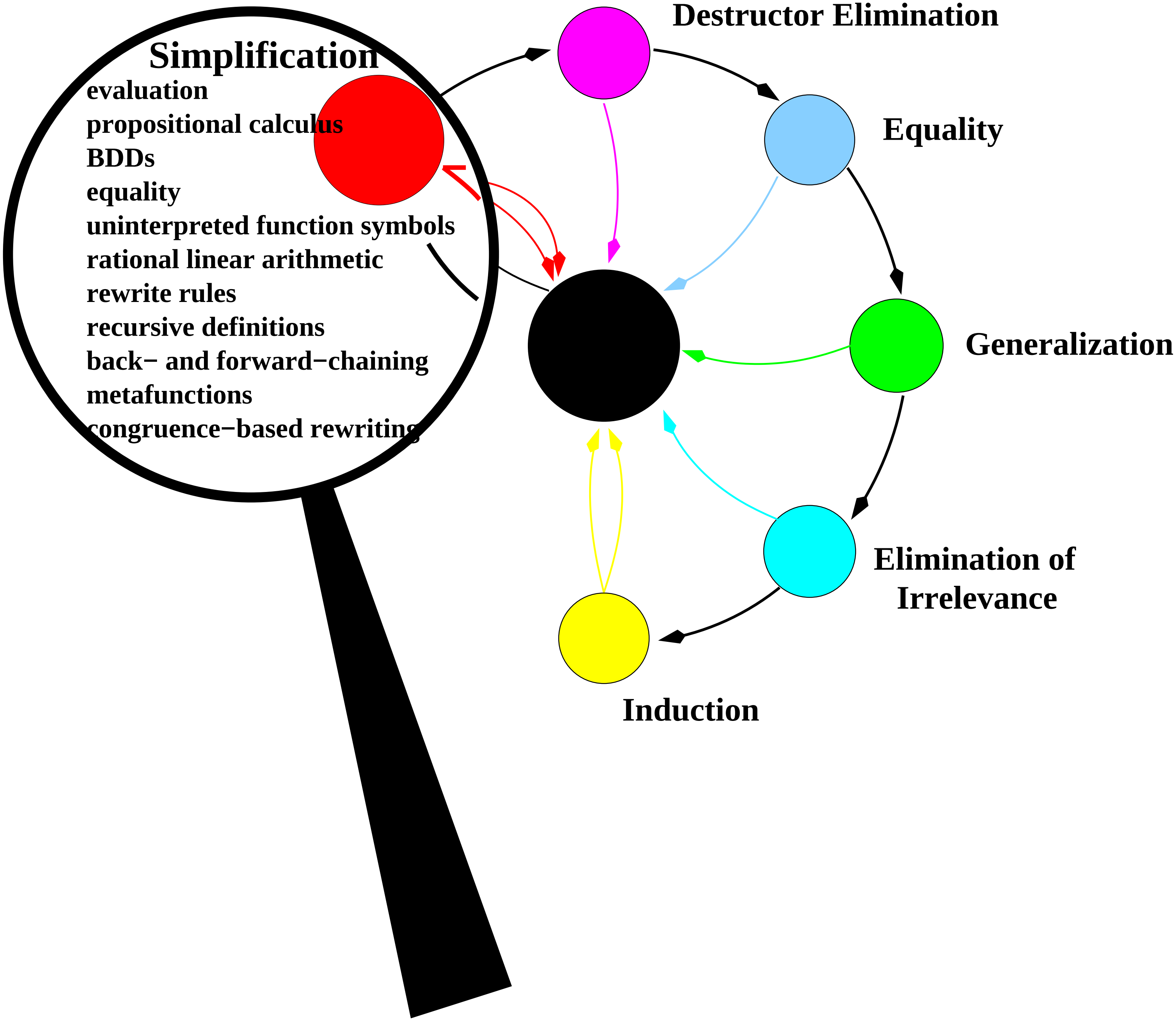
Proof: by induction on a.

Q.E.D.

Demo 1


database composed
of ‘‘books’’ of definitions,
theorems, and advice
proposed definitions
conjectures and
advice
User
Memory
Gates
Arith
Vectors
proofs
theorem
prover
Q.E.D.


Destructor Elimination
Simplification
Equality
User
formula
Generalization
pool
Elimination of
Irrelevance
Induction


Simplification
evaluation
propositional calculus
BDDs
equality
uninterpreted function symbols
rational linear arithmetic
rewrite rules
recursive definitions
back− and forward−chaining
metafunctions
congruence−based rewriting
Destructor Elimination
Equality
Generalization
Elimination of Irrelevance
Induction

Demo 2

Some More Realistic Examples

- a simple stack machine model
- a simple expression language model
- a simple compiler
- proof that the compiler is correct

The Stack Machine

We will formalize a machine:

$$\mathtt{m} : \mathit{programs} \times \mathit{environments} \times \mathit{stacks}$$
$$\longrightarrow$$
$$\mathit{stacks}$$

Sample program:

```
((LOAD A)
 (PUSH 3)
 (MUL))
```

Instruction set:

(`LOAD` var

(`PUSH` $const$)

(`ADD`)

(`MUL`)

(`DUP`)

No instruction will change the environment (no `STORE` instruction).

Sample environment:

```
((A    . 20)
 (B    . 30)
 (X    . -5)
 (TEMP . 18))
```

Sample stack:

```
(push 1 (push 2 (push 3 nil)))
```

$\Rightarrow$

```
(1 2 3)
```

Expressions

We will compile simple expressions into this language and prove that we did it correctly.

The expression language is

```
<expr> := <variable> |
          <constant> |
          <unary-application> |
          <binary-application>
```

```
<unary-application>
  := (- <expr>) |
     (sq <expr>)

<binary-application>
  := (<expr> + <expr>) |
     (<expr> - <expr>) |
     (<expr> * <expr>)
```

Obvious Criticisms

Everything is trivial!

No `STORE` instruction.

No program counter.

No branch or conditionals.

No iteration or loops.

No object creation.

No method invocation.

No exceptions or errors.

Response to the Criticism

I will show you a completely realistic model when we're done with this one.

But for now I want you to really *understand* what is involved in modeling two computational paradigms and proving their correspondence formally.

I'll follow "The Method" to do my proofs and show you everything (except some trivial "abstract data type" work at the bottom).

Demo 3

JVM Operational Semantics

Our “M6” model is based on an implementation of the J2ME KVM. It executes most J2ME Java programs (except those with significant I/O or floating-point).

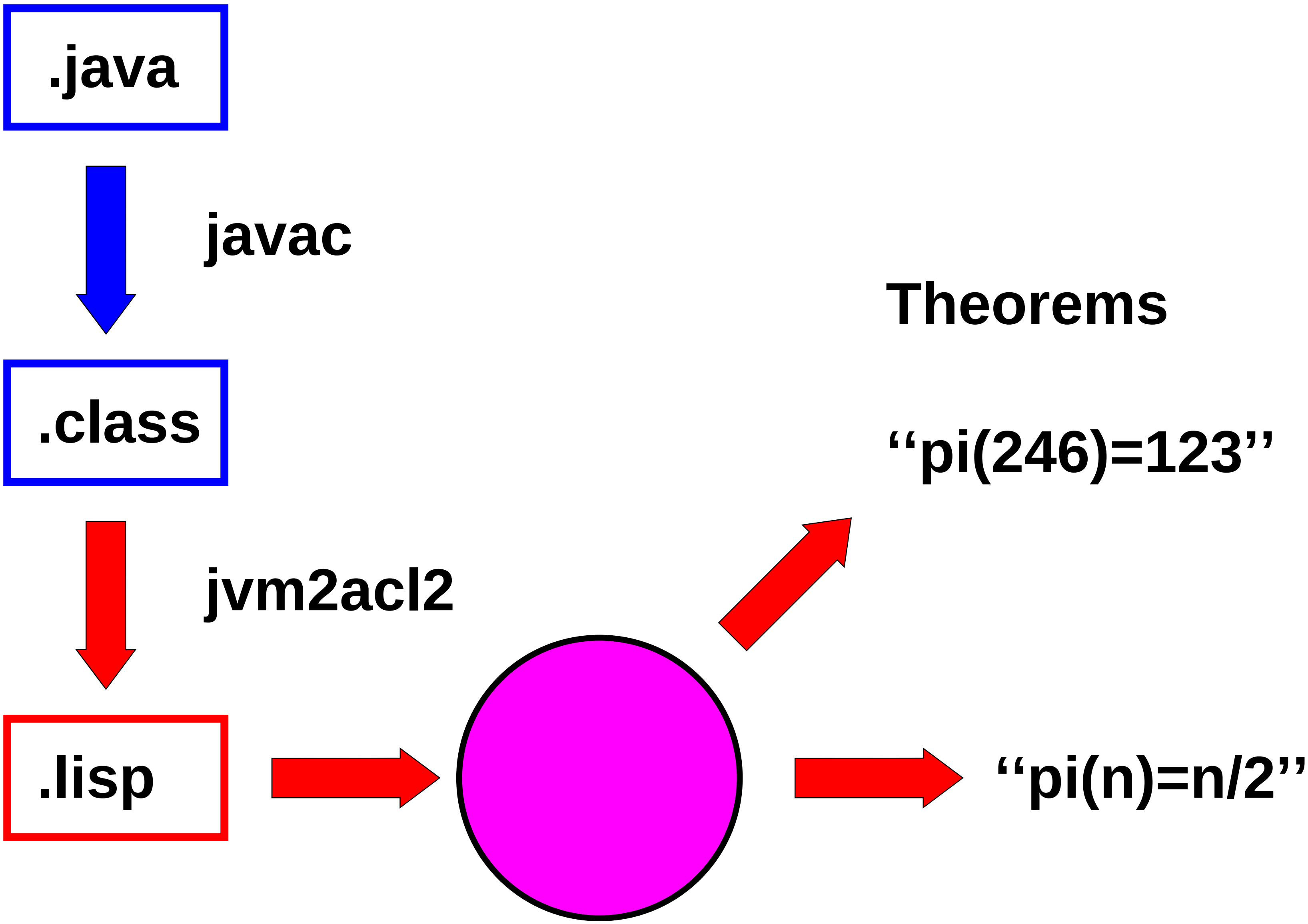
M6 supports all CLDC data types, multi-threading, dynamic class loading, class initialization and synchronization via monitors.

We have translated the entire Sun CLDC API library implementation into our representation with 672 methods in 87 classes. We provide implementations for 21 out of 41 native APIs that appear in Sun's CLDC API library.

We prove theorems about bytecoded methods with the ACL2 theorem prover.

The executable model is 160 pages of ACL2. This doesn't count over 500 pages of data (the CLDC API) built into the model.

This work is supported by a gift from Sun Microsystems.


.java
javac
.class
jvm2acl2
.lisp
Theorems
“‘pi(246)=123’’
“‘pi(n)=n/2’’

Demo 4

Key Research Problems

1. Automatic Invention of Lemmas and New Concepts

2. How to use Examples and Counterexamples

3. How to use Analogy, Learning, and Data Mining

4. How to Architect an Open Verification Environment

5. Parallel, Distributed and Collaborative Theorem Proving

6. User Interface and Interactive Steering

7. Education of the User Community – and Their Managers

8. How to Build a Verified Theorem Prover

Our Hypothesis

The "high cost" of formal methods

– to the extent the cost is high –

is a *historical anomaly* due to the fact that virtually every project formally recapitulates the past.

The use of mechanized formal methods will ultimately

- *decrease* time-to-market, and
- *increase* reliability.

Conclusion

Mechanical reasoning systems are changing the way complex digital artifacts are built.

Complexity not an argument *against* formal methods.

It is an argument *for* formal methods.

References

Computer-Aided Reasoning: An Approach, Kaufmann, Manolios, Moore, Kluwer Academic Publishers, 2000.

Computer-Aided Reasoning: ACL2 Case Studies, Kaufmann, Manolios, Moore (eds.), Kluwer Academic Publishers, 2000.

http://www.cs.utexas.edu/users/moore/acl2